Guide to C-Ware Wireless Network Interface Applications

WNI Version 2.0

CALWNIAG-UG/D

Rev 5

REVISION HISTORY

Revision	Date	Description
1	18SEP2002	WNI 1.0.0. release document.
2	04NOV2002	WNI 1.1.0 beta release document. Includes AAL-1, IMA, SONET APS, and more host documentation.
3	27MAR2003	WNI 1.1.0 release document. Updated host API, console commands, and resource utilization.
4	17JUN2003	WNI 1.1.1 release document. Updated host PPP section.
5	15 APR2004	WNI2.0.0 release dicynent.

Table of Contents

1 OVERVIEW

This document describes the purpose, design, and implementation of the C-Ware Wireless Network Interface (WNI) applications. These applications are released as part of the Wireless Network Interface software support for C-Port family-based Wireless Infrastructure product development.

The WNI applications are offered as a package of two C-Port Network Processor (NP) applications and one host processor application. (The host processor application can be configured and built to interoperate with either of the WNI NP applications.) These applications provide comprehensive examples of those network interface functions needed to implement a 2.5G/3G Wireless wniNicIp (BTS/Node B) and a 2.5G/3G Wireless wniNicAtm (BSC/RNC).

The WNI application package depends upon a given release of the C-Ware Software Toolset (CST) but is released independently of the CST product. Specific dependencies and installation instructions for a given release of the WNI applications package are documented in the package's accompanying README file.

The WNI application package consists of a collection of 2.5G/3G network interface protocol components. A protocol component typically maps to one channel processor on the C-3e/C-5e NPU. Each component has well defined interfaces for its initialization and its fast path traffic flow functions. An application can pick and choose these components and map them to channel processors. The WNI release 2.0 consists of two applications **wniNicIp** and **wniNicAtm**. The wniNicIp application contains all the components that are required in an IP network interface in 2.5G/3G network elements. It includes 2x10/100 Ethernet ports, 2xOC-3c POS ports, 16 T1/E1 TDM ports and 1 UL2 ATM backplane port for physical interfaces and ML/MCPPP, PPPMux, IP header compression, DiffServ, AAL5, AAL2 and AAL2 SSSAR protocols. The wniNicAtm contains the components required on an ATM network interface in 2.5G/3G network elements. It includes 2xOC-3c ATM ports, 16 T1/E1 TDM ports and 1 UL2 ATM backplane port for physical interfaces and protocol support for IMA, AAL2, AAL5, AAL2 SSSAR, ATM Traffic Management and AAL2 CID level Qos.

This document will describe each component provided in the WNI2.0 package along with the data and control flow among the components. The component mapping into wniNicIp and wniNicAtm applications is described next. These applications demonstrate the features typical of wireless infrastructure applications.

The WniBaseStation and WniBaseController applications remain unchanged from the previous release (see release WNI1.1.1), and are available separately

Please refer to WNI-README.txt in the release to determine which features are tested in hardware and which features are only tested in software simulation..

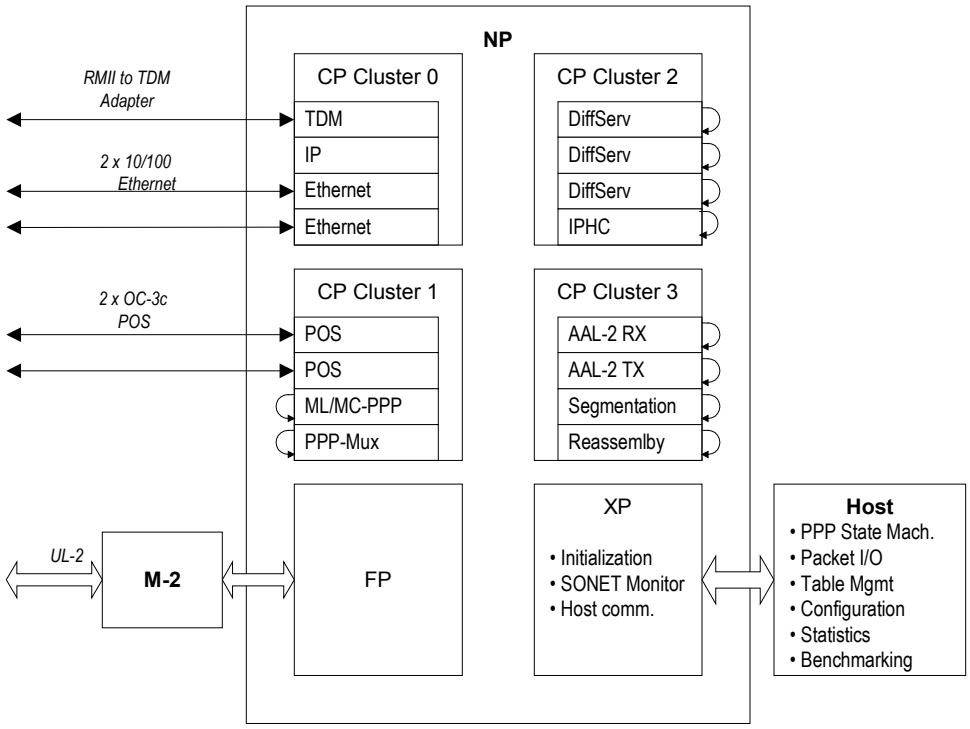
2 Related Documents

Wireless Network Interface Application Fact Sheet
Packet Over SONET to Ethernet Switch Application Guide, CST 2.1.1
AAL-2 Switch Application Guide, CST 2.1.1
ATM Cell Switch Application Guide, CST 2.1.1
AAL-5 SAR to Gigabit Ethernet Switch Application Guide, CST 2.1.1
Application Note: IPv6 Implementation on the C-Port C-5 Network Processor
C-Port APS White Paper
RFC 760, Internet Protocol
RFC 768, User Datagram 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 2460, Internet Protocol, Version 6 (IPv6) Specification
RFC 2472, IP Version 6 over PPP
RFC 2474, Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers
RFC 2475, An Architecture for Differentiated Services
RFC 2507, IP Header Compression
RFC 2509, IP Header Compression over PPP
RFC 2597, Assured Forwarding PHB Group
RFC 2598, An Expedited Forwarding PHB
RFC 2684, Multiprotocol Encapsulation over ATM Adaptation Layer 5
RFC 2686, The Multi-Class Extension to Multi-Link PPP
RFC 2697, A Single Rate Three Color Marker
RFC 3153, PPP Multiplexing
ITU I.361, B-ISDN ATM Layer Specification
ITU I.363.1, B-ISDN ATM Adaptation Layer Specification: Type 1 AAL
ITU I.363.2, B-ISDN ATM Adaptation Layer Specification: Type 2 AAL
ITU I.363.5, B-ISDN ATM Adaptation Layer Specification: Type 5 AAL
ITU I.366.1, Segmentation and Reassembly Service Specific Convergence Sublayer for the AAL Type 2
ITU I.610 B-ISDN Operation and Maintenance Principles and Functions
ATM Forum, Inverse Multiplexing for ATM (IMA) Specification Version 1.1

3 THE WNINICIP APPLICATION

The wniNicIp application delivers functionality needed on an IP network interface of 2.5G/3G network elements. Figure 1 shows the application mapping:

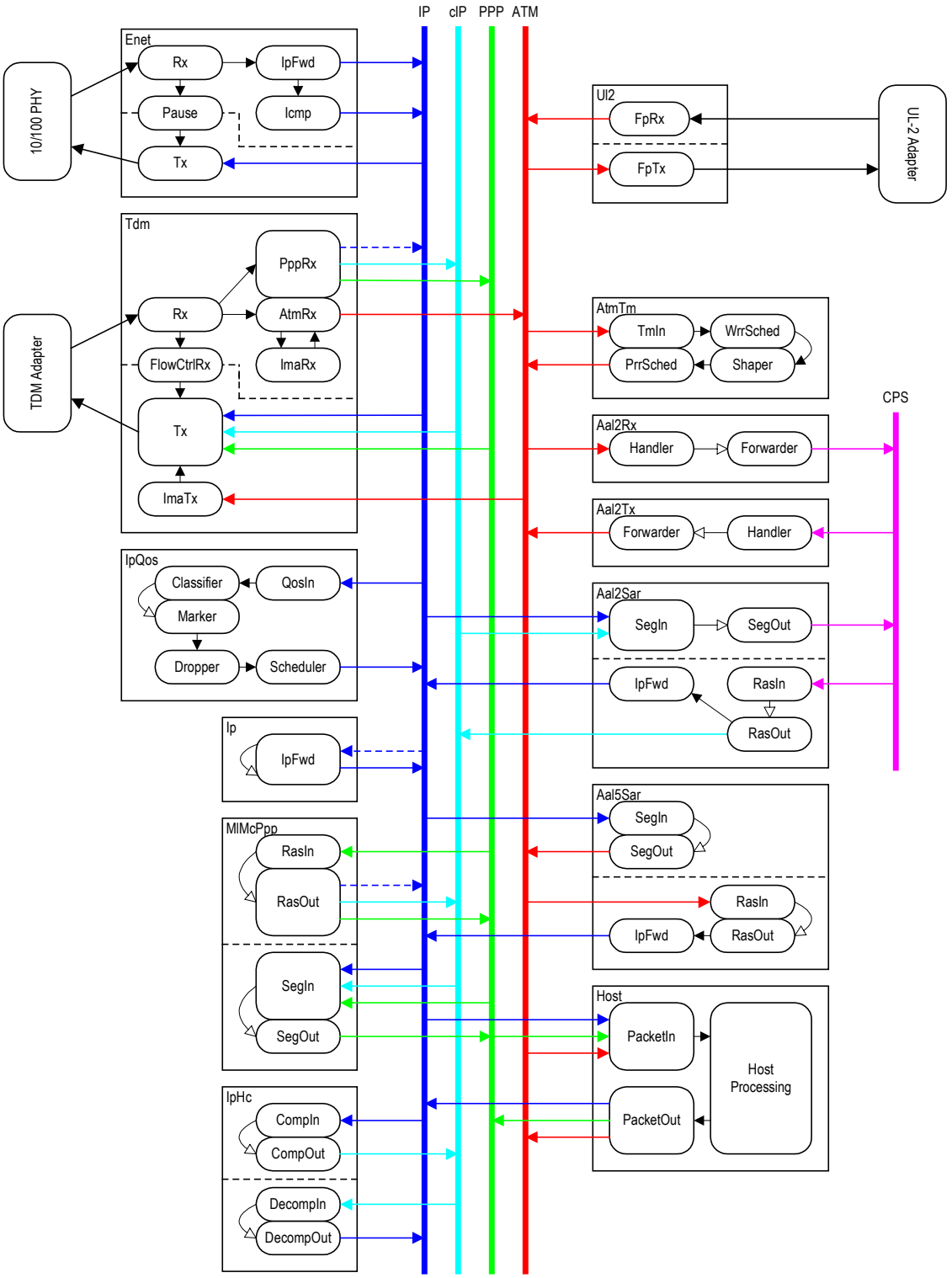
Figure 1 *wniNicIp Functional Mapping*



A UL-2 ATM backplane is assumed in this application. This allows showcasing of IP-ATM interworking while focusing on IP network interfaces to the outside world. There are two Ethernet and two POS components. The TDM component supports 16 T1/E1 lines with up to 128 logical channels. All the TDM channels are configured for the IP protocol suite over PPP. The IP component provides UDP/IP routing lookup services to IP packets received over the TDM PPP links and for AAL-5 IP packets reassembled over the backplane. The two POS and two Ethernet components provide a choice for WAN and LAN connectivity. The ML/MC-PPP component provides Multilink/Multiclass PPP support and the PPP-Mux component provides support for PPP Multiplexing.There are three instances of the DiffServ component. The first instance serves the TDM interfaces, the second instance serves the POS0 port and the third one serves the POS1 port. The IPHC component provides UDP/IP header compression services for data going out the TDM links. AAL2-Rx and TX components provide support for AAL2 CPS packet assembly and CID level switching over the ATM backplane. The segmentation and reassembly components perform AAL5 and SSSAR segmentation and reassembly.

The data paths between all 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 busses implies a different buffer and descriptor format (for IP, ATM, and so on). Figure 3 shows the conceptualized data flow bus .

Figure 2 *Example dataflow in the wniNicIP application*



The interface to a component can be specified by a buffer and buffer descriptor format. Table 1 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 5.24.

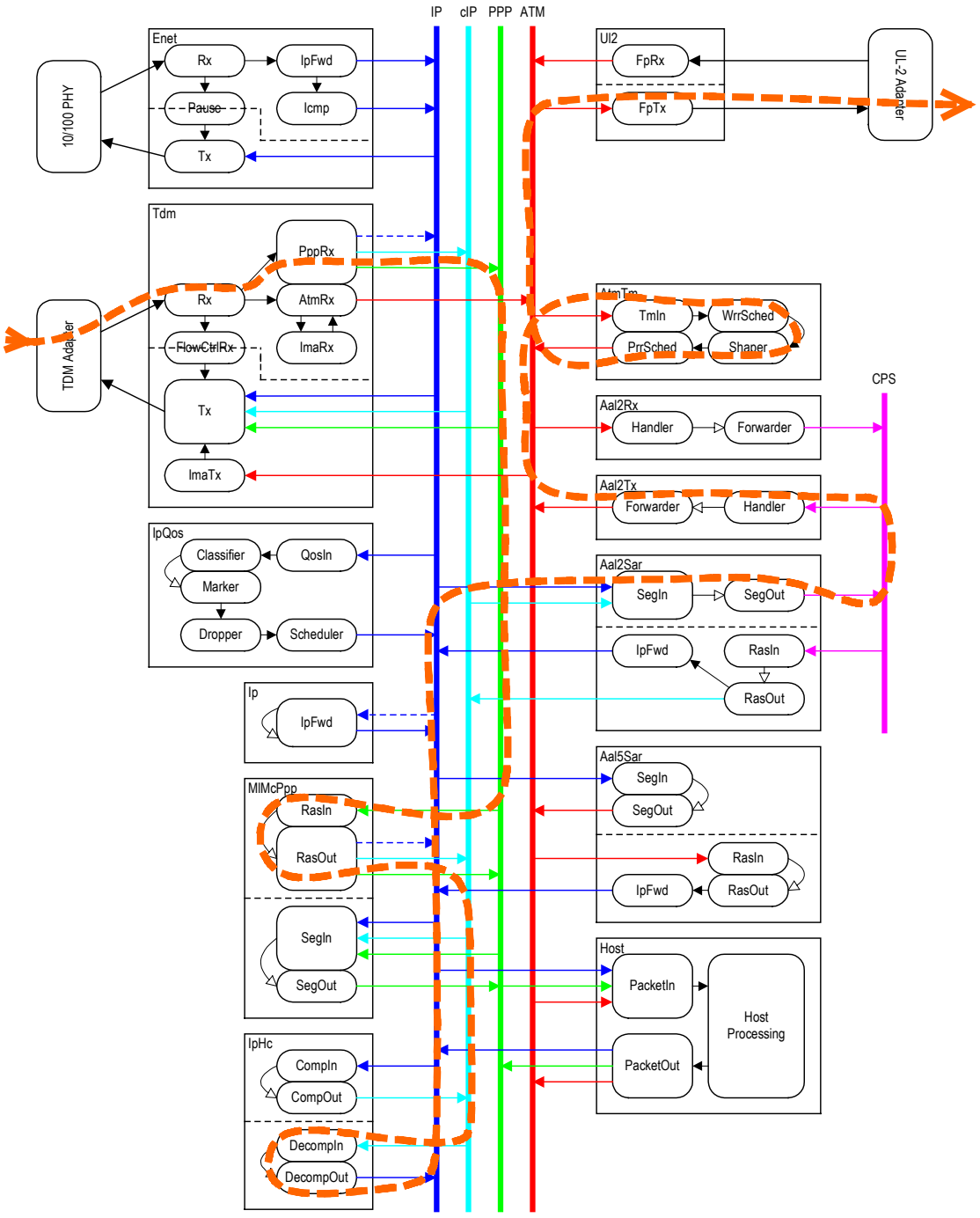
Table 1 *Component Interfaces to Conceptual Bus*

Component	Buffer Formats	Descriptor Formats	Comments
TDM	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/MC-PPP	BT_HDLC, BT_PPP, BT_MLPPP	ML-PPP	port indicates input port; reassembly will be performed
	BT_IPxx, BT_FULL_HEADER, BT_COMPRESSED_xxx, BT_CONTEXT_STATE, BT_PPPMUX, BT_NCP_xxx	TDM	only mcClass field required; segmentation will be performed
PPP-Mux	BT_HDLC, BT_PPP, BT_PPPMUX	N/A	port indicates input port; demultiplexing will be performed
	all others	N/A	multiplexing will be performed
10/100 Ethernet	BT_IPxx	MAC	port is unused
	BT_MAC	N/A	port is unused
IPHC	BT_IPxx	TDM	only egressQueue and compressionId required; compression will be performed
	BT_FULL_HEADER, BT_COMPRESSED_xxx, BT_CONTEXT_STATE	N/A	port indicates input port; decompression will be performed
IP	BT_IPxx	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
Seg.	BT_IPxx	Seg.	cid field is non-zero for AAL-2 segmentation; egressQueue field required only for AAL-5 segmentation
Ras.	BT_ATM	ATM	port indicates input port; only header and vcIndex fields required; AAL-5 reassembly will be performed
DiffServ	BT_CPS	AAL-2	length is AAL-2 LI field; port is unused; only index and uui fields required; index field is the call ID; AAL-2 reassembly will be performed
	BT_IPv4	MAC, TDM	if port > MAX_TDM_CHANNELS, use MAC descriptor, otherwise use TDM descriptor
ATM TM	BT_ATM	ATM	

AAL-2 RX	BT_ATM	ATM	port is unused; only header and vcIndex fields required
AAL-2 TX	BT_CPS	ATM	length is the AAL-2 LI field; index field is the vcIndex
UL-2	BT_ATM	ATM	port is unused; only header field required
Host	BT_ATM	ATM	port indicates the input port; only header field required
	all others	N/A	port indicates the input port

An example data flow scenario is shown in Figure 3. In this example, compressed IP datagrams are received on a ML-PPP bundle. After decompression, the IP packet is segmented through the AAL-2 SSSAR and some level of quality of service is applied before the cells exit through the UL-2 interface.

Figure 3 *Example Data Flow*



3.1 Management functionality on the XP

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 of the system components. The application's 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.

3.1.1 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).

3.1.2 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 .
- Starts some of the SDPs.
- Initializes the SONET monitoring process, if necessary.
- Initializes the OAM processing component.
- Initializes the host communication component.
- Enters the main loop.

The main loop within the XP performs processing for SONET monitoring, if necessary, described in section 3.2.1, OAM handling described in section 5.22, host communication, and the print service.

3.2 SONET Monitoring

Both the WNI2.0 applications support SONET monitoring on the two OC-3c ports. The SDP, CPRC and the XPRC code share SONET error/defect state monitoring responsibility. The SDP is responsible for framing/unframing of ATM payload into SONET envelopes. It detects error conditions and reports them asynchronously to the CPRC as SONET event interrupts. The CPRC examines the interrupts and reports the critical event change notifications to the XPRC. The XPRC performs soak processing on the notifications and reports alarm conditions when defects have persisted over the duration of the soak timer.

The RxSonet and TxSonet SDP microsequencers perform the following SONET tasks:

- SONET OC-3c framing/mapping (RxSonet /TxSonet Block in SDP).
- Identification and notification of defect conditions by the SONET framer (RxSONETblock writes to CREGs). The hardware times out defect conditions.
- Configurable SONET interrupt to notify the CPRC of interesting SONET events.
- Configurable automatic forwarding of RDI-L, RDI-P, REI-L, and RDI-P (via clearing of Manual FEBE bit in SDPMode3).

– Configurable insertion of AIS-L and AIS-P in the SONET transmit stream.
– Visibility into received SONET frames (rxSonet CREGs).
– Ability to specify insert transmit overhead into the SONET transmit stream using the txSONET CREGs.

The CPRC and the XPRC perform the remaining portion of the SONET monitoring. The C-Ware API's SONET Protocol Services include a module that provides access and configuration control to the SONET blocks in the hardware. The SONET monitoring module uses the C-Ware API's SONET Protocol Services to perform the following functions:

– Configure fixed transmit overhead.
– Monitor and soak defects (for example, LOS, LOF, and so on).
– Maintain statistics for B1, B2, B3, REI-L, and REI-P errors

3.2.1 XPRC

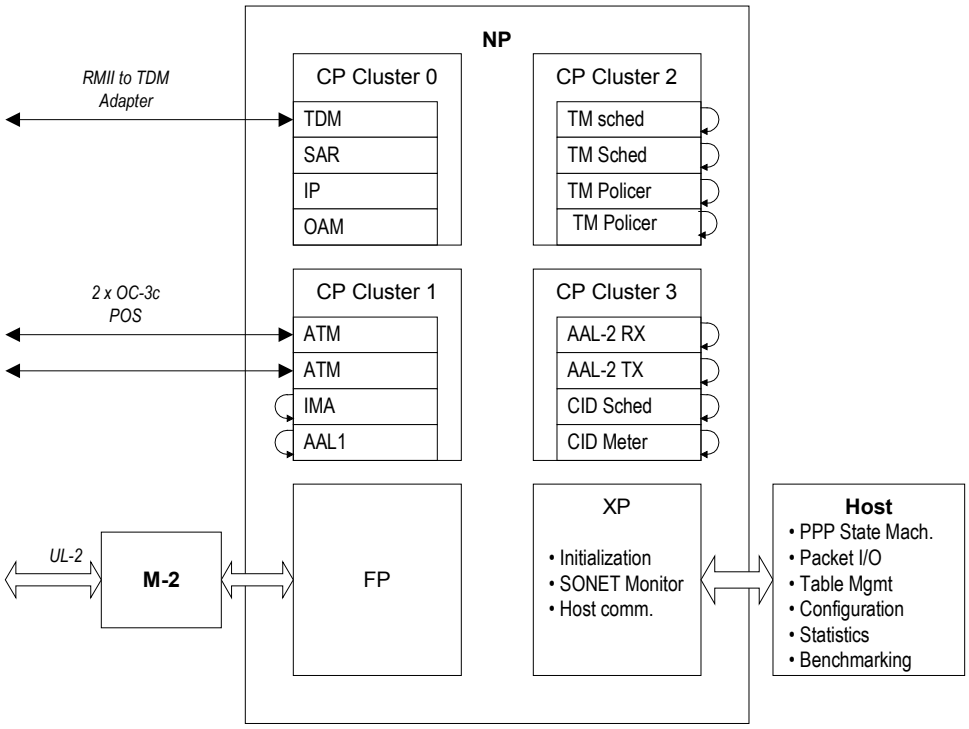
After performing its regular system initialization routines, the XP sits in a tight loop waiting for SONET error notification messages from CP 2 and CP 3. When it receives an event state change notification, it performs the following actions:

– The XP maintains the current state of the defect and whether the defect has been soaked ON or OFF to determine whether the defect has transitioned to the FAILURE state.
– Using a soak timer mechanism determine if a defect state has persisted for long enough that an alarm condition should be raised.
– The alarm condition is cleared when the defect-state has changed to normal and persisted in normal for another soak timer duration.
– A running count of the number of error notifications received is stored locally.

4 THE WNINICATMAPPLICATION

The wniNicAtm application delivers functionality needed on an ATM network interface of 2.5G/3G network elements. The following figure shows the application mapping:

Figure 4 *WniNicAtm application mapping*



The TDM component provides 16 T1/E1 ports with up to 64 logical channels all configured either for ATM or transparent data. The IP component provides UDP/IP routing services to IP packets reassembled over AAL5 and AAL2 SSSAR. The two ATM components provide highspeed WAN interfaces. The backplane is assumed to be UL2 ATM. The OAM component provides forward performance measurement functionality over the two OC-3 and TDM ports. The IMA component supports bundling of TDM links using inverse multiplexing over ATM. The SAR component provides segmentation and reassembly support for AAL5 and AAL2 SSSAR. AAL-1 component implements the circuit emulation service between the OC-3c lines and transparent TDM channels. There are two instances each of the ATM TM policer and scheduler components. The first instance of policer and scheduler provides ATM VC level QoS whereas the second combination provides VP level QoS. The AAL2 Rx and Tx components provide AAL2 CPS packet assembly and CID level switching. The CID QoS component provides QoS for switched AAL2 traffic.

The interface to a component can be specified by a buffer and buffer descriptor format. Table 1 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 5.24.

4.1 Management functionality on the XP

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 of the system components. The application's 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.

4.1.1 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).

4.1.2 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 .
- Starts some of the SDPs.
- Initializes the SONET monitoring process, if necessary.
- Initializes the OAM processing component.
- Initializes the host communication component.
- Enters the main loop.

The main loop within the XP performs processing for SONET monitoring, if necessary, described in section 3.2.1, OAM handling described in section 5.22, host communication, and the print service.

4.1.3 XPRC

After performing its regular system initialization routines, the XP sits in a tight loop waiting for SONET error notification messages from CP 2 and CP 3. When it receives an event state change notification, it performs the following actions:

- The XP maintains the current state of the defect and whether the defect has been soaked ON or OFF to determine whether the defect has transitioned to the FAILURE state.
- Using a soak timer mechanism determine if a defect state has persisted for long enough that an alarm condition should be raised.

– The alarm condition is cleared when the defect-state has changed to normal and persisted in normal for another soak timer duration.
– A running count of the number of error notifications received is stored locally.

5 PROTOCOL COMPONENTS

This section describes each of the componentes in detail and explains how it interfaces with other components. Each component is designed to run on a single CPRC. When the throughput delivered by a single instantiation is not sufficient, applications will instantiatite the component on multiple CPRCs. Such components that can be instantiated multiple times contain standard hooks for load sharing.

This section first describes the physical interface components. A physical interface component is responsible for Layer 1 protocol termination and PDU forwarding to either a Layer 2 protocol component for higher level processing or to another physical interface component for switching. These components read their Rx data from the SDP Rx path and write their Tx data to the SDP Tx path. They send PDUs to other components using QMU message queues.

The protocol components are described next. A protocol component provides Layer-2/3/4 termination and routing. It receives its input PDU over a QMU message queue and typically uses the SDP in recirculation mode. After the higher level protocol processing it sends PDUs to either a physical interface component for TX or to another protocol component for further processing.

5.1 TDM

The TDM component is used to receive and transmit TDM like data. This interface can be configured as 16 T-1/E-1 interfaces supporting up to 128 logical channels. Each channel may carry either ATM or PPP traffic. For more detailed information on the interface to the Channel Adapter device, see its documentation. <<This CP also performs OAM processing as described in section 5.11 ATM OAM.>>

Note the following:

- 32 byte user chunk size is not supported.
- The maximum number of channels supported is 128.
- Transparent data chunks are not supported.
- 8 bit GMII mode is available.
- Flow control is disabled through define of the flag TDM_DISABLE_FLOW_CONTROL in tdmIf.h
- Loopback can be enabled by defining TDM_LOOPBACK_ON in tdmIf.h.
- For extended debug printfs define the flag DEBUG_TDM in tdmIf.h.

5.1.1 SDP

The SDP moves data from the TDM interface adapter to the RC in the receive direction and from the RC to the TDM interface adapter in the transmit direction. The functions provided by each of its component processors are described below.

5.1.1.1 RxBit

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

- Determines chunk delineation through RX_EN signal (wired to PhyStatus0 signal in the rxBit processor).
- Determines whether incoming chunk is a flow control or data chunk.
- Initializes byte counter in internal register to 64 for data chunks or to 16 for flow control chunks.
- Receives bytes from TDM adapter interface and increments byte counter for each byte received.
- Passes bytes to RxSync while calculating BIP-8.
- When byte counter reaches all ones, sends status code with data9 set indicating BIP-8 or short/long chunk length error indication.
- RxBit is not configurable through its control space.

5.1.1.2 RxSync

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

- Transparently passes bytes from RxBit to RxByte.
- For ATM cells, calculates the CRC-10 for all ATM cells.
- For ATM cells, if status code from RxBit indicates no error, passes an indication of CRC-10 validity with data9 set after the cell.
- RxSync is not configurable through its control space.

5.1.1.3 RxByte

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

- Receives bytes from the RxSync processor.
- Writes chunk header information (chunk type, channel, channel type, length) to extract space.
- Determines whether a user or a flow control chunk has been received.
- Flow control chunk processing:
  - Verifies flow control chunk valid bit
  - Writes flow control chunk count to extract space.
  - Indicates processing complete to the RC.
- User chunk processing:
  - For PPP chunks, outermost PPP encapsulation is identified including protocol, compression, PPP buffer type (destination), protocol, protocol length and the PPP header is written to extract space and forwarded to DMEM.
  - For ATM cells, launches lookup into ATM VPI/VCI table for user cells. Identifies and reports OAM/RM cells. Writes ATM cell header to extract space.
  - Payload is written to DMEM.
  - For ATM cells, writes the payload to extract as well.
- When data9 is received, writes the chunk status code to extract space and switches scope.

The RC writes to RxByte control space to configure the operation of the RxByte processor. The data structure is described in section 5.1.3.2.

The RxByte processor writes information about incoming chunks into extract space for the RC to use in its processing. The data structure is described in section 5.1.3.3.

5.1.1.4 TxByte

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

- Reads chunk header information (channel Id and channel type) from merge space and transmits chunk header.
- For ATM SOM chunks, reads ATM cell header from merge space, adds HEC and transmits.
- For ATM OAM cells, generates CRC-10.
- Reads payload from DMEM.
- Transmits payload bytes and padding until byte counter reaches zero.
- Switches scope.

The RC writes to TxByte control space to configure the operation of the TxByte processor. The data structure is described in section 5.1.3.4.

The RC writes information about outgoing chunks to merge space for TxByte to use in its processing. The data structure is described in section 5.1.3.5.

