Freescale Semiconductor

HCL TECHNOLOGIES

Multi Service Access Line Card Application Note

Preliminary Draft

Rev 1.2

HCLMSP-AN/D

Table of Contents

1 Modification History

Rev	Date	Author	Department	Changes
1.0	06-Feb-03	HCL Technologies	Networking	Draft version.
1.1	14-Feb-03	HCL Technologies	Networking	Additional information is added on Q-3, Performance calculations, IMEM/DMEM estimates, ATM TM, IP QoS, Host APIs
1.2	14-Mar-03	J.Bednarek	Motorola, C-Port	Changes for draft for SNDF CD

2 Overview

This document describers the design of a Multi Service Access (MSA) line card. The intended audiences of this document are system architects, hardware designers, software designers, testers and programmers of the line card based on the C-Port network processor family.

The reader of this document is expected to have a fair understanding of the C-3e NP architecture and the associated co-processor such as Q-3 (Traffic Management Co-processor) with the basic understanding of C-Port Family of TDM Adapters (or Twister) (Mt-21) used in the design of the MSA line card.

MSA line card application provides multiple services on different ports. It can be connected to two Twister Mt-21 chips. The twister Mt-21 chip can have 32 T1/E1 interfaces supporting upto 1K channels.(note, Mt-21 can support even higher channels, ie 2048, but in this example, we are assuming 1000 channels).

Feature Overview and Standards Support

This application supports the following features:

- 64 T1/E1 interfaces supporting 2K channels
- PPP / FR header processing and reassembly
- MLPPP segmentation and reassembly
- AAL1/5 segmentation and reassembly
- ATM Cell switching
- FR switching
- IPv4 Unicast Routing on all interfaces (PPP/ATM/FR)
- Multi-Protocol Label Switching (MPLS) on all interfaces
- Ingress / Egress packet processing for MPLS
- IP IntServ and DiffServ
- ATM Traffic Management 4.1
- MPLS QoS
- IMA

MSA line card is intended to work in a stack of MSA cards connected on the switching fabric for communication with the other MSA cards as well as line cards that terminate ATMs. The host module manages and maintains the statistics for the entire system. The communication of the host with the line cards is through the PCI interface. Figure 1 helps in understanding the intended use of the MSA line card.

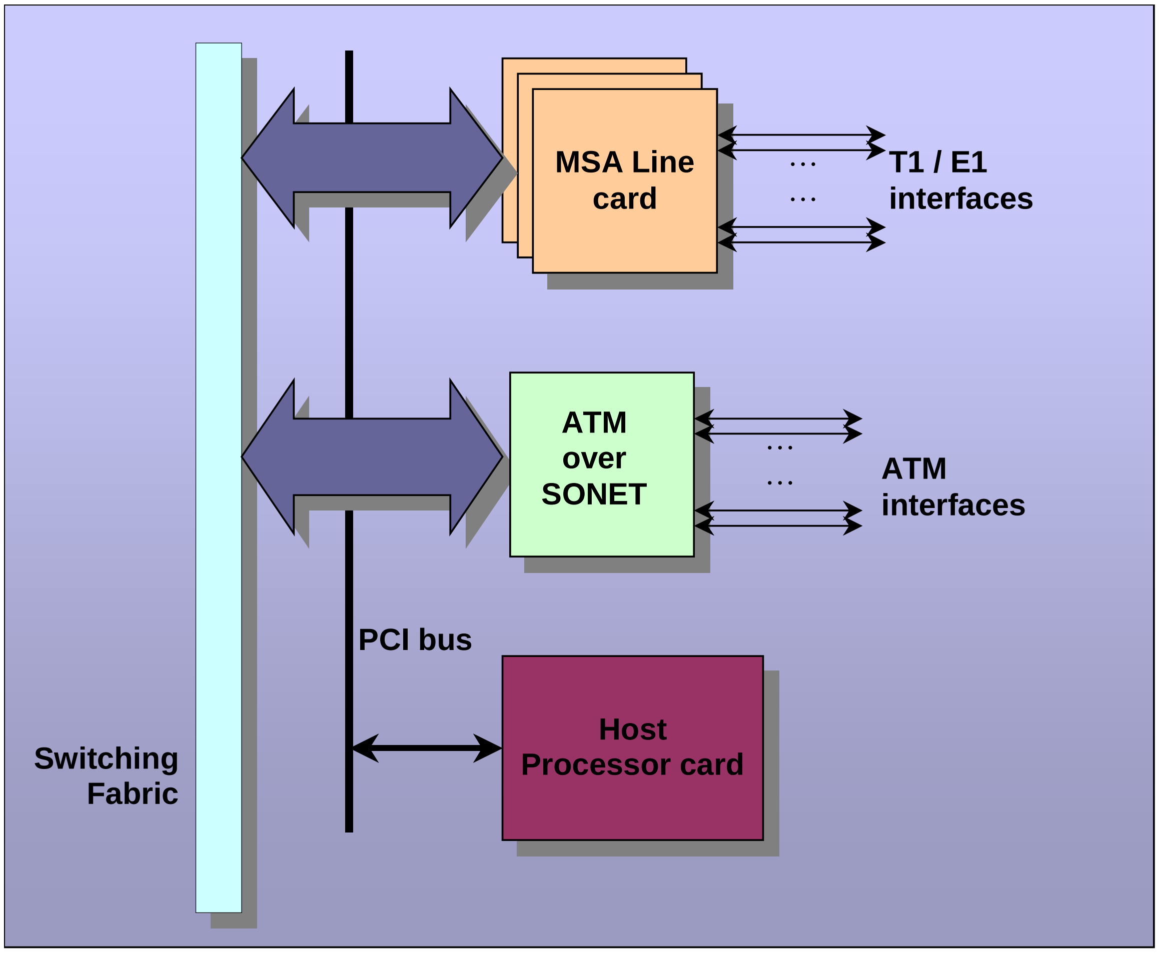

Figure 1: MSA card within the access platform

3 Definitions, Acronyms and Abbreviations

Abbreviation	Description
AAL	ATM Adaptation Layer
AAL-1	ATM Adaptation Layer 1
ABR	Available Bit Rate
AF	Assured Forwarding
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
ATM TM	ATM Traffic Management
BE	Best effort
BOM	Beginning of Message.
CBR	Constant Bit Rate
CID	Channel ID
CIDR	Classless Inter Domain Routing
CPI	Common Part Indicator.
CPRC	Channel Processor RISC core.
CRC	Cyclic Redundancy Check.
DLCI	Data Link Connection Identifier
DWRR	Dynamic Weighted Round Robin
EF	Expedited Forwarding
EOM	End of Message.
FEC	Forwarding Equivalence class
FR	Frame Relay
HDLC	High Level Data Link Control
HEC	Header Error Control.
HTK	Hash Trie Key.
ICMP	Internet Control Message Protocol
ICP	IMA Control Protocol.
IP	Internet Protocol
LCP	Link Control Protocol
LLC	Logical Link Control
LMI	Local Management Interface
LPM	Longest Prefix Match.
LSP	Label switched path
MIB	Management Information Block
ML/PPP	Multi-Link PPP
MPLS	Multi-Protocol Label Switching
MTU	Maximum transmission Unit
NCP	Network Control Protocol
NLPID	Network Layer Protocol ID
OAM	Operation, Administration and Maintenance.
PDU	Protocol Data Unit.
PHB	Per Hop Behavior
PPP	Point to Point Protocol
QoS	Quality Of Service

RARP	Reverse Address Resolution Protocol

Abbreviation	Description
RED	Random Early Discard
RM	Resource Management.
RR	Round Robin
SDU	Service Data Unit.
SNAP	Subnetwork Access Protocol.
TCP	Transport Control Protocol
TDM	Time Division Multiplexing
TLU	Table Lookup Unit.
TMC	Traffic Management Co-Processor
TOS	Type of Service
TTL	Time To Live
UUI	User-to-User Interface.
VC	ATM Virtual Connection
VOP	Virtual Output Port
VP	ATM Virtual Path
VPCI	Virtual Path Identifier/Virtual Channel Identifier
WFQ	Weighted Fair queueing

4 Related Documents

This section lists down the various documents used as reference while developing this application notes.

- MSA Line card software requirement specifications from C-Port.
- Guide to C-Ware WNI Applications, CST2.2
- ATM Cell Switch Application Guide, CST 2.1.1
- RFC 791, Internet Protocol
- RFC 1332, The PPP Internet Protocol Control Protocol (IPCP)
- RFC 1471, The Definitions of Managed Objects for the Link Control Protocol of the Point-to-Point Protocol
- RFC 1473, The Definitions of Managed Objects for the IP Network Control Protocol of the Point-to-Point Protocol
- RFC 1661, The Point to Point Protocol (PPP)
- RFC 1812, Requirements for IP Version 4 Routers
- RFC 1990, The PPP Multilink Protocol (MP)
- RFC 2427, Multiprotocol Interconnect over Frame Relay
- RFC 2474, Definition of the Differentiated Services Field (DS Field) in the IPv4
- RFC 2475, An Architecture for Differentiated Services
- RFC 2597, Assured Forwarding PHB Group

- RFC 2598, An Expedited Forwarding PHB
- RFC 2702, Requirements for Traffic Engineering Over MPLS
- RFC 2684, Multi Protocol Encapsulation over ATM Adaptation Layer 5
- RFC 2697, A Single Rate Three Color Marker
- ITU I.361, B-ISDN ATM Layer Specification
- ITU I.363.1, B-ISDN ATM Adaptation Layer Specification: Type 1 AAL
- ITU I.363.5, B-ISDN ATM Adaptation Layer Specification: Type 5 AAL
- ITU I.610 B-ISDN Operation and Maintenance Principles and Functions
- ATM Forum, Inverse Multiplexing for ATM (IMA) Specification Version 1.1
- Frame Relay to ATM to 10/100 Ethernet Switch Router Application Guide, CST2.1
- C-ware Q-5 TMC API User guide Rev 00
- DiffDocQ-512003.doc - Functionality Comparison Between “Old” Q-5 TMC Design and Projected Q-5 TMC FPGA
- RFC3034 –Use of Label Switching on Frame Relay Networks specification
- RFC 2702 – Requirements for traffic engineering over MPLS
- RFC3270 – Multi-protocol label switching support of differentiated services
- RFC 3031 – Multi-protocol label switching architecture
- RFC 3032 – MPLS label stack encoding
- Draft-ietf-mpls-ttl-04.txt - Time to Live Processing in MPLS networks.
- RFC 3035 – MPLS using LDP and ATM VC switching

5 Application Mapping

This application is comprised of many software components, each of which is divided into smaller components. The functional partitioning of the software is depicted in figure 2 with the clustering and re-circulation information. The partitioning is designed to handle 1024 channels per cluster. C-3e NP is chosen for implementing the MSA line card as the processing power of the NP and the T1/E1 line interfaces match and also Q-3 is used for managing the traffic management for IP, ATM and MPLS.

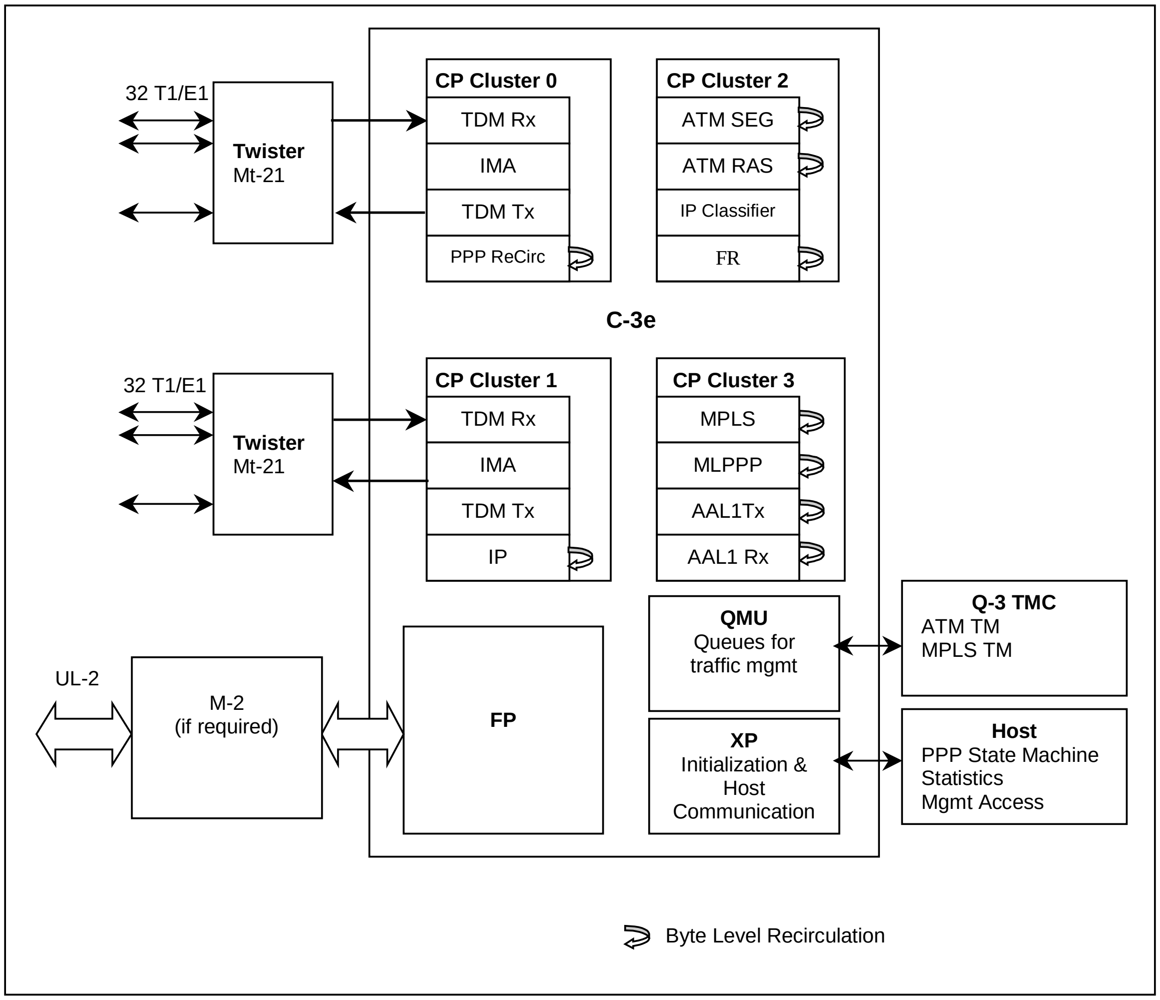

Figure 2: MSA line card functional mapping on C-3e NP

6 Network processor architecture

The MSA application consists of many software components. One component executes on the host and the other components execute on the various CPs within the C-3e. Each of the NP software components provides a subset of the features of the application. Mapping between software components and CPs was shown in the figure 2. The data paths between these components can be conceptualized as a group of busses. In this context, a bus is the combined use of queues and buffer memory to forward data between two components. The queue number is analogous to the address on the bus. Each of the buses implies a different buffer and descriptor format (for TDM, IP, MPLS, and so on). The traffic originating from TDM channels will be HDLC or ATM or Transparent Chunk based on the channel configuration. HDLC traffic will further be identified as FR traffic or PPP traffic.

A buffer and a buffer descriptor can specify the interface to a component. Table below lists each of the components and describes their interface. A component may have multiple interfaces and therefore multiple entries in the table. Unless specified otherwise in the table, the port field indicates the output port and the length field indicates the number of bytes in the buffer. The various buffer formats are described in section 7.19 and the buffer descriptor formats are described in appropriate sections.

Component	Buffer Format	Descriptor format	Comments
TDM Rx	BT_ATM, BT_MPLS_FR, BT_MPLS_PPP, BT_MPLS_ATM	ATM	Only header field required
	BT_MPLS	MPLS	
	BT_HDLC	N/A	
	BT_TDM_TRANSPARENT	Transparent Chunk	
TDM Tx	BT_ATM	ATM	Only header field required
	BT_HDLC	N/A	
IMA Tx	BT_ATM	ATM	Port indicates outport which maps to IMA group; only header field required
IMA Rx	BT_IMA_CP, BT_IMA_FILLER	OAM	Port indicates input port
	BT_ATM	ATM	Port indicates input port
ML PPP	BT_HDLC, BT_PPP, BT_MLPPP	ML-PPP	Port indicates input port; reassembly will be performed
	BT_IPV4, BT_CONTEXT_STATE, BT_NCP_xxx	TDM	Only mcClass field required; segmentation will be performed
IP	BT_IPV4	N/A	Port indicates input port; IP forwarding will be performed
	BT_HDLC, BT_PPP	N/A	Port indicates input port; PPP encapsulation will be removed and IP forwarding performed
FR	BT_MPLS_FR, BT_MPLS_PPP, BT_MPLS_ATM	MPLS	
	BT_FR	FR	
MPLS	BT_MPLS_PPP, BT_MPLS_FR, BT_MPLS_IPV4, BT_MPLS_ATM	MPLS	Buffer type identifies the egress port interface type.
Segmentation	BT_IPV4	Seg	EgressQueue field is required.
Reassembly	BT_ATM	ATM	Port indicates input port; AAL-5 reassembly will be performed
IP QoS	BT_IPV4	TDM	
AAL-1 Rx	BT_TDM_TRANSPRENT BT_ATM	TDM ATM	

AAL-1 Tx	BT_TDM_TRANSPRENT BT_ATM	TDM ATM	
UL-2	BT_ATM	ATM	
Host	BT_ATM	ATM	Port indicates the input port; only header field required
	All others	N/A	Port indicates the input port

6.1 Data Paths

This section explains about various data paths originating from T1/E1 interfaces, flowing through other components in the NP and going out through T1/E1 interfaces. Two Twisters (Mt-21) are supported in this application.

6.1.1 Data Paths for FR frames

This section describes the conceptualized data flows for FR frame received in TDM RX. The FR chunks will be reassembled as FR frame in TDM RX, recirculated in other component CPs and finally transmitted as FR / PPP / AAL5 chunks via TDM Tx. The detailed flow shown in figure 3 is described as follows.

- FR frame is received as HDLC chunks in TDM RX, gets reassembled and identified as FR frame based on the channel configuration. Then it will be enqueued to FR queue for further FR processing.
- FR component performs the DLCI lookup. Based on lookup response, it will enqueue the frame into IP queue or MPLS queue. For FR switching, it will modify the FR header (with new DLCI value) and enqueue into appropriate TDM Tx queue.
- IP component dequeues the FR frame from its queue, removes the FR header and enqueues the IP packet into destination queue (TDM recirculation queue or ATM Segmentation queue or MPLS queue or IP QoS queue determined by IP lookup and port lookup result.
- MPLS removes the FR header (if it exists) from the frame, performs the label processing and then enqueues the packet to destination queue (TDM recirculation queue or ATM Segmentation queue or IP queue or Q-3 traffic queue determined by MPLS lookup result.
- ATM segmentation component will segment the IP packet into AAL5 cells, inserts the ATM header and enqueues these cells to appropriate TDM Tx queue or to Q-3 traffic queue.
- TDM recirculation component encapsulates the packet into FR or PPP frame, enqueues it to TDM Tx for final transmission over TDM channel.
- TDM Tx dequeues the HDLC frame(FR frame or PPP frame, No differentiation is made between PPP and FR frame) or ATM cells from its queue. It transmits TDM chunks of size 64 bytes. For ATM, each cell will fit into a TDM chunk. For FR or PPP, it segments the frame into TDM chunks.
- If QoS treatment is needed, packets will be enqueued to Q-3 traffic queue from IP QoS classifier or ATM Segmentation or MPLS components for applying various QoS parameters. Q-3 TMC provides marking/dropping, policing and traffic shaping for the packet based on configured traffic parameters. Q-3 TMC

will enqueue the conformant packets into QMU queue. Non-conformant packets will either be discarded or marked.

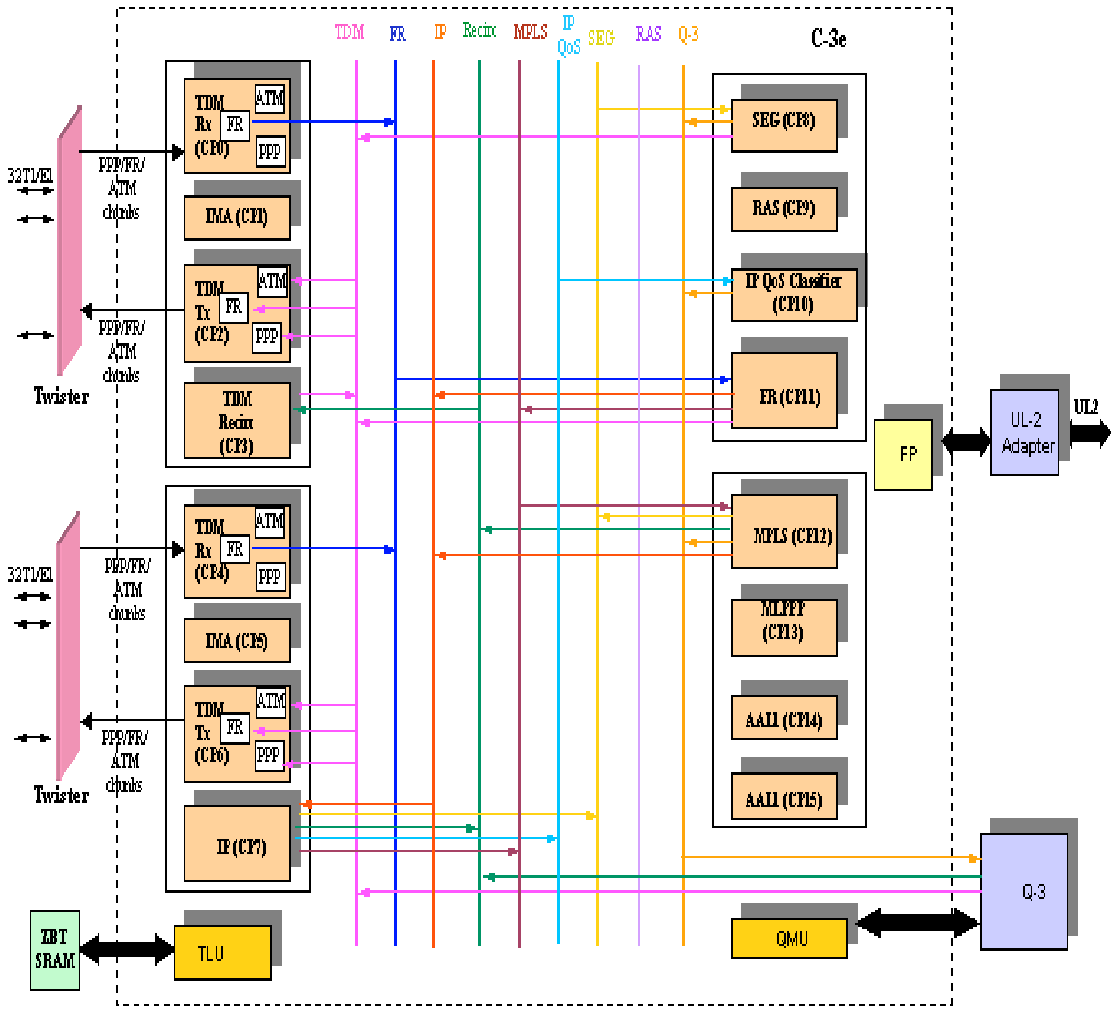

Figure 3. Data paths for FR frames

6.1.2 Data Paths for PPP frames

This section describes the data flow for PPP frames received in TDM RX. The PPP chunks will be reassembled as PPP frame in TDM RX, recirculated in other component CPs and finally transmitted as FR / PPP / AAL5 chunks via TDM Tx. The detailed flow shown in figure 4 is described as follows.

- PPP frame is received as HDLC chunks in TDM RX, gets reassembled and identified as PPP frame based on the channel configuration. Then it will be enqueued to IP queue or MLPPP queue or MPLS queue component based on PPP protocol field in frame.
- IP component dequeues the PPP frame from its queue, removes the PPP header and enqueues the IP packet into destination queue (TDM recirc queue or ATM Segmentation queue or MPLS queue or IP QoS queue determined by IP lookup and port lookup result.
- MPLS removes the PPP header (if it exists) from frame, performs the label processing and then enqueues the packet to destination queue (TDM recirc queue or ATM Segmentation queue or IP queue or Q-3 traffic queue (if QoS is needed)) determined by MPLS lookup result.
- MLPPP component dequeues from its queue, removes MLPPP encapsulation from the frame, and reassembles MLPPP fragments and enqueues the reassembled fragment to IP queue.
- ATM segmentation component will segment the IP packet into AAL5 cells, inserts the ATM header and enqueues these cells to appropriate TDM Tx queue or to Q-3 traffic queue.
- TDM recirculation component encapsulates the packet into FR or PPP frame, enqueues it to TDM Tx for final transmission over TDM channel.
- TDM Tx dequeues the FR frame or PPP frame or AAL5 cells from its queue. It transmits TDM chunks of size 64 bytes. For ATM, each cell will fit into a TDM chunk. For FR or PPP, it segments the frame into TDM chunks.

If QoS treatment is needed, packets will be enqueued to Q-3 traffic queue from IP QoS classifier or ATM Segmentation or MPLS components for applying various QoS parameters. Q-3 TMC provides marking/dropping, policing and traffic shaping for the packet based on configured traffic parameters. Q-3 TMC will en-queue the conformant packets into QMU queue. Non-conformant packets will either be discarded or marked.

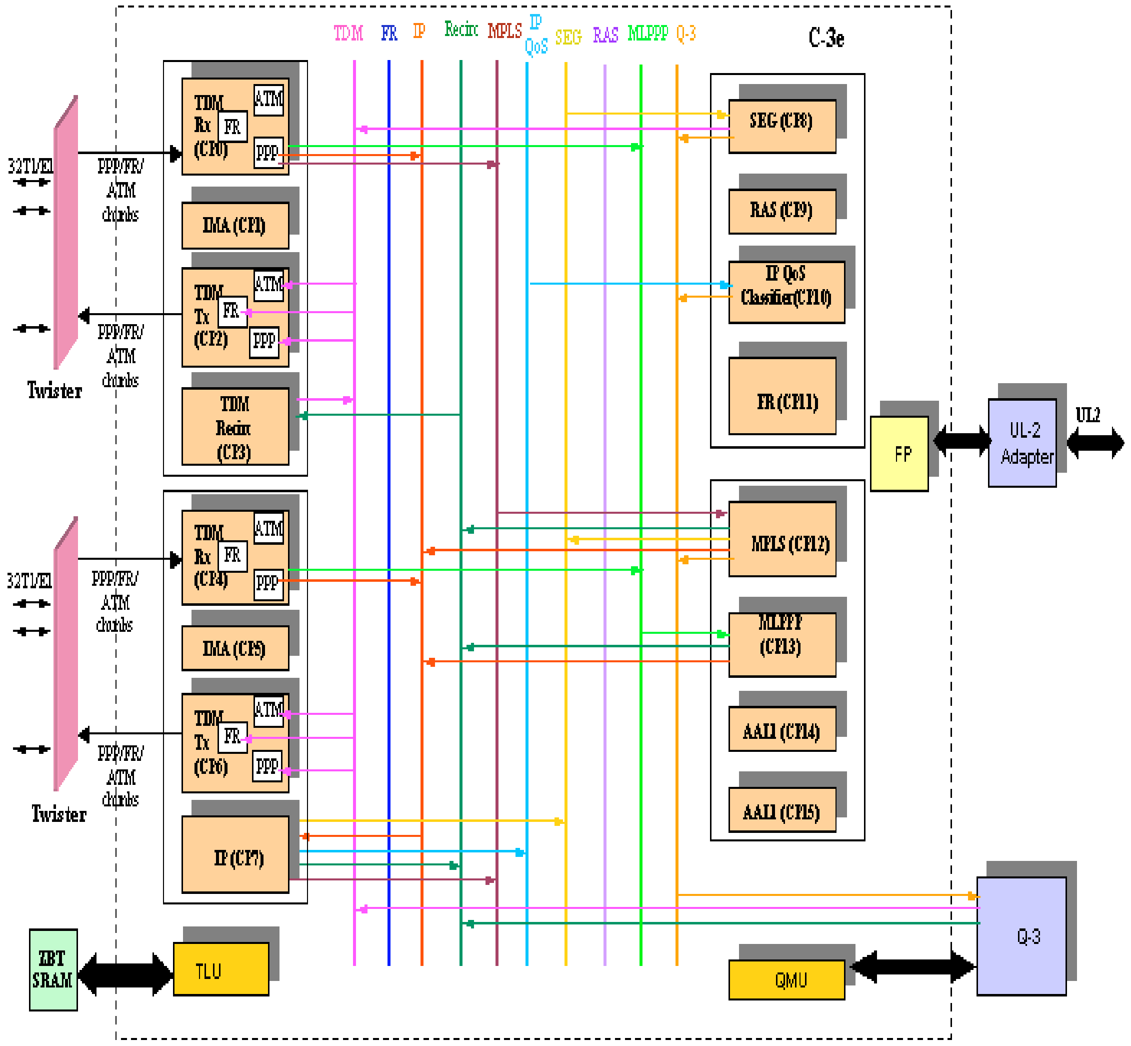

Figure 4. Data paths for PPP frames

6.1.3 Flows for ATM cells

This section describes the data flow for ATM cells received in TDM RX. The ATM cells will be recirculated in other component CPs and finally transmitted as FR / PPP / AAL5 chunks via TDM Tx. The detailed flow shown in figure 5 is described as follows.

- ATM cells are received as chunks in TDM RX. These will be enqueued to TDM Tx queue (ATM switching) or FP queue.
- AAL5 cells are enqueued into reassembly queue by TDM RX that performs the VC table lookup to send the new VPI/VCI values into the reassembly queue.
- ATM reassembly component will de-queue and reassembles the cells into AAL5 PDU. It will then be enqueued into IP queue or MPLS queue,
- IP component dequeues the reassembled AAL5 PDU from its queue, enqueues it into destination queue (TDM recirc queue or ATM Segmentation queue or MPLS queue or IP QoS queue (if QoS is needed)) determined by IP lookup and port lookup result.
- MPLS performs the label processing and then enqueues the packet to destination queue (TDM recirc queue or ATM Segmentation queue or IP queue or Q-3 traffic queue (if QoS is needed)) determined by MPLS result entry.
- ATM segmentation component will segment the IP packet into AAL5 cells, modifies the AAL5 header (with new VPI/VCI) and enqueues these cells to TDM Tx queue or to Q-3 traffic queue (if QoS is needed).
- TDM recirc component encapsulates the packet into FR or PPP frame, enqueues it to TDM Tx for final transmission over TDM channel.
- TDM Tx dequeues the FR frame or PPP frame or AAL5 cells from its queue. It transmits TDM chunks of 64 bytes. For ATM, each cell will fit into a TDM chunk. For FR or PPP, it segments the frame into TDM chunks.
- If QoS is needed, packets will be enqueued to Q-3 traffic queue from IP QoS classifier or ATM Segmentation or MPLS components for applying various QoS parameters. Q-3 TMC provides marking/dropping, policing and traffic shaping for the packet based on configured traffic parameters. Q-3 TMC will en-queue the conformant packets into QMU queue. Non-conformant packets will either be discarded or marked.

