Azcom eNodeB Layer-2 and Layer-3

System Architecture Specification

This document describes the architectural design of Azcom LTE eNodeB L2-L3 stack

Document Revision: 1.0
Date: July 24th, 2018

Azcom Technology srl
Headquarter:
Centro Direzionale Milanofiori
Strada 6, Palazzo N/2
20089 Rozzano (MI) – Italy
Tel.: +39 02 82450311

Development Center (Gurgaon, India)
Azcom Infosolutions (India) Pvt. Ltd.
3rd Floor, Tower B, UM House
Plot #35P, Sector 44
GURUGRAM-122002 (Haryana). INDIA.
Tel. +91-124-4937650

Website: http://www.azcomtech.com
E-mail: info@azcom.in

ABBREVIATIONS AND ACRONYMS

3GPP	Third-Generation Partnership Project
AM	Acknowledged Mode
BCCH	Broadcast Control Channel
BCH	Broadcast Channel
CCCH	Common Control Channel
CQI	Channel Quality Indicator
Cxxx	Control xxx
C-RNTI	Cell Radio Network Temporary Identifier
DCCH	Dedicated Control Channel
DL	Downlink
DL-SCH	Downlink Shared Channel
DTCH	Dedicated Traffic Channel
eNodeB	Evolved NodeB
EPC	Evolved Packet Core
FDD	Frequency Division Duplex
GMM	GPRS Mobility Management
HARQ	Hybrid Automatic Repeat Request
LTE	Long Term Evolution
MAC	Medium Access Control
MM	Mobility Management
NAS	Non Access Stratum
PBCH	Physical Broadcast Channel
PCCH	Paging Control Channel
PCH	Paging Channel
MCH	Multicast Channel
MT	Module Test
ngSCBP	Next-generation Small Cell Baseband Platform (Azcom LTE dual-mode baseband board)
PCH	Paging Channel
RACH	Random Access Channel
TDD	Time Division Duplexing
UCI	Uplink Control Information
UL-SCH	Uplink Shared Channel
PCFICH	Physical Control Format Indicator Channel
PDCCH	Physical Downlink Control Channel
PDCP	Packet Data Convergence Protocol
PDSCH	Physical Downlink Shared Channel

PHICH	Physical Hybrid HARQ Indicator Channel
PRACH	Physical Random Access Channel
P-RNTI	Paging Radio Network Temporary Identifier
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
QoS	Quality of Service
RACH	Random Access Channel
RLC	Radio Link Control
ROHC	Robust Header Compression
RRC	Radio Resource Control
Rx	Receive
SAP	Service Access Point
SI-RNTI	System Information Radio Network Temporary Identifier
SM	Session Management
TB	Transport Block
TCP	Transmission Control Protocol
TDD	Time Division Duplex
TM	Transparent Mode
Tx	Transmit
UE	User Equipment
UL	Uplink
UL-SCH	Uplink Shared Channel
UM	Unacknowledged Mode
PS	Protocol Stack

Contents

LIST OF FIGURES

LIST OF TABLES

1 Scope

This document describes the architectural design of Azcom LTE eNodeB Layer-2 and Layer-3 stack. It gives an overview of the software structure, its components, role of each component and the interfaces.
This document is intended for the Azcom eNB Protocol Stack team, test team and customers.

2 Introduction

This document describes the architecture of Azcom Technology's LTE eNodeB Layer-2 and Layer-3, herein referred to as Protocol Stack Software for the rest of this document. It covers high level key architectural design of the Protocol Stack layers for Linux as well as TI (Texas instruments) C66x SoC.

3 System Description

3.1 LTE protocol stack

Long Term Evolution (LTE) is designed to support packet-switched services providing seamless Internet Protocol (IP) connectivity between the User Equipment (UE) and the Packet Data Network (PDN), without disruption during mobility.
The complete network, known as Evolved Packet System (EPS), includes the Evolved Node B (eNB) implementing the new OFDMA/SC-FDMA radio access technology, the Evolved Packet Core (EPC) and the UE (mobile user device).

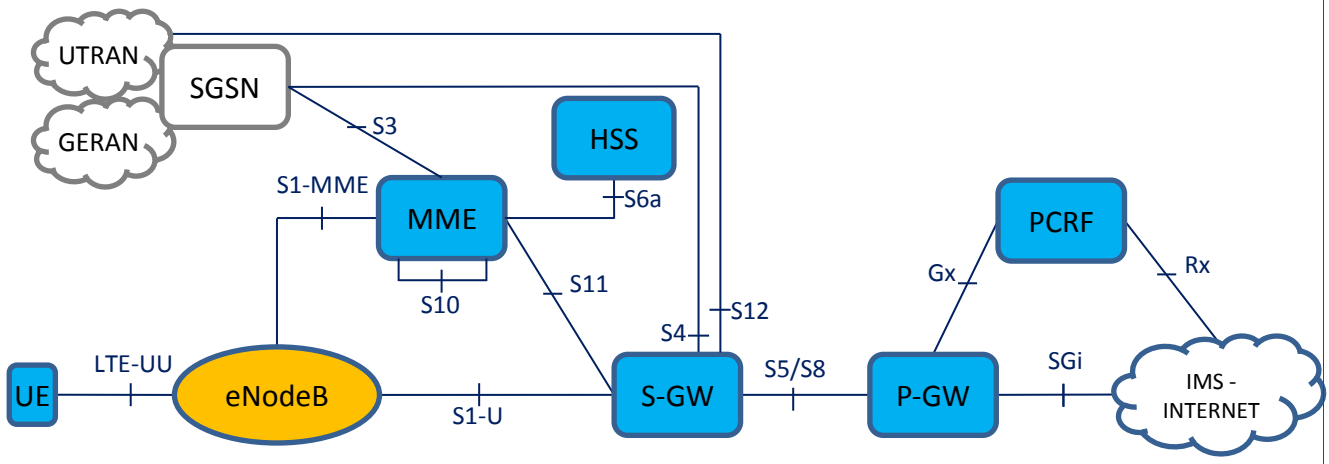


Figure 1 LTE network reference architecture

Azcom LTE eNodeB is the network element required to implement an LTE small cell eNB.
It provides high-speed radio connectivity to LTE devices which can be mobile phones, tablets, dongles etc.

3.2 System on Chip (SOC)

- Protocol stack software runs on TI's TCI66x System-on-Chip with Keystone 2 architecture. The SoC integrates Eight TMS320C66x DSP Core Subsystems (C66x CorePacs), Each With 1.0 GHz or 1.2 GHz C66x Fixed and Floating-Point DSP Core with Four ARM® Cortex® -A15 MPCore™ Processors at up to 1.4 GHz on a single chip.
- Stack processing uses hardware accelerators needed for real time packet processing, integrity, ciphering and fast path handling of GTP traffic. Hardware queues provide performance oriented inter-process communication across intra core as well as inter core threads.
- Stack processing is performed within the "Application Layer" that is hosted on DSP cores. In addition, a number of ancillary components are needed for proper functioning of the application layer.

All these components are defined herein as "Stack Operational Framework", which is composed of:

- SoC silicon (TCI6638K2K)
- The Real Time Operating System (RTOS) for the Layer 2, which is TI's native SYS/BIOS

- The operating system for the Layer 3, which is Yocto Linux as distributed by TI for the own ARM architecture
- The Platform software

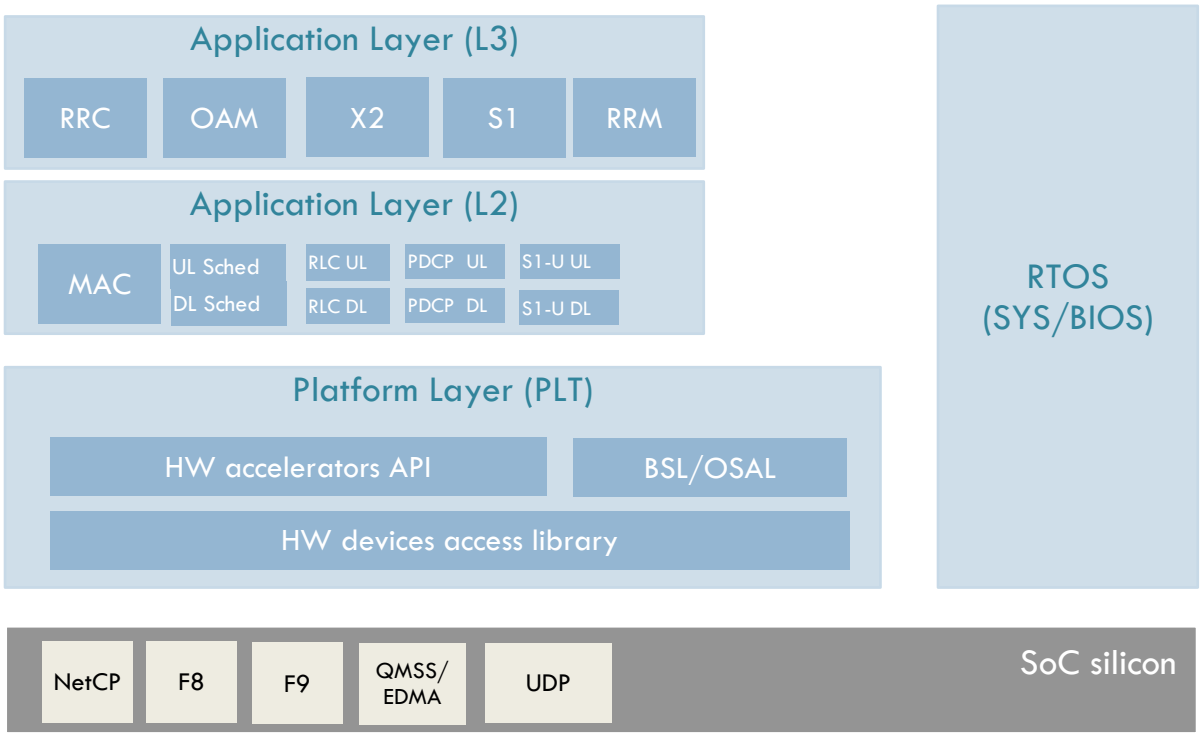


Figure 2 Operational Framework

3.2.1 Operating System

3.2.1.1 On DSP CorePac

TI provides TI-RTOS (Real Time Operating System) which scales from a real-time multitasking kernel (SYS/BIOS) [1] to a complete RTOS solution including additional middleware components and device drivers.

- SYS/BIOS is a scalable real-time kernel. It is designed for applications that require real-time scheduling and synchronization or real-time instrumentation.
- Provides preemptive multithreading, hardware abstraction, real-time analysis, and configuration tools.
- Helps minimize memory and CPU requirements on the target.
- Runs natively on DSP silicon and provides services such as tasks, SWIs, HWIs, semaphores, timers, mailboxes, queues etc. These services are exploited from this kernel as deemed necessary in stack software development.