5.1.1.5 TxBit

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

- Sends idle bytes (0x7E) without TX_EN set when no chunk is available from txByte.
- Receives bytes from the TxByte processor. A byte received with Data9 set signifies the end of a chunk.
- Calculates BIP-8 on each byte and inserts after final byte received with Data9 set.
- Sends all bytes of the chunk to TDM adapter interface with TX_EN set.

5.1.2 RC

The RC performs higher level processing of chunks, including reassembling chunks into packets, segmenting packets into chunks, receiving ATM cells, maintaining statistics and making forwarding decisions. The functions provided by each of its components are described below.

5.1.2.1 Initialization

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

- Initializes statistics and chunk reassembly/segmentation control structures.
- Initialize SDP control space, txCB/rxCB DMA control blocks, SDP ATM VC lookup txMsg registers.

5.1.2.2 Receive

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

- Waits for indication of header processing completion.
- Processes chunk based on chunk type (flow control or user) in extract space after checking for errors.
- If user, processes user chunk based upon channel type (ATM or PPP) in extract space.
- Gives scope back to SDP.

5.1.2.2.1 ATM RX

The ATM RX component 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 counter 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 AAL-2/5 cells, enqueues descriptor to appropriate reassembly queue.
- For other cells, launches lookup into port table.
- Waits for port lookup to complete.
- Enqueues descriptor to egress queue or QoS queue indicated by port lookup.

The descriptor format used to transport ATM cells to the next functional block has the following data structure:

```
struct {
   BsBufferHandle bufHandle;
   int16u         length;
   int16u         port_bufType;
   CellHeader     CellHeader;
   int16u         vcIndex;
   int16u         egressQueue;
} descriptor;
```

5.1.2.2.2 PPP RX

The PPP RX component handles chunks of PPP packets. Specifically, it does the following:

- For SOM chunks, allocates new buffer and determines destination queue based on PPP protocol field value (host queue (LCP/NCP), IP queue, ML-PPP queue, or IPHC queue).
- For non-SOM chunks, retrieves reassembly state from DMEM (RxCCB data structure).
- Initiates payload transfer from DMEM to SDRAM if no error indicated in chunk.
- For EOM chunks, builds descriptor.
- For non-EOM chunks, updates reassembly state in DMEM.
- Waits for payload transfer to complete.
- For EOM chunks, enqueues descriptor to the destination queue.

The descriptor format used to transport PPP packets to the next functional block has the following data structure:

```
struct {
   BsBufferHandle bufHandle;
   int16u         length;
   int16u         port_bufType;
   int32u         mlHeader;
} descriptor;
```

The state maintained for each PPP packet being reassembled has the following data structure:

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

5.1.2.2.3 Flow Control RX

The Flow Control RX component handles flow control chunks. Specifically, it does the following:

- Updates the credit counter for the channel specified in extract space with the free chunk value specified in extract space.
- Counts of the credits available to each channel are maintained in the following data element of the TdmTxCCB data structure:
  `int8u chFlowChunksAvail`

5.1.2.3 Transmit

The transmit component services the TDM channel egress queues and passes chunks to the SDP. Specifically, it does the following:

- Checks channels round robin for credits available (chFlowChunksAvail).
- When credits are available retrieves segmentation state from DMEM for the matching channel.
- If no segmentation for channel is in progress, dequeues descriptor from QMU queue.

– Waits for scope to be available from the SDP.
– Fills in merge space with chunk header information and channel type specific information.
– Waits for payload transfer of previous chunk to complete.
– Frees buffer associated with previous chunk if it was an EOM chunk.
– Initiates payload transfer from SDRAM to DMEM.
– Updates segmentation state in DMEM.
– Decrements credit counter for the matching channel.
– If EOM chunk, resets state information.

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

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

5.1.3 Data Structures

5.1.3.1 Config Parameters

The TDM component needs the following configuration parameters to be set from the XP:

– TDM port number.
– Buffer pool # of the pool to be used for Rx frames.

5.1.3.2 RxByte Control Space

The CPRC configures the RxByte processor through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	recirc	pad[0..7]		
4				

recirc – recirculation mode
pad – unused

5.1.3.3 Extract Space

RxByte writes information about received packets into extract space for the RC to use in its processing. The first 6 words of the data structure have the following format:

Byte Offset:	0	1	2	3
0	type	ChanID	ChanType	userValid
4	flowChunkCnt	chunkStatus	droppedChunks	pad

type– indicate user type or flow control
chanId– channel ID
chanType- PPP, ATM, or Transparent
userValid- validate user
flowChunkCnt- flow control counter
chunkStatus- indicates good, short, long, or error with the chuck.
DroppedChunks- chuck drop indicator
Pad- unused

The format of the remainder of extract space depends on the type of frame received. For ATM cell, the remainder of extract space has the following format:

Byte Offset:	0	1	2	3
8	header			
12	bip16		encodedPti	Payload[48]
…	…			
60	…			

header – ATM header
bip16–
encodedPti –
Payload – user payload

For PPP packets, the remainder of extract space has the following format:

Byte Offset:	0	1	2	3
8	userInd	crcInd	chunkLength	protocolLen
12	bufferType	pad1	protocol	
32-44	mlHeader			

userInd – indicate this is BOM, EOM, COM, or BOMEOM
crcInd – CRC 16 or 32
chunkLength – chuck lenght
protocolLen –
bufferType –
pad1– unused
protocol –
mlHeader – Multilink header

5.1.3.4 TxByte Control Space

The CPRC configures the TxByte processor through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	recirc	pad		
4	pad			

recirc – number of bytes to be used as the Tx Fifo setting to prevent fifo underruns.
pad – unused

5.1.3.5 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	type	ChanID	userChunkLength	userInd

type– indicate user type or flow control
chanId– channel ID

userChunkLength – chuck length indicator
userInd - indicate this is BOM, EOM, COM, or BOMEOM

The format of the remainder of merge space depends on the type of frame transmit. For ATM cell, the remainder of merge space has the following format:

Byte Offset:	0	1	2	3
4	atmHeader			
8	txType	AtmPad[3]		
12	atmPayload[48]			
…	…			
56	…			

atmHeader – ATM header
txType – OAM cell or not.
AtmPad – unused
atmPayload – user payload

For PPP frame, the remainder of merge space has the following format:

Byte Offset:	0	1	2	3
8	Header(4B)/Pad1(1B)			
12	headerLen			

PppRecirculation will have Header(4B) and headerLen(1B). Otherwise regular PPP frame will just have pad1(1B).

5.1.3.6 TDM Interface Statistics

Statistics are maintained in the following data structure.

Byte Offset:	0	1	2	3
0	chRxChunks		chRxPdus	
4	chRxBytes		chRxBip8Errs	
8	chRxLenErrs		chRxInvalidErrs	
12	chRxCrcErrs		chRxFlowChunks	
16	chRxLookupErrs		chTxChunks	
20	chTxPdus		chTxBytes	

5.2 Ethernet

The Ethernet component is used to receive and transmit IPv4 and IPv6 datagrams. It also responds to and may generate MAC pause frames for flow control. It uses the two TLU PFX tables viz., IPv4 and IPv6 for IP packet routing.

5.2.1 SDP

The SDP for Ethernet is responsible for decapsulation/encapsulation of MAC, IPv4, IPv4/UDP and IPv6 framing. The receive processing is also responsible for detecting errors with the frame and launching the initial lookup required by the CPRC for routing. The transmit processing is responsible for decrementing the IPv6 Hop Limit or IPv4 header TTL field, adjusting the IPv4 header checksum accordingly, and generating the final CRC value for transmission.

5.2.1.1 RxBit

The RxBit processor performs the following functions:

- Serial to parallel conversion – the 8bit data is converted from serial to parallel format for processing by RxBit.
- Preamble detection – RxBit is programmed to detect the Ethernet preamble and start of frame (SOF) delimiter. After the bit pattern is detected, these bytes are stripped off of the frame before additional RxSDP functional parsing of the Ethernet frame.
- PCS receive state machine – RxBit implements the Physical Coding Sub-layer (PCS) receive state machine. That is, RxBit detects the start of frame and end of frame control characters. It also validates that the control characters are received at the appropriate time (such that no idle characters are received between the start and end of frame).
- Frame length validation – the RxBit validates that incoming frames are in the range of 64 to 1518 bytes.

The RC configures RxBit through a data structure mapped into its control space. The data structure is described in section 5.2.3.2.

5.2.1.2 RxSync

RxSync is configured to be in pass-through mode.

5.2.1.3 RxByte

The RxByte processor performs the following functions:

- Header detection – the RxByte processor detects the start of the layer 2 MAC header, layer 3 IPv4 and IPv6 headers and layer 4 IPv4/UDP header of incoming Ethernet frames. If the MAC DA does not match the internal MAC address of the Ethernet node, the packet is dropped.
- Header validation – RxByte validates the layer 3 IPv4 headers of the incoming Ethernet frame. The RxByte processor validates the IP header with the exception of the IP DA, which is done by virtue of not installing illegal IP Addresses into the Table Lookup Unit (TLU), which results in failed lookup results from the TLU. If the packet is IPv4/UDP, then the UDP checksum verified.
- Initiation of IPv4 DA, IPv4/UDP and IPv6 Lookups – the RxByte processor launches lookups to the TLU for evaluation by the CPRC while the packet is being received.
- Detection of control frames – RxByte detects control frames such as flow control frames and autonegotiation pages so that the CPRC can process them accordingly.
- Validation of the Ethernet CRC – RxByte's CRC-32 engine calculates an ongoing Ethernet CRC for the received frame and verifies that the CRC on the frame is correct.
- Determination of CPRC forwarding path – the RxByte processor provides information to the CPRC informing it of what forwarding path (such as unicast, routed, multicast, broadcast) to use for the frame.
- Error and status reporting – the RxByte processor program reports errors and frame status to the CPRC via extraction space. This includes any physical layer error, CRC, frame length, header validation errors, and so on.

The RC configures RxByte through a data structure mapped into its control space. The data structure is described in section 5.2.3.3. The RxByte processor writes information about incoming data packets into extract space for the RC to use in its processing. The data structure is described in section 5.2.3.4.

5.2.1.4 TxByte

The TxByte processor performs the following functions:

- Transmits bytes – the TxByte processor reads bytes from CP DMEM and then loads the bytes into the large FIFO in order to stage transmission to the TxBit processor.

– Frame modification – in the case of IPv4 and IPv6 routing, the TxByte processor modifies the data link header according to the MAC DA and MAC SA in Merge Space, which was populated by the CPRC. In the case of IPv4, the TxByte processor decrements the TTL, and updates the IPv4 Checksum. In the case of IPv6, the TxByte processor decrements the Hop Limit.
– CRC generation – the Ethernet CRC is recalculated and appended to the end of the Ethernet frame.
– Ethernet padding – if the length of the Ethernet frame to be transmitted is less than 60 bytes (without the CRC), the TxByte processor pads the frame out to 60 bytes to meet minimum Ethernet packet size requirements.
– CPRC coordination – for transmit error handling (such as collision detection when operating in half-duplex mode), the RxByte processor coordinates and communicates status information with the CPRC via both Merge Space and Control Space.

The RC configures TxByte through a data structure mapped into its control space. The data structure is described in section 5.2.3.5. The RC writes information about outgoing data packets into merge space for TxByte to use in its processing. The data structure is described in section 5.2.3.6.

5.2.1.5 TxBit

The TxBit processor performs the following functions:

– IFG generation – the TxBit processor generates the correct Inter-Frame Gap (IFG) according to the speed of the physical interface.
– Preamble generation – the TxBit processor generates both the Ethernet preambleand start-of-frame delimiter for the Ethernet frame.
– PCS transmit state machine – the TxBit processor implements the PCS transmitstate machine, Inserting the start-of-frame, end-of-frame, and idle characters into the Ethernet data stream.

The RC configures TxBit through a data structure mapped into its control space. The data structure is described in 5.2.3.7.

5.2.2 RC

The RC performs higher level processing of data packets to determine where the packet descriptor that contains information about the packet and the buffer id to the actual packet in memory needs to be sent to next. The Ethernet RC also receives packet descriptors of Ethernet packets that need to be transmitted and de-queues the packet and transmits it. The threads that run on the Ethernet RC are described next.

5.2.2.1 Initialization

During initialization the following things happen:

Buffer pools, queues, port numbers, internal MAC address, internal IP address, table lookup entries, interrupts, SDP PHY parameters and various internal data structures are all initialized.
launches other threads – receive and transmit.

5.2.2.2 Receive

This thread handles incoming data packets. Specifically, it does the following:

– If a packet is received from the RxByte processor that contained a frame or header error (detected by RxByte) statistics are updated and the frame is dropped.
– If a packet is received from the RxByte processor that contains a expired IPv4 TTL (detected by RxByte) statistics are updated and an ICMP TTL Time expired packet is sent to IP SA of the sender
– If the packet received was a Multicast, Broadcast, 802.1 Tagged, Mac type not IPv4, IPv6 or Mac Control the receive thread passes the packet descriptor to the Host CPU.

– If the received packet is a Mac Control pause frame, this thread will command the TxBit processor to pause for the requested amount of time.
– For received IPv4 or IPv4/UDP packets, this thread will wait for the TLU to respond with the IPDA/UDP lookup result.
– If the lookups fails, the packet is dropped and an ICMP destination unreachable packet is sent in response.
– If the lookup succeeds, the port number returned from the IPDA/UDP in the result is then used to launch a TLU lookup into the port table. The port table lookup will always return a valid result, which is then used to determine which Queue to post the packet descriptor. If the port is not currently valid (offline) the packet is dropped.

For received IPv6 packets, this thread will wait for the TLU to respond with the first IPv6 DA lookup result.

– If the first lookups fails, the packet is dropped.
– If the first lookup succeeds, the second IPv6 DA lookup is launch with the tag value from the first lookup and the next 64bits of the IPv6 DA. This thread then waits for the result.
– If the second lookup fails, the packet is dropped.
– If the second lookup succeeds, the port number returned from the second IPv6 DA lookup result is then used to launch a TLU lookup into the port table. The port table lookup will always return a valid result, which is then used to determine which queue to post the packet descriptor. If the port is not currently valid (offline) the packet is dropped.

This thread also keeps track of various receive statistics on packet size, drops etc.
If the receive thread can't forward a packet descriptor due to queue's being full, this thread will build a pause frame and queue it to the TxByte processor sending a pause frame with a default pause time.

5.2.2.3 Transmit

This thread handles outgoing data packets. Specifically, it does the following:

This thread waits for packet descriptors to be queue that it needs to transmit. The incoming packets are already formatted (IPv4, IPv4/UDP, IPv6, IPv4/ICMP or Pause frame) and this thread will simply fill in merge space with the appropriate MAC header values, type of packet and notify the TxByte processor to transmit the packet.
This thread also keeps track of various Transmit statistics on packet size etc.

5.2.3 Data Structures

5.2.3.1 Configuration parameters

The Ethernet component needs the following configuration parameters to be set from the XP:

– Ethernet port number.
– Buffer pool # of the pool to be used for Rx frames.

5.2.3.2 RxBit Control Space

The CPRC configures the RxBit processor through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	idleLamp	DuplexMode	wireSpeed	cpRcReady
4	pad			

idleLamp – unused
duplexMode – configuration setting for duplex mode - full or half duplex
wireSpeed – wirespeed indication 10Mbit or 100Mbit
cpRcReady – unused
pad – unused

5.2.3.3 RxByte Control Space

The CPRC configures the RxByte processor through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	ifNum	macInt[0..2]		
4	MacInt[3..5]			pad

ifNum – interface number of this port
macInt[6] – internal MAC address of this port
pad – unused

5.2.3.4 Extract Space

RxByte writes information about received packets into extract space for the RC to use in its processing. The first 6 words of the data structure have the following format:

Byte Offset:	0	1	2	3
0	hdrStatus	FrameStatus	rxPath	badFrameCount
4	macDaHi			
8	macDaLo		pad	
12	macSaHi			
16	macSaLo		pad	
20	macType		frameLen	

hdrStatus – status byte indicating success or failure status of header parsing
frameStatus – status byte indication success or error code of entire frame parsing
rxPath – indicates the code path the receive function should follow
badFrameCount – a count indicating the number of frames that were dropped because the SDP did not have scope
macDaHi, macDaLo – the received MAC destination address
macSaHi, macSaLo – the received MAC source address
macType – the MAC protocol field
frameLen – count of number of bytes in the frame
pad – unused

The format of the remainder of extract space depends on the type of packet received. For IPv4 packets, the remainder of extract space has the following format:

Byte Offset:	0	1	2	3
24	version	IHL	tos	pad
28	length		id	
32	flags	pad	fragOffset	
36	ttl	protocol	checksum	
40	srcaddr			
44	destaddr			
48	udpSrcPort		udpDestPort	

version – IP version
IHL – header length
tos – type of service field
pad – unused
length – IP total length

id – IP identification field
flags – IP flags field
fragOffset – IP fragmentation offset field
ttl – IP time to live field
protocol – IP protocol field
checksum – IP header checksum field
srcaddr – IP source address
destaddr – IP destination address
udpSrcPort – UDP source port
udpDespPort – UDP dest port

For IPv6 packets, the remainder of extract space has the following format:

Byte Offset:	0	1	2	3
24	vers_tclasshi	tclasslo_flowhi	flowlo	
28	len		nexthdr	ttl
32-44	srcaddr			
48-60	destaddr			

vers_tclasshi – IPv6 version and 4 MSBs of traffic class
tclasslo_flowhi – 4 LSBs of traffic class and 4 MSBs of flow ID
flowlo – remaining bits of flow ID
len – IPv6 total length
nexthdr – IPv6 next header field
ttl – IPv6 hop limit field
srcaddr – IPv6 source address
destaddr – IPv6 destination address

5.2.3.5 TxByte Control Space

The CPRC configures the TxByte processor through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	fifoPrimeDepth	pad	DroppedFrames	
4	pad			

fifoPrimeDepth – number of bytes to be used as the Tx Fifo setting to prevent fifo underruns.
droppedFrames – number of frames the TxByte processor has dropped
pad – unused

5.2.3.6 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	pauseTime		txAlgorithm	datalinkHdrSize
4	macDaHi			
8	vlagTag		macDaLo	
12	macSaHi			
16	pad		macSaLo	

pauseTime – for pause frames, the amount of time the peer is requested to pause

txAlgorithm – indicates the processing that TxByte should apply to the packet
datalinkHdrSize – the size of the L2 header
macDaHi, macDaLo – the outgoing MAC destination address
vlanTag – the outgoing VLAN tag to apply (unused)
macSaHi, macSaLo – the outgoing MAC source address
pad – unused

5.2.3.7 TxBit Control Space

The CPRC configures the TxBit processor through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	Cmd	numMaxColl…	pauseTime	
4	NumLateCollisions		duplexMode	wireSpeed

cmd – command being sent to TxBit processor
numMaxCollisionsExceeded – count of the number of times max collisions detected
pauseTime – time value used when executing a pause frame
numLateCollisions – count of the number of late collisions (out of window) detected
duplexMode – full or half duplex
wireSpeed – wire speed setting 10Mbit or 100Mbit

5.3 OC-3c ATM

This component is used to receive and transmit ATM cells. The CPRC module consists of two functions, namely atmRx and atmTx running in two separate contexts. The Rx context receives cells from the SDP and forwards them based on the results of ATM VC table lookup on the received cell header. The Tx context feeds cells into the SDP for transmission. This CP also performs Sonet monitoring as described in section 5.11.

5.3.1 SDP

5.3.1.1 RxBit

The RxBit processor gets the bit stream from the PHY and passes it to SONET processor, which strips off all the SONET overhead data. The RxBit is responsible for detecting a SONET frame and notifying the framer once it has found the correct A1/A2 sequence. The RxBit processor is not configurable through control space.

5.3.1.2 RxSync

After the SONET framer strips SONET overhead, RxSync then extracts cells through Header Error Control (HEC) recognition. Immediately after successful cell header recognition, RxSync starts de-scrambling cells and checks to see if the cell is unassigned/idle (VPI=0 and VCI=0). If so, then the cell is discarded, and no further processing is necessary. All the other cells are forwarded to RxByte along with CRC10 flag. This flag eventually makes its way to CP which will discard all errored cells. The RxSync processes gets its configuration parameters from control space as defined in section 5.3.3.2.

5.3.1.3 RxByte

The RxByte processor parses the cell header to determine the cell type. The XP assigns a unique port number to each ATM port. This number is written to the RxByte control space as defined in section 5.3.3.3. When a cell arrives at the RxByte processor, it performs the following sequence of operations:

- Create an ATM VC table lookup key using the cell header and the port number.
- Examine the PTI field of the ATM header to determine the cell type which may be user data, OAM, RM etc. This is done using a CAM match on the PTI bits.
- For all user data cells, a VC table lookup is launched. The VC table key consists of the entire cell header plus the port number from the control space.
- The cell header and cell status is written out to the extract space.

The extract space data structure is described in section 5.3.3.4.

5.3.1.4 TxByte

Upon receiving a scope, TxByte generates the cell header and sends it to the large FIFO followed by the cell payload. It performs the following sequence of operations on every cell given to it for transmit:

- Read the cell header information from the merge space, create the 4-byte header and transmit it.
- Read the txType indicator. If it indicates an OAM cell, CRC 10 should be accumulated on each byte of the payload data.
- Read payloadLength field from the merge space and transmit that many bytes of payload data.
- If it is an OAM cell, append the accumulated CRC to the end of the payload.

The merge space data structure is described in section 5.3.3.5. TxByte is not configurable through control space.

5.3.1.5 TxBit

TxBit monitors the data stream for out of frame signal and transmits the data to the output FIFO, as a bit-stream. TxBit is not configurable through control space.

5.3.2 RC

The RC performs higher level processing of cells. It creates the cell descriptor containing the buffer handle to the cell payload and other relevant contextual information associated with the cell. The descriptor is the forwarded to other CPs for higher layer processing. The threads that run on the ATM RC are described next.

5.3.2.1 ATM Receive

The ATM Rx function waits for results of the ATM VC table lookup launched by the RxByte. Depending on the results of this lookup the cell is forwarded to either the AAL2 Rx CPs or the AAL5 RAS CP or switched to the other ATM CP or the TDM CP. It performs the following sequence of operations on each cell:

- Wait for an Rx scope and read the cell header and cell status when the scope is available. If the cell status is set to an error, then discard the current cell.
- A message descriptor is created with bufType set to BT_ATM and the buffer handle referring to the payload SDRAM buffer of the newly arrived cell.
- Wait for the ATM VC table lookup to return. This lookup contains information on how the cell should be treated. If the AAL5 flag bit is set in the flags field of the TLU lookup results, the cell is forwarded to the RAS CP. If the AAL2 flag bit is set it is forwarded to either one of AAL2 Rx0 CP or AAL2 Rx1 CP depending on the vcIndex. If the vcIndex is lesser than 256, it is forwarded to AAL2 Rx0 CP and otherwise its forwarded to AAL2 Rx1 CP.
- If the cell status field indicates that this is an OAM or RM cell, the cell is forwarded to the host through the XP. On OAM cells, BIP-16 calculation and verification can be performed optionally. A flag in the VC table lookup

results indicates whether BIP-16 calculation should be performed. The cell is forwarded to the XP irrespective of BIP-16 status. BIP-16 errors are tracked along with other ATM level statistics.

– If no flag bits are set, this cell needs to be switched out to another ATM port. The egress cell header from the VC table is written into the ATM cell header field of the message descriptor. The CPRC then launches a port table lookup on the egress port returned in the VC table lookup. The destination queue is determined from the port table lookup and the cell is forwarded on to that queue.

5.3.2.2 ATM Transmit

The ATM transmit function runs the following sequence of steps:

Wait for a cell to arrive on the input queue. Dequeue it and read the cell header and cell type.
Allocate a Tx scope and copy the cell header, cell type and payload length in the merge space.
Release the Tx scope thus starting the TxByte operations on the cell.

5.3.3 Data Structures

5.3.3.1 Configuration parameters

The ATM component needs the following configuration parameters to be set from the XP:

– ATM port number.
– Buffer pool # of the pool to be used for Rx cells.

5.3.3.2 RxSync Control Space

The RxSync processor is configured through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	deltaCnt	AlphaCnt	state	pad
4	pad			

deltaCnt – specifies the number of cell header errors that the RxSync will tolerate before declaring a synchronization loss on the OC-3c link.
alphaCnt – specifies the number of good cells that should be seen before RxSync goes in the ATM sync state.
state – tells the current state of the link.
pad – unused.

5.3.3.3 RxByte Control Space

The RxByte processor is configured through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	PortNumL	PortNumU	pad	
4	pad			

portNumL – lower half of the port number corresponding to this CP
portNumU – upper half of the port number corresponding to this CP
pad – unused

5.3.3.4 Extract Space

RxByte writes information about received cells 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	cellHeader			
4	VccIndex		pduHdrStatus	congestDrops
8	crc10Indicator	EncodedPti	camValue	pad

cellHeader – the received cell header
vccIndex – unused
pduHdrStatus – indicates success or error code following header processing
congestDrops – count of cells dropped due to congestion (RxByte could not get scope)
crc10Indicator – indicates whether or not the CRC-10 value was correct (applies only to OAM cells).
encodedPti – the PTI field from the cell header
camValue – used for OAM debug
pad – unused

5.3.3.5 Merge Space

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

Byte Offset:	0	1	2	3
0	cellHeader			
4	PayloadLength		txType	pad

cellHeader – the outgoing cell header
payloadLength – negated payload length of the ATM cell, should always be –48
txType – non-zero for OAM cells, in which case CRC-10 is applied
pad – unused

5.4 OC-3c Packet Over Sonet (POS)

These POS component is used to receive and transmit IPv4 and IPv6 packets encapsulated in PPP. The POS CPRC module consists of two functions, namely posRx and posTx running in two separate contexts. The Rx context receives IP packets from the SDP and forwards them based on the results of IP routing table lookup on the received packet header. The Rx context is also responsible for generating ICMP packets in response to IP level error conditions. The Tx context feeds packets into the SDP for PPP transmission. This CP also performs Sonet monitoring as described in section 5.11.

5.4.1 SDP

5.4.1.1 RxBit

The RxBit processor gets the bit stream from the PHY and passes it to SONET processor, which strips off all the SONET overhead data. The RxBit is responsible for detecting a SONET frame and notifying the framer once it has found the correct A1/A2 sequence. The RxBit processor is not configurable through control space.

5.4.1.2 RxSync

After the SONET framer strips SONET overhead, RxSync then extracts PPP frames using the HDLC frame delimiters. It also destuffs the PPP frame to recover the IP payload. RxSync then proceeds to verify that the packet is not larger than the supported MTU and not shorter than the minimum number of bytes required for IP and TCP/UDP headers. If the packet fails any of these tests it is dropped. RxSync gets its configuration parameters from control space as defined in section 5.4.3.2.

5.4.1.3 RxByte

The RxByte determines the payload type looking at the protocol field in the PPP header. IPv4 and IPv6, both with UDP, are supported in this release. Depending on payload type, it then launches either an IPv4 routing table lookup or an IPv6 routing table lookup. It then streams the entire UDP/IP header and payload to the DMA launcher and copies the also headers to the extract space where the CPRC can read them. RxByte accumulates the CRC-32 FCS on each header and payload byte and verifies CRC match at the end of the frame. For a packet that fails the CRC check, a flag is set in the extract space to inform the CPRC about the failure.

The extract space data structure is described in section 5.4.3.4.

5.4.1.4 TxByte

Upon receiving a scope, TxByte generates the PPP header and sends it to the large FIFO. It then reads each byte of the pdu and generates stuffing bytes whenever necessary. It also accumulates CRC-32 on all the payload bytes and attaches the FCS at the end of the frame.

5.4.1.5 TxBit

TxBit monitors the data stream for out of frame signal and transmits the data to the output FIFO, as a bit-stream. TxBit is not configurable through control space.

5.4.2 RC

The RC performs higher level processing of data packets to determine where the packet descriptor, that contains information about the packet and the buffer id to the actual packet in memory, needs to be sent to next. In the transmit direction the POS RC receives packet descriptors, fetches the packets and transmits them. The RC code consists of two threads running in separate CPRC contexts. PosIn thread is responsible for Rx processing and posOut is responsible for Tx processing.

5.4.2.1 Initialization

During initialization the following things happen:

- Buffer pools, queues, port numbers, internal MAC address, internal IP address, table lookup entries, interrupts, SDP PHY parameters and various internal data structures are all initialized.
- Receive and transmit threads are launched.

5.4.2.2 posIn

This thread handles incoming data packets. It performs the following sequence of operations:

- If a packet is received from the RxByte processor that contained a frame or header error (detected by RxByte and reported in extract space) statistics are updated and the frame is dropped.
- If a packet is received from the RxByte processor that contains a expired IPv4 TTL (detected by RxByte) statistics are updated and an ICMP TTL Time expired packet is sent to IP SA of the sender
- For received IPv4 or IPv4/UDP packets, this thread will wait for the TLU to respond with the IPDA/UDP lookup result.
    - If the lookups fails, the packet is dropped and an ICMP destination unreachable packet is sent in response.
    - If the lookup succeeds, the port number returned from the IPDA/UDP in the result is then used to launch a TLU lookup into the port table. The port table lookup returns the current profile for the port. This profile tells whether the port is enabled, whether QoS is enabled on the port and whether header compression is enabled on the port. The destination queue for the packet is determined using this port

profile and the packet descriptor is enqueued to it. If the port is not currently valid (offline) the packet is dropped.