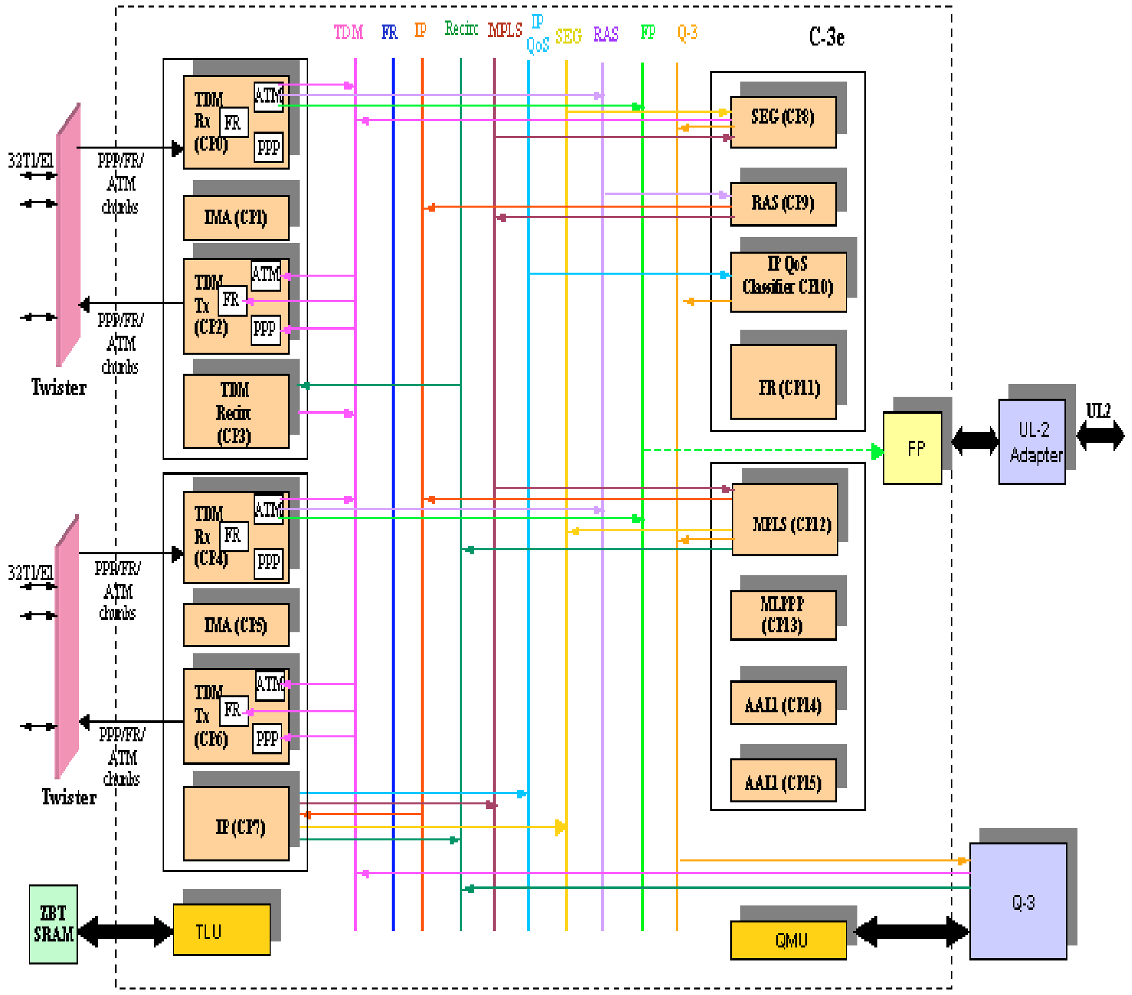

Figure 5. Data paths for ATM Cells

7 Network processor components

This section describes each of the features of the applications in detail and explains how each component or resource within the NP is used to provide the applications' features.
The Executive Processor RISC Core (XPRC) is a general-purpose processor that provides management, control, and exception processing functions. The XP controls NP boot up, configuration, and initialization of all system components.

The Channel Processors (CPs) are the components most closely associated with processing data from a physical interface. There are 16 CPs organized as four clusters, each of which contains four CPs. Each cluster performs several functions that aid in the processing of data packets.

7.1 XP

The XP program is partitioned into distinct 'initialization' and 'main' executables. After loading and running the initialization executable, the main executable is loaded and overlayed on the initialization executable, reducing the IMEM used at run-time. This partitioning scheme uses the available IMEM resource to its fullest.

7.1.1 Initialization Program

The initialization executable performs service initialization, configures system resources, and loads the CPs. In particular, the initialization executable does the following:

- Allocates buffer pools.
- Allocates and configures queues.
- Configures the fabric port.
- Configures the PHY interfaces.
- Loads the CPs.
- Defers to the main XP executable program.

Arrays of parameter values are used to initialize the buffer pools and queues. The arrays are made up of macros defined in the top-level configuration file (config.h).

7.1.2 Main Program

The main executable completes any necessary initialization and starts the CPs before entering the main loop. In particular, the main executable does the following:

- Prints the application banner including version number
- Restores the offline table data. Offline table data is used to initialize the TLU tables without host intervention for simulation purposes.
- Initializes the CRC correction table
- Starts the CPs and enables the fabric port.
- Starts some of the SDPs
- Initializes the OAM processing component.
- Initializes the host communication component
- Enters the main loop

The main loop within the XP performs processing for OAM handling described in section "OAM processing" and host communication for updating statistics.

7.2 OAM Processing

OAM cells received by the TDM CPs are forwarded to the XP for processing. OAM support in the application includes the following:

- Forward Performance Monitoring – Receive Monitoring

- o Blocks of user cells on a limited number of VCCs (128) are monitored for errors per flow. A BIP-16 is generated for all the cell payloads for each block where the block size is configurable. The block size is defaulted to 128 cells.
- o The receiver checks the parity on the received block data and compares its results with the received BIP-16. The number of errors is determined and written to a statistics counter for the indicated VC.

OAM processing uses the ATM VC table as described in section 7.18.4.

7.2.1 SDP

The TDM CPs support OAM performance monitoring. The SDP processors on the CPs do the following:

7.2.1.1 RxSync

The RxSync processor performs the following OAM functions:

- Determines the CRC-10 for each cell received (regardless of whether the cell is OAM or not) and forwards a pass-fail notification to the RxByte processor.

RxSync is not configurable through its control space.

7.2.1.2 RxByte

The RxByte processor performs the following OAM functions:

- Determines whether an F4/F5 OAM cell has been received and indicates this in extract space.
- Writes cell payload overhead to extract space.
- Forwards CRC-10 pass/fail indication to the RC through extract space.
- Determines the BIP-16 value on each cell received and writes this value to extract space.
- Determines whether a user cell has been received and writes this information to extract space.

7.2.2 RC

The RC performs higher level processing of data packets to support OAM –FPM.

7.2.2.1 Initialization

During initialization, the 128 entry OAM PM table is initialized.

7.2.2.2 Receive

The receive thread handles incoming data packets and performs OAM specific operations. Specifically, it does the following:

- Checks whether a received cell is on a VC where OAM FPM is being performed. This information is stored in the ATM VC table (the oamPm field). If this cell is a user cell, it does the following:
    - XORs the current value of the BIP-16 into OAM FPM table running total for all user cells.
    - Increments and masks the CurrentBlockValue (ranges from 0 to BlockSize-1).

- If the received cell is not a user cell, then the code checks whether an OAM cell has been received. If OAM but not of the type OAM FPM cell, the cell is forwarded to the XP. Otherwise, it does the following:
    - Compares the CurrentBip16 value with the value received in the OAM FPM Cell. If these values are XOR-ed, the number of bits set indicates the number of errors. The number of bits set is determined through a lookup into a 16-byte table (where each byte in the table indicates the number of bits set for the index) for each nibble (oamPmErrTab). The information is used to update TotalBip16Errs counter.

7.2.3 **Data Structures**

7.2.3.1 OamPmTable

This OAM processor maintains OAM performance monitoring state information in the following data structure:

Bytes	**0**	**1**	**2**	**3**
0	TotalBip16Errs		CurrentBlockValue	
4	CurrentBip16		blockSize	
8	SeqnumExpect	Pad		

- totalBip16Errs – count of BIP-16 errors calculated so far
- currentBlockValue – the number of the cell in the current block
- currentBip16 – the value of the BIP-16 calculated so far
- blockSize – the block size (in cells)
- seqNumExpect – the expected sequence number to be received
- pad – unused

7.3 Statistics Managem ent

XPRC maintains all the statistics for the MSA applications. Periodically, XP updates host with the statistics to be available to the end user. It passes the statistics storage pointer to the CP's at initialization. CPs update the statistics maintained in XPs at run time. As XPRC has only 16kB of DMEM shared across CPs, the time for XP to update the host should depend on the amount of storage, which is needed to store the statistics.

To synchronize between CPs updating XP DMEM and XP updating host, following implementation is used:

XP maintains two banks of 8KB each in its FAR DATA section (data section which is accessible to CPs). XP keeps shuffling the pointer between these two data banks periodically, so that at the time of updating the host, CPs should access and update the other bank of 8KB.

A list of TDM statistics to be maintained on a per channel basis is as follows:

- chRxChunks – Number of received chunks
- chRxPdus – Number of received PDUs
- chRxBytes – Number of received bytes
- chRxLenErrs – Number of chunks having invalid length (e.g. short chunk, long chunk)
- chRxCrcErrs – Number of chunks having invalid CRC
- chRxBip8Errs – Number of chunks having BIP8 errors
- chRxInvalidErrs – Number of chunks having other errors
- chRxFlowChunks – Number of Flow chunks
- chRxLookupErrs – Number of chunks that caused lookup failure
- chTxChunks – Number of transmitted chunks
- chTxPdus – Number of transmitted PDUs
- chTxBytes – Number of transmitted bytes

List of IP statistics:

- IpInReceives - Total number packets received in IP module
- IpInHdrErrors - The number of input datagrams discarded due to errors in their IP headers.
- IpForwDatagrams - Number of input datagrams forwarded
- IpOutPayloadErrors - Number of packets discarded due to payload errors
- IpOutInvalidPortErrors - Number of packets discarded because its route entry mapped to an invalid egress port.
- IpOutNoRoutes - Number of IP datagrams discarded because no route could be found to transmit them to their destination

The following statistics to be provided on a per VC basis (a maximum of 2048)

- AAL5 CRC Errors
- Over Sized PDUs
- AAL5InReceives

Within the 8 KB of DMEM, the following calculation of time and number of bits to be used to hold values, is implemented:
Generally, if we assign 2 bits to store value for each statistic parameter, a total of 8Kb is not sufficient to hold all the parameters. So some of the parameters may have to be removed from this list of statistics (or) the statistics have to be provided using table support. If AAL5 statistics parameters are removed, a total of 8KB is sufficient to hold all the parameters.

Calculating time for updating the host: -
Per channel, 128 chunks to be received in one second i.e., 8K of bytes per second. If 2 bits are used for chRxChunks, then we will have to update host after every 4 chunk.
That works out to be around 30 milliseconds.

7.4 TDM RX (CP 0 and CP 4)

CP0 and CP4 implement the TDM Rx components for the MSA application. The Channel Processors are directly connected to Twister Mt-21 from outside world. The number of channels to be supported is 2048 (1024 by each TDM Rx component). The channel may carry ATM/FR/PPP traffic. Chunk size support in this application is 64-bytes.

7.4.1 RxSDP

The SDP receives data from Twister Mt-21 to the RC in the receive direction and from the RC to the Twister Mt-21 in the transmit direction. The chunks that will originate from Twister Mt-21 will be either ATM or HDLC chunks. During the initialization phase itself the TDM interfaces will be configured as PPP or FR. Since it is not possible to differentiate the HDLC frames as PPP or FR in the TDM RxByte, The TDM Rx CPRC will differentiate the PPP and FR chunks based on the channel ID information filled on the extract space. The channel ID will be stored in extract space by the RxByte processor. The channel configuration is stored in DMEM, which is used for identifying the protocol running on that channel. The functions provided by each of its component processors are described below.

7.4.1.1 RxByte

The RxByte processor performs the following:

- Checks for recirculation mode.
- If it is in Non-recirculation mode, it receives the bytes from the RxSync processor. It also writes the previously dropped chunks counter in extract space.
- Initializes the SOP and DroppedChunks registers to 0.
- Reads chunk type, channel Id and channel type into extract space.
- Reads UserValid, HDLC chunk length, HDLC UserInd (SOP/non-SOP) into extract space for User chunks. (Chunk type will determine whether it is the user chunk or flow chunk).
- Reads the HDLC crcInd into extract space.
- For flow chunk,
    - Verifies flow control chunk valid bit
    - Writes flow control chunk count to extract space.
    - Indicates processing complete to the CPRC.
- For user chunk, check the channel type and determine whether it is HDLC chunk or ATM chunk.

HDLC chunks

- For HDLC chunks, check whether the chunk is SOP or non-SOP.
- For SOP chunk,
    - Write the first ten bytes after the TDM chunk header into the extract space. The reason for writing the ten bytes into extract space is both FR and PPP need ten bytes and eight bytes respectively to specify the header information (if the packet is a MPLS packet).
    - Set L1 done so that RxCPRC can start processing based on extract space values.

  - o Send remaining bytes of payload of the chunk to DMEM.
- For non-SOP TDM chunk, it sets L1 done for RxCPRC and sends remaining bytes of payload of the chunk to DMEM.
- When data9 is received, it writes the chunk status code to extract space and switches scope.

ATM chunks

- For ATM chunks, it launches lookup into ATM VPI/VCI table for user cells. Identifies and reports OAM/RM cells. Writes the cell header to extract space also.
- Writes the ATM cell payload to extract space also.
- Sets L1 done for RxCPRC and sends remaining bytes of cell payload of the chunk to DMEM.
- When data9 is received, it writes the chunk status code to extract space and switches scope.

Transparent chunks

- For transparent chunks, it writes the channel id (port) to extract space.
- Set L1 done so that RxCPRC can start processing based on extract space values.
- Sends remaining bytes of payload of the chunk to DMEM.
- When data9 is received, it writes the chunk status code to extract space and switches scope.

7.4.2 RC (CP0 and CP4)

The TDM Rx RC performs higher level processing of chunks. The functions provided by each of its components are described below:

7.4.2.1 Initialization

The TDM Rx component initializes the data structures and registers used by RC. Specifically, it does the following:

- Initializes statistics and chunk reassembly control structures
- Initializes RxSDP control space, Rx DMA control blocks and ring bus Tx registers for ATM VPI/VCI table lookups.

7.4.2.2 Chunk Processing

The Rx RC handles incoming ATM cells or HDLC frames or Transparent chunks. Specifically, it does the following:

HDLC Rx:

- Waits for L1 done so that SDP has completed the header processing and put the necessary information into extract space.
- Make the pointer (chRxCCBPtr) to TDM Rx Control block in DMEM. chRxCCBPtr will depict the TDM chunks reassembly information in DMEM.
- Processes chunk based on chunk type (flow control or user) in extract space after checking for errors.

- For User chunk, differentiate the HDLC frame as PPP or FR chunks based on the Channel ID filled in the extract space and the channel configuration information.

FR processing

- For SOM chunk,
  - Allocate new buffer for reassembling the FR chunks
  - Destination queue will be the FR processing queue.
  - Get the FR header information by properly interpreting the extract space.
  - Write the FR header into TDM Rx channel control block (chRxCCBPtr).
- Initiate the payload transfer from DMEM to SDRAM if no error is indicated in chunk. Update the buffer offset in TDM Rx channel control block (chRxCCBPtr) by incrementing it with chunk length.
- For non-SOM chunks, retrieve reassembly state information (buffer handle and buffer offset) from TDM Rx channel control block (chRxCCBPtr) and initiate the payload transfer from DMEM to SDRAM if no error is indicated in chunk. Also, update the buffer offset in TDM Rx channel control block (chRxCCBPtr).
- For EOM chunk, build the descriptor with buffer handle, buffer length and FR header (DLCI value). En-queue it to the FR processing queue.

PPP processing

- For SOM chunk,
  - Read the 4-byte PPP header from extract space and check for the PPP protocol length. The PPP protocol may be of 1-byte or 2- byte. If the least significant bit of first protocol byte is cleared, then the protocol will be of 2 bytes.
  - Set the buffer type for PPP as following based on protocol field value:
    - For value 0x0021, bufferType will be BT_IPV4
    - For value 0x0821, bufferType will be BT_MPLS
    - For value 0x003d, bufferType will be BT_MLPPP
  - Read the 4-bytes MLheader from extract space for MLPPP.
  - Allocate new buffer and determine destination queue based on buffer type. The destination queues will be IP_QUEUE, MPLS_QUEUE and MLPPP_QUEUE

  - Write bufferHandle, destination queue and MLheader into TDM Rx control block (ChRxCCBPtr). Buffer offset will be 0 for SOM.

- Initiate the payload transfer from DMEM to SDRAM if no error is indicated in chunk. Update the buffer offset in TDM Rx channel control block (chRxCCBPtr) by incrementing it by chunk length.
- For non-SOM chunks, retrieve reassembly state information (buffer handle and buffer offset) from TDM Rx channel control block (chRxCCBPtr) and initiate the payload transfer from DMEM to SDRAM if no error is indicated in chunk. Also update the buffer offset in TDM Rx channel control block (chRxCCBPtr).
- For EOM chunk, build the queue descriptor and en-queue it to the destination queue taken from TDM Rx channel control block (chRxCCBPtr). The port buffer

type will be filled with the concatenation of input channel Id and BT_HDLC in the descriptor. For MPLS the inIfType_action will be filled as MPLS_PPP.
- Gives scope back to SDP.

ATM Rx:

ATM Rx component in TDM Rx CP handles ATM cells. Specifically, it does the following:

- Waits for ATM VPI/VCI lookup to complete
- Lookup failure causes the cell to be dropped and a statistics counter is incremented.
- Allocates new buffer and initiates payload transfer from DMEM to SDRAM.
- Builds descriptor with forwarding information from lookup response
- Waits for payload transfer to complete.
- Determines whether OAM FPM is being performed on this VC. If so:
    - For user cells, read the current BIP16 value from extract space and XOR with current value. Update OAM fields.
    - For OAM FPM cells, check BIP16 and maintain count of total BIP16 errors.
- For cells other than AAL-5 or AAL-1, it launches port table lookup.
    - Waits for port lookup to complete
    - En-queues descriptor to egress queue or QoS queue indicated by port lookup
- For AAL-5 cells, with the lookup response from the ATM VC table, the decision is made if the AAL-5 PDU has to be MPLS switched.
- With last AAL-5 cell, fills the descriptor with the information required for MPLS processing. The reassembly module forwards this information to the MPLS processing module.
- For AAL-1 cells, build the descriptor with the following fields:
    - AAL1 header from extract space
    - egress port and vcIndex from the ATM VC lookup response.
- For AAL-1 cells, set the destination queue to AAL1Rx queue.
- En-queues the descriptor to the destination queue

Transparent chunk processing:

This component handles transparent TDM chunks. Specifically, it does the following

- Fetch the egress port and ATM cell header from the Rx channel control block indexed by the channel Id.
- Allocates new buffer and initiates payload transfer from DMEM to SDRAM.
- Builds descriptor with AAL1 information (egress port, egress cell header).
- Waits for payload transfer to complete.
- En-queues the descriptor to AAL1Tx queue

7.4.3 Data Structures

7.4.3.1 Extract Space

RxByte writes information about data-grams into extract space for the RC.

```
typedef volatile struct {
        int16u chnkType_chanId;
        int8u chanType;
        int8u userValid;
        int8u flowChunkCnt;
        int8u chunkStatus;
        int8u droppedChunks;

        union {
                struct {
                        CellHeader header;
                        int16u bip16;
                        int8u encodedPti;
                        int8u payload[48];
                }atm;

        struct {
                int8u userInd;
                int8u crcInd;
                int8u chunkLength;

                int32u header1;
                int32u header2;
                int16u header3;

                }hdlc;
        } proto;
} TdmExtract;
```

The explanations for the above-mentioned fields will be as follows:

- chnkType_chanId – a bitmap defined as follows:
  - b15 : chnkType – specifies the chunk type(user chunk or flow chunk)
  - b14 -11: unused.
  - b10 - 0 : chanId – specifies the input channel ID
- chanType – specifies the channel type (HDLC/ATM).
- userValid – specifies the user valid indicator.
- flowChunkCnt – specifies the flow chunk counter.
- chunkStatus – specifies the chunk status (good or bad chunk).
- droppedChunks – number of dropped chunks.
- userInd – user indicator (BOM/COM/EOM).
- crcInd – CRC indicator (for CRC16 or CRC32 calculation).
- chunkLength – specifies the chunk length.
- header1, header2, header3 – HDLC header information (could be FR or PPP)

- Cell header – cell header of the ATM AAL-5 cell.
- EncodedPti – field to identify the ATM payload type.
- atmPayload – 48 bytes of AAL-5 cells.

7.4.3.2 TDM Rx Control Block

The state maintained (in DMEM) for each HDLC frame being reassembled has the following data structure:
Note: Some of the fields of this block are replaced to maintain the egress port and ATM cell header for every transparent channel configured.

Byte Offset	**0**	**1**	**2**	**3**
0	chBufHandle			
4	chBufOffset		chDestQ	
16	chMlHeader			

```
typedef struct {
        BsBufHandle chBufHandle;
        int16u chBufOffset;
        int16u chDestQ;
        int32u chMlHeader;
} TdmRxCCB;
```

The explanations for the above-mentioned fields will be as follows:

- chBufHandle: – specifies the handle of the reassembled buffer.
- chBufOffset: – specifies the offset in the reassembled buffer.
- chDestQ: – destination queue where the EOM chunk will be en-queued. This field corresponds to 'egressPort' in the case of transparent chunks.
- chMlHeader: – ML header/FR header for PPP/FR respectively. This field corresponds to ATM 'cellHeader' in the case of transparent chunks.

7.4.3.3 Descriptor Structure

The following is the data structure of the descriptor to be en-queued.

Byte Offset	**0**	**1**	**2**	**3**
0	bufHandle			
4	Length		Port_bufType	
8	AppData			
12	AppData			

```
typedef struct {
        BsBufHandle bufHandle;
        int16u     length;
```

```
            int16u      port_bufType;
            union {
                        int8u           byte[8];
                        int16u          hword[4];
                        int32u          word[2];
                        AtmDescData     atm;
                        FrDescData      frameRe;
                        SegDescData     seg;
                        TdmDescData     tdm;
                        MlPppDescData   mlPpp;
                        MplsDescData    mpls;
                        Aal1TxDescData aal1Tx;
                        Aal1RxDescData aal1Rx;

            } appData;
} DescriptorMsg;
```

The explanations for the above-mentioned fields will be as follows:

- bufHandle – specifies the handle of the reassembled buffer.
- length – specifies the chunk length.
- port_bufType – specifies the input port and buffer type of the next module.
- appData - Application specific data (FR/PPP/ATM/Transparent TDM).

7.4.3.4 Ring Bus Slots

TDM Rx CP needs to launch lookups in following tables for various packet processing:

- ATM VC Table and Port Table.

ATM VC table lookup uses these slots:

• ATM VC request slot	0
• ATM VC response slot	0

Port table lookup uses these slots:

• Port table request slot	1
• Port table response slot	4

7.4.4 TDM statistics

```
    typedef struct {
        int16u      chRxChunks;
        int16u      chRxPdus;
        int16u      chRxBytes;
        int16u      chRxBip8Errs;     // for ATM cells
        int16u      chRxLenErrs;
        int16u      chRxInvalidErrs;
        int16u      chRxCrcErrs;
        int16u      chRxFlowChunks;
        int16u      chRxLookupErrs;      //for ATM cells
        int16u      chTxChunks;
        int16u      chTxPdus;
        int16u      chTxBytes;
    } TdmStats;
```

The description of each field is given below.

chRxChunks – Number of received chunks
chRxPdus – Number of received PDUs
chRxBytes – Number of received bytes
chRxLenErrs – Number of chunks having invalid length (e.g. short chunk, long chunk)
chRxCrcErrs – Number of chunks having invalid CRC
chRxBip8Errs – Number of chunks having BIP8 errors
chRxInvalidErrs – Number of chunks having other errors
chRxFlowChunks – Number of Flow chunks
chRxLookupErrs – Number of chunks that caused lookup failure
chTxChunks – Number of transmitted chunks
chTxPdus – Number of transmitted PDUs
chTxBytes – Number of transmitted bytes

7.5 IMA (CP 1 and CP 5)

CP 1 and CP5 implement the IMA processing. In the transmit direction, this processor handles outgoing cells from other ATM processes and sends them in a round robin fashion among several TDM links in the IMA group. It generates ICP and filler cells and maintains the link and group state machines necessary for IMA connections. In the receive direction, the IMA processor receives cells from the TDM links and performs synchronization to reconstruct the ATM cell stream. It handles ICP and filler cells and maintains the link and group state machines necessary for IMA connections. The IMA component does not use the SDP.

7.5.1 SDP

Since IMA component does not use SDP, this section is not filled in.

7.5.2 RC

The component uses three threads to perform its task, namely, one Receive thread and two Transmit threads.

7.5.2.1 IMA Receive

The IMA Receive thread processes cells received from the TDM links in the following manner:

- Waits for a descriptor to be available in the IMA RX queue then de-queues it.
- Determines the IMA group and link based on the input port from the descriptor.
- For ICP cells, does the following:
    - Initiates a DMA transfer of the cell payload from SDRAM to local DMEM.

    - If the ICP cell is a stuffed ICP cell, it is dropped.
    - Run the link state machine based on link state information from the ICP cell.
    - Run the group state machine based on group state information from the ICP cell.
    - Run the frame synchronization state machine.
    - If frame sync, put the ICP cell in the link differential delay queue as a filler cell.
- For filler and user cells, if frame synchronization has been attained and the link is active, put the cell in the link differential delay queue.
- Uses round robin to determine which link differential delay queue of the group to service. Removes a cell from the link differential delay queue and does the following:
    - If the cell is a filler cell, it is dropped.
    - If the cell is a user cell, it is en-queued to the next processing block (AAL-1, ATM TM, etc.) determined by data in the descriptor.
- Switches to the next context.
- Loops and waits for next descriptor.

7.5.2.2 IMA Transmit Input

The IMA Transmit Input thread de-queues cells and puts them in the transmit soft queues in the following manner:

- Waits for a descriptor to be available in the IMA Tx queue, then de-queues it.
- Determines the IMA group based on the output port from the descriptor.
- Puts the user cell in the transmit soft queue for the group
- Switches to the next context
- Loops and waits for the next descriptor.

7.5.2.3 IMA Transmit Output

The IMA Transmit Output thread runs at the group cell rate. On each tick of the group cell rate clock, the thread does the following:

- Determines the link within the group, which should receive the next cell. This is done in a round robin fashion among all active or usable links in the group.
- If it is time to send an ICP cell on the link, the group state ICP cell storage is updated for the current link and the cell is transferred to an SDRAM buffer via DMA. A descriptor is built and en-queued to the queue of the target TDM link.
- Otherwise, if the link is in the active state and a user cell is available in the transmit soft queue for the link, the descriptor for the user cell is removed from the soft queue and en-queued to the queue of the target TDM link.
- Otherwise, a filler cell is en-queued to the queue of the target TDM link.
- Updates link and group state for link and group on which the cell was just sent.
- Switches to the next context.
- Loops and waits for the next tick.

7.5.3 Data Structures

The IMA component uses the following data structures to maintain state and translate data.

7.5.3.1 ImaLinkState

The IMA link state structure saves state information for each of the available ATM TDM links. The structure has the following format:

Byte Offset	0	1	2	3
0	LinkId	flags	rxState	txState
4	feState	frameSync	frameOffset	rxFrameSeqNum
8	txFrameSeqNum	numIcpValid	numIcpErrors	numIcpInvalid

- linkId – the link ID number
- flags – a bitmask defined as follows:
  - b7-5: unused
  - b4: link ID valid – the received link ID is valid
  - b3: RX failure – a receive failure has occurred
  - b2: RX fault – a receive fault has occurred
  - b1: Tx fault – a transmit fault has occurred
  - b0: inhibit – the link is being inhibited
- rxState – value of the RX link state machine
- txState – value of the Tx link state machine
- feState – values of the far end link state machine and defects
- frameSync – value of the IMA frame synchronization state machine
- frameOffset – offset within the IMA frame at which the ICP cell should appear
- rxFrameSeqNum – the expected received frame sequence number
- txFrameSeqNum – the frame sequence number to be transmitted
- numIcpValid – number of consecutive frames with valid ICP cells used in frame synchronization
- numIcpErrors – number of consecutive frames with ICP error cells used in frame synchronization
- numIcpInvalid – number of consecutive frames with invalid ICP cells used in frame synchronization

7.5.3.2 ImaGroupState

The IMA group state structure saves state information for each of the IMA groups. The structure has the following format:

Byte Offset	0	1	2	3
0-11	icpCell			
12	linkIdAlloc			
16-47	linkIdToLinkMap			
48	frameLen		state	feState
52	change	rxchangenum	txChangeNum	numLinks
56	suffLinks	flags	rxOamLabel	rxImaId

The explanation of fields are as given below:

- icpCell – the ICP cell to be sent by the Tx thread
- linkIdAlloc – a bitmap indicating which link IDs are in use

- linkIdToLinkMap[32] – a mapping from link ID to link state storage structure
- frameLen – length of IMA frame (32, 64, 128, or 256)
- state – group state machine state
- feState – far end group state machine state
- change – flag indicating next ICP cell transmitted will have a change in it
- rxChangeNum – received status and control sequence number
- txChangedNum – next status and control sequence number to transmit in ICP cell
- numLinks – number of active links
- suffLinks – number of active links needed to leave insufficient links state
- flags – a bitmap as follows:
    - b7-1: unused
    - b0: inhibit – the group is being inhibited
- rxOamLabel – the received OAM label in the ICP cell
- rxImaId – the received IMA ID in the ICP cell

7.5.3.3 ImaParams

The IMA parameters structure stores information that controls behavior of the IMA unit. The host or other management agent may set these parameters. The structure has the following format:

Byte Offset	0	1	2	3
0	alpha	beta	gamma	pad

- Alpha – the number of consecutive invalid ICP cells that must be received before frame synchronization is lost
- Beta – the number of consecutive errored ICP cells that must be received before frame synchronization is lost
- Gamma – the number of consecutive valid ICP cells that must be received before frame synchronization is declared

7.5.3.4 ImaPortToGroupMap

The IMA port to group map is an array of bytes MAX_TDM_CHANNELS (1024 per CP cluster) long. The index into the array is the port number and the value of the array elements is the group to which the port belongs. If the port does not belong to any group, the value is IMA_INVALID_GROUP.

7.5.3.5 ImaPortToLinkMap

The IMA port to link map is an array of bytes MAX_TDM_CHANNELS long. The index into the array is the port number and the value of the array elements is the link state index associated with the port. This index is used to index the IMA link state structure array.

7.5.3.6 Merge space

This information need not be provided because SDPs are not going to be used in IMA CP.

7.5.3.7 Extract Space

This information need not be provided because SDPs are not going to be used in IMA CP.

7.5.3.8 Descriptor information

Not available

7.5.3.9 Ring Bus Slots

None

7.5.4 Issues/Enhancements

The IMA component is placed in two CPs. This is because of the DMEM size required for IMA component.

7.6 TDM Tx (CP 2 and CP 6)

CP2 and CP6 implement the TDM Tx components for the MSA application.

7.6.1 TxSDP

The TxSDP moves data from the RC to the Mt-21 chip. The functions provided by each of its component processors are described below.

7.6.1.1 TxByte

The TxByte processor performs the following functions as part of the TDM interface:

- Reads 4 bytes of chunk header information from merge space and transmits the chunk header.
- Sets the chunk length counter.
- If channel type is HDLC, reads the payload bytes from DMEM and transmits them.
- Transmits the padding bytes (zero) in the last until chunk length counter reaches zero
- For ATM SOM chunks, reads ATM cell header from merge space adds HEC and transmits.
- For ATM OAM cells, generates CRC-10.
- For transparent chunks, read the payload bytes from DMEM and transmits them.
- Set the merge9 with the last byte.
- Switches the scope for CPRC.

7.6.2 RC (CP2 and CP6)

The TDM Tx RC performs higher level processing of chunks. The functions provided by each of its components are described below:

7.6.2.1 Initialization

The TDM Tx component initializes the data structures and registers used by RC. Specifically, it does the following:

- Initializes statistics and chunk segmentation control structures

- Initializes TxSDP control space and Tx DMA control blocks

7.6.2.2 Chunk Transmit

The output thread handles outgoing cells or datagrams. Specifically, it does the following:

- Check channels in a round robin manner for credits available (chFlowChunksAvail).
- Get state information for current channel i.e. get the pointer chTxCCBPtr which points to TDM Tx channel control block in DMEM. This control block will contain the segmentation state information.
- If chFlowChunksAvail is true, check whether this channel is in the process of segmenting the PDU into chunks i.e. transmitting the chunks of a PDU. If it is so, the next chunk of PDU will be transmitted.
- If no segmentation is in progress, the incoming descriptor will be de-queued from its queue. Note that the descriptor would have been en-queued by
  - TDM Recirculation (CP3 - for FR or PPP encapsulation)
  - FR module (CP11 - for FR switching)
  - ATM segmentation (CP8)
  - AAL1 Tx(CP14)
  - AAL1 Rx(CP15)

- After de-queuing, the segmentation state will be updated with the values fetched from descriptor. Segmentation state values to be updated are: Buffer Handle, Buffer offset, length and port buffer type taken from incoming descriptor. Offset will be filled as zero
- PortBufferType will be checked to determine the incoming module i.e. from which module it has come (BT_ATM or BT_TRANSPARENT or others) so that it will segment the PDU accordingly.
- If portBufferType is BT_TRANSPARENT, it is a transparent chunk from AAL1 Rx CP. It fills the merge space with chanId_chanType, chunkLength.
- If portBufferType is BT_ATM, it is an ATM cell and hence will be switched. If portBufferType is other than BT_ATM, the frame needs to be segmented into number of chunks based on the channel length in chTxCCBPtr.
- For segmenting into chunks, chunk length (1 - 64 bytes) will be calculated based on offset in ChTxCCBPtr (for first chunk, the offset will be 0). Also if chunk length is less than 64 bytes, it will set the EOM flag stating that it is the last chunk.
- The offset in chTxCCBPtr will be incremented by chunk length for the next chunk.
- It waits for the scope to be available from SDP.
- It fills the merge space with chanId_chanType, chunkLength and userInd.
- For ATM, fills cell header also into the merge space.
- It then waits for payload transfer of previous chunk to complete.
- It initiates the payload transfer from SDRAM to DMEM for that chunk.
- For EOM chunk, it frees the buffer associated with previous chunk and resets the state information (in chTxCCBPtr) for the new PDU.

