ICSS SWITCH FIRMWARE DESIGN GUIDE

ICSS Cut-Through Switch

Applies to Product Release: 01.00.00.07
Publication Date: Apr 16, 2018

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 Switch firmware implementation details along with any features added on top of the basic Switch firmware. It mentions the memory maps, structures and software design flow of the firmware.

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

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

1 Introduction

ICSS based Cut-Through Switch is a three port learning Ethernet switch. The development goal of Switch was to apply Ethernet to automation applications which require short cut-through latency and low hardware costs.

Typical automation networks are characterized by chains of slaves connected in a serial manner, thus making it necessary to forward frames at each node as soon as possible. If frame is stored and forwarded on every node then it leads to delays in the propagation of frames and thus poor overall network performance.

With Cut-Through Switch, the Ethernet packet or frame is no longer received and then forwarded at every node. Cut-Through latency depends on when the forwarding decision is made for a frame which is being received. If cut-through decision is taken after six bytes of destination address then the cut-through latency is 1.28us including the eight bytes of preamble. In keeping the requirements of various industrial protocols, the decision point at which cut-through decision is taken can be configured.

In industrial protocols there is Real Time Phase (RT) when process data is exchanged and Non-Real Time Phase (NRT) when TCP/IP frames are transmitted. Switch handles the NRT phase of the protocol cycle.

2 Requirements

2.1 Cut-Through Switch High Level Requirements

Requirements	Remarks
Cut-Through	Supported
Store and Forward	Supported
1 ms buffering per port	Supported
Host IRQ	Supported
Ethernet QoS	Supported but with 4 queues instead of 8. So, it is not a standard Ethernet QoS implementation.
802.1 learning switch	Supported
Statistics	Supported
Queue-Contention Handling on each port	Supported
Three-Port Switch	Supported
Storm Prevention	Supported

Table 1. High Level Requirements

3 Design Description

This section discusses the overall flow & interaction.

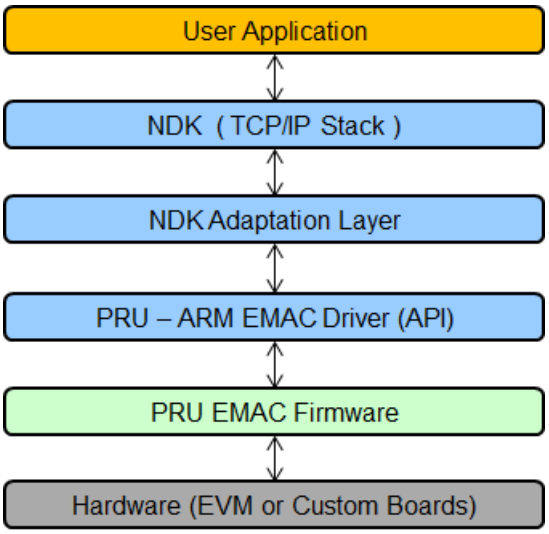


Figure 1. Switch Overall Design

3.1 System Decomposition Diagram

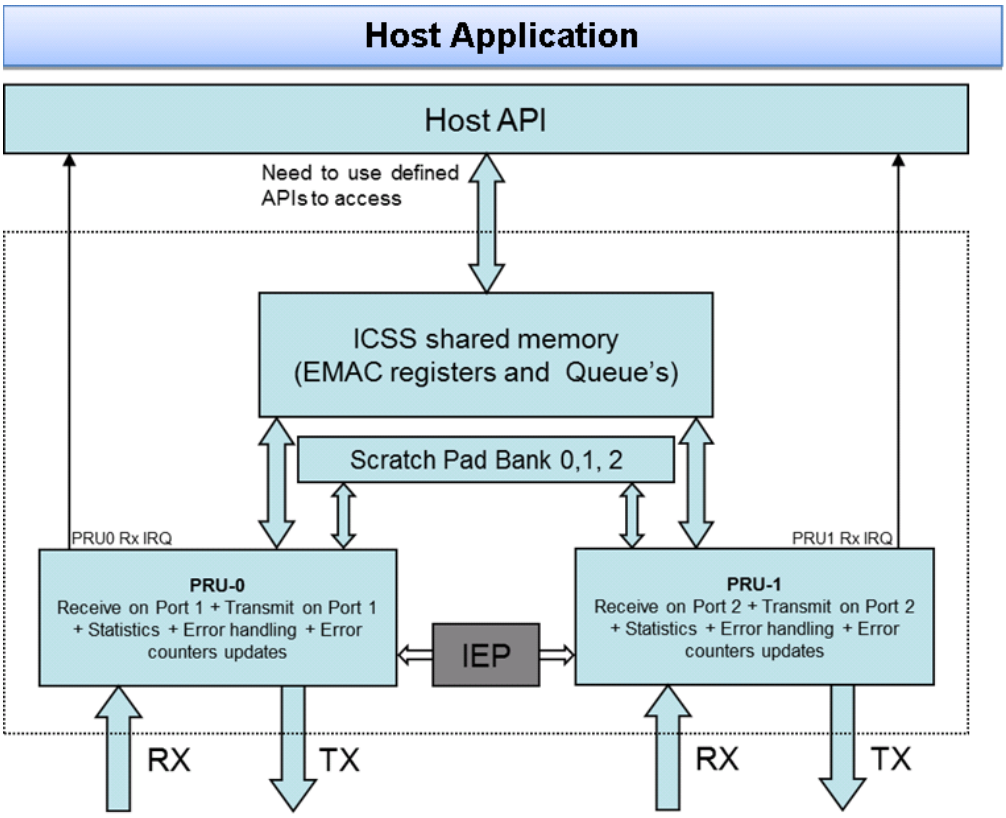


Figure 2. System Decomposition Diagram

4 Firmware Detailed Design

4.1 Firmware Architecture Overview

4.1.1 Architecture and Design

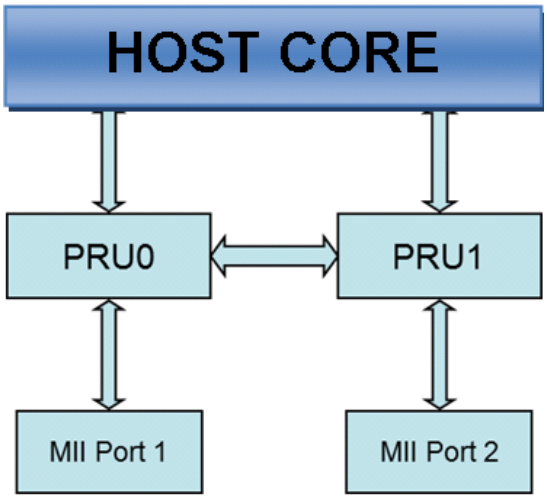


Figure 3. Switch Firmware high level architecture

1. Host Core functions (ARM, DSP)
    i. Run NDK Host stack
    ii. Switch initializations

2. PRU0 functions
    i. Receive frames on Port 1
    ii. Transmit frames on Port 2
    iii. Statistics for received frames on Port1 and transmit frames on Port 2.
    iv. Queue contention handling for Host port and Port 2.
    v. Cut-Through frames received on Port 1 to Port 2.
    vi. Store & Forward frames received on Port 1 to Port2.
    vii. Receive frame in Shadow buffer in case of queue contention on host and Port 2.

3. PRU1 functions are exactly symmetrical to PRU0.

4.1.2 PRU0/1 DMEM and ICSS Shared RAM memory map for Switch firmware

4.1.2.1 PRU0 DMEM memory map ICSS Shared RAM memory map. Switch doesn't use PRU0 DMEM memory in general; most of it is free and can be utilized by other tasks. Whatever is used is documented below. Switch Statistics memory layout documented separately.

Definition	Address map	Remarks
Free space	0x0000 to 0x1EFF	This memory space is free.
Switch Statistics	0x1F00 to 0x1F8C	Switch Statistics for PRU1 are stored in this memory space.
STORM_PREVENTION_OFFSET	0x1F8C	Used to store storm prevention credits and status for Port 0
PHY_SPEED_OFFSET	0x1F90	Phy speed status. Indicates to the firmware whether PHY is in 10 or 100 mbps.
Reserved	0x1F94-0x1FFF	Reserved for future use.