– For received IPv6 packets, this thread will wait for the TLU to respond with the first IPv6 DA lookup result.
  – If the first lookups fails, the packet is dropped.
  – If the first lookup succeeds, the second IPv6 DA lookup is launch with the tag value from the first lookup and the next 64bits of the IPv6 DA. This thread then waits for the result.
  – If the second lookup fails, the packet is dropped.
  – If the second lookup succeeds, the port number returned from the second IPv6 DA lookup result is then used to launch a TLU lookup into the port table. The port table lookup returns the current profile for the port. This profile tells whether the port is enabled, whether QoS is enabled on the port and whether header compression is enabled on the port. The destination queue for the packet is determined using this port profile and the packet descriptor is enqueued to it. If the port is not currently valid (offline) the packet is dropped.
  – This thread also keeps track of various receive statistics on packet size, drops etc.

5.4.2.3 posOut

This thread handles outgoing data packets. It performs the following operations:

This thread waits for packet descriptors on an input queue. The incoming packets are already formatted (IPv4, IPv4/UDP, IPv6, IPv4/ICMP or Pause frame) and this thread will simply fill in merge space with the appropriate IP header values, type of packet and notify the TxByte processor to transmit the packet.
This thread also keeps track of various Transmit statistics on packet size etc.

5.4.3 Data Structures

5.4.3.1 Configuration parameters

The POS component needs the following configuration parameters to be set from the XP:

– POS port number.
– Pool # to be used for RX packets.

5.4.3.2 RxSync Control Space

The RxSync processor is configured through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	crcMode	pad	MTU	

crcMode – specifies whether the FCS is a 16 bit CRC or a 32 bit CRC.
MTU – The maximum PPP frame size supported.
pad – unused.

5.4.3.3 RxByte Control Space

The CPRC configures the RxByte processor through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	crcMode	pad		

crcMode – specifies whether the FCS is a 16 bit CRC or a 32 bit CRC.

5.4.3.4 Extract Space

RxByte writes information about received packets into extract space for the RC to use in its processing. The first 6 words of the data structure have the following format:

Byte Offset:	0	1	2	3
0	IPv4/IPv6 header			
20 (IPv4)/ 40 (IPV6)	UDP source port		UDP destination port	
24(IPv4)/44(IPv6)	UDP packet length		UDP Checksum	
28(IPv4)/ 48(IPv6)	ProtocolType		FrameStatus	DropCount
32(IPV4)/52(IPv6)	PayloadChecksum		Pad	

IPv4/IPv6 header – The entire IPv4 or IPv6 header (shown in following tables).
UDP source port – UDP source port number
UDP destination port – UDP destination port number
UDP Packet length – The length
protocolType– The protocol type field from the PPP header
FrameStatus – Flag indicating whether RxByte completed streaming the entire frame without errors.
Payload checksum – the CRC accumlated over the PPP frame and the FCS.
pad – unused

For IPv4 packets, the first 20 bytes of extract space have the following format

Byte Offset:	0	1	2	3
0	version	IHL	tos	pad
4	length		id	
8	flags	pad	fragOffset	
12	ttl	protocol	checksum	
16	srcaddr			
20	destaddr			

version – IP version
IHL – header length
tos – type of service field
pad – unused
length – IP total length
id – IP identification field
flags – IP flags field
fragOffset – IP fragmentation offset field
ttl – IP time to live field
protocol – IP protocol field
checksum – IP header checksum field
srcaddr – IP source address
destaddr – IP destination address

For IPv6 packets, the remainder of extract space has the following format:

Byte Offset:	0	1	2	3
0	vers_tclasshi	tclasslo_flowhi	flowlo	
4	len		nexthdr	ttl
8 - 23	srcaddr			
24-39	destaddr			

vers_tclasshi – IPv6 version and 4 MSBs of traffic class
tclasslo_flowhi – 4 LSBs of traffic class and 4 MSBs of flow ID
flowlo – remaining bits of flow ID
len – IPv6 total length
nexthdr – IPv6 next header field
ttl – IPv6 hop limit field
srcaddr – IPv6 source address
destaddr – IPv6 destination address

5.4.3.5 TxByte Control Space

The CPRC configures the TxByte processor through a structure in control space. The data structure has the following format:

Byte Offset:	0	1	2	3
0	OC-12	crcAlgorithm	Pad	

OC-12: Set to 0 in this app. It should be set to 1 if in OC-12 aggregtaed mode is being supported.
CrcAlgorithm: Whether CRC-16 or CRC-32 for FCS.
pad – unused

5.4.3.6 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	ProtocolType		packetAlg	dropHdrSize
4	PacketLength		pad	

ProtocolType – The protocol type field to put in the PPP header
Packet Alg – Whether to decrement the TTL field in the IP header or not.
DropHdrSize – Number of bytes to drop from the header before PPP encapsulation; Set to zero in this app.

5.5 UL-2 Fabric Port component

The Fabric Port (FP) 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 FP sends cells received from other ATM processing blocks to the UL-2 interface.

5.5.1 FpTx

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

– 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. The data structure is defined in section 5.5.4.1.

5.5.2 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. This helps in extraction of the STF (and potentially the CPS) header from a cell carrying AAL2 traffic. 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 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. FpRx extract space is defined in section 5.5.4.2.

5.5.3 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. The descriptor data structure is described in section 5.5.4.1.

5.5.4 Data Structures

5.5.4.1 Merge Space

The FpTx hardware copies the descriptor data from the payload bus to merge space after a dequeue. Merge space 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

5.5.4.2 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

5.6 ML/MC-PPP component

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 5.23.2.

5.6.1 SDP

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

5.6.1.1 RxByte

The RxByte processor receives ML-PPP fragments from the TxByte processor and handles them differently depending if 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.
- Receives control information (port, index, fragType, length) and puts it in extract.
- For SOM fragments, also receives the bufType and puts it in extract.
- 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 initiates 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 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 fragement.
– Writes the payload to DMEM.
– When data9 is received, switches scope and waits for the next operation code.

The RxByte processer communicates with the RC through extract space. Extract space is described in section 5.6.3.2. The RxByte processor is not configurable through its control space.

5.6.1.2 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.
– 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 reassmbled 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, bufType to RxByte.
– Sends any remainder bytes from merge space to RxByte.
– Streams the payload from DMEM to RxByte.
– 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 does the following:
– Waits for scope to be available.
– Sends the segmentation operation code to RxByte.
– Sends the port to RxByte.
– Sends the HDLC/PPP/ML-PPP header from merge space to RxByte.
– Reads payload from DMEM and sends the number of bytes specified in merge space to RxByte.
– 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 5.6.3.3. The TxByte processor is not configurable through its control space.

5.6.2 RC

The RC manages the enqueuing and dequeuing 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.

5.6.2.1 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.
– Sets up the DMA engines and initializes the SDP scopes.

– Enables the SDPs.

5.6.2.2 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.
– Dequeues 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 5.6.3.4.

For segmentation, the input thread dequeues IP or compressed IP datagrams or NCP packets and enqueues descriptors to the soft scheduler. Specifically, it does the following:

– Enqueues descriptor to MC queue based on bundle and class from descriptor. The MC queues are soft queues as described in section 6.3.
– Adds the MC queue to the active queue list for the bundle.
– Adds the bundle to the active queue list.

For reassemby, the input thread dequeues ML-PPP fragments, possibly encapsulated in HDLC or PPP and sends them to the SDP for recirculation. 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 the not the next expected, places the descriptor in a linked list.
– If the sequence number is expected,
  – For non-SOM fragments, launch lookup into ML-PPP remainder table to get remainder bytes.
  – Waits for scope to be available from the SDP.
  – Writes reassembly state information and a subset of the ML-PPP header to merge space.
  – Writes the remainder bytes and length to merge space, if necessary.
  – Waits for the previous payload transfer to complete.
  – Initiates payload transfer from SDRAM to DMEM.
  – Updates reassembly state in DMEM.
  – If linked list has an expected sequence number in it, repeat procedure for that fragment.

5.6.2.3 Scheduler Thread

The scheduler thread services the MC queues for each bundle. It dequeues 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, dequeues 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.

5.6.2.4 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 enqueues 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 enqueues it to the destination queue based on the port field in extract space. The descriptor format is specified in section 5.6.3.2.
- 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 enqueues 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 enqueues 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 RxCB is initiated.

5.6.3 Data Structures

5.6.3.1 Configuration parameters

The ML component needs the following configuration parameters to be set from the XP:

- Pool # of the BMU buffer pool to be used for the per class soft queues.
- Pool # of the BMU buffer pool to be used for reassembled packets.

5.6.3.2 Extract Space

When reassembling, RxByte 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 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

5.6.3.3 Merge Space

When reassembling, the RC writes information about ML-PPP fragments to merge space for TxByte 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
8	remainder[0..7]			
12	remainder[8..14]			RemainderLen

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 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

5.6.3.4 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.

5.6.3.5 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 indices for each bundle is stored in the MlPppBundleParams structure. This structure is initialized and modified by the host.

5.6.3.6 MlPppQuantums

This data structure is an 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 exectued in the scheduler thread. This structure is initialized and modified by the host.

5.6.3.7 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.

5.6.3.8 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.

5.6.3.9 MlPppQueueHandles

This data structure is a two dimensional array of soft queue handles as described in section 6.3. 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.

5.6.3.10 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	next_sequence			

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.

5.6.3.11 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.

5.6.3.12 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 deallocated as elements are added and removed from the linked list. All bundles and classes shared this “heap” of linked list memory.

5.7 PPP-MUX component

This component provides PPP multiplexing and demultiplexing as defined in RFC3153.

5.7.1 SDP

The SDP is configured for byte level recirculation. The SDP recirculates multiplexed PPP frames to remove HDLC/PPP encapsulation and demultiplex the packets present in the frame.

5.7.1.1 TxByte

The TxByte processor performs the following functions:

Receives multiplexed PPP frame from DMEM.
Strips of HDLC/PPP header if present.
For every packet in the multiplexed frame, does the following:

- Extracts the length from the PPP-Mux length field.
- Determines whether the protocol field is present.
- Sends the operation code (demultiplex) to RxByte.
- Sends the port to RxByte.
- Translates the protocol ID into a buffer type and sends that to RxByte.
- Streams length bytes of the frame.
- Sends an end of packet byte.

If more data is present, another packet is assumed and processed as described previously.
Switches scope and waits for more data to be available in DMEM.

The RC writes information about frames needing recirculation into merge space for TxByte to use in its processing. The data structure is defined in section 5.3.3.5. TxByte is not configurable through control space.

5.7.1.2 RxByte

The RxByte processor performs the following functions:

- Waits for a receive scope to become available.
- Receives control information from TxByte and places it in extract space.
- Streams the packet payload to DMEM.
- Switches scope and waits for another to become available.

The RxByte processor writes information about received packets into extract space for the RC to use in its processing. The data structure is defined in section 5.10.3.2. RxByte is not configurable through control space.

5.7.2 RC

The PPP-Mux component uses three threads to perform its task, namely, an input thread, an output thread, and a scheduling thread. The initialization code starts the threads. Each of these is described next.

5.7.2.1 Initialization

The PPP-Mux component does the following things during its initialization:

- Initializes the buffer pool used for outgoing packets and frames.
- Creates the input, output, and scheduling threads.
- Initializes both scopes by giving the SDP ownership.
- Starts the SDP in byte loopback mode.
- Jumps to the first thread.

5.7.2.2 Input Thread

The input thread handles dequeues descriptors and, depending on the buffer type multiplexes packets into PPP frames or demultiplexes PPP frames into incoming multiplex. Specifically, it does the following:

- Monitors its one queue waiting from a descriptor to be present then dequeues it.
- Extracts fields from the descriptor to decide how to process the packet.

If the buffer type received in the descriptor indicates HDLC, PPP, or PPP-Mux, the frame will be demultiplexed. In this case, the component does the following:

- 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 and the default protocol ID from the PPP-Mux port parameters structure.
- 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.

For all other buffer types (e.g. IPv4), the component multiplexes packets into a single multiplexed frame. In this case, the component does the following:

- Maps the buffer type to the PPP protocol for later use.
- If the length of the new packet would exceed the maximum frame length, sends out the multiplexed frame as it exists.
- If the length of the new packet is greater than the maximum subframe length, sends the packet without change out the TDM port. Switches context to the next thread and loops to the beginning to wait for another descriptor.
- Copies the remainder bytes from the PPP-Mux state structure to a local DMEM buffer.
- Constructs the PPP-Mux header using the PPP protocol from before and the length from the descriptor.
- Transfers the packets payload from SDRAM to the local DMEM buffer.
- Calculates the size of the hole created due to the unaligned transfer.
- Updates the PPP-Mux state structure and if this is the first packet in the multiplexed buffer:
  - Allocates a new buffer.
  - Adds the port to the linked list.
  - Saves the wall clock as the start time for this multiplexed buffer.
- Locks the state structure and waits for the DMA transfer to complete.

– Removes the hole in the local DMEM buffer, if one exists.
– Transfers the local DMEM buffer back to SDRAM.
– Copies the remaining unaligned bytes to the state structure.
– Waits for the DMA transfer to complete.
– Unlocks the state structure.
– Switches context to the next thread.
– Loops to the beginning to wait for another descriptor.

5.7.2.3 Output Thread

The output thread handles outgoing packets which have been demultiplexed. 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.
– Fills in a descriptor using information from extract space.
– Waits for the payload transfer to complete.
– Sends the descriptor to the appropriate destination determined by the buffer type.
– Switches context to the next thread.
– Loops to the beginning to wait for another scope to be available.

5.7.2.4 Scheduling Thread

The scheduling thread implements a timeout which ensures that multiplexed buffers which have not received packets for some time are not held indefinitely. It accomplishes this by examining the linked list of multiplexing operations in progress and comparing the buffer's start time against the current time. Specifically, it does the following:

– Gets the port of the multiplexed buffer at the head of the linked list.
– Waits for the lock on the port to be freed.
– Compares the current time versus the start time of the multiplexed buffer for the port.
– If the timeout is exceeded, send out the frame as is.
– Switches context to the next thread.
– Loops to the beginning to check the head of the linked list again.

5.7.3 Data Structures

5.7.3.1 Configuration parameters

The PPP-Mux component needs the following configuration parameters to be set from the XP:

– Pool # to be used for RX packets.

5.7.3.2 Merge Space

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

Byte Offset:	**0**	**1**	**2**	**3**
0	opCode	port	DefaultPid	

opCode – indicates that demultiplexing must be performed
port – the input port on which the multiplexed frame was received
defaultPid – the default protocol ID to use if not present in the multiplexed frame

5.7.3.3 Extract Space

RxByte writes information about received packets 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	bufType	pad

opCode – indicates that demultiplexing was performed
port – the input port on which the packet was received
bufType – the buffer type of the received packet
pad – unused

5.7.3.4 PPP-Mux Port Parameters

The PPP-Mux port parameters structure holds the PPP-Mux specific parameters for each PPP TDM port. The data structure has the following format:

Byte Offset:	0	1	2	3
0	defaultPid		MaxFrameLen	
4	maxSubFrameLen		Pad	

defaultPid – the default protocol ID
maxFrameLen – maximum size of a multiplexed PPP frame
maxSubFrameLen – maximum size of a packet to be multiplexed in a PPP frame
pad – unused

5.7.3.5 PPP-Mux State

The PPP-Mux state structure holds the PPP-Mux state that must be saved to properly multiplex packets into a single PPP frame. One structure is necessary for each PPP TDM port. The data structure has the following format:

Byte Offset:	0	1	2	3
0	btag		Offset	
4	lastPid		next	prev
8	StartTime			
12	remainder[0..3]			
16	remainder[4..7]			
20	remainder[8..11]			
24	remainder[12..14]			lock

btag – the buffer tag of the multiplexed buffer (the pool is constant, so it is not saved)
offset – the offset into the multiplexed buffer at which the next packet should be placed
lastPid – the protocol ID of the last packet that was put in the multiplexed buffer
next – a linked list pointer to the next port that has a multiplexed buffer in progress
prev – a linked list pointer to the previous port that has a multiplexed buffer in progress
startTime – the wall clock time at which multiplexing of this buffer began
remainder[15] – the remaining unaligned bytes from the previous DMA action
lock – a semaphore
pad – unused

5.8 IMA component

This component implements the IMA functionality over ATM TDM circuits. 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.

5.8.1 IMA Rx

The IMA Rx 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 dequeues 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.
- Runs the link state machine based on link state information from the ICP cell.
- Runs the group state machine based on group state information from the ICP cell.
- Runs 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, puts the cell in the link differential delay queue.
Uses round robin the 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 enqueued to the next processing block (AAL-2, ATM TM, etc.) determined by data in the descriptor.
- Switches to the next context.
- Loops and waits for next descriptor.

5.8.2 IMA Tx Input

The IMA Tx Input thread dequeues 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 dequeues 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.

5.8.3 IMA Tx Output

The IMA Tx 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 enqueued 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 enqueued to the queue of the target TDM link.
– Otherwise a filler cell is enqueued to the queue of the target TDM link.
– Updates link and group state for the link and group on which the cell was just sent.
– Switches to the next context.
– Loops and waits for the next tick.

5.8.4 Data Structures

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

5.8.4.1 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 errored ICP cells used in frame synchronization
numIcpInvalid – number of consecutive frames with invalid ICP cells used in frame synchronization

5.8.4.2 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			
16-44	linkIdToLinkMap[32]			
48	frameLen		state	feState
52	change	rxChangeNum	txChangeNum	numLinks
56	suffLinks	flags	rxOamLabel	rxImaId

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

5.8.4.3 ImaParams

The IMA parameters structure stores information that controls behavior of the IMA unit. These parameters may be set by the host or other management agent. 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

5.8.4.4 ImaPortToGroupMap

The IMA port to group 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 group to which the port belongs. If the port does not belong to any group, the value is IMA_INVALID_GROUP.

5.8.4.5 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.

5.9 IP Header compression

The IPHC component supports IP header compression as specified in RFC 2507 and RFC 2509. This component compresses IPv4, IPv6, and UDP datagrams and decompresses compressed IPv4, IPv6, and UDP datagrams. The component maintains the necessary state information as a compression context in buffer memory. The IPHC component does not use the SDP. The RC does all processing.

5.9.1 Initialization

The initialization component initializes the necessary services and data structures including compression contexts to a known state before entering its main loop. The host configures other parameters needed by the IPHC engine. For more information see section 11.5.

5.9.2 Main Loop

The main loop of the IPHC component does the following things:

- Waits for a descriptor to be present in its queue then dequeues it.
- Transfers payload from SDRAM buffer specified in descriptor to local DMEM.
- Waits for the payload transfer to complete.
- Based on buffer type indicated in descriptor, compresses or decompresses packet.
  - If the buffer type is BT_IPv4 or BT_IPv6, the buffer is compressed using the compression algorithm.
  - Otherwise decompression is performed, after removing any L2 encapsulation.
- Transfers the compressed or decompressed buffer in DMEM to SDRAM.
- Fills in the output descriptor with buffer information, the new buffer type and the application data from the input descriptor.
- Waits for the buffer transfer to complete.
- Enqueues the descriptor to the egress queue if compression was performed, or the IP component if decompression was performed.
- Returns to the top of the loop

5.9.3 Compressor

The compression algorithm takes as input a compression ID, an input buffer, an output buffer, and a length. The compression ID is an index into the compression context table. A compression context is maintained in SDRAM for each flow being compressed. The format of a compression context is described in section 5.9.5.2. The input buffer holds an uncompressed packet and as the algorithm performs the compression, it transfers the data to the output buffer. The length indicates the number of bytes in the input buffer. The algorithm returns the type of the buffer that was created (e.g. full header or compressed non-TCP header). The compression algorithm works as described in RFC 2507.

5.9.4 Decompressor

The decompression algorithm complements the compression algorithm. It takes as input a buffer type, an input buffer, an output buffer, and a length. The buffer type indicates whether the buffer is a full header or compressed non-TCP header. Other buffer types are passed through, untouched. The input buffer contains the compressed packets requiring decompression. The output buffer is where the decompressed packets will be placed. Finally, the length indicates how many bytes are in the input buffer. The algorithm returns the type of buffer that was decompressed. The decompressor maintains compression context state in SDRAM. The format of the compression context is described in section 5.9.5.2.

5.9.5 Data Structures

5.9.5.1 Configuration parameters

The IMA component needs the following configuration parameters to be set from the XP:

– The BMU pool # of the buffer pool to be used for cells.
– The BMU pool # of the buffer pool to be used for soft queues.

5.9.5.2 Compression Context

The compression context stores a portion of the packet header and parameters needed by the compressor and decompressor. If has the following format:

Byte Offset:	0	1	2	3
0-232	Header			
236	cid		hlen	
240	gen	pad		
244	cnum		fperiod	
248	Ftime			
252	Gtime			

header – the header context
cid – the compression ID
hlen – actual length of headers
gen – the generation of the full header
pad – unused
cnum – count of compressed packets sent since last full header
fperiod – number of compressed packets that may be sent before a full header
ftime – time the last full header was sent

gtime – time the generation was set to zero

5.10 IP component

This component provides IP header parsing and forwarding for components that need assistance - the TDM interface and the AAL-2/5 reassembly components. The IP processor uses the IP routing tables defined in sections 5.23.3 and 5.23.4 and the port table defined in section 5.23.1.

5.10.1 SDP

The SDP is configured for byte level recirculation. The SDP recirculates the IP packet to remove HDLC/PPP encapsulation if present, validates the IP header, and possibly verifies the UDP checksum. It also launches the IP destination address lookup to retrieve forwarding parameters for the datagram.

5.10.1.1 TxByte

The TxByte processor performs the following functions:

– Receives IP datagram from DMEM.
– Strips of HDLC/PPP header if present.
– Sends control information about the datagram to RxByte, including the buffer handle, buffer type, and input port.
– Sends the IP datagram to RxByte.

– If IP protocol is UDP, calculates the UDP checksum.
– 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 recirculation into merge space for TxByte to use in its processing. The data structure is defined in section 5.3.3.5. TxByte is not configurable through control space.

5.10.1.2 RxByte

The RxByte processor performs the following functions:

– Waits for a receive scope to become available.
– Receives control information from TxByte and places it in extract space.
– Validates the IP header including version and header length and IP checksum.
– If the header is valid, launches a lookup of the IP destination address. If the protocol is UDP, the UDP destination port is concatenated with the IP 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.
– 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 5.10.3.2. RxByte is not configurable through control space.

5.10.2 RC

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

5.10.2.1 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 loopback mode.
– Jumps to the first thread.

5.10.2.2 Input Thread

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

– Monitors its one queue waiting from a descriptor to be present.
– Dequeues an IP descriptor.
– 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.

5.10.2.3 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 a counter.
- Waits for the IP route lookup to complete and if the lookup fails, drops the packet and increments a counter.
- Launches a lookup of the port indicated in the IP route lookup response.
- Waits for the port lookup to complete, and if it port is invalid, drops the packets and increments a counter.
- Fills in a descriptor using information from the IP route and port lookups.
- Waits for the payload transfer to complete.
- Sends the descriptor to the appropriate destination determined by the lookups.
- Switches context to the next thread.
- Loops to the beginning to wait for another scope to be available.

5.10.3 Data Structures

5.10.3.1 Merge Space

The RC writes information about datagrams needing recirculation into merge space for TxByte 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 BT_IPv4, BT_HDLC, or BT_PPP)

pad – unused

5.10.3.2 Extract Space

RxByte writes information about recirculated datagrams into extract space for the RC to use in its processing. 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, BT_HDLC, or BT_PPP)

headerError – zero if no errors in header, error code otherwise
payloadError – zero if not errors in payload, error code otherwise

The format of the next part of extract space depends on whether the IP packet was IPv4 or IPv6. In the case of IPv4, the format looks like:

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

In the case of IPv6, the format looks like:

Byte Offset:	0	1	2	3
8	vers_tclasshi	tclasslo_flowhi	flowlo	
12	Len		nexthdr	ttl
16-28	srcaddr			
32-44	destaddr			

vers_tclasshi – header version and upper nibble of traffic class
tclasslo_flowhi – lower nibble of traffic class and upper nibble of flow ID
flowlo – remaining bits of flow ID
len – IP total length
nexthdr – next header following this one
ttl – time to live
srcaddr – IP source address
destaddr – IP destination address
Following the IP header in extract space is the UDP header with the following format:

Byte Offset:	0	1	2	3
48	Srcport		destport	
52	Len		cks	

srcport – UDP source port
destport – UDP destination port
len – UDP length
cks – UDP checksum

5.11 ATM OAM

The ATM OAM component implements the following features:

1. AIS and RDI cell generation.
2. Continuity check (CC) monitoring.
3. Host command driven Loopback (LB) cell insertion.
4. Forward performance estimation (FPM)

All these functions will be implemented at both VPC and VCC levels in both segment and end-to-end modes of operation. The following section describes the design and implementation of each function. Only the FPM function is implemented on the CPRC. The other functions are implemented on the XPRC and host.

5.11.1 AIS and RDI alarm conditions

The AIS and RDI conditions are relevant to cells that are switched from one ATM interface to another. The AIS condition indicates a transmission path defect downstream from the ATM NIC and RDI indicates a defect upstream. AIS and RDI alarm conditions occur when:

- There is a Sonet level alarm on one of the interfaces: AIS condition is declared on all the switched VPCs and VCCs on the other interface. AIS cells are generated on all such connections. AIS cells need not be generated on AAL2 and AAL5 VCs.
- Loss of cell delineation (LCD) is detected on one of the interfaces: AIS condition is declared on all the switched VPCs and VCCs on the other interface. AIS cells are generated on all such connections. The RDI condition is declared on the interface experiencing LCD and RDI cells are generated on all the switched VPCs and VCCs on this interface.
- Receiving an AIS cell on a VPC/VCC: When an AIS cell is received on a connection for which we are a segment end point or a connection end point, the AIS condition is declared on the connection.
- Receiving an RDI cell on a VPC/VCC: When an RDI cell is received on a connection for which we are a segment end point or a connection end point, the AIS condition is declared on the connection.

Once an AIS/RDI connection is declared, it persists for 2.5 seconds. If the root cause of the condition does not occur during the 2.5second window, the condition is cleared. AIS and RDI cells are always generated one cell per second.

5.11.2 Implementing AIS and RDI alarm conditions

The AIS and RDI conditions are detected by the XP. The XP polls both the Sonet interfaces once every second to ascertain the cell delineation status. Any Sonet alarms on the interfaces are reported to the XP. All OAM cells received on both the interfaces are forwarded to the XP.

The XP maintains a list of all the switched VPCs and VCCs states in a TLU data table. When an AIS or RDI condition is detected it walks through this table and generates AIS/RDI cells for each one the entries in the list. Each entry in the list also holds the current alarm state of the corresponding connection. This alarm state is set and cleared by the XP in response to the occurrence and clearance of alarm conditions respectively. The host, to determine the alarm status on any VPC/VCC, can also access this list. As long as the offending alarm condition persists, the XP performs the table walk once every second. If a period of 2.5 seconds goes by without the alarm condition being reported, the alarm condition is cleared for all the entries in the list. The following figure illustrates the list data structure.

```
Struct ConnState  {
      Int8u   level ;  // whether VPC or VCC
      Int8u   type;   // whether segment or end-to-end or neither
      Int8u   currentAlarmState;  // Whether AIS or RDI
      Int8u  vpi;
```

```
        Int16u  vci;
        Int16u  vcIndex;  //The unique id from the ATMVC table.
        Int16u  ccTimer;
};
```

The timers needed for LCD polling and table walks are implemented using the cycle count register. A target cycle count equal to (timerPeriod/clockFrequency) + currentCycleCount is associated with each of these periodic events. When the current cycle count equals the targeted cycle count, the timer has fired and the associated event action is performed.

5.11.3 Continuity check

Certain connections (both VPCs and VCCs) are marked for active continuity check. On these connections, a special OAM CC cell is sent once every second. If a CC cell is not received on these connections for 2.5 seconds, the AIS alarm condition is declared for the connection.

The list of connections used for AIS/RDI alarms is also used for continuity checks. Continuity check enable is marked by a bit in the type field of the list structure. During the one second table walks, the CC cell is sent on all the specially marked connections. The CC timer field is then set to the current time plus 2.5 seconds. When a CC cell arrives on this VC, the CCTimer field is set back to zero and the timer is cancelled. If a CC cell does not arrive before the timer expiry the currentAlarmState is set to AIS.

5.11.4 FPM CP implementation

The FPM CP recieves cells on all flows marked for FPM processing. It recirculates the ATM cell accumulating BIP-16 over the payload. The accumulated BIP-16 for each stream is stored in a TLU data table. Upon receiving a FPM cell from peer, this component verifies the BIP-16 in the FPM cell with accumulated BIP-16 and tracks the number of bit errors if any. In the transmit direction, this component generates an FPM cell at periodic cell intervals.

5.11.4.1 SDP

The SDP is used in recirculation mode to calculate BIP-16. The TxByte processor performs the parity calculations and sends the parity half word to RxByte. RxByte launches a TLU table lookup to fetch the previously accumulated BIP on this stream and hands over the BIP for current cell to the CPRC. The CPRC updates the BIP and writes it back to the TLU table

The TxByte processor performs the following sequence of steps:

Determine the payload type. If it is a FPM cell simply forward the payload type, cell context and BIP to RxByte.

If it is a user cell then perform the following operation: Read the next 16 bytes from cell payload. Xor them with previous 16 bytes. Repeat this operation till the entire cell payload has been processed. The cumulative XOR is the BIP-16 for the current cell. Send it to the RxByte along with the payload type and cell context,

The RxByte processor performs the following sequence of steps:

- The cell context consists of the index into BIP-16 accumulation table for the current cell's traffic stream.
- Launch a TLU table lookup to read this value.
- Copy the cell context and BIP-16 to extract space and switch scopes.
- If the cell is an FPM/BR cell, RxByte also parses the mcsn, bedc, tuc0 and tuc0+1 fields and writes them to the extract space.