7.6.3 Data Structures

7.6.3.1 Merge space

The RC writes information about datagrams into merge space. The data structure of merge header looks like following.

```
typedef volatile struct {

        int16u chanId_chanType;
        int8u userChunkLength;
        int8u userInd;
        union {
                struct {
                        CellHeader atmHeader;
                        int8u atmPayload[48];
                        }atm;

                struct {
                        int8u pad1;
                        }hdlc;

                struct {
                        int32u header;
                        int8u headerLen;
                        }pppRecirc;
                }proto;
        int8u pad[8];
} TdmMerge;
```

The explanations for the above-mentioned fields will be as follows:

- chanId_chanType – a bitmap defined as follows:
  - b15 : unused.
  - b14 - 4 : chanId – specifies the input channel ID
  - b3-1 : chanType – specifies the channel Type (ATM/HDLC/Transparent)
  - b0 : unused.

- userChunkLength – chunk length (1-64 bytes)
- userInd – specifies the user chunk indicator (SOM/COM/EOM)
- atmHeader – ATM Cell header
- atmPayload – 48 bytes of ATM cells.

7.6.3.2 TDM Tx Control Block

The state maintained (in DMEM) for segmentation on each outgoing channel is stored in the following data structure:

Byte Offset	0	1	2	3

0	chBufHandle		
4	chBufOffset		chLength
8	ChFlowChunksAvail	Pad	

```
typedef struct {
        BsBufHandle chBufHandle;
        int16u chBufOffset;
        int16u chLength;
        int8u chFlowChunksAvail;
} TdmTxCCB;
```

The explanations for the above-mentioned fields will be as follows:

- ChBufHandle : specifies the Buffer handle that has to be transmitted.
- ChBufOffset : specifies the offset of the chunk in the buffer.
- chLength: specifies the chunk length.
- ChFlowChunksAvail : specifies the counts of credits available to each channel.
- Pad: unused.

7.6.3.3 Descriptor Structure

The descriptor structure for TDM Tx is same as for TDM Rx. The structure is defined in section 7.4.3.3

7.6.4 TDM Statistics

The statistics structure for TDM Tx is same as for TDM Rx. The structure is defined in section 7.4.4

7.7 TDM Recirculation (CP3)

CP3 is used to recirculate PPP and FR frames destined for the TDM Tx (CP2/CP6) for transmission. Its purpose is to add the PPP header or FR header in the descriptor to the PPP/FR frame before transmission.

7.7.1 SDP

The SDP adds the PPP or FR header to the frame. The functions provided by each of its component processors are described below. Only RxByte and TxByte are used for this purpose.

7.7.1.1 TxByte

The TxByte processor performs the following functions:

- Reads channel Id from merge space and transmits as first byte of the chunk.
- Read the Buftype from merge space. It will also differentiate between PPP and FR. If Buftype is not BT_IP_FR or BT_MPLS_FR, it will be the PPP frame.

- For PPP,
  - Reads the PPP header length from merge space, then reads that many bytes from the header field in merge space and transmits these.
- For FR,
  - Reads the first two bytes of FR header (that contains DLCI values) from merge space and transmits these.
  - For bufferType BT_IP_FR, transmits out the control byte (0x00) and NLPID (0xcc) byte.
  - For bufferType BT_MPLS_FR, construct the LLC SNAP header encapsulation bytes and transmit these bytes.
- Reads remaining payload from DMEM until data9 is observed.
- Switches scope.

7.7.1.2 RxByte

The RxByte processor performs the following functions:

- Receives bytes from the TxByte processor.
- The first byte of the chunk contains the channel Id. This is written to extract space and header ready is indicated in the rxStatus register.
- All bytes are written to DMEM.
- When data9 is received, switches scope.

7.7.2 RC

The RC manages the TDM recirculation. All CPs that want to transmit PPP/FR frames, en-queue their packets to the TDM recirculation CP3 so that the PPP/FR header is added to the frame before transmission.

7.7.2.1 Initialization

The initialization component initializes the data structures and registers used by the RC. Specifically, it does the following:

- Initializes statistics and chunk control structures.
- Initialize SDP control space, Tx/Rx DMA control blocks. Indicate TDM RxSDP to run the recirculation portion of the code.

7.7.2.2 RxCPRC

The receive component handles incoming packets from the SDP. Specifically, it does the following:

- Waits for indication of header processing completion.
- Creates descriptor with buffer handle and length.
- En-queues packets based upon the channel Id indicated in extract space.
- Allocates buffer, sets up DMA block and gives scope back to SDP.

7.7.2.3 TxCPRC

The transmit component services its TDM Recirculation egress queue (all PPP or FR traffic which needs a header inserted in the frame gets en-queued here). This CP then passes chunks to the SDP. Specifically, it does the following:

- De-queues the packet descriptor from its TDM Recirculation egress queue. This descriptor would have been en-queued by IP or MPLS channel processors.
- Waits for scope to be available from SDP.
- Gets channel Id (input port) and buffer type from the descriptor. Buffer type may be one of the following.
    - BT_IP_FR: if the FR frame originates from IP module
    - BT_MPLS_FR: if the FR frame originates from MPLS module
    - Otherwise it is the PPP frame originating from IP/MPLS module
- Determines protocol type based on buffer type.

PPP processing:

For the buffer type BT_IP_PPP and BT_MPLS_PPP,

- Determines the 4-byte PPP header including protocol, and calculates the header length.
- Fills the channel Id, BufType, PPP header and header length into the merge space.

FR processing:

For the buffer type BT_IP_FR and BT_MPLS_FR

- Fill the channel Id, BufType and FR header information (2-bytes containing DLCI) from descriptor into the merge space.
- Sets up DMA engine with buffer handle from descriptor.
- Initiates payload transfer from SDRAM to DMEM and gives scope to the SDP.
- Frees buffer from previous transmit in this scope.

7.7.3 Data Structures

7.7.3.1 Merge space

The merge space structure is defined below.

For FR

Byte Offset	0	1	2	3
0	chanId_chanType		userChunkLength	userInd
4	BufType	Fr header		
12	Fr header	Pad		
16	Pad			
20	Pad			

The explanations for the above-mentioned fields will be as follows

- chanId_chanType – a bitmap defined as follows:

  - b15 : unused.
  - b14 - 4 : chanId – specifies the input channel ID
  - b3-1 : chanType – specifies the channel Type (ATM/HDLC/Transparent)
  - b0 : unused.

- UserChunkLength: specifies the length of the user chunk.
- UserInd: specifies the user chunk indicator (BOM/COM/EOM)
- BufType: specifies the type of the buffer.
- FR header: specifies the DLCI value and the congestion control information.
- Pad: unused

For PPP

Byte Offset	0	1	2	3
0	chanId_chanType		userChunkLength	userInd
4	BufType	PPP header		
12	PPP header	PPPheaderLen	Pad	
16	Pad			
20	Pad			

The explanations for the above-mentioned fields will be as follows

- chanId_chanType – a bitmap defined as follows:
  - b15 : unused.
  - b14 - 4 : chanId – specifies the input channel ID
  - b3-1 : chanType – specifies the channel Type (ATM/HDLC/Transparent)
  - b0 : unused.

- UserChunkLength: specifies the length of the user chunk.
- UserInd: specifies the user chunk indicator (BOM/COM/EOM)
- BufType: specifies the type of the buffer.
- PPP header: specifies the PPP header information.
- PPP headerLen: specifies the length of the PPP header.
- Pad: unused

7.7.3.2 Extract space

The extract space structure is defined below.

Byte Offset	0	1	2	3
0	chanId		Pad	

The explanations for the above-mentioned fields will be as follows
- ChanId: specifies the channel ID.
- Pad: unused.

7.7.3.3 Descriptor

The descriptor structure is same as for TDM Rx. The structure is defined in section 7.4.3.3.

7.8 IPv4 (CP 7)

CP 7 implements the IPv4 (Layer 3 forwarding) component for the MSA application.

IP routing is the process of forwarding IP frames at layer 3 based upon the IP Destination Address (IP DA). An advantage of IP routing is that it can be used between dissimilar network media types. This application covers IP routing over ATM or Frame Relay or PPP/MLPPP.
Assumptions and notes for use:

- IP header options will not be recognized.
- IP Fragmentation and reassembly not supported
- Lookup is launched on the IP DA.
- Application generates two types of ICMP messages, which are based on the events that happen in the data path:
  - ICMP Time exceeded
  - ICMP destination unreachable

The IP address is provided to the XP via the appData parameters in the shared HCA (Host Communication Area) structure. The XP passes this address to the CP in the initialization descriptor. The CP uses this as the IP source address for all NP generated ICMP messages.

7.8.1 SDP

The SDP is configured for byte level re-circulation. The SDP re-circulates the IP packet to remove HDLC/PPP/FR encapsulation if present and validates the IP header. It also launches the IP destination address lookup to retrieve forwarding parameters for the datagram.

7.8.1.1 TxByte

The TxByte processor performs the following functions:

- Receives IP datagram from BMU through DMEM.
- Based on the Buffer Type, the following operations are done:
  - BT_HDLC/BT_PPP/BT_FR: strips the respective headers and does IP parsing
  - BT_IP: Parses the IP header
- Sends control information about the datagram to RxByte processor, including the buffer handle, buffer type, and input port.
- Sends the IP datagram to RxByte processor.
- Sends an end of packet byte which indicates errors if non-zero.
- Switches scope and waits for more data to be available in DMEM.

The RC writes information about datagrams needing re-circulation into merge space for TxByte processor to use in its processing. The data structure is defined in section 7.8.3.1. TxByte processor is not configurable through control space.

7.8.1.2 RxByte

The RxByte processor performs the following functions:

- Waits for a receive scope to become available.
- Receives control information from TxByte processor and places it in extract space.
- Validates the IP header including version and header length and IP checksum and TTL field.
- If the header is valid, launches a lookup of the IP destination address
- If the header is not valid, places an error code in the header status field of extract space and does not launch a lookup.
- If no error, TTL verification and TTL decrement operations are done and checksum modification is done and updated accordingly
- Streams the remaining payload to DMEM and writes the payload status field of extract space.
- Switches scope and waits for another to become available.

The RxByte processor writes information about recirculated datagrams into extract space for the RC to use in its processing. The data structure is defined in section 7.8.3.2. RxByte is not configurable through control space.

7.8.2 RC

The IP forwarding component uses two threads to perform its task, namely, an input thread and an output thread. The initialization code starts the two threads. Each of these is described next.

7.8.2.1 Initialization

The IP component does the following things during its initialization:

- Creates the input and output threads.
- Initializes the IP route lookup launched by the SDP.
- Initializes both scopes by giving the SDP ownership.
- Starts the SDP in byte loop back mode.
- Jumps to the first thread.

7.8.2.2 Input Thread

The input thread handles incoming datagrams. Specifically, it does the following:

- Monitors its queue.
- De-queues an IP descriptor.
- Increments a statistics counter
- Waits for a transmit scope to be available from the SDP.
- Fills in merge space with data from the descriptor including buffer handle and input port.
- Waits for the previous DMA transfer to complete.
- Begins the DMA transfer for the current buffer being processed.
- Switch context to the next thread.

- Loops to the beginning to wait for another descriptor.

7.8.2.3 Output Thread

The output thread handles outgoing datagrams. Specifically, it does the following:

- Waits for a scope to become available from the SDP.
- Begins the DMA transfer from DMEM to the SDRAM buffer indicated in extract space.
- Checks for header errors, and if one has occurred, drops the packet and increments ipInHdrErrors counter. If the error is because of TTL expiry, ICMP time expired message is sent to the source.
- Waits for the IP route lookup to complete and if the lookup fails, drops the packet and increments ipOutNoRoutes counter. Send ICMP destination unreachable message to the source.
- If the lookup response indicates that the packet is to be MPLS switched, then
    - Fills in the information from the IP route lookup
    - Destination queue = MPLS_QUEUE

  Else
    - Launches a lookup of the port indicated in the IP route lookup response.
    - Waits for the port lookup to complete, and if the port is invalid, drops the packets and increments ipOutInvalidPortError counter.
    - Fills in a descriptor using information from the IP route and port lookups.
- Waits for the payload transfer to complete.
- Check for any payload error, if so drops the packet and increments ipOutPayloadError counter.
- Sends the descriptor to the appropriate destination determined by the lookups. If QoS enabled then sends the descriptor to QoS queue based on the port table entry.
- Increments ipForwDatagrams counter
- Switches context to the next thread.
- Loops to the beginning to wait for another scope to be available.

7.8.3 Data Structures

7.8.3.1 Merge space

The RC writes information about datagrams needing recirculation into merge space for TxByte processor to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	bufHandle			
4	port_bufType		pad	

- bufHandle – handle of the buffer being recirculated
- port_bufType – a bitmask defined as follows:
    - b15-5: port – the input port on which this datagram was received

- b4-0: bufType – the type of buffer being recirculated (could be one of BT_IPv4, BT_FR ,BT_PPP)
- pad – unused

7.8.3.2 Extract Space

RxByte writes information about re-circulated datagrams into extract space for the RC to use in its processing. Entire IP header is moved into the extract space. The first 8 bytes of the data structure has the following format:

Byte Offset	**0**	**1**	**2**	**3**
0	bufHandle			
4	port_bufType		headerError	payloadError

- bufHandle – handle of the buffer being recirculated
- port_bufType – a bitmask defined as follows:
  - b15-5: port – the input port on which this datagram was received
  - b4-0: bufType – the type of buffer being recirculated (could be BT_IPv4)

The format of the next part of extract space

Byte Offset	**0**	**1**	**2**	**3**
8	vers_hlen	tos	len	
12	id		flags_fragOffset	
16	ttl	protocol	cks	
20	srcaddr			
24	destaddr			
28-44	pad			

- vers_hlen – header version and length
- tos – type of service
- len – IP total length
- id – identification field
- frags_fragOffset – fragmentation flags and offset
- ttl – time to live
- protocol – IP protocol
- cks – IP header checksum
- srcaddr – IP source address
- destaddr – IP destination address
- Pad – unused

7.8.3.3 Queue Descriptor information

One queue is allocated for IP module to communicate with other CPs and XPRC. The Queue descriptor structure is as follows:

Byte Offset	**0**	**1**	**2**	**3**

0	BufHandle	
4	Length	Port_bufType
8	appData	
12	appData	

The explanation of each field is as follows:

- bufHandle – handle to the buffer this descriptor describes
- length – length of data in the buffer
- port_bufType – a bit field structure as follows:
- b15-5: port – ingress port or egress port depending on descriptor type
- b4-0: bufType – the type of buffer
- appData – application specific data as defined below:

  Application specific data, it is a union of
  - int8u byte[8];
  - int16u hword[4];
  - int32u word[2];
  - AtmDescData atm;
  - FrDescData frameRe;
  - SegDescData seg;
  - TdmDescData tdm;
  - MlPppDescData mlPpp;
  - MplsDescData mpls

The appData field can have different interpretations depending on the outgoing interface type.

The ATM appData field has the following format:

Byte Offset	**0**	**1**	**2**	**3**
0	cellHeader			
4	VcIndex		egressQueue	

The explanation of each field is as follows:

- cellHeader – the cell header (VPI/VCI) to apply at the egress
- vcIndex – the index associated with egress VPI/VCI
- egressQueue – the egress queue, necessary as the cell passes through several processing blocks

The TDM appData field has the following format:

Byte Offset	**0**	**1**	**2**	**3**
0	McClass	Flags	Egress Queue	

The explanation of each field is as follows:

- mcClass – MC class for ML-PPP's use
- egressQueue – the egress queue, necessary as the packet passes through several processing blocks

The MPLS appData field has the following format:

Byte Offset	0	1	2	3
0	LabelSwap		LabelPush	
4	HopCount	InIfType_action	EgressQueue	
8	AppHdrData			
12	Vc Index		Pad	

The explanation of each field is as follows:

- LabelSwap - Label to be swapped
- LabelPush – Label to be added
- InIfType_action- MPLS action to be performed
- HopCount - count to be decremented in TTL of the shim header
- Egress queue – Queue assigned to the egress interface
- AppHdrData – ATM Cell header in case of egress ATM interface and FR header if it is a FR interface.
- VcIndex – ATM VC index of the ATM interface if its an AAL-5
- Pad - unused

The FrameRelay appData field has the following format:

Byte Offset	0	1	2	3
0	FrHeader			
4	EgressQueue		pad	

The explanation of each field is as follows:

- FrHeader – DLCI value
- EgressQueue – Final Queue
- Pad – unused

7.8.3.4 Counters

IP module has these counters for statistics purpose; it's stored in XPRC's shared DMEM.

S.No	Counter	Purpose
1.	ipInReceives	Total number packets received in IP module
2.	IpInHdrErrors	The number of input datagrams discarded due to errors in their IP headers.
3.	IpForwDatagrams	Number of input datagrams forwarded
4.	IpOutPayloadError	Number packets discarded due to payload errors
5.	IpOutInvalidPortError	Number of packets discarded because it route entry mapped to egress port which is invalid (or temporarily made as inaccessible)
6.	IpOutNoRoutes	Number of IP datagrams discarded because no route could be found to transmit them to their destination

7.8.3.5 Ring Bus Slots

IPv4 uses these slots:

- IPv4 route request slot 0

• IPv4 route response	0
• Port request slot	2
• Port response slot	2

7.9 Segmentation (CP 8)

CP 8 implements AAL5 segmentation module. This CP gets message descriptors with bufType set to IPv4/MPLS.

7.9.1 SDP

The SDP is configured for byte level re-circulation. It streams the IP packet and accumulates CRC. For AAL5 cells, it adds the necessary pad bytes to make the SDU length a multiple of 48 and creates the trailer. It then delivers fixed size chunks of the SDU to the CPRC. On the first chunk for an SDU, the SDP launches a port table lookup, whose results are used by the CPRC to determine the destination queue. The SDP does not interleave segmentations – it completely segments one SDU and delivers it to the CPRC before proceeding to the next SDU. The SDP accumulates payload CRC for all IP packets.

7.9.1.1 TxByte

The TxByte processor performs the following operations on every packet:

- Reads the segType, pduSize and UUI from the merge space. Sends them to RxByte processor.
- Read all the merge space fields following it. Send them to RxByte processor.
- Initialize a counter with the negative of payload size.
- Start sending the payload bytes, incrementing the counter for each byte sent out. Accumulate CRC for each transmitted byte. Stop when the counter hits 0xff.
- Initialize a counter with the number of pad bytes.
- Start transmitting zeroes, incrementing this counter for every zero byte transmitted. Accumulate the CRC on each zero byte. Stop when the counter hits 0xff.
- The trailer should be sent for AAL-5 SDU. Send the UUI, CPI and payload length, accumulating the CRC on all of them. Now send the four bytes of the CRC. This completes the AAL5 trailer.
- Release the transmit scope and wait for data available from the RC.

TxByte gets the payload information from merge space as defined in section 7.9.3.1. TxByte is not configurable through control space.

7.9.1.2 RxByte

The RxByte processor performs the following sequence of operations:

- Receive the segType, pduSize and uui from the TxByte and copy them to the extract space.
- Clear the lastCell flag.
- Receive the atmEgressQueue and destQueue from the TxByte and write them to the extract space.
- Receive the egress port from the TxByte and launch a port table lookup.
- Initialize a counter Counter1 with the negative of pduSize(0xc1).

- Stream out payload bytes to the Rx stage area. Increment Counter1 for every byte transmitted. When Counter1 hits 0xff, hand the current scope over to the CPRC. If the data-9 bit is seen at any time in the payload, go to the next step. Wait for the next scope to become available and go back to the previous step.
- At the end of the payload, data the TxByte sets the Data-9 bit. When this bit is seen, set the last cell flag to true. Copy the current value of Counter1 to the numBytesLastPdu field and hand the current scope over to the CPRC.
- The packet segmentation is complete at this point, wait for more data to be available from TxByte

The RxByte processor writes information about incoming cells into extract space for the RC to use in its processing. The data structure is described in section 7.9.3.2. RxByte is not configurable through control space

7.9.2 RC

The segmentation RC component uses two threads to perform its task, namely, an input thread and an output thread. The initialization code starts the two threads. Each of these is described next.

7.9.2.1 Initialization

The initialization phase in the segmentation CP does the following:

- Initializes buffer pools.
- Creates contexts for the input and output threads
- Initializes ring bus Tx message registers used by RxByte for launching port lookup.
- Setup DMA engines and initializes the SDP scopes.
- Enables the SDPs

7.9.2.2 Input Thread

The input thread handles incoming datagrams. Specifically, it does the following:

- De-queue message descriptors from the input queue
- Reads the length of the IP packet to be segmented.
- Determines the pad size
- Calculate the number of cells that would be generated for this SDU.
- Calculates the necessary pad bytes that should be added at the end of the packet to make its size a multiple of 48 bytes.
- Allocates a Tx scope
- Writes the IP packet buffer handle, egress port and egress queue from the IP packet descriptor and the pad length to the merge space. Free the Tx scope, thus starting the SDP processing on this packet.

7.9.2.3 Output Thread

The output thread handles outgoing datagrams. This logical function gets ATM cells from the SDP and en-queues them to the appropriate destination queue. Specifically, it carries out the following sequence of operations:

- Allocate an Rx scope and read the extract space.

- Check the Rx scope to see if the 'new_sdu' flag is set. If set, then the SDP would have launched a port table lookup. Read the egress queue from the extract space.
- Wait for the port table lookup results.
  - If QoS is enabled for the port, the destination queue is the QoS queue from the port table lookup.
  - If QoS is not enabled the destination queue is same as the egress queue from the table lookup.
- Create a message descriptor with bufType set to BT_ATM. Set the buffer handle to that of the cell delivered by the SDP and en-queue it to the destination queue.

7.9.3 Data Structures

7.9.3.1 Merge space

The RC writes information about outgoing packets to merge space for TxByte to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	SegType	PduSize	Uui	Cpi
4	PayloadLength		PadLength	pad
8	CellHeader			
12	EgressPort		AtmEgressQueue	
16	DestQueue		Pad	

```
typedef struct {
        int8u   segType;
        int8u   pduSize;
        int8u   uui;
        int8u   cpi;
        int16u  payloadLength;
        int8u   padLength;
        int8u   pad;
        CellHeader  cellHdr;
        int16u  egressPort;
        int16u  atmEgressQueue;
        int16u  destQueue;
} SegMergeSpace;
```

- segType – indicates whether the packet is to be segmented for AAL5 (SEG_AAL5)
- pduSize – specifies the size of chunks into which the packet is to be segmented (always 48 for AAL5)
- uui – specifies the user-to-user information
- cpi – reserved, set to zero
- payloadLength – size of the IP packet
- padLength – number of zero bytes that need to be added to the payload
- pad – Reserved

- cellHdr – specifies the egress cell header to be used for AAL5
- egressPort – the destination port number
- atmEgressQueue – specifies the final queue that should be used to reach the ATM port
- destQueue – specifies the immediate destination queue from the segmentation CP

7.9.3.2 Extract Space

RxByte writes information about re-circulated datagrams into extract space for the RC to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	SegType	PduSize	Uui	pad
4	atmCellHdr			
8	egressPort		AtmEgressQueue	
12	destQueue		LastCell	NumBytesLastPdu

```
typedef struct
{
            int8u   segType;
            int8u   pduSize;
            int8u   uui;
            int8u   pad;
            int32u  atmCellHdr;
            int16u  egressPort;
            int16u  atmEgressQueue;
            int16u  destQueue;
            int8u   lastCell;
            int8u  numBytesLastPdu;
} SegExtractSpace;
```

- segType – indicates whether the delivered chunk is an ATM AAL-5 cell payload
- pduSize – specifies the size of the ATM cell (always 48 for ATM AAL-5 cells)
- pad –zero for ATM cells
- uui – specifies the user-to-user information that arrived with the packet being segmented
- atmCellheader – specifies the egress cell header to be used for AAL-5.
- EgressPort – the destination port number.
- atmEgressQueue – final queue number to be used to reach the egress port
- destQueue – specifies the queue number on which the cell packet should be transmitted
- lastCell – specifies the end of an SDU
- numBytesLast – the size of the last ATM PDU

7.9.3.3 Descriptor information

The following is the data structure of the descriptor:

Byte Offset	0	1	2	3

0	bufHandle	
4	Length	port_bufType
8	appData	

```
typedef struct {
        BsBufHandle bufHandle;
        int16u     length;
        int16u     port_bufType;     /* 15:5 - port, 4:0 - bufType */
        union {
                int8u          byte[8];
                int16u         hword[4];
                int32u         word[2];
                AtmDescData    atm;
                FrDescData     frameRe;
                SegDescData    seg;
                TdmDescData    tdm;
                MlPppDescData  mlPpp;
                MplsDescData   mpls;
        } appData;
} DescriptorMsg;
```

- bufHandle – reassembly buffer handle
- length – chunk length
- port_bufType – input port and buffer type of the next module
- appData - Application specific data.

The AppData in the segmentation CP is of following format:

```
typedef struct SegDescData_s {
   CellHeader  cellHeader;
   int8u      flags_cpsPduLen; /* 7:6 flags; 5:0 cpsPduLen */
   int8u      pad;
   int16u     egressQueue;
} SegDescData;
```

Byte Offset	0	1	2	3
0	CellHeader			
4	Flags_cpsPduLen	pad	egressQueue	

7.9.3.4 Ring Bus Slots

The segmentation CP launches a lookup into Port Table. It uses following slots:

- Request Slot: 0
- Response Slot: 0

7.9.4 Issues/Enhancements

None

7.10 Reassembly (CP 9)

AAL5 re-assembly module is implemented in CP 9. An incoming message descriptor can have different buffer types distinguishing between various packet types. A buffer type of BT_ATM indicates that it is an ATM cell descriptor and needs to be processed for AAL5 reassembly, while buffer types of BT_MPLS_FR, BT_MPLS_PPP and BT_MPLS_ATM indicate that this is the last ATM cell of the MPLS packet and the reassembled MPLS packet and the information contained in the descriptor has to be passed to the MPLS processing.

7.10.1 SDP

The SDP for reassembly is configured for byte level loop back. It performs payload CRC and initiates the DMA transaction to append the current cell packet at the end of the SDU. On the non-last cell packet, the SDP initiates a CRC table update using the XOR command in the non-last mode. The TLU CRC table is indexed by vcIndex. On the last cell for a SDU, the XOR command is used CRC Rx last mode to verify the accumulated CRC. The results come back on the ring bus and are examined by the CPRC. Specifically, functions performed by each of the components of the SDP are described below:

7.10.1.1 TxByte

The TxByte processor performs the following functions:

- Wait for Tx scope from the CPRC
- Initialize the CRC accumulator.
- Forward 12 bytes that may contain MPLS related information if the SDU is an MPLS packet. The RxByte of this CP takes care to collect 12 bytes prior to collecting any other bytes.
- Forward the rasType, eom_offset, vcCidIndex, numalignedBytes and numUnalignedByteCount to RxByte. The rasType indicates whether it is AAL5 or AAL5_MPLS. The VcCidIndex holds vcIndex value for AAL5. The eom field holds the eom flag indicating whether this is the last cell of the packet. NumUnalignedBytes will always be zero for AAL5.
- Initialize a counter Counter1 (with 0xc1) to the value of numAlignedBytes. For AAL5, this parameter is always 48. Initialize another counter Counter2 (with value 0xc2) numBytesPartialPayload.
- Start sending partial payload bytes from merge space, incrementing the Counter2 for every byte. When this counter hits 0xff, all unaligned bytes would have been sent. Accumulate the partial CRC every byte transmitted.
- Now start sending the payload bytes, incrementing Counter1 for every byte and accumulate the CRC. Stop accumulating CRC when the counter hits 0xff.
- Send the remaining payload bytes. These are the unaligned bytes and will be recirculated for the next packet. Hence CRC should not be computed on these bytes.
- After sending all payload bytes, the partial CRC accumulated thus far is sent.
- At this point TxByte is done with processing the cell. It gives up the scope to the CPRC and waits for the next cell/packet.

The RC writes information about cells to be reassembled into merge space for TxByte to use in its processing. The data structure is described in section 7.10.3.1. The TxByte processor is not configurable through control space.

7.10.1.2 RxByte

The RxByte processor performs the following functions:

- Wait for extract scope from CPRC.
- Streams in 12 bytes, which may contain information relevant to MPLS packet processing
- Stream the rasType, eom, vcCidIndex, pduLength and numUnalignedByteCount to the extract space. pduLength is the aligned byte count received from the TxByte.
- Make the scope available for CPRC, by setting L1_DONE flag.
- Start streaming payload into the Rx staging area. The CPRC will setup the DMA to transfer it into SDRAM to the correct offset. Use a counter to determine when the payload ends.
- Copy the last six bytes of the payload into the extract space structures for uui, cpi, length and CRC. In case this is the last cell, the CPRC will need this information for forwarding this packet to the IP CPRC. This information should also go to the staging area.
- After the payload ends, the next six bytes are the accumulated CRC. Initiate a TLU XOR command using this CRC.
- Mark the scope status flags as L2_DONE, thus giving the trailer to the CPRC.
- Wait for another scope to be available from the CPRC.

RxByte writes information about PDUs being reassembled into extract space for the RC to use in its processing. The data structure is described in section 7.10.3.2. The RxByte processor is not configurable through control space.

7.10.2 RC

The reassembly RC component uses two threads to perform its task, namely, an input thread and an output thread. The initialization code starts the two threads. Each of these is described next.

7.10.2.1 Initialization

The initialization phase in the segmentation CP does the following:

- Initializes buffer pools.
- Creates contexts for the input and output threads
- Initializes ring bus Tx message registers used by RxByte for RAS CRC Table.
- Setup DMA engines and initializes the SDP scopes.
- Enables the SDPs

7.10.2.2 Input Thread

The input thread handles incoming datagrams. Specifically, it does the following:

- De-queues input ATM cell descriptors

- Checks the buffer type of the incoming AAL-5 SDU. The Buffer Type can hold following values:
  - BT_ATM: - It indicates the packet is an AAL-5 PDU. It fills the merge space with information required by RxByte to reassemble the ATM cells at output thread.

  - BT_MPLS_ATM: - It indicates the AAL-5 PDU is to be MPLS switched and the egress interface in an ATM interface.

  - BT_MPLS_PPP: - It indicates the AAL-5 PDU is to be MPLS switched and the egress interface in a PPP interface.

  - BT_MPLS_FR: - It indicates the AAL-5 PDU is to be MPLS switched and the egress interface in an FR interface.

- For AAL-5 SDUs which are to be MPLS switched, there is an additional 12 bytes of information which needs to be sent across the MPLS processing module after the cells are reassembled. These 12 bytes of information is as follows:
  - LabelSwap (2-Bytes)
  - LabelPush (2-Bytes)
  - Egress Port buffer type (2-Bytes)
  - Hop count and MPLS action (2-Bytes)
  - Egress cell header if the egress interface is an ATM interface (4-Bytes).These bytes are streamed out by TxSDP before ant other information.
- Starts the DMA of the packet and releases the scope for merge space.

7.10.2.3 Output Thread

The output thread handles the outgoing reassembled AAL-5 SDUs. Specifically, it performs following activities:

- Maintains a rasList structure to track re-assembly state per VC. An array of 1024 RasList structures is maintained in DMEM. This array is indexed by the vcIndex from the ATM VC table for the input cell's VPI and VCI. The vcIndex for AAL5 VCCs is hence restricted to the range [0, 1023]. Each element of this array is initialized with a valid buffer handle and offset set to zero.
- After it receives a cell for AAL-5 reassembly, it locates its state entry in the rasList array, using the cell's vcIndex. The buffer handle in this specifies the SDRAM buffer at the end of which this cell needs to be appended. The offset specifies the length of AAL-5 SDU reassembled so far. This new cell needs to be written at the 'offset' location within the SDRAM buffer.

- Reads the extract space and updates the offset in the corresponding rasList array entry.
- If the EOM Flag is set in the extract space, the current SDU has completed. When this flag is set, the output thread performs the following operations:

  - Reads the rasType if it indicates an MPLS switched SDU.

- If the RAS type is RAS_MPLS_ATM, then it fills in the descriptor the 12-bytes of information for MPLS processing module.

- Wait for CRC results from TLU: - The SDP accumulates CRC for each SDU in a TLU table, using the XOR command in CRC mode. On the last cell, the SDP issues the XOR command with CRC Rx last option. Upon getting this command, the TLU first accumulates the CRC in the command and then checks to see if the accumulated CRC indicates CRC success. It returns a ring bus success response on CRC success and a ring bus error otherwise.

- If the response is a success, an IP message descriptor is created for the SDU. It then waits for the SDP to deliver the aal5 trailer. The trailer contains SDU length, the aal5 UUI and the CRC. It fills the length field in the IP message descriptor using the length field from the trailer and dispatches the message descriptor to the IP CPRC. It then sets the offset field in the rasList entry to zero and allocates a new buffer for the next SDU on this VCC.