3.2.1.2 On ARM CorePac

On the ARM corepac, the OS is linux, which is a custom made Yocto distribution by TI.
The software architecture of the operating system is Symmetric multi-processing (SMP) which dynamically determines the roles of individual processors. Each core in the cluster has the same view of memory and of shared hardware. Any application, process or task can run on any core and the operating system scheduler can dynamically migrate tasks between cores to achieve

optimal system load. The scheduler in an SMP system can dynamically re-prioritize tasks. This dynamic task prioritization enables other tasks to run while the current task sleeps.

3.3 Core Architecture

The Azcom eNB protocol stack software runs across 4 DSP cores and 1 ARM Processor of TI C66x SoC. In detail:

- All control plane components RRC, X2-AP/S1-AP i.e. the Layer 3 runs on ARM Processor.
- Layer 2 protocols PDCP/RLC and S1-U run on DSP Core 0 while MAC runs on DSP Core 1.
- Layer 1 software runs on two cores where DSP Core 2 is used for all UL channels and DSP Core 3 is used for all DL channels.
- DSP Core 0 is also responsible for system resource initialization and management.
- Logging infrastructure is based on both DSP Core 0 and Core 1.

As shown in the following figure, eNodeB software is distributed across multiple DSP cores and ARM processor of TI C66x based SoC.

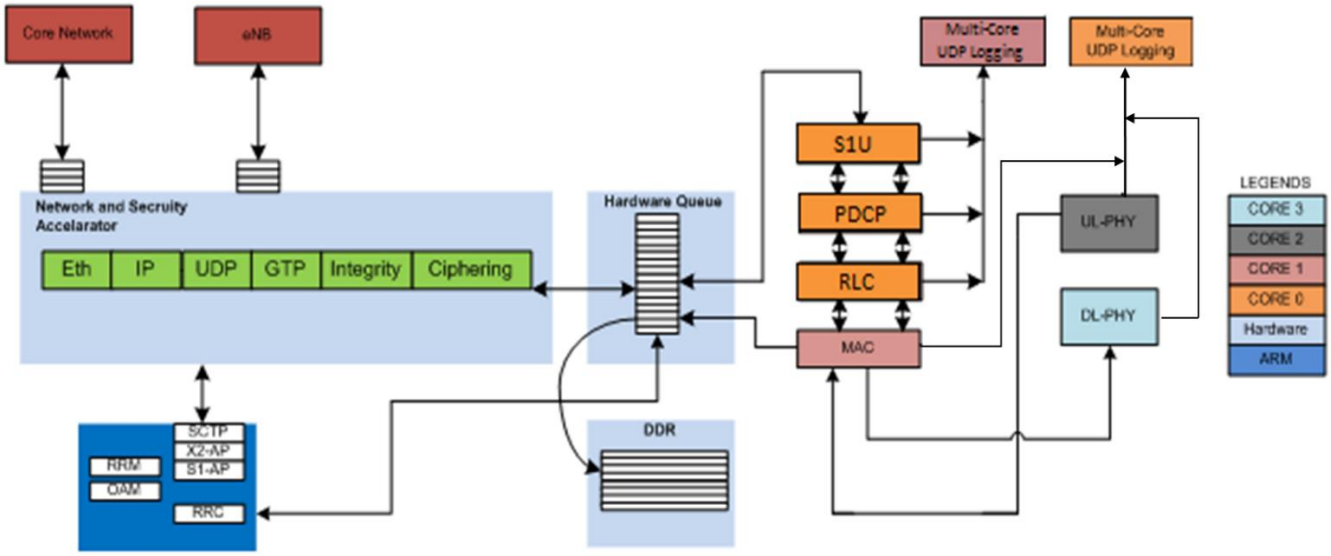


Figure 3 Core architecture

Computationally expensive functionality is offloaded to hardware accelerators e.g. integrity/ciphering with security accelerator being used for integrity and ciphering of air interface messages and the Network Co-processor for IP stack and fast path GTP processing Platform hardware queues are used for inter-process communication within a core and between multiple cores.

3.4 Memory Architecture

Each C6xx CorePac of K2 contains

- a 1024KB level-2 memory (L2)
- a 32KB level-1 program memory (L1P)
- a 32KB level-1 data memory (L1D)

A 6144KB i.e. 6MB multicore shared memory (MSM) SRAM and 524288 KB i.e. 512 MB of external DDR3 memory is also available in K2K, shared among all cores.

3.4.1 Memory partitioning

As mentioned above, each core has its dedicated L1P, L1D and L2 memories. Out of available MSM, a dedicated chunk is assigned to each core for its local usage and a memory section is shared among currently in-use cores. Similarly, for DDR3 a dedicated chunk of memory is available to each core and a sharable section is maintained which is used by cores in-use. Following table shows the details of how memory is partitioned (in terms of size) among cores.

Memory Areas	Memory	Size (in KB)	Description
Shared (among all cores)	MSMC	1952	MSMC memory to place common configuration modules i.e. netfp
	DDR3	51199	Shared between all cores, contains shared structures i.e. ltesharedinfo, tracing, qmss, dss etc.
Core 0	L2	512	512 KB of L2 is used as cache, remaining is used as SRAM
	MSMC	384	Core 0 specific memory
	DDR3	319487	Core 0 specific memory
Core 1	L2	512	512 KB of L2 is used as cache, remaining is used as SRAM
	MSMC	200	Core 1 specific memory
	DDR3	57344	Core 1 specific memory
Core 2 & Core 3	L2	896 + 896	Each core uses 128 KB of L2 as cache, remaining used as SRAM
	MSMC	1807 + 1799 + 1	Core 2 uses 1807 KB of MSMC, Core 3 uses 1799 KB of MSMC whereas 1 KB is shared among these two cores
	DDR3	49152 + 49152 + 16384	Each core uses 49152 KB of DDR3, 16384 KB is shared among two cores
Fault Management	DDR3	128 * 4	Each core has its dedicated memory reserved for storing fault management related information

Table 1 Memory Partition

Each core places its data into the available memory according to its usage i.e. data that requires faster access could be placed in MSMC or L2. Following table depicts the detail of memory sections arrangement.

p

Memory Areas	Memory Section	Description
	.uialogBuffer	UIA buffers for all the cores are placed under this section. This section is loaded into DDR3 shared memory region
	.netFPSharedMemory	NETFP Shared Data structures are placed under this section.

Shared (among all cores)		It is loaded into dedicated MSMC region for quick access.
	.appSyncSharedMem	Information used to sync SoC Init core & other cores on Init completion is placed in this section. It is loaded into MSMC for faster access.
	.cfgMemorySection	Resource Manager Configuration information is placed in this section. It is also loaded into MSMC.
	.pktlibSharedMemory/ .msgComSharedMemory	Packet library and Msgcom Multicore data structures are placed in this section. It is loaded in MSMC.
	.lteSharedInfo	Contains information related to system statistics encapsulated in a structure modified by all the cores. It is loaded into dedicated DDR3 memory area.
	DSSsection/QMSS_SHARED	Contains related shared information, placed into dedicated DDR3 memory regions.
Core 0	*.data/.code/.bss*	The code and data segments are loaded into L2 SRAM.
	.const/.stack/.text	The stack area, constant and text sections are placed in DDR3 cacheable region.
	.cinit/.etb_buffer	Exception trace buffers, are loaded into DDR3 non-cacheable region.
	Critical sections	Core's local heaps and data/text critical information are placed into L2 SRAM.
	libraries	Libraries are loaded into DDR3 cacheable region.
	MultiBuf Heap	Placed into dedicated DDR3 memory region
Core 1	*.data/.code/.bss/.stack*	The code and data segments are loaded into L2 SRAM.
	.const /.text	The stack area, constant and text sections are placed in DDR3 cacheable region. Some part of const critical information is placed in MSMC region for quicker access.
	.cinit/.etb_buffer	Exception trace buffers, are loaded into DDR3 non-cacheable region.
	Critical sections	Core's local heaps and data/text critical information are placed into L2 SRAM.
	libraries	Libraries are loaded into DDR3 cacheable region.
	MultiBuf Heap	Placed into dedicated DDR3 memory region

Table 2 Memory Sections

3.4.2 Heaps

Each core creates one or multiple heaps out of the memory allocated to it. These can be local or shared (available for access to all cores) heaps which are created as multiple buffer heaps (provides configured memory chunks) or pktlib heaps (used by navigator data structures) and are used for dynamic memory allocation or to perform various navigator functionalities etc.

3.4.2.1 HeapBuf

HeapBuf is used for allocating fixed-size blocks of memory, and is designed to be fast and deterministic. HeapBuf is ideal for allocating space for fixed size objects, since it can handle the request quickly and without any fragmentation. A HeapBuf may also be used for allocating objects of varying sizes when response time is more important than efficient memory usage. Allocating from and freeing to a HeapBuf always takes the same amount of time, so a HeapBuf is a deterministic memory manager.

3.4.2.2 HeapMultiBuf

TI SYS/BIOS provides a memory manager for dynamic memory allocations. The memory manager can be configured into multiple buckets of different sizes, which can be used to reserve memory at the system start.
For dynamic memory allocations, a default *HeapMultiBuf* is used on both Core 0 and Core 1. *HeapMultiBuf* manages multiple fixed-size *HeapBufs*. Each buffer contains a fixed number of allocable memory 'blocks' of the same size. The *HeapMultiBuf* configuration used on Core 0 and Core 1 is given in tables below. The alignment for each block is 128 bytes.

HeapMultiBuf Division on Core 1	
Blocksize	**NumBlocks**
128	11100
512	8000
1024	2000
2048	4000
8192	1000
10240	200
16384	50
32768	20
80896	20
131072	4
176256	4
206464	4

HeapMultiBuf Division on Core 0	
Blocksize	**NumBlocks**
128	136640
512	170000
1024	4000
2048	9000
8192	9980
16512	70
40960	582
1048576	1

Table 3 Heap MultiBuf Division

Each packet is allocated from the heap where its utilization varies with number of active UEs in the system. Memory statistics of HeapMultiBuf can be logged using UDP trace for analysis.

3.4.2.3 Packet Library Heaps

Pktlib is a library provided by SYSLIB which abstracts the navigator data structures from application providing more generic data structures. Pktlib provides heaps for using this functionality, it uses the CPPI hardware descriptors for optimal usage at application layer. Descriptors are small memory areas that describe the packet of data to be transferred through the system.
For heap creation, the configuration encompasses the required number of descriptors, size of data buffer to be associated with each descriptor, number of descriptors with no data buffer attached to it i.e. zero buffer descriptor, memory region from which descriptors/buffer to be allocated and some additional parameters. Following are the details of pktlib heaps created at core 0 and core 1.