5.11.4.2 RC

The OAM component on the RC consists of two CPRC contexts. The AtmOamIn context dequeues input cells and feeds them to the SDP. Rx cells will have the bufferType field in their message descriptor set to BUFTYPE_ATM_OAM_RX and Tx cells will have it set to BYFTYPE_ATM_OAM_TX. FPM and BR cells will have their bufferTypes set to BUFTYPE_ATM_FPM and BUFTYPE_ATM_BR respectively. This will allow the OAM component to distinguish between Rx ata, Tx data and OAM cells. As stated previously, the BIP-16 state is maintained in a TLU data table. A VC/VP will have two entries in this table: one for Rx state and another one for Tx state. This table will also track the PM block size and the number of cells received/sent since the last FPM cell was received/sent. For both Rx and Tx cells, the SDP accumulates the BIP16 on cell payload and makes it available to AtmOamOut context. AtmOamOut accumulates it in the TLU table. In the Tx path, AtmOamOut generates the FPM cells once for every block. In the Rx path, AtmOamOut compares the BIP-16 reported on an FPM cell from the remote peer with accumulated BIP-16. Errors detected in this comparison by will be sent to the host for statistics monitoring. The BIP-16 block size will be set and maintained in the state table by the host.

The entire message descriptor is written to the merge space. The SDP recirculates the entire descriptor and makes it available in the RxByte extract space. This establishes a work context for AtmOamOut context. The TxByte then reads the entire cell payload and accumulates the BIP-16. The BIP-16 is sent to the RxByte processor – cell payload is not streamed through.

AtmOamOut reads the extract space to determine whether the cell is an Rx cell or Tx cell. It then creates the index into BIP-16 TLU state table as vcIndex for Rx and (vcIndex + (1 << 15)) for Tx. If the Rx cell was an FPM cell, the tuc01, bedc and tuc0 fields contain the TUC0+1, BEDC0+1 and TUC0 count fields from the FPM cell. The state table is the read at this index to fetch the current state. The following structure will be used for the state table entry:

The bip16 from the extract space is XOR'ed with the corresponding value from the table. The numCellsProcessed count is incremented and the entry written back to the TLU table. The destination queue number is then read from the length field of the message descriptor and the length field is set back to 48 (cell payload length). The message descriptor itself is then forwarded out on the read destination queue number.

FPM and BR cell descriptors received by AtmOamIn are recirculated but the BIP-16 accumulation is not done on them. TxByte extracts the FPM counts and recirculates them. AtmOamOut reads the state table and compares the BIP count reports with the accumulated BIP. Any error detected is forwarded to the host for error count soaking. The state entry is then reset and written back to the table.

Similar to Rx, Tx cells sent to OAM component have their final destination queue numbers written in the length field of their cell descriptors. Ex: An ATM cell going out TDM5 port will have the length field in its descriptor set to TDM_QUEUE5 when it is sent to OAM component. AtmOamIn writes the entire cell descriptor to the merge space and the SDP accumulates BIP-16 in the exact same manner as done for Rx. After updating the BIP16 in state table, atmOamOut compares the numCellsProcessed count to the block size. If the count equals block size an FPM cell is generated and sent to the destination cell Tx component before sending the current cell. The BIP state is then reset and written back to the table.

5.11.5 Data Structures

5.11.5.1 Configuration parameters

Pool # of the BMU buffer pool to be used when creating FPM cells.

This component is dependent on the existence of ATM physical ports in the application.

5.11.5.2 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	cellType	reserved	Pm Index	
4	Cell Descriptor			
8				
12				
16				

cellType – indicates whether the cell is a user data cell or an OAM (i.e, FPM or BR) cell.
PmIndex – index into the TLU table containing accumulated BIP-16 for this cell stream.
CellDescriptor – the complete cell descriptor for current cell.

5.11.5.3 Extract Space

RxByte writes information about received packets into extract space for the RC to use in its processing. The data structure have the following format:

Byte Offset:	0	1	2	3
0	BIP-16		cellType	MCSN
4	tuc01		bedc	
8	tuc0		Pad	
12	Cell Descriptor			
16				
20				
24				

BIP16 – BIP16 for current cell : valid only for data cells.
CellType – whether user data or OAM.
MCSN – FPM/BR cell sequence number : for OAM cells only
Tuc01 – Total number of CLP0+1 cells seen by peer: from corresponding field FPM/BR packet.
BEDC – BIP16 reported by peer: BEDC field from the FPM/BR packet.
TUC0 – Total number of CLP0 cells : from corresponding field in FPM/BR packet.
Cell Descriptor – the entire cell descriptor passed from TxByte.

5.12 Segmentation

This component implements AAL5 and AAL2 SSSAR segmentation. RFC 2263 LLC/SNAP encapsulation of IP packets in AAL5 SDUs is not supported. It gets message descriptors with bufType set to IPv4. If the segmentation portion of the packet descriptor has a CID of zero, the packet is processed for AAL5 segmentation into ATM cells. If the CID is non zero, it is processed for SSSAR segmentation into CPS packets.

5.12.1 SDP

The SDP is configured for byte level recirculation. It streams the IP packet and accumulates CRC. In case of AAL5, it adds the necessary pad bytes to make the SDU length a multiple of 48 and then creates the trailer. It then chunks up the SDU into fixed sized pieces and gives these pieces to the CPRC, one piece per scope. On the first chunk for a

SDU, the SDP launches a port table lookup, whose results are used by the CPRC to determine the destination queue (in case of AAL5) or the destination VCC (in case of AAL2).

The SDP does not interleave segmentations – it completely segments one SDU and delivers all the segments to the CPRC before proceeding to the next SDU. The same microcode is used to perform both AAL5 and AAL2 segmentation. The SDP accumulates payload CRC for all IP packets. However for AAL2 segmentation, it throws away the resulting CRC. In the future, this CRC could be used to support SSTED segmentation.

5.12.1.1 TxByte

The TxByte processor performs the following operations on every packet:

- Read the segType, pduSize and UUI from the merge space. Send them to RxByte.
- Read the CID and all the merge space fields following it. Send them to RxByte.
- Initialize a counter with the payload size.
- Start sending the payload bytes, accumulating CRC for each transmitted byte. Stop when the SDU payload is exhausted.
- Determine the number of necessary pad bytes and transmit that many zeroes, accumulating the CRC on each zero byte.
- If the segTye is AAL5 (or SSTED), the trailer should be sent. 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/SSTED 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 5.12.3.2. It is not configurable through control space.

5.12.1.2 RxByte

The RxByte processor performs the following sequence of operations:

- Receive the segType, pduSize, uui and CID from the txByte and copy them to the extract space.
- 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 ‘C1’ with the pduSize.
- Stream out payload bytes to the Rx stage area. Decrement C1 for every byte transmitted. When C1 hits 0, 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. Otherwise, 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, the CPRC needs to be told that the SDU has ended with this cell. Set a lastCell flag in the extract space to indicate this.
- 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 5.12.3.3. RxByte is not configurable through control space.

5.12.2 RC

The AAL5 segmentation CPRC code consists of two logical functions, namely aal5SegIn() and aal5SegOut(). These functions run in separate CPRC contexts. Aal5SegIn() dequeues message descriptors from the input queue and reads the length of the IP packet to be segmented. It then calculates the necessary pad bytes that should be added at the end of the packet to make its size a multiple of 48 bytes. The pad size and the packet size are given to the SDP along with the buffer handle of the IP packet. The SDP adds the pad bytes, the AAL5 trailer thus forming the

AAL5 SDU from the IP packet. It also accumulates the CRC over the AAL5 SDU and puts it in the trailer. Finally, it chops up the SDU into 48 byte chunks and delivers each chunk back to the CPRC as an ATM cell.

If an AAL5 VCC is configured for GFR traffic management category, Aal5SegIn() is also responsible for ensuring conformance to the PCR portion of the traffic contract. It calculates the number of cells that would be generated for the SDU after the padding. It runs GCRA on the hypothetical cell stream that would be generated by this SDU. If any cell in the SDU is indicated as non-conforming by GCRA, the entire PDU is dropped. This 'early packet discard' behavior saves on bandwidth and system resources by preventing transmission of SDus that are bound to be incomplete/corrupted because of dropped cells.

For the first cell in an SDU, the SDP initiates a port table lookup on the egress port. Aal5SegOut() receives these ATM cells from the SDP. If the SDP has indicated the start of a new SDU, it waits for the port table lookup results. Based on the port table lookup results, it decides the destination queue for all the cells in this SDU. Finally it builds a message descriptor for each cell from the SDP and forwards it to the destination queue.

5.12.2.1 Aal5SegIn()

SegIn can be logically considered as being split into aal5SegIn() and aal2CpsSegIn(). A zero CID in the input message descriptor causes the descriptor to be processed by the aal5SegIn(). It performs the following operations:

- Determine the pad size.
- Calculate the number of cells that would be generated for this SDU. Using the current time as the time of arrival for all these cells, run the GCRA algorithm. If all cells pass GCRA proceed to the next step. Otherwise drop the IP packet
- Allocate a Tx scope.
- Write 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.

5.12.2.2 Aal5SegOut()

This logical function gets ATM cells from the SDP and enqueues them to the appropriate destination queue. 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, the SDP would have launched a port table lookup. Wait for the port table lookup results. Read the egress queue from the extract space
  - 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 enqueue it to the destination queue.

The AAL2 CPS segmentation CPRC code consists of two logical functions, namely, aal2SegIn() and aal2SegOut(). These functions run in separate CPRC contexts. Aal2SegIn() dequeues message descriptors from the input queue and reads the length of the IP packet to be segmented. From the segmentation descriptor portion of the message descriptor, it determines the CPS packet size and the destination CID. It then feeds all this information along with the IP packet's buffer handle to the SDP. The SDP segments the IP packet into CPS packet size units and delivers them to the aal2SegOut() function. On the first CPS segment for an IP packet, the SDP launches a port table. The aal2SegOut() function gets the egress VPI , VCI and vcIndex from the port table lookup. It creates a message descriptor with bufType set to BT_CPS for every CPS packet and fills in the port table lookup information in the descriptor. Finally it enqueues the descriptor to one of the two AAL2 Tx CPs.

5.12.2.3 Aal2SegIn()

For every IP packet received, aal2SegIn() performs the following sequence of operations:

– Read the IP packet length, the CPS payload length and destination CID.
– Allocate a Tx Scope and write all this information along with the IP packet's buffer handle in the merge space.

5.12.2.4 Aal2SegOut()

Aal2SegOut() performs the following sequence of operations:

– Wait for an Rx scope and read the extract space when it is ready.
– Check if the 'new_sdu' flag is set. If set, the SDP would have launched a port table lookup. Wait for the port table lookup results. From the port table lookup results read the destination VPI, VCI and ATM VC table vcIndex. This information will have to be forwarded with every CPS packet descriptor for the current IP packet being segmented. If port table lookup returns failure, all the cells to the segments in the current SDU are dropped.
– Create a message descriptor with bufType set to BT_CPS. Fill in VPI, VCI and vcIndex from the port table lookup results. If the vcIndex is less than 256, the descriptor is enqueued to the second AAL-2 Tx CP. Otherwise it is enqueued to the first AAL-2 Tx CP.
– Wait for DMA completion from stage area to the SDRAM. At this point examine the lastCell flag in the extract space. If the flag is set, the current SDU ended with this cell. Set the new_sdu flag so that the the CPRC waits for a TLU lookup on the next cell.

Given this design of the segmentation function, only one AAL5 SDU or an AAL2 SSSAR SDU can be segmented at a time.

5.12.3 Data Structures

5.12.3.1 Configuration parameters

– Pool # of the buffer pool to be used for segmented cells or CPS packets.
– AAL5 VC Index range

5.12.3.2 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	cid
8	CellHdr			
12	EgressPort		AtmEgressQueue	
16	DestQueue		Pad	

segType – indicates whether the packet is to be segmented for AAL5 (SEG_AAL5) or AAL2 CPS (SEG_AAL2_CPS)
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 (always zero for AAL2 CPS)
cid – specifies the AAL2 CID (always zero for AAL5)
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
pad – unused

5.12.3.3 Extract Space

RxByte writes information about received packets into extract space for the RC to use in its processing. The data structure have the following format:

Byte Offset:	0	1	2	3
0	SegType	PduSize	uui	cid
4	AtmEgressQueue		DestQueue	
8	LastCell	NumBytesLast	Pad	

hdrStatus – status byte indicating success or failure status of header parsing
segType – indicates whether the delivered chunk is an ATM cell payload or a CPS packet
pduSize – specifies the size of the CPS packet (always 48 for ATM cells)
cid – specifies the destination AAL2 CID for CPS packets and is zero for ATM cells
uui – specifies the user-to-user information that arrived with the packet being segmented
atmEgressQueue – final queue number to be used to reach the egress port
destQueue – specifies the queue number on which the cell/CPS packet should be transmitted
lastCell – specifies the end of an SDU
numBytesLast – the size of the last CPS packet

5.13 Reassembly component

This component implements AAL5 and AAL2 SSSAR re-assembly functionality. An incoming message descriptor that has bufType field set to BT_ATM is an ATM cell descriptor and needs to be processed for AAL5 reassembly. A bufType of BT_CPS means that the descriptor points to an AAL2 CPS packet that needs to be processed for AAL2 SSSAR reassembly.

5.13.1 SDP

The SDP for reassembly is configured for byte level loopback. It performs payload CRC verification and initiates the DMA transaction to append the current cell/CPS packet at the end of the SDU. On the non-last cell/CPS 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 in case of AAL5 and by the callId is in case of SSSAR/SSTED. Both AAL5 and SSSAR/SSTED reassembly share the same CRC table and hence it should be ensured that vcIndices and callIds do not overlap. On the last cell/CPS packet 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.

5.13.1.1 TxByte

The TxByte processor performs the following functions:

- Wait for Tx scope from the CPRC.
- Initialize the CRC accumulator.
- Forward the rasType, eom_offset, vcCidIndex, numalignedBytes and numUnalignedByteCount to RxByte. The rasType indicates whether its is AAL2 or AAL5. The VcCidIndex holds vcIndex value for AAL5 and callId for AAL2 SSSAR. The eom field holds the eom flag indicating whether this is the last cell/CPS packet. NumUnalignedBytes will always be zero for AAL5.

- Initialize a counter ‘C1’to the value of numAlignedBytes. For AAL5, this parameter is always 48. Initailize another counter ‘C2’ numBytesPartialPayload.
- Start sending partial payload bytes from merge space, decrementing the C2 for every byte. When the counter hits 0, all unaligned bytes would have been sent. Accumulate the partial CRC every byte transmitted.
- Now start sending the payload bytes, decrementing C1 for every byte and accumulate the CRC. Stop accumulating CRC when the counter hits 0.
- 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 processing the cell/CPS packet. 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 5.12.3.2. The TxByte processor is not configurable through control space.

5.13.1.2 RxByte

The RxByte processor performs the following functions:

- Wait for extract scope from CPRC.
- 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 also goes 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 extrace space for the RC to use in its processing. The data structure is described in 5.12.3.3. The RxByte processor is not configurable through control space.

5.13.2 RC

5.13.2.1 AAL5 Reassembly

The AAL5 CPRC reassembly module consists of two functions, namely, aal5RasIn() and aal5RasOut() running in two separate CPRC contexts. The aal5RasIn() context dequeues input ATM cell descriptors, locates their reassembly state information and sends the cell description and the state information to the SDP for recirculation. The SDP accumulates CRC on the cell payload and appends the payload to the end of the corresponding AAL5 SDU. aal5RasOut() then receives the control and updates the reassembly state for the AAL5 SDU. If the SDU has completed, it is forwarded to the IP CPRC for further routing.

The RasList structure is used to track re-assembly state per VC and is described in section 5.13.3.4. An array of 128 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, 127]. Each element of this array is initialized with a valid buffer handle and offset set to zero.

When a cell arrives for AAL5 reassembly, aal5RasIn() 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 AAL5 SDU reassembled so far. This new cell needs to be written at the 'offset' location within the SDRAM buffer.

aal5RasIn() then allocates the free Tx scope and writes the buffer handle and the offset from the rasList entry, buffer handle of the ATM cell and vcIndex in the merge space. The EOM bit in the ATM cell header for the last cell comprising the SDU marks the end of the AAL5 header. aal5RasIn() checks the cell header to determine if the current cell is the last one for the SDU. It then writes a flag indicating the EOM status in the merge space.

The SDP accumulates CRC on the ATM cell payload and initiates the DMA transaction to append the cell at the specified offset. It then writes the EOM status, vcIndex and new offset to the extract space. aal5RasOut() 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, aal5RasOut() performs the following operations:

– Wait for CRC result 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 is always assumed that an AAL5 SDU has an IP packet in the payload. aal5RasOut() 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 maximum permitted SDU size is restricted to 2048 bytes in current release. 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 aal5RasOut() 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.

5.13.2.2 AAL2 CPS reassembly

The AAL2 CPS reassembly module consists of two functions, namely, aal2CpsRasIn() and aal2CpsRasOut(). These functions run in separate contexts – aal2CpsRasIn() lives in the same context as aal5RasIn() and aal2CpsRasOut() in the same context as aal5RasOut(). Both AAL5 and AAL2 CPS use the same SDP microcode.

The aal2CpsRasIn() function gets a dequeued CPS message descriptor. A call Id uniquely identifies each SSSAR reassembly flow. The call Id is a concatenation of the vcIndex of the cell on which this packet arrived and its CID. The AAL2 Rx CPRC determines the call Id and sends it in every CPS packet descriptor. aal2CpsRasIn() uses the call id to locate the reassembly state.

A SSSAR flow is at the AAL2 CID level and not at the VCC level like AAL5. Hence there are a large number of SSSAR flows even if the number of AAL2 VCCs is small, making it impossible to hold the reassembly state information for all of them in DMEM. Instead a TLU SRAM data table is used to hold the reassembly state information. This table is indexed by the call Id. An array of 128 state table entries is also maintained locally in DMEM as a cache into this table. The TLU state table is read only when an entry for the call Id cannot be found in the cache. State table updates are always made both to the cache as well as to the TLU based table. The state information maintained for AAL2 CPS reassembly is described in section 5.23.8.

The bufHandle refers to the BMU SDRAM buffer where the current SSSAR SDU is being reassembled. As with AAL5 reassembly, the offset refers to the length of the SDU accumulated so far. AAL2 CPS packets can come in any sizes between 1 and 64. Hence, offset could take any value from 1 to 64. This leads to a problem when appending the current CPS payload. The BMU SDRAM is addressable at 16 byte offsets only. The BMU hardware disallows writes at offsets that are not a multiple of 16 bytes. This problem does not exist for AAL5 because ATM cells are 48 bytes (a multiple of 16 bytes) in size.

To satisfy this BMU restriction, not all bytes of the CPS packet are appended to the BMU buffer. Instead only N number of bytes are appended, where N is the largest multiple of 16 less than the CPS packet size. The remaining bytes in the CPS payload are called the 'unaligned bytes' and are stored in the state table entry. The partial payload array in the RasDataTableEntry structure holds these un-aligned bytes. If the payload size is less than 16 bytes, the entire payload is considered unaligned. The numUnalignedBytes field holds the count of the actual number of unaligned bytes in the entry.

When aal2CpsRasIn() gets a CPS packet for reassembly, it computes a hash on the callId and determines the cache location. The entries of the cache are defined in section 5.13.3.5. It then examines the cache location to see if the callId in the cache and the current callId match. If they match, the reassembly state from the cache is used. If they do not match, a TLU read command is issued and the aal2CpsRasIn() waits to get the response. The TLU read results are written into the cache, thus evicting the entry that existed there.

The unaligned bytes from the previous CPS packet should be appended to the current SSSAR SDU before appending the payload from the current packet. Hence the unaligned bytes are written to the merge space and the SDP processes them before the payload. The unaligned bytes from the current CPS packet are determined using the equation:

```
num_unaligned_bytes = (prev_num_unaligned_bytes + CPS_packet_size)%16
```

This number is fed to the SDP through the merge space. Also the number of bytes that should be appended to the SDU (these bytes are referred to as 'aligned bytes') is determined using the following equation:

```
num_aligned_bytes = (prev_num_unaligned_bytes + CPS_packet_size) –
num_unaligned_bytes
```

Again this number is fed to the SDP through the merge space. The SDP appends num_aligned_bytes from the merge space partial payload and the CPS packet payload to the SDRAM buffer. The remaining CPS payload bytes are the new unaligned bytes. The SDP writes them to the extract space after recirculation.

The last CPS packet in an SSSAR SDU is has a UUI of 27. When UUI of 27 is seen, all bytes in the current payload are considered as aligned. The SDP appends the entire payload to the end of the SDRAM buffer.

After recirculating a CPS packet down through the SDP, aal2CpsRasIn() marks the corresponding cache entry as locked. If another CPS packet requiring the same entry arrives, its processing is stopped till aal2CpsRasOut() finishes copying the unaligned bytes into the cache entry and the TLU state table. The lock ensures that unaligned bytes are consistently tracked in the system.

When the aal2CpsRasOut() context gets the scope from the SDP, it copies the unaligned bytes from the extract space into the cache entry, updates the offset in the cache state entry and initiates a TLU table update. It then unlocks the cache entry.

If the current CPS packet is the last one in an SSSAR SDU, the following additional steps are taken:

1. As the same SDP microcode is being used for both AAL5 and SSSAR reassembly, CRC accumulations and updates are done for AAL2 as well. However, SSSAR does not have a CRC trailer and hence the CRC lookup is certain to return failure. aal2CpsRasOut() waits for this result, ignores the error and frees up the ring bus slot associated with the TLU CRC response.
2. It is assumed that the SSSAR SDU contains an IP packet as its payload. The payload is extracted and dispatched to the IPCPRC.

3. As with AAL5, if the offset exceeds the maximum possible SSSAR SDU length of 2048, the SDU needs to be discarded. When this is the case, the offset is reset to zero making the same buffer available for the next SDU.
4. For all other SSSAR SDUs, the buffer handle in the state entry is marked as invalid and the length is set to zero. If aal2CpsRasIn() finds an invalid buffer handle in a valid cache entry or after a TLU table read, it means that a new SSSAR SDU is starting. It hence allocates a new SDRAM buffer and writes it into the cache entry.

Common SDP microcode and aal2CpsRasOut() waiting for CRC table lookups will make it possible to support SSTED in the future. The SSTED SDU ends with a trailer very similar to the AAL5 trailer.

5.13.3 Data Structures

5.13.3.1 Configuration parameters

– Pool # of the buffer pool to be used for reassembled IP packets.
– AAL5 VC Index range.

5.13.3.2 Merge Space

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

Byte Offset:	0	1	2	3
0	RasType	Eom	vcCidIndex	
4	NumAligned	NumUnAligned	numBytesPartial	pduLen
8	PartialPayload[0..3]			
12	PartialPayload[4..7]			
16	partialPayload[8..11]			
20	partialPayload[12..15]			

rasType – 0x00 for AAL5 and 0xf0 for AAL-2
eom – 0x00 if this is not the last cell and 0x80 if it is the last cell of a SDU
vcCidIndex – the VC Index, in case we are doing AAL5 reassembly, the call ID, in case we are doing AAL2 CPS reassembly
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 CPS packet
pduLen – the size of the CPS packet
partialPayload[16] – the unaligned bytes from previous CPS packet

5.13.3.3 Extract Space

RxByte writes information about received packets into extract space for the RC to use in its processing. The data structure have the following format:

Byte Offset:	0	1	2	3
0	RasType	eom	vcCidIndex	
4	PduLength	numUnAligned	pad	
8	uui	cpi	payloadLength	
12	crc			
16	partialPayload[0..3]			
20	partialPayload[4..7]			
24	partialPayload[8..11]			
28	partialPayload[12..14]			pad

rasType – 0x00 for AAL5 and 0xf0 for AAL-2
eom – 0x00 if this is not the last cell and 0x80 if it is the last cell of a SDU
vcCidIndex – VC index, in case we are doing AAL5 reassembly, call ID, in case we are doing AAL2 CPS reassembly
pduLength – the size of the CPS packet
numUnAligned – the number of unaligned bytes in the extract space
pad – unused
uui – user-to-user indication from the AAL5 trailer or CPS header
cpi – reserved
payloadLength – size of the reassembled AAL5 or SSSAR SDU
crc – CRC-32 from the AAL5 trailer
partialPayload[15] – the unaligned bytes from current CPS packet

5.13.3.4 RasList

The RasList structure is used to track re-assembly state per VC. It is an array of 128 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

5.13.3.5 RasCache

The reassembly cache is an array structures used to store parameters of recently reassembled CPS packets. The data structure has the following format:

Byte Offset:	0	1	2	3
0	validFlag	lockFlag	callId	
8-36	entry			

validFlag – indicates validity of an entry, the cache is initialized with all entries marked as invalid
lockFlag – indicates who is currently accessing the cache, aal2CpsRasIn() or aal2CpsRasOut()
callId – call ID of the SSSAR SDU referred to by the entry
entry – a reassembly data table entry as defined in section 5.23.8

5.14 SAR component

This component is a combination of the SEG and RAS components described in the previous two sections. It runs both SEG and RAS processing on a single CP. This component will have lower throughput than the combination of

SEG and RAS components but it is attractive in applications where AAL5 traffic forms a very small percentage of ATM traffic and dedicating two CPRCs for the AAL5 functionality would be an overkill.

5.15 IP QoS DiffServ

The DiffServ component provides quality of service as specified in RFC 2474 and RFC 2475. Supported service classes include an expedited forwarding class as specified in RFC 2598, four assured forwarding classes, each with three levels of drop precedence, as specified in RFC 2597, and a best effort class.

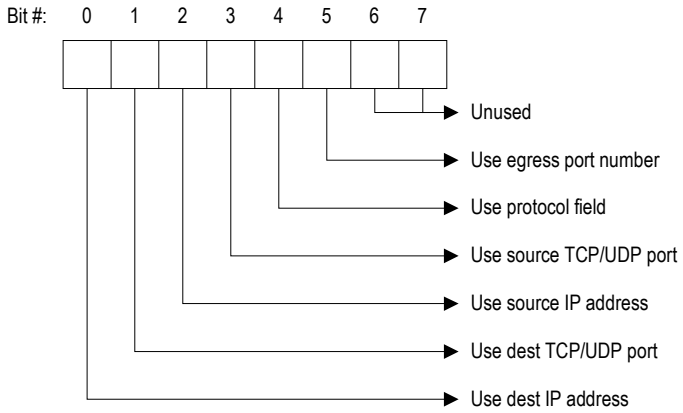
This component does not use the SDP. It uses a single CPRC to perform DiffServ QoS for all egress ports in the system. It has an input QMU queue for every egress port. Multiple fields from the IP and TCP/UDP headers are used to determine the traffic category and apply the appropriate marking and shaping strategy on it.

5.15.1 DiffServ Ingress

The DiffServ CPRC has one input QMU queue for each egress port. IP packet descriptors are dequeued from these queues in a strict round robin fashion. Packets coming to the DiffServ 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 for the IP packet, are specified in a Multi Field (MF) bit mask. They are as shown in Figure 5.

Figure 5 *DiffServ Multi-Field Bitmask*



The header fields specified by the MF mask are concatenated to form up to a 14-byte TLU lookup key. A longest prefix match table (IP flow table) is maintained at the TLU to match these keys to a flow id and a DiffServ PHB. The longest prefix match table enables a wild card prefix pattern matching on the MF mask.

5.15.2 Meter and Marker

After launching a flow table lookup, the ingress switches context to the meter context, which waits for the results of the flow table lookup. The meter context implements the single rate three color marker scheme specified by RFC 2697. It maintains a TLU data table containing the RFC 2697 parameters per flow. This 'meter' table is indexed by the flow id. The entry structure for the meter table is shown below:

```
/* The single rate three color marker RFC 2697 */
struct Srtcm
{
    int16u  cbs;        /* The committed burst size */
    int16u  ebs;        /* The excess burst size */
    int16u  increment;  /* The number of tokens added to
                           token buckets at every tick */
    int16u  tc;         /* The green token bucket */
    int16u  te;         /* The yellow token bucket */
};

struct TokenBucket
{
    int32u  cir;        /* The committed information rate */
    Srtcm   srtcm;
    int16u  lastUpdateTime;
};
```

Time is measured in ticks of 512 core clock cycles. The increment entry in the above table entry is the number of tokens that need to be added to each of the buckets at the end of 512 cycles.

The table entries are initialized in the following manner:

- Increment is calculated as CIR/(time equivalent of 512 cycles).
- TC is initialized to CBS and TE to EBS.
- The lastUpdateTime member is set to zero.

When an IP packet arrives for this TLU entry, the current cycle count is divided by the number of cycles per tick to get the number of ticks elapsed since system start time. The lastUpdateTime from the table lookup is compared to this tick count. If they are not equal, the tokens need to be added to TC and TE or TP. TC is incremented by the number Increment*(curr_tick_count – lastUpdateTime). If TC exceeds CBS, it is reduced down to CBS. Similarly TE is incremented by Increment*(curr_tick_count – lastUpdateTime) and capped at EBS.

If (TC – packet_size >= 0), then packet is within the committed information rate and is hence colored green. If (TE – packet_size >= 0), the packet is within the permitted burst excess and is hence colored yellow. Otherwise the packet is non-conformant and hence colored red . If packet is colored green, TC is decremented by packet size. Similarly, if it is colored yellow, TE is decremented by packet size.

If the PHB is an AF PHB, then green packets are marked using the low drop precedence label, yellow are marked using the medium drop precedence label, and red are marked with high drop precedence level within the PHB. The appropriate DS code point for the PHB is determined and updated in the IP header (stored in SDRAM) using a bsBufferWrite() function.

If it is an EF PHB, any packet not marked red is dropped. Green and yellow packets are marked with the EF PHB label and immediately enqueued to the QMU queue for their destination egress CPRC.

If the PHB is a best-effort category, none of the above actions are taken. Instead the packet is enqueued directly into the best effort shaper queue.