Table 2. PRU0 DMEM Memory Map

4.1.2.2 PRU1 DMEM memory map

Definition	Address map	Remarks
Free space.	0x0000 – 0x1CFF	This memory space is free.
Rx/Tx Queues Context Data	0x1D00 to 0x1DFF	Queue context data for all the Rx and Tx queues.
QUEUE_DESCRIPTOR_OFFSET_ADDR	0x1E00	Table offset for queue descriptors
QUEUE_OFFSET_ADDR	0x1E18	Table offset for queue
QUEUE_SIZE_ADDR	0x1E30	Table offset for queue size
INTERFACE_IP_ADDR	0x1E5C	Offset of Interface IP address. Reserved for future usage.
P1_MAC_ADDR	0x1E60	Offset of Port1 MAC address. Reserved
P2_MAC_ADDR	0x1E66	Offset of Port2 MAC address. Reserved
SWITCH_CONTROL_ADDR	0x1E6C	Contains Port Enable/Disable information for Port 0 and Port 1.
INTERFACE_MAC_ADDR	0x1E70	Contains the MAC address of switch.
SWITCH_STATUS_ADDR	0x1E78	Reserved for switch status variables.
BRIDGE_OPTION_ADDR	0x1E7C	Reserved for status variables
BROADCAST_SEND_STATUS_ADDR	0x1E80	Reserved for future use
COLLISION_STATUS_ADDR	0x1E84	Offset of queue contention status register
P0_COL_QUEUE_DESC_OFFSET	0x1E88	Offset of queue contention descriptor of port 0
P1_COL_QUEUE_DESC_OFFSET	0x1E90	Offset of queue contention descriptor of port 1

P2_COL_QUEUE_DESC_OFFSET	0x1E98	Offset of queue contention descriptor of port 2
P0_QUEUE_DESC_OFFSET	0x1EA0	4 host receive queue descriptors for port 0
P1_QUEUE_DESC_OFFSET	0x1EC0	4 queue descriptors for port 1
P2_QUEUE_DESC_OFFSET	0x1EE0	4 queue descriptors for port 2
Switch Statistics	0x1F00 to 0x1F8C	Switch Statistics for PRU1 are stored in this memory space.
STORM_PREVENTION_OFFSET	0x1F8C	Used to store storm prevention credits and status for Port 1
Reserved	0x1F90 to 0x1FFF	Reserved for future usage

Table 3. PRU1 DMEM Memory Map

4.1.2.3 ICSS Shared RAM memory map

Below are offset addresses of the buffer descriptors for the default configuration of queue sizes in "icss_switch.h" file. 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_BD_OFFSET	0x0000	Buffer descriptors for the priority 0 host receive queue.
P0_Q2_BD_OFFSET	0x0308	Buffer descriptors for the priority 1 host receive queue.
P0_Q3_BD_OFFSET	0x0610	Buffer descriptors for the priority 2 host receive queue.
P0_Q4_BD_OFFSET	0x0918	Buffer descriptors for the priority 3 host receive queue.
P1_Q1_BD_OFFSET	0x0C20	Buffer descriptors for the priority 0 transmit queue on Port 1.
P1_Q2_BD_OFFSET	0x0DA4	Buffer descriptors for the priority 1 transmit queue on Port 1.
P1_Q3_BD_OFFSET	0x0F28	Buffer descriptors for the priority 2 transmit queue on Port 1.
P1_Q4_BD_OFFSET	0x10AC	Buffer descriptors for the priority 3 transmit queue on Port 1.
P2_Q1_BD_OFFSET	0x1230	Buffer descriptors for the priority 0 transmit queue on Port 2.
P2_Q2_BD_OFFSET	0x13B4	Buffer descriptors for the priority 0 transmit queue on Port 2.
P2_Q3_BD_OFFSET	0x1538	Buffer descriptors for the priority 0 transmit queue on Port 2.
P2_Q4_BD_OFFSET	0x16BC	Buffer descriptors for the priority 0 transmit queue on Port 2.
P0_COL_BD_OFFSET	0x1800	48 Collision buffer descriptors for port 0 send queue
P1_COL_BD_OFFSET	0x18C0	48 Collision buffer descriptors for port 1 send queue
P2_COL_BD_OFFSET	0x1980	48 Collision buffer descriptors for port 2 send queue
STATIC_MAC_TABLE_RCV	0x1A40	256 Bytes Static MAC Table for multicast receive filtering

STATIC_MAC_TABLE_FWD	0x1B40	256 Bytes Static MAC Table for multicast forward filtering
MULTICAST_FIRST_VALID_ADDR_OFFSET	0x1C40	6 Bytes First Address in Multicast Group
MULTICAST_LAST_VALID_ADDR_OFFSET	0x1C48	6 Bytes Last Address in Multicast Group
PORT_STATUS_OFFSET	0x1C4B	Contains information about PHY being in half or full duplex.
COLLISION_COUNTER_PORT0	0x1C50	Temporary variables used in Firmware for Half duplex implementation for Port0
COLLISION_COUNTER_PORT1	0x1C51	Temporary variables used in Firmware for Half duplex implementation for Port1
STAT_CTRL_OFFSET	0x1C52	Statistics Enable/Disable information.
Reserve for protocols.	0x1C53 to 0x2FFF	This memory space is reserved for protocol specific usage. Do not store generic switch control variables here.

Table 4. ICSS Shared RAM Memory Map

4.1.2.4 L3 OCMC RAM memory map

Below are offset addresses of the queue data buffers for the default configuration of queue sizes in "icss_switch.h" file. 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 highest 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 least 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 first priority Port2 queue in L3 RAM
P2_Q3_BUFFER_OFFSET	0xA9C0	Offset of data buffers of first priority Port2 queue in L3 RAM
P2_Q4_BUFFER_OFFSET	0xB5E0	Offset of data buffers of first 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.
P0_COL_BUFFER_OFFSET	0xEE00	Offset of data buffers of host shadow queue in L3

		RAM
P1_COL_BUFFER_OFFSET	0xF400	Offset of data buffers of Port1 shadow queue in L3 RAM
P2_COL_BUFFER_OFFSET	0xFA00	Offset of data buffers of Port2 shadow queue in L3 RAM

Table 5. L3 OCMC RAM Memory Map

4.1.2.5 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
LATE_COLLISION_OFFSET	0x1F20	Number of packets which suffered late collision
SINGLE_COLLISION_OFFSET	0x1F24	Number of bytes which suffered only one collision
MULTIPLE_COLLISION_OFFSET	0x1F28	Number of bytes which suffered more than one collision
EXCESS_COLLISION_OFFSET	0x1F2C	Number of bytes which suffered more than 15 collisions
TX_OVERFLOW_OFFSET	0x1F30	Reserved
RX_MISALIGNMENT_COUNT_OFFSET	0x1F34	Packets which had an odd number of nibbles
STORM_PREVENTION_COUNTER	0x1F38	Multicast and Broadcast Packets which were discarded by Storm prevention
RX_ERROR_OFFSET	0x1F3C	Number of packets which triggered Rx MAC errors or number of instances where a MAC error was detected.
SFD_ERROR_OFFSET	0x1F40	Number of Packets with incorrect preamble.
TX_DEFERRED_OFFSET	0x1F44	Packets which were deferred from transmission at least once.
TX_ERROR_OFFSET	0x1F48	Reserved
RX_OVERSIZED_FRAME_OFFSET	0x1F4C	Number of packets with byte size greater than 1522. (Default value. This is a programmable value in MII RT)
RX_UNDERSIZED_FRAME_OFFSET	0x1F50	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	0x1F54	Number of Packets with CRC/FCS Error.
RX_DROPPED_FRAMES_OFFSET	0x1F58	Number of frames dropped due to link loss/same dst as host.
TX_64_BYTE_FRAME_OFFSET	0x1F5C	Number of transmitted packets of size 64 bytes
TX_65_127_BYTE_FRAME_OFFSET	0x1F60	Number of transmitted packets of size between 65-127 bytes.
TX_128_255_BYTE_FRAME_OFFSET	0x1F64	Number of transmitted packets of size between 128-255 bytes.
TX_256_511_BYTE_FRAME_OFFSET	0x1F68	Number of transmitted packets of size

		between 256-511 bytes.
TX_512_1023_BYTE_FRAME_OFFSET	0x1F6C	Number of transmitted packets of size between 512-1023 bytes.
TX_1024_MAX_BYTE_FRAME_OFFSET	0x1F70	Number of transmitted packets of size greater than 1023 bytes.
RX_64_BYTE_FRAME_OFFSET	0x1F74	Number of received packets of size 64 bytes
RX_65_127_BYTE_FRAME_OFFSET	0x1F78	Number of received packets of size between 65-127 bytes.
RX_128_255_BYTE_FRAME_OFFSET	0x1F7C	Number of received packets of size between 128-255 bytes.
RX_256_511_BYTE_FRAME_OFFSET	0x1F80	Number of received packets of size between 256-511 bytes.
RX_512_1023_BYTE_FRAME_OFFSET	0x1F84	Number of received packets of size between 512-1023 bytes.
RX_1024_MAX_BYTE_FRAME_OFFSET	0x1F88	Number of received packets of size greater than 1023 bytes.

Table 6. Statistics Offsets

4.1.3 Scratchpad usage design

Three shared scratchpads (10, 11 and 12) with 30 registers each between PRU0 and PRU1 is used for keeping Receive Task and Transmit task contexts. Receive Task has two contexts to save, one for host receive and second one is for port receive.