	Heap	Memory requirement		Description
		[Descriptors, Zero Buffer Descriptors, Data Buffer Size, Descriptor Size, Memory Region]	**Consumption**	
Shared	*MsgcomQProxySharedHeap*	{25, 3, 3072, 128, MSMC}	78 KB	Used for transmission of GTP path management messages and with NETCP for using its ciphering/integrity functionality
	PA_CommandHeap	{5, 0, 576, 128, MSMC}	3 KB	Heap for sending commands to PA
	NetFP_PktTxHeap	{20, 100, 1536, 128, MSMC}	45KB	Used to create SA flow for fast path handling
	MyExtMemHeap	{128, 0, 9088, 128, DDR3}	1152KB	Used for inter/intra core message communication
	MyMSMCHeap	{100, 0, 64, 64, MSMC}	12 KB	Used for inter/intra core message communication
	Reassembly_Heap	{60, 5, 1536, 128, MSMC}	98 KB	Used for NETCP IP packet reassembly
	QoS_Heap	{128, 0, 1536, 128, MSMC}	208 KB	Used by QoS subsystem to receive packets from SA or PA.
Core 0	*MyNETFPFlowHeap*	{7158, 1024, 8192, 128, DDR3}	58286 KB	Used for sending uplink data and receiving downlink data to/from NETCP respectively
	MyNETFPSignallingFlowHeap	{10, 0, 1500, 128, DDR3}	27 KB	Used for ingress packet flow (from NetCP to SW), security flows (from SA to SW)
	Core0_NETFP_HeaderHeap	{50, 0, 256, 128, MSMC}	18 KB	Used to create networking headers while sending out packets
	Core0_NETFP_FragmentHeap	{100, 0, 0, 128, MSMC}	12 KB	Used to fragment IP packets
	Core0_NetFP_PktRxHeap	{10, 0, 8192, 128, MSMC}	81 KB	Used for NETFP reassembly for packet size > 10 Kbytes
Core 1	*Core1_NETFP_HeaderHeap*	{156, 0, 256, 64, MSMC}	48 KB	Used to create networking headers while sending out packet
	Core1_NETFP_FragmentHeap	{100, 0, 0, 128, DDR3]	12KB	Used to fragment IP packets

Table 4 Pktlib Heaps

3.4.3 Cacheability

- It is possible to convert the entire L1P or a part of L1P into cache. L1P supports cache sizes of 4K, 8K, 16K, and 32K.
- L1D memory can be configured as a one-way associative cache with supported cache sizes of 4KB, 8KB, 16KB or 32KB.

- A part of L2 memory can be configured as four-way associative cache with supported cache sizes of 32KB, 64KB, 128KB, 256KB, 512KB or 1MB. The L2 memory is always initiated to all SRAM after reset. The part of L2 memory configured as SRAM is by default cacheable.
- The MSM is always configured as all SRAM and by default cacheable. When configured as L2 shared, it is cacheable by L1P and L1D cache. When configured as L3 shared, it is cacheable by L1 and L2 cache.
- DDR3 can be configured as all cacheable, but with the current implementation some portion of DDR3 memory is made cacheable which requires faster access. Each core can configure the cacheable region in DDR3 memory using system provided APIs (uses MAR registers).

Following table provides the detail of cache and cacheable regions on core 0 and core 1.

Core Pac	Memory	Cache Size (in KB)	Cacheable Region	Description
Core 0	L1P & L1D	32	-	Configured as all cache by default
	L2	512	Remaining 512 KB SRAM is cacheable	512 KB of L2 is used as cache
	DDR3	-	*DDR3_ML_HEAP0* [229376 KB] *DDR3_CORE0_CACHE* [32768 KB]	Cache is enabled on DDR3 region where dynamic heap is placed i.e. multibuf heap and where core's local data, stack, const and text sections are placed.
Core 1	L1P & L1D	32	-	Configured as all cache by default
	L2	512	Remaining 512 KB SRAM is cacheable	512 KB of L2 is used as cache
	DDR3	-	*DDR3_ML_HEAP* [32768 KB] *DDR3_CORE1_CACHE* [16384 KB]	Cache is enabled on DDR3 region where dynamic heap is placed i.e. multibuf heap and where core's local data, stack, const and text sections are placed.

Table 5 Cacheable regions

3.4.4 Dimensioning

At core 0 the memory is majorly utilized by RLC and PDCP layer which maintains information of the UE per bearer basis. A new RLC or PDCP entity has been created for each UE per bearer, the memory has been allocated from heap multibuf as the corresponding RLC or PDCP entities are created dynamically whenever a UE attaches to the system. Following table depicts the memory utilization from heap multibuf at core 0.

	Block Size	Number of blocks	Description
RLC	1024 bytes	1 * 3	One for each bearer i.e. SRB 1, SRB 2 and DRB [*DL entity*]
	2048 bytes	1 * 3	One for each bearer i.e. SRB 1, SRB 2 and DRB [*DL entity*]
	8192 bytes	3 * 3	Three for each bearer i.e. SRB 1, SRB 2 and DRB [*2 for DL entity & 1 for UL entity*]
	40960 bytes	1 * 3	One for each bearer i.e. SRB 1, SRB 2 and DRB [*DL entity*]
PDCP	128 bytes	1 * 2	SRB DL entity
	512 bytes	1 * 2	SRB UL entity
	16512 bytes	2	1 for UL entity and 1 for DL entity in case of handover

Table 6 Layer wise memory utilization

Block Size	Total available blocks	Number of blocks used	Number of user estimation
128 bytes	136640	2	~ 64 UEs
512 bytes	170000	2	
1024 bytes	4000	3	
2048 bytes	9000	3	
8192 bytes	9980	9	
16512 bytes	70	1	
40960 bytes	580	3	

Table 7 Memory distribution at Core 0

On Core 1 the memory is allocated statically for N number of UEs, not constraining the maximum number of supported users due to surplus memory availability. This N is a result of memory constraints on core 0 limiting the number to **64 simultaneous UEs** (the memory blocks can be rearranged to support more number of users).

4 Platform Services

The platform layer includes all the services provided to the application layer for proper functioning. It provides a particular functionality which is common between several software components and usually abstracts the hardware layer from the software functionality.

4.1 Platform Abstraction

Platform abstraction layer is used to provide an abstraction over underlying OS and platform API's enabling seamless portability of stack software to a new platform. Azcom protocol stack software defines platform abstraction layers (BSL and OSAL) using which it can be easily ported from a standard Linux platform to a TI or any other platform.

4.1.1 BSL and OSAL

BSL (Back Support Layer) is an interface used by Protocol Stack application for any system calls. OSAL is the Operating System Adaptation Layer which maps BSL API's to a particular OS/platform. The advantage here is that for porting to a new platform, the application code i.e. protocol stack does not need any modification.

BSL is based on OSAL, where OSAL is platform dependent, Azcom eNB PS supports Linux and TCI66xx platform. Each platform has a corresponding OSAL for mapping BSL APIs and BSL provides macros (wrappers) over the OSAL functions. Only BSL macros can be used in the functional code base to access any resources from the OSAL.

BSL supports all types of services like threading, inter-thread communication, memory management, semaphore acquiring and releasing mutex, string handling, file handling, timers and logging of the logs and trace messages.

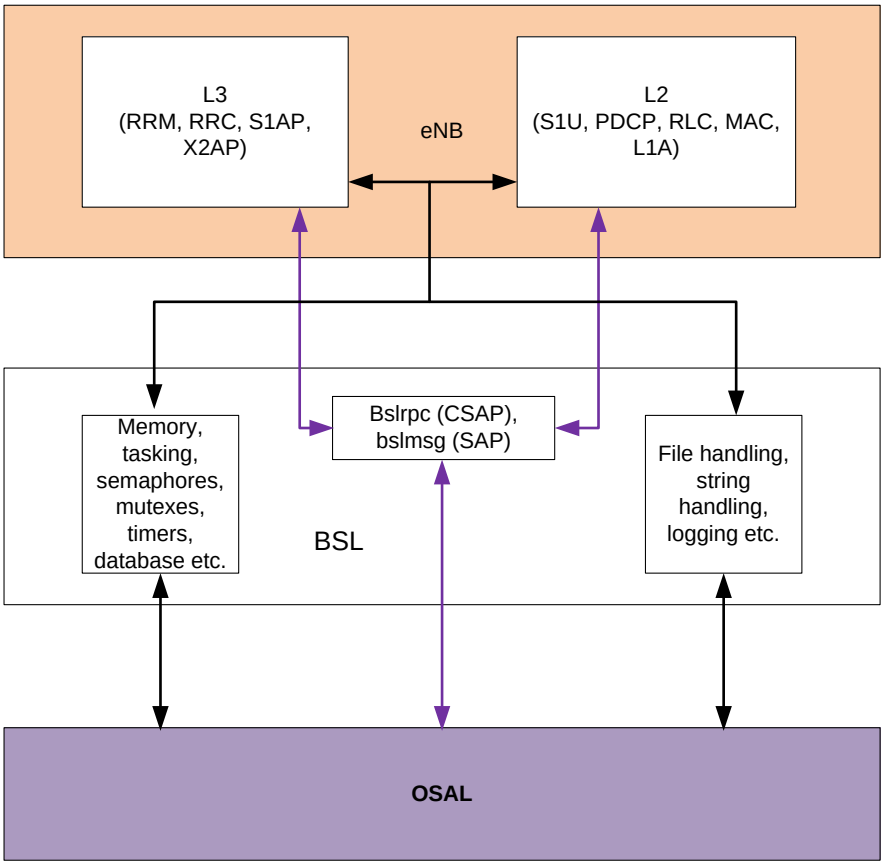


Figure 4 BSL/OSAL Architecture

The above figure depicts how it is structured with the functional code base. All services provided by BSL is implemented in OSAL.

4.2 IPC mechanisms

IPC or Inter-Process Communication is an entity defined within the framework of the Azcom eNodeB protocol stack which is used for inter-thread communication and creating SAPs (Service Access Point). IPC can be used to communicate with the following:

- Other threads on the same processor,
- Threads on other processors (DSP or ARM).

Since, it is based on an underlying Multicore Navigator (CPPI) [2] mechanism, it uses a Queue Manager Subsystem (QMSS) and a Packet DMA (PKTDMA) to control and implement high-speed data packet movement within the device. This functionality is encapsulated by MSGCOM, a library provided in SYSLIB [3]. The MSGCOM library is an IPC mechanism which allows messages to be exchanged between a reader and writer. The reader & writers could be located on the same DSP core, different DSP cores or even between an ARM and DSP as explained below:

- Inter & Intra DSP communication
  This type of communication is fulfilled via Queue Channels. In Queue channels messages are placed into a specific well define queue. The message memory has to be shared between the reader and writer making it similar to an IPC mechanism using shared memory but since MSGCOM is based on the Navigator hardware infrastructure, atomicity

across multiple cores is guaranteed by the hardware which makes the software free of any multi-core locks.

- ARM-DSP Communication
  The communication is carried over Queue Ring channels, which provide the ability for remote entities such as ARM and DSP to communicate over a dedicated set of infrastructure DMA queues. The DSP Cores and ARM Cores are accessible and connected to each other via the QMSS infrastructure queues. On DSPs, PKTDMA is used to interact with the QMSS infrastructure DMA queues whereas ARM uses the UDMA library to communicate between user and kernel space using UDMA memory.

Each thread is associated with a SAP, which has associated message queues. The messages to be sent to a thread are sent on these associated messages queues, while the threads wait on these queues for the messages. IPC has following functions to operate on SAPs:

- *Sap_init():*
  Initializes a SAP. Every SAP must be initialized before using it as initialization will allocate message queue, set its length, create lock for its use and set its parent ID.

- *Msgsend():*
  Whenever a thread wants to send a message to other thread, this function is called, take the lock and add the message to the queue.