5.15.3 Shaper

Packets mapped to the AF PHBs are processed through the shaper function. There is one soft queue per AF PHB per egress IP port. This single soft queue simulates the three queues required by RFC 2498. The token bucket scheme ensures that all yellow packets follow the green packets and all red packets follow the yellow packets. Green packets are hence serviced before the yellows and yellows are serviced before the reds. This behavior is equivalent to having three soft queues, one for each color, and priority servicing the green queue over the yellow and yellow over the red.

Packets are not dropped on the basis of their color or the token bucket's state. Instead the random early detect (RED) algorithm is used to decide drop behavior on every soft queue.

5.15.3.1 RED

After a packet is marked it is run through the random early detect (RED) algorithm before enqueueing it to the soft queue. RED works by tracking the average depth of a queue and dropping packets so that this average never exceeds a threshold. This average is maintained as an exponential weighted moving average – typically measured using the characteristic equation of a low pass filter. If adding a packet causes this queue depth to exceed the minimum threshold queue depth, the packet is dropped with a certain probability. As the average queue depth keeps building beyond the minimum threshold, packets are dropped with increasing probability. If the average queue depth hits the maximum threshold, all subsequent packets are dropped until the average queue depth goes below the maximum threshold. There is one instance of RED for each AF PHB and one for the best effort PHB. The RED parameters are set as listed below:

- Minimum queue depth threshold (*MinTh):* 128 packets
- Maximum queue depth threshold*(MaxTh):* 256 packets
- Queue weight (*Wq):* 2^{-9}
- Maximum drop probability: 2^{-7}

These parameters are chosen to be optimal for the maximum soft queue depth of 256 packets. If the soft queue depth is changed, these parameters have to be recomputed. RED ensures that the drop behavior is sufficiently randomized to prevent all the TCP connections from increasing and decreasing their window sizes at the same time. It also makes sure that no connection is given unfair drop preference (or lack thereof) in the event of congestion.

An ideal RED implementation would use floating-point numbers to store the parameters and to calculate the exponential weighted moving average. However the CPRC MIPS processor does not provide floating point number support, hence the following 32 bit representation is used to store numbers at a higher resolution: The 8 most significant bits of the word represent the integer part of the number and the remaining 24 bits are the fractional part. As the soft queue max threshold is fixed at 256, the average can never exceed 256, hence 8 bits is sufficient to represent the integer part of the average. The following examples illustrate this number format:

– The number 1 is represented as 0x01000000 and the number 256 is 0xff000000.
– The number 7.5 is 0x03200000 i.e., (0x0E000000 << 1; note that 0x0E000000 is 15)
– The number 127 is 0x7E000000 i.e., (0x3F200000 >> 1; note that 0x3F200000 is 63.5)

This representation gives enough bits to represent the fractional parts of a number while retaining the equivalence of left and right shift operations to multiply and divide respectively.

The RED algorithm is implemented as follows:

1. Initialize *count* to 0xff (i.e., -1) and *average* to 0. Build a 256 entry table of random numbers between 0 and 1 (in the special extended precision format described above) offline and load it into DMEM. Repeat thru step 7 for every packet arrival event.
2. If the soft queue for the PHB is not empty, calculate the new average as *average* += (*curr. Queue depth – average*) >> (-1 * log(*Wq*)). Note that *Wq* is a negative power of two and hence log(*Wq*) is an integer.
3. If the soft queue is empty, the average needs to be updated using this equation:

   $average = (1 - Wq)^{(time\_now - last\ dequeue\ time)/s} * average.$

   Where 's' is the number of packets dequeued from the soft queue in a unit time at the line rate. This equation causes the average to decay over periods of queue inactivity. It can be approximated to the following behavior: If no packet was seen for duration *t1* or lesser, then the average remains unchanged. If no packet was seen for duration greater than *t1* but lesser than *t2*, the average is reduced to half. If no packet was seen for duration greater than *t2* but lesser than *t3*, the average is reduced by one fourth. If no packet was seen for duration greater than *t3* but lesser than *t4*, the average is reduced by one eighth. The points *t1, t2, t3 and t4* are calculated by solving the equation for average decaying to half, one fourth and one eighth respectively, for the given line rate. The egress module writes the 'last_dequeue_time' for a PHB.
4. If the average lies between the minimum threshold and the maximum threshold execute through step 7. Otherwise if the average exceeds maximum threshold, drop the packet. Otherwise enqueue the packet. In either case set *count* back to –1.
5. Claculate the drop probability *Pb* as Pb = *(average* << *log(C1))* + *C2* where *C1* is (*maximum drop probability/(maximum threshold – minimum threshold))* and *C2* is given by *(maximum drop probability * minimum threshold)/(maximum threshold – minimum threshold)*. *C1, C2* and *log(C1)* are calculated at compile time. Note that they are powers of 2 as all the numbers involved in their calculation are powers of two.
6. If *count* equals zero, fetch the next random number *R* from the table.
7. Approximate the factor *(R/Pb)* as: *R* << *(# of leading zeroes in fractional part of Pb)*. Note that the number of leading zeroes in the fractional part of *Pb* approximates to *–log(Pb).* If the *count* exceeds this approximation, drop the packet and set *count* to zero. Otherwise enqueue the packet and increment *count.*

If the soft queue was empty before the packet en-queue, its descriptor is added to an active list. The WFQ egress module picks up queues for de-queue processing from this active list in FIFO order.

5.15.4 WFQ Egress

The egress function runs in a separate context and performs deficit round robin dequeue from the AF PHB soft queues. In accordance with the AF PHB specification (RFC 2498), each PHB is associated with a minimum guaranteed bandwidth. Based on this bandwidth, a quantum is associated with each soft queue. The quantum is the maximum number of bytes that may be transmitted from the soft queue for every round of the round-robin visitation. It is initialized to the average packet size times the ratio of the bandwidth apportioned to the PHB. The host processor maintains a global array of quanta, with one entry for each PHB in every port. This array can be updated dynamically from the host.

A *next_service_time* variable is associated with every port. After transmitting a packet on the port, this variable is updated by adding the transmission time for the packet size at the port's line rate. This port will be serviced next time only when current time equals or exceeds this variable. This ensures that the diffserv CPRC does not clobber the QMU queues to the egress port and uses the RED protected soft queue to buffer traffic spikes.

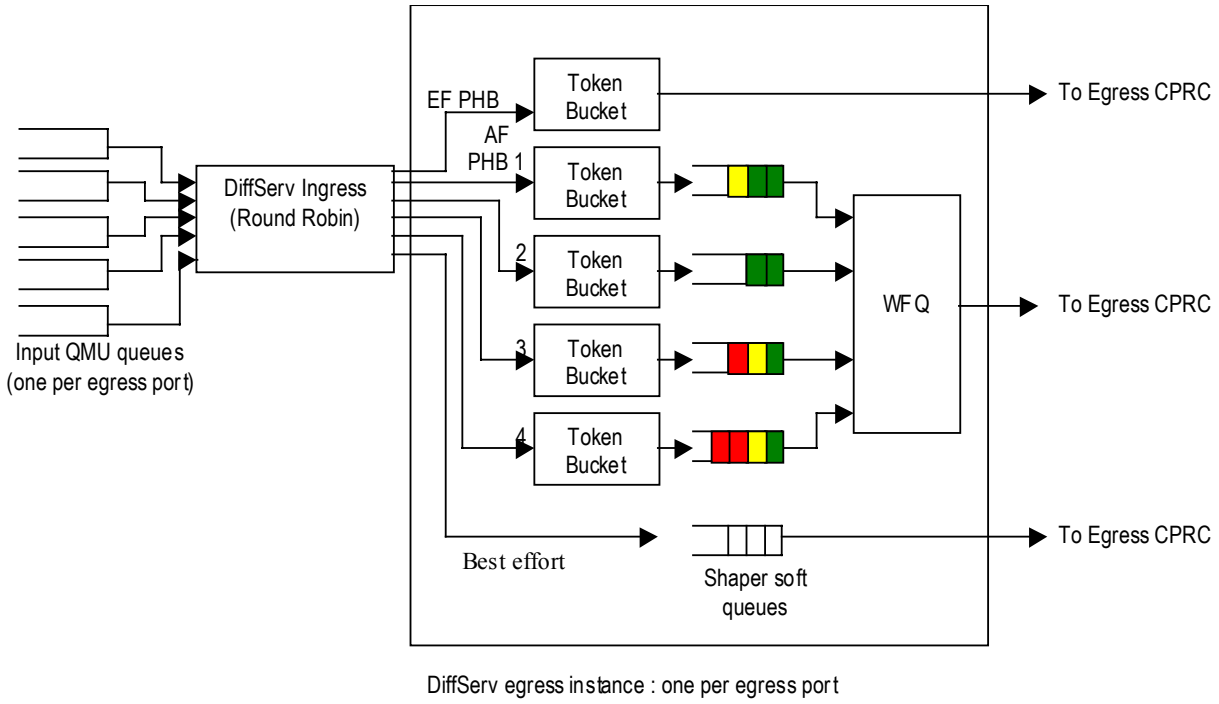
A deficit counter is associated with each PHB soft queue. It is initialized to zero. At the beginning of the round robin visitation, this counter is incremented by the quantum. If the packet at the head of the soft queue is smaller in size than the deficit counter, then the packet is dequeued from the soft queue and shipped to its destination CPRC. The deficit counter is decremented by packet size. If the packet is larger in size, than the accumulated deficit count, the

round robin procedure visits the next soft queue. A queue is serviced only after it has accumulated a sufficient number of deficits.

When the egress function finds that a soft-queue has gone empty, it removes the soft queue from the active list and resets the deficit counter for this soft queue to zero. It also sets the last enqueue time variable in the RED state entry to the current time. If the soft queue is not empty, its descriptor is added back to the active list at the tail. When the active list is empty, the egress function services the best effort soft queue.

The block diagram in the Figure 6 illustrates the functional components of the DiffServ engine.

Figure 6 *DiffServ Engine Block Diagram*



5.16 ATM TM Component

The ATM TM component provides quality of service to ATM connections. The service classes supported include CBR, rt-VBR, nrt-VBR, UBR, GFR, and UBR+. This component does not use the SDP. It consists of two subcomponents: ATM TM Policer and ATM TM shaper. These two sub-components run on separate CPs.

5.16.1 Overview

The traffic manager uses soft queues for shaping and scheduling. It supports six categories of traffic: constant bit rate (CBR), real time variable bit rate (rtVBR), non real time variable bit rate (nrtVBR), unspecified bit rate (UBR), unspecified bit rate with niminum cell rate (UBR+) and generic frame rate (GFR). Each class of service is defined in terms of a peak cell rate (PCR) in cells per second, a sustained cell rate (SCR) in cells per second, and the burst tolerance (BT) in number of cells, over PCR or SCR. GFR and UBR+ are characterized by a minimum cell rate (MCR) rather than an SCR. Conformance to these cell rates is determined using the generic cell rate algorithm (GCRA).

CBR traffic contract is specified in terms of PCR. By definition, only a small amount of burst can be present on CBR. CBR traffic should not be subject to long latencies anywhere in the system. The maximum amount of queueing delay

permissible, is specified by the parameter 'max. cell transfer delay' (maxCTD). If the traffic manager cannot guarantee maxCTD for a cell, the cell is dropped.

VBR traffic contract is specified using a PCR, SCR and a BT. It also has the maxCTD parameter similar to CBR. The maxCTD parameter for rtVBR is always small and comparable to that for CBR. nrtVBR typically has a longer maxCTD.

UBR traffic does not have any PCR or SCR requirements. It can tolerate long amounts of queue and transfer delays.

UBR+ and GFR can tolerate long delays. They are specified by a PCR and an MCR. Traffic exceeding the PCR is dropped immediately. The traffic manager buffers up UBR+ and GFR traffic and schedules it out at their MCR.

GFR traffic contract is used for AAL5 traffic. It requires the traffic manager to recognize AAL5 SDU boundaries. If a cell belonging to an AAL5 SDU on a GFR VCC is dropped, the traffic manager is required to drop all subsequent cells to the end of the current SDU. This saves bandwidth and resources as the entire packet SDU will most likely be retransmitted anyway. This behavior is called partial packet discard (PPD).

The ATM TM policer component implements the traffic contracts on all the ATM egress ports at the virtual circuit (VC) level. Every VC is associated with a GCRA record that specifies its PCR, SCR and BT. Traffic management is not implemented on cell ingress.

5.16.2 Generic Cell Rate Algorithm

Not all egress ports have the same line rate. Some egress ports could be as fast as an OC-3 line while others could be as slow as single DS0 channel. ATM TM works on the basis of the ATM cell period, i.e., time between two ATM cells at the port line rate, for each egress port.

ATM TM sets up a CPRC timer to generate a timer interrupt every cell period for the fastest egress port. It maintains a time counter that is incremented on the cell period timer expiry. This counter is used as a 'wall clock' to determine the current time.

The PCR, SCR and BT are specified in terms of the following parameters:

- *theoretical arrival time (TAT)* – the wall clock time at which the next cell is expected on this VC.
- *limit* – the number of cells that can be expected on this VC, at the TAT.
- *increment* – the number of wall clock timer ticks by which TAT should be incremented after getting *limit* number of cells on the VC.
- *maxCTD* – this parameter specifies the maximum delay that can be tolerated by a cell on this VC in terms of wall clock timer ticks.

TAT is initialized to infinity. When a cell arrives on a VC, if the current time is less than or equal to the TAT the cell is considered to be conformant. If the cell arrives (limit + maxCTD) units of time ahead of the TAT, it is considered non-conformant and is potentially subject to traffic shaping. If the cell arrives at a time earlier than (limit + maxCTD) time units from the TAT, it is considered severely non-conforming and will be dropped. After every cell that is passed (i.e., not dropped), the TAT is increased by the increment parameter.

A variant of the GCRA algorithm called the F-GCRA is used for GFR traffic. F-GCRA has the same parameters as regular GCRA and initializes TAT to infinity. However it checks for timeliness of cell arrival only on the first cell of an AAL5 SDU. If this cell is passed, all the subsequent cells until the last cell in the current SDU are passed. Note that a passed cell can be non-conforming and will be subject to shaping. TAT is increased by increment for every passed cell. If the first cell in the SDU is severely non-conforming then all the subsequent cells in the SDU up to but not including the last cell are dropped.

When more cells than limit number of cells arrive before the TAT, they cannot be transmitted immediately as that would violate the traffic contract. Such cells need to be stored away and transmitted at the next TAT. In absence of

buffer resources, they are discarded. When a GFR cell needs to be discarded because of lack of buffer resources all the subsequent cells in the SDU are discarded.

5.16.3 Traffic Shaping and Scheduling

Traffic shaping and scheduling is implemented by the ATM TM scheduler component. It receives the results of GCRA computation from the policer component. Based on these results, a cell may need to be stored away for transmission at a future point of time. WNI 1.1 and prior releases provide soft queue based scheduling. WNI 2.0 uses calendar structures to obtain tighter jitter control. Both these schemes are described below.

5.16.3.1 Soft queue based ATM scheduling

The ATM TM component maintains 30 soft queues per traffic category for the purpose of storing cells away for future transmission. A different set of per traffic category soft queues is used per egress port. Among the 30 soft queues one queue in each category is marked as the current queue. The cells determined to be 'conformant' by GCRA go into the current queue within their category.

One soft queue in each pool is labeled as the 'current' soft queue. On every wall clock tick, a cell is dequeued from one of the set of current queues using a priority round robin algorithm, for each port. The CBR queue has the highest priority, rtVBR has medium and nrtVBR has the lowest priority. The current rtVBR queue is serviced only if the current CBR queue is empty and the current nrtVBR queue is serviced only if the current CBR and rtVBR queues are empty. If all the current queues are empty, the UBR queue is serviced. The current queue pointer in every category is updated to point to the next soft queue after every 'max_queue_depth' ticks of the wall clock, i.e., after a potentially full soft queue has been completely drained.

By design, the current queue contains all those cells that need to go out during the current cell period. However, only one cell can be transmitted during the current cell period. The transfer delay (or latency) experienced by a cell is equal to the number of cells ahead of it in the current queue. To track this number of cells, a depth counter is maintained for every soft queue. It is incremented for every cell enqueued and decremented for every cell dequeued.

For a cell determined to be 'non-conformant' by GCRA, a destination queue is decided using the following equation:

```
dest_queue = (TAT – wall_clock)/max_queue_depth;
```

Let Qi, j denote soft queue i in pool j. A priority round robin algorithm used to dequeue cells from the soft queues. Hence the time spent by a cell in the soft queue (i.e., its transfer delay) is determined as:

```
transfer_delay = Depth(Qi, j) + Depth(Qi, j-1) + Depth(Qi, j-2)
   + … + Depth(Qi, 1);
```

Pool 0 is the CBR soft queue pool, pool 1 is rtVBR soft queue pool, and pool 2 is the nrtVBR soft queue pool. The transfer delay is determined this way before a cell is enqueued. If it exceeds the maxCTD for the cell, it means that the maxCTD cannot be satisfied and the cell is dropped.

5.16.3.2 Calendar based scheduling

This method of scheduling organizes the descriptor storage buffer as a calendar array of cell slots. One of the slots in this array is always marked as current. Based on the results of the GCRA computation, if a cell is conformant it is inserted into the current slot. If a cell has been found to arrive *n* cell periods prior to its TAT, it is inserted in the slot *n* slots away from the current slot. If the target storage slot for a cell is occupied by another cell the cell is inserted in the next immediately available vacant slot in the calendar. Slot occupancy is tracked using a bitmap in CPRC DMEM. The calendar is serviced once every cell slot and the cell from the current slot is extracted and sent to the destination.

There is one calendar array per traffic category. Calendars are serviced in a strict priority order with CBR having highest priority and UBR having the lowest. A cell is extracted from the current slot of the rtVBR calendar only if the current CBR slot is empty. Similarly a cell is extracted from the current nrtVBR calendar only if the current CBR and

rtVBR slots are empty. UBR+ and GFR share a common calendar. This calendar is serviced only if the current slots in all other calendars are empty.

5.16.4 RC

The ATM TM component runs on a two CPRCs: ATM TM Policer CP and ATM TM scheduler CP. There are two instances of the ATM TM component - one for VP level QoS and the second for VC level QoS The policer has one input queue for each ATM egress port serviced. After processing an input cell using the GCRA algorithm, a time_to_send is associated with the cell. The cell, along with the 'time_to_send,' is dispatched to the scheduler CP. The scheduler buffers it in a soft queue according to the algorithm discussed in section 5.16.3 and drains the queues in strict priority order at the egress port's drain rate.

5.16.4.1 Policer module

This module dequeues cell descriptors using a fair round robin scheme across all ports. After dequeueing a descriptor, this function launches a TLU lookup to fetch the GCRA parameters for the dequeued cell's VC. It then puts the cell descriptor in a work queue and proceeds to service the next input queue in round robin order. The work queue is a four element DMEM array that is accessed in FIFO order by the ingress and shaper modules. The ingress module writes to it and the shaper reads from it. This module does a context switch to the GCRA context whenever it needs to wait for a message receive to complete. The GCRA context runs the algorithm presented in section 5.16.2, determines an exit time for the cell, and enqueues the cell to the shaper module.

5.16.4.2 Shaper module

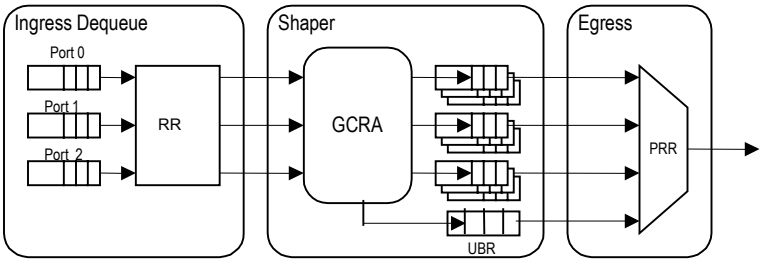
This module receives the cell from the policer module and enqueues it to one of the soft queues according to the algorithm discussed in section 5.16.3 Soft queues were used in and prior to WNI1.1. WNI2.0 uses the calendar based scheme.

5.16.4.3 Egress module

When soft queues are used, this function performs priority round robins dequeue from the soft queues marked as current. It then updates the soft queue depth count and, when necessary, it updates the 'current' mark to point to the next soft queue.

The block diagram of Figure 7 illustrates this design.

Figure 7 *ATM TM Component*



When calendars are used, this function performs strict priority calendar servicing.

5.17 AAL-2 Receive component

The AAL-2 receive component splits AAL-2 ATM cells into AAL-2 CPS packets and forwards them to the next processing block to provide CID switching and AAL-2/SSSAR functionality.

The OC-3c ATM Rx, TDM ATM Rx CPRCs and UL-2 RxFDP copy the first byte from the cell payload and forward it as a part of the cell descriptor. For AAL-2 VCCs, this byte is the STF header. The AAL-2 Rx CP uses it to parse the cell payload for CPS packets. This byte is ignored by the other CPs (like AAL-5 RAS and ATM QoS) that receive ATM cell descriptors.

5.17.1 SDP

The SDP is configured for byte level recirculation. The SDP verifies the CRC-5 of the CPS header and launches CID lookups.

5.17.1.1 TxByte

The TxByte processors performs the following functions:

- Reads the buffer handle and vcIndex from merge space and streams it out to the RxByte.
- Streams the CPS packet header.

5.17.1.2 RxByte

The RxByte performs the following operations:

- Copies the buffer handle and CID to extract space.
- Performs CRC-5 verification on the CPS header.
- Concatenates the vcIndex and CID to form the CID table lookup key.
- If the CRC-5 passes, it launches a CID table lookup.
- Gives the extract space to the CPRC.

5.17.2 RC

The AAL-2 Rx CPRC consists of two modules running in two separate contexts. The AAL-2 RxIn module receives input from a QMU queue, parses it for CPS packets, sends CPS headers down to SDP for correctness verification and accumulates CPS packets in SDRAM. The AAL-2 RxOut context receives the CPS header verification result and the CID table lookup results from the SDP and disposes the CPS packets according to the CID table lookup.

5.17.2.1 AAL-2 Rx Input

The AAL-2 RxIn module performs the following sequence of steps:

- Dequeue a cell descriptor from input queue.
- Issue a bsBufferRead() to start a DMA transfer of cell payload to SDRAM.
- Verify that the STF byte has correct parity. This is done using a DMEM based parity check table.
- Verify that the STF byte has the correct sequence number (SN) for the VCC. This is done by comparing the SN value in the STF byte to the nextSN field in the state table. If they match, the STF byte has the correct SN and the nextSN field in the state table is toggled.
- Extract the OSF portion of the STF byte.
  - If the OSF is zero, a CPS header immediately follows the OSF header.
  - If OSF is not zero, it means that this cell payload starts with the second part of a split CPS packet – the first part had arrived in the previous cell on this VC

- If OSF is zero and we are holding unaligned bytes in the state table, it's an error condition. We need to discard the previous CPS packet in this case and reinitialize the state table entry

Wait for the DMA transfer to complete and then do the following:

- If OSF is zero, then read the CPS header immediately following it. If the LI field indicates that the entire CPS packet is present in this cell, do a memcpy() of the payload to a word aligned dmem location. Copy the header into the merge space, allocate a SDRAM buffer for the CPS packet and transfer the CPS packet body by doing a bsBufferWrite(). Repeat this for all complete CPS packets found in the cell payload.
- If OSF is non-zero, then use the VC index to fetch the reassembly state record from the state table. Fetch the CPS header from the reassembly state table. If it is incomplete, complete it using information from current cell payload. Copy out the unaligned bytes to the beginning of a word aligned dmem location. Copy out the payload from offset location 2 to OSF from the cell payload to the dmem location following the unaligned bytes. If the CPS packet is still not complete, extract the unaligned bytes into the reassembly state table and DMA the rest of the payload. If it is complete, DMA the entire payload and copy the CPS header to the merge space. Repeat for all the remaining CPS packets in the payload.

5.17.2.2 AAL-2 Rx Output

The AAL-2 RxOut module performs the following sequence of steps:

- Wait for the SDP to hand over the extract space.
- The extract space contains the buffer handle, the CPS header and a flag indicating whether the CRC-5 passed on the header.
- If the CRC-5 flag shows that the header is correct, the SDP would have launched a CID table lookup. Wait for the lookup results. If the header is incorrect, free the buffer associated with the packet and loop.
- If the CID table lookup indicates that this CPS packet needs to be processed from SSSAR, dispatch the packet descriptor to the SSSAR CPRC. Otherwise change the CID and cell header as specified in the table lookup and forward it to the AAL-2 Tx CPRC.
- Loop and wait for next extract space.

5.17.3 Data Structures

5.17.3.1 AAL-2 Rx CPS Reassembly State

CPS packets can be potentially split across two ATM cells. When a cell with a partial CPS packet is received the partial payload needs to be stored away and the header should be processed only when the remaining part of the CPS packet arrives on the next cell. The AAL-2 RxIn context stores CPS payloads in BMU controlled SDRAM. The BMU hardware requires all SDRAM write operations to start at a 16 byte aligned address. But CPS packets do not come in sizes that are multiples of 16. When a partial CPS packet is received, it may not end at a 16 byte boundary, but the CPS payload from the next packet has to be appended to it.

A split CPS packet is not completely written to the BMU. If the size of the packet is *s* bytes, only (s - *s%16)* bytes are written out to the BMU. The remaining bytes are called the 'unaligned bytes' and are stored in a DMEM state table. When the next packet arrives on this VCC, these unaligned bytes are attached at the beginning of the payload and then dispatched to the BMU along with the rest of the packet. Unaligned bytes are extracted and stored every time a CPS packet straddles cell boundaries.

The CPS header too could straddle a cell boundary. When this is the case, the partial cell header is extracted and stored in the reassembly state table. When the next cell arrives on this VCC, the remaining CPS header bytes immediately follow the STF header.

The VC Index (from the ATM VC table) indexes this reassembly state table. This table can be moved to TLU if more than 256 entries needed or if the table cannot fit in DMEM.

The RC saves reassembly state information in a data structure that has the following format:

Byte Offset:	0	1	2	3
0	bufHandle			
4	Offset		nextSN	pad
8	numUnaligned	unalignedBytes[0..2]		
12	unalignedBytes[3..6]			
16	unalignedBytes[7..10]			
20	UnalignedBytes[11..14]			
24	numCpsHdrBytes	cpsHdrBytes[3]		

bufHandle – handle of the buffer containing the reassembled CPS packet
offset – the offset at which the next part of the CPS packet should be placed
nextSN – the next expected sequence number
pad – unused
numUnaligned – number of unaligned bytes
unalignedBytes[15] – the unaligned bytes to preceed the next recirculation
numCpsHdrBytes – number of saved CPS header bytes
cpsHdsBytes[3] – the saved header bytes of a partial CPS header

5.18 AAL-2 Transmit component

The AAL-2 transmit component receives AAL-2 CPS packets that were CID switched or segmented via SSSAR and merges them into AAL-2 ATM cells before forwarding them to the next processing block.

5.18.1 SDP

The SDP is configured for byte level loopback. The SDP is responsible for CPS header generation and separating out unaligned bytes.

5.18.1.1 TxByte

The TxByte processor does the following:

- Reads the CID, LI and UUI from the merge space.
- Performs the bit manipulations necessary to pack these three fields into the CPS header.
- Accumulates the CRC-5 on the generated CPS header.
- Streams out the state table offset, CPS length followed by the CPS header to the RxByte processor.

5.18.1.2 RxByte

The RxByte processor simply copies all fields received from the TxByte processor into the extract space and transfers control to the CPRC.

5.18.2 RC

The AAL-2 Tx component consists of two modules, namely aal2TxIn and aal2TxOut, running in two separate CPRC contexts. Aal2TxIn dequeues CPS descriptors from the input queue and feeds them to the SDP. The aal2TxOut module is responsible for running the CU timer. It also dispatches the cells returned by the SDP.

5.18.2.1 AAL-2 Tx Input

The AAL-2 TxIn module dequeues CPS packet descriptors from the input queue and fetches the payload data into DMEM. It maintains two 64 byte arrays to hold the payload data, called 'cpsStageArea' and uses them alternately for successive packets. It performs the following sequence of operations on each CPS packet:

- Reads payload into one of the cpsStageArea arrays in dmem, via a DMA operation.
- While the DMA is in progress, reads the CID, LI and UUI fields from the CPS descriptor and copies them to the merge space space.
- Based on the VC index, determines the offset into the Tx state table and writes the offset to the merge space.
- Hands the extract space over to the SDP. The SDP will construct the CPS header.
- After the payload read DMA transaction has completed, loop back to service the next CPS packet.

5.18.2.2 AAL-2 Tx Ouput

The AAL-2 TxOut module is responsible for packing the CPS packets into ATM cells. It maintains a 64-byte DMEM array called the 'launchPad'. CPS payload from the cpsStageArea is packed with the CPS header from the SDP into the launchPad. Aal2TxOut executes the equivalent of the following algorithm:

- Read the CPS header, state table offset and CPS packet length from the extract space. Increment the CPS length by three to account for the header. The CPS length will be decremented for each transmitted byte of the CPS packet.
- Repeat the following actions while the CPS length is greater than zero.
- If the offset field in the state table is zero, then we are starting a new cell. If we are at the beginning of a new CPS packet at this time, create the STF header byte using an offset of zero. Otherwise use the current value of the CPS length field as the offset and create the STF header. Copy the STF header into the launchPad.
- If the offset is not equal to zero, this is not the first CPS packet in the cell. Copy the unaligned bytes from the state table into the launchPad.
- Copy the CPS header at launchPad[launchPadIndex] one byte at a time.
  - With each byte of the CPS header increment the offset field in the state table entry.
  - If the offset equals the cell size of 48 at any point during this copy, a complete cell has been accumulated in the launch pad. Do a DMA write from the launch pad to the SDRAM and perform the transmitCell() procedure. Loop to start a new cell.
- Copy the CPS payload into the launchPad, one byte at a time.
  - Increment the offset for each byte of payload.
  - If the offset equals 48 at any point, a complete cell has been accumulated. Do a DMA write from launchPad to the SDRAM and perform the transmitCell() procedure. Loop to start a new cell.
- If this CPS packet did not fill the cell payload, we need to wait CU time units from now before transmitting this incomplete cell. Transfer the launchPad to the SDRAM. If this is the first CPS packet in the cell, update the startTime field in the state table. The prevNode and nextNode fields form a logical doubly linked list structure on the state table entries. Add the current state table entry to the end of this list. Note that the list always contains entries in increasing order of startTime fields.
- If the CPS packet was *s* bytes long *(s – s%16)* is the number of unaligned bytes at its end. Copy these unaligned bytes into the state table.
- Check the current time and calculate the difference between the time now and the start time of the node at the head of the linked list. If this difference exceeds CU, then transmit the incomplete cell associated with that node. After transmitting the cell, update the head of the linked list to point to the next element.
- Control reaches here when the entire CPS packet has been processed. At this point keep performing the the timer check until the SDP hands over a new CPS header, then loop and start the next CPS packet.