//Structure for MII TX context scratchpad entry

```
.struct MII_TX_CONTEXT
        .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_CONTEXT
        .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
```

//Structure for MII Port RX context scratchpad entry

```
.struct MII_PORT_RCV_CONTEXT
        .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 ….. REG 17	
PRU1 Tx Context			REG 13 ….. REG 17
PRU0 Host RCV Context		REG 25 ….. RGE 29	
PRU1 Host RCV Context			REG 25 ….. REG 29
PRU0 Port RCV Context	REG 0 ….. REG 3		
PRU1 Port RCV Context	REG 4 ….. REG 7		

Table 7. Scratchpad Register Usage

Separate registers are need for host receive and port receive on scratchpad because broadcast and possibly a multicast frame is received on host as well as port queue.

Below are defines used for storing and reading context/data from/to the scratchpad and RX L2 Fifo:

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

4.1.4 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 PRU's , same allocation is true for both PRU's.

R22 is used as a persistent register on both PRU's. Bits 16-21 are used by PTP/1588 implementation (details in 4.1.17) while bits 15 and 23-31 are used by switch. Bits 0 to 14 are free for usage by other protocols.

Classification	MII Context	Descriptor Context	Temp Registers	Permanent Registers
RCV Host Queue	MII_RCV(R25..R29) RCV_DATA(R2..R9) Length(R18)	RCV_QUEUE_DESC_REG (R20-R21)	RCV_TEMP_REG_x (R20,R21,R13)	None
RCV Forward Queue	MII_RCV_PORT(R25..R29) RCV_DATA(R2..R9) Length(R18)	RCV_QUEUE_DESC_REG (R20-R21)	RCV_TEMP_REG_x (R20,R21,R13)	None
Shadow Queue	Same as RCV Host Queue	Same as Rx Host Queue QUEUE_DESCRIPTOR_REG (R10,R11) COLLISION_QUEUE_DESCRIPTOR_REG (R12,R13)	COLLISION_STATUS_REG (R21) Various Uses (R6..R9) (R15..R18)	None
Cut-Through	Same as RCV Host Queue	Same as RCV Host Queue	Same as RCV Host Queue	CUT_THROUGH_BYTE_CNT (R23.w2) PREVIOUS_R18_RCV_BYTECOUNT (R23.b1)
Tx Task	TX_CONTEXT(R13..R17) TX Data(R2..R9)	QUEUE_DESC_REG(R2..R3)	TEMP_REG_x (R0,R2,R3) Others (R10,R11)	TX_DATA_POINTER (R1.b3) TX Flags (R22.b3)
Micro-Scheduler	None	None	TEMP_REG_2 (R4)	TASK_TABLE_ROW0 (R19) CURRENT_TASK_POINTER (R1.b2)

Table 8. PRU Register Usage

4.1.5 Micro Scheduler Task design

Micro Scheduler is central to our Switch Design. It schedules various tasks which collectively implement the functionality of Switch. 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. Rx Task
2. Tx Task
3. STAT Task
4. Queue Contention Task

Micro-Scheduler 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. SOF: Start of a new receive packet at Port
2. Rx_EOF: Rx packet has been completely received
3. Tx_EOF: Tx packet has been completely transmitted

SOF:

Start of Frame signals a new receive packet at the physical port. MS reads data from the RX L2 Fifo in it's registers and checks the 6th bit of the R10 register. 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.

Rx_EOF:

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

Note: Bit 20 of R31 corresponds to RX_EOF for MII RX Data to PRU R31 ® and RX FIFO. We can't use this bit for ICSS revision 1 as on AM335x and AM437x, RX_EOF is auto-clear when a new frame arrives in RX L2 mode.

Tx_EoF:

Transmit End of Frame event signals the completion of the transmission of a frame and Tx Fifo is empty. Once this event has come, 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.

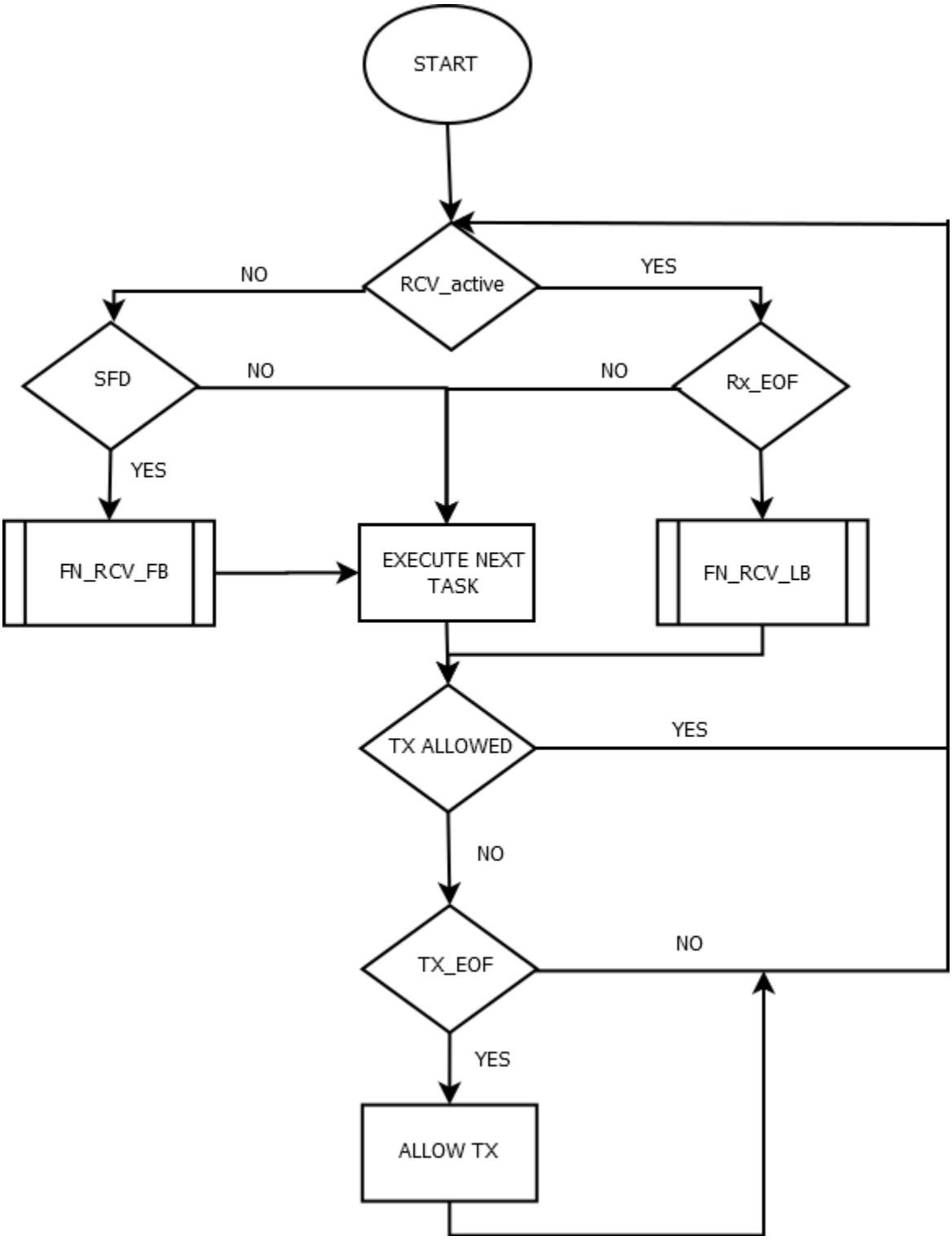


Figure 4. Micro Scheduler

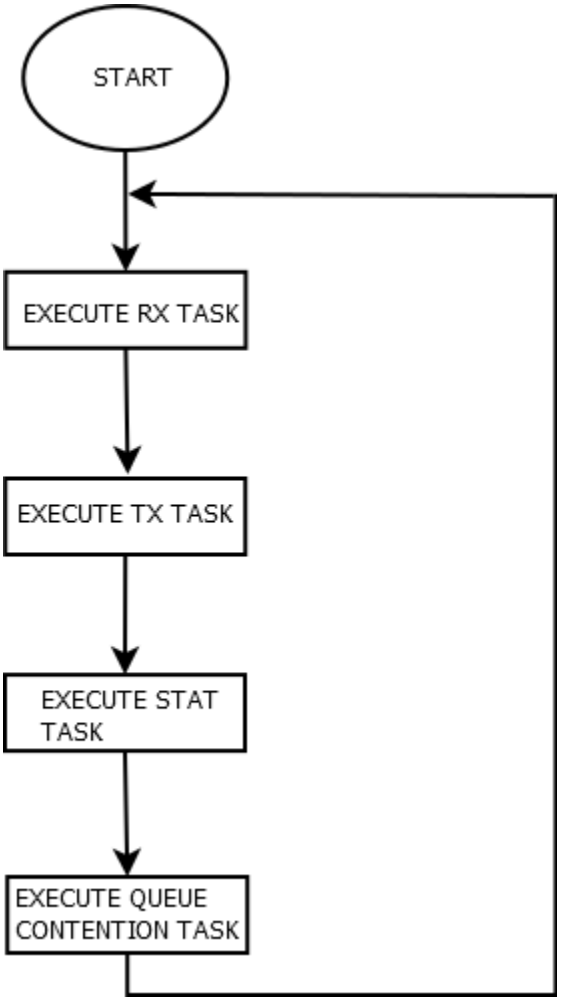


Figure 5: MS schedules tasks in round robin manner

4.1.6 Receive Task design

4.1.6.1 Quick review of L2 FIFO architecture

The incoming frames are stored in RXL2 FIFO which is composed of 2 banks.

Each bank has 32bytes of data, 16bytes of status and a 5bit 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.

4.1.6.2 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 oxffffff 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:

Unicast Frame:

If a frame is unicast then it's 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 it is either stored & forwarded on the other MII port or it cut's-through. It cut-through if the other MII port is not already occupied.

Broadcast Frame:

Broadcast frame is received in the host queue as well as forwarded. If the other MII port is not occupied then it cut's–through otherwise it is stored and forwarded.

Multicast Frame:

If a frame is multicast then a receive table is looked up using the destination address to determine whether the frame is received or not. Similarly, a look up also happens in the forward table to determine whether it is forwarded or not. If the destination address doesn't fall in the defined group of multicast addresses then it is received and forwarded like a broadcast frame. Firmware expects the multicast filter tables for receive and forward are provided by the application in a specific format. This format is described in the section 4.1.8.

4.1.6.3 Receive Task Design

Receive (Rx) Task handles the reception of frames. It can store/store & forward/cut-through a frame depending on the destination address of the incoming frame. 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 a 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. fwd_flag: This flag is set when a frame is stored & forwarded on the other MII port.
3. cut_through_flag: This flag is set when a frame cut-through's.
4. qos_queue: This field stores the priority queue as determined by the frame.
5. byte_cntr: Number of bytes already received for the frame being received.
6. buffer_index: Offset of the data buffer in L3 RAM where the receive frame is stored.
7. base_buffer_index: Offset of the data buffer in L3 RAM corresponding to the first buffer descriptor in the queue.
8. rcv_queue_pointer: Offset of the receive queue which is selected for the receive frame.
9. base_buffer_desc_offset: Offset of the base buffer descriptor for the queue selected.
10. 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. Offset's 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. FN_RCV_FB
2. FN_RCV_NB
3. FN_RCV_LB

FN_RCV_FB: Receive First Block

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 forwarded or both. If a frame

is to be received then "host_rcv_flag" is set and if the frame is to be forwarded then "fwd_flag" is set. If frame is received as well as forwarded then both the flags are set. Depending on the flags set it initializes the Rx context for the host receive or port receive or both.

When a frame has to be forwarded then first it checks whether the other MII port is free or occupied. It is possible that a host frame might be getting transmitted on the other MII port then the incoming Rx frame cannot cut-through. If the other MII port is not free then the incoming frame is stored and forwarded. Depending on the priority of the frame it is first stored/queued in one of the port queues. Later this stored frame is forwarded when the corresponding MII port becomes free. If the other MII port is free then cut-through of the incoming frame is possible. It set's the "cut_through_flag" and starts the transmission of frame by pushing the first 18 bytes of the incoming frame to the Tx Fifo.

At the end the Rx context for both the host receive and port receive are saved. If the cut-through is happening then control transfers to FN_RCV_NB otherwise it returns to the Micro-scheduler.

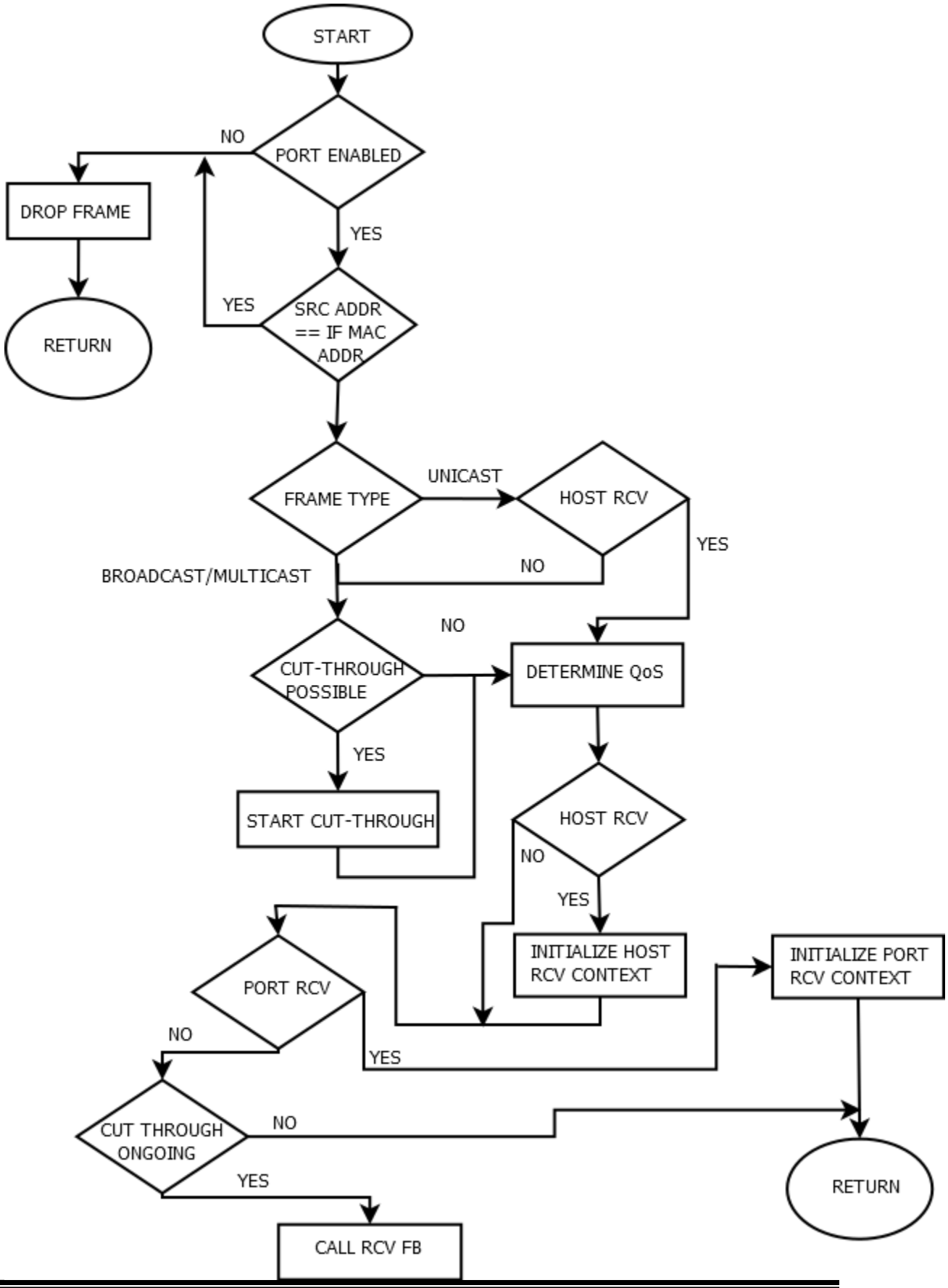

START
PORT ENABLED
NO
DROP FRAME
RETURN
YES
SRC ADDR == IF MAC ADDR
YES
FRAME TYPE
UNICAST
HOST RCV
YES
NO
BROADCAST/MULTICAST
NO
CUT-THROUGH POSSIBLE
DETERMINE QoS
YES
START CUT-THROUGH
HOST RCV
NO
YES
INITIALIZE HOST RCV CONTEXT
INITIALIZE PORT RCV CONTEXT
PORT RCV
NO
YES
CUT THROUGH ONGOING
NO
YES
RETURN
CALL RCV FB

Figure 6: RCV FIRT BLOCK

FN_RCV_NB: Receive Next Block

This block stores the data of the receive frame in the host queue or port queue or both. Maximum it can store a block of 32 bytes per call. If there is a on-going cut-through for the frame being received then it sustains the cut-through by pushing the next available data to the Tx Fifo.

Before it starts storing the frame in the queue, it calls following arbitration function with a parameter which indicates whether is called for host queue or port queue:

- FN_QUEUE_ARBITRATION: called separately for the receive in host queue and port queue

Call to the arbitration function returns the queue where the frame is to be received. Returned queue can be either the main receive queue or shadow queue.