- *Msgrcv():*
  Every thread keeps on calling this function to receive messages from others. Whenever, there is a message, it is de-queued from the message queue and handled appropriately.

- *Sap_free():*

When to release a particular SAP, this function is used and the SAP is freed by de-allocating the message queue.

4.3 Synchronization mechanisms

4.3.1 Semaphore

Semaphores are often used to coordinate access to a shared resource among a set of competing tasks. These can also be used as signals for task synchronisations. SYS/BIOS provides a fundamental set of functions based upon semaphores which uses the ti.sysbios.knl.Semaphore module. This functionality is encapsulated by the BSL and OSAL.

4.3.1.1 Semaphore Creation

A semaphore can be initialized by any of the tasks at runtime or at system startup using BSL_SEM_CREATE, which returns a handle to the semaphore, BSL_SemHandle (semHandle).

```
BSL_SemHandle my_sem;
BSL_SEM_CREATE(BSL_SemHandle * semHandle, int32_t count, uint32_t isBinaryMode);
```

Semaphore objects can be declared as either counting or binary semaphores. The argument *isBinaryMode* defines the type of semaphore, if 1 then it's a binary semaphore else it's a counting semaphore.

4.3.1.2 Semaphore Pend

BSL_SEM_TAKE should be called before accessing a shared resource. To take a semaphore, the task has to wait for particular time period or till another task releases the resource (BSL_SEM_WAIT_FOREVER) i.e. infinite wait, which is specified as an argument of BSL_SEM_TAKE (*timeout*).
BSL_SEM_TAKE(BSL_SemHandle * semHandle, int32_t timeout);

4.3.1.3 Semaphore Post

After a particular task has operated on the shared resource then it should use BSL_SEM_GIVE to release the resource.
BSL_SEM_GIVE(BSL_SemHandle * my_sem);

4.3.1.4 Semaphore Deletion

Once the use of semaphore is over it can be deleted to free up the resources by using BSL_SEM_DELETE. An example usage is as follows:
BSL_SEM_DELETE(my_sem);

4.3.1.5 Semaphore Wrapper

Class *BslSemaphore* wraps the functionality of semaphore (creation, pend, post and deletion). The constructor creates the semaphore and destructor deletes it.
This is further wrapped in another template class *BslGuard* which can be used for semaphore/mutex. It pends the semaphore in constructor and posts it in destructor.
BslGuard<BslSemaphore> BslSemGuard;
The object of *BslSemGuard* can be created to guard the critical section wherever required.

4.3.2 Mutex

Mutex are used for preventing concurrent accesses to critical regions of code. Threads can be preempted by other threads of higher priority (unless priority inheritance mutex is used in order to prevent priority inversion), and some sections of code need to be completed by one thread before they can be executed by another thread. BSL and OSAL encapsulate the Gate module (ti.sysbios.gates.GateMutex and ti.sysbios.gates.GateMutexPri) provided by SYS/BIOS to provide the mutex functionality in the system.

4.3.2.1 Mutex Creation

A mutex can be initialized by using BSL_MUT_CREATE and by BSL_MUT_CREATE_PRI_INH for a mutex which implements priority inheritance. Both the MACROs are mapped to *azosal_mutex_init*. This call returns a BSL_MutexHandle which should be used for all access to the mutex.

BSL_MutexHandle my_mutex;
BSL_MUT_CREATE(azosal_mut_t* azosal_mutex_Handle,NULL);
or
BSL_MUT_CREATE_PRI_INH(azosal_mut_t* azosal_mutex_Handle, uint8_t prio_inh);

4.3.2.2 Mutex Take

BSL_MUT_TAKE should be called before going into the shared region to get the lock.
BSL_MUT_TAKE(azosal_mut_t* azosal_mutex_Handle, int32_t timeout_ms);

4.3.2.3 Mutex Give

To release the shared resource and unlock the mutex, BSL_MUT_TAKE can be used.
BSL_MUT_GIVE(my_mutex);

4.3.2.4 Mutex Delete

To delete the mutex:
BSL_MUT_DELETE(my_mutex);

4.3.2.5 Mutex Wrapper

Class *BslMutex* and *BslMutexPri* wraps the functionality of mutex (creation, pend, post and deletion).
The constructor creates the mutex and destructor deletes it.
This is further wrapped in class *BslGuard and BslMuxPriGuard.* It takes the mutex in constructor and gives it in destructor.
BslGuard<BslMutex> BslMuxGuard;
BslMuxPriGuard obj;
The object of *BslMuxPriGuard* or *BslMuxGuard* can be created to guard the critical section wherever required.

4.4 Timing modules

The timing managenment on stack is based on following:

- **Timing registers provided by hardware**
  TSCH, TSCL and few other are provided by hardware. These are used in managing timing information of logging framework.

- **A Timing module implemented on software**
  A timing module is implemented which is responsible for maintainance of timing information at layer 2 and layer 3.
  - It uses time-base which is provided by the platform operating system or the LTE physical layer.
  - If configured to use a time-base provided by the LTE physical layer, it works on signal received from the Physical layer, which becomes a reference as the start of a TTI.
  - When configured to use a time-base provided by the platform, on DSP it uses the *ti.sysbios.knl.Clock* module of SYS/BIOS and on ARM it uses the System Time module of Linux.
  - One-shot and periodic timer services are provided by the time server, triggering either call-back functions (for short procedures) or semaphores for synchronizing independent threads.
  - DSP Core 1 uses time-base provided by LTE physical layer and Core 0 uses time-base provided by Core 1. ARM uses time-base provided by the platform

4.5 Logging

High performance multi core UDP logging mechanism is provided for generating logs. Debug traces are generated from the software application and passed over a UDP socket which are captured into a log file by an external UDP log server.
The logging mechanism uses common APIs within both L2 and L3 code to send messages to the external log collector.
Logging system is divided into five severity levels, *tTraceLevel.* Any one type (or all types) of log can be captured as per requirement. Following are the trace levels provided:

1. TRACE_CRITICAL: Critical problem occured
2. TRACE_ERROR: Error occurred
3. TRACE_WARNING: Warning occurred
4. TRACE_INFO: All Traces need to be logged
5. TRACE_EVENT: Some System event occurred

TRACE_L2L3 is used for sending logs from layer 2 and layer 3.
Exmaple:
TRACE_L2L3(*tTraceLevel*, *log mask*, "log string\n", arguments);

4.5.1 DSP Logging

A log mask of 64-bit is available for DSP (common for Core 0 and Core 1) to further control the logs captured, in which upper 32 bits are for DEBUG and lower 32 bits are for imporatant information logging.

4.5.2 ARM logging

A log mask of 64-bit is available for ARM to further control the logs captured. Similar to DSP, in which upper 32 bits are for DEBUG and lower 32 bits are for imporatant information logging.

4.6 Third Party Components

The third-party components used by Azcom eNB protocol Stack are:

4.6.1 ASN Library

The ASN.1 (Abstract Syntax Notation One) runtime library contains the low-level constants, types, and functions that are assembled by the compiler to encode/decode more complex structures. This library is provided by Objective Systems, Inc.

5 External Interfaces

Azcom eNodeB Protocol Stack contains multiple interfaces with external components. These are described as follows:

5.1 MAC-PHY

Azcom's protocol stack software implements standard Femto Forum LTE compliant L1 interface (the P5 and P7 interface as shown in figure 4) v 1.1. All the interfaces that we discussed in this section so far are external interfaces of Azcom eNodeB i.e the interfaces that lie outside of eNodeB. The L1-L2 interface i.e the interface between MAC and PHY is internal to eNodeB.

The FAPI interface specification divides the messages in two types:

- P5 Messages: messages for PHY configuration procedures: infrequent, for PHY bootstrap and initial configuration; they are exchanged between the PHY and the "PHY control" block of the protocol stack software.
- P7 Messages: messages containing real user's data: very frequent (for example, each subframe), are used for continuous radio resources configuration and data to be sent on the physical channels. These are exchanged with MAC layer both in uplink and in downlink directions.

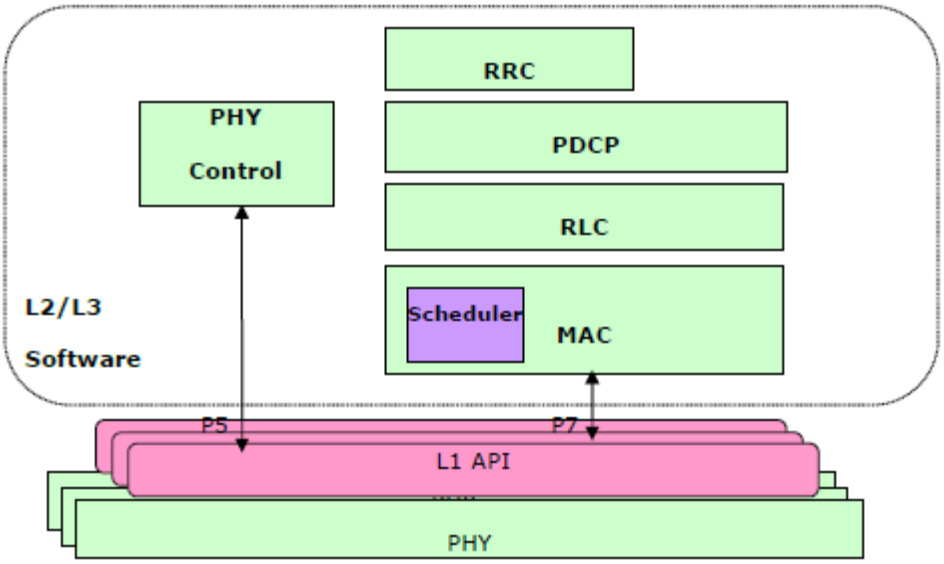


Figure 5 FAPI Interface

The actual data exchanged between PHY and L2 is stored in a shared DDR memory and only pointers of transport blocks gets exchanged.
Only data for control information (DCI, HARQ, etc.) is copied between L1-L2.
High performance hardware queues (using QMSS which is part of TI platform libraries) are used for communication between MAC and PHY.

5.2 S1 AP

- The LTE S1-MME interface is used for exchanging signalling traffic between eNodeB and the MME.
- S1-MME interface consists of Stream Control Transmission Protocol (SCTP) over IP and supports multiple UEs through a single SCTP connection. SCTP over IP provides guaranteed data delivery.
- The LTE S1-MME is responsible for Evolved Packet System (EPS) bearer setup/release procedures, the handover signalling procedure, paging procedure and the NAS transport procedure.
- S1-AP interface uses SCTP as the underlying transport mechanism. At eNodeB side, SCTP of ARM Linux is used for transport between eNodeB and MME.