The transmitCell() procedure is as follows:

- Create a cell descriptor using the cell header from the state table entry.

– Do a port table lookup for the egress port to determine the egress queue for the cell.
– If the port table lookup indicates that the egress port is in service, send the cell descriptor on the egress queue.

The above procedure is a canonical description of the Aal2Tx algorithm. The actual implementation has several optimizations so that it uses memcpy() (or equivalent) operations instead of iterative loops to transfer one byte at a time.

When the CU timer expires, the remaining unfilled portion of the cell needs to be zeroed out. The unaligned bytes are copied to the launchPad and the rest of the launchPad is filled with zeroes. The launchPad is then DMA transferred to SDRAM and the transmitCell() procedure is invoked to dispatch the cell.

5.18.3 Data Structures

5.18.3.1 AAL-2 Tx Reassembly State

The BMU restricts SDRAM writes to start from 16 byte boundaries. CPS packets do not necessarily come in sizes that are multiples of 16 bytes. However CPS packets need to be appended one behind the other in an ATM cell. Hence when assembling the ATM cell, it might become necessary to write starting from a non-16 byte boundary. To satisfy the BMU constraints, the partially filled 16-byte line at the end of a packet is not written immediately to the BMU. Instead it is stored in a dmem based state table.

In addition to holding unaligned bytes, the reassembly state table entry for a VC also holds the reassembly start time i.e., the time at which the first CPS packet arrived at this VC. When CU time units have expired since the arrival of the first packet, the cell is padded out and dispatched. When a cell payload fills up, the cell is dispatched irrespective of whether the CU timer has fired or not.

An array of these structures is maintained in dmem. The entry for a VCC is located at the index equal to its VC index from the ATM VC table.

The following structure is used for each entry in the reassembly state table:

Byte Offset:	0	1	2	3
0	Btag		EgressPort	
4	offset	PrevNode	nextNode	sn
8	numUnaligned	unalignedBytes[0..2]		
12	unalignedBytes[3..6]			
16	unalignedBytes[7..10]			
20	UnalignedBytes[11..14]			
24	StartTime			
28	CellHeader			

btag – the buffer tag of the buffer containing the packed ATM cell
egressPort – the egress port for the ATM cell
offset – the offset within the cell payload at which the next CPS packet should be placed
prevNode – linked list pointer to previous node used for CU timer functionality
nextNode – linked list pointer to next node used for CU timer functionality
sn – the sequence number
numUnaligned – number of unaligned bytes
unalignedBytes[15] – the unaligned bytes to preceed the next recirculation
startTime – the time at which creation of this ATM cell started
cellHeader – the cell header to be applied to the ATM cell being created

5.19 CID QoS component

The primary objectives of the CID QoS component are the following:

– CID connection metering and policing to ensure that misbehaving and bursty connections cannot deny service to other well behaved connections.
– To prioritize smaller packets over larger ones when packing packets from different CIDs together. This reduces the net latency on the smaller packets especially on low speed TDM links.

The metering and policing is performed using a leaky bucket construction and the packet scheduling is performed using deficit round robin(DRR) as an approximation to weighted fair queueing (WFQ). One CP is used for metering and a second CP is used for DRR scheduling. The SDPs are not used in the CID QoS component.

5.19.1 Metering and Policing

A per connection token bucket mechanism is used to meter and police the CID connections. Each CID connection is associated with a dynamic number of tokens and this number along with other token replenishment state information is stored in a TLU data table. The following structure describes the relevant state information

```
Struct CidMeter
{
        int32u  numTokens;
        int32u  maxTokens;
        int32u  lastPacketTime;
        int16u  increment;
        int8u  priorityClass;
}
```

The numTokens field holds the current number of tokens, where 1 token provides for 1 byte of data to be passed through the token bucket. The maxTokens field is an upper bound on the number of tokens. It defines the maximum tolerated burst size. The priority class field is used by the scheduler – the meter does not use it. The increment parameter is the number of tokens to be added to the bucket over a time period of one wall clock tick. A wall clock tick is defined as 512 NPU core clock cycles. The wall clock itself is determined as contents of the NPU cycle count register divided by 512 (right shifted by 9 bits). One tick of the wall clock hence corresponds to an addition of one increment worth of tokens to the bucket.

When a packet arrives at the metering CPRC, the stored wall clock time in the CidMeter structure is subtracted from the current wall clock time. The difference is multiplied by the increment to determine the additional number of tokens. This product is then added to the numTokens field to give the current number of tokens in the bucket. If the resulting number exceeds the maxTokens, it is brought down to maxTokens. If the packet size exceeds the number of tokens, the packet is discarded. Otherwise, the packet is sent to the scheduler CPRC and the numTokens field is reduced by the packet size.

Two CPRC contexts viz., cidMeterIn and cidMeterOut are used at the meter CPRC. This CPRC has one input QMU queue per physical egress port. The cidMeterIn context performs a fair round robin packet dequeue from the QMU queues and launches a TLU lookup to fetch the CidMeter state for the packet's connection. The cidMeterOut context receives the results of the TLU lookup and runs the token bucket algorithm discussed in the previous paragraph. After processing the packet (i.e, either dropping it or enqueueing it to scheduler CPRC), cidMeterOut writes the CidMeter state back to the TLU.

5.19.2 Scheduling

Scheduling is performed using DRR on soft queue pools holding CID level packets. Four CID priority classes are defined and one soft queue pool is assigned to each class. This collection of four soft queue pools is then replicated

once for each physical egress port. The four soft queue pools for each port are serviced using deficit round robin algorithm similar to DiffServ. The pools for each port are serviced at the port's line rate. This ensures that DRR has sufficient time to schedule smaller packets ahead of larger ones on low speed TDM ports.

The scheduler CPRC runs two contexts viz., cidSchedulerIn and cidSchedulerOut. CidSchedulerIn dequeues input packet descriptors from the meter CPRC. The packet descriptor contains the priority class info. from the CID meter table. The packet is enqueued into the soft queue corresponding to the traffic class.

CidSchedulerOut is responsible for DRR processing from the soft queue pools. Every OC3 cell period (800 cycles at core frequency of 266MHz) it services the soft queue pools for the two OC-3 ports and it services the TDM ports once for after processing the OC-3 ports 75 times.

5.20 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 CPRC and the OC-3c ATM CPRCs provide the ATM interface on which TDM circuit traffic is sent AAL-1 SDUs. The ATM VCCs used for circuit emulation is configured for constant bit rate (CBR) in the ATM traffic manager.

This component supports up to 64 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 16394 KBps.

This component gets clear channel TDM chunks from the TDM CPRC. 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.

5.20.1 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 of the dequeued 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 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.

5.20.2 Aal1TxOut

This thread is responsible for enqueueing completed ATM cells to its destination. It waits for the results from the port table TLU lookup launched by aal1TxIn. If the RTS is not zero, it is copied on all the SDUs in the current cycle, as specified by I.363. It creates a cell descriptor and based on the lookup results enqueues the cell descriptor either to the ATM TM CPRC or directly to an ATM egress CPRC.

5.20.3 Data Structures

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

5.20.3.1 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		nextSn	offset
4	pointerSent	structSpill	UnalignedBytes[0..1]	
8	unalignedBytes[2..5]			
12	unalignedBytes[6..9]			
16	unalignedBytes[10..13]			
20	unalignedBytes[14]	Pad		

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
nextAal1Sn – AAL-1 sequence number to be used on the next SDU

pointerSent – boolean value specifies whether an AAL1 pointer has been sent in this cycle
structSpill – field specifies the number of payload bytes before start of the next structure
unalignedBytes[15] – the unaligned bytes to be appended to the start of the next chunk
pad - unused

5.21 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 CPRC and the OC-3c ATM CPRCs provide the ATM interface on which TDM circuit traffic is sent AAL-1 SDUs. The ATM VCCs used for circuit emulation is configured for constant bit rate (CBR) in the ATM traffic manager.

This component supports up to 64 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 16394 KBps.

5.21.1 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:

- Dequeue 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 AAL-1 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 be corrected, discard the cell and go back to process the next cell.
- 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 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 CPRC with the next chunk descriptor and clear it. The TDM CPRC forwards it to Twister, 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 CPRC. Repeat this till the entire AAL-1 SDU payload has been exhausted.

5.21.2 Data Structures

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

5.21.2.1 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

5.22 OAM Processing

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

Forward Performance Monitoring – Receive Monitoring

- 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.
- 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.

Forward Performance Monitoring

- Handling of the OAM Activation/Deactivation cell for PM activation.
- Maintenance of user cell counts and lost cell counts.
- Transmit FPM Insertion: An OAM -FPM cell is generated by the transmitter every block with the correct BIP-16 calculated for even parity when enabled. This feature will be added in a future release.
- Backward Performance Monitoring
- If backward reporting has been enabled, an OAM-BR (Backward Reporting) cell indicating the number of errors (along with other information - e.g. if a cell was dropped) is transmitted upon reception of an OAM-FPM cell.
- Fault Management (used for fault localization)
- ATM Layer VP-AIS and VP-RDI defect indications – transmitted upon detection of path errors from the physical layer (e.g. SONET AIS-P) or LCD (loss of cell delineation) or LOC (loss of continuity at the VP level).
- ATM Layer VC-AIS and VC-RDI defect indications – transmitted upon detection of path errors from the VP layer (VP-AIS) or detection of LOC (loss of continuity at the VC level).
- VP/VC Loopback – loopback cells injected/enabled via TMN or customer.
- VPC/VCC Continuity Check – enabled or disabled via TMN for a limited number of connections, OAM continuity check cells are sent and monitored to ensure continued connectivity.

OAM processing uses the ATM VC table as described in <<section 5.23.5.>>

5.22.1 SDP

The TDM CP and the OC3c ATM CPs support OAM performance monitoring. The SDP processors on these CPs do the following:

5.22.1.1 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 forward a pass-fail notification to the rxByte processor.

RxSync is not configurable through its control space.

5.22.1.2 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.

The RC configures RxByte through a data structure mapped into its control space. See the relevant OC3c ATM and TDM RxByte sections for this information and information about extract space.

5.22.2 RC

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

5.22.2.1 Initialization

During initialization the following things happen:

– The 128 entry OAM PM table is initialized.

5.22.2.2 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. See section 5.22.3.1 for a description of the OAM FPM table.
– Increments and masks the CurrentBlockValue (ranges from 0 to BlockSize-1).

Else, 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 indicate 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). Increment the TotalBip16Errs counter with this information.

Handling of OAM FPM cells received beyond the BlockSize is TBD. Double buffering of state information might be required in the future.

5.22.3 Data Structures

5.22.3.1 OamPmTable

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

Byte Offset:	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

5.22.3.2 SONET Monitoring

The SONET monitoring function for the C5e will work as it does for the C5 except for how B1/B2/B3 and REI_L error counts are collected. In the C5, the errors are detected and collected on every frame and then processed in the soaking function and reported to the XP. In the C5e, the errors are accumulated in overhead accumulation registers for each of the errors. The registers have the ability to accumulate up to one second of errors. There is a bip_accum_interrupt that will occur if ANY of the 4 errors occur during the last framecnt period, which has a max

value of 8192. The SONET monitoring code will set up the framecnt to a large value and when a bip_accum_framecnt interrupt occurs, the SONET ISR will read the B1/B2/B3 and REI_L accum overhead bytes and add then to BIP_COUNTS variables. Then it toggles the Frame_Mode bit in the SDP_Mode4 register to cause the overhead accum counts to reset. This process will be used on the C5e when APS is disabled. When APS is enabled, the SONET monitoring process does not change, but the occurrence of interrupt based on framecnt will increase to allow APS to monitor the B2 errors more frequently to ensure that the system meets the APS switch times.

Signal Failure (SF) and Signal Degraded (SD) will be set and cleared by each of the CPRC receive contexts using the new SDP mode and SONET register fields, a timer, and BER thresholds. The implementation of the SONET monitoring software will remain the same except for the process for accumulating the B1/B2/B3 and REI error counts.

Currently, the error counts are accumulated in the CPRC using the error count that is associated with N frame's via an interrupt. Since APS runs in frame mode, setting bit 31 in the SDP_Mode4 register will change the meaning in the SONET_Event register bit 19 for B2 errors interrupt. This forces a change in the way errors are accumulated in the CPRC. The component will accumulate the B1,B2,B3 and REI_L errors in overhead accumulation bytes listed above. Whenever an error occurs, the overhead accumulation bytes will be read and the values added to the software accumulation variables in the CPRC. The reading of the overhead accumulation bytes will change slightly depending on if the current state is SF, this is described below.

Signal Degraded is only declared based on BER's. Signal Failure is declared based on BER's OR LOS, LOF or AIS-L defects. The two types of Signal Failure are handled differently in the CPRC and are referred to as SF-DEF and SF-BER. SD can both co-exist and will be detected and reported independently of each other. The Host/XP code only cares that SD and/or SF are declared or cleared and not if the SF condition was due to BER or a Defect.

When APS is enabled, the Bit Error Rates (BER) for declaring will default to (1E3) for Signal Failure and (1E5) for Signal Degraded. The BER for clearing SF and SD will be ten times lower that the declaration BER as per GR-253. The user can configure the SF and SD declaration BER's these when APS is enabled.

5.23 Component table usage

All components use TLU tables for PDU switching/routing and maintaining protocol state. Some tables need to be shared by multiple components Ex: IPv4 and IPv6 routing tables need to be shared by Ethernet, IP and POS components as they all have IP routing functionality. The Table Lookup Unit (TLU) provides lookup table management. The TLU stores its table information in external SRAM that is managed by the TLU's SRAM controller. The application creates multiple tables in the TLU for data path forwarding and state memory support. The tables are summarized in Table 2 and described in more detail in subsequent sections.

Table 2 *TLU Table Summary*

Table Name	Table Type	Table ID	Key Size (b)	Entry Size (B)
Port Table	Data	0	32	8
ML-PPP Remainder	Data	1	32	16
IPv4 Route Table	LPM	2	48	16
IPv6 Route Table	LPM	3	96	16
ATM VC Table	LPM	4	48	16
CID Table	HTK	5	32	16
Reassembly CRC	Data	6	32	8
Reassembly Data	Data	7	32	32
DiffServ Flow Table	LPM	8	112	8
DiffServ Meter Table	Data	9	32	16

Table Name	Table Type	Table ID	Key Size (b)	Entry Size (B)
ATM TM Table	Data	10	32	16
CRC-32 Correction Table	Data	11	32	8

5.23.1 Port Table

When the ingress processor launches a lookup to make a forwarding decision, the response usually returns the port number and layer 2 address. 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 meaning of each field is as follows:

portType – the type of egress port, ATM or PPP
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
- b5: Compression – flag indicating whether compression 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 (IP packets only)

5.23.2 ML-PPP Remainder Table

When ML-PPP reassembly takes place, buffer writes can not 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 then be fetched an 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]			remainerLen

The meaning 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

5.23.3 IPv4 Route Table

The IPv4 routing table is a LPM table with a 48-bit key. The key is equal to the concatenation of IP destination address and UDP destination port of the datagram of interest as shown here:

Bits:	47 16	15 0
	IP Dest Address	UDP Dest Port

If the datagram is not a UDP datagram, the last 16 bits of the key are set to 0.

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 meaning 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 meanings depending on the egress. For example, an Ethernet egress would have a MAC address in this field, while an ATM egress would have a VPI/VCI. The MAC appData field has the following format:

Byte Offset:	0	1	2	3
0	macDa[0..3]			
4	macDa[4..5]		pad	

The meaning of each field is as follows:

macDa – the MAC destination address to use at the egress
pad – unused

The ATM appData field has the following format:

Byte Offset:	0	1	2	3
0	egressCellHeader			
4	CpsPduLen	Cid	pad	

The meaning of each field is as follows:

egressCellHeader – the cell header (VPI/VCI) to use at the egress
cpsPduLen – the CPS payload size for AAL-2 SSSAR
cid – the AAL-2 channel ID, if zero AAL-5 segmentation is required
pad – unused

The TDM appData field has the following format:

Byte Offset:	0	1	2	3
0	compressionId			
4	McClass	pad		

The meaning of each field is as follows:

compressionId – the compression ID used by IPHC
mcClass – the MC-PPP class
pad – unused

5.23.4 IPv6 Routing Table

Because IPv6 addresses are 128 bits long, lookups are done in two stages. The first lookup uses a key that includes the first 64 bits of the IPv6 destination address. The response may indicate that another lookup is not necessary, in which case the response data includes the necessary routing data. Otherwise, another lookup must be launched. The key associated with the second lookup is a 16-bit number from the first lookup's response data concatenated with the remaining 64 bits of the IP destination address and the UDP destination port. If the datagram is not a UDP datagram, the last 16 bits of the key are set to 0. The key is shown here:

Bits:	95 80	79 16	15 0
	Tag	Half IPv6 Dest Addr	UDP Dest Port

Each entry of the table has the following format:

Byte Offset:	0	1	2	3
0	Port		type	maskBits
4	appData			
8	appData			
12	Tag		numLinks	

The meaning 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
tag – if zero, entry data is valid, otherwise launch another lookup
numLinks – number of other table entries that share the same initial key

The appData field has the same format as specified for the IPv4 route table in section 5.23.3.

5.23.5 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:

Bits:	47 44	43 36	35 20	19 16	15 0
	0	VPI	VPI	0	Port

Each entry of the table has the following format:

Byte Offset:	0	1	2	3
0	egressCellheader			
4	VcIndex		flags_egressPort	
8	EgressQueue		port_bufType	
12	DestQueue		oamPm	pad

The meaning 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 the AAL-2 and AAL-5 SARs to index their state tables
flags_egressPort – a bitmap as follows:

- b15: AAL2 VC – flag indicating the VC is an AAL-2 type VC
- b14: AAL5 VC – flag indicating the VC is an AAL-5 type VC

- b13-11: 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
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

5.23.6 CID Table

The AAL-2 CID table is used to determine where to forward packets to for a particular CID. The key is made up of the VC index and CID as follows:

Bits:	31 24	23 8	7 0
	0	VC Index	CID

Each entry of the table has the following format:

Byte Offset:	0	1	2	3
0	Flag	DestVpi	sssarCallId	
4	DestVci		destPort	
8	DestVcIndex		destCid	pad
12	pad			

The meaning of each field is as follows:

flag – whether this CPS packet should be CID switched or processed for SSSAR reassembly
destVpi – destination VPI for CID switching
sssarCallId – call ID to use for SSSAR reassembly
destVci – destination VCI for CID switching
destPort – destination port for CID switching
destVcIndex – destination vcIndex for CID switching
destCid – destination CID for CID switching
pad – unused

5.23.7 Reassembly 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	
8	crcPartial			

The meaning 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

5.23.8 Reassembly Data Table

The reassembly data table stores state information for both AAL-2 SSSAR and AAL-5 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	bufHandle			
4	offset		startTime	
8	partialPayload[0..3]			
12	paritalPayload[4..7]			
16	partialPayload[8..11]			
20	partialPayload[12..14]			numBytes
24-28	pad			

The meaning of each field is as follows:

bufHandle – reassembly buffer handle
offset – buffer offset at which next cell payload should be placed
startTime – time at which reassembly began
partialPayload[15] – number of unaligned bytes to save for next time
numBytes – length of partial payload
pad – unused

5.23.9 DiffServ Flow Table

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

Bits:	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	
4	Pad			

The meaning of each field is as follows:

maskBits – number of significant bits in the key.
phb – per hob forwarding behavior to apply.
flowId – an ID for the flow used as in index in the meter table.
pad – unused.

5.23.10 DiffServ Meter Table

The DiffServ meter table stores the parameters for the QoS algorithms used by the DiffServ component. The key is the flowId retrieved by a lookup into the DiffServ flow table. Each entry of the table has the following format:

Byte Offset:	0	1	2	3
0	cbs		ebs	
4	increment		pad	
8	te		tc	
12	LastUpdateTime			

The meaning of each field is as follows:

cbs – committed burst size in bytes
ebs – excess burst size in bytes
increment – frequency that a burst size of CBS is expected
te – current token bucket size for excess bursts
tc – current token bucket size for committed bursts
lastUpdateTime – last time when traffic was seen on this flow

5.23.11 ATM TM Table

The ATM traffic management table stores the GCRA parameters for each ATM VC. The key is the VC index, which was retrieved from the ATM VC table. Each entry of the table has the following format:

Byte Offset:	0	1	2	3
0	TAT			
4	maxCTD			
8	Increment			
12	limit			
16	Drop behavior			
20	Category		TAT wrap	
24	Reserved			
28	Reserved			

The meaning of each field is as follows:

TAT – theoretical arrival time
maxCTD – maximum cell delay transfer time
Increment – number of credits the VC received per interval
Limit – maximum number of credits the VC may have
Category – whether CBR, rtVBR, nrtVBR, UBR, UBR+ and GFR
TAT wrap – whether TAT has wrapped.

5.23.12 CRC-32 Correction Table

The CRC-32 correction table is used in conjunction with partial CRC-32 calculations during reassembly operations. The table is a data table with 1367 32-bit entries, each of which is a correction factor for a multiple of 48 bytes. The correction factor is XOR'ed with the partial CRC result to compensate for the initial CRC value of all zeros rather than all ones. This table is used exclusively by the TLU.

5.24 Component Buffer Management

The applications each allocate a number of buffer pools. The details of the buffer sizes and how these pools are used is described in Table 3 and **Error! Reference source not found.**. Pools are allocated and initialized by the XP.

Table 3 *wniNicIp Buffer Pool Allocations*

Buffer Pool Owner	Number of Pools	Number of Buffers	Buffer Size (B)	Buffer Use
TDM	1	256	2048	IP datagrams, PPP packets, ML-PPP fragments
TDM	1	256	64	ATM cells
IMA	1	256	64	ATM cells
IMA	1	128	4096	IMA link delay and transmit soft queues
ML-PPP	1	256	256	ML-PPP class soft queues
ML-PPP	1	256	2048	ML-PPP reassembly buffer
PPP-Mux	1	256	2048	PPP-MUX packets, IP datagrams, PPP packets
Ethernet	2	128	2048	IP datagrams
IPHC	2	1024	256	Header compression contexts
Segmentation	1	384	64	ATM cells, CPS packets from AAL-2/5 segmentation
Reassembly	1	384	2048	IP datagrams from AAL-2/5 reassembly
DiffServ	1	2560	1024	DiffServ service class soft queues
ATM TM	1	2560	1024	ATM TM service class soft queues
AAL-2 RX	1	776	64	AAL-2 CPS packets
AAL-2 TX	1	776	64	ATM cells
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
ICMP	1	128	64	ICMP messages

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 are listed in Table 4.

Table 4 *Buffer Formats*

Buffer Type	Buffer Description
BT_UNKNOWN	Contents of the buffer are unknown
BT_IPv4	IPv4 datagram
BT_IPv6	IPv6 datagram
BT_FULL_HEADER	Compressed IP datagram with full header format
BT_COMPRESSED_TCP	Compressed IP datagram with compressed TCP header
BT_COMPRESSED_NON_TCP	Compressed IP datagram with compressed non-TCP header
BT_COMPRESSED_RTP_8	Compressed IP datagram with compressed RTP header and 8-bit CID
BT_COMPRESSED_UDP_8	Compressed IP datagram with compressed UDP header and 8-bit CID
BT_COMPRESSED_TCP-_NODELTA	Compressed IP datagram with uncompressed TCP header
BT_COMPRESSED_RTP_16	Compressed IP datagram with compressed RTP header and 16-bit CID
BT_COMPRESSED_UDP_16	Compressed IP datagram with compressed UDP header and 16-bit CID
BT_CONTEXT_STATE	IP header compression context state
BT_MLPPP	ML-PPP fragment including ML-PPP header, but no PPP or HDLC encapsulation
BT_PPPMUX	multiplexed PPP frame including PPP-Mux header, but no PPP or HDLC encapsulation
BT_LCP	PPP link control protocol frame, including PPP header but no HDLC encapsulation
BT_NCP_IPv4	IPv4 network control protocol frame, including PPP header but no HDLC encapsulation
BT_NCP_IPv6	IPv6 network control protocol frame, including PPP header but no HDLC encapsulation
BT_NCP_PPPMUX	PPP-Mux network control protocol frame, including PPP header but no HDLC encapsulation
BT_PPP	PPP encapsulated data packet; PFC might be in use
BT_HDLC	HDLC encapsulated PPP frame; FCS may not be present; length in descriptor does not include FCS; if ACFC in use, buffer is the same as BT_PPP
BT_MAC	Ethernet encapsulated data packet; CRC may not be present; length in descriptor does not include CRC
BT_PAUSE	Ethernet MAC pause frame
BT_ATM	ATM cell payload; length will be 48
BT_CPS	CPS packet payload; length in descriptor is the LI field of the CPS packet, i.e. it is one less than the actual length of the packet
BT_IMA_CP	IMA control protocol payload; length will be 48
BT_IMA_FILLER	IMA filler cell payload; length will be 48
BT_TDM_TRANSPARENT	Transparent TDM data used with AAL-1
BT_LAST	Place holder to define limit of buffer type enumeration

5.24.1.1 Soft Scheduling Buffers

The IP QoS and ATM TM components use soft scheduling to accomplish their service goals. The scheduling requires the maintenance of more queues than are provided by the QMU. Therefore, each of these components uses a BMU buffer to store descriptors in queue like structures. The queues are nothing more than circular buffers. The head and

tail pointers as well as the offsets and queue lengths for the soft queues are maintained in DMEM by the components. Each line in the buffer is 16 bytes long and has the format of a descriptor. For more details, see section 6.3.

6 Implementation Details

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

6.1 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.

6.2 ATM Traffic Management

This section covers the limitations and some planned future enhancements to the ATM traffic management component.

6.2.1 CLP Bit Marking

As of current design, the ATM TM does not set the CLP bit in ATM header. Non-conformant cells are never transmitted in a manner so as to compromise their traffic contract. However the design does not preclude setting the CLP bit. It might be considered desirable to mark the CLP bit in the following cases:

- for VBR traffic – cells that are transmitted such that they exceed SCR and BT, but are within the PCR.
- for CBR traffic – cells that are transmitted such that they exceed the PCR but are within the BT.

The issue of CLP marking is under discussion and investigation and hence deferred to future releases of the applications.

6.3 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, 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.

6.3.1 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 to simply 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.

6.3.2 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 to complete 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

6.3.3 Issues and enhancements

6.3.3.1 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.

6.3.3.2 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.

6.3.3.3 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.

6.4 AAL-2

The AAL2 Rx and Tx state tables are maintained in DMEM. Given the structure of the tables, one entry is needed for each supported AAL2 VCC. The amount of available DMEM restricts both these table sizes to 256 entries each, implying that the maximum number of VCCs that can be supported is 256. Should it be necessary to support more AAL2 VCCs, these state tables will need to be moved to the TLU controlled SRAM and the DMEM arrays will have to used as local caches. This is not supported in the current release of WNI.

7 HOST PROCESSOR ARCHITECTURE

The WNI host component uses an object oriented (OO) design similar to that used in the gbeSwitch and posOc12Perf host applications in previous CST releases in order to maximize code re-use between the two WNI applications.

The WNI host component 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.

Where 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.

The WNI Host Component is coded such that it does not preclude using it in multi-task environment that would be present when a IP routing/call-processing stack is in use.

The PPP LCP and NCP processing code is derived from the publicly available Linux PPP driver (pppd) version 2.4.1 which supports multi-link PPP and IPv6. Functionality has been added for multi-class PPP (additional LCP options) and PPP-Mux (new NCP).

The host component is available as a single downloadable image (via the console command shell "ld" command) for the wniNicAtm and the wniNicIp applications. It is already linked with the host version of services from the CST.

8 Host Processor Components

This section describes in detail each of the host components used in the applications.

8.1 WniStack

The WniStack class is an abstract base class that provides the common driver initialization, table interface routines, and port interface routines necessary for a host stack to interact with a NP. For the case of the WNI applications, the host stack is simply the NP command shell.

Additionally, the WniStack class provides:

Callback mechanisms for received network traffic
Default packet-oriented receive and transmit routines
MDIO access routines

8.2 WniNicAtmHost and WniNicIpHost

The WniNicAtmHost and WniNicIpHost classes are both derived from the WniStack class. These derivations provide the necessary customization and additional functionality needed. A single instance of the class is instantiated via a "start" command from the NP command shell. This object via its constructor provides for all necessary set-up and initialization for the hardware and software resources needed by the WNI application:

- Creates and opens Pipe structures needed for packet I/O & PPP LCP/NCP.
- Creates and initializes all necessary TLU tables.
- Instantiates all Port objects and maintains Port table.
- Initializes all peripheral hardware (C-Port Family TDM Channel Adapter, C-Port Family UL2 Interface Adapter, backhaul framer) 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, base MAC address).
- Spawns all necessary host tasks (receive, PppMgr, backhaul framer, statistics, APS).
- Releases chip code.