Once a queue is acquired, it starts storing the frame in blocks of 32 bytes. It checks whether new 32 bytes have become available, if yes then stores them otherwise transfers control to Micro-scheduler. If a frame is received as well as forwarded then the 32 byte data is stored in both the queues. 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.

If cut-through is happening then it doesn't return the control and stays in till the entire frame has been received and cut-through of the frame is complete.

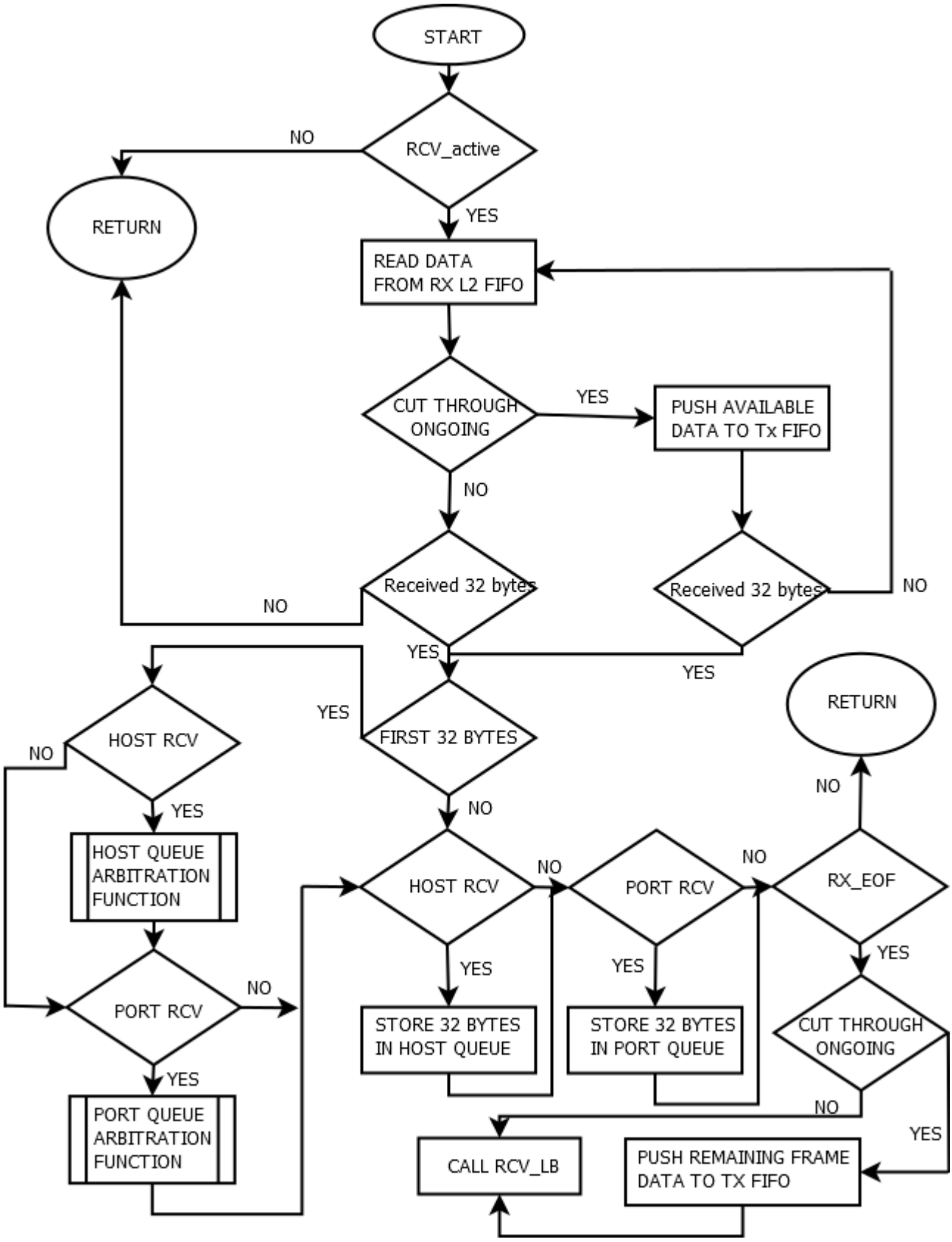


Figure 7: RECEIVE NEXT BLOCK

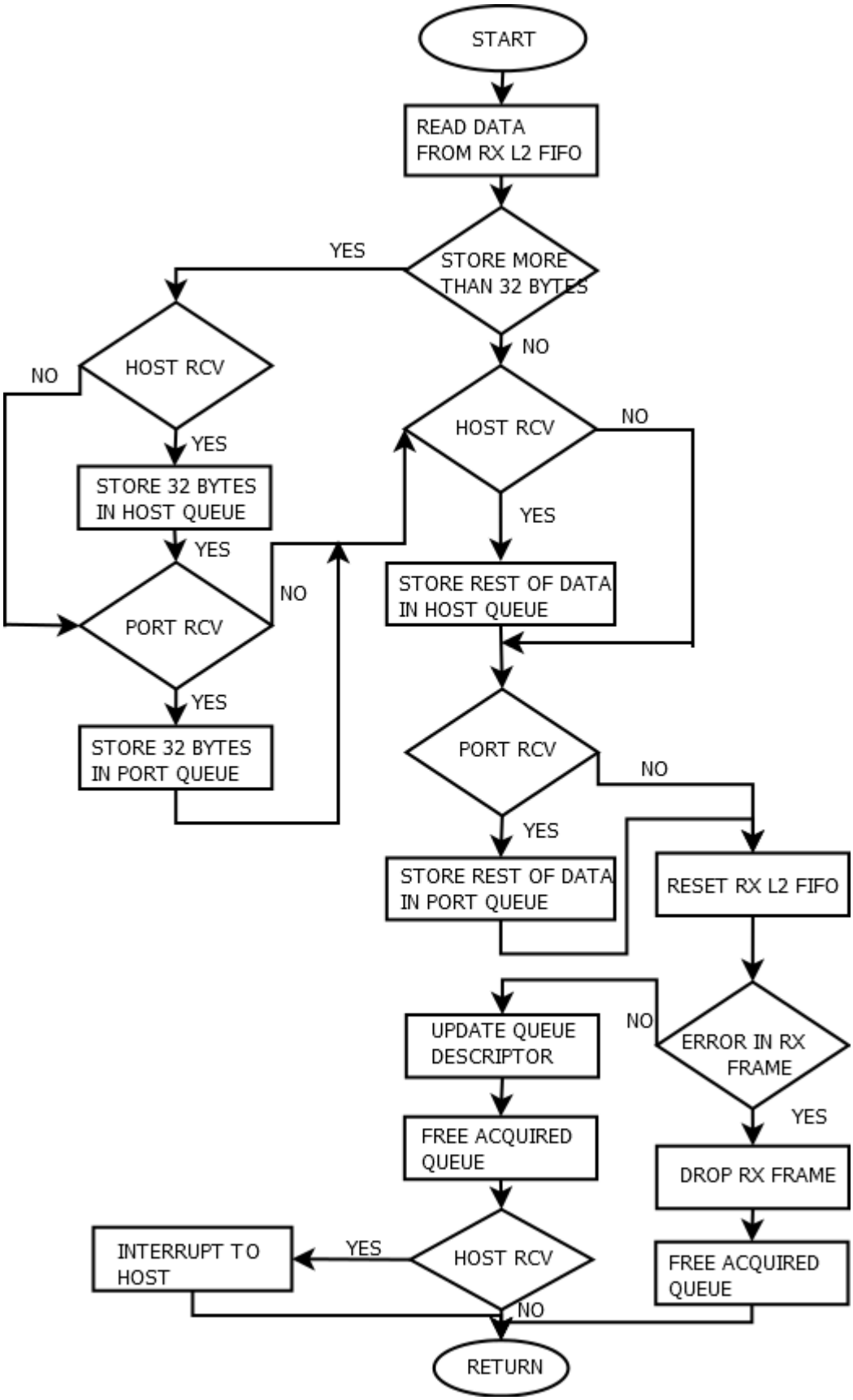
FN_RCV_LB: Receive Last Block

This block is executed when the RX EOF event has occurred. This block may store less the 32 bytes or exact 32 bytes or more then 32 bytes. In-case where it stores more than 32 bytes of data, it stores from both the bank's 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. For stored and forwarded frames, the Tx Task comes to know about the received frame only when the queue descriptor for the port queue is updated.

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 STAT's task. Error handling is discussed in a separate section.

At last it clears the RCV_active bit, clears the RX EOF in ICSS INTC and transfers control back to the Micro-scheduler.


START
READ DATA FROM RX L2 FIFO
STORE MORE THAN 32 BYTES
YES
NO
HOST RCV
NO
YES
STORE 32 BYTES IN HOST QUEUE
YES
PORT RCV
NO
YES
STORE 32 BYTES IN PORT QUEUE
HOST RCV
NO
YES
STORE REST OF DATA IN HOST QUEUE
PORT RCV
NO
YES
STORE REST OF DATA IN PORT QUEUE
RESET RX L2 FIFO
UPDATE QUEUE DESCRIPTOR
NO
ERROR IN RX FRAME
YES
FREE ACQUIRED QUEUE
DROP RX FRAME
INTERRUPT TO HOST
YES
HOST RCV
FREE ACQUIRED QUEUE
NO
RETURN

4.1.7 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	inidcates that "index" is pointing into shadow buffer
15	TimeStamp	indicates that this packet has time stamp in seperate buffer - only needed of PTCP runs on host
16..17	Port	different meaning for ingress and egress, ingress Port=0 inidcates phy port 1 and Port = 1 inidcates phy port 2. Egress: 0 sends on phy port 1 and 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 access only one BD
29	VlanTag	indicates that packet has Length/Type field of 0x08100 with VLAN tag in following byte
30	Broadcast	inidcates 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 9. Buffer Descriptor Table

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.

0..15	Rd_ptr	Read pointer. This points to the last buffer descriptor that points to valid data, when Rx tasks 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	

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 10. Queue Descriptor Table

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 (See section 4.1.3) 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.

4.1.8 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 each for the host and port. A received frame is parsed using quality of service rules to determine in which queue the frame would be received. QoS rules are same for the host and port receive queues. Following QoS rules are implemented by the firmware:

1. All the non VLAN tagged frames are stored in the lowest priority queue (queue priority 3).
2. VALN tagged frames with "Priority code point (PCP)" value of 6 and 7 are stored in highest priority queue (queue priority 0).
3. VALN tagged frames with "Priority code point (PCP)" value of 4 and 5 are stored in second highest priority queue (queue priority 1).
4. VALN tagged frames with "Priority code point (PCP)" value of 2 and 3 are stored in third highest priority queue (queue priority 2).
5. VALN tagged frames with "Priority code point (PCP)" value of 1 and 2 are stored in lowest priority queue (queue priority 3).

QoS rules are under the "ETHERNET_QOS" define in the "switch_MII_Rcv.p" file.

4.1.9 Multicast Filter table definition/format

Firmware expects the multicast filter table to be provided in a specific format. Following is an example of a valid multicast filter table where each bit position covers one multicast address. Multicast address of the received frame is mapped to the table by the firmware to read the corresponding bit. If the value of bit is 1 then the frame is received and if it is 0 then it is not received.