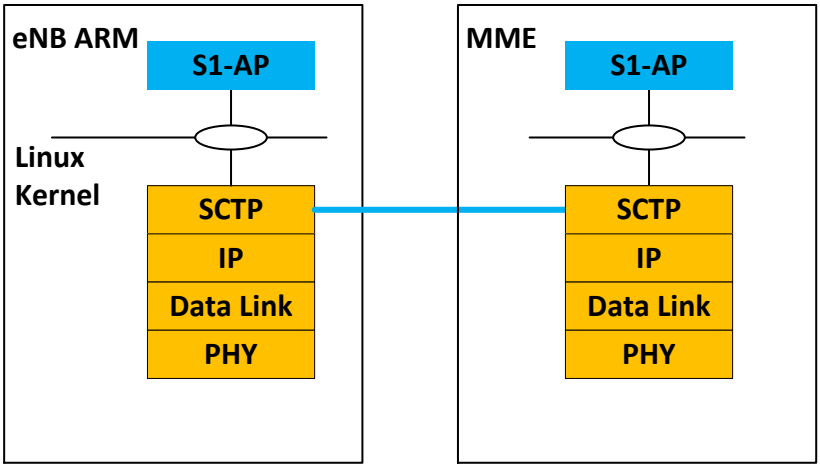


Figure 6 : S1-AP interface

5.3 S1 U

- The S1-U interface is responsible for delivering user data between the eNodeB and the S-GW.
- The S1-U interface provides non-guaranteed data delivery of LTE user plane Protocol Data Units (PDUs) between eNodeB and S-GW. Transport network layer is built on IP and GTP-U. UDP/IP carries the user plane PDUs between the eNodeB and the S-GW. A GTP tunnel per radio bearer carries user traffic.

In Azcom eNodeB stack, the processing of GTP packets is offloaded to a hardware accelerator for faster processing of incoming/outgoing user traffic between eNodeB and S-GW. Packet accelerator which is part of a network co-processor on TI SOC is configured for every active GTP tunnel at eNodeB. These packets are moved to high performance hardware queues which are then processed by S1-U layer at eNodeB. Packet acceleration helps in offloading IP stack processing through deep inspection of GTP packets at Ethernet level itself. This mechanism helps in achieving high throughput and improves overall system performance.

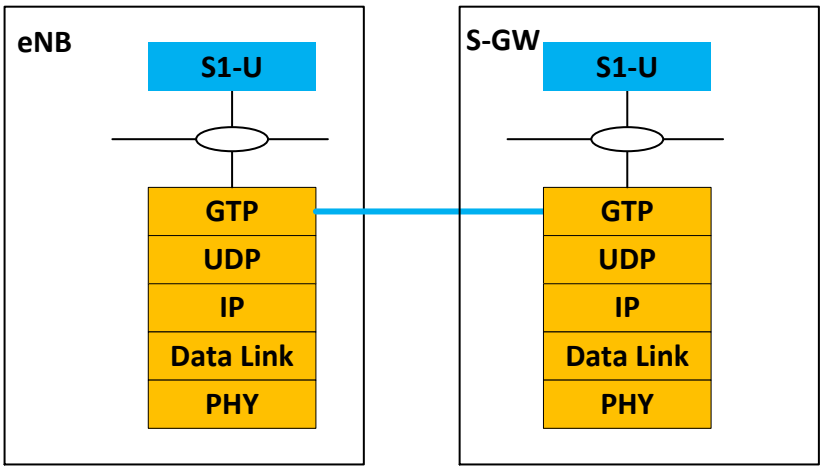


Figure 7 S1-U interface

5.4 X2

X2 interface is used to connect 2 neighbouring eNodeB's. X2 protocol includes control plane and user plane interfaces to exchange signalling and user traffic between 2 eNodeB's.
Primarily, the signalling traffic consists of the following functionalities:

- Load management and exchange of traffic information across eNodeB's to manage traffic in coordination
- Handover management, including establishment/release of tunnels between source and target eNB's basically exchanging signalling traffic across eNodeB's for X2 handover. It also helps in handover preparation of target eNodeB.

X2-C protocol uses SCTP as the underlying transport mechanism.

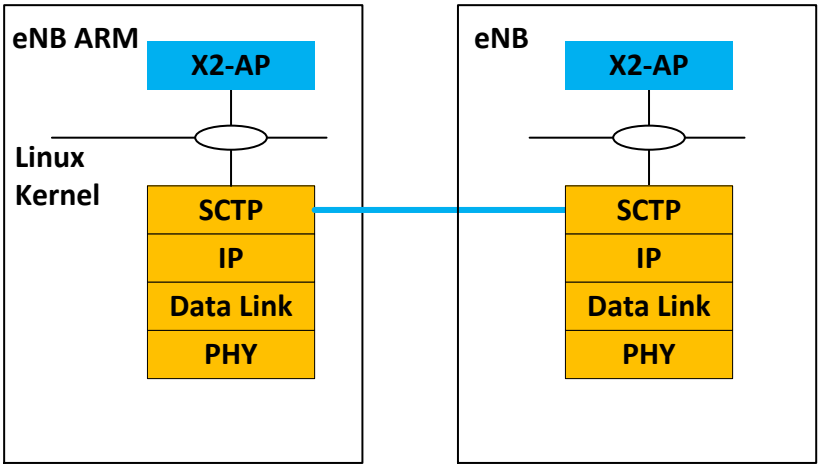


Figure 8 X2-C interface

X2-U interface enables end to end packets transfer across eNodeB's during handover.
Once the target cell is informed about the user's handover from source to target cell, the GTP packets buffered at the source eNodeB are transferred to target cell using X2 interface.
Underlying transport protocol used for sending/receiving packets over X2 interface is GTP.

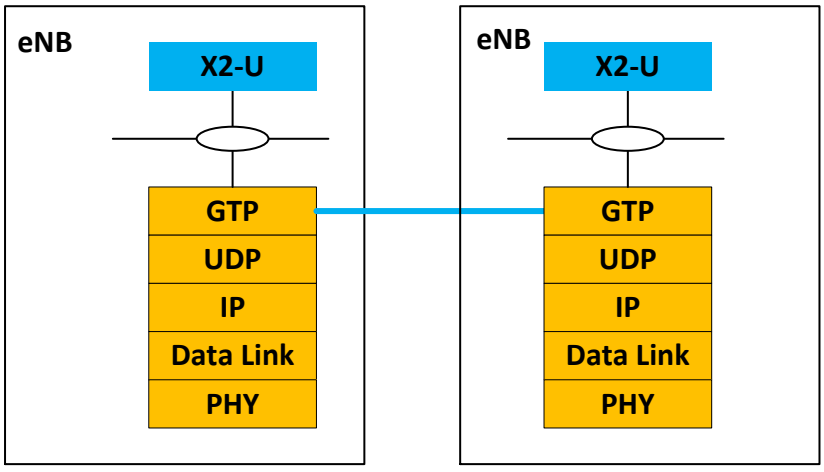


Figure 9 X2-U interface

5.5 O&M

O&M agent interacts with eNodeB via the following mechanisms:

- Message passing over UDP Socket connection
  AzOamAgent.bin running as a background on ARM receives and sends messages to the Layer 3 over port number 5050. These messages are received by oam thread integrated in Layer 3 application, which further interacts with RRC and other modules.
- Shared Memory
  azOamAgent.bin also interacts directly with LteSharedInfo.bin to execute various functionalies e.g. setting the severity & masks of Tracing, acquiring system statistics, getting load of each core, etc.
- AzBus
  OAM agent communicates with layer 3 application via another interface based on AzBus. The AzBus encapsulates the DBUS functionality provided in the linux platform.

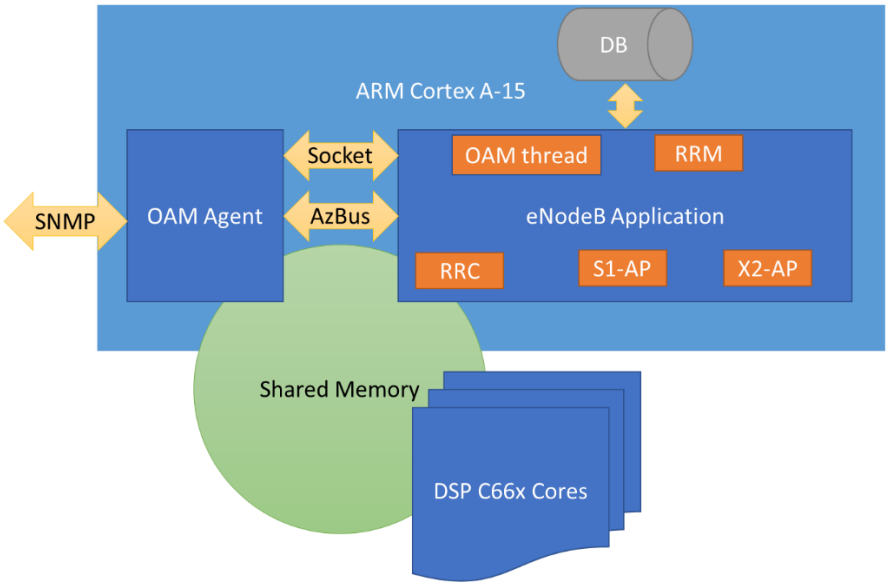


Figure 10 OAM Interface

6 SW Architecture

6.1 Thread Architecture

The Azcom eNodeB Protocol Stack is implemented in C++ as a multi-layered and multi-threaded system. The functionality of protocol layers is distributed into various tasks that run on DSP/ARM processors. Layer-2 executes on two DSP cores (Core-1 and Core-0) and Layer-3 runs on single ARM processor. Figure below depicts the distribution of tasks on the protocol stack.

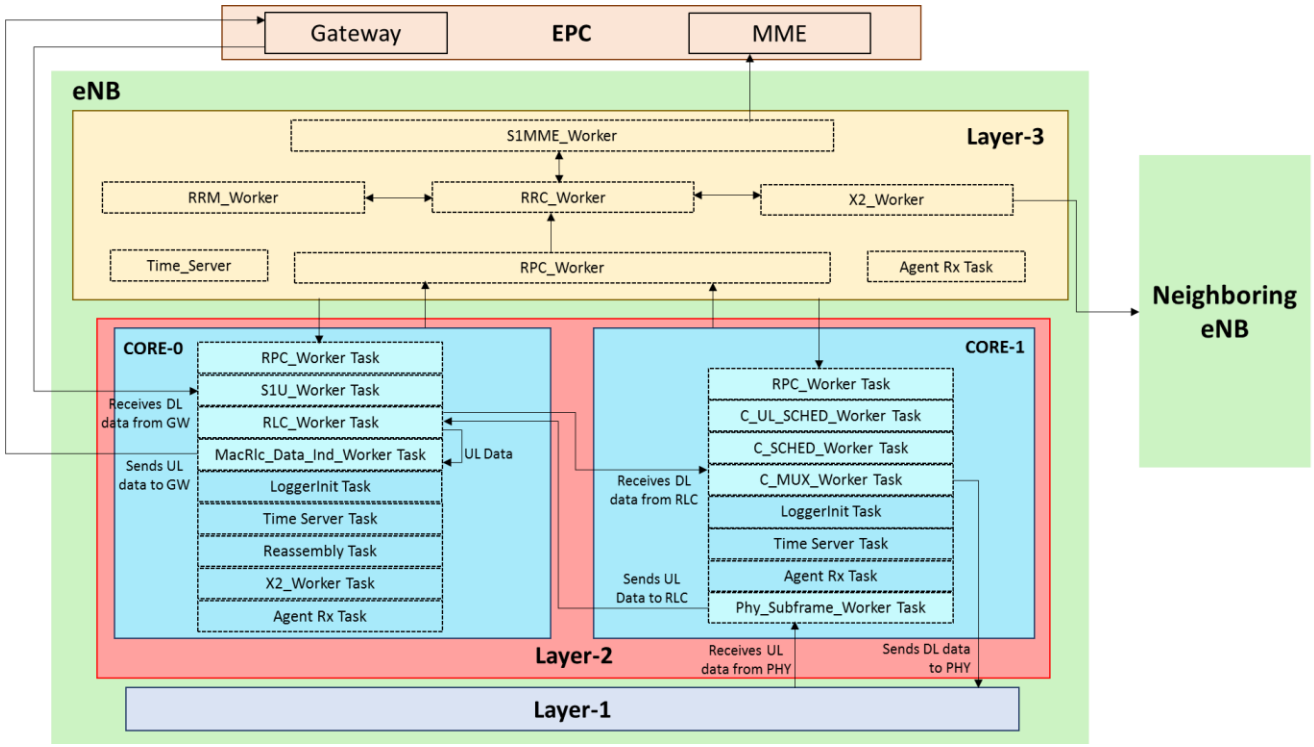


Figure 11 Task distribution for protocol stack layers

Following sections describe the distribution of tasks on DSP Cores and ARM processor for the protocol stack software.