Once all resources are initialized and running, these classes provide the public interface to all NP resources for the purpose of:

- Link/channel configuration changes and associated logical channel/port table management
- Statistics gathering (via Port objects)
- Table updates
- APS commands

These commands all execute in the Console task context. The WniNicAtmHost and WniNicIpHost objects provide an API for cell/packet transmission to a specified port or logical TDM channel.

Received packets or cells destined for the host are delivered via an upcall that was registered with the WniNicAtmHost and WniNicIpHost object. The default handler dumps them to the console

8.3 WniPort Class and Derivations

The abstract WniPort class provides a common interface for getting and setting port parameters and reading statistics to the containing WniStack class while hiding the actual mechanics of getting/deriving the statistics in the derived classes.

WniPort objects are instantiated by the WniNicAtmHost or WniNicIpHost constructor and are accessed via its Port table in order to:

Get/set port parameters (if applicable)
Get port statistics

The Port objects typically access information residing on the NP via the PCI bus with the notable exception being the FastEnetPort which accesses PHY information using the XP's MDIO access routines.

8.3.1 WniAtmPort Class

This class represents the native NP OC-3c ATM port (single CP) used in the wniNicAtm application.

8.3.2 WniFabricPort Class

This class represents the NP's . The has no configurable parameters so the major function of the class is to return the port statistics.

8.3.3 WniTdmPort Class

This class represents the NP port connected to the C-Port Family TDM Channel Adapter. Configuration parameters and statistics for this port are TBD.

8.3.4 WniEnetPort Class

This class represents the native NP 10/100 RMII Ethernet port (single CP) use in the wniNicIp application.

8.4 PppMgr Class

A single instance of this class is instantiated by the WniNicAtmHost or WniNicIpHost constructor. It will run in its own task context. The PppMgr class will be responsible for:

Distributing received PPP LCP and NCP messages to the appropriate PppLink object associated with that TDM channel or ML-Bundle.
Transmitting LCP & NCP messages generated from the PppLink's Fsm objects.
Maintaining the PPP & ML-PPP on-chip information.
Instantiating and deleting PppLink and MLBundle objects in response to configuration changes.

8.5 PppLink Class

PppLink objects are instantiated by the PppMgr class in response to configuration information received either at initialization or from the NP command shell. There will be one PppLink object per logical channel configured for HDLC and one per ML Bundle. These objects will execute in the PppMgr task context. Each PppLink object will contain one Lcp object and multiple NCP objects. The PppLink class also contains a NULL authentication layer.

A PppLink object represents the state of a PPP connection over a logical channel. It will:

Maintain statistics for that connection
Maintain state of the connection (via contained LCP and NCP objects)

8.6 PppMLBundle Class

A PppMLBundle object is instantiated when a PppLink object successfully negotiates the ML-PPP options and no previously existing MLBundle contains the same Endpoint Discriminator, otherwise the PppLink object would be associated with an existing MLBundle. The PppMgr object will maintain a table of PppMLBundle pointers.

A PppMLBundle object represents a single ML-PPP bundle and contains a “virtual” PppLink object for the purpose of having a single set of NCP objects per bundle.

8.7 PppCp Class and Derivations

The PppCp class is an abstract base class from which all the PPP Control Protocol objects will be derived. It contains a nested Fsm object for maintaining state according to RFC 1661. The Fsm callback functions are pure virtual functions within the PppCp class. The public functions correspond to the entries in the ppp-2.4.1 protent structure and are pure virtual. A PppLink reference is maintained for the link that contains this PppCp.

The classes derived from PppCp implement the Fsm callback functions, the public interface functions, and have their own specific negotiation options.

8.7.1 Lcp Class

This class implements the PPP Link Control Protocol per RFC 1661.

8.7.2 Ipcp Class

This class implements the IPv4 Network Control Protocol per RFC 1332.

8.7.3 Ipv6cp Class

This class implements the IPv6 Network Control Protocol per RFC 2472.

8.7.4 PppMuxCp Class

This class implements the PPPmux Network Control Protocol per RFC 3153.

8.7.5 Fsm Class

The Fsm class is a nested class within the PppCp abstract base class. The Fsm class will respond to input messages according to the PPP state machine defined in RFC 1661 generating response messages if required and maintain the current state.

8.8 Interface and Console Command Shell Routines

The interface routines will be used to provide a set of “C” language API’s that are exported to the host environment for the purpose of providing access to the various components of the WNI applications. These routines will primarily utilize the public methods of the WniNicAtmHost and WniNicIpHost classes but may also access external hardware directly if appropriate.

The first user of the public API’s will be the console command shell routines. The dshell tool will be used to generate the console command shell commands.

9 Host Packet I/O

Processes running on the host can send and receive packets/cells via the NP using the services provided in the WniNicAtmHost and WniNicIpHost classes. 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

The host to perform packet input/output uses these resources.

- 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. Upcalls are registered with the WniNic*Host object at initialization time. 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 WniNic*Host's public sendPacket() member 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. The usage model for these commands is a flat hierarchy so that all commands are available at any time and each command has a unique name. The user should refer to the console "help" to get a full listing of arguments for each command.

10.1 Application Control

The following command is used control the application, either wniNicAtm (bs) or wniNicIp (bc) at startup:

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

10.2 Table Maintenance and Display

For each table in the application, the following operations will be supported:

TableEntry`Get {key}` – gets the entry corresponding to provided key
TableEntry`Delete {key}` – deletes the entry corresponding to the provided key
TableEntry`Set {key} {entry info}` – adds/modfies the entry corresponding to the provided key
Table`Display` – displays the entire table
Table`Flush` – 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 Traffic Manager (AtmTm)
ATM Virtual Circuit (AtmVc)
Connection ID (Cid)
Port (port)
IPv4 Routing Table (ipv4Rte)
IPv6 Routing Table (ipv6Rte)
DiffServ Flow (dsflow)
DiffServ Meter (dsmeter)

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".

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. 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.

`linkConfig{linkIndex} {T1|E1} [Clear|Channelized] [ATM|PPP]` – configures a link
`linkRemove {linkIndex}` – removes the link
`linkGet {linkIndex}` – displays the link configuration
`linkDisplay` – displays all configured links

10.4 Channel configuration and Status

These commands configure the individual DS0 channels on links that have been configured for channelized operation for either ATM (default) or PPP operation.

`chanConfig {linkIndex} {channelIndex} {ATM|PPP}` – configure the channel
`chanRemove {linkIndex} {channelIndex}` – removes the channel
`chanGet {linkIndex} {channelIndex}` – displays the channel configuration
`chanDisplay {linkIndex} {channelIndex}` – displays all configured channels on the link
`chanGroupConfig {linkIndex} {channelMask} {ATM|PPP}` – configure a fractional T1/E1 channel group on a link
`chanGroupRemove {linkIndex} {channelMask}` - remove a fractional T1/E1 channel group

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

10.5 Logical TDM Channel Configuration

These commands display information about how the logical TDM channels are configured.

`tdmDisplay` – display all configured TDM channels
`tdmGet {tdmChannel}` – display the specified TDM channel

10.6 IMA Configuration and Status

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

`imaGroupAdd {baseLinkIndex}` – creates the IMA group
`imaLinkAdd {groupIndex} {linkIndex}` {linkId}– adds link to an existing IMA group
`imaLinkRemove {groupIndex} {linkIndex}` – removes link from an existing IMA group
`imaGroupRemove {groupIndex}` – removes the IMA group
`imaGroupDisplay` – shows all configured IMA groups
`imaGroupStart {groupIndex}` – starts the IMA group
`imaGroupStop {groupIndex}` – stops the IMA group
`imaGroupLASR {groupIndex}` – launches a LASR procedure on the IMA group (the links participating in LASR procedure are those in UNUSABLE state)
`imaLinkDisplay {groupIndex}` – shows all configured links for an IMA group
`imaGroupStatsShow {groupIndex}`– shows IMA group statistics
`imaGroupStateShow {groupIndex}`– shows IMA group state
`imaLinkStatsShow {groupIndex} {linkIndex}` – shows IMA link statistics
`imaLinkStateShow {groupIndex} {linkIndex}` – shows IMA link state

10.7 PPP Configuration and Status

These commands allow for the configuration of PPP parameters on clear channel links and individual channels configured for PPP operation. The logical TDM channel that has been assigned when the link or channel is configured is used to select the desired PPP channel.

10.7.1 LCP

These commands allow for the configuration and monitoring of the PPP Link Control Protocol

`pppLcpSet {tdmChannel} {LcpParameter} {LcpParameterValue}` – configures the LCP parameter on the specified channel

`pppLcpGet {tdmChannel} {LcpParameter}` – displays the LCP parameter on the specified channel

The following LCP parameters are supported:
- negotiate MRU
- negotiate Multilink PPP
- negotiate Multiclass
- MRU value
- MRRU vlaue
- Number of classes
- Address field compression
- Protocol field compression
- ML Short Sequence Numbers

`pppLcpOpen {tdmChannel}` – opens LCP on the specified channel
`pppLcpClose {tdmChannel}` – closes LCP on the specified channel
`pppLcpStats {tdmChannel}` – displays LCP statistics from the specified channel

10.7.2 IPCP

These commands allow for the configuration and monitoring of the PPP IPv4 Network Control Protocol (IPCP)

`pppIpcpSet {tdmChannel} {IpcpParameter} {IpcpParameterValue}` – configures the IPCP parameter on the specified channel
`pppIpcpGet {tdmChannel} {IpcpParameter}` – displays the IPCP parameter on the specified channel

The following IPCP parameters are supported:
- negotiate address
- negotiate IP header compression
- local address
- remote address
- compression protocol
- compression TCP space
- compression non-TCP space
- compression fMaxPeriod
- compression fMaxTime
- compression maxHeader
- compression rtpComp

`pppIpcpOpen {tdmChannel}` – opens IPCP on the specified channel
`pppIpcpClose {tdmChannel}` – closes IPCP on the specified channel
`pppIpcpStats {tdmChannel}` – displays IPCP statistics from the specified channel

10.7.3 IPv6CP

These commands allow for the configuration and monitoring of the PPP Ipv6 Network Control Protocol (Ipv6CP)

`pppIpv6cpSet {tdmChannel} {Ipv6cpParameter} {Ipv6cpParameterValue}` – configures the IPV6CP parameter on the specified channel
`pppIpv6cpGet {tdmChannel} {Ipv6cpParameter}` – displays the Ipv6CP parameter on the specified channel

The following Ipv6CP parameters are supported:

- negotiate ID
- negotiate IP header compression
- local ID 0, 1
- remote ID 0, 1
- compression protocol
- compression TCP space
- compression non-TCP space
- compression fMaxPeriod
- compression fMaxTime
- compression maxHeader
- compression rtpComp

`pppIpv6cpOpen {tdmChannel}` – opens IPV6CP on the specified channel
`pppIpv6cpClose {tdmChannel}` – closes IPV6CP on the specified channel
`pppIpv6cpStats {tdmChannel}` – displays IPV6CP statistics from the specified channel

10.7.4 PPPMuxCP

These commands allow for the configuration and monitoring of the PPP Mux Control Protocol (PPPMuxCP)

`pppMuxcpSet {tdmChannel} {PPPMuxcpParameter} {PPPMuxcpParameterValue}` – configures the PPPMuxCP parameter on the specified channel
`pppMuxcpGet {tdmChannel} {PPPMuxcpParameter}` – displays the PPPMuxCP parameter on the specified channel

The following PPPMuxCP parameters are supported:
- negotiate default PID
- default PID value

`pppMuxcpOpen {tdmChannel}` – opens PPPMuxCP on the specified channel
`pppMuxcpClose {tdmChannel}` – closes PPPMuxCP on the specified channel
`pppMuxcpStats {tdmChannel}` – displays PPPMuxCp statistics from the specified channel

10.7.5 Link Status

The following command displays the status of PPP LCP/NCP negotiation for the PPP link on the specified TDM channel:

- `pppLinkStatusGet {tdmChannel}`

10.7.6 Bundle Status

The following command displays the status of PPP NCP negotiation for the PPP Multilink bundle on the specified TDM channel:

- `pppBundleStatusGet {tdmChannel}`

10.8 AAL5

The following commands allow the configuration and monitoring of the AAL5 component.

`aal5MaxSduSizeSet` – Set the maximum AAL5 SDU size
`aal5MaxSduSizeGet` - Get the maximum AAL5 SDU size

`aal5StatsShow` – Display AAL5 protocol statistics
`aal5RxFreeze` – Freeze the AAL5 Rx CP
`aal5RxUnfreeze` – Unfreeze the AAL5 Rx CP
`aal5TxFreeze` – Freeze the AAL5 Tx CP
`aal5TxUnfreeze` – Unfreeze the AAL5 Tx CP

10.9 DiffServ

The following commands allow the configuration and monitoring of the DiffServ component.

`diffServQuantumSet {egressPortIndex} {trafficCategory} {quantum}` – Set a new quantum for DiffServ's egress WRR queueing discipline
`diffServQuantumGet {egressPortIndex} {trafficCategory}` – Get a the current WRR quantum for the Diffserv egress queue
`diffServWREDParamsGet` – Display the weighted Random Early Detect drop algorithm parameters
`diffServTrackStats {portIndex}` – Start accumulating statistics for the port
`diffServStatsShow` – Retrieve and display DiffServ statistics

10.10 AAL2

The following commands allow the configuration and monitoring of the AAL2 component.

`aal2CUTimerSet {timerValue}`– Set the CU timer interval (in # of NP cycles)
`aal2CUTimerGet` – Display the CU timer interval
`aal2StatsShow` – Display AAL2 level statistics

10.11 ATM Traffic Manager

The following commands allow the configuration of the ATM Traffic manager component.

`atmTmStatsTrack {portIndex}` – Start tracking ATM Traffic Manager statics for desired port
`atmTmStatsShow` – Display ATM TM statistics

10.12 SSSAR

The following command allows monitoring of the SSAR component.

`sssarStatsShow` – Display SSSAR statistics

10.13 TDM Port Configuration and Status

The following commands allow the configuartion and monitoring of the TDM port.

`tdmConfig {framerId} {quadrantId}` – configure TDM clocking
`tdmIpAddrSet {tdmIpV4Addr}` – set the IPv4 Address assigned to all TDM channels
`tdmIpAddrGet` – display the IPv4 Address assigned to all TDM
`tdmStatsShow {tdmChannel}` – display the statistics for the specified TDM channel

10.14 Ethernet Port Configuration and Status

These commands allow for the configuration and monitoring of the 2 10/100 Ethernet ports (wniNicIp only):

`enetMacAddrGet {enetPortIndex}` – get Ethernet Port Mac address:

`enetIpAddrGet {enetPortIndex}` – get Ethernet Port IP address
`enetIpAddrSet {enetPortIndex} {Ipv4Address}` – set Ethernet Port IPv4
`enetPauseModeGet {enetPortIndex}` – get Ethernet Port MAC Pause Mode
`enetPortStateGet {enetPortIndex}` – get Ethernet Port state
`enetSpeedDuplexGet {enetPortIndex}` – get current Ethernet Speed/Duplex mode
`enetAnCapsGet {enetPortIndex}` – get Ethernet Port Auto-Negotiation mode
`enetRemoteAnCapsGet {enetPortIndex}` – get Ethernet Port link parter Auto-Negotiation mode
`enetAnStatusGet {enetPortIndex}` – get Ethernet Port Auto-Negotiation status
`enetLinkStatusGet {enetPortIndex}` – get Ethernet Port current link status
`enetSpeedDuplexSet {enetPortIndex}` – set Ethernet Port speed/duplex (disables Auto-negotiation)
`enetAnCapsSet {enetPortIndex}` – set Ethernet Port Auto-Negotiation advertisement
`enetAnEnable {enetPortIndex}` – Enable/Disable Ethernet Auto-Negotiation
`enetAnRestart {enetPortIndex}` – restart Auto-Negotiation on the Ethernet Port
`enetStatsShow {enetPortIndex}` – show Ethernet Port statistics (MAC and IP)

10.15 ATM Configuration and Status

These commands allow for the configuration and monitoring of the ATM-related parameters.

`atmPortFreeze {atmPortIndex}` – Freeze the specified ATM port
`atmPortUnfreeze {atmPortIndex}` – Unfreeze the specified ATM
`atmIfTypeGet {atmPortIndex}` – Get the interface type for the specified port
`atmScrambleModeSet {atmPortIndex}` – Set scramble mode for the specified port
`atmScrambleModeGet {atmPortIndex}` – Get scramble mode settings for the specified port
`atmLoopbackModeSet {atmPortIndex}` – Put the specified port in local loopback mode
`atmLoopbackModeGet {atmPortIndex}` – Get the loopback mode for the specified port
`atmL1StateGet {atmPortIndex}` – Get the current ATM state on the sepcified port
`atmStatsShow {atmPortIndex}` – Display ATM level stats for the specified

10.16 Framer Configuration and Status

These commands allow for the configuration and monitoring of the framer.

framerStatusGet {framerIndex} – display the framer status

10.17 Mt-4 Configuration and Status

These commands allow for the configuration and monitoring of the Mt-4.

`mt4ChannelStatsShow {tdmChannel}` – display the Mt-4 channel statistics
`mt4GlobalStatsShow` – display the Mt-4 global
`fpLoad {fileName}` – load the Mt-4 FPGA contents

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 Link Parameters

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

Byte Offset:	0	1	2	3
0	flags	segSize	mru	
4	defaultPid		pad	

flags: a bitmap as follows:
- b7: port valid
- b6: IPv4 NCP up
- b5: IPv6 NCP up
- b4: PPP-Mux 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
defaultPid: PPP-Mux default protocol ID for the link
pad: unused

11.2 ML/MC-PPP

The following data structures relate to the ML/MC-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	numClasses	mru	
4	firstPort	numPorts	pad	

flags: a bitmap as follows:
- b7: bundle valid
- b6: IPv4 NCP up
- b5: IPv6 NCP up
- b4: PPP-Mux NCP up
- b3: short sequence number
- b2-b0: unused

numClasses: number of MC-PPP classes

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. The size and location of the block is defined by firstPort and numPort fields of the ML bundle parameter structure.

11.3 ATM

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

11.3.1 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 indicating whether or not PM is to be performed on the channel.

11.4 Ethernet

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

11.4.1 MAC Address

This MAC address is provided to the XP via the appData parameters in the shared HCA structure. The XP passes this address to the CP in the initialization descriptor. The CP uses this as the MAC source address for all traffic leaving the Ethernet port.

11.4.2 IP Address

This IP address is provided to the XP via the appData parameters in the shared HCA 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.

11.5 IP Header Compression

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

11.5.1 IPHC Configuration

This structure holds configuration parameters for the IP header compression engine that have been negotiated by the IPv4 or IPv6 NCP.

Byte Offset:	0	1	2	3
0	minWrap		maxFullHeaderTime	
4	maxFullHeaderPeriod		maxHeader	
8	maxNonTcpSpace		pad	

minWrap: minimum number of clock ticks that must pass before reusing a compression ID
maxFullHeaderTime: maximum number of clock ticks that may pass before a full header is sent again
maxFullHeaderPeriod: maximum number of packets that may be sent before a full header is sent again
maxHeader: the maximum number of bytes in a compressible header
maxNonTcpSpace: the maximum value of a compression ID for non-TCP flows
pad: unused

12 HOST API REFERENCE

12.1 Table API

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

12.1.1 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 5.23.1.

12.1.1.1 Data Types

12.1.1.1.1 PortTableInfo

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 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)

12.1.1.2 Functions

12.1.1.2.1 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)

12.1.1.2.2 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.

12.1.1.2.3 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.

12.1.1.2.4 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

12.1.1.2.5 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

12.1.2 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 5.23.3.

12.1.2.1 Data Types

12.1.2.1.1 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 int8u cpsPduLen – AAL2 CPS PDU Length int8u cid – AAL2 CID

12.1.2.1.2 MacFwd

Description:	This structure contains fields that contain the necessary information to correctly forward a packet or cell to an Ethernet port
Type:	struct
Usage:	The definition for the fields of this structure are as follows: int8u macDa[6] – Ethernet MAC layer Destination Address

12.1.2.1.3 TdmFwd

Description:	This structure contains fields that contain the necessary information to correctly forward a packet or cell to a TDM port
Type:	struct
Usage:	The definition for the fields of this structure are as follows: int32u compressionId – IP Header Commpression ID int8u mcClass – Multiclass PPP class

12.1.2.1.4 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

12.1.2.1.5 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.
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 udpDestPort – UDP Destination Port(key) union { int32u word[2] – provide word access int16u hword[4] – provide half-word access int8u byte[8] – provide byte access MacFwd macFwd – Ethernet Data Link forwarding information AtmFwd atmFwd – ATM Port forwarding information TdmFwd tdmFwd – TDM Port forwarding nformation } appData; int32u queueId – Destination queue ID int8u fabricId – Destination fabric ID int8u type – route type

12.1.2.2 Functions

12.1.2.2.1 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)

12.1.2.2.2 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 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.

12.1.2.2.3 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.

12.1.2.2.4 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

12.1.2.2.5 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

12.1.3 IPv6 Route Table API

This API allows an application to maintain the IPv6 Route Table. This table stores forwarding information for IPv6 and UDP datagrams. For more information, see 5.23.4.

12.1.3.1 Data Types

12.1.3.1.1 HalfIpv6Addr

Description:	This structure contains the representation of half (8 bytes) of an IPv6 Network Address.
Type:	union
Usage:	The definition for the fields of this union are as follows: int8u byte[8] – allow byte access int32u word[2] – allow word access int16u hword[4] – allow halfword access

12.1.3.1.2 IpV6RouteInfo

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 keyTag – tag field to be used if another lookup is required (key) HalfIpv6Addr halfAddress – the IPv6 Network layer half address (key) int16u udpDestPort – the UDP destination port (key) int16u port – destination port Index int8u type – route type int8u maskBits – number of bits in the IPv6 Network mask union { int32u word[2] – allow word access int16u hword[4] – allow half-word access int8u byte[8] – allow byte access MacFwd macFwd – Ethernet Data Link port forwarding information AtmFwd atmFwd – ATM port forwarding information TdmFwd tdmFwd – TDM port forwarding information } appData int16u tag – tag field for next lookup if required int16u numLinks – use count for this IPv6 half address

12.1.3.2 Functions

12.1.3.2.1 getIpV6RouteTable

Function:	int getIpV6RouteTable(int npIndex, IpV6RouteInfo* info)
Description:	This function attempts to get the entry data in the IPV6 Route 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)

12.1.3.2.2 getNextIpV6RouteTable

Function:	int getNextIpV6RouteTable(int npIndex, IpV6RouteInfo* info)
Description:	This function is used to "walk" the IPV6 Route 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.

12.1.3.2.3 setIpV6RouteTable

Function:	int setIpV6RouteTable(int npIndex, IpV6RouteInfo* info)
Description:	This function is used to add or modify an entry in the IPV6 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.

12.1.3.2.4 deleteIpV6RouteTable

Function:	int deleteIpV6RouteTable(int npIndex, IpV6RouteInfo* info)
Description:	This function removes an entry from the IPV6 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

12.1.3.2.5 flushIpV6RouteTable

Function:	int flushIpV6RouteTable(int npIndex, int& flushCount)
Description:	This function removes all entries from the IPV6 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

12.1.4 ATM VC Table API

This API allows an application to maintain the ATM VC Table. This table stores connection information for ATM virtual circuits. For more information, see 5.23.5.

12.1.4.1 Data Types

12.1.4.1.1 AtmVcInfo

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) int32u egressCellHeader – egress port cell header, bit 3:0 are the traffic class int16u vcIndex – VC index int16u flags_egressPort – flag and egress port, bits 15:11 are flags, 10:0 is the port int16u egressQueue – egress queue int16u port_bufType – port and buffer type int16u destQueue – final destination queue int8u oamPm – OAM PM

12.1.4.2 Functions

12.1.4.2.1 getAtmVcTable

Function:	int getAtmVcTable(int npIndex, AtmVcInfo* info)
Description:	This function attempts to get the entry data in the ATM VC 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)

12.1.4.2.2 getNextAtmVcTable

Function:	int getNextAtmVcTable(int npIndex, AtmVcInfo* info)
Description:	This function is used to "walk" the ATM VC 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.

12.1.4.2.3 *setAtmVcTable*

Function:	int setAtmVcTable(int npIndex, AtmVcInfo* 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.

12.1.4.2.4 *deleteAtmVcTable*

Function:	int deleteAtmVcTable(int npIndex, AtmVcInfo* 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

12.1.4.2.5 *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

12.1.5 CID Table API

This API allows an application to maintain the AAL-2 CID Table. This table stores forwarding and connection information for AAL-2 channels. For more information, see 5.23.6 above.

12.1.5.1 Data Types

12.1.5.1.1 CidInfo

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: int8u vcIndexHi – VC index high byte (key) int8u vcIndexLo – VC index low byte (key) int8u cid – AAL2 CID (key) int8u flags – flags indicating SSSAR or CID switching required int8u destVpi – destination VPI int16u sssarCallid – SSSAR Caller ID int16u destVci – destination VCI int16u destPort – destination port int16u destVcIndex – destination VC index int8u destCid – destination CID int8u destUui – destination Uui

12.1.5.2 Functions

12.1.5.2.1 getCidTable

Function:	int getCidTable(int npIndex, CidInfo* info)
Description:	This function attempts to get the entry data in the CID 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)

12.1.5.2.2 getNextCidTable

Function:	int getNextCidTable(int npIndex, CidInfo* info)
Description:	This function is used to "walk" the CID 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.

12.1.5.2.3 setCidTable

Function:	int setCidTable(int npIndex, CidInfo* info)
Description:	This function is used to add or modify an entry in the CID 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.

12.1.5.2.4 deleteCidTable

Function:	int deleteCidTable(int npIndex, CidInfo* info)
Description:	This function removes an entry from the CID 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

12.1.5.2.5 flushCidTable

Function:	int flushCidTable(int npIndex, int& flushCount)
Description:	This function removes all entries from the CID 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

12.1.6 ATM TM Table API

This API allows an application to maintain the ATM TM Table. This table stores traffic management parameters for the GCRA algorithm for ATM virtual circuits. For more information, see 5.23.11.

12.1.6.1 Data Types

12.1.6.1.1 AtmTmInfo

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 vcIndex – VC index (key) int32u TAT – theoretical arrival time int132u maxCTD – maximum cell transfer delay int32u Increment – credits to give to VC per time internal int32u Limit – maximum credits a VC may have int32u dropFlag – drop flag int32u category – Traffic category i.e., one of CBR, rtVBR, nrtVBR, UBR+ and GFR.

12.1.6.2 Functions

12.1.6.2.1 getAtmTmTable

Function:	int getAtmTmTable(int npIndex, AtmTmInfo* info)
Description:	This function attempts to get the entry data in the ATM TM 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)

12.1.6.2.2 getNextAtmTmTable

Function:	int getNextAtmTmTable(int npIndex, AtmTmInfo* info)
Description:	This function is used to "walk" the ATM TM 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.

12.1.6.2.3 setAtmTmTable

Function:	int setAtmTmTable(int npIndex, AtmTmInfo* info)
Description:	This function is used to add or modify an entry in the ATM TM 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.

12.1.6.2.4 deleteAtmTmTable

Function:	int deleteAtmTmTable(int npIndex, AtmTmInfo* info)
Description:	This function removes an entry from the ATM TM 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

12.1.6.2.5 flushAtmTmTable

Function:	int flushAtmTmTable(int npIndex, int& flushCount)
Description:	This function removes all entries from the ATM TM 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

12.1.7 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 5.23.9.

12.1.7.1 Data Types

12.1.7.1.1 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

12.1.7.2 Functions

12.1.7.2.1 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)

12.1.7.2.2 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.

12.1.7.2.3 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.

12.1.7.2.4 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

12.1.7.2.5 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

12.1.8 Diffserv Meter Table API

This API allows an application to maintain the DiffServ Meter Table. This table stores QoS service parameters for IPv4 flows. For more information, see 5.23.10.

12.1.8.1 Data Types

12.1.8.1.1 DiffServMeterInfo

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 flowId – flow ID (key) int16u cbs – committed burst size in bytes int16u ebs – excess burst size in bytes int16u increment – frequency that a burst size of CBS is expected int16u te – current token bucket size for excess bursts int16u tc – current token bucket size for committed bursts int32u lastUpdateTime – last time when traffic was seen on this flow

12.1.8.2 Functions

12.1.8.2.1 getDiffServMeterTable

Function:	int getDiffServMeterTable(int npIndex, DiffServMeterInfo* info)
Description:	This function attempts to get the entry data in the DiffServ Meter 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)

12.1.8.2.2 getNextDiffServMeterTable

Function:	int getNextDiffServMeterTable(int npIndex, DiffServMeterInfo* info)
Description:	This function is used to "walk" the DiffServ Meter 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.

12.1.8.2.3 setDiffServMeterTable

Function:	int setDiffServMeterTable(int npIndex, DiffServMeterInfo* info)
Description:	This function is used to add or modify an entry in the DiffServ Meter 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.

12.1.8.2.4 deleteDiffServMeterTable

Function:	int deleteDiffServMeterTable(int npIndex, DiffServMeterInfo* info)
Description:	This function removes an entry from the DiffServ Meter 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

12.1.8.2.5 flushDiffServMeterTable

Function:	int flushDiffServMeterTable(int npIndex, int& flushCount)
Description:	This function removes all entries from the DiffServ Meter 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

12.2 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.