- If the response indicates a CRC failure, it means that the AAL5 SDU suffered errors in transit and should be discarded. The offset is set to zero so that the next SDU on this VC can reuse the current SDRAM buffer.

- The descriptor in case of valid SDUs is then queued to the appropriate queue. For MPLS Switched SDUs, it queues it to MPLS CP, else to the IP module.
- The maximum permitted SDU size is restricted to 2048 bytes in MSA application. If the offset field in the rasList entry exceeds 2048, the SDU is to be discarded. For an SDU that exceeds 2048 bytes, subsequent ATM cells are appended in the last 48 bytes of its buffer – when the output thread finds the offset to exceed 2048, it always copies the current cell into the last 48 bytes. When the last cell for this SDU arrives, the cell is copied into the last 48 bytes and the offset is then set to zero. This effectively discards the SDU and makes the buffer available for the next SDU.
- Frees the scope to RxSDP

7.10.3 Data Structures

7.10.3.1 Merge space

The RC writes information about outgoing packets to merge space for TxByte to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	rasType	Eom_offset	vcCidIndex	
4	NumAlignedBytes	NumUnalignedBytes	NumBytes PartialPayload	pduLen
8	PartialPayload			
12	PartialPayload			
16	PartialPayload			
20	PartialPayload			

24	AppData
28	AppData
32	AppHdrData

```
typedef struct{
        int8u       rasType;
        int8u       eom_offset;
        int16u       vcCidIndex;
        int8u       numAlignedBytes;
        int8u       numUnAlignedBytes;
        int8u       numBytesPartialPayload;
        int8u       pduLen;
        int8u       partialPayload[16]; /* Partial Payload */
        int16u        data[4];
        int32u        appHdrdata;
} RasMergeSpace;;
```

- rasType – 0x00 for AAL5 and 0x01 for MPLS Switched AAL-5
- eom – 0x00 if this is not the last cell and 0x80 if it is the last cell of a SDU
- vcCidIndex – the VC Index
- numAligned – the number of bytes in the data stream that should be transferred to SDRAM
- numUnAligned – the number of bytes in the data stream that will be transferred to the extract space
- numBytesPartial – the number of unaligned bytes from the previous ATM cells
- pduLen – the size of the ATM PDU
- partialPayload[16] – the unaligned bytes from previous ATM PDU
- AppData – application specific data for MPLS switched AAL-5 SDU
- AppHdrData – the egress cell header.

7.10.3.2 Extract Space

RxByte writes information about recirculated datagrams into extract space for the RC to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	rasType	Eom_offset	vcCidIndex	
4	PduLength	NumUnalignedBytes	pad	
8	Uui	Cpi	PayloadLength	
12	CRC			
16	PartialPayload			
20	PartialPayload			
24	PartialPayload			
28	PartialPayload			pad
32	AppData			
36	AppData			

40	AppHdrData

```
typedef struct {
    int8u     rasType;
    int8u     eom_offset;
    int16u     vcCidIndex;
    int8u     pduLength;
    int8u     numUnAlignedBytes;
    int8u     reserved[2];
    int8u     uui;
    int8u     cpi;
    int16u      payloadLength;
    int32u     crc;
    int8u     partialPayload[15];
    int16u     descData[4];
    int32u     appHdrData;

} RasExtractSpace;
```

- rasType – 0x00 for AAL5 and 0x01 for MPLS Switched AAL-5
- eom – 0x00 if this is not the last cell and 0x80 if it is the last cell of a SDU
- vcCidIndex – VC index
- pduLength – the size of the ATM PDU
- numUnAligned – the number of unaligned bytes in the extract space
- pad – unused
- uui – user-to-user indication from the AAL5 trailer
- cpi – reserved
- payloadLength – size of the reassembled AAL5
- crc – CRC-32 from the AAL5 trailer
- partialPayload[15] – the unaligned bytes from current ATM PDU
- AppData – application specific data for MPLS switched AAL-5 SDU
- AppHdrData – the egress cell header

7.10.3.3 RasList

The RasList structure is used to track re-assembly state per VC. It is an array of 1024 structures with the following format:

Byte Offset	0	1	2	3
0	BufHandle			
4	Offset			

- bufHandle – the handle of the buffer in which cells are being reassembled
- Offset – the offset in the reassembly buffer at which the next cell should be placed

7.10.3.4 Descriptor information

The following is the data structure of the descriptor, which is en-queued or de-queued:

Byte Offset	0	1	2	3
0	bufHandle			
4	Length		port_bufType	
8	appData			
12	appData			

```
typedef struct {
        BsBufHandle bufHandle;
        int16u      length;
        int16u      port_bufType;      /* 15:5 - port, 4:0 - bufType */
        union {
                int8u           byte[8];
                int16u          hword[4];
                int32u          word[2];
                AtmDescData     atm;
                FrDescData      frameRe;
                SegDescData     seg;
                TdmDescData     tdm;
                MlPppDescData   mlPpp;
                MplsDescData    mpls;
        } appData;
} DescriptorMsg;
```

- bufHandle – reassembly buffer handle
- length – chunk length
- port_bufType – input port and buffer type of the next module
- appData - Application specific data.

7.10.3.5 Ring Bus Slots

The reassembly CP launches a TLU XOR command into RAS CRC table. It uses following slots for the purpose:

Request Slot	0
Response Slot	2

7.10.4 Issues/Enhancements

None

7.11 IP QoS Classifier (CP 10)

CP10 implements the IP QoS Classifier component that provides classification for quality of service (as specified in RFC 2474,RFC 2475, RFC 1633, RFC 2211 and RFC 2212) that will be configured in Q-3 TMC. The QoS implemented are DiffServ and IntServ in Q-3. This component does not use the SDP. It uses a single CPRC to perform QoS classification for DiffServ and IntServ at all ingress queues in the system. The IP QoS Classifier CPRC has an input QMU queue for every egress IP port. This component uses multiple fields from the IP and TCP/UDP headers to determine the traffic category (i.e., the flow for an IP packet) and sends it to Q-3 that will apply the appropriate marking and shaping strategy on it. Currently, the design supports IPv4 only.

7.11.1 XP Initialization

XP initialization is mentioned in section 7.1.1.

7.11.2 Host Configurations

Host is responsible to perform the Q-3 TMC configurations as shown in figure 6. It configures the following parameters in the Q-3 map.

- Since total number of channels to be supported are 2K. Each channel consists of 4 traffic queues for both DiffServ and IntServ QoS treatment. i.e. the four queues per channel will either be used by DiffServ or IntServ based on PHB in flow table. For DiffServ it will support 2 AF queues, 1 EF queue and 1 Best effort queue per channel. Whereas for IntServ it will support 2 guaranteed service traffic queues and 2 controlled load service traffic queues. These traffic queues are configured in flow table using PHB field. So total number of traffic queues to be supported = 2K *4 = 8K (for either DiffServ or IntServ)
- One Level2 scheduler (each having 4 inputs) will be configured for each channel. So it will need 2K level2 schedulers for normal path. Discard path will also need one scheduler. So Total number of level2 schedulers = 2K+1
- Each Level1 scheduler will consist of 1K inputs. But total number of incoming inputs will be 2K. So total number of level1 schedulers = 2K/1K = 2
- At the top level, it is mandatory to have one level0 scheduler in Q-3 hierarchy.
- Total number of VOPs to be created will be 128 for normal path and one for discard path. VOPs are created for every T1/E1 interface. VOPs are also created for inter CP communication.

The IP QoS related Q-3 configuration steps are given here. The Q-3 configurations for other C-3e components are given in section 7.21.2. The configuration steps starting from top to bottom in Q-3 hierarchy (i.e. from level 0 scheduler to traffic queues) are described as follows.

- Initializes the Q-3 TMC using qsTmcInitialize ().
- Creates one level0 RR scheduler with 3 input legs using qsTmcSchedCreate ().
- Creates the discard path as follows:
    - Creates one level2 RR scheduler with one input leg using qsTmcSchedCreate ().
    - Creates the discard queue using qsTmcTrafficQueueCreate () and associates it with the level2 scheduler created as above.
    - Creates the VOP as 0 for discard path using qsTmcVopCreate() and associates it with first leg of level0 scheduler.
- Creates parent buffer pool and buffer pool associated with it using qsTmcBufferPoolCreate(). These buffer pools will be used by traffic queues.
- Creates the 2-level1 WFQ schedulers each having 1024 (1K) input legs. These schedulers will feed the level0 RR scheduler. The maximum number of level1 schedulers (having 1K inputs) to be supported is 18 in new Q-5.

- Creates the 2K-level2 WFQ schedulers each having 4 input legs that will receive the input descriptor from four traffic queues via four scheduler queues. Each scheduler queue is mapped with one traffic queue. This mapping is not shown in the figure 3.0. For DiffServ treatment, the input descriptor may originate from AF traffic queue or EF traffic queue or best effort traffic queue that will be configured in flow table. For IntServ treatment it may originate from either guaranteed service traffic queues or control load service queues configured for it in flow table. These schedulers will feed the level1 WFQ scheduler. The max number of level2 schedulers to be supported is 18K.
- The following traffic parameters will be configured in each level2 scheduler for each traffic flow.

    - For DiffServ treatment

        - CBS: Committed Burst Size
        - EBS; Excess Burst Size
        - Increment; The number of tokens added into token buckets at every tick
        - Tc; Current token bucket size for committed bursts
        - Te; Current token bucket size for excess bursts
        - CIR; Committed Information Rate
        - LastUpdateTime; Last time when traffic was seen on this flow

    - For IntServ treatment (Guaranteed service parameters)
        - Token Bucket Rate (r)
        - Token Bucket size (b)
        - Peak data rate (p)
        - Min Policed Unit (m)
        - Max Policed Unit (M)
        - Rate (R)
        - SlackTerm (S)

    - For IntServ treatment (Controlled load service parameters)
        - Token Bucket Rate (r)
        - Token Bucket size (b)
        - Peak data rate (p)
        - Min Policed Unit (m)
        - Max Policed Unit (M)

- Creates two discard blocks using qsTmcDiscardCreate() and associates these with the discard queue as created in above step. These discard blocks are as follows:
    - Discard block1
        - Type: Single token bucket (qsDiscTypeSingBucket)
        - Discard parameters: 1b1Limit and 1b1Increment
    - Discard block2
        - Type: RED (qsDiscTypeRED).
        - Discard parameters: probabilityMax

- Creates 8K traffic queues that will pass the traffic to the Q-3 hierarchy and associates these queues with buffer as allocated above. It also associates the queues with the discard blocks. One channel will have 4 traffic queues. Max number of traffic queues to be supported in new Q-5 is 128K.
- The 128 different VOPs for normal path have been created in section 7.21.2. It configures among these 128 VOPs with each of the 2K traffic paths.
- Enables the Q-3 configuration map using qsTmcEnqueueEnable().
- Communicates with XP to indicate that Q-3 configuration is done.

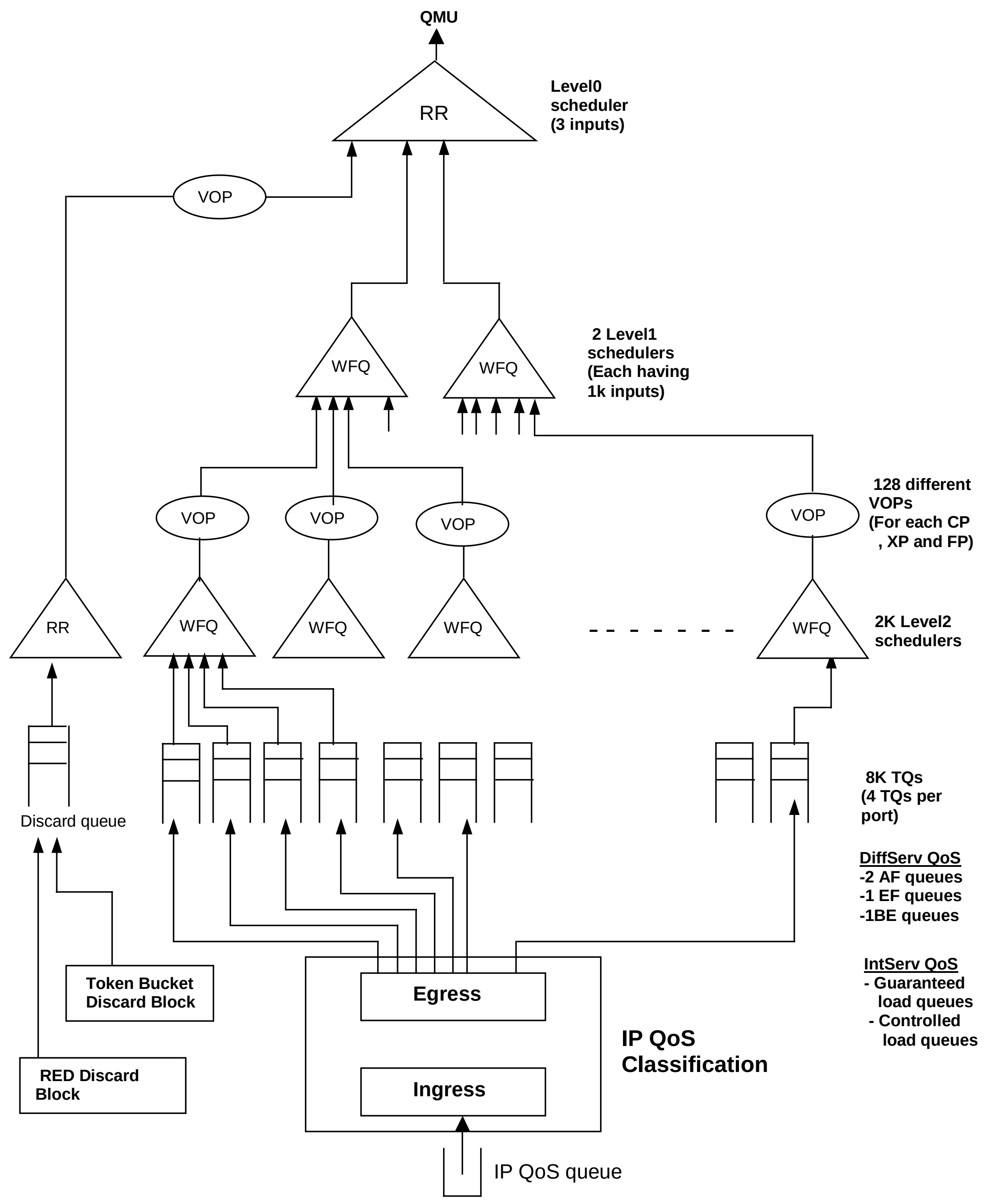

Figure 6- Q-3 configuration map for IP DiffServ and IntServ

7.11.3 RC

7.11.3.1 Initialization

The IP QoS Classifier component does the following things during its initialization:

- Creates the input and output threads.
- Initializes the flow table lookup.
- Communicates with XP to indicate that it has enabled.
- Jumps to the ingress thread.

7.11.3.2 Ingress Thread

IP packet descriptor is de-queued from IP QoS queue. Packets coming to the IP QoS CPRC already have the complete IP header. The first 64 bytes of the packet contain all the protocol headers. They are DMA transferred into DMEM buffer using bsBufferRead(). The specific header fields that should be used to determine the traffic category (that is, the flow) for the IP packet are specified in a Multi Field (MF) mask. They are as shown in Figure 7.

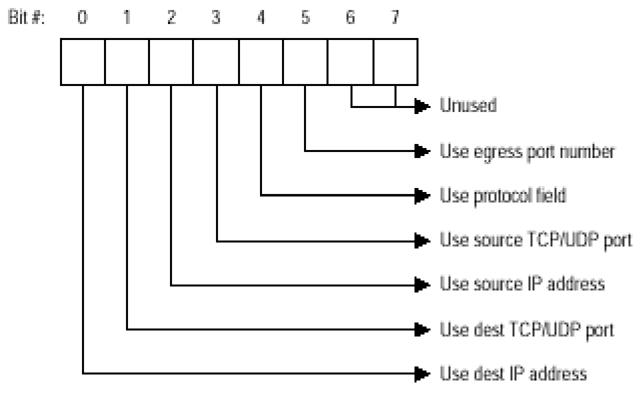

Figure 7- DiffServ Multi field bitmask

The header fields specified by the MF mask are concatenated to form up to a 14-byte TLU lookup key. An exact match table (IP flow table) is maintained in TLU table, to match these keys to a flow id and a DiffServ PHB.

Ingress performs the following functions:

- Dequeues the IP packet descriptor from IP QoS queue.
- Gets the port and QoS queue from the incoming descriptor enqueued by IP component. This QoS queue denotes the Q-3 base traffic queue. IP component would have got the QoS queue by performing port lookup and sent it into IP QoS classifier component.
- Forms the 14-byte TLU lookup key based on the 5-tuple fields in the IP packet header.
- Launches flow table lookup.

- Repeats the above steps for all packets that are present in IP QoS Queue.

7.11.3.3 Egress Thread

After launching a flow table lookup, the context switches from ingress thread to egress thread, which waits for the results of the flow table lookup. It performs the following functions.

- Waits for the flow table lookup to be completed.
- Gets the PHB and traffic queue offset based on lookup result. Flow table gives the appropriate traffic queue offset corresponding to the QoS treatment. The QoS treatment will be configured as DiffServ or IntServ in the flow table. In the case of DiffServ, it will denote the AF or EF or BE. Whereas in the case of IntServ it will denote guaranteed service or control load service.
- Determines the destination traffic queue as follows:
  Destination traffic queue = QoS queue + traffic queue offset
  QoS queue will be determined from incoming IP descriptor in Ingress function.
- For AF and EF, it determines the appropriate DS code point, updates it in IP header (stored in SDRAM) using a bsBufferWrite() function and adjusts the IP checksum accordingly.
- Enqueues the packet descriptor into destination traffic queue using qsEnqueueExt().

7.11.4 Q-3 Functionality

IP QoS classifier component en-queues the packet into Q-3 traffic queue to apply various QoS parameters (DiffServ or IntServ). IP component performs the port table lookup to get the base traffic queue corresponding to the port and sends this queue information to IP QoS classifier component if QoS treatment is needed. IP QoS classifier then determines the destination traffic queue based on the offset in flow table and finally enqueues the descriptor into Q-3 traffic queue. Q-3 TMC provides marking/dropping, policing and traffic shaping for the packet based on configured traffic parameters. Q-3 TMC will enqueue the conformant packets into QMU queue via VOPs. Non-conformant packets will either be discarded using discard path or marked as low priority. Traffic parameters and scheduling in Q-3 is described in host configuration section 7.11.2.

7.11.5 Data Structures

7.11.5.1 Descriptor information

IP descriptor will be used as given in section 7.8.3.3.

7.11.6 Flow Table

Each entry of the flow table has the following format:

Byte Offset	0	1	2	3
0	destIp			
4	srcIp			

8	destPort		srcPort	
12	protocol	egressPort	maskBits	phb
16	flowId		queueOffset	
20	pad			

The description of each field is as follows:

DestIp:	destination IP address of the packet
SrcIp:	Source IP address of the packet
DestPort:	TCP/UDP destination port of the packet
SrcPort:	TCP/UDP source port of the packet
protocol:	protocol (TCP/UDP) in the packet
egressPort:	output port
maskBits:	number of significant bits in the key
phb:	per hob forwarding behavior to apply. It will denote DiffServ (AF/EF/BE) or IntServ (Guaranteed service/Control load service).
flowId:	Flow Id that defines packet flow
queueOffset:	Traffic queue offset

7.11.7 Issues/Enhancements

- Since total number of channels to be supported is 2048 so egressPort field will require 11 bits but egressPort field in flow table lookup key is of size 1-byte because of key size limitation (max 14 byte).

7.12 FR processing – sw itching (CP11)

The frame relay processing for the application is distributed into two CPs (CP11 and CP3). CP11 will perform the FR switching and CP3 will perform the FR encapsulation. All the packets, which belong to FR, will be queued to CP11 and it decides whether the packet needs the MPLS, IP or FR processing. The packets that originate from MPLS or IP module, which have to pass through FR interface, need FR header insertion, which will be carried out by CP3.

CP11 is used to re-circulate the FR frames destined for TDM Tx for transmission. Its main functions are:

- Launch lookup based on the DLCI value from the descriptor.
- Based on the response, en-queue the packets to the appropriate queue (IP or MPLS) if needed.
- Perform the FR switching by properly modifying the FR header with the outgoing DLCI value.
- Properly en-queue the modified buffer to the EgressQueue.

7.12.1 SDP

The SDP is configured for byte-level re-circulation. The SDP adds the FR header to the frame. The functions provided by each of its component processors are described below.

7.12.1.1 RxByte

The RxByte processor fills the extract space based on the descriptor information sent by the TxByte processor and streams the modified payload to the output thread (receive side) of the RC .The RxByte processor performs the following functions as part of the FR recirculation:

- Waits for a receive scope to be available.
- Receives the bytes from the TxByte processor.
- The first two bytes of the payload contain the egress_queue. This is written to the extract space. The next two bytes contain the outgoing port information and buffer type information. This will also be written to the extract space.
- The packet length is indicated in the next two bytes. This is written to the extract space and header ready is indicated in the rxStatus register.
- Stream the remaining payload to DMEM.
- When data9 (i.e., ninth bit is set in the incoming payload) is received switches scope.

7.12.1.2 TxByte

The TxByte processor modifies the FR header based on the information given by the input thread (transmit side) of RC and transmits the remaining payload data to RxByte processor. TxByte processor performs the following functions as part of the FR re-circulation:

- Reads the payload from DMEM until Data9 is observed.
- Reads the outgoing DLCI value from the merge space and modify the existing value in the packet with the new value.
- Set the congestion control information fields in the FR header to zero.
- Transmit the egress_queue, port_buftype and length of the packet before the payload data so that the RxByte receives it and places in the extract space.
- Sends the FR payload data to RxByte.
- Switches scope and waits for next packet to be available.

7.12.2 RC

The RC manages the FR re-circulation. All CPs that want to transmit FR frames, en-queue their packets to the FR re-circulation CP so that the FR header is modified in the frame before transmission.

7.12.2.1 Initialization

The initialization component initializes the data structures and registers used by the RC. The following activities are done during the initialization:

- Initializes the buffer pools.
- Creates input and output threads.
- Initializes the FR table lookup slots.

7.12.2.2 Input Thread

The Input thread services all the FR traffic which needs the FR header modification in the frame is queued here. This passes the frame to SDP.

- De-queues the descriptor information.
- Launch a lookup based on the DLCI value from the descriptor. If the lookup fails, silently drop the packet.
- While waiting for the response, process the next packet from the queue.
- Based on the egress_port (from the lookup response), en-queue the packet to IP or MPLS queue.
- Depending on whether the packet goes to IP or MPLS build the descriptor with the appropriate fields.
- For MPLS the fields that need to be filled in the descriptor are: port_buftype, LabelsSwap, LabelPush, EgressQueue, HopCount, InIfType_action
- For IP only port_buftype is filled in the descriptor.
- If the packet is FR switched, fill the merge space with outgoing DLCI value (from the lookup response) and header information.
- Initiates payload transfer from SDRAM to DMEM and switches context to the next thread.
- Loops to the beginning to wait for another descriptor.

7.12.2.3 Output Thread

The output thread handles the incoming packets from the RxByte processor.

- Waits for the header processing completion on the RxByte processor indicated by L1Done in the rxStatus register.
- Allocates new buffer and initiates the payload transfer from DMEM to the SDRAM.
- Creates the descriptor with buffer handle, buffer length and port_bufType information.
- Enqueue the descriptor to the appropriate destination based on the egress_queue.
- Switches context to the next thread.
- Loops to the beginning to wait for another scope to be available.

7.12.3 Data Structures

The merge space and Extract space structures used in the FR processing are explained below:

7.12.3.1 Merge space

The RC writes information needed for FR encapsulation and descriptor fields to be used by Rx CPRC, into merge space. TxByte processor does FR encapsulation and sends the descriptor fields to RxByte. The data structure has the following format:

Byte Offset	0	1	2	3
0	FrHeader			
4	port_bufType		Length	

8	EgressQueue	Pad

The explanations for the above-mentioned fields will be as follows

- FrHeader: The outgoing DLCI value.
- EgressQueue: specifies the final queue.
- port_buftype – Outgoing interface information and the buffer type
- Length – Length of the packet.
- Pad: unused.

7.12.3.2 Extract Space

RxByte processor writes descriptor information into extract space for the RC to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	EgressQueue		Pad	
4	port_buftype		Length	

The explanations for the above-mentioned fields will be as follows

- EgressQueue: specifies the final queue.
- port_buftype: Outgoing interface information and the buffer type
- Length: Length of the packet.
- Pad: unused.

7.12.3.3 Descriptor information for FR

The TDM RX module will fill the FR descriptor information to be used by the FR re-circulation module for processing. The format of the descriptor will be as follows:

Byte Offset	0	1	2	3
0	BufHandle			
4	Length		PortBufType	
8	FrHeader			

The explanations for the above-mentioned fields will be as follows

- BufHandle: Handle of the buffer being re-circulated.
- Length: specifies the length of the buffer.
- Port_bufType: a bitmask is defined as follows:
  - b15-5: port – the output port on which this datagram to be transmitted.
  - b4-0: bufType – the type of buffer being re-circulated and the egress port type (could be BT_IPV4, BT_MPLS_FR,BT_FR)
- FrHeader: specifies the DLCI value and the congestion control Information.

7.12.3.4 Ring Bus Slots

FR uses these slots

• FR Request slot	1
• FR Response slots	2,3

7.12.4 Issues/Enhancements

- At present the congestion control information in the FR header is not processed.
- The two bytes address format of DLCI (10-bit) is supported now.
- FR module can be Extended to support three (16-bit DLCI) and four bytes (23-bit DLCI) address format of DLCI.

7.13 MPLS (CP12)

CP12 implements the MPLS component for the MSA application. This component performs MPLS lookup and operation (MPLS actions performed for label encapsulation or label switching). The MPLS operation is done based on the MPLS entry information (labelSwap, labelPush, MPLS action, hopCount) maintained in the different TLU tables.

The tables mentioned below maps the input frame/packet type (PPP / ATM / FR / IP) to the TLU table in which MPLS entry is maintained.

Input frame /packet type	TLU table	CP from which lookup launched	Notes
PPP	MPLS Table (This is defined in the section 7.18.8)	MPLS CP	This table is also used for POP action (which needs additional label lookup based on the next label in the stack) in the case of MPLS packet over ATM and FR
ATM	ATM VC table	TDM CP	
FR	FR Table	FR CP	
IP	IPv4 Routing table	IP CP	This table handle MPLS ingress packets

The top label of the MPLS packet maps to VPI/VCI for ATM cell and DLCI for FR frame. Hence, the lookup response of ATM VC table and FR table is the MPLS entry information.

7.13.1 SDP

The SDP is configured for byte level re-circulation. The SDP re-circulates the packet to remove PPP/FR header, sends descriptor information to be used by Rx CPRC and to perform MPLS operation.

7.13.1.1 TxByte

The TxByte processor, removes the PPP/FR header, sends descriptor information to RxByte and performs MPLS operation.

As MPLS operation involves addition and removal of shim header fields (LabelSwap, LabelPush, BOS, ttl) to the packet, TxSDP is the appropriate place chosen to perform this functionality.

The TxCPRC writes information about the MPLS data needed for MPLS operation and the descriptor information (to be queued to the respective module based on the egress port type) into merge space. The data structure is defined in section 7.13.3.1. TxByte is not configurable through control space.

The detailed flow of Transmit Byte processing is described below:

- Wait for Transmit scope from the CPRC.
- Removes FR/PPP header(if any) based on the header length value set from Transmit CPRC.
- Sends the following descriptor information in merge space to the RxByte,
    - rxAlgorithm
    - Port_bufType (15:5 – egress port, 4:0 – egress bufType)
    - Packet header (CellHeader for ATM or DLCI for FR) and egressQueue.
- MPLS Operation is done based on the MPLS command, whose functionality is described below:
    - Get the MPLS command and TTL value from the merge space and save the same to the temporary registers.
    - Check for POP command. If the command is POP, Get the number of labels to be removed. Remove the shim header and repeat the removal of header until the count reaches the number of labels to be removed. Jump to check for SWAP_PUSH command.
    - Check for SWAP_PUSH command. If the command is SWAP_PUSH, Perform the PUSH operation followed by a SWAP. For a PUSH operation, Construct MPLS header using 'labelPush' from merge space, the BOS bit with zero as the value and the stored TTL value. Finally, Jump to SWAP operation. Else, Jump to check for SWAP command.
    - Check for SWAP command. If the command is SWAP, Send the 'labelSwap' from merge space, the old BOS bit and the stored TTL value. Save the old BOS bit and Jump to check for the presence of shim header(s). Else, Jump to check for PUSH_PUSH command.
    - Check for PUSH_PUSH command. If the command is PUSH_PUSH, Perform, a PUSH operation and do a next PUSH by jumping to PUSH operation. Send the 'labelPush 'from merge space, the BOS bit with zero value and the stored ttl value else, Jump to check for PUSH command.
    - Check for PUSH command. If the command is PUSH, Send the label to be pushed from merge space, the BOS bit (set only for the packets received from IPv4 module) and the stored TTL value. Jump to check for the presence of shim header(s). Else, Jump to check for the presence of shim header(s).

    - Check for the presence of shim header based on the BOS bit saved during the MPLS operation. If BOS bit is not set, (i.e. presence of one or more shim header)
    - Stream the shim header until the last shim header is reached. Update the TTL from merge space for all shim header(s). Else, Jump to check for POP_IPv4 command.

  - Check for POP_IPv4 command based on the Tx ALG value set to BT_MPLS_IPv4.If POP_IPv4, update the ttl and IP checksum from the merge space to the IP header else jump to stream the remaining payload.
- Stream the payload until the Data9 set in the last byte.
- Sends an end of packet byte with Merge9 set.
- Switches scope and waits for next packet.

7.13.1.2 RxByte

The RxByte processor, extracts the descriptor information and performs NULL Label encapsulation for MPLS packet over ATM / FR.

The RxByte processor writes information about the descriptor into extract space for the RxCPRC to use in its processing. The data structure is defined in section 7.13.3.2. RxByte is not configurable through control space.

The detailed flow of RxByte processing is described below:

- Waits for a receive scope to become available.
- Receives descriptor information from TxByte and places it in extract space.
- If RxAlgorithm is set to MPLS_NULL_LABEL, the top label is passed with the NULL label (only for MPLS packet over ATM/FR)
- Streams the remaining bytes to DMEM
- Switches scope and waits for another to become available.

7.13.2 RC

The MPLS component uses two threads to perform its task, namely, an input thread and an output thread. The initialization code starts the two threads. Each of these is described in the following sections.

7.13.2.1 Initialization

The MPLS component does the following things during its initialization:

- Initializes the buffer pools.
- Allocate the buffers for the receive path. There are two data scopes for handling inbound packets. Allocate a buffer for each data scope and setup the registers for receiving data.
- Allocate the buffers for the transmit path, which are never used. This is used to simplify the logic of freeing buffers in the input thread.
- Creates the input and output threads
- Initializes the MPLS Table lookup to be launched from TxCPRC.
- Initializes both scopes by giving the SDP ownership.
- Enable the SDP.
- Jumps to the first thread.

7.13.2.2 Input Thread

The Input thread fills the merge space with the MPLS forwarding information for MPLS operation and the descriptor fields to be passed to RxCPRC for further processing. The merge space information is obtained from the MPLS entry maintained in different TLU tables. (Refer section 7.18.8)

The detailed flow of Input thread processing is described below:

- Wait for a descriptor to be present in its queue then de-queues it. The descriptor is filled by any of the following CP components:
    - TDM CP for MPLS over ATM packets.
    - TDM CP for MPLS over PPP packets.
    - FR CP for MPLS over FR packets.
    - IP CP for IP packets entering MPLS domain.

Note: For all cases except PPP packets and POP operations, the descriptor holds the necessary information to fill the merge space.

- Read the first 64 bytes of SDRAM buffer to get the Label Stack.
- Adjust the buffer offset pointing to the start of Label stack.
- Get the input port type. (ATM /FR/PPP/IPv4) from the descriptor.
- If the input port type is ATM/FR/IPv4, perform the following functions:

    - Get the TTL from the incoming packet (If the input port type is ATM/FR, TTL is extracted from the shim header and is extracted from IP header if the input port type is Ipv4).
    - Decrement the TTL value with the descriptor field 'hopCount'
    - Save the updated TTL value and port_buftype.
    - Get the MPLS action and check for POP command.
    - If POP command and Label stack is with more than one MPLS label, then set a flag to perform additional label lookup. For all other cases, save the MPLS information (labelSwap, labelPush, cmds, ttl, port_bufType) to be copied in merge space.