6.1.1 Core 0

The functionality of protocol layers S1U, PDCP and RLC along with other features, is implemented through 9 tasks on DSP Core-0. Figure below depicts the thread architecture of Core-0.

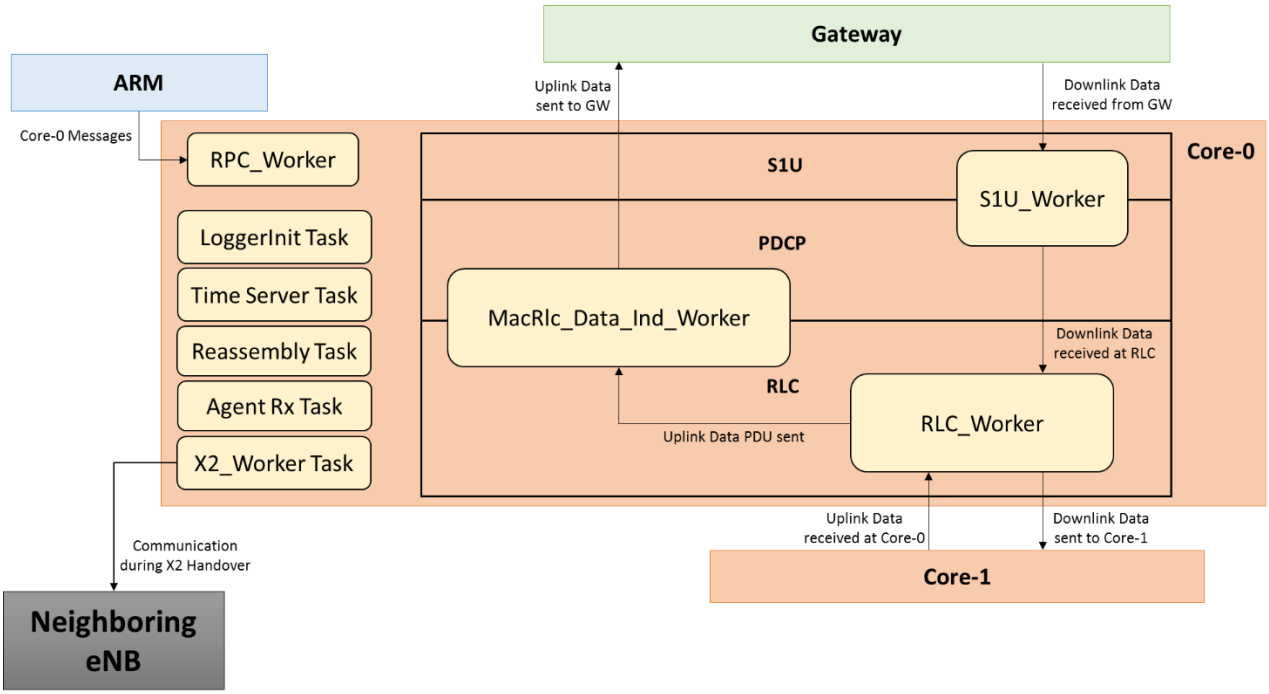


Figure 12 Thread Architecture on Core-0

6.1.1.1 Agent Rx Task

Priority: 4
Major functionalities:

- It is responsible for receiving named resource configuration pushed by ARM to communicate over MsgCom Qring channels.

6.1.1.2 Reassembly Task

Priority: 4
Major functionality:

- It reassembles the packets coming from NetCP.

6.1.1.3 LoggerInit Task

Priority: 1
Major functionality:

- It sends system logs as UDP packets to NetCP.

6.1.1.4 RLC Worker Task

Priority: 6
Major functionality:

- It receives DL data request from MAC, processes the DL data at RLC and sends it to MAC.
- It also receives UL data from MAC, copies it and sends an indication to the data indication worker task to execute the further procedure.
- It also receives the timer tick from Core-1 for time synchronization.

6.1.1.5 S1U Worker Task

Priority: 3
Major functionality:

- It receives DL data packets from gateway and passes them to PDCP.
- It also handles the DL data processing functionality of PDCP.

6.1.1.6 MacRlc_Data_Ind_Worker Task

Priority: 4
Major Functionality:

- It handles the functionality of UL data processing at RLC and PDCP and then sends it to gateway.

6.1.1.7 Time Server Task

Priority: 2
Major Functionality:

- The purpose of this task is to provide timer functionality. User of this task can register periodic and 'one shot' time based events. Once the registered event time is reached, this task triggers callback function that is registered with the event.

6.1.1.8 X2_Worker Task

Priority: 1
Major functionality:

- This task handles communication over X2-U interface.

6.1.1.9 RPC Worker Task

Priority: 5
Major functionality:

- This task receives messages from ARM over Msgcom Qring channel to facilitate the communication of Layer-3 with Layer-2.

6.1.2 Core 1

The functionality of protocol layer MAC and FAPI interface is implemented through 8 tasks on DSP Core-1. Figure below depicts the thread architecture of Core-1.

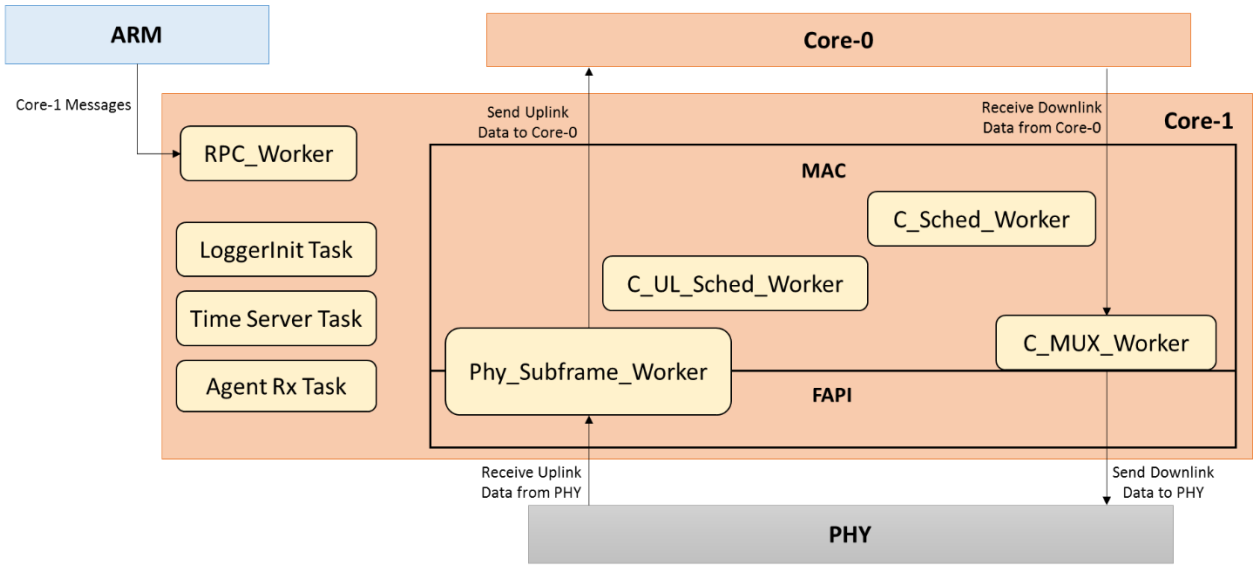


Figure 13 Thread Architecture on Core-1

6.1.2.1 Agent Rx Task

Priority: 4
Major functionalities:

- It is responsible for receiving named resource configurationw pushed by ARM to communicate over MsgCom Qring channels.

6.1.2.2 LoggerInit Task

Priority: 1
Major functionality:

- It sends system logs as UDP packets to NetCP.

6.1.2.3 Time Server Task

Priority: 1
Major Functionality:

- The purpose of this task is to provide timer functionality. User of this task can register periodic and 'one shot' time based events. Once the registered event time is reached, this task triggers callback function that is registered with the event.

6.1.2.4 RPC Worker Task

Priority: 4
Major functionality:

- This task receives messages from ARM over Msgcom Qring channel to facilitate the communication of Layer-3 with Layer-2.

6.1.2.5 DL Scheduler Task

Priority: 5
Major Functionality:

- It takes all the Scheduling decisions of data/signal flow in downlink.

- It takes the Ack/Nack information from MAC and then takes scheduling decisions accordingly.
- Selects which UEs need to be scheduled in the next TTI.
- Performs RB allocation corresponding to each scheduling decision.
- It is responsible for prioritization of logical channels of a UE. SRB data is prioritized over DRB data.
- Responsible for prioritization of transport channels.

6.1.2.6 UL Scheduler Task

Priority: 5
Major Functionality:

- It takes all the Scheduling decisions of data/signal flow in uplink.
- It takes decisions for SRS, CSI, ACK/NACK and ULSCH in uplink.
- Selects which UEs need to be scheduled in the next TTI.
- Performs RB allocation corresponding to each scheduling decision.

6.1.2.7 MUX Worker Task

Priority: 5
Major Functionality:
UPLINK

- Receives scheduling decisions from UL scheduler.
- Prepares the parameters for DCI 0/3/3A and HI PDUs to be passed to FAPI at nth TTI.
- Prepares UL configuration after k sub-frames, where k is 4 in case of FDD and it depends on the configuration in case of TDD.

DOWNLINK

- Receives scheduling decisions from DL scheduler.
- Prepares the DCI PDUs.
- Requests for data from RLC and prepares data PDUs for the sub-frame.
- It multiplex UEs and LCs (Logical Channels) to prepare Physical Transport blocks.