```
const unsigned int Host_Receive_Decision_Table[] =  {
    0x00000003,
    0x000FF000,
    0xFFFFFFFF,
    0xFFFFFFFF,
    0x00000000
}
```

By default there is a space of 256 bytes reserved each for the static multicast receive and forward table. In total 2048 multicast addresses are covered by 256 bytes filter table.

Firmware assumes that table corresponds to a continuous set of multicast addresses. Application needs to specify the first and last addresses of the multicast group for which the filter table has been provided.

4.1.10 Transmit Task design

4.1.10.1 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 then first it 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:

6. BUFFER_INDEX : Offset address of the data buffer in L3 RAM which contains first 32 bytes of the frame.
7. Packet_Length: Length of the transmit frame in number of bytes.
8. BYTE_CNT: Number of bytes already pushed to the Tx Fifo for the transmit frame.
9. BUFFER_DESC_OFFSET: Offset of the first buffer descriptor for the transmit frame. First buffer descriptor contains the length of transmit frame.
10. BASE_BUFFER_DESC_OFFSET: Offset of the base buffer descriptor of the queue in which transmit frame is queued.
11. BUFFER_OFFSET: Offset of the base data buffer in the L3 RAM for the send queue.
12. TOP_MOST_BUFFER_INDEX: Offset of the top most data buffer in the L3 RAM for the send queue.
13. TOP_MOST_BUFFER_DESC_OFFSET: Offset of the top most buffer descriptor of the send queue.

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. Offset's 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:

4. XMT_FB
5. XMT_NB
6. XMT_LB

XMT_FB: Transmit First Block

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.

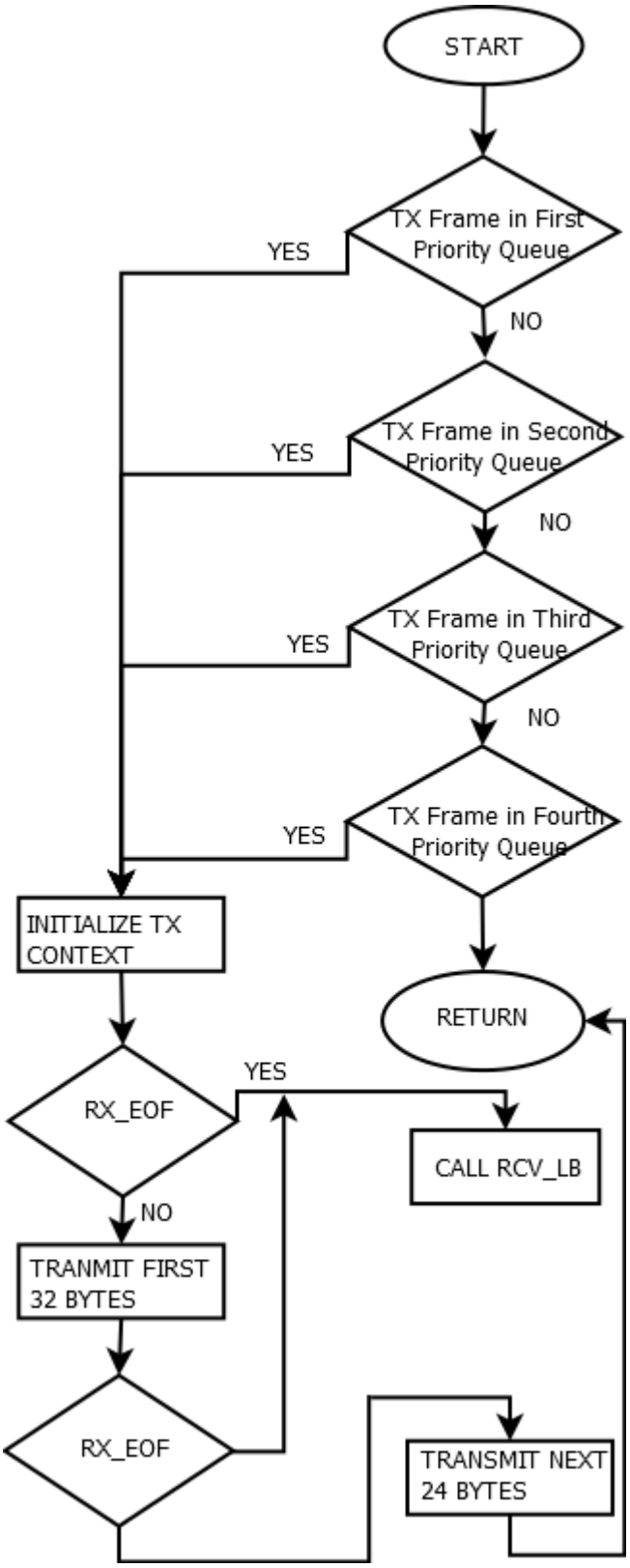


Figure 8 : Transmit First Block

XMT_NB: Transmit Next Block

This block of code fetches and transmit's rest of the frame apart from the last 32 bytes of data. It computes the Tx Fifo fill level using the IEP counter and fills the available free space in Tx Fifo with the subsequent data of the frame. Maximum it fills 32 bytes of data.

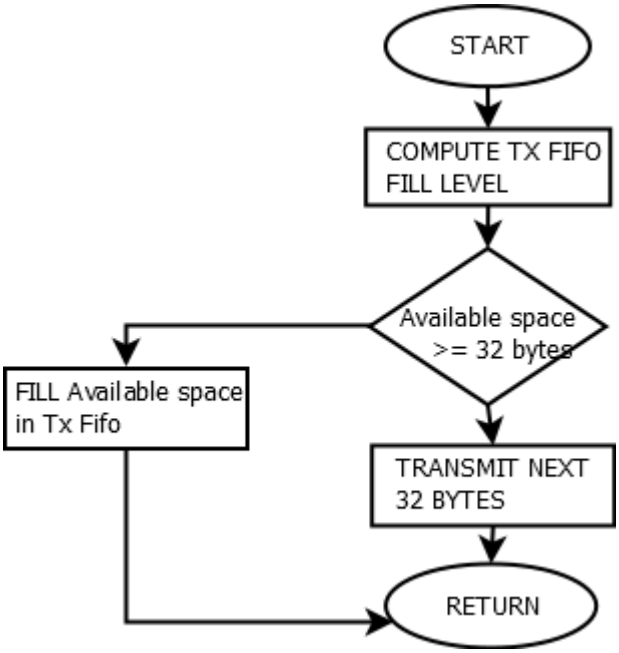


Figure 9 : Transmit Next Block

XMT_LB: Transmit Last Block

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.

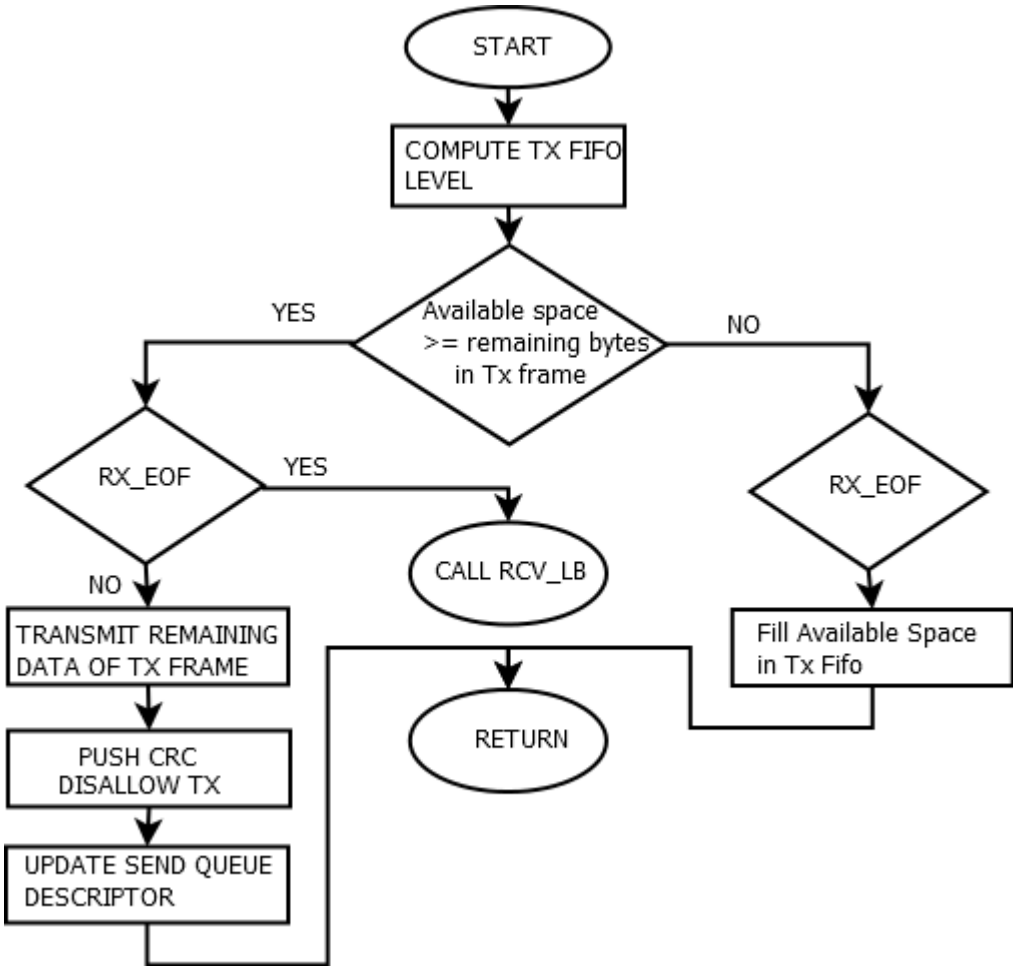


Figure 10 : Transmit Last Block

4.1.11 Queue Contention Task Design

Queue contention happens when two PRU or a PRU and Host want to acquire the same queue at same time. There is a defined queue arbitration scheme according to which ownership of queue is allocated when two PRU or a PRU and host try to acquire same queue.

Queue arbitration scheme defines a Master and a Slave. A core can be Master or Slave depending on the scenario. Below are definitions of Master and Slave:

Master:

Master first checks the slave ownership bit. If this bit is set then it means that slave has already acquired the queue. Master then acquires the Shadow queue and stores the frame in Shadow queue. If the Slave ownership bit is clear then Master sets the master ownership bit and acquires the queue.

Slave:

Slave first checks the master ownership bit. If this bit is set then it means that master has already acquired the queue. It then acquires the Shadow queue and stores the frame in Shadow queue. If Slave finds the master ownership bit as clear then it sets the Slave ownership bit and reads again the master ownership bit to check that master didn't acquire the queue in-between. If the master has acquired the queue in-between then slave releases the queue by clearing the slave ownership bit. Slave then acquires the shadow queue and stores the packet in it.