- If the input port type is PPP or POP command (which needs additional lookup) perform the following.
    - Launch the lookup with the MPLS label in the shim Header.
    - Wait for the MPLS lookup to complete and if the lookup fails, drop the packet.
    - Save some of the merge space information (port_bufType, TTL) from the lookup response.
    - For POP action, repeat the MPLS Label lookup. The number of POP operations to be performed is stored in the 3 least significant bits of the cmds field of merge space. In addition, more than one MPLS operation like POP and SWAP, POP and PUSH are also stored in cmds field. (Refer merge space structure for more details)
    - Based on the MPLS action, save the remaining merge space information (cmds-MPLS command, labelSwap, labelPush).

- If the egress-port type is ATM /FR, fill the egressQueue and the header information (Cell Header for ATM and DLCI for FR).
- Fill the txAlgorithm with egressBufType
- Fill the rxAlgorithm with MPLS_NULL_LABEL for MPLS packet destined to ATM /FR interface.
- For the MPLS command 'POP_IPv4_LOOKUP', form the IP checksum adjusted for the new TTL value.
- Fill the headerLen with the negation of packet header length.
- Waits for a transmit scope to be available from the SDP.
- Fills in merge space from the saved merge space.
- Waits for the previous DMA transfer to complete.
- Begins the DMA transfer for the current buffer being processed.
- Switch context to the next thread.
- Loops to the beginning to wait for another descriptor.

7.13.2.3 Output Thread

The output thread handles sending the descriptor to the appropriate modules based on the egress port type (ATM /FR /PPP/IPv4).

The detailed flow of Output thread processing is described below:

- Waits for a scope to become available from the SDP.
- Fills in a descriptor using information from the extract space.
- Waits for the payload transfer to complete.
- Allocates new buffer and initiates payload transfer from DMEM to the SDRAM.
- Sends the descriptor to the appropriate destination based on the egress port type.
- Switches context to the next thread.
- Loops to the beginning to wait for another scope to be available

7.13.3 Data Structures

7.13.3.1 Merge space

The RC writes information needed for MPLS operation and descriptor fields to be used by Rx CPRC, into merge space. TxByte processor does MPLS operation and sends the descriptor fields to RxByte processor. The data structure has the following format:

Byte Offset	0	1	2	3
0	Pad	txAlgorithm	cmds	ttl
4	Label Push			
8	Label Swap			
12	headerLen	rxAlgorithm	port_bufType	
16	appData			
20	appData			

- Pad-– unused

- TxAlgorithm – holds the egressbuftype based on which TxByte identifies the MPLS action 'POP_IPV4_LOOKUP' and the necessary action is carried out (i.e. updation of ttl and IP checksum in IP header is performed).
- cmds – MPLS command used by TxByte Processor for MPLS operation. Input thread sets these values based on the MPLS actions (For, POP_IPv4_LOOKUP action the POP command is set). The 3 least significant bits are used by POP command to store the number of labels to be removed. Possible enumerated values are shown below:

```
typedef enum {
   MPLS_CMD_NONE = 0x00,
   MPLS_CMD_ADD = 0x10,
   MPLS_CMD_PUSH_PUSH = 0x08,
   MPLS_CMD_SWAP = 0x20,
   MPLS_CMD_SWAP_PUSH = 0x40,
   MPLS_CMD_POP = 0x80

} MplsCommands;
```

- ttl –time to live
- labelPush –label to be added.
- labeSwap – Label to be swapped.

The above fields are used for MPLS operation in TxByte processor.

- headerLen – Length of the header (for PPP and FR) bytes to be removed. It holds the negative value of the header length.

The descriptor fields sent to RxByte processor are mentioned below:
- rxAlgorithm – NULL label is sent for MPLS packet over ATM and FR, if this field is set to 0x10 by MPLS Tx CPRC.
- port_bufType – a bitmask defined as follows:
    - b15-5: port – the output port on which this datagram to be transmitted.
    - b4-0: bufType – the type of buffer being re-circulated (could be BT_MPLS_PPP, BT_MPLS_ATM, BT_MPLS_FR, BT_MPLS_IPv4)
- appData – application specific data as defined below

The appData field has meaning specific to the processing block to which it has been en-queued. There are several formats for the appData field.

The segmentation appData field for ATM has the format: specified in the Section 7.4.3.3

The appData field for FR has the following format:

Byte Offset	0	1	2	3
0	frHeader			
4	egressQueue		pad	

- frHeader – DLCI value
- egressQueue – Queue for the egress interface
- pad -unused

7.13.3.2 Extract Space

RxByte processor writes descriptor information into extract space for the RC to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	pad		port_bufType	
4	AppData			
8	AppData			

- pad – unused
- port_bufType – a bitmask defined as follows:
  - b15-5: port – the output port on which this datagram to be transmitted.
  - b4-0: bufType – the type of buffer being recirculated (could be BT_MPLS_PPP, BT_MPLS_ATM, BT_MPLS_FR, BT_MPLS_IPv4)
- appData – application specific data as defined below

The appData field has meaning specific to the processing block to which it has been enqueued. There are several formats for the appData field.
The segmentation appData field for ATM has the format specified in the section 7.9.3.3
The FrDescData appData field for FR has the format: specified in the section 7.13.3.1

7.13.3.3 Descriptor information

MPLS Descriptor is filled by the following CP components.

- TDM CP for MPLS over ATM packets.
- FR CP for MPLS over FR packets.
- TDM CP for MPLS over PPP packets.
- IP CP for IP ingress packet.

Note: For all cases except PPP packets and POP operations, the descriptor holds the necessary information to perform MPLS operation.

Byte Offset	0	1	2	3
0	labelSwap		labelPush	
4	hopCount	inIfType_action	egressQueue	
8	appHdrData			
12	vcIndex		Pad	
16	Pad			
20	Pad			
24	Pad			

- labelPush – Label to be added.
- labeSwap – Label to be swapped.
- hopCount – value to be decremented to the ttl.
- inIfType_action // 0-3 MPLS action 4-7 inIfType

The enumerated values for MPLS action is provided in Section 7.13.3.1. The enumerated values for input interface type are shown below.

```
typedef enum {
   MPLS_PPP=0x10,
   MPLS_ATM=0x20,
   MPLS_FR=0x30,
   MPLS_IPv4=0x40
} portType;
```

- egressQueue – egress queue of the packet.
- appHdrData – header information (Cell Header for ATM and DLCI for FR).
- vcIndex – ndex used by AAL-5 SARs to index their tables.
- Pad –Unused.

7.13.3.4 Ring Bus Slots

MPLS uses these slots:

MPLS request slot	0
MPLS response slot	0,1

7.14 MLPPP (CP 13)

CP13 implements ML-PPP for the application. This component segments datagrams into fragments for transport over ML-PPP bundles and provides multiple classes of service. The component also reassembles ML-PPP fragments and forwards them to the next processing block. This CP uses the ML-PPP remainder table discussed in section 7.18.2

7.14.1 SDP

The SDP is configured for byte re-circulation and provides byte level processing of ML PPP fragments including header parsing and generation. RxBit, RxSync, and TxBit are not used.

7.14.1.1 RxByte

The RxByte processor receives ML-PPP fragments from the TxByte processor and handles them differently depending on whether segmentation or reassembly is needed. In the reassembly direction, the RxByte processor does the following:

- Waits for and receives the reassembly operation code byte and puts it in extract space.
- Receives control information (port, index, fragType, length) and puts it in extract space.
- For SOM fragments, also receives the bufType and puts it in extract space.
- Indicates header processing is done.

- For EOM fragments, streams all payload bytes to DMEM, switches scope, and waits for next operation code.
- For non-EOM fragments, streams 'length' bytes of payload to DMEM.
- Streams any remaining payload bytes to the ring bus Tx message register.
- When data9 is received, fills in count of remaining bytes in ring bus Tx message register and initiate TLU write command to send the remainder bytes to the ML-PPP remainder table.
- Switches scope and waits for next operation code.

In the segmentation direction, RxByte processor does the following:

- Waits for and receives the segmentation operation code and puts it in extract.
- Receives the port number and puts it in extract.
- Indicates header processing is done.
- Receives the ML-PPP encapsulated fragment.
- Writes the payload to DMEM.
- When data9 is received, switches scope and waits for the next operation code.

The RxByte processor communicates with the RC through extract space. Extract space is described in section 7.14.3.1. The RxByte processor is not configurable through its control space.

7.14.1.2 TxByte

The TxByte processor transmits ML-PPP fragments to the RxByte processor. In the reassembly direction, the TxByte processor does the following:

- Waits for scope to be available
- Send the reassembly operation code to RxByte processor.
- Sends control information to RxByte including port, index, and fragType from merge space.
- Removes any HDLC, PPP, or ML-PPP encapsulation from the fragment.
- For SOM fragments, parses the reassembled PPP protocol field and maps it to the buffer type.
- Calculates the length of the fragment including for non-SOM fragments, but not including discarded framing bytes.
- Sends the length, and if necessary send bufType to RxByte processor
- Sends any remainder bytes from merge space to RxByte processor.
- Streams the payload from DMEM to RxByte processor.
- Transmits data9 with a dummy byte to indicate end of packet.
- Switches scope and waits for scope to be available again.

In the segmentation direction, TxByte processor does the following:

- Waits for scope to be available
- Sends the segmentation operation code to RxByte processor.
- Sends the port to RxByte processor.
- Sends the HDLC/PPP/ML-PPP header from merge space to RxByte processor.
- Reads payload from DMEM and sends the number of bytes specified in merge space to RxByte processor.

- Transmits data9 with a dummy byte to indicate end of packet
- Switches scope and waits for scope to be available again.

The TxByte processor communicates with the RC through merge space. Merge space is described in section 7.14.3.2. The TxByte processor is not configurable through its control space.

7.14.2 RC

The RC manages the en-queuing and de-queuing of ML-PPP fragments and IP datagrams. It also maintains state information necessary for segmentation and reassembly. In addition, the RC schedules packets for segmentation.

7.14.2.1 Initialization

The initialization component initializes the data structures and registers used by the RC. Specifically, it does the following:

- Initializes the buffer pools.
- Creates the contexts for the input, output, and scheduler threads.
- Initializes the ring bus Tx message registers used by RxByte processor.
- Sets up the DMA engines and initializes the SDP scopes.
- Enables the SDPs.

7.14.2.2 Input Thread

The input thread monitors the input queues and processes received descriptors differently depending on whether segmentation or reassembly is required. In both cases, the input thread does the following:

- Waits for queue status to indicate its queue is non-empty.
- De-queues the ML-PPP packet descriptor from the QMU queue.
- Maps the TDM port number from the descriptor to a ML-PPP bundle via the port to ML-PPP bundle map data structure. This structure is described in section 7.14.3.3

For segmentation, the input thread de-queues IP or NCP packets and en-queues descriptors to the soft scheduler. Specifically, it does the following:

- En-queues descriptor to MC queue based on bundle and class from descriptor. The MC queues are soft queues as described in Section 12.2
- Adds the MC queue to the active queue list for the bundle.
- Adds the bundle to the active queue list.

For reassembly, the input thread de-queues ML-PPP fragments, possibly encapsulated in HDLC or PPP and sends them to the SDP for re-circulation. Specifically, it does the following:

- Retrieves the reassembly state information from DMEM, based on ML-PPP bundle and MC class.
- If the sequence number is not in order, then it places the descriptor in a linked list.
- If the sequence number is expected,

- o For non-SOM fragments, launch lookup into ML-PPP remainder table to get remainder bytes.
- o Waits for scope to be available from the SDP.
- o Writes reassembly state information and a subset of the ML-PPP header to merge space.
- o Writes the remainder bytes and length to merge space, if necessary.
- o Waits for the previous payload transfer to complete.
- o Initiates payload transfer from SDRAM to DMEM.
- o Updates reassembly state in DMEM.
- o If linked list has an expected sequence number in it, repeat procedure for that fragment.

7.14.2.3 Scheduler Thread

The scheduler thread services the MC queues for each bundle. It de-queues descriptors and sends buffers to the SDP for segmentation. Specifically, it does the following:

- Gets the next active bundle from the active bundle list.
- Gets the next active class from the active queue list for bundle.
- Uses DRR to determine from which class to send a fragment.
- Retrieves segmentation state for the bundle and class from DMEM.
- If no segmentation in progress, de-queues from the class queue of the ML queue table.
- Waits for scope to be available from the SDP
- Increments sequence number for this bundle and class.
- Determines which link of the bundle should receive the fragment.
- Fills in merge space with ML header.
- Waits for previous payload transfer to complete and frees buffer if it was an EOM.
- Initiates payload transfer from SDRAM to DMEM.
- Updates segmentation state in DMEM.

7.14.2.4 Output Thread

The output thread waits for a receive scope to be available for the SDP and then processes the ML-PPP fragment differently depending on the operation code in extract space. For segmentation, the output thread receives ML-PPP fragments from the SDP and en-queues them to their destination queue. Specifically, it does the following:

- Waits for scope to be available from the SDP
- Initializes a payload transfer from DMEM to SDRAM.
- Waits for the payload transfer to complete.
- Returns the scope to the SDP
- Builds a descriptor and en-queues it to the destination queue based on the port field in extract space.
- The descriptor format is specified in section 7.4.3.3.
- Allocates a new buffer for the next time.

For reassembly, the output thread receives ML-PPP fragments from the SDP and coordinates their transfer to the correct location in SDRAM. When reassembly is complete, the thread en-queues a descriptor to a destination queue based on the protocol of the reassembled frame. Specifically the thread does the following:

- Waits for scope to be available from the SDP
- Updates reassembly state information
- Initializes payload transfer with buffer and offset specified in extract space.
- Waits for the payload transfer to complete.
- Returns the scope to the SDP.
- For EOM fragments, builds a descriptor and en-queues it to the destination queue based on the buffer type in extract space.
- Allocates a new buffer, if necessary, for next time.

Because the RX DMA engine and supported hardware is optimized to transfer 64B chunks of data, sometimes it is necessary to initiate two payload transfers. The first will be from a non-64B aligned offset and used the WrCB to transfer payload bytes from DMEM to SDRAM, once this is complete; the normal transfer using the Rx control block is initiated.

7.14.3 Data Structures

7.14.3.1 Extract Space

When reassembling, RxByte processor writes information about the fragments into extract space for the RC to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	opCode	Port	Index	fragType
4	length		bufType	pad

- opCode – indicates reassembly
- port – port on which fragment was received
- index – concatenation of ML bundle and MC class
- fragType – type of fragment indicated in ML header (SOM, EOM, COM, FOM)
- length – number of payload bytes to write to DMEM (only valid for non-EOM fragments)
- bufType – the buffer type of the reassembled PPP frame (only valid for SOM fragments)
- pad – unused

When segmenting, RxByte processor writes information about fragments to extract space for the RC to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	opCode	port	pad	

- opCode – indicates segmentation
- port – port on which the fragment was received

7.14.3.2 Merge Space

When reassembling, the RC writes information about ML-PPP fragments to merge space for TxByte processor to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	opcode	port	index	fragType
4	length		shortseq	pad
12	Remainder [0..3]			
16	Remainder [4..7]			
20	Remainder [8..11]			
24	remainder[12..14]			pad

- opCode – indicates reassembly
- port – port on which fragment was received
- index – concatenation of ML bundle and MC class
- fragType – type of fragment indicated in ML header (SOM, EOM, COM, FOM)
- length – number of payload bytes in the fragment
- shortSeq – boolean indicating if this fragment uses short sequence numbers
- pad – unused
- remainder[15] – remainder bytes to be sent before payload
- remainderLen – number of remainder byte present in merge space

When reassembling, the RC writes information about ML-PPP fragments to merge space for TxByte processor to use in its processing. The data structure has the following format:

Byte Offset	0	1	2	3
0	opCode	port	length	
4	headerLen		header[0..1]	
8	header[2..5]			
12	header[6..9]			

- opCode – indicates segmentation
- port – port on which packet was received
- length – negated length of payload bytes to send in fragment
- headerLen – negated length of the header to prepend to fragment
- header[10] – header to prepend to fragment

7.14.3.3 MlPppPortBundleMap

This data structure is an array of bytes. The length of the array is equal to the number of TDM ports in the application. The index into the array is the TDM port number and the value of each array element is the bundle number to which the TDM port belongs. A value of 0xff indicates the port is not a member of a ML-PPP bundle. This structure is initialized and modified by the host. There can be at most 256 MLPPP bundles.

7.14.3.4 MlPppBundlePortList

This data structure is an array of bytes. The length of the array is equal to the number of TDM ports in the application. The elements of the array are the list of TDM ports in each bundle. For example array elements 0 through 7 might be the channels belonging to ML-PPP bundle 0, elements 8 through 11 might be the channels belonging to ML-PPP bundle 1, and so on. The index for each bundle is stored in the MlPppBundleParams structure. This structure is initialized and modified by the host.

7.14.3.5 MlPppQuantums

This data structure is a two dimensional array of bytes. One dimension is the maximum number of ML-PPP bundles in the application and the other is the number of MC classes in the application. The elements of the array are the quantums used by the DRR algorithm executed in the scheduler thread. This structure is initialized and modified by the host.

7.14.3.6 MlPppBundleParams

This data structure is an array with the following format:

Byte Offset	0	1	2	3
0	flags	numClasses	mrru	
4	firstPort		numPorts	pad

- flags – indicates valid bundle and which NCP are up
- numClasses – number of MC classes supported by the bundle
- mrru – maximum received reconstructed unit for the bundle
- firstPort – index into the MlPppBundlePortList for the first port of the bundle
- numPorts – number of ports that are in the bundle
- pad – unused

The index into the array is the ML-PPP bundle number. The host initializes and modifies this structure.

7.14.3.7 MlPppActiveList

This structure maintains a list of active entities. One structure maintains a list of active bundles. Another array of structures maintains a list of active class queues (that is, non-empty queues) for each bundle. This structure has the following format:

Byte Offset	0	1	2	3
0	listHi			
4	listLo			
8	map		len	

- listHi – 8 most recent entities added to the active list
- listLo – 8 least recent entities added to the active list
- map – a bitmap indicating which entities are in the list
- len – number of active entities in the list

There may be up to 16 active entities. The entities maintained in the list are numbered 0 to 15. The entity to be serviced next is specified by the least significant nibble of listLo. When an entity is added to the list, it is placed at the nibble specified by len when {listHi, listLo} is considered as a 64-bit word. len is then incremented.

7.14.3.8 MlPppQueueHandles

This data structure is a two dimensional array of soft queue handles as described in Section 12.2. One dimension is the maximum number of ML-PPP bundles in the application and the other is the number of MC classes in the application.

7.14.3.9 MlPppRasState

This data structure is a two dimensional array with the following format:

Byte Offset	0	1	2	3
0	bufHandle			
4	Offset		port	lock_bufType
8	nextSequence			

- bufHandle – reassembly buffer handle
- offset – buffer offset at which next fragment should be placed
- port – port on which fragment was received
- lock _bufType – a bitmap as follows
  - – b7: lock – bit indicating if reassembly is in progress for this bundle and class
  - – b6-0: bufType – enum indicating the type of buffer being reassembled
- next_sequence – a bitmap as follows
  - – b31-24: next – pointer to structure in the linked list
  - – b23-0: sequence – expected sequence number

One dimension is the maximum number of ML-PPP bundles in the application and the other is the number of MC classes in the application.

7.14.3.10 MlPppSegState

This data structure is a two dimensional array with the following format:

Byte Offset	0	1	2	3
0	bufHandle			
4	length		proto	
8	offset		counter	
12	sequence			

- bufHandle – segmentation buffer handle
- length – length of buffer
- proto – PPP protocol field for the buffer type
- offset – buffer offset from where the next fragment should come
- counter – DRR defecit counter
- sequence – next sequence number to transmit

One dimension is the maximum number of ML-PPP bundles in the application and the other is the number of MC classes in the application.

7.14.3.11 MlPppLinkedList

This data structure is an array with the following format:

Byte Offset	0	1	2	3
0	bufHandle			
4	length		port	next
8	fragType_Sequence			

- bufHandle – fragment buffer handle
- length – length of buffer
- port – port on which this fragment was received
- next – pointer to next structure in linked list, 0 if none
- fragType_sequence – a bitmap as follows
    - b31-24: fragType – fragment type (SOM, EOM, COM, FOM)
    - b23-0: sequence – sequence number of fragment

There are a fixed amount of structures that are allocated and de-allocated as elements are added and removed from the linked list. All bundles and classes shared this “heap” of linked list memory.

7.15 AAL1-Tx (CP 14)

CP 14 implements the AAL-1 Tx component. The AAL-1 component uses the transparent mode support in the TDM interface adapter to access the raw TDM timeslots. The TDM CPRCs provide the ATM interface on which TDM circuit traffic is sent as AAL-1 SDUs.

This component supports up to 410 channels of TDM circuits, with each circuit providing a maximum traffic of 256 KBps (the equivalent of all the timeslots in an E1). Therefore, the maximum CES throughput supported is 107MBps.

This component gets transparent TDM chunks from the TDM Rx CPRCs. The AAL-1 Tx module is responsible for creating 48 byte AAL-1 SDUs that will be transmitted in the payload of ATM cells. It consists of two CPRC functions, namely, aal1TxIn and aal1TxOut that run in independent CPRC contexts. The SDP is not used for AAL-1.

TDM chunk sizes do not come in multiples of the AAL-1 structure size and the structure size does not have to be a multiple of the cell payload size of 48 bytes. Structures can begin and end on non-16 byte boundaries in the SDRAM. During cell assembly, this requires AAL-1 Tx to maintain the unaligned portion of the SDU payload in DMEM.

7.15.1 RC

7.15.1.1 AAL1TxIn

The AAL-1 header includes a CRC-3 over the 4 bit CSI-SN field and a parity bit over the 7 bits of CSI-SN-CRC. As there can be only 16 possible 4 bit CSI-SN values, the CRC-3 values are calculated offline and stored in a DMEM lookup table. Similarly, as there can be only 128 combinations of CSI-SN-CRC, a 128 entry parity lookup table is kept in DMEM.

Aal1TxIn maintains a 16 byte aligned, launchPad area of 64 bytes in DMEM. The unaligned bytes and chunk payload are written to the launchPad. When all chunk processing is complete, the launchPad is DMA written to the SDRAM.
To perform its processing, Aal1TxIn does the following things:

- Dequeue a chunk descriptor from the input queue.
- Fetch the Tx state from the state array using the channel id(input port) of the de-queued descriptor. If the tdmSn field in the descriptor does not equal the nextSn field, some chunks have been dropped. Determine the number of chunks dropped and hence the number of TDM data bytes dropped.
- Increment the nextSn field modulo 8.
- If the residual time stamp field is not zero, record it. This will have to be sent on odd numbered AAL1 SDU headers.
- Start a DMA read of the chunk payload into DMEM, by issuing a bsBufferRead call.
- If the offset is zero, a new AAL1 SDU should be started in a new ATM cell. Use the nextAal1Sn field to create the AAL1 header and increment it modulo 8. If nextAal1Sn equals zero, clear the pointerSent field. Copy the header field at the beginning of the launch pad.
- If a new cell is being created, check if the pointer field in the AAL-1 header should be sent on this cell. If nextAal1Sn is an even number (including 0), the pointerSent flag is not set and the structSpill field is less than 48, insert the AAL-1 pointer field with the pointer containing the structSpill value plus one. Use the DMEM based parity table to get the pointer parity bit. If nextAal1Sn is equal to 6 and the structSpill is greater than 48, insert a pointer field with the dummy pointer value of 127. Copy the pointer byte into the launch pad. If the pointer was set in this step set the pointerSent flag.
- If this is not the beginning of a new cell, there can be unaligned bytes from the previous chunk. Copy them into the launch pad.
- If it was determined that certain chunks were lost, insert zeros in the launch pad to account for the number of lost bytes.
- By now the DMA read should have completed. Copy as many bytes of chunk payload as can fit in the 47 byte SDU payload in the launch pad. Update the offset field by the number of bytes copied. If this chunk does not complete the cell, extract the unaligned bytes and copy them into the state table entry.
- Launch a DMA write from the launch pad area into the SDRAM. If a complete cell has been assembled, launch a port table lookup and context switch to aal1TxOut context.
- After transmitting a cell, if there are still some bytes left from the chunk, a new cell has to be started. Update the structSpill field to reflect the number of bytes from the structure that will spill over into the next cell. Perform another DMA read and create a new cell with the remaining chunk bytes.
- Loop to the top of the thread.

7.15.1.2 AAL1TxOut

This thread is responsible for en-queuing a completed ATM cell to its destination. It waits for the results from the port table TLU lookup launched by aal1TxIn. If the RTS is not zero, this context puts it on the SDUs in a cycle, as specified by I.363. It creates a cell descriptor and based on the lookup results en-queues the cell descriptor to the ATM TM CPRC or to TDM Tx CPRC.

7.15.2 Data Structures

The AAL-1 Tx component uses the following data structures to maintain state.

7.15.2.1 Aal1TxState

The following structure is used by aal1Tx to track SDU assembly state in DMEM. An array of these structures, indexed by the channel Id is maintained in DMEM. The offset field gives the position in the buffer, where the next chunk should be written. The number of unaligned bytes is equal to the offset mod 16. These bytes should be appended to the start of the next chunk so that it lines up to a 16-byte boundary. The structure has the following format:

Byte Offset	0	1	2	3
0	sduBufHandle		pointerSent_nextSn	Offset
4	structSpill	unalignedBytes[0:12]		
8	unalignedBytes[3:6]			
12	unalignedBytes[7:10]			
16	unalignedBytes[11:14]			

- sduBufHandle – contains the handle to SDRAM buffer where the payload is being assembled
- nextSn – next sequence number expected for this channel
- offset – the offset within the buffer where the next chunk will be written
- pointerSent_ nextAal1Sn – a bitmap defined as follows
-
    - b 7 :pointerSent – boolean value specifies whether an AAL1 pointer has been sent in this cycle
    - b 6-4 :unused.
- structSpill – field specifies the number of payload bytes before start of the next structure
- unalignedBytes[0..15] – the unaligned bytes to be appended to the start of the next chunk

7.15.2.2 Descriptor information

AAL1 Tx Descriptor is filled by the TDM Rx CPs, has the following format :

Byte Offset	0	1	2	3
0	CellHeader			

4	egressPort	tdmSn	pad

The explanation of each field is as follows :

- cellHeader – the cell header (VPI/VCI) to be added to form ATM AAL1 cell.
- egressPort –TDM egress port through which ATM AAL1 cell is transmitted.
- tdmSn – sequence count value of the transparent TDM chunk.

7.16 AAL-1 RX (CP 15)

CP 15 implements the AAL-1 RX component. The AAL-1 component uses the transparent mode support in the TDM interface adapter to access the raw TDM timeslots. The TDM CPRCs provide the ATM interface on which TDM circuit traffic is sent AAL-1 SDUs.
This component supports up to 410 channels of TDM circuits, with each circuit providing a maximum traffic of 256 KBps (the equivalent of all the timeslots in an E1). Therefore, the maximum CES throughput supported is 107MBps. Maximum number of VCCs that can be supported is 410.

7.16.1 RC

7.16.1.1 AAL1Rx

This module is responsible for receiving ATM cells with AAL-1 payloads and creating TDM chunks out of them for clear channel transport. Aal1Rx maintains a 16 byte aligned launch pad area of 64 bytes in DMEM. The unaligned bytes and the ATM cell payload are written to the launch pad. The launch pad is DMA written to the SDRAM when a complete cell payload has been assembled or when a chunk has been completely processed.

The following sequence of steps are executed by AAL1 Rx:

- De-queue a cell descriptor from the input queue. Read the VC index from the descriptor and fetch the state table entry corresponding to the index.
- Launch a DMA read to fetch the cell payload bytes into DMEM.
- The first byte of the payload is contained in the STF field of the cell descriptor. For AAL1 this field is the AAL-1 header. Extract the CSI and SN fields from this header. Using the DMEM based CRC tables verify that the CRC and the parity are correct.
- If the parity or CRC are incorrect, check if they can be corrected as specified in I.363.1. If they cannot, discard the cell and loop to the top of the thread.
- Validate the SN field. If it is invalid, drop the cell and loop to the top of the thread.
- If the offset field in the state entry is zero, a new chunk needs to be started.
- The DMA read should have been completed by now. If a new chunk is not being started, copy the unaligned bytes into the launch pad area. Copy as many bytes from payload as necessary to fill up the chunk. Create a chunk descriptor and transmit the chunk to the TDM Tx CPRC.
- If the SN field is an odd number, extract the CSI bit and accumulate it in the RTS. When SN is equal to 7, check the accumulated RTS. If it is non zero, send it out to the TDM Tx CPRC with the next chunk descriptor and clear it. The TDM Tx CPRC forwards it to the MT-21, which applies the appropriate timing correction.

- If there are more bytes left in AAL-1 SDU payload, start a new chunk and copy the remaining bytes. If the chunk is completed, transmit it to the TDM Tx CPRC. Repeat this till the entire AAL-1 SDU payload has been exhausted.
- Loop to the top of the thread.

7.16.2 Data Structures

The AAL-1 RX component uses the following data structures to maintain state.

7.16.2.1 Aal1RxState

The following structure is used to track the conversion from AAL-1 SDU payloads to chunks. An array of Aal1RxState structures is kept in DMEM. This array contains one entry per AAL-1 VCC and is indexed by the VC index from the ATM VC table.

Byte Offset	0	1	2	3
0	chunkBufHandle		offset	nextChunkSn
4	nextAAL1Sn	unalignedBytes[0:2]		
8	unalignedBytes[3:6]			
12	unalignedBytes[7:10]			
16	unalignedBytes[11:14]			

- chunkBufHandle – SDRAM buffer where the chunk is being accumulated
- offset – offset in the chunk, where the next cell payload should be written
- nextChunkSn – sequence number to be used on next chunk
- nextAAL1Sn – sequence number expected on the next AAL-1 SDU on this VC
- unalignedBytes[15] – the unaligned bytes to be appended to the start of the next cell

7.16.2.2 Descriptor information

AAL1 Rx Descriptor is filled by the TDM Rx CPs has the following format :

Byte Offset	0	1	2	3
0	vcIndex		egressPort	
4	aal1Hdr	pad		

The explanation of each field is as follows:

- vcIndex – the index associated with egress VPI/VCI
- egressPort –TDM egress port through which transparent chunks are transmitted
- aal1Hdr –AAL1 header extracted from the ATM AAL1 cell

7.17 Fabric Port

The fabric port implements the Utopia Level 2 interface. This port handles only ATM cells. Cells received on the interface are typically forwarded to other ATM processing blocks based on VPI/VCI lookup. The Fabric Port sends cells received from other ATM processing blocks to the UL-2 interface.

7.17.1 FpTx

The FpTx component de-queues descriptors and sends utopia cells to the UL-2 adapter. The micro-code uses information in the descriptor to construct the ATM header then transmit it. The Hardware then transmits the buffer contents specified by the buffer handle in the descriptor. Specifically, FpTx does the following:

- Sends the 4-byte ATM header provided by the descriptor
- Asserts the EOM bit in the PTI byte, if necessary
- Sends the 48-bytes of payload
- Switches to the next scope

FpTx microcode may also assist with segmentation. FpTx assumes that the PDU length is a multiple of 48 bytes, which will be the case for any PDU sent by the segmentation cluster. If the PTI field indicates EOM, then FpTx will segment the PDU and mark the PTI EOM bit appropriately in the last cell. Otherwise, FpTx sends the cell with the PTI indicating no EOM. FpTx receives information through merge space about the PDU to be sent.

7.17.2 FpRx

The FpRx component receives ATM-like cells from the UL-2 adapter, validates them, and forwards descriptors to the next ATM processing block. Every cell is considered an independent PDU and marked as FOM. The HW splits the cell into header (8 bytes) and payload (48 bytes). The lower four bytes of the header overlap are the same as the first four bytes of the payload. The micro-code parses the ATM header and launches a lookup of the VPI/VCI. If the lookup fails, the cell is dropped by the Fabric Port hardware.Otherwise, the ATM header, the STF header and lookup response data are copied into the descriptor, the HW moves the payload into a buffer, and the descriptor is placed in the queue specified by the lookup response. In particular, FpRx does the following:

- Writes the ATM header to extract space
- Writes the VPI/VCI into the TLU Tx message slot
- Sets the flow ID to a constant (0)
- Sets the segment type to first and only message (FOM)
- Sets the PDU size to a constant (52)
- Launches the VPI/VCI lookup
- Switches to the next scope

Because the FpRx is not able to preserve much state information, it is incapable of determining when the first cell of an AAL-5 PDU is received. For this reason, the FP is not able to perform the reassembly operation and so each cell is treated independently. FpRx uses extract space to store information about the incoming segment.

7.17.3 DBE

The descriptor build engine (DBE) component uses the data put into extract space by the FpRx and the results of the TLU to build a descriptor that will be queued to the next block for further processing. The DBE microcode fills in the descriptor data structure when the TLU signals that the lookup has completed. If the lookup fails the DBE will drop the incoming cell.