6.1.2.8 Phy Subframe Worker Task

Priority: 4
Major Functionality:
This is corresponding to FAPI - the interface between layer 2 and layer 1 (Physical Layer). It will take indications from Physical Layer and send it to MAC via callback functions. There are four main callback functions:

- RXCB: It is for ULSCH data, Rach indication, CQI and SRS indication.
- MSCB: It is for sub-frame indication.
- HARQ_CB: It is for HARQ feedback.
- MEAS_CB: It is for handling of Layer 1 measurements.

6.1.3 ARM

ARM constitutes 9 tasks and the functionality of whole Layer-3 (RRC and other management modules) is implemented through these tasks. Figure below depicts the thread architecture of ARM.

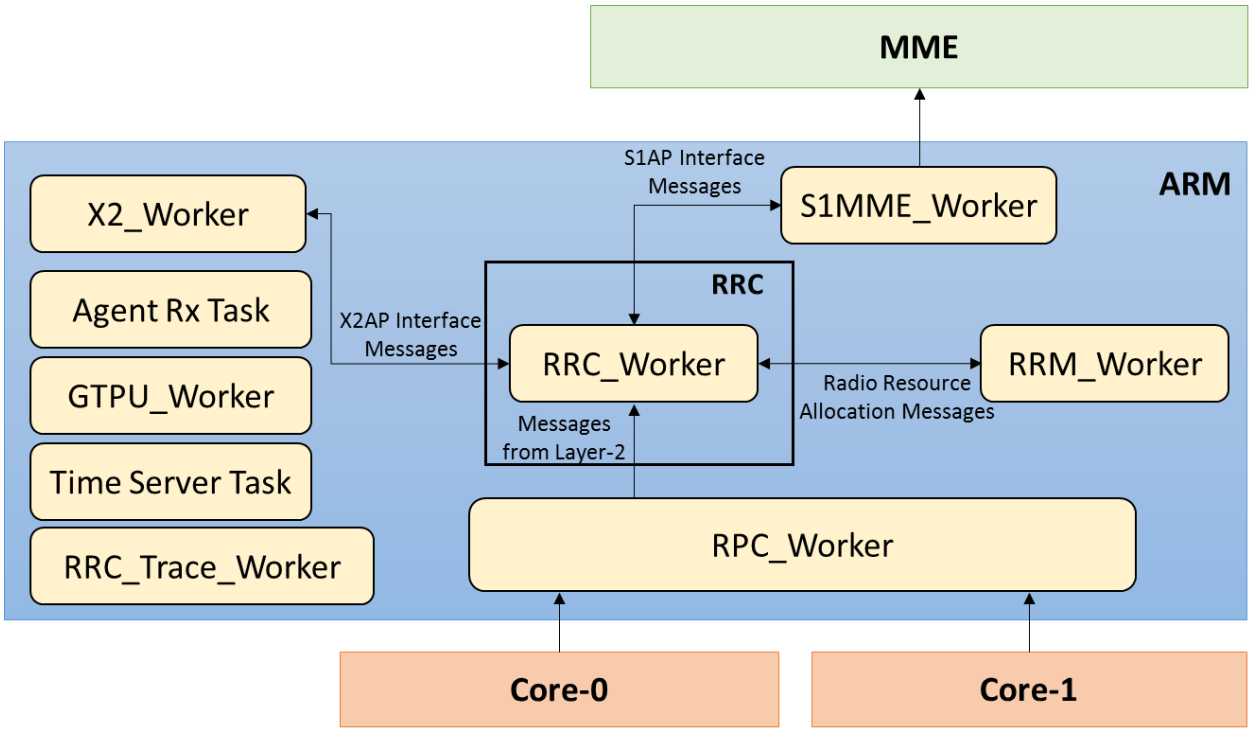


Figure 14 Thread Architecture on ARM

6.1.3.1 Time Server Task

Priority: 2

Major Functionality:

- The purpose of this task is to provide timer functionality. User of this task can register periodic and 'one shot' time based events. Once the registered event time is reached, this task triggers callback function that is registered with the event.

6.1.3.2 RRM Worker

Priority: 5

Major Functionality:

- This task provides Radio Resource Management (RRM) functionality on Layer-3. It receives messages from RRC layer, allocates the radio resources, if available and sends the notification back to RRC layer.

6.1.3.3 RPC Worker

Priority: 3

Major Functionality:

- This task receives messages from Layer-2 and executes the corresponding callback functions.

6.1.3.4 RRC Worker

Priority: 5

Major Functionality:

- This task provides the RRC layer functionality.
- It handles RRC/NAS messages for the eNB.
- It creates and maintains the UE contexts for each UE admitted in the system.
- It communicates with RRM for radio resource allocations.
- It also interacts with other entities on Layer-3 for message communication.

6.1.3.5 S1MME Worker

Priority: 5

Major Functionality:

- It handles the functionality of S1AP interface and communicates with MME.
- It maintains an object of the S1MME class and sends/receives messages from MME.

6.1.3.6 GTPU Worker

Priority: 1

Major Functionality:

- It is used to communicate with gateway/other eNBs via GTPU.

6.1.3.7 X2 Worker

Priority: 5

Major Functionality:

- It works on the functionality of X2AP interface and communicates with neighboring eNodeB.

6.1.3.8 RRC Trace Worker

Priority: 3

Major Functionality:

- It helps to receive and print debug traces on ARM.

6.1.3.9 Agent Rx Task

Priority:

Major Functionality:

- It is responsible for receiving named resource configuration on ARM as pushed by any of the DSP Cores to communicate over MsgCom Qring channels.

6.2 Call Flow

Following sections describe the signaling and data flow between eNB and UE.

6.2.1 Packet Handling

Each data packet received at the eNB is first encapsulated as a PDU and then processed. These PDUs are stored and managed using data structure *ref_data_unit* which is similar for both the data directions, i.e. Uplink and Downlink.
For efficiency, a zero-copy approach is followed where each layer creates a virtual PDU, which encapsulates pointers of lower/higher layer PDUs in case of uplink and downlink respectively.

Each virtual PDU also maintains a reference counter value, which indicates how many entities are using the PDU. The reference counter is initialized to one when a PDU is created and incremented whenever the PDU is passed to another entity. Each entity upon completion of the use of the PDU, decrements the reference counter. The PDU is deleted finally when the reference counter becomes equal to zero.
The functionality of virtual PDUs is encapsulated in a class *ref_data_unit*. Following are the main APIs provided to handle a PDU:

- **alloc() :** To allocate the memory to a PDU and initialize the reference counter
- **produce_virtual_copy() :** Increments the reference counter first for current PDU and then recursively for each contained virtual PDU
- **discard() :** To decrement the reference counter first for current PDU and then recursively for each contained virtual PDU. It also deletes the PDU, if reference counter is zero
- **get_reference_counter() :** To get the value of reference counter of the current virtual PDU

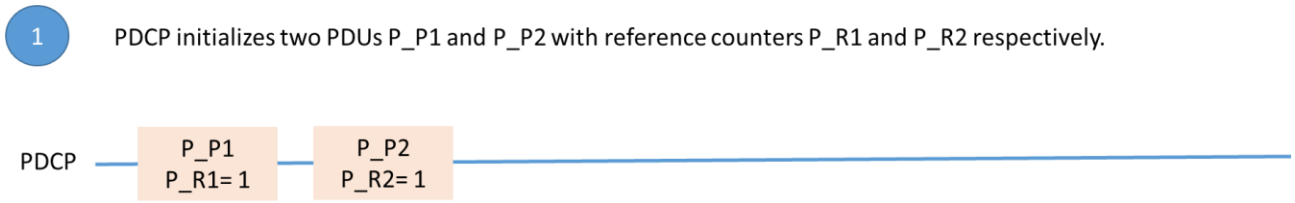
The approach for the data handling at eNB is described in following sub-sections.

6.2.1.1 PDU discard Procedure

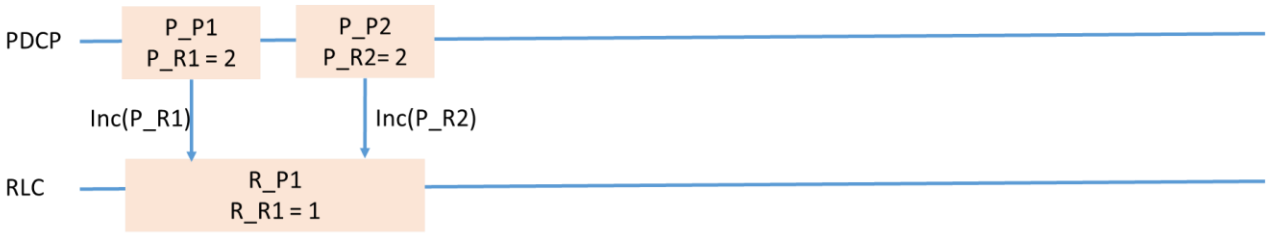
6.2.1.1.1 Downlink Data

Each layer creates a virtual PDU, which encapsulates pointers of the higher layer PDUs. The discard of the DL data PDUs at RLC and MAC is maintained separately. Lower layer, i.e. MAC discards its own PDU depending on the ACK or NACK received from the UE while higher layers, i.e. RLC and PDCP exercise a different discard procedure for their PDUs. For this mechanism a time window is implemented at RLC by maintaining an array of list of RLC PDUs. Each index of this array corresponds to a timestamp value and the list of RLC PDUs attached them contains the PDUs that were allocated at this timestamp. Every millisecond, the PDUs which were inserted the specified time window ago are deleted, provided they have been marked for discard (if their ACKs have been received). If a PDU has not been marked to be discarded, it is pushed back to the same location and would be checked again after the declared time window.

Following figures describe the implementation of reference counter in the PDUs and the discard procedure at RLC.