In the queue descriptor there is a separate ownership bit for master and slave. These bits are in separate bytes.

As per the scheme, below cores are the defined masters and slave with respect to the queues:

- Host Queues: PRU0 is master and PRU1 is slave.
- Port Queues: PRUx is master and Host is slave.

Queue Contention on Host Queue:

Queue contention happens on a host queue when both the PRU's try to acquire a queue to store the frame at the same time. Only one PRU can hold the queue at one point of time. Following are possible scenarios:

1. PRU0 is already holding the queue then PRU1 stores the frame in shadow queue.
2. PRU1 is already holding the queue then PRU0 stores the frame in shadow queue.
3. Queue is free and both PRU's try to acquire it by calling the arbitration function. PRU0 competes as master and PRU1 competes for the queue as slave. Depending on the relative timing between the two PRU's one will successfully acquire the main queue whereas the other one will acquire the shadow queue.

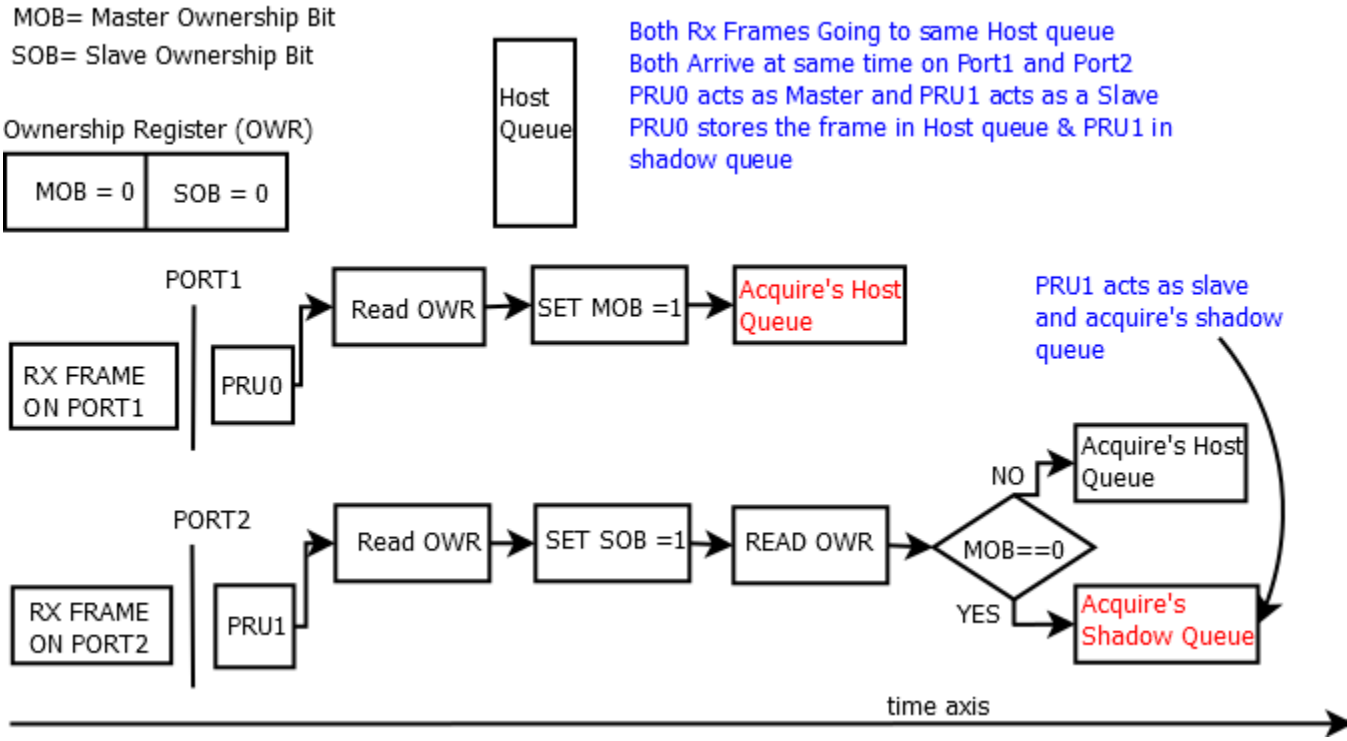


Figure 11 : Host Queue Contention Scenario

Queue Contention on Port Queue:

Queue contention happens on a port queue when Host and PRU try to acquire a queue to store the frame at the same time. This can happen on the queues of both the ports:

- On Port1, contention can happen between Host and PRU1.
- On Port2, contention can happen between Host and PRU0.

As per the scheme, Host is always a slave and PRUx is always a master when competing for a port queue. PRU's call a separate arbitration function to acquire the port queue. A separate function is needed because with respect to port queues both PRU's act as master.

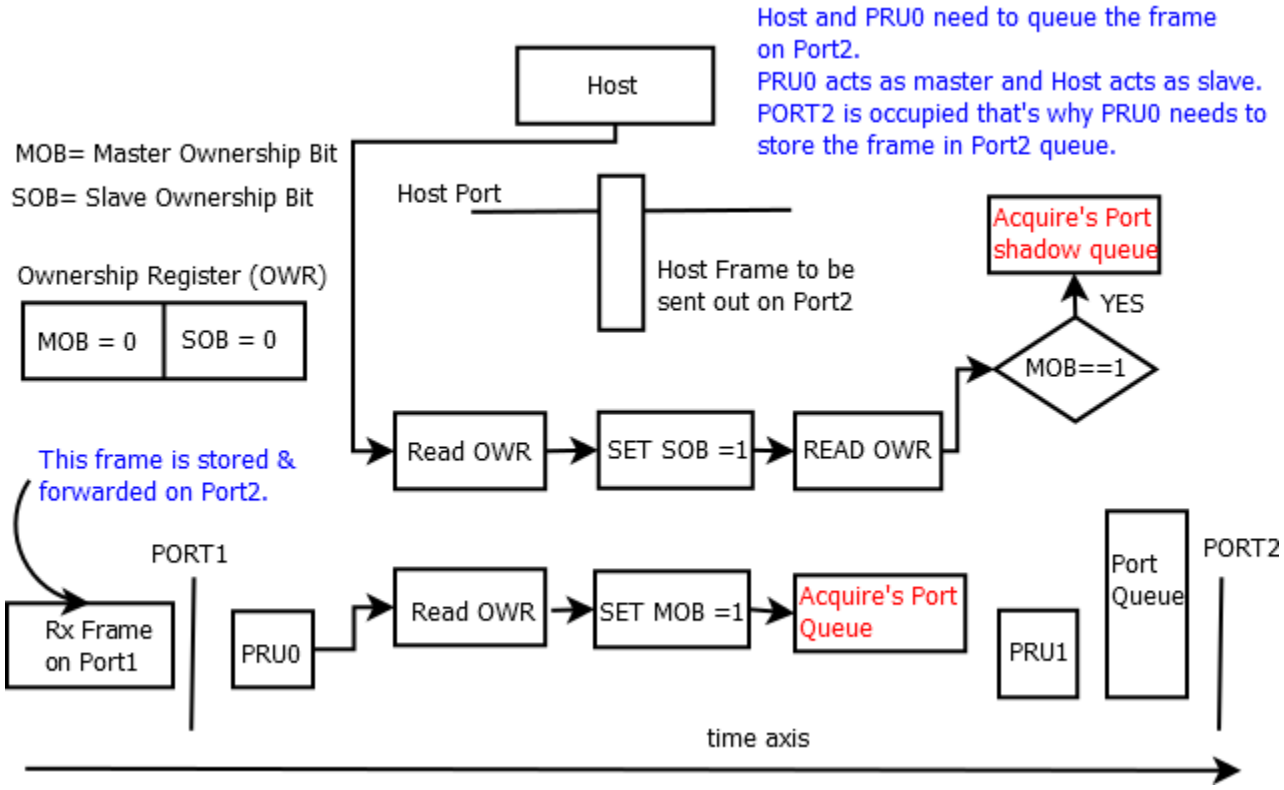


Figure 12 : Port queue contention scenario

Queue Contention Resolution

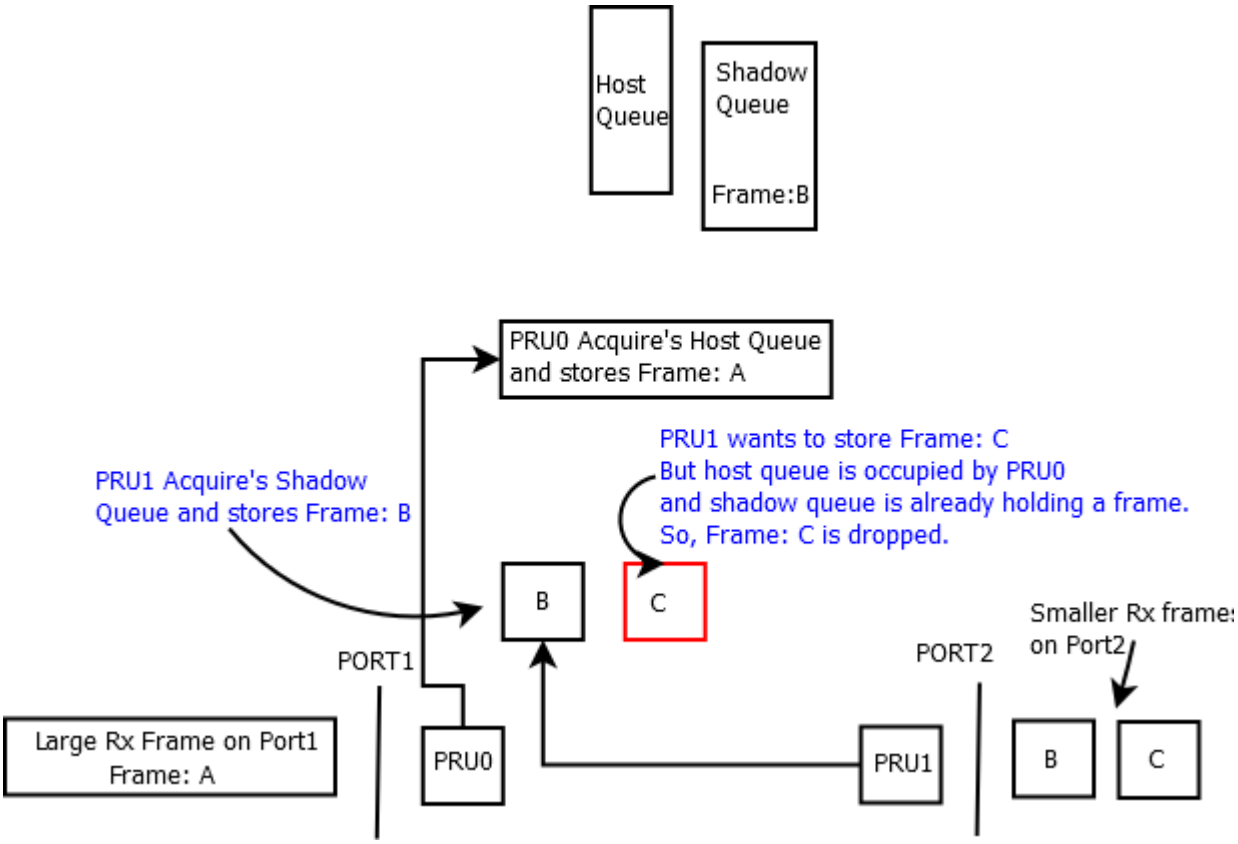
When queue contention happens then one of the core will store/queue the frame in the shadow buffer. Frame stored in the shadow buffer has to be moved to the main queue. Host always receives the frames through the four host queues and frames are always transmitted from the four send queues on both the ports. There is queue contention task which resolves the queue contention and moves the frame from the shadow buffer to one of the four main queue's depending on the priority of the frame. Queue contention task runs on both the PRU's:

- Queue contention task on PRU0 resolves contention for Host port and Port2.
- Queue contention task on PRU1 resolves contention for Port1.

So, if there is frame waiting in shadow buffer of Port1 then it is moved to one of the send queue of Port1 by the PRU1. This distribution of queue contention resolution to two PRU's helps in resolving the queue contentions quickly.

This task checks the queue contention status register to find whether it has to resolve a queue contention. If there is no contention to resolve then it transfers the control back to the micro scheduler. There is one byte each for the three ports in the queue contention status register. If there is a queue contention then this task clears the contention in status register after it has successfully resolved it.

Shadow buffer can have maximum one frame at a time. If a queue contention happens and new a frame has to be stored in shadow buffer while it is the already holding a frame then the new frame is dropped.