7.17.4 Data Structures

7.17.4.1 Merge space

The FpTx hardware copies the descriptor data from the payload bus to merge space after a de-queue operation. The data structure has the following format:

Byte Offset	0	1	2	3
0	bufHandle			
4	Length		port_bufType	
8	CellHeader			
12	Appdata			

- bufHandle – the handle to the buffer containing cell payload
- length – length of the payload in the buffer
- port_bufType – unused by the FP
- cellHeader – the cellHeader to be prepended to outgoing cells, PTI indicates EOM if segmentation is desired
- appData – unused by the FP

7.17.4.2 Extract Space

The FpRx hardware fills in extract space before passing control to the DBE. Extract space has the following format:

Byte Offset	0	1	2	3
0	cellHeader			

cellHeader – the cell header from the received cell

7.18 Table Lookup Unit

The table lookup unit provides lookup table management. The TLU stores its table information in external SRAM that is managed by TLU's SRAM controller. The application creates multiple tables in the TLU for data path forwarding and state memory support. The tables are summarized below and described in more detail in subsequent sections.

Table Name	Table Type	Table ID	Key Size(b)	Entry Size (B)
Port Table	Data	0	11	8
MLPPP Remainder Table	Data	1	10	16
IPV4 Route Table	LPM	2	32	16
ATM VC Table	HTK	4	48	16
Reassembly CRC Table	Data	6	11	4
Diffserv Flow Table	HTK	7	112	8
MPLS Lookup Table	Data	10	16	16

FR DLCI Table	HTK	11	21	16

The table fields and lookup key construction are defined in the appropriate sections where this lookup is launched.

7.18.1 Port Table

When the ingress processor launches a lookup to make a forwarding decision, the response usually returns the port number and layer 2 addresses. In order to map the port to a queue and determine if processing other than simple forwarding is needed, the ingress process must also launch a lookup in the port table. The port table is a data table with a 32-bit key, only 12 of which are significant. The key (or index) is equal to the port number of interest.

Each entry of the table has the following format:

Byte Offset	0	1	2	3
0	PortType	Flags	egressQueue	
4	QosQueue		pad	

The description of each field is as follows:

- portType – the type of egress port, ATM,PPP,FR, Transparent chunk
- flags – a bitmap as follows
    - b7: Valid – flag indicating whether or not the port is valid
    - b6: QoS – flag indicating whether QoS must be performed on this port
    - b4-0: reserved for future use
- egressQueue – the egress queue of the packet or cell
- qosQueue – the queue of the QoS processor for this packet

7.18.2 MLPPP REM Table

When ML-PPP reassembly takes place, buffer writes cannot always be accomplished in multiples of 16B. In this case, the bytes left over that do not fill a 16B DMA line are stored in the ML-PPP remainder table along with a count. These bytes will be fetched and proceed the data from the next fragment being reassembled.
Each entry of the table has the following format:

Byte Offset	0	1	2	3
0	Remainder [0..3]			
4	Remainder [4..7]			
8	Remainder [8..11]			
12	Remainder [12..14]			remainderLen

The description of each field is as follows:

- remainder – the remaining bytes that could not fill a 16B line
- remainderLen – the number of remaining bytes in the entry

7.18.3 IP V4 Routing Table

The IPv4 routing table is a longest prefix match table with a 32-bit key. The key is equal to the IP destination address.

Each entry of the table has the following format:

Byte Offset	0	1	2	3
0	Port		Type	MaskBits
4	AppData			
8	AppData			
12	Pad			

The description of each field is as follows:

- port – the egress port number
- type – the type of route entry (internal, local, gateway, etc.), unused
- maskBits – the number of significant bytes in the key
- appData – application specific data, usually contains L2 address, see below
- pad – unused

The appData field can have different interpretations depending on the egress. For example, an ATM egress would have a VPI/VCI.

The ATM appData field has the following format:

Byte Offset	0	1	2	3
0	EgressCellHeader			
4	EgressQueue		pad	

The description of each field is as follows:

- EgressCellHeader – the cell header (VPI/VCI) to use at the egress
- EgressQueue - TrafficQueue mapped to Q-3 for applying ATM TM
- pad – unused

The MPLS appData field has the following format:

Byte Offset	0	1	2	3
0	LabelPush		LabelSwap	
4	Action	Hop_count	EgressBufType	Pad

The description of each field is as follows:

- labelSwap – Label to be swapped.
- labelPush - Label to be added
- action – MPLS action to be performed. The supported MPLS actions are listed in section 7.18.8
- hopCount – count to be decremented in TTL of the shim header
- EgressBufType – a bitmask defined as follows:
    - b 15-5: port – the output port on which this datagram to be transmitted.
    - b 4-0: bufType – the type of buffer at the egress being recirculated and the egress port type.
- Pad – unused

The FR appData field has the following format:

Byte Offset	**0**	**1**	**2**	**3**
0	egressFrHeader			
4	Pad			

The description of each field is as follows:

- egressFrHeader – the FR header to use at the egress.
- Pad – unused

7.18.4 ATM VC Table

Every ATM cell that enters the application has its VPI/VCI looked up in the ATM VC table to determine if the cell is on a valid VC and how to process it. The ATM VPI/VCI table is a HTK table with a 48-bit key. The key is constructed as follows:

47 44	**43 36**	**35 20**	**19 16**	**15 0**
0	VPI	VCI	0	Port

Each entry of the table has the following format:

Byte Offset	**0**	**1**	**2**	**3**
0	Flags_egressPort		VcIndex	
4	EgressCellHeader			
8	EgressQueue		Port_bufType	
12	DestQueue		OamPM	pad

The description of each field is as follows:

- egressCellHeader – the ATM cell header (not including HEC) to be applied to the cell at the egress
- vcIndex – the VC index used by AAL-5 SARs to index their state tables
- flags_egressPort – a bitmap as follows:
    - b15: AAL1 VC – flag indicating the VC is an AAL-1 type VC
    - b14: AAL5 VC – flag indicating the VC is an AAL-5 type VC
    - b13: MPLS VC – flag indicating the VC is an MPLS VC.
    - b12-11: reserved for future use
    - b10-0: egressPort – the egress port from which the cell must exit
- egressQueue – the egress queue of the ATM cell, this is necessary for the FP which can launch only one lookup. This also represents the trafficQueue, to be used if QoS is enabled for this egress port
- port_bufType – the egress port and buffer type packaged in a way the FP can use
- destQueue – the queue of the next ATM processing block, this is necessary for the FP which cannot act on TLU response data
- oamPm – MSB indicates OAM processing required on this VC, remaining bits are index into local OAM table

- pad – unused

For MPLS switched ATM AAL-5 PDUs, following table entry is used:

Byte Offset	0	1	2	3
0	Flags_egressPort		VcIndex	
4	LabelPush		LabelSwap	
8	EgressQueue		HopCount_action	egress_bufType
12	appData			

The description of MPLS related fields are as follows:

- LabelPush – Label to be swapped
- LabelSwap – Label to be added
- HopCount_action – One Byte containing MPLS action to be performed and the hop count to be decremented in TTL of the shim header. The supported MPLS actions are listed in section 7.18.8
- AppData – Application Specific data. If the egress interface is FR, it is FR header. In case of ATM egress interface, it ATM cell header.

7.18.5 RAS CRC Table

The reassembly CRC table stores the partial CRC result calculated during reassembly. The key is the VC index retrieved from the ATM VC table. Each entry of the table has the following format:

Byte Offset	0	1	2	3
0	Pad		CrcLength	
4	CrcPartial			

The description of each field is as follows:

- pad – unused
- crcLength – number of cells whose CRC has been calculated so far
- crcPartial – partial CRC value calculated so far

7.18.6 Diff Serv Flow Table

The DiffServ flow table is a HTK table with a 112-bit key. The key is constructed as follows:

111 80	79 64	63 32	31 16	15 8	7 0
IP dest Addr	L4 Dest Port	Ip Src Addr	L4 Src Port	Proto	Port In

Each entry of the table has the following format:

Byte Offset	0	1	2	3
0	MaskBits	Phb	FlowId	

	QoSQueueoffset	Flags	Pad

The description of each field is as follows:

- maskBits – number of significant bits in the key
- phb – per hob forwarding behavior to apply
- flowId – Classifier Flow ID
- QoSQueueOffset – the offset of the QoS queue from the base queue obtained from port table lookup
- Flags – The drop flags to be applied for this flow.
- pad – unused

7.18.7 FR DLCI Table

FR table is HTK table with 32-bit key. The key is formed by the concatenation of 10-bit DLCI value and 11-bit interface ID. Currently the two-byte address format is used to for DLCI (10-bit).

Bits: 20 **10**	**Bits: 9** **0**
Interface ID	DLCI

The format of each entry in the FR table

Byte Offset	**0**	**1**	**2**	**3**
0	Label (both push and swap labels)			
4	Action	HopCount	port_bufType	
8	EgressQueue		flags_egressPort	

The explanations for the above-mentioned fields will be as follows

- Label - specifies the labels that are to swapped and pushed for MPLS. In case of FR switching the lower 16-bits specify the DLCI value.
- Action - specifies the MPLS action to be performed. The actions that are supported:

```
typedef enum {
    MPLS_ACTION_NONE,
        MPLS_ACTION_POP,
        MPLS_ACTION_SWAP,
        MPLS_ACTION_SWAP_PUSH,
        MPLS_ACTION_FORWARD,
        MPLS_ACTION_ADD,
        MPLS_ACTION_PUSH_PUSH,
        MPLS_ACTION_POP_IPv4_LOOKUP
} mplsActions;
```

- HopCount -Count to be decremented in TTL of the shim header
- Port_bufType -a bitmask is defined as follows:
    - b15-5: port – the output port on which this datagram to be transmitted.

- b4-0: bufType – the type of buffer being re-circulated and the egress port type (could be BT_MPLS_PPP, BT_MPLS_ATM, BT_MPLS_FR, BT_MPLS_IPv4)

- EgressQueue -specifies the final queue
- Flags_egressPort - specifies the next queue that the packet needs to be sent.

7.18.8 MPLS Table

MPLS table maps the input MPLS label to the MPLS action to be performed in MPLS CP. This is a data table with a 32-bit key, only 16 of which are significant. MPLS label is the key (or index) to this table. It is used by MPLS packet(s) over PPP and for pop action (which needs additional label lookup based on the next label in the stack) in the case of MPLS packet over ATM and FR.

Each entry of the table has the following format:

Byte Offset	**0**	**1**	**2**	**3**
0	labelSwap		labelPush	
4	Action	hopCount	port_bufType	
8	egressQueue		Pad	
12	appHdrData			

- labelSwap – Label to be swapped.
- labelPush - Label to be added.
- action – mpls action to be performed. The following MPLS actions are supported.

```
typedef enum {
   MPLS_ACTION_NONE,
   MPLS_ACTION_POP,
   MPLS_ACTION_SWAP,
   MPLS_ACTION_SWAP_PUSH,
   MPLS_ACTION_FORWARD,
   MPLS_ACTION_ADD,
   MPLS_ACTION_PUSH_PUSH,
   MPLS_ACTION_POP_IPv4_LOOKUP
} mplsActions;
```

Based on the MPLS action the MPLS command (cmds in merge space) is set. There can be more than one POP operations and also POP can combine with PUSH/SWAP actions. For “POP_IPv4_LOOKUP” MPLS action, the command is set for POP and the packet is forwarded to IPv4 module.

- hopCount – count to be decremented in ttl of the shim header
- port_bufType – a bitmask defined as follows:
  - b15-5: port – the output port on which this datagram to be transmitted.

- b4-0: bufType – the type of buffer being recirculated and the egress port type (could be BT_MPLS_PPP, BT_MPLS_ATM, BT_MPLS_FR, BT_MPLS_IPv4)

- egressQueue – the egressQueue of the packet (represents the Traffic Queue if QoS treatment has to be given)
-
- pad - unused
- appHdrData – CellHeader if ATM as the egress port type or DLCI if FR as the egress port type.

7.19 Buffer Management Unit

Application allocated number of buffer pools. These pools are allocated and initialized by XP. List of pools are:

Buffer Pool Owner	**Number of pools**	**Number of buffers**	**Buffer Size**	**Buffer Use**
TDM	1	1024	4096	IP datagrams, PPP packets, ML-PPP fragments
TDM	1	512	64	ATM cells
IMA	1	128	64	ATM cells
ML-PPP	1	256	256	ML-PPP class
ML-PPP	1	256	2048	ML-PPP reassembly buffer
MPLS	1	256	2048	
Segmentation	1	384	64	ATM cells from AAL-5 segmentation
Reassembly	1	384	2048	IP datagrams from AAL-5 reassembly
AAL –1 Rx	1	776	64	TDM chunks
AAL – 1 Tx	1	776	64	ATM cells
UL-2	1	1024	64	ATM cells
Host	1	128	2048	IP datagrams, ATM cells

There are several formats for the data in a BMU buffer. Various formats support communication between different application components. The formats are specified by an enumeration that is typically included in the buffer descriptor. The enumeration and a descriptor of each buffer type is listed:

Buffer Type	**Buffer Description**

BT_UNKNOWN	Contents of the buffer are unknown
BT_IPv4	IPv4 datagram
BT_MLPPP	ML-PPP fragment including ML-PPP header, but no PPP or HDLC encapsulation
BT_PPP	PPP encapsulated data packet
BT_HDLC	HDLC encapsulated PPP frame
BT_ATM	ATM cell payload; length will be 48
BT_IMA_CP	IMA control protocol payload; length will be 48
BT_TDM_TRANSPARENT	Transparent TDM data used with AAL-1
BT_FR	FrameRelay data
BT_MPLS	MPLS data
BT_MPLS_PPP	MPLS switched AAL-5 SDUs with egress port as PPP interface
BT_MPLS_ATM	MPLS switched AAL-5 SDUs with egress port as PPP interface
BT_MPLS_Ipv4	MPLS switched datagrams with IP egress processing required
BT_MPLS_FR	MPLS switched AAL-5 SDUs with egress port as FR interface
BT_LAST	Place holder to define limit of buffer type enumeration

7.20 Queue Management Unit

The information provided in this section is the mapping of queues, which are used by different Channel processors. The QMU is configured to use 32 byte descriptors. The number of Queues allocated to each application block will be as follows:

Queue owner	Number of queues	Queue use
TDM0	32	TDM Tx processing.
TDM1	32	TDM Tx processing.
TDM Recirculation	2	Recirculation of PPP and FR frames.
IMA0	1	IMA Rx processing.
IMA1	1	IMA Rx processing.
IP	1	IP forwarding assistance for TDM channels.
Segmentation	1	AAL-5 segmentation.
Reassembly	1	AAL-5 reassembly.
IP QoS	16	IP Qos for the TDM ports.
FR	1	FR Tx processing.
MPLS	1	MPLS Tx processing.
ML-PPP	1	ML-PPP segmentation and reassembly.
AAL-1	1	AAL-1 Tx processing.
XP	5	ATM control and host.
FP (UL-2)	32	UL-2 egress.

Communications between the afore mentioned application blocks running on different Channel processors is accomplished by the source CP en-queuing a descriptor to the queue associated with the destination CP. All the QMU queues, including the inter module communication queues are mapped to Q-3 VOPs. Channel processors use extended en-queue mechanism for enqueuing packets to Q-3.

7.21 Q-3 configurations for CPs, XP and FP

7.21.1 XP Initialization

XP initialization component does the following:

- Initializes the system services. Queuing services will be initialized in external mode using qsExtendedInitialize() with descriptor size 32 bytes.
- Waits for the host message for completion of Q-3 TMC configuration.
- Configures the 128 VOPs (5 for XP, 32 for FP and remaining for CPs) using qsQueueCreate() and qsQueueConfig() functions. The VOPs will denote the QMU queues for dequeing the descriptor by other C-3e components (XP, FP and CPs). For information about number of VOPs for each component see the queue assignment information in Section 7.20
- Performs other initializations as mentioned in Section 7.1.1.

7.21.2 Host Configurations

Host is responsible to perform the Q-3 TMC configurations as shown in figure 8. It configures the following parameters in the Q-3 map.

- The total number of QMU queues assigned for CPs, XP and FP are 128 (See queue assignment in Section 7.20). One QMU queue will be mapped to one Q-3 traffic queue. So total number of Q-3 traffic queues to be configured will be 128.
- One Level2 scheduler having one input leg will be configured for each traffic queue. So it will need 128 level2 schedulers for normal forwarding path in Q-3.
- Since Q-3 is used to forward the descriptor from traffic queue all the way to C-3e QMU via Q-3 hierarchy, there will be no discard path to be configured.
- One level1 scheduler having 128 input legs will be required. Max number of input legs in level1 scheduler will be 1K.
- At the top level, it is mandatory to have one level0 scheduler in Q-3 hierarchy. It will be configured to have one input leg.
- Number of VOPs to be configured will depend on the number of QMU queues. So 128 VOPs will be configured.

The steps for host configuration starting from top to bottom in Q-3 hierarchy (i.e. from level 0 scheduler to traffic queues) are described as follows.

- Initializes the Q-3 TMC using qsTmcInitialize ().
- Creates one level0 RR scheduler with one input using qsTmcSchedCreate ().
- Creates parent buffer pool and buffer pool associated with it using qsTmcBufferPoolCreate(). These buffer pools will be used by traffic queues.
- Creates one level1 RR scheduler with 128 inputs. This scheduler will feed the level0 RR scheduler.

- Creates the 128-level2 schedulers each having one input legs that will receive the input descriptor from traffic queues configured for CPs, XP and FP. These schedulers will feed the level1 RR scheduler. The total number of level2 schedulers to be supported is 18K.
- Creates 128 traffic queues that will pass the traffic to the Q-3 hierarchy. Total number of traffic queues to be supported is 64K.
- Creates the 128 VOPs using qsTmcVopCreate(). These VOPs are mapped to 128 QMU queues of C-3e. CPs, XP and FP in C-3e will use these VOPs to dequeue the traffic. The total number of VOPs to be supported is 512.
- Enables the Q-3 configuration map using qsTmcEnqueueEnable().
- Communicates with XP to indicate that Q-3 configuration is done.

7.21.3 RC

Each component's RC will enqueue the descriptor to its configured Traffic queue using qsEnqueueExt() and dequeue it from its configured VOP using qsDequeue().

7.21.4 Assumptions

- As per "Functionality comparison document between old Q-5 and projected Q-5 TMC FPGA", new Q-5 can have up to 256 discard configurations. It is assumed that discard configuration refers to discard block. So two discard blocks (token bucket discard blocks and RED discard block) will be configured for each of 8K traffic flows having one discard queue for IP QoS. Similarly one token bucket discard block will be configured for all 8K traffic flows for ATM TM.

- 3 Levels of scheduler Level 0, Level 1 and Level 2, each supporting SP, RR and WFQ algorithms will be supported.

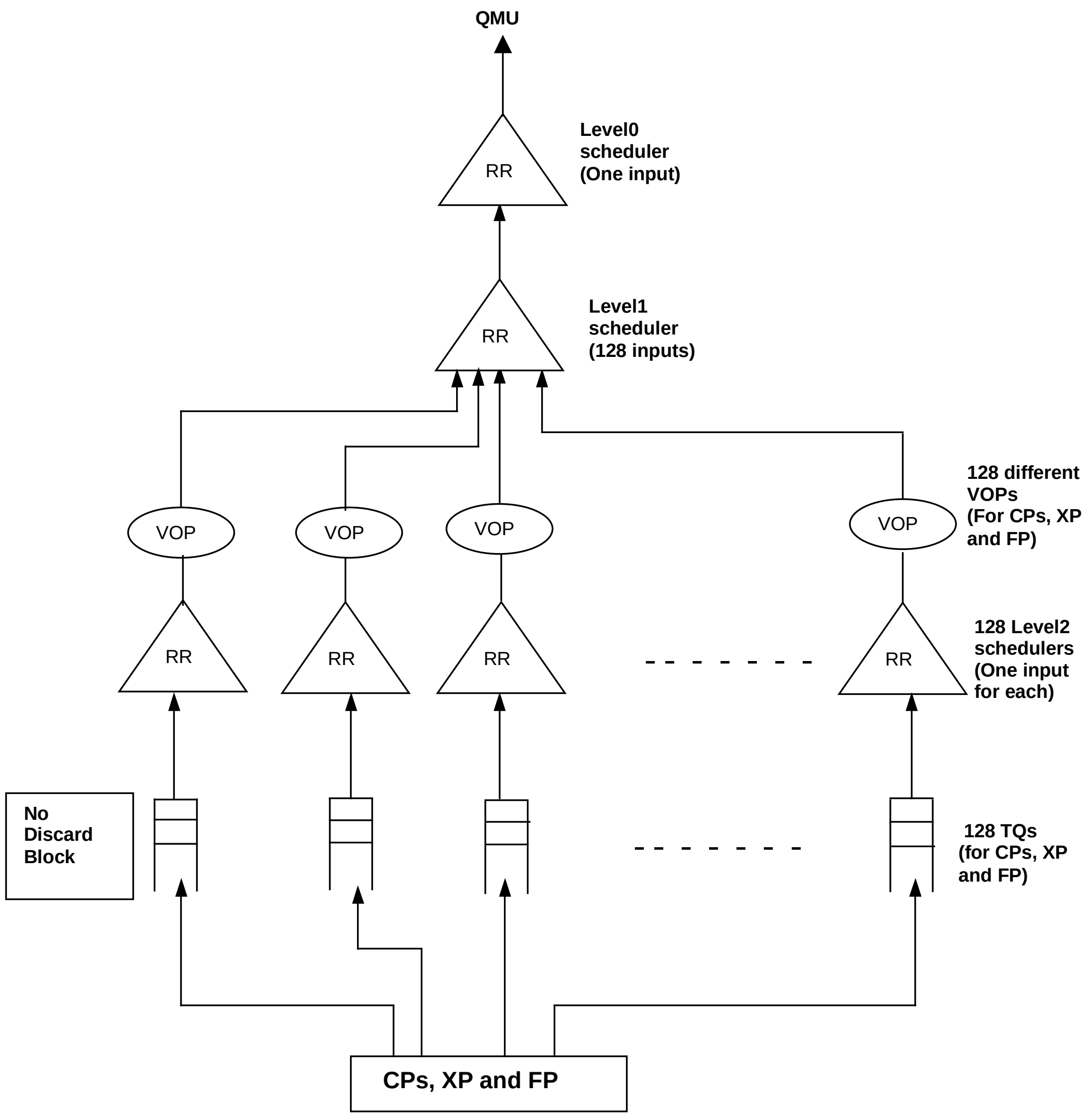
QMU
RR
Level0 scheduler (One input)
RR
Level1 scheduler (128 inputs)
VOP
VOP
VOP
VOP
128 different VOPs (For CPs, XP and FP)
RR
RR
RR
RR
128 Level2 schedulers (One input for each)
No Discard Block
128 TQs (for CPs, XP and FP)
CPs, XP and FP

Figure 8 - Q-3 configuration map for CPs, XP and FP

7.22 ATM TM

ATM traffic management is supported using Q-3 Traffic Management Coprocessor (TMC).

Assumption:
Application supports maximum of 4 VCs per channel.
Q-3 supports the following features:

- Single Leaky Bucket policing per traffic flow.

- 3 levels of scheduler Level 0, Level 1 and Level 2 supporting SP,RR and WFQ algorithms.

7.22.1 Initialization of TMC

XP initialization:
In C3e to support external Queuing (i.e Q-3) Queuing services are initialized to support extended mode queuing. This is achieved by calling qsExtendedInitialize function in XP with descriptor size as 32 and mode as 2 to represent extended mode.

Host initialization:
TMC is initialized by host by calling qsTmcInitialize function.

Enabling Process:
The host enables TMC by calling qsTmcEnable function after configuring TMC. XP enables QMU by calling qsEnable() call after host enables TMC.

For more detailed information refer Section 7.21

7.22.2 Host Configuration

Host does the Q-3 TMC configuration as shown in figure 10 for ATM TM. It configures following parameters in the Q-3 map.

- For each TDM channel, 4 Traffic queues are configured (1 for each VC's). Since 2K channels are supported in this application,
  Total number of TrafficQueues for ATM TM = 2K * 4 = 8K
- Single Leaky Bucket algorithm is applied on a per traffic queue basis and 1 Discard Traffic Queue is associated with all this discard block having VOP=0 (Allowing XP to read the discarded packets)
- For each channel, one Level 2 scheduler (WFQ) is configured with 4 inputs (4 VCs Traffic Queues of a channel)
  Total number of Level 2 Schedulers (WFQ) for ATM TM = 2K
- 1 RR Level 2 scheduler is configured for Discard queue.
- VOP is defined at level 2 leg. One VOP for every TDM T1/E1 interface. The channels configured for first TDM T1/E1 interface will have VOP corresponding to the first TDM T1/E1's VOP and similarly for other channels
  Total number of VOPs = 64 (1 for every TDM T1/E1 interface) + 1 (for Discard Queue)
- Level 1 scheduler is configured for RR with each supporting 1K inputs
  Total number of Level 1 Schedulers (RR) for ATM TM = 2
- Level 0 scheduler is configured for RR.

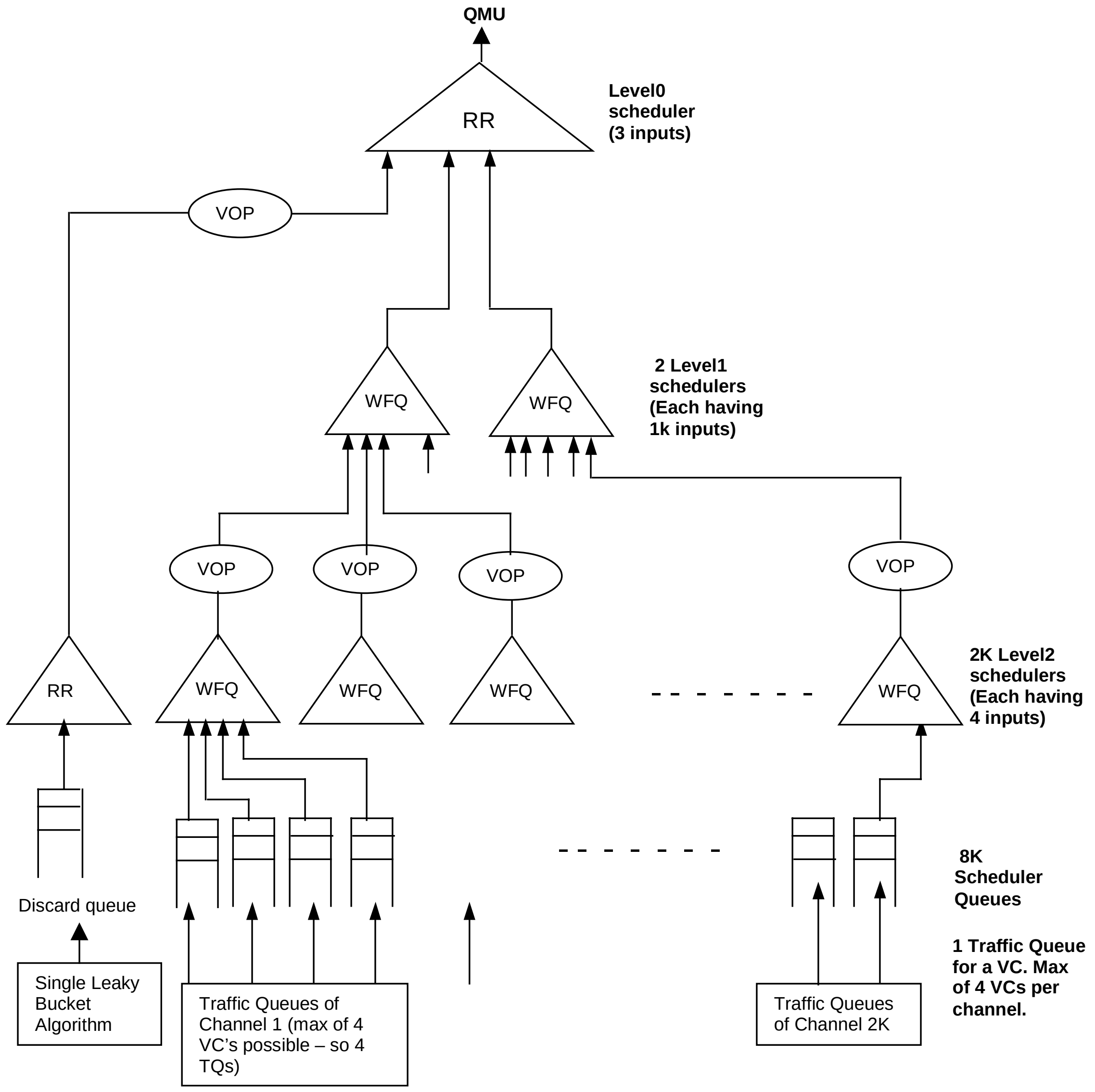
QMU
RR
Level0 scheduler (3 inputs)
VOP
WFQ
WFQ
2 Level1 schedulers (Each having 1k inputs)
VOP
VOP
VOP
VOP
RR
WFQ
WFQ
WFQ
WFQ
2K Level2 schedulers (Each having 4 inputs)
8K Scheduler Queues
1 Traffic Queue for a VC. Max of 4 VCs per channel.
Discard queue
Single Leaky Bucket Algorithm
Traffic Queues of Channel 1 (max of 4 VC's possible – so 4 TQs)
Traffic Queues of Channel 2K

Figure 9 Q-3 ATM TM Traffic flow

7.22.3 ATM TM Traffic Flow

7.22.3.1 Input

AAL-5 segmentation module / AAL-1 Tx module / Tdm Rx module are the modules that sends the descriptor to ATM TM Traffic Queue. These modules perform PortTableLookup. Based on the PortTableLookup response, the corresponding module sends the descriptor to corresponding ATM TM (Traffic Queue) queue if QoS is supported for that port.

Input to Q-3 for ATM TM from AAL – 5 segmentation module:
AAL-5 segmentation module receives data from IP/MPLS module. IP/MPLS module provides following fields corresponding to ATM specific information in the descriptor:

- egressCellHeader – the cell header (VPI/VCI) for ATM encapsulation
- egressQueue – Traffic queue (mapped to Q-3) for applying QoS

AAL-5 segmentation module sends the descriptor to this Traffic Queue if QoS is enabled for that port (based on the port lookup) otherwise it sends the descriptor to whatever egressQueue assigned for that port (obtained from the PortTableLookup response).

Input to Q-3 for ATM TM from AAL-1 Tx:
AAL-1 Tx module does PortTableLookup. Based on the Port table lookup response, it gets the VPI/VCI pair and the Traffic Queue information apart from other information. AAL-1 Tx module sends the descriptor to the TraffcQueue if QoS is enabled for that port otherwise it sends the descriptor to whatever egressQueue assigned for that port (got from the PortTableLookup response).

Input to Q-3 for ATM TM from TDM Rx
TDM Rx identifies the packet type as ATM/PPP/FR. After identifying that the packet is ATM, VC table lookup is launched. It gets the TrafficQueue information from VC table lookup response apart from other information. If the packet has to be ATM switched, it does a portable lookup. It sends the descriptor to Traffic Queue if QoS is enabled for that port (Information got from PortTableLookup response) otherwise it sends the descriptor to whatever egressQueue(non-QoS queue) assigned for that port (got from the PortTableLookup response).

For more detailed information refer appropriate modules (IP,MPLS, ATM AAL-5 segmentation, AAL-1 Tx and TDM Rx).

7.22.3.2 Policing

GCRA is applied to ATM TM traffic queues on a per traffic flow basis. It's mapped to Single Leaky Bucket algorithm. Configurable parameters are:

- Limit; – Leaky bucket 1 limit in nanoseconds.
- Increment; – Leaky bucket 1 Increment in nanoseconds
- Support for Early packet discard

One Discard queue is associated for all the ATM TM traffic queues and it is assigned VOP= 0 to allow XP to read the discarded packets.

7.22.3.3 Scheduling

There are three levels of Schedulers in Q-3: Level 2, Level 1 and level 0. For ATM TM in Level 2, WFQ schedulers (one for each TDM channel) are used based on priority values. Each VC Traffic queue can be configured for CBR/rtVBR/nrtVBR/UBR based on the priority value set. A TDM channel can have maximum of 4 VCs.The priority value set for these VC traffic queues are:

VCs Traffic Queue configured for CBR – Highest priority

VCs Traffic Queue configured for rtVBR
VCs Traffic Queue configured for nrtVBR
VCs Traffic Queue configured for UBR – Lowest priority

If all the VCs are configured for the same type then equal priority is set for all the VCs Traffic Queues of a channel.
VOP is associated to each TDM channel's queue at level 2. It corresponds to TDM T1/E1 interface queue of QMU. Then from RR Level 1 scheduler it goes to RR Level 0 scheduler and then to C3e's QMU queue.