12.2.1 Data Types

12.2.1.1 WniLinkType

Description:	This structure contains an enumeration of the supported link types.
Type:	enum
Usage:	The definition for the fields of this structure are as follows: WNI_LINK_NULL – null type WNI_LINK_T1 – T1 link WNI_LINK_E1 – E1 link WNI_LINK_LAST – last value

12.2.1.2 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 or bit mask of channels when used with channel group (fractional T1/E1) functions int8u **ppp** – if “0” configured for ATM, if non-zero configured for PPP int8u **clearChannel** – if non-zero, configured for Clear Channel int8u **qos** – QOS flag int8u **comp** – compression flag int **activeChannelCount** – number of active sub-channels WniLinkType **linkType** – link type int **logChannel** – logical channel index (used to return the allocated logical channel when a link or channel is created)

12.2.2 Functions

12.2.2.1 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

12.2.2.2 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.

12.2.2.3 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

12.2.2.4 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.

12.2.2.5 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.

12.2.2.6 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

12.2.2.7 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

12.2.2.8 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)

12.2.2.9 addChannelGroup

Function:	int addChannelGroup(int npIndex, LinkChannelInfo* info)
Description:	This function is used to add a group of sub-channels (all corresponding to the same logical TDM channel) to a previously configured link
Parameters:	npIndex – index of the NP for the operation info – structure to be used for specifying the link and channel mask
Returns:	0 – operation was successful 1 – operation was not successful (sub-channels already in use)

12.2.2.10 removeChannelGroup

Function:	int removeChannelGroup(int npIndex, LinkChannelInfo* info)
Description:	This function is used to remove an existing sub-channel group 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 (sub-channel mask not correct)

12.2.2.11 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)

12.2.2.12 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)

12.2.2.13 addImaGroup

Function:	int addImaGroup(int npIndex, int imaGroupIndex, int numLinks, int frameLen)
Description:	This function is used to create a new IMA Group
Parameters:	• npIndex – index of the NP for the operation imaGroupIndex – returned value of new IMA Group index if successful numLinks – number of links frameLen – frame length
Returns:	0 – operation was successful 1 – operation was not successful (base link not configured correctly)

12.2.2.14 addImaLink

Function:	int addImaLink(int npIndex, int imaGroupIndex, int linkIndex, int linkId)
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 linkId – IMA LID assigned to the link
Returns:	0 – operation was successful 1 – operation was not successful (link not configured correctly)

12.2.2.15 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)

12.2.2.16 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)

12.2.2.17 startImaGroup

Function:	int startImaGroup(int npIndex, int imaGroupIndex)
Description:	This function is used to start an existing IMA Group
Parameters:	npIndex – index of the NP for the operation imaGroupIndex – existing IMA group to be started
Returns:	0 – operation was successful 1 – operation was not successful

12.2.2.18 stopImaGroup

Function:	int stopImaGroup(int npIndex, int imaGroupIndex)
Description:	This function is used to stop an existing IMA Group
Parameters:	npIndex – index of the NP for the operation imaGroupIndex – existing IMA group to be stopped
Returns:	0 – operation was successful 1 – operation was not successful

12.2.2.19 lasrImaGroup

Function:	int stopImaGroup(int npIndex, int imaGroupIndex)
Description:	This function is used to stop an existing IMA Group
Parameters:	npIndex – index of the NP for the operation imaGroupIndex – existing IMA group to be stopped
Returns:	0 – operation was successful 1 – operation was not successful

12.2.2.20 getActiveImaGroups

Function:	int getActiveImaGroups(int npIndex, int32u& groupMask)
Description:	This function is used to determine which IMA groups are currently active
Parameters:	npIndex – index of the NP for the operation groupMask – used to return a bit mask of active IMA groups
Returns:	0 – operation was successful 1 – operation was not successful

12.2.2.21 getActiveImaLinks

Function:	int getActiveImaLinks(int npIndex, int imaGroupIndex, int32u& linkMask)
Description:	This function is used to determine the active links in a IMA group
Parameters:	npIndex – index of the NP for the operation imaGroupIndex – existing IMA group to be removed linkMask – used to return a bit mask of active links
Returns:	0 – operation was successful 1 – operation was not successful (not all links have been removed first)

12.2.2.22 getImaGroupStats

Function:	int getImaGroupStats(int npIndex, void* stats, time_t time, int imaGroupIndex)
Description:	This function is used to retrieve IMA Group statistics
Parameters:	npIndex – index of the NP for the operation stats – buffer used to return the statistics tine – current time imaGroupIndex – existing IMA group to get statistics from
Returns:	0 – operation was successful 1 – operation was not successful

12.2.2.23 getImaGroupState

Function:	int getImaGroupState(int npIndex, void* state, int imaGroupIndex)
Description:	This function is used to retrieve IMA Group state
Parameters:	npIndex – index of the NP for the operation state – buffer used to return state information imaGroupIndex – existing IMA group to get state from
Returns:	0 – operation was successful 1 – operation was not successful

12.2.2.24 getImaLinkStats

Function:	int getImaLinkStats(int npIndex, void* stats, time_t* time, int imaGroupIndex, int linkIndex)
Description:	This function is used to retrieve IMA link statistics.
Parameters:	npIndex – index of the NP for the operation stats – buffer used to return statistics time – current time imaGroupIndex – existing IMA group containing the link of interest linkIndex – link to get statistics from
Returns:	0 – operation was successful 1 – operation was not successful

12.2.2.25 getImaLinkState

Function:	int getImaLinkState(int npIndex, void* state, int imaGroupIndex, int linkIndex)
Description:	This function is used to retrieve IMA link state
Parameters:	npIndex – index of the NP for the operation state – buffer used to return state information imaGroupIndex – IMA group containing the link of interest linkIndex – link to get state from
Returns:	0 – operation was successful 1 – operation was not successful

12.2.2.26 registerImaEventCallBack

Function:	int registerImaEventCallBack(int npIndex, void (*imaCallBack)(int group, int event))
Description:	This function is used to register the IMA event callback function
Parameters:	npIndex – index of the NP for the operation imaCallBack – IMA event callback function
Returns:	0 – operation was successful

1 – operation was not successful

12.2.2.27 unregisterImaEventCallBack

Function:	int unregisterImaEventCallBack(int npIndex)
Description:	This function is used to unregister the IMA Event callback function
Parameters:	• npIndex – index of the NP for the operation
Returns:	0 – operation was successful 1 – operation was not successful

12.3 PPP API

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

12.3.1 Data Types

12.3.1.1 WniPppParam

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_OPT_MC – LCP Multiclass Option • LCP_OPT_PFX – LCP Prefix Elision Option • LCP_VAL_MC – LCP Multiclass, number of supported classes • 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 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 • IPCP_SLOTID_COMP – IPCP VJ Slot ID • IPCP_VAL_TCPSPACE – IPHC TCP-Space value • IPCP_VAL_NONTCPSPACE – IPHC non-TCP space value • IPCP_VAL_FMAXPERIOD – IPHC fMaxPeriod value • IPCP_VAL_FMAXTIME – IPHC fMaxTime value • IPCP_VAL_MAXHEADER – IPHC maxHeader value • IPCP_VAL_RTPCOMP – IPHC RTP Compression value • IPCP_VAL_OURADDR – IPCP local IP Address value • IPCP_VAL_REMADDR – IPCP remote IP Address value • IPCP_VAL_DNS1 – IPCP Primary DNS value • IPCP_VAL_DNS2 – IPCP Secondary DNS value • IPCP_VAL_WINS1 – IPCP Primary WINS value

- IPCP_VAL_WINS2 – IPCP Secondary WINS value
- IPV6CP_OPT_IFACEID – IPV6CP Negotiate interface identifier option
- IPV6CP_OPT_VJ – IPV6CP Van Jacobson Compression option
- IPV6CP_REQ_IFACEID – IPV6CP Ask peer to send interface identifier
- IPV6CP_ACCEPT_LOCAL – IPV6CP accept peer's value for iface id
- IPV6CP_LOCAL_OPT – IPV6CP our token set by option
- IPV6CP_REMOTE_OPT – IPV6CP his token set by option
- IPV6CP_USE_IP – IPV6CP use IP as interface identifier
- IPV6CP_COMP_PROT – IPV6CP Compression Protocol value
- IPV6CP_VAL_TCPSPACE – IPHC TCP-Space value
- IPV6CP_VAL_NONTCPSPACE – IPHC non-TCP space value
- IPV6CP_VAL_FMAXPERIOD – IPHC fMaxPeriod value
- IPV6CP_VAL_FMAXTIME – IPHC fMaxTime value
- IPV6CP_VAL_MAXHEADER – IPHC maxHeader value
- IPV6CP_VAL_RTPCOMP – IPHC RTP Compression value
- PPPMUXCP_OPT_DEFPID – PPPMuxCp Default Protocol ID option
- PPPMUXCP_VAL_DEFPID – PPPMuxCp Default Protocol ID value

12.3.2 LCP API

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

12.3.2.1 getPppLcpParam

Function:	int getPppLcpParam(int npIndex, int tdmChannel, WniPppParam 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

12.3.2.2 setPppLcpParam

Function:	int setPppLcpParam(int npIndex, int tdmChannel, WniPppParam 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

12.3.2.3 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

12.3.2.4 openPppLcp

Function:	int openPppLcp(int npIndex, int tdmChannel)
Description:	This function is used to open the LCP protocol on the specified PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link
Returns:	0 – operation was successful 1 – operation was not successful

12.3.2.5 closePppLcp

Function:	int closePppLcp(int npIndex, int tdmChannel)
Description:	This function is used to close the LCP protocol on the specified PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link
Returns:	0 – operation was successful 1 – operation was not successful

12.3.3 IPCP API

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

12.3.3.1 getPppIpCpParam

Function:	int getPppIpCpParam(int npIndex, int tdmChannel, WniPppParam 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

12.3.3.2 setPppIpCpParam

Function:	int setPppIpCpParam(int npIndex, int tdmChannel, WniPppParam 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

12.3.3.3 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)

12.3.3.4 openPppIpCp

Function:	int openPppIpCp(int npIndex, int tdmChannel)
Description:	This function is used to open the IPCP protocol on the specified PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link
Returns:	0 – operation was successful 1 – operation was not successful

12.3.3.5 closePppIpCp

Function:	int closePppIpCp(int npIndex, int tdmChannel)
Description:	This function is used to close the IPCP protocol on the specified PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link
Returns:	0 – operation was successful 1 – operation was not successful

12.3.4 IPv6CP

This API allows an application to configure the IPv6 CP parameters of a PPP link.

12.3.4.1 getPppIpV6CpParam

Function:	int getPppIpV6CpParam(int npIndex, int tdmChannel, WniPppParam param, int32u* value)
Description:	This function is used to get the specified IPV6CP 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

12.3.4.2 setPppIpV6CpParam

Function:	int setPppIpV6CpParam(int npIndex, int tdmChannel, WniPppParam param, int32u* value)
Description:	This function is used to set the specified IPV6CP 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

12.3.4.3 getPppIpV6CpStats

Function:	int getPppIpV6CpStats(int npIndex, int tdmChannel, void* statBuffer)
Description:	This function is used to get the IPV6CP 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

12.3.4.4 openPppIpV6Cp

Function:	int openPppIpV6Cp(int npIndex, int tdmChannel)
Description:	This function is used to open the IPV6CP protocol on the specified PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link
Returns:	0 – operation was successful 1 – operation was not successful

12.3.4.5 closePppIpV6Cp

Function:	int closePppIpV6Cp(int npIndex, int tdmChannel)
Description:	This function is used to close the IPV6CP protocol on the specified PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link
Returns:	0 – operation was successful 1 – operation was not successful

12.3.5 PPP-MUX CP

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

12.3.5.1 getPppMuxCpParam

Function:	int getPppMuxCpParam(int npIndex, int tdmChannel, WniPppParam param, int32u* value)
Description:	This function is used to get the specified PPPMuxCP 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 (no more active channels)

12.3.5.2 setPppMuxCpParam

Function:	int setPppMuxCpParam(int npIndex, int tdmChannel, WniPppParam param, int32u* value)
Description:	This function is used to set the specified PPPMuxCP 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

12.3.5.3 getPppMuxCpStats

Function:	int getPppMuxCpStats(int npIndex, int tdmChannel, void* statBuffer)
Description:	This function is used to get the PPPMuxCP 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

12.3.5.4 openPppMuxCp

Function:	int openPppMuxCp(int npIndex, int tdmChannel)
Description:	This function is used to open the PPPMuxCP protocol on the specified PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link
Returns:	0 – operation was successful 1 – operation was not successful

12.3.5.5 closePppMux6Cp

Function:	int closePppMuxCp(int npIndex, int tdmChannel)
Description:	This function is used to close the PPPMuxCP protocol on the specified PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link
Returns:	0 – operation was successful 1 – operation was not successful

12.3.6 Link Status

This API allows an application to determine the status of the LCP and NCP negotiations on a given PPP link

12.3.6.1 getPppLinkStatus

Function:	int getPppLinkStatus(int npIndex, int tdmChannel, int* lcpStatus, int* ipcpStatus, int* ipv6cpStatus, int* pppMuxCpStatus)
Description:	This function is used to get the status of the LCP and NCP negotiations on a PPP link
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired PPP link lcpStatus – returned LCP status (1 = up, 0 = down) ipcpStatus – returned IPCP status ipv6cpStatus – returned Ipv6CP status pppMuxCpStatus – returned pppMuxCp status
Returns:	0 – operation was successful 1 – operation was not successful

12.3.7 Bundle Status

This API allows an application to determine the status of the NCP negotiations on a given ML-PPP bundle.

12.3.7.1 getPppBundleStatus

Function:	int getPppBundleStatus(int npIndex, int tdmChannel, , int* ipcpStatus, int* ipv6cpStatus, int* pppMuxCpStatus)
Description:	This function is used to get the status of the NCP negotiations on a ML-PPP bundle
Parameters:	npIndex – index of the NP for the operation tdmChannel – logical TDM channel of the desired ML-PPP bundle ipcpStatus – returned IPCP status (1 = up, 0 = down) ipv6cpStatus – returned Ipv6CP status pppMuxCpStatus – returned pppMuxCp status
Returns:	0 – operation was successful 1 – operation was not successful

12.4 Control API

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

12.4.1 Functions

12.4.1.1 startWniNicIp

Function:	int startWniNicIp(int32u npIndex, int8u fabricId, int32u macAddrHi32, int16u macAddrLo16, int(*rxIP)(int,int32u,void*,int16u,int8u*), int(*rxPPP)(int,int32u,void*,int16u,int8u*), int(*rxATM)(int,int32u,void*,int16u,int8u*), void(*imaCb)(int, int), int defaultConfig))
Description:	This function is used to start the WniNicIp application
Parameters:	npIndex – index of the NP for the operation fabricId – fabricId macAddrHi32 – high order 32 bits of base Ethernet MAC address (also used for PPP Endpoint disciminator) macAddrLo16 – low order 16 bits of base Ethernet MAC address (also used for PPP Endpoint discriminator) rxIP – IP upcall function pointer rxPPP – PPP upcall function pointer rxATM – ATM upcalll function pointer imaCb – IMA event callback 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.

12.4.1.2 startWniNicAtm

Function:	int startWniNicAtm(int32u npIndex, int8u fabricId, int32u macAddrHi32, int16u macAddrLo16, int(*rxIP)(int,int32u,void*,int16u,int8u*), int(*rxPPP)(int,int32u,void*,int16u,int8u*), int(*rxATM)(int,int32u,void*,int16u,int8u*), void(*imaCb)(int, int), int defaultConfig))
Description:	This function is used to start the wniNicAtm application
Parameters:	npIndex – index of the NP for the operation fabricId – fabricId macAddrHi32 – high order 32 bits of base Ethernet MAC address (also used for PPP Endpoint disciminator) macAddrLo16 – low order 16 bits of base Ethernet MAC address (also used for PPP Endpoint discriminator) rxIP – IP upcall function pointer rxPPP – PPP upcall function pointer rxATM – ATM upcalll function pointer imaCb – IMA event callback 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.

12.4.1.3 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)

12.4.1.4 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)

12.4.1.5 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

12.4.1.6 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

12.5 NP Port API

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

12.5.1 Data Types

12.5.1.1 WniPortParam

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: ENET_SPEED_DUPLEX – Ethernet speed/duplex mode ENET_AN_CAPS – Ethernet auto-negotiation capablities ENET_REM_AN_CAPS – Ethernet auto-negotiation capabilities of link partner (get-only) ENET_AN_STATUS – Ethernet auto-negotiation status ENET_AN_RESTART – force Ethernet re-negotiation ENET_LINK_STATUS – Ethernet link status (get-only) ENET_PORT_ENABLE – Ethernet port enable/disable ENET_PORT_MAC_ADDR –Ethernet Port MAC address ENET_PORT_IPv4_ADDR – Ethernet Port IP v4 address ENET_PORT_IPv6_ADDR – Ethernet Port IP v6 address ENET_PORT_PAUSE_MODE – Ethernet Port MAC PAUSE mode ENET_PORT_STATE – Ethernet Port state ATM_FREEZE – ATM freeze port ATM_UNFREEZE – ATM unfreeze port ATM_IF_TYPE – ATM interface type ATM_PAYLOAD_SCRAMBLE – ATM Payload scrambling ATM_L0_STATE – ATM Layer 0 state ATM_LOOPBACK_MODE – ATM loopback mode TDM_CHUNK_SIZE – 32/64 bytes (not currently supported) TDM_IPv4_ADDR – Ipv4 address used for all TDM channels

12.5.2 ATM Port API

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

12.5.2.1 getAtmPortParam

Function:	int getAtmPortParam(int npIndex, int atmPortIndex, WniPortParam param, int32u* value)
Description:	This function is used to get the specified ATM parameter from the specified ATM port
Parameters:	npIndex – index of the NP for the operation atmPortIndex – index of the desired ATM port param – the parameter to get value – the returned value of the parameter if successful
Returns:	0 – operation was successful 1 – operation was not successful

12.5.2.2 setAtmPortParam

Function:	int setAtmPortParam(int npIndex, int atmPortIndex, WniPortParam param, int32u* value)
Description:	This function is used to set the specified ATM parameter on the specified ATM port
Parameters:	npIndex – index of the NP for the operation atmPortIndex – index of the desired ATM port param – the parameter to set value – the value of the parameter to set
Returns:	0 – operation was successful 1 – operation was not successful

12.5.2.3 getAtmPortStats

Function:	int getAtmPortStats(int npIndex, int atmPortIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to get the ATM port statistics from the specified ATM port.
Parameters:	npIndex – index of the NP for the operation atmPortIndex – index of the desired ATM port statBuffer – user buffer to hold returned statistics lastTime – user structure to hold last the returned value of last time the statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.5.3 Ethernet API

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

12.5.3.1 getEnetPortParam

Function:	int getEnetPortParam(int npIndex, int enetPortIndex, WniPortParam param, int32u* value)
Description:	This function is used to get the specified Ethernet parameter from the specified Ethernet port
Parameters:	NpIndex – index of the NP for the operation enetPortIndex – index of the desired Ethernet port param – the parameter to get value – the returned value of the parameter if successful
Returns:	0 – operation was successful 1 – operation was not successful

12.5.3.2 setEnetPortParam

Function:	int setEnetPortParam(int npIndex, int enetPortIndex, WniPortParam param, int32u* value)
Description:	This function is used to set the specified Ethernet parameter on the specified Ethernet port
Parameters:	npIndex – index of the NP for the operation enetPortIndex – index of the desired Ethernet port param – the parameter to set value – the value of the parameter to set
Returns:	0 – operation was successful 1 – operation was not successful

12.5.3.3 getEnetPortStats

Function:	int getEnetPortStats(int npIndex, int enetPortIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to get the Ethernet port statistics from the specified Ethernet port.
Parameters:	npIndex – index of the NP for the operation enetPortIndex – index of the desired Ethernet port statBuffer – user buffer to hold returned statistics lastTime – user structure to hold the returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.5.4 TDM API

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

12.5.4.1 getTdmPortParam

Function:	int getTdmPortParam(int npIndex, WniPortParam 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

12.5.4.2 setTdmPortParam

Function:	int setTdmPortParam(int npIndex, WniPortParam 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

12.5.4.3 getTdmPortStats

Function:	int getTdmPortStats(int npIndex, void* statBuffer, time_t* lastTime, int tdmChannel)
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 lastTime – user structure to hold returned value of last time statistics were read tdmChannel – TDM channel to get statistics from
Returns:	0 – operation was successful 1 – operation was not successful

12.5.4.4 setTdmConfig

Function:	int setTdmConfig(int npIndex, int framerIndex, int quadrantIndex)
Description:	This function is used to configure the TDM port clocking.
Parameters:	npIndex – index of the NP for the operation framerIndex – framer to use as clock source quadrantIndex – framer quadrant to use as clock source
Returns:	0 – operation was successful 1 – operation was not successful

12.5.5 Fabric API

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

12.5.5.1 getFabricPortParam

Function:	int getFabricPortParam(int npIndex, WniPortParam param, int32u* value)
Description:	This function is used to get the specified parameter from the
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

12.5.5.2 setFabricPortParam

Function:	int setAtmPortParam(int npIndex, WniPortParam param, int32u* value)
Description:	This function is used to set the specified parameter on the
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

12.5.5.3 getFabricPortStats

Function:	int getFabricPortStats(int npIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to get the statistics.
Parameters:	npIndex – index of the NP for the operation statBuffer – user buffer to hold returned statistics lastTime – user structure to hold returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.5.6 Mt-4 API

These API's allow an application to retrieve Mt-4 statistics.

12.5.6.1 getMtxChannelStats

Function:	int getMtxChannelStats(int npIndex, void* statBuffer, int tdmChannel, time_t* lastTime)
Description:	This function is used to retrieve Mt-4 channel statistics.
Parameters:	npIndex – index of the NP for the operation statBuffer – user buffer to hold returned statistics tdmChannel – TDM channel to retrieve statistics from lastTime – user structure to hold returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.5.6.2 getMtxGlobalStats

Function:	int geMtxGlobalStats(int npIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to retrieve the Mt-4 global statistics.
Parameters:	npIndex – index of the NP for the operation statBuffer – user buffer to hold returned statistics lastTime – user structure to hold returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.5.7 Framer API

This API allows an application to retrieve the framer status.

12.5.7.1 GetFramerStatus

Function:	int getFramerStatus(int npIndex, int framerIndex)
Description:	This function is used to retrieve the framer status.
Parameters:	npIndex – index of the NP for the operation framerIndex – framer to retrieve status from
Returns:	0 – operation was successful 1 – operation was not successful

12.6 Protocol API

This section lists the API's available on the host to configure and monitor the various WNI protocols.

12.6.1 Data Types

12.6.1.1 WniProtoParam

Description:	This structure contains an enumeration of the supported protocol parameters.
Type:	Enum
Usage:	The definition for the fields of this structure are as follows: AAL5_MAX_SDU_SIZE – Maximum AAL5 packet size supported. AAL5_RX_STOP - AAL5_RX_GO - AAL5_TX_STOP - AAL5_TX_GO - AAL5_RAS_TIMER - AAL2_CU_TIMER - AAL2_RX_STOP - AAL2_RX_GO - AAL2_TX_STOP - AAL2_TX_GO - DIFFSERV_WFQ_QUANTUM_AF1 - DIFFSERV_WFQ_QUANTUM_AF2 - DIFFSERV_WFQ_QUANTUM_AF3 - DIFFSERV_WFQ_QUANTUM_AF4 - DIFFSERV_WRED_PARAMS - DIFFSERV_STATS_TRACK - DIFFSERV_STATS_GET - DIFFSERV_TRACK_PORT_NUMBER - ATM_TM_TRACK_PORT_NUMBER -

12.6.2 AAL5

12.6.2.1 GetAal5Param

Function:	int getAal5tParam(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to get the specified parameter from the AAL5 protocol
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

12.6.2.2 SetAal5Param

Function:	int setAal5Param(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to set the specified parameter for the AAL5 prptocol
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

12.6.2.3 GetAal5Stats

Function:	int getAal5Stats(int npIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to get the AAL5 protocol statistics.
Parameters:	NpIndex – index of the NP for the operation statBuffer – user buffer to hold returned statistics lastTime – user structure to hold returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.6.3 AAL2

12.6.3.1 getAal2Param

Function:	int getAal2tParam(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to get the specified parameter from the AAL2 protocol
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

12.6.3.2 setAal2Param

Function:	int setAal2Param(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to set the specified parameter for the AAL2 protocol
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

12.6.3.3 getAal2Stats

Function:	int getAal2Stats(int npIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to get the AAL2 statistics.
Parameters:	NpIndex – index of the NP for the operation statBuffer – user buffer to hold returned statistics lastTime – user structure to hold returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.6.4 SSSAR

12.6.4.1 getSssarParam

Function:	int getSssartParam(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to get the specified parameter from the SSSAR protocol
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

12.6.4.2 setSssarParam

Function:	int setSssarParam(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to set the specified parameter for the SSSAR protocol
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

12.6.4.3 getSssarStats

Function:	int getSssarStats(int npIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to get the SSAR statistics.
Parameters:	NpIndex – index of the NP for the operation statBuffer – user buffer to hold returned statistics lastTime – user structure to hold returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.6.5 ATM TM

12.6.5.1 getAtmTmParam

Function:	int getAtmTmtParam(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to get the specified parameter from the ATM TM protocol
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

12.6.5.2 setAtmTmParam

Function:	int setAtmTmParam(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to set the specified parameter for the ATM TM protocol.
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

12.6.5.3 getAtmTmStats

Function:	int getAtmTmStats(int npIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to get the ATM TM statistics.
Parameters:	NpIndex – index of the NP for the operation statBuffer – user buffer to hold returned statistics lastTime – user structure to hold returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.6.6 DiffServ

12.6.6.1 getDiffServParam

Function:	int getDiffServtParam(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to get the specified parameter from the DiffServ protocol
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

12.6.6.2 setDiffServParam

Function:	int setDiffServParam(int npIndex, WniProtoParam param, int32u* value)
Description:	This function is used to set the specified parameter for the DiffServ protocol
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

12.6.6.3 getDiffServStats

Function:	int getDiffServStats(int npIndex, void* statBuffer, time_t* lastTime)
Description:	This function is used to get the DiffServ statistics.
Parameters:	NpIndex – index of the NP for the operation statBuffer – user buffer to hold returned statistics lastTime – user structure to hold returned value of last time statistics were read
Returns:	0 – operation was successful 1 – operation was not successful

12.7 I/O API

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

Note that packet reception is handled by the upcalls that were registered in the application “start” functions.

12.7.1 Data Types

12.7.1.1 BufferType

Description:	This structure contains an enumeration of the supported packet formats. This is a subset of the BufferType enum previously described.
Type:	Enum
Usage:	The definition for the fields of this structure are as follows: BT_LCP – PPP LCP BT_NCP_IPv4 – PPP IPCP BT_NCP_IPv6 – PPP IPV6CP BT_NCP_PPPMUX – PPPMUXCP BT_Ipv4 – IPv4 BT_Ipv6 – IPv6 BT_ATM – ATM

12.7.1.2 TdmDescData

Description:	This structure contains the fields necessary to send a packet on a TDM channel
Type:	Struct
Usage:	The definition for the fields of this structure are as follows: Int32u compressionId Int8u mcClass Int8u flags Int16u egressQueue

12.7.1.3 AtmDescData

Description:	This structure contains the fields necessary to send an ATM cell.
Type:	Struct
Usage:	The definition for the fields of this structure are as follows: CellHeader cellHeader Int16u vcIndex Int8u stf Int8u hecError

12.7.1.4 MacDescData

Description:	This structure contains the fields necessary to send an Ethernet packet.
Type:	Struct
Usage:	The definition for the fields of this structure are as follows: Int8u macDa[6] Int16u egressQueue

12.7.2 Functions

12.7.2.1 sendPacket

Function:	int sendPacket(int npIndex, int portIndex, int32u packetLength, void* thePacket, BufferType packetType, int32u* appData)
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 Ethernet port PacketLength – length of the packet in bytes ThePacket – user buffer to hold returned statistics PacketType – type of packet to send AppData – interface dependent forwarding information – must be one of the following previously described structures: TdmDescData AtmDescData MacDescData
Returns:	0 – operation was successful 1 – operation was not successful

13 DEFINITIONS

The following is a list of acronyms used in this document.

Acronym	Meaning
AAL	ATM Adaptation Layer
AF	Assured Forwarding
APS	Automatic Protection Switching
ATM	Asynchronous Transfer Mode
CBR	Constant Bit Rate
CID	Channel ID, Compression ID
CES	Circuit Emulation Service
CLI	Command Line Interface
CPS	Common Part Sublayer
cUDP	Compressed UDP
EF	Expedited Forwarding
FM	Fault Management (OA&M)
GCRA	Generic Cell Rate Algorithm
GFR	Guaranteed Frame Rate
GTP	GSM Tunneling Protocol
HDLC	High level Data Link Control
ICMP	Internet Control Message Protocol
IMA	Inverse Multiplexing over ATM
IP	Internet Protocol
IPC	Inter-Processor Communication
IPHC	IP Header Compression
MAC	Media Access Control
MC-PPP	Multi-class PPP
ML-PPP	Multi-link PPP
OA&M	Operations, Administration, and Maintenance
OC	Optical Carrier
PHB	Per Hop Behavior
PM	Performance Monitoring (OA&M)
PPP	Point to Point Protocol
QoS	Quality of Service
RFC	Request For Comments
RTP	Real-time Transport Protocol
SAR	Segmentation And Reassembly
SONET	Synchronous Optical NETwork
SSSAR	Service Specific SAR
STM	Synchronous Transfer Mode

Acronym	Meaning
TBD	To Be Determined
TDM	Time Division Multiplex
TM	Traffic Management (ATM)
UBR	Unspecified Bit Rate
UDP	User Datagram Protocol
UL-2	Utopia Level 2
VBR-nrt	Variable Bit Rate – non-real time
VBR-rt	Variable Bit Rate – real time
VC	Virtual Channel
VCC	Virtual Channel Connection
VT	Virtual Tributary
WFQ	Weighted Fair Queueing
WNI	Wireless Network Interface
WRED	Weighted Random Early Drop