PRU has to acquire the main queue before queue contention can be resolved. As per the scheme, PRU is designated as master while it resolves the queue contention. For example, PRU1 acquires the one of the send queue on Port1 as master when it moves a frame from the shadow buffer to the send queue. At this point of time if host wants to queue a frame in the same send queue then it will have to drop the frame. It can't store the frame in shadow buffer as well because it is already holding a frame which is being moved to the send queue.

4.1.12 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 stat task checks for two flags RX_STAT_PEND and TX_STAT_PEND which are in turn set by the RX and TX task respectively (The 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.

The statistics memory map is shown in section 4.1.2.5.

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.

4.1.13 Storm Prevention

Storm prevention is primarily done on PRU's using a credit based scheme. It is explained below.

- The Host (Cortex A8) 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 Cortex A8 writes the value once again. Function `resetStormPreventionCounter()` in file `icss_StormControl.c`
- Storm Control can be enabled and disabled on a per port basis.
- List of API's (Mentioned in switch API doc in section 2.2)

4.1.14 Learning Bridge

- The learning bridge is implemented on the Host (Cortex A8) side. The switch learns source addresses of packets that are either Multicast/Broadcast or directed at the Host (Unicast with Host MAC ID).
- All API's are contained in the file `icss_learning.c`. When a packet is received on the Host using the RX Interrupt mechanism explained above in this document the function `updateHashTable()` is called for that packet with the correct port number. The function adds an entry for the MAC ID and updates the table.
- To retrieve the port number on which that particular MAC ID was present, the API `findMAC()` is called.
- Ageing is done automatically by the function `incrementCounter()` **and** `ageingRoutine()` (Called inside the driver periodically)
- List of API's (Mentioned in switch API doc in section 2.2)

4.2 Firmware sources description

ICSS Switch firmware source includes the following files:

Source File	Remarks
firmware_version.h	ICSS Switch Firmware Version Control
icss_defines.h	ICSS Global Defines
icss_switch_macros.h	ICSS Switch Macros and Defines
switch_collision_task.asm	ICSS Switch collision handling 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 11. Firmware Sources Description

5 Revision History

Version #	Date	Author Name	Revision History
0.1	11 JUNE 2013	Robin Singh	First Draft
0.2	23rd July 2014	Robin Singh	Made modifications to align with latest firmware
0.3	24th July 2014	Robin Singh	Incorporated review comments
0.4	25th July 2014	Vineet Roy	Incorporated review comments
1.0	25th July 2014	Robin Singh	Final document which aligns with latest switch firmware.
1.1	2nd Jan 2014	Vineet Roy	1. Modified the memory map based on recent changes and modified statistics task 2. Modified the statistics task section 3. Added section on half duplex 4. Added section on Link Status Change 5. Added section on PTP
1.2	11-Feb-2014	Vineet Roy	Added section on Buffer and Queue Descriptors
1.3	6-September-2017	Suraj Das	Modified the documentation for Proc SDK release.
1.4	16-Apr-2018	Aravind Batni	Added note for bit 20 of RX_EOF on why it can't be used for AM335x and AM437x

«««« § »»»»