7.22.4 Issues/Enhancements

Issue

- Lack of Q-3 specification.
- Traffic Management done only for egress port.
- Currently TrafficQueue mapping for a VC is through either IP's destination address (part of IP lookup)/ MPLS label (part of MPLS lookup)/ egress port (part of port lookup)/VPI- VCI (part of VC lookup)

8 HOST PROCESSOR ARCHITECTURE

The MSA Line Card host component uses an object oriented (OO) design .It is layered on top of the host services layer and utilizes the provided host services API's wherever possible when interacting with the NP. Direct calls to NP driver functions may be necessary for implementation of certain functions.

Wherever applicable, the host components use the OS abstraction layer (OSAL) services for timers, queues, tasks, threads, etc. to allow porting to other RTOS's and also to support native host simulation for development and regression testing.

Host provide the implementation of PPP Link control protocol(as per RFC1661) and IPv4 Network control protocol (as per RFC1332).The processing code is derived from the publicly available Linux PPP driver (pppd)version 2.4.1 which supports multi-link PPP.

The following activities performed at Host:

- Creation and Initialization of TLU tables.
- Creation of pipes needed for packet I/O and PPP LCP/NCP
- Initializes all peripheral hardware (C-Port Family TDM Channel Adapter, C-Port Family UL2
  Interface Adapter) and sets a default configuration.
- Initializes any required on-chip structures in appropriate DMEM.
- Provides needed info to chip code via HCA (for example, tableID's).
- Spawns all necessary host tasks (receive, PppMgr, statistics, Command Console).

- Configuration and statistics gathering of TLU Tables.
- Configuration and statistics gathering of Link /channel (PPP/ATM/FR)
- Configuration and statistics gathering of ports (TDM/FP)

9 HOST PACKET I/O

Processes running on the host can send and receive packets/cells via the NP using the provided Host services. These services interact with code running on the NP that performs the actual packet/cell transfer between the host and the NP and in the case of transmission perform the actual queuing of the packet to the appropriate output port.

9.1 Resources

These resources are used by the host to perform packet input/output.

- Bi-directional DMA Pipe between host and NP
- Dedicated host transmit BMU buffer pool

9.2 Packet Reception

This section describes packet flow from the ingress, through the NP, to the host.

9.2.1 Network Processor

Traffic intended for the host is processed identically to traffic destined for forwarding to other NP ports except the QMU descriptor for the host packet is placed in a specially designated "host RX" queue by the receiving CP. This could either be as the normal result of a TLU lookup or as the result of the CP matching against a certain field within the packet.

The "host RX" queue is serviced by the XP. When the XP detects the queue is non-empty, it dequeues the descriptor and from the data contained within it, it creates a structure that it sends to the host to inform it of a received packet via the host services Pipe mechanism. This structure contains the packet type, length, and BMU BufferHandle of the data.

9.2.2 Host

A dedicated receive task on the host checks the status of the DMA pipe and upon finding an entry, sets up a DMA operation on the NP to move the data from the NP's BMU buffer to a buffer residing in host memory. When the DMA operation completes, the receive task upcalls based on the packet type. The host also initiates a BMU buffer free operation for the packet's buffer on the NP using the doXpRequest mechanism (currently a direct call to the dcpMgr object).

9.3 Packet Transmission

This section describes packet flow from the host, through the NP, and to the egress.

9.3.1 Host

A process running on the host that wants to transmit a packet calls the sendPacket () function with a buffer pointer, buffer length, port handle, and packet type. The sendPacket function will initiate a DMA operation (controlled by the NP) to move the packet from host memory to a BMU buffer allocated from the host's dedicated transmit pool. When the DMA has completed, the host will place a structure describing the packet (currently the same one used on packet reception) in the DMA pipe to inform the NP that there is something to transmit. The host uses the doXpRequest mechanism to interrupt the XP to do the actual notification.

9.3.2 Network Processor

Based on the host structure read from the DMA Pipe, the XP will construct a QMU descriptor for the host packet and then queue it based on the port. The XP will also allocate a new buffer from the host's pool and give it to the host via the response to the doXpRequest transmit command for use by the host's next transmit. The BMU buffer from the current transmit will be returned to the host's pool via the normal buffer freeing following transmission.

10 CONSOLE COMMAND SHELL COMMANDS

This section defines the commands that are available at the console.

10.1 Application Control

This command is used to control the application at startup. Execute this command, after loading the application package.

- `start` – after a packload, initializes host and NP to default settings and releases the XP.

10.2 Table Maintenance and Display

Tables used in MSA application are created and initialized by the Host. For each table in the application, the following operations will be supported:

- `getEntry Table {key}` – gets the entry corresponding to the provided key
- `deleteEntry Table {key}` – deletes the entry corresponding to the provided key
- `setEntry Table {key} {entry info}` – adds/modfies the entry corresponding to the provided key
- `display Table` – displays the entire table
- `flush Table` – flushes the entire table

In each of the above commands, the italicized "Table" part of the command is replaced with the name of the table. The following tables are supported:

- Atm
- Port
- IPv4Route
- IPv4Flow
- MPLS
- FR

Having a command set for each table type allows the "help" facility for each command to show only the required key and entry parameters for that particular table type. The alternate approach of having the table type as a parameter would require that all possible key and entry types be displayed in the "help".

For the case of MPLS entrie(s) maintained in different tables (IPv4Route/ATM/FR), the above said table operations are performed based on the flag field set for MPLS in the entry.

10.3 Link Configuration and Status

These commands allow the individual links (multiple T-1/E-1's) to be configured for line protocol, either "clear channel" operation (default) or channelized. If "clear channel" is selected, the link is then configured for either ATM (default) or PPP or FR . Attempts to configure links not supported by the PIM in use will not be allowed. Attempting to remove a channelized link before all the sub-channels have been removed will not be allowed. Attempting to remove a non-channelized link that is part of an IMA group will not be allowed.The user will be returned an error status if the attempted configuration causes the maximum number of permitted logical channels to be exceeded.

- `configLink {linkIndex} {T1|E1} [Clear|Channelized] [ATM|PPP|FR]` - configures a link
- `removeLink {linkIndex}` – removes the link
- `showLink {linkIndex}` – displays the link configuration

10.4 Channel configuration and Status

These commands configure the individual channels on links that have been configured for channelized operation for either ATM (default) or PPP or FR operation.

- `configChannel {linkIndex} {channelIndex} {ATM|PPP|FR }`– configure the channel
- `removeChannel {linkIndex} {channelIndex}` – removes the channel
- `showChannel {linkIndex} {channelIndex}` – displays the channel configuration

An error status is returned if the attempted configuration causes the maximum number of permitted logical channels to be exceeded.

10.5 IMA Configuration and Status

These commands allow IMA bundles to be configured on clear channel links that have been configured for ATM.

- `createImaGroup {baseLinkIndex}` – creates the IMA group with the first link
- `addImaLink {groupIndex} {linkIndex}` – adds link to an existing IMA group
- `removeImaLink {groupIndex} {linkIndex}` – removes link from an existing IMA group
- `removeImaGroup {groupIndex}` – removes the IMA group
- `showIma` – shows all configured IMA groups

10.6 PPP Configuration and Status

These commands allow for the configuration of PPP parameters on clear channel links and individual channels configured for PPP operation.

- `configPpp {linkIndex} {channelIndex} {PppParameter} {PppParameterValue}` – configures the PPP channel with one of the following parameters:
    - multilink
    - mru
    - mrru
- `showPpp {linkIndex} {channelIndex}` – displays the PPP configuration on the specified channel

The default PPP configuration is:

- multilink enabled with 4 classes
- MRU size = 1500
- MRRU size = 1500

10.7 FR Configuration and Status

These commands allow for the configuration of FR parameters on clear channel links and individual channels configured for FR operation.

- `configFr {linkIndex} {channelIndex} {FrParameter} (FrParameterValue}`
- – configures the FR channel with one of the following parameters:
    - mru
- `showFr {linkIndex} {channelIndex}` – displays the FR configuration on the specified channel

The default FR configuration is:

- MRU size = 4096

10.8 Statistics

These commands displays statistics gathered from various ports.

10.8.1 ATM statistics

The below command display the statistics gathered from the ATM ports.

- `getAtmStats {linkIndex} {channelIndex}` – displays the statistics for the specified ATM TDM channel

10.8.2 PPP statistics

The below command display the statistics gathered from the PPP ports.

- `getPppStats {linkIndex} {channelIndex}` – displays the statistics for the specified PPP TDM channel

10.8.3 FR statistics

The below command display the statistics gathered from the FR ports.

- `getFrStats {linkIndex} {channelIndex}` – displays the statistics for the specified FR TDM channel.

11 HOST PROCESSOR TO NETWORK PROCESSOR INTERFACE

This section details the control structures residing in the various NP DMEM's that the host component must initialize and maintain. These structures are exported in the NP source code so that the host can acquire the DMEM address from the package file.

11.1 PPP

The following data structures relate to the PPP components running on both the host and network processors.

11.1.1 PPP Link Parameters

This structure holds parameters specific to each PPP link that has been negotiated by LCP or NCP.

Byte Offset	0	1	2	3
0	flags	segSize	mru	

- flags: a bitmap as follows:
  - b7: port valid
  - b6: IPv4 NCP up
  - b5: unused
  - b4: MPLS NCP up
  - b3: Address and control field compression
  - b2: protocol field compression
  - b1-b0: unused
- segSize: ML-PPP segment size to send on this link
- mru: maximum received unit size that can be received on this link

11.2 ML-PPP

The following data structures relate to the ML-PPP components running on both the host and network processors.

11.2.1 ML Bundle Parameters

This structure holds parameters specific to each ML-PPP bundle that have been negotiated by LCP.

Byte Offset	0	1	2	3
0	flags	Pad	mru	
4	firstPort		NumPorts	pad

- flags: a bitmap as follows:
  - b7: bundle valid
  - b6: IPv4 NCP up
  - b5: unused
  - b4: MPLS NCP up
  - bb3: short sequence number
  - b2-b0: unused
- mrru: maximum receive reconstructed unit
- firstPort: index of first PPP link in ML bundle channel list
- numPorts: number of links in bundle
- pad: unused

11.2.2 TDM Channel to ML Bundle Map

This structure is a byte array, MAX_TDM_CHANNELS long. The index into the array is the TDM channel (PPP link) and the value of the array at that index is the ML bundle to which the TDM channel belongs. A value of 0xFF indicates the channel does not belong to a bundle.

11.2.3 ML Bundle Channel List

This structure is a byte array, MAX_TDM_CHANNELS long. A contiguous block of elements in the array indicates the TDM channels (PPP links), which belong to a particular ML bundle. FirstPort and numPort fields of the ML bundle parameter structure define the size and location of the block.

11.3 FR

The following data structures relate to the FR components running on both the host and network
processors.

11.3.1 FR Link Parameters

This structure holds parameters specific to each FR link.

Byte Offset	**0**	**1**	**2**	**3**
0	flags	mru		pad

- flags: a bitmap as follows:
  - o b7: port valid
  - o b6-b0: unused
- mru: maximum received unit size that can be received on this link

11.4 IMA

The following data structures relate to the IMA components running on both the host and network processors.

11.4.1 IMA Group Parameters

This structure holds parameters specific to each IMA group that has been configured.

Byte Offset	**0**	**1**	**2**	**3**
0	flags	firstPort		numPorts

- flags: a bitmap as follows:
  - o b7: group valid
  - o b6-b0: unused
- firstPort: index of first ATM link in IMA group channel list
- numPorts: number of links in group
- pad: unused

11.4.2 TDM channel to IMA Group Map

This structure is a byte array, MAX_TDM_CHANNELS long. The index into the array is the TDM channel (ATM link) and the value of the array at that index is the IMA group to which the TDM channel belongs. A value of 0xFF indicates the channel does not belong to a bundle.

11.4.3 IMA Group Channel List

This structure is a byte array, MAX_TDM_CHANNELS long. A contiguous block of elements in the array indicates the TDM channels (ATM links), which belong to a particular IMA group. The size and location of the block is defined by firstPort and numPort fields of the IMA group parameter structure.

11.5 ATM

The following data structures relate to the ATM components running on both the host and network
processors.

11.5.1 OAM Performance Monitoring Channel Map

This is a byte array, MAX_TDM_CHANNELS long. The index into the array is the TDM channel (ATM link) and the value of the array is a Boolean value indicating whether or not PM is to be performed on the channel.

12 IMPLEMENTATION DETAILS

This section covers some general implementation details as well as limitations, caveats, and other general information about the design of the applications.

12.1 ML-PPP

This section lists some of the details of the ML-PPP implementation.

- All ML-PPP negotiations (LCP options) are handled by the host. LCP and NCP packets pass
  transparently through the NP as they travel between host and peer.
- Fragmentation is always performed (which is permissible). Each member can specify the amount of payload to put into a fragment. This number must be a multiple of 16(default is 64).
- Both short and long sequence number header formats are supported.
- Self-Describing-Padding (which is optional) is not supported.
- Sequencing is handled by means of a linked list with a finite number of elements, limited by DMEM.Bundles with many links (> 16) may experience packet loss when fragments arrive out of order.
- Minimal attempt to detect lost fragments is made in the ML-PPP code. Packet loss can be
  detected by IP or UDP length, header, or checksum errors.

12.2 Soft Queues

Traffic management and quality of service (QoS) enforcing applications require a large number of queues.These queues typically are used as FIFO buffers rather than for inter-processor communication. Without an external hardware queuing-engine like the Q-5, it is possible to run out of QMU buffers when implementing traffic management/QoS for multiple channels/ports/flows/label-switched paths. This document proposes a design for a software based queuing component that will use BMU buffers to hold queue data and DMEM based structures to maintain FIFO ordering. These queues (hereafter referred to as soft queues) provide the FIFO buffer paradigm requires by QoS applications.

12.2.1 BMU buffer structure

A large BMU buffer, whose size either 1K or 2K or 4K depending on the maximum required queue depth is allocated. This buffer is logically partitioned into either 16byte or 32 byte slots. A packet descriptor is expected to occupy one slot. As all slots are aligned to 16byte boundaries, they can be read from and written to by DMA using bsBufferRead and bsBufferWrite API calls. This array is slots, is treated as a circular FIFO by maintaining the following data-structure in DMEM:

```
struct SoftQHandle
{
int16u btag; // All the buffers used for soft queues are allocated from the same pool ID.
int8u front;
int8u rear;
};
```

This structure needs 4 bytes of DMEM capacity per queue. It is a design constraint to keep the size of this structure to a minimum, as there are hundreds of soft queues per processor and QoS applications typically share DMEM with other applications in a cluster.

All the buffers used by soft queue component will be allocated from a single pool-id. Hence it is sufficient tosimply store the btag number in each queue handle. The pool-id will be stored in a global variable and will be private to the soft queue component.

The front and rear members of the above structure hold slot numbers for the descriptors at the head and tail of the circular buffer of slots. The physical location of slots is calculated by multiplying the slot number by 16 or more efficiently by left shifting the slot number four times. An empty queue is indicated by front being equal to rear. When the queue is full, front will be equal to (rear + 1)%#slots.

12.2.2 Soft Queue API

This section enumerates the soft queue API and describes their function signature and implementation.

Function	void sqInitialize (int16u poolId, int16u maxQueueDepth)
Parameters	• poolId – pool-id of the pool to be used for soft queues. Typically this would be created by the XP. maxQueueDepth – the buffer size for the pool-id. It can be one of the three numbers: 1024, 2048 or 4096.
Returns	void
Implementation	• Invoke bsBufPoolInitialize () • Store pool-id and maxQueueDepth in global variables.

Function	int8u sqCreate (SoftQueueHandle* qHandle)
Parameters	qHandle – a pointer to a valid soft-queue-handle structure

Returns	success on successful soft-queue creation and fail when out of buffers in the allocated pool
Implementation	• Allocate a pool from global pool-id. If cannot allocate return failure • Fill in the btag information in the handle and set front and rear members to 0xff • Return success

Function	int8u sqEnqueueStart (SoftQueueHandle* qHandle, int32u* desc)
Parameters	• qHandle – a pointer to a valid soft-queue-handle structure returned from a call to sqCreate • desc – a valid pointer to a 16 byte aligned packet descriptor
Returns	success on successful en-queue and fail when the fifo is full
Implementation	• If front == (rear + 1)%maxSlots then return failure • Do a bsBufferWrite of 16 bytes from desc to the offset (front << 4). Do not wait for the write tocomplete • rear: = (rear+1)%maxSlots and return success

Function	int8u sqEnqueueComplete (SoftQueueHandle* qHandle)
Parameters	qHandle – a pointer to a valid soft-queue-handle structure returned from a call to sqCreate
Returns	success if the bsBufferWrite, initiated by the enqueue has completed and failure otherwise
Implementation	Invoke a bsBufferWriteComplete on the bufferhandle in the queue handle and return the result

Function	int8u sqDequeueStart (SoftQueueHandle* qHandle, int32u* desc)
Parameters	• qHandle – a pointer to a valid soft-queue-handle structure • desc – a valid pointer to a 16 byte aligned packet descriptor in DMEM
Returns	success on successful de-queue and fail when the fifo is empty
Implementation	• If rear == front return NULL • Do a bsBufferRead of 16 bytes from offset (rear << 4) to desc • front: = (front + 1)%maxSlots • Return success

Function	int8u sqDequeueComplete (SoftQueueHandle* qHandle)
Parameters	qHandle – a pointer to a valid soft-queue-handle structure returned from a call to sqCreate
Returns	success if the bsBufferRead, initiated by the en-queue has completed and failure otherwise
Implementation	Invoke a bsBufferReadComplete on the buffer handle in the queue handle and return the result

Function	int8u sqDestroy (SoftQueueHandle* qHandle)
Parameters	qHandle – a pointer to a valid soft-queue-handle structure returned from a call to sqCreate
Returns	success on successful destroy and failure if qHandle is invalid
Implementation	De-allocate the buffer associated with this queue and return the result

12.2.3 Issues and enhancements

12.2.3.1 Support for Larger Queue Depths

The front and rear pointers are 8 bits wide and can hence point only up to the 255th slot. With a slot size of 16 bytes and 255 slots the maximum buffer size possible is 4K bytes. Increasing pointer size again leads to increase in DMEM requirements. Larger depths will also be possible with wider slot sizes as described in the previous paragraph.

12.2.3.2 Using Soft Queues for IPC

Soft queues are intended for single processor usage only. They can be logically extended for interprocessor communication by sharing and protecting the soft queue handle. However this configuration is not supported as of now.

12.2.3.3 Support for Non-FIFO Operations

With this design, it is possible to implement alternative strategies like head drop instead of normal tail drop. Such enhancements can be made as and when necessary.

13 HOST API REFERENCE

13.1 Table API

This section lists the functions and data types available on the host for table maintenance.

13.1.1 Port Table API

This API allows an application to maintain the port table. This table stores parameters for each of the ports. For more information, see Section 7.18.1

13.1.1.1 Data Types

1.1.1.1.1. PortTableInfo

Type	struct
Description	This structure contains fields that contain the necessary information to form a key for this table and also contains fields that correspond to those of this table’s entry type.
Usage	The definition for the fields of this structure are as follows: • int16u channel – the logical channel index of the port (key) • int8u portType – the type of port (entry) • int8u flags – flags associated with this port (entry) • int16u egressQueue – the egress queue for this port (entry) • int16u qosQueue – the QOS queue for this port (entry)

13.1.1.2 Functions

1.1.1.1.2. getPortTable

Function	int getPortTable(int npIndex, PortTableInfo* info)
Description	This function attempts to get the entry data in the Port Table for the key specified.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (lookup failed)

1.1.1.1.3. getNextPortTable

Function	int getNextPortTable(int npIndex, PortTableInfo* info)
Description	This function is used to “walk” the Port table returning the key and entry data for the next valid entry.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (no more valid entries)
Implementation	The “next” entry is determined by an index assigned to the entry by Table Services when it was created.

1.1.1.1.4. setPortTable

Function	int setPortTable(int npIndex, PortTableInfo* info)
Description	This function is used to add or modify an entry in the Port Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and containing the entry data
Returns	0 – operation was successful 1 – operation was not successful
Implementation	The entry is looked up first and if it is “found”, a table modify is performed otherwise a table add is performed.

1.1.1.1.5. deletePortTable

Function	int deletePortTable(int npIndex, PortTableInfo* info)
Description	This function removes an entry from the Port Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key
Returns	0 – operation was successful 1 – operation was not successful

1.1.1.1.6. flushPortTable

Function	int flushPortTable(int npIndex, int& flushCount)
Description	This function removes all entries from the Port Table
Parameters	• npIndex – index of the NP for the operation • flushCount – used to return the number of table entries that were removed
Returns	0 – operation was successful (flushCount is valid) 1 – operation was not successful

13.1.2 IPv4 Route Table API

This API allows an application to maintain the IPv4 Route Table. This table stores forwarding information for IPv4 and UDP datagrams. For more information, See Section 7.18.3.

13.1.2.1 Data Types

1.1.1.1.7. AtmFwd

Description	This structure contains fields that contain the necessary information to correctly forward a packet or cell to an ATM port.
Type	Struct
Usage	The definition for the fields of this structure are as follows: • int32u egressCellHeader – ATM cell header to be used • int16u egressQueue –trafficQueue mapped to Q-3 for applying ATM TM

1.1.1.1.8. MplsFwd

Description	This structure contains fields that contain the necessary information to perform MPLS forwarding to ingress packet or cell.
Type	Struct
Usage	The definition for the fields of this structure are as follows: • int16u labelPush1 –Label to be added. • int16u labelPush2 –Second Label to be added for the MPLS action PUSH_PUSH. • int8u action –MPLS action to be performed. • int8u hopCount –count value to be decremented from TTL. • int8u egressBufType –buffer type of the egress interface.

1.1.1.1.9. FrFwd

Description	This structure contains fields that contain the necessary information to correctly forward a packet or cell to a FR port
Type	Struct
Usage	The definition for the fields of this structure are as follows: • int32u egressFrHeader – 31-16: DLCI value of the FR header.

1.1.1.1.10. IpAddr

Description	This structure contains the representation of an Ipv4 Network Address.
Type	Union
Usage	The definition for the fields of this union are as follows: • int32u full – access to entire address • int8u bytes [4] – allow access to individual bytes

1.1.1.1.11. IpV4RouteInfo

Description	This structure contains fields that contain the necessary information to form a key for this table and also contains fields that correspond to those of this table's entry type (ATM/MPLS/FR).
Type	struct

Usage	The definition for the fields of this structure are as follows: • IpAddr address – IPv4 Network Layer address (key) • IpAddr mask – IPv4 Net Mask (key) • int16u port – egress port. • int8u flag – if "0" configured for IP route, if non-zero configured for MPLS route. • int8u maskBits – the number of significant bytes in the key • union { int32u word [2] – provide word access int16u hword[4] – provide half-word access int8u byte[8] – provide byte access AtmFwd atmFwd – ATM Port forwarding information MplsFwd mplsFwd – MPLS forwarding information FrFwd frFwd – FR forwarding information } appData; • int32u appHdrData – Egress Cell header (valid for MPLS entry only) • int16u qosQueue – Qos Queue value (valid for MPLS entry only).

13.1.2.2 Functions

1.1.1.1.12. getIpV4RouteTable

Function	int getIPv4RouteTable(int npIndex, IPv4RouteInfo* info)
Description	This function attempts to get the entry data in the Ipv4 RouteTable for the key specified.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (lookup failed)

1.1.1.1.13. getNextIpV4RouteTable

Function	int getNextIPv4RouteTable(int npIndex, IPv4RouteInfo* info)
Description	This function is used to "walk" the IPv4 Route Table returning the key and entry data for the next.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (no more valid entries)
Implementation	The "next" entry is determined by an index assigned to the entry by Table Services when it was created.

1.1.1.1.14. setIpV4RouteTable

Function	int settIPv4RouteTable(int npIndex, IPv4RouteInfo* info)
Description	This function is used to add or modify an entry in the IPv4 Route Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and containing the entry data
Returns	0 – operation was successful 1 – operation was not successful
Implementation	The entry is looked up first and if it is "found", a table modify is performed otherwise a table add is performed.

1.1.1.1.15. deleteIpV4RouteTable

Function	int deleteIPv4RouteTable(int npIndex, IPv4RouteInfo* info)
Description	This function removes an entry from the IPv4 Route Table
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key
Returns	0 – operation was successful 1 – operation was not successful

1.1.1.1.16. flushIpV4RouteTable

Function	int flushIPv4RouteTable(int npIndex, int& flushCount)
Description	This function removes all entries from the IPv4 Route Table.
Parameters	• npIndex – index of the NP for the operation • flushCount – used to return the number of table entries that were removed
Returns	0 – operation was successful (flushCount is valid) 1 – operation was not successful

13.1.3 ATM VC Table API

This API allows an application to maintain the ATM VC Table. This table stores connection information for ATM virtual circuits and MPLS entry if VPI/VCI maps to the MPLS Label. For more information, see section 7.18.4.

13.1.3.1 Data Types

1.1.1.1.17. AtmEntryInfo

Description	This structure contains fields that contain the necessary information to form a key for this table and also contains fields that correspond to those of this table's entry type.
Type	Struct
Usage	The definition for the fields of this structure are as follows: • int16u gfcVpiVciHi – GFC, VPI, and upper byte of VCI header fields (key) • int16u vciLoPtiClp – VCI lower byte, PTI< and CLP header fields (key) • int16u port – port index (key) For ATM VC entry the following fields are used, • int16u flags_egressPort – flag and egress port, bits 15:11 are flags, 10:0 is the port • int16u vcIndex – VC index used by AAL5 SAR to index their state tables. • int32u egressCellHeader – egress port cell header, • int16u egressQueue –The egress queue of the ATM cell, this is necessary for the FP which can launch only one lookup. This also represents the trafficQueue, to be used if QoS is enabled for this egress port • int16u port_bufType – the egress port and buffer type packaged in a way the FP can use • int16u destQueue – the queue of the next ATM processing block, this is necessary for the FP which cannot act on TLU response data For ATM MPLS entry the following fields are used, • int16u flags_egressPort – flags and egress port, bits 15:11 are flags, 10:0 is the port • int16u vcIndex – VC index used by AAL5 SAR to index their state tables. • int16u labelSwap – Label to be swapped or added (for PUSH_PUSH action) • int16u labelPush – Label to be added • int16u egressQueue – egress queue • int8u hopCount _action – MPLS action to be performed (4bits) and the count to be decremented in TTL of the shim header(4bits). • int8u egressBufType – the type of buffer being recirculated and the egress port type (could be BT_MPLS_PPP, BT_MPLS_ATM, BT_MPLS_FR, BT_MPLS_IPv4)

	• int32u appHdrData – CellHeader if ATM as the egress port type or DLCI if FR as the egress port type.

13.1.3.2 Functions

1.1.1.1.18. getAtmVcTable

Function	int getAtmVcTable(int npIndex, AtmEntryInfo* info)
Description	This function attempts to get the entry data in the ATM VC/MPLS Table for the key specified.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (lookup failed)

1.1.1.1.19. getNextAtmVcTable

Function	int getNextAtmVcTable(int npIndex, AtmEntryInfo* info)
Description	This function is used to “walk” the ATM VC/MPLS table returning the key and entry data for the next valid entry.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (no more valid entries)
Implementation	The “next” entry is determined by an index assigned to the entry by Table Services when it was created.

1.1.1.1.20. setAtmVcTable

Function	Int setAtmVcTable(int npIndex, AtmEntryInfo* info)
Description	This function is used to add or modify an entry in the ATM VC Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and containing the entry data
Returns	0 – operation was successful 1 – operation was not successful
Implementation	The entry is looked up first and if it is “found”, a table modify is performed otherwise a table add is performed.

1.1.1.1.21. deleteAtmVcTable

Function	int deleteAtmVcTable(int npIndex, AtmEntryInfo* info)
Description	This function removes an entry from the ATM VC Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key
Returns	0 – operation was successful 1 – operation was not successful

1.1.1.1.22. flushAtmVcTable

Function	int flushAtmVcTable(int npIndex, int& flushCount)
Description	This function removes all entries from the ATM VC Table
Parameters	• npIndex – index of the NP for the operation

	• flushCount – used to return the number of table entries that were removed
Returns	0 – operation was successful (flushCount is valid) 1 – operation was not successful

13.1.4 Diffserv Flow Table API

This API allows an application to maintain the DiffServ Flow Table. This table stores the per-hop behavior for IPv4 flows. For more information, see section 7.18.6.

13.1.4.1 Data Types

1.1.1.1.23. DiffServFlowInfo

Description	This structure contains fields that contain the necessary information to form a key for this table and also contains fields that correspond to those of this table's entry type.
Type	Struct
Usage	The definition for the fields of this structure are as follows: • int16u ipDestAddrHi – high bytes of destination IPv4 Network Address (key) • int16u ipDestAddrLo – low bytes of destination IPv4 Network Address (key) • int16u destPort – destination UDP port (key) • int32u ipSrcAddr – source IPv4 Network Address (key) • int16u srcPort – source UDP port (key) • int8u protocol – protocol (key) • int8u egressPort – egress port (key) • int8u maskBits – mask bits • int8u phb – per hop behavior • int16u flowId – flow ID • int16u qosQueueOffset – used for Q3 based QOS • int8u flags – bit 0: 1 –send ;0 –Drop

13.1.4.2 Functions

1.1.1.1.24. getDiffServFlowTable

Function	int getDiffServFlowTable(int npIndex, DiffServFlowInfo* info)
Description	This function attempts to get the entry data in the DiffServ Flow Table for the key specified.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (lookup failed)

1.1.1.1.25. getNextDiffServFlowTable

Function	int getNextDiffServFlowTable(int npIndex, DiffServFlowInfo* info)
Description	This function is used to "walk" the DiffServ Flow table returning the key and entry data for the next valid entry.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data

	1 – operation was not successful (no more valid entries)
Implementation	The "next" entry is determined by an index assigned to the entry by Table Services when it was created.

1.1.1.1.26. *setDiffServFlowTable*

Function	Int setDiffServFlowTable(int npIndex, DiffServFlowInfo* info)
Description	This function is used to add or modify an entry in the DiffServ Flow Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and containing the entry data
Returns	0 – operation was successful 1 – operation was not successful
Implementation	The entry is looked up first and if it is "found", a table modify is performed otherwise a table add is performed.

1.1.1.1.27. *deleteDiffServFlowTable*

Function	int deleteDiffServFlowTable(int npIndex, DiffServFlowInfo* info)
Description	This function removes an entry from the DiffServ Flow Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key
Returns	0 – operation was successful 1 – operation was not successful

1.1.1.1.28. *flushDiffServFlowTable*

Function	int flushDiffServFlowTable(int npIndex, int& flushCount)
Description	This function removes all entries from the DiffServ Flow Table
Parameters	• npIndex – index of the NP for the operation • flushCount – used to return the number of table entries that were removed
Returns	0 – operation was successful (flushCount is valid) 1 – operation was not successful

13.1.5 MPLS Table API

This API allows an application to maintain the MPLS Table. This table stores MPLS processing information for PPP frames and also for POP action (which needs additional label lookup based on the next label in the stack) in the case of MPLS packet over ATM and FR. For more information, see section 7.18.8.

13.1.5.1 Data Types

1.1.1.1.29. MplsEntryInfo

Description	This structure contains fields that contain the necessary information to form a key for this table and also contains fields that correspond to those of this table's entry type.
Type	Struct
Usage	The definition for the fields of this structure are as follows: • int16u keyLabel – Label which forms the index (key) to MPLS table. MPLS entry fields are mentioned below : • int16u labelSwap – Label to be swapped or added (for PUSH_PUSH action) • int16u labelPush – Label to be added. • int8u action – MPLS action to be performed • int8u hopCount - count to be decremented in TTL of the shim header • int16u port_bufType – a bitmask defined as follows: b15-5: port – the output port on which this datagram to be transmitted. b4-0: bufType – the type of buffer being recirculated and the egress port type (could be BT_MPLS_PPP, BT_MPLS_ATM, BT_MPLS_FR, BT_MPLS_IPv4) • int16u egressQueue – the egress queue for the egress port. • int16u pad -unused • int32u appHdrData – egressCellHeader for ATM or egressFrHeader for FR

13.1.5.2 Functions

1.1.1.1.30. geMplsTable

Function	int getMplsTable(int npIndex, MplsEntryInfo * info)
Description	This function attempts to get the entry data in the MPLS Table for the key specified.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (lookup failed)

1.1.1.1.31. getNextMplsTable

Function	int getNextMplsTable(int npIndex, MplsEntryInfo * info)
Description	This function is used to "walk" the MPLS Table returning the key and entry data for the next.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (no more valid entries)
Implementation	The "next" entry is determined by an index assigned to the entry by Table Services when it was created.

1.1.1.1.32. setMplsTable

Function	int setMplsTable(int npIndex, MplsEntryInfo * info)
Description	This function is used to add or modify an entry in the MPLS Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and containing the entry data
Returns	0 – operation was successful 1 – operation was not successful
Implementation	The entry is looked up first and if it is "found", a table modify is performed otherwise a table add is performed.

1.1.1.1.33. deleteMplsTable

Function	Int deleteMplsTable(int npIndex, MplsEntryInfo * info)
Description	This function removes an entry from the MPLS Table
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key
Returns	0 – operation was successful 1 – operation was not successful

1.1.1.1.34. flushMplsTable

Function	Int flushMplsTable(int npIndex, int& flushCount)
Description	This function removes all entries from the MPLS Table.
Parameters	• npIndex – index of the NP for the operation • flushCount – used to return the number of table entries that were removed
Returns	0 – operation was successful (flushCount is valid) 1 – operation was not successful

13.1.6 FR Table API

This API allows an application to maintain the FR Table. This table stores MPLS processing information for PPP frames and also for POP action (which needs additional label lookup based on the next label in the stack) in the case of MPLS packet over ATM and FR. For more information, see section 7.18.7.

13.1.6.1 Data Types

1.1.1.1.35. FrEntryInfo

Description	This structure contains fields that contain the necessary information to form a key for this table and also contains fields that correspond to those of this table's entry type.
Type	Struct
Usage	The definition for the fields of this structure are as follows: • int32u interfaceID_DLCI – 20:10 interfaceID (11bits), 9:0 DLCI (10). FR entry field is listed below: • int16u FrHeader -specifies the DLCI value and the congestion control information. MPLS entry fields are listed below: • int16u labelSwap – Label to be swapped or added (for PUSH_PUSH action) • int16u labelPush – Label to be added. • int8u action – MPLS action to be performed • int8u hopCount - count to be decremented in TTL of the shim header • int16u port_bufType – a bitmask defined as follows: b15-5: port – the output port on which this datagram to be transmitted. b4-0: bufType – the type of buffer being recirculated and the egress port type (could be BT_MPLS_PPP, BT_MPLS_ATM, BT_MPLS_FR, BT_MPLS_IPv4) • int16u egressQueue – the egress queue for the egress port (entry). • int8u flags – bit 0: value 1 - MPLS entry, value 0-FR entry.

13.1.6.2 Functions

1.1.1.1.36. getFrTable

Function	int getFrTable (int npIndex, FrEntryInfo* info)
Description	This function attempts to get the entry data in the FR Table for the key specified.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (lookup failed)

1.1.1.1.37. getNextFrTable

Function	int getNextFrTable (int npIndex, FrEntryInfo* info)
Description	This function is used to “walk” the FR Table returning the key and entry data for the next.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and returning the lookup results
Returns	0 – operation was successful and the info structure contains valid data 1 – operation was not successful (no more valid entries)
Implementation	The “next” entry is determined by an index assigned to the entry by Table Services when it was created.

1.1.1.1.38. setFrTable

Function	int setFrTable (int npIndex, FrEntryInfo* info)
Description	This function is used to add or modify an entry in the FR Table.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key and containing the entry data
Returns	0 – operation was successful 1 – operation was not successful
Implementation	The entry is looked up first and if it is “found”, a table modify is performed otherwise a table add is performed.

1.1.1.1.39. deleteFrTable

Function	Int deleteFrTable (int npIndex, FrEntryInfo* info)
Description	This function removes an entry from the FR Table
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for generating a key
Returns	0 – operation was successful 1 – operation was not successful

1.1.1.1.40. flushFrTable

Function	Int flushFrTable(int npIndex, int& flushCount)
Description	This function removes all entries from the FR Table.
Parameters	• npIndex – index of the NP for the operation • flushCount – used to return the number of table entries that were removed
Returns	0 – operation was successful (flushCount is valid) 1 – operation was not successful

13.2 Link, Channel, and IMA API

This section lists the API available on the host for maintaining links, channels, and IMA groups at the TDM interface.

13.2.1 Data Types

13.2.1.1 MsaLinkType

Description	This structure contains an enumeration of the supported link types.
Type	Enum
Usage	• MSA_LINK_NULL – null type • MSA _LINK_T1 – T1 link • MSA _LINK_E1 – E1 link • MSA _LINK_LAST – last value

13.2.1.2 LinkChannelInfo

Description	This structure contains fields that contain the necessary information to describe a T1/E1 link and DS0 sub-channels.
Type	Struct
Usage	The definition for the fields of this structure are as follows: • int8u **linkIndex** – index of the desired link • int8u **channelIndex** – index of the desired channel • int8u **ifType** – if "0" configured for ATM, 1 for PPP and 2 for FR • int8u **clearChannel** – if non-zero, configured for Clear Channel • int **activeChannelCount** – number of active sub-channels • MsaLinkType **linkType** – link type • int **logChannel** – logical channel index (used to return the allocated logical channel when a link or channel is created)

13.2.2 Functions

13.2.2.1 addLink

Function	int addLink(int npIndex, LinkChannelInfo* info)
Description	This function is used to add a new link configuration.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for configuring the link
Returns	0 – operation was successful 1 – operation was not successful

13.2.2.2 removeLink

Function	int removeLink(int npIndex, LinkChannelInfo* info)
Description	This function is used to remove an existing link.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for specifying the link to be removed
Returns	0 – operation was successful 1 – operation was not successful

Implementation	The link must not have any configured sub-channels or be part of an IMA group.

13.2.2.3 getLink

Function	int getLink(int npIndex, LinkChannelInfo* info)
Description	This function is used to get the configuration of the specified link.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for specifying the link and returning the configuration
Returns	0 – operation was successful 1 – operation was not successful

13.2.2.4 getNextLink

Function	int getNextLink(int npIndex, LinkChannelInfo* info)
Description	This function is used to “walk” all active links and return their configuration.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for returning the link configuration
Returns	0 – operation was successful 1 – operation was not successful (no more active links)
Implementation	The links are traversed in ascending link index order.

13.2.2.5 addChannel

Function	int addChannel(int npIndex, LinkChannelInfo* info)
Description	This function is used to add a new sub-channel to an existing link.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for specifying the link and channel.
Returns	0 – operation was successful 1 – operation was not successful
Implementation	The link must be configured for non-clear channel operation.

13.2.2.6 removeChannel

Function	int removeChannel(int npIndex, LinkChannelInfo* info)
Description	This function is used to remove an existing sub-channel from a link.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for specifying the link and channel.
Returns	0 – operation was successful 1 – operation was not successful

13.2.2.7 getChannel

Function	int getChannel(int npIndex, LinkChannelInfo* info)
Description	This function is used to get the configuration of an existing sub-channel on a link.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for specifying the link and channel

Returns	0 – operation was successful 1 – operation was not successful

13.2.2.8 getNextChannel

Function	int getNextChannel(int npIndex, LinkChannelInfo* info)
Description	This function is used to get the configuration of the "next" active sub-channel in a link
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for specifying the link and channel
Returns	0 – operation was successful 1 – operation was not successful (no more active channels)

13.2.2.9 getLogChannel

Function	int getLogChannel(int npIndex, int logChannelIndex, LinkChannelInfo* info)
Description	This function is used to get the link/channel information corresponding to a particular logical channel.
Parameters	• npIndex – index of the NP for the operation • logChannelIndex – index of the logical channel • info – structure to be used for returning the link/channel info if successful
Returns	0 – operation was successful 1 – operation was not successful (logical channel not in use)

13.2.2.10 getNextLogChannel

Function	int getNextLogChannel(int npIndex, LinkChannelInfo* info)
Description	This function is used to get the configuration of the "next" active logical channel.
Parameters	• npIndex – index of the NP for the operation • info – structure to be used for returning the link/channel info if successful
Returns	0 – operation was successful 1 – operation was not successful (no more active channels)

13.2.2.11 addImaGroup

Function	int addImaGroup(int npIndex, int baseLinkIndex, int& imaGroupIndex)
Description	This function is used to create a new IMA Group
Parameters	• npIndex – index of the NP for the operation • baseLinkIndex – first link to be placed in the new IMA group • imaGroupIndex – returned value of new IMA Group index if successful
Returns	0 – operation was successful 1 – operation was not successful (base link not configured correctly)

13.2.2.12 addImaLink

Function	int addImaLink(int npIndex, int imaGroupIndex, int linkIndex)
Description	This function is used to add a link to an existing IMA Group
Parameters	• npIndex – index of the NP for the operation • imaGroupIndex – existing IMA group to be added to • linkIndex – index of link to be added
Returns	0 – operation was successful 1 – operation was not successful (link not configured correctly)

13.2.2.13 removeImaLink

Function	int removeImaLink(int npIndex, int imaGroupIndex, int linkIndex)
Description	This function is used to remove a link from an existing IMA Group
Parameters	• npIndex – index of the NP for the operation • imaGroupIndex – existing IMA group to be removed from • linkIndex – index of link to be removed
Returns	0 – operation was successful 1 – operation was not successful (link not a member of the group)

13.2.2.14 removeImaGroup

Function	int removeImaGroup(int npIndex, int imaGroupIndex)
Description	This function is used to remove an existing IMA Group
Parameters	• npIndex – index of the NP for the operation • imaGroupIndex – existing IMA group to be removed
Returns	0 – operation was successful 1 – operation was not successful (not all links have been removed first)

13.3 PPP API

This section lists the API available on the host for configuring and maintaining PPP links.

13.3.1 Data Types

13.3.1.1 MsaPppParam

Description	This structure contains an enumeration of the supported PPP parameters.
Type	enum
Usage	The definition for the fields of this structure are as follows: • LCP_OPT_MRU – LCP MRU size option • LCP_VAL_MRU – LCP MRU value • LCP_OPT_ASYNC – LCP async map option (not supported) • LCP_OPT_UPAP – LCP PAP option (not supported) • LCP_OPT_CHAP – LCP CHAP option (not supported) • LCP_OPT_MAGIC – LCP magic number option • LCP_OPT_PCOMP – LCP Protocol Field compression option • LCP_OPT_ACCOMP – LCP Address/Control Field compression option • LCP_OPT_LQR – LCP Link Quality Reporting Option (not supported) • LCP_OPT_CBCP – LCP Callback Option (not supported) • LCP_OPT_MRRU – LCP MRRU Option • LCP_VAL_MRRU – LCP MRRU value • LCP_OPT_SSN – LCP Send Short Sequence Number Option • LCP_OPT_EPD – LCP Endpoint Discriminator Option • LCP_PASSIVE – keep LCP active if no responses are heard • LCP_SILENT – let the other end start LCP negotiation first • LCP_RESTART – restart (vs. exit) after LCP closes • IPCP_OPT_ADDR – IPCP IP Address negotiation option • IPCP_OPT_VJ – IPCP Van Jacobsen compression option • IPCP_REQ_ADDR – IPCP request peer to send its IP address • IPCP_OLD_VJ – IPCP use old (short) form of VJ option • IPCP_OLD_ADDRS – IPCP use old (IP-Addresses) option

	• IPCP_ACCEPT_LOCAL – IPCP accept peer's value for our address • IPCP_ACCEPT_REMOTE – IPCP accept peer's value for his address • IPCP_REQ_DNS1 – IPCP Ask peer to send primary DNS address • IPCP_REQ_DNS2 – IPCP Ask peer to send secondary DNS address • IPCP_COMP_PROT – IPCP Compression Protocol Value • IPCP_MAX_SLOT – IPCP Max Slot Index value

13.3.2 LCP API

This API allows an application to configure the LCP parameters of a PPP link.

13.3.2.1 getPppLcpParam

Function	int getPppLcpParam(int npIndex, int tdmChannel, MsaPppParam param, int32u* value)
Description	This function is used to get the specified LCP parameter from the specified PPP link
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • param – the parameter to get • *value – the returned value of the parameter if successful
Returns	0 – operation was successful 1 – operation was not successful

13.3.2.2 setPppLcpParam

Function	int setPppLcpParam(int npIndex, int tdmChannel, MsaPppParam param, int32u* value)
Description	This function is used to set the specified LCP parameter on the specified PPP link
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • param – the parameter to set • value – the value of the parameter
Returns	0 – operation was successful 1 – operation was not successful

13.3.2.3 getPppLcpStats

Function	int getPppLcpStats(int npIndex, int tdmChannel, void* statBuffer)
Description	This function is used to get the LCP statistics from the specified PPP link
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • statBuffer – user buffer to hold returned statistics
Returns	0 – operation was successful 1 – operation was not successful

13.3.3 IPCP API

This API allows an application to configure the IPv4 CP parameters of a PPP link.

13.3.3.1 getPppIpCpParam

Function	int getPppIpCpParam(int npIndex, int tdmChannel, MsaPppParam param, int32u* value)
Description	This function is used to get the specified IPCP parameter from the specified PPP link
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • param – the parameter to get • value – the returned value of the parameter if successful
Returns	0 – operation was successful 1 – operation was not successful

13.3.3.2 setPppIpCpParam

Function	int setPppIpCpParam(int npIndex, int tdmChannel, MsaPppParam param, int32u* value)
Description	This function is used to set the specified IPCP parameter on the specified PPP link
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • param – the parameter to set • value – the value of the parameter
Returns	0 – operation was successful 1 – operation was not successful

13.3.3.3 getPppIpCpStats

Function	int getPppIpCpStats(int npIndex, int tdmChannel, void* statBuffer)
Description	This function is used to get the IPCP statistics from the specified PPP link
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • statBuffer – user buffer to hold returned statisatics
Returns	0 – operation was successful 1 – operation was not successful

13.4 ATM API

This section lists the API available on the host for configuring and maintaining ATM links.

13.4.1 getAtmPortStats

Function	int getAtmStats(int npIndex, int tdmChannel, void* statBuffer)
Description	This function is used to get the ATM port statistics from the specified ATM Link.
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • statBuffer – user buffer to hold returned statisatics
Returns	0 – operation was successful 1 – operation was not successful

13.5 FR API

This section lists the API available on the host for configuring and maintaining FR links.

13.5.1 Data Types

13.5.1.1 MsaFrParam

Description	This structure contains an enumeration of the supported FR parameters.
Type	enum
Usage	The definition for the fields of this structure are as follows: • mru - maximum received unit size that can be received on this link.

13.5.1.2 getFrPortParam

Function	int getFrParam(int npIndex, int tdmChannel, MsaFrParam param, int32u* value)
Description	This function is used to get the specified FR parameter from the FR Link
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • param – the parameter to get • value – the returned value of the parameter if successful
Returns	0 – operation was successful 1 – operation was not successful

13.5.1.3 setFrPortParam

Function	int setFrParam(int npIndex, int tdmChannel, MsaFrParam param, int32u* value)
Description	This function is used to set the specified FR parameter on the FR Link
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • param – the parameter to set • value – the value of the parameter
Returns	0 – operation was successful 1 – operation was not successful

13.5.1.4 getFrPortStats

Function	int getFrStats(int npIndex, int tdmChannel, void* statBuffer)
Description	This function is used to get the FR port statistics from the specified FR Link.
Parameters	• npIndex – index of the NP for the operation • tdmChannel – logical TDM channel of the desired PPP link • statBuffer – user buffer to hold returned statisatics
Returns	0 – operation was successful 1 – operation was not successful

13.6 Control API

This section lists the API available on the host for control the applications and processing their outputs.

13.6.1 Functions

13.6.1.1 startMsaLineCard

Function	int startMsaLineCard(int32u npIndex, int8u fabricId, int(*rxIP)(int,int32u,void*,int16u,int8u*), int(*rxMPLS)(int,int32u,void*,int16u,int8u*), int(*rxPPP)(int,int32u,void*,int16u,int8u*), int(*rxATM)(int,int32u,void*,int16u,int8u*), int(*rxFR)(int,int32u,void*,int16u,int8u*), int defaultConfig))
Description	This function is used to start the MSA Line Card application
Parameters	• npIndex – index of the NP for the operation • fabricId – fabricId • rxIP – IP upcall function pointer • rxMPLS - MPLS upcall function pointer • rxPPP – PPP upcall function pointer • rxATM – ATM upcalll function pointer • rxFR – FR upcalll function pointer • defaultConfig – if set, a default link/channel configuration and table contents are loaded
Returns	0 – operation was successful 1 – operation was not successful
Implementation	NULL pointers are allowed for the upcalls. The rxPPP upcall is not currently used.

13.6.1.2 registerIpUpcall

Function	int registerIpUpcall(int32u npIndex, int(*rxIP)(int,int32u,void*,int16u,int8u*))
Description	This function is used to register an IP upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation • rxIP – IP upcall function pointer
Returns	0 – operation was successful 1 – operation was not successful (an upcall was previously registered and not de-registered)

13.6.1.3 registerMplsUpcall

Function	int registerMplsUpcall(int32u npIndex, int(*rxMPLS)(int,int32u,void*,int16u,int8u*))
Description	This function is used to register an MPLS upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation • rxMPLS – MPLS upcall function pointer
Returns	0 – operation was successful 1 – operation was not successful (an upcall was previously registered and not de-registered)

13.6.1.4 registerPppUpcall

Function	int registerPppUpcall(int32u npIndex, int(*rxPPP)(int,int32u,void*,int16u,int8u*))
Description	This function is used to register a PPP upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation • rxPPP – PPP upcall function pointer
Returns	0 – operation was successful 1 – operation was not successful (an upcall was previously registered and not de-registered)

13.6.1.5 registerAtmUpcall

Function	int registerAtmUpcall(int32u npIndex, int(*rxATM)(int,int32u,void*,int16u,int8u*))
Description	This function is used to register an ATM upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation • rxATM – ATM upcall function pointer
Returns	0 – operation was successful 1 – operation was not successful (an upcall was previously registered and not de-registered)

13.6.1.6 registerFrUpcall

Function	int registerFrUpcall(int32u npIndex, int(*rxFR)(int,int32u,void*,int16u,int8u*))
Description	This function is used to register an FR upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation • rxFR – FR upcall function pointer
Returns	0 – operation was successful 1 – operation was not successful (an upcall was previously registered and not de-registered)

13.6.1.7 deregisterIpUpcall

Function	int deregisterIpUpcall(int32u npIndex)
Description	This function is used to de-register an IP upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation
Returns	0 – operation was successful 1 – operation was not successful

13.6.1.8 deregisterMplsUpcall

Function	int deregisterMplsUpcall(int32u npIndex)
Description	This function is used to de-register an MPLS upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation
Returns	0 – operation was successful 1 – operation was not successful

13.6.1.9 deregisterPppUpcall

Function	int deregisterPppUpcall(int32u npIndex)
Description	This function is used to de-register a PPP upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation
Returns	0 – operation was successful 1 – operation was not successful

13.6.1.10 deregisterAtmUpcall

Function	int deregisterAtmUpcall(int32u npIndex)
Description	This function is used to de-register an ATM upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation
Returns	0 – operation was successful 1 – operation was not successful

13.6.1.11 deregisterFrUpcall

Function	int deregisterFrUpcall(int32u npIndex)
Description	This function is used to de-register an FR upcall function once the application has been started.
Parameters	• npIndex – index of the NP for the operation
Returns	0 – operation was successful 1 – operation was not successful

13.7 NP Port API

This section lists the API available on the host to configure the NP network interfaces (ports) and get the interface statistics.

13.7.1 Data Types

13.7.1.1 MsaPortParam

Description	This structure contains an enumeration of the supported NP Port parameters.
Type	Enum
Usage	The definition for the fields of this structure are as follows: • TDM_CHUNK_SIZE – 32/ 64 bytes.

13.7.2 TDM API

This API allows an application to configure the TDM network interfaces on the NP.

13.7.2.1 getTdmPortParam

Function	int getTdmPortParam(int npIndex, MsaPortParam param, int32u* value)
Description	This function is used to get the specified TDM parameter from the TDM port
Parameters	• npIndex – index of the NP for the operation • param – the parameter to get • value – the returned value of the parameter if successful
Returns	0 – operation was successful 1 – operation was not successful

13.7.2.2 setTdmPortParam

Function	int setTdmPortParam(int npIndex, MsaPortParam param, int32u* value)
Description	This function is used to set the specified TDM parameter on the TDM port
Parameters	• npIndex – index of the NP for the operation • param – the parameter to set • value – the value of the parameter to set
Returns	0 – operation was successful 1 – operation was not successful

13.7.2.3 getTdmPortStats

Function	int getTdmPortStats(int npIndex, void* statBuffer)
Description	This function is used to get the TDM port statistics.
Parameters	• npIndex – index of the NP for the operation • statBuffer – user buffer to hold returned statistics
Returns	0 – operation was successful 1 – operation was not successful

13.7.3 Fabric API

This API allows an application to configure the fabric port interface on the NP.

13.7.3.1 getFabricPortParam

Function	int getFabricPortParam(int npIndex, MsaPortParam param, int32u* value)
Description	This function is used to get the specified parameter from the Fabric port
Parameters	• npIndex – index of the NP for the operation • param – the parameter to get • value – the returned value of the parameter if successful
Returns	0 – operation was successful 1 – operation was not successful
Implementation	No Fabric parameters are currently defined

13.7.3.2 setFabricPortParam

Function	int setAtmPortParam(int npIndex, MsaPortParam param, int32u* value)
Description	This function is used to set the specified parameter on the Fabric port
Parameters	• npIndex – index of the NP for the operation • param – the parameter to set • value – the value of the parameter to set
Returns	0 – operation was successful 1 – operation was not successful
Implementation	No Fabric parameters are currently defined

13.7.3.3 getFabricPortStats

Function	int getFabricPortStats(int npIndex, void* statBuffer)
Description	This function is used to get the Fabric port statistics.
Parameters	• npIndex – index of the NP for the operation • statBuffer – user buffer to hold returned statistics
Returns	0 – operation was successful 1 – operation was not successful

13.8 I/O API

This section lists the API available on the host to perform send and receive packets to and from the NP.

13.8.1 Data Types

13.8.1.1 MsaPktType

Description	This structure contains an enumeration of the supported packet formats.
Type	Enum
Usage	The definition for the fields of this structure are as follows: • MSA _PKT_PPP_LCP – PPP LCP • MSA _PKT_PPP_IPCP – PPP IPCP • MSA_PKT_PPP_IPV4 – PPP IPv4 • MSA _PKT_PPP_MLPP – PPP Multilink • MSA_PKT_PPP_MPLS – PPP MPLS • MSA _PKT_ATM – ATM • MSA_PKT_ATM_OAM – ATM OA&M • MSA _PKT_ATM _MPLS– ATM MPLS • MSA _PKT_FR – FR • MSA _PKT_FR_IPV4 – FR IPv4 • MSA_PKT_FR_MPLS – FR MPLS

13.8.2 Functions

13.8.2.1 sendPacket

Function	int sendPacket(int npIndex, int portIndex, int32u packetLength, void* thePacket, MsaPktType packetType)
Description	This function is used to send a packet or cell residing in a user buffer out a given NP port or logical TDM channel.
Parameters	• npIndex – index of the NP for the operation • portIndex – index of the desired port • packetLength – length of the packet in bytes • thePacket – user buffer to hold returned statistics

	packetType – type of packet to send
Returns	0 – operation was successful 1 – operation was not successful

Appendix A –

13.9 SEGMENTATION

The performance data for the segmentation module is captured using a packet that flows from the PPP interface to the ATM interface. In this setup, the packets flow from the PPP to ATM, the TDM chunks (IP packet) received on the PPP interface will be segmented into ATM cells as follows:

TDM chunk(s)	ATM cells
1	2
2	3
3	4
4	6

The following table specifies the number of cycles executed corresponding to varying number of TDM chunks with different sizes. The packet flow used for measuring the performance data for Segmentation is PPP – IP – ATM. IP packet received on the PPP interface is segmented into ATM cells and sent via the ATM interface. The packet used for testing the packet flow contains the UDP, IP and PPP headers. The payload data size (including the IP and UDP headers) of the packet received on the segmentation module is specified in this column.

Packet flow and Number of TDM chunks.	Payload Data Size	Number of cycles
PPP – IP – ATM (1 TDM chunk)	60	727
PPP – IP – ATM (2 TDM chunks)	124	883
PPP – IP – ATM (3 TDM chunks)	188	1111
PPP – IP – ATM (4 TDM chunks)	252	1423

13.10 REASSEMBLY

The performance data for the Reassembly module is captured using a packet that flows from the ATM interface to the PPP interface. The packet flow used for measuring the performance data for Reassembly: ATM – MPLS – PPP. MPLS packet received on the ATM interface will be reassembled on the RAS module and will be sent to the MPLS module for label processing. The number of cycles executed for reassembling the varying number of TDM chunks (1 to 4) with different payload sizes on the RAS module will be specified on the following table. The packet used for testing the packet flow contains the MPLS, UDP, IP and ATM headers. The payload data size (including the MPLS, IP and UDP headers) of the packet received on the Reassembly module is specified in this column.

Packet flow and Number of TDM chunks.	Payload Data Size	Number of cycles
ATM – MPLS – PPP (1 TDM chunk)	40	518
ATM – MPLS – PPP (2 TDM chunks)	88	898
ATM – MPLS – PPP (3 TDM chunks)	136	1298
ATM – MPLS – PPP (4 TDM chunks)	184	1698

13.11 MPLS

The performance data for the MPLS module is captured using a packet that flows from the PPP to PPP interface through the MPLS module. The packet flow used for measuring the performance data for MPLS: PPP – MPLS – PPP. MPLS packet received on the PPP interface will be reassembled on the RAS module and will be sent to the MPLS module for label processing. The number of cycles executed for processing the different MPLS actions with different payload sizes on the MPLS module will be specified on the following table. Using this packet flow, the performance is measured for different MPLS actions on the MPLS module. The packet used for testing the packet flow contains the MPLS, UDP, IP and PPP headers. The payload data size (including the IP and UDP headers) of the packet received on the MPLS module is specified in this column.

MPLS Actions	Payload Data Size	Number of cycles
Pop a label	60	788
Pop 3 labels	60	1128
Swap a label	60	808
Swap and Push a label	60	808
Push 2 labels	60	800
Push a label	60	788
Pop and swap	60	972
Pop and Push	60	964

MPLS lookup response waiting time for the pop and swap actions.

MPLS Actions	Payload Data Size	Waiting time
Pop	60	136
Swap	60	140

13.12 TDM RECIRCULATION

The performance data for the TDM Recirculation module is captured using a packet that flows from the PPP to PPP interface through the recirculation module. The packet flow used for measuring the performance data for TDM Recirculation: PPP – IP – PPP. IP packet received on the PPP interface will be sent to the IP module for processing and will be sent to the TDM Recirculation module for PPP header insertion and finally transmitted via the PPP interface. The packet used for testing the packet flow contains the UDP, IP and PPP headers. The payload data size (including the IP and UDP headers) of the packet received on the Recirculation module is specified in this column.

Packet flow and Number of TDM chunks.	Payload Data Size	Number of cycles
PPP – IP – PPP (1 TDM chunk)	60	286
PPP – IP – PPP (2 TDM chunks)	124	373
PPP – IP – PPP (3 TDM chunks)	188	479
PPP – IP – PPP (4 TDM chunks)	252	554

13.13 FR SWITCHING

The performance data for the FR Switching module is captured using a packet that flows from the PPP to PPP interface through the FR switching module. The packet flow used for measuring the performance data for FR switching: PPP – FR – PPP. FR packet received on the PPP interface will be sent to the FR module for DLCI based switching and transmitted via the PPP interface. The packet used for testing the packet flow contains the UDP, IP and PPP headers. The payload data size (including the IP and UDP headers) of the packet received on the FR switching module is specified in this column.

Packet flow and Number of TDM chunks.	Payload Data Size	Number of cycles
PPP – FR – PPP (1 TDM chunk)	60	397
PPP – FR – PPP (2 TDM chunks)	124	484
PPP – FR – PPP (3 TDM chunks)	188	590
PPP – FR – PPP (4 TDM chunks)	252	665

FR lookup response waiting time DLCI based switching.

Packet flow and Number of TDM chunks.	Payload Data Size	Waiting time
PPP – FR – PPP (1 TDM chunk)	60	111 cycles.

IMEM Consumption

CP	Percentage	In KB
TDM Rx	36	11.52
TDM Tx	25	8
TDM Recirc	29	9.28
IP	25	8
Seg	35	11.2
Ras	42	13.44
IP QoS classifier	46	14.72
MPLS	36	11.52
MLPPP	44	14.08
FR		11.04
XP	52	24.96

DMEM Consumption

Cluster wise consumption

Cluster0 98 % (47.04 KB)
It major includes RxCCB blocks and TxCCB blocks. (For 1K channels)

Cluster 1 90% (43.2 KB)
It also includes RxCCB blocks and TxCCB blocks. (For 1K channels)

Cluster 2 96 % (46.08 KB)

Cluster 3 86 % (41.28 KB)

CP consumption

In CP-14 (AAL1 Tx), CP converts Transparent chunk to ATM AAL-1 cell. It maintains SDUs state information in its local DMEM for each configured transparent channel.
For each channel, 20 bytes of state information is maintained. Hence, to support 410 channels, 8K DMEM is needed from its 12k DMEM.

In CP-15 (AAL1 Rx), CPRC converts ATM AAL-1 cell to transparent TDM chunk. It maintains state information in its local DMEM for each configured ATM VCC. For each VCC, 20 bytes of state information is maintained. Hence, to support 410 VCCs, 8K DMEM is needed from its 12k DMEM.

IMA Group is valid only to the physical TDM links (i.e. T1/E1 TDM links). An IMA Group should have atleast 2 TDM links and can have at the maximum of 32 links.

In MSA line card application, there are 2 IMA CPs. Each IMA CP will handle 32 E1/T1 links.

For an IMA CP, the maximum number of links and IMA groups supported are:
Maximum number of links = 32
Maximum number of IMA groups = 16

IMA CP maintains 12 bytes of Link state for every ATM TDM links and 60 bytes of IMA group state for every IMA group.

DMEM for maintaining all link states
= Link state bytes x max. Number of links
= 12 x 32 = 384 bytes
DMEM for maintaining max IMA groups
= IMA group state bytes x max. Number of IMA groups
= 60 x 16 =960 bytes
DMEM usage = 384 +960 = 1344 bytes = 1.313 KB for an IMA CP

Note: DMEM usage for all the above 3 CP components is computed based on the data structures used to maintain table(s) in DMEM.

14 Appendix C – Optimizations done in the application

Reason for allocation of different CPs for TDM Rx and TDM Tx
MSA application configures two CPs for receive and transmit processing of TDM chunks, respectively. The requirement for having Rx and Tx in different CPs arose from the amount of DMEM required to support 1024 channels per cluster.

For receive processing of TDM chunks, 12-Bytes of information is required on a per-channel basis. Similarly, 9-Bytes of information is required for transmit processing. Refer to TdmRxCCB in section 7.4.3.2 and TdmTxCCB in section 7.6.3.2. With 1024 channels per twister, a total of 21KB (12K for TDM Rx control block and 9K for TDM Tx control block) of memory is required which is beyond the available DMEM with one CP. To cater to this restriction, receive and transmit processing of TDM chunks are done at two CPs.

Alternate solution to this restriction would have been to access XPRC DMEM for the TDM control block information of the channel. However, the XPRC DMEM is being used to maintain statistics collected from CPs to make it available to the Host processor.

Reason for providing separate CP for Frame Relay processing
When a HDLC frame is received from the TDM links, it is not possible to classify the packet as PPP/FR in TDM Rx Byte processor itself. This is because, the protocol running on the channel is configurable on per channel basis and this information is

available only to TDM Rx CPRC. So the packet is seen as HDLC frame in TDM Rx Byte processor. RxByte processor extracts initial 10 bytes to the extract space. In TDM Rx CPRC, the packet is classified as PPP/FR. In the case of FR, DLCI lookup needs to be launched. If this happens, since the lookup is launched from Rx CPRC, the utilization of the TDM Rx CPRC is affected because of the round-trip time for the TLU lookup. This is the reason for creating a separate FR CP, which handles the FR specific processing. Another reason would be modularity achieved by the proper segregation of the FR processing.

Reason for doing FR DLCI lookup FR input thread

In the case of FR packets, DLCI lookup is launched in CPRC itself. This is because if the lookup is launched from Rx SDP, The utilization of the re-circulation CP for FR is affected because it does the FR DLCI lookup alone and the FR encapsulation (modification of existing FR header information) cannot be done in the same CP.

Reason for doing MPLS label lookup in MPLS Input thread

Currently MPLS lookup is launched in MPLS Input thread itself. This is because the label related processing could be completed in the receive-thread itself and MPLS encapsulation can be completed in the TxSDP. So when the packet reaches the output thread, it will have the complete packet, which can be sent to appropriate interface's queue.

Reason for supporting 410 channels for transparent chunks

Transparent chunk is converted to ATM AAL-1 cell by AAL-1 Tx CP.It maintains SDUs state information in its local DMEM for each configured transparent channel. For each channel, 20 bytes of state information is maintained. Hence, for supporting 410 channels, 8K DMEM is utilized from its 12k DMEM.

Reason for supporting 410 ATM VCCs for ATM AAL-1 cells

ATM AAL-1 cell is converted to transparent TDM chunk by AAL-1 Rx CP.It maintains state information in its local DMEM for each configured ATM VCC.For each VCC, 20 bytes of state information is maintained. Hence, for supporting 410 VCCs, 8K DMEM is utilized from its 12k DMEM.