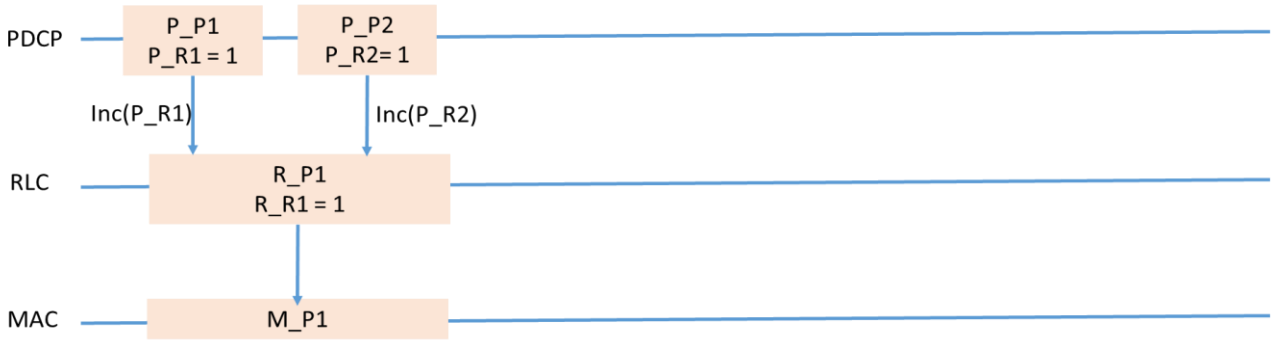


2 RLC creates a PDU R_P1 with reference counter R_R1 from two PDCP PDUs



3 RLC then discards the two SDUs

4 MAC creates a PDU (M_P1) from the RLC PDU and stores it in its HARQ buffer



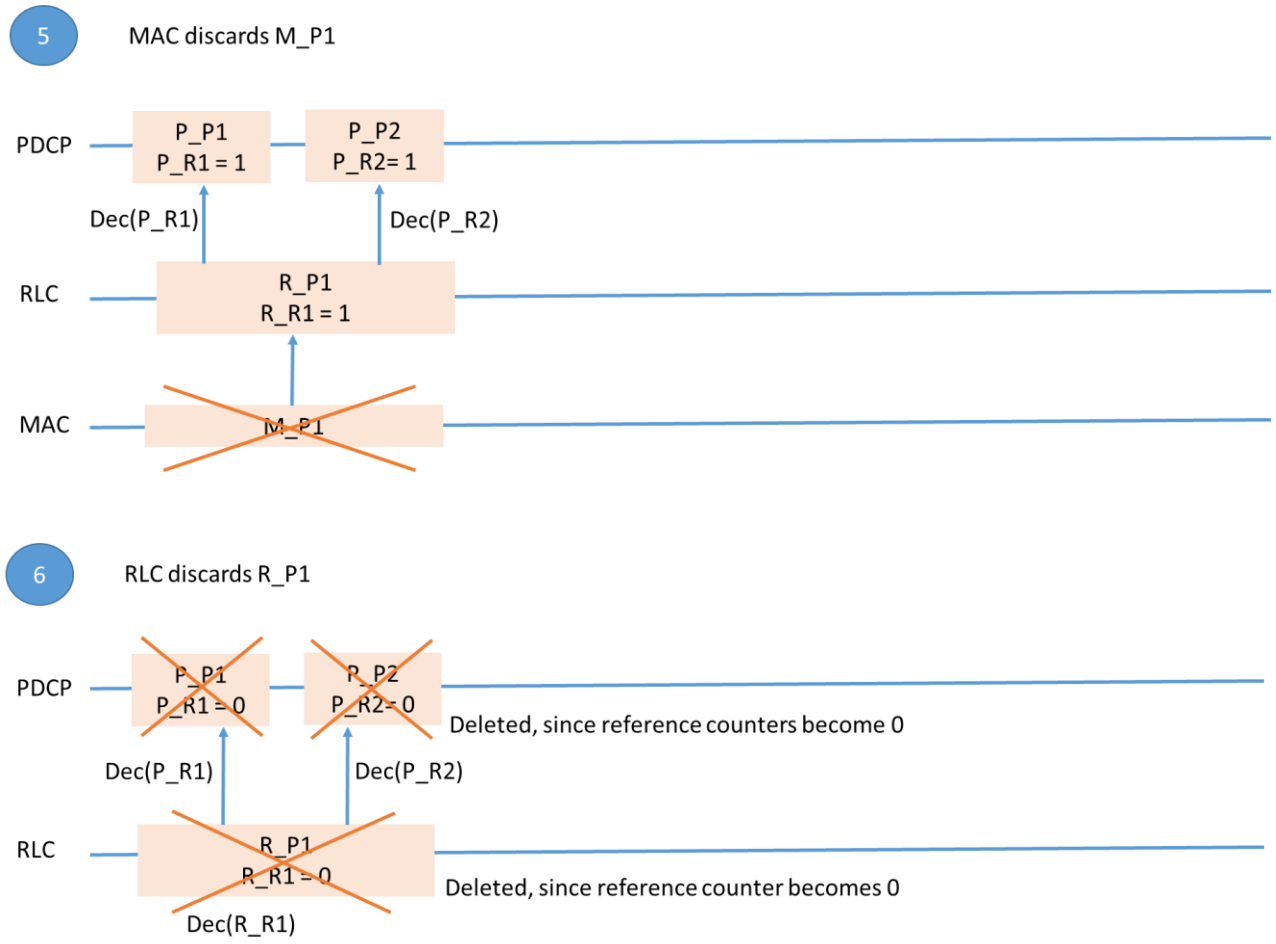


Figure 15 PDU Management in Dowlink

6.2.1.2 Uplink Data

Lower layer receives the UL data and copies it into a local memory location. It then allocates a transport block (tb) that contains a pointer to the memory location of UL data. Higher layers create a virtual PDU, which encapsulates this tb and respective offset provided by the corresponding lower layer.
The discard procedure for UL data PDUs is same as in DL data PDUs using reference counter.

6.2.2 Packet Flow

6.2.2.1 Signaling Flow

The information signaled across SRBs are relayed from Layer 3 (on ARM) to Layer 2 (on DSP) which then further transmitted to UE via lower layers. *Msgcom qring channels* provided by SYSLIB, based on infrastructure DMA are used for Layer3-Layer2 (i.e. ARM-DSP) communication. Once PDCP receives this data, PDCP starts functioning it i.e. apply integrity protection, if required and pass it to lower layers.
The integrity application/verfication procedure is offloaded to hardware, done by NETCP. PDCP passes the data to NETCP and receives the encoded/decoded data via dedicated CPPI queues, similar for uplink and downlink. The signaling data PDUs are handled similar to data PDUs, describe in section 6.2.1. Following figure depicts the call flow of signaling data including integrity application/verification procedure via lower layers.

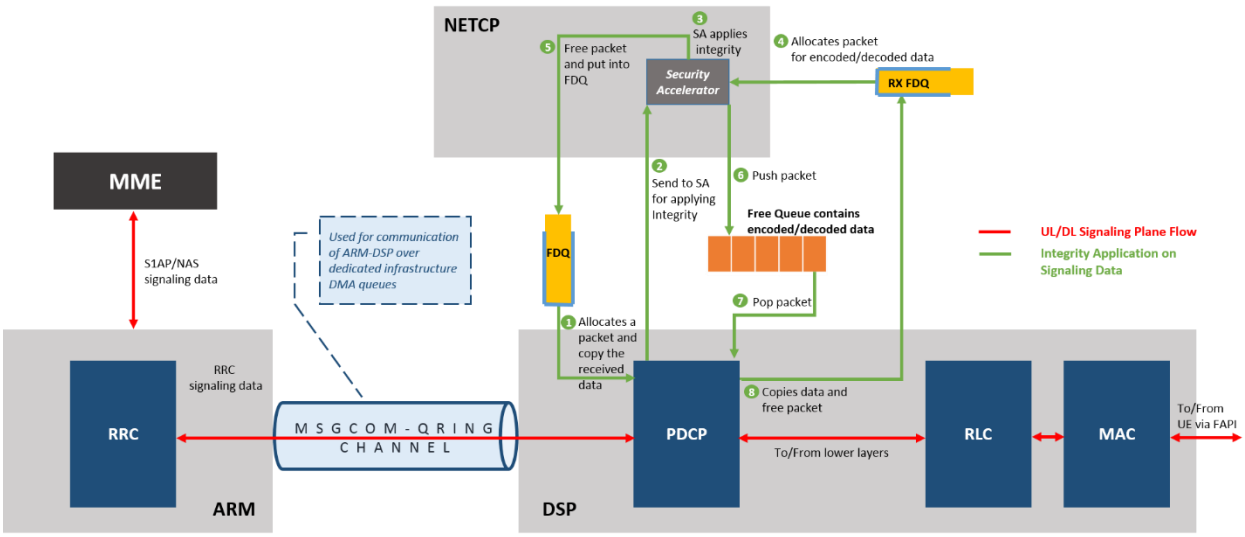


Figure 16 Signaling Data Flow

6.2.2.2 Data Flow

6.2.2.2.1 Downlink Data

The downlink data packets are first received by NETCP from the Gateway and then passed to the lower layers to be transmitted to the UE. NETCP contains a Packet Acceleartor (PA) that handles the packet functionality on board. It also stores the common context of the UE and its bearers to map the tunnels with gateway. The packet received from gateway is passed to PA to be decoded and sorted according to its tunnel id.
Secrity Accelerator (SA) is another component of NETCP that performs the security operations (ciphering and integrity protection) on data. Data packets from PA are sent to SA for ciphering if it is enabled for that particular UE and bearer. PA receives the ciphered data packets and sends them to PDCP using *Msgcom Channels* provided by SYSLIB. The DL data packet is then handled by eNB and sent to UE.

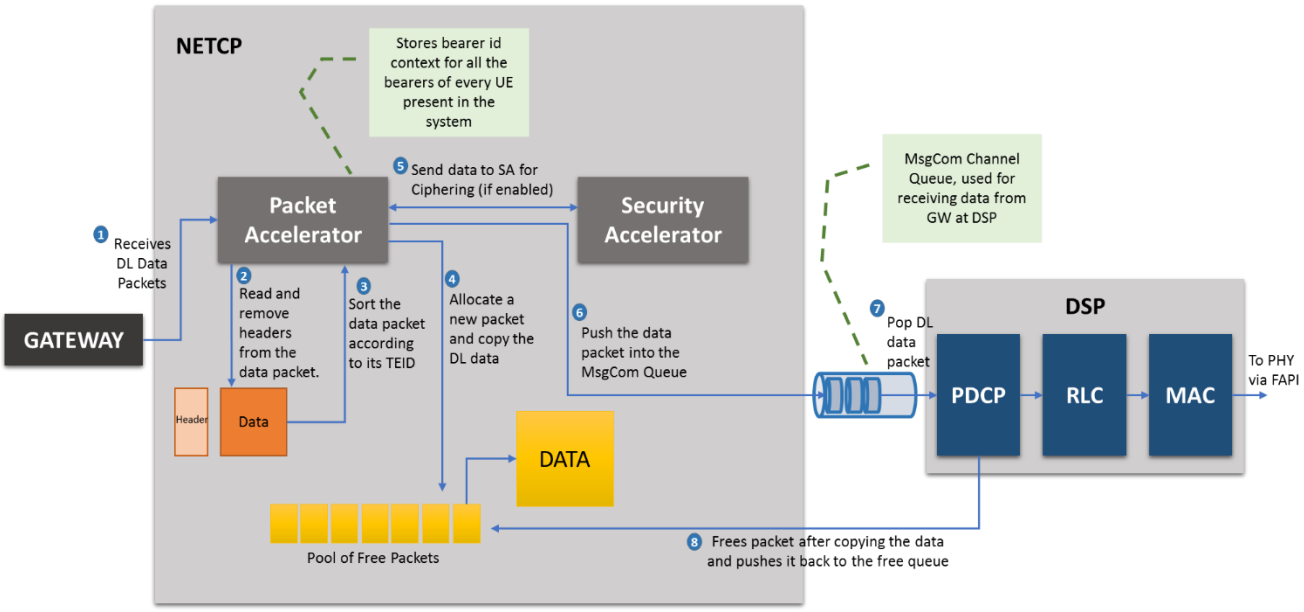


Figure 17 Downlink Data Flow

6.2.2.2.2 Uplink Data

UL data receieved at DSP Core-1 is decoded and then transferred to Core-0 using Msgcom Channel. This data is copied at Core-0and processed at upper layers. Then PDCP allocates a new packet and links the data with it. If the UL data is ciphered it is first deciphered at PDCP. For this, it is sent to NETCP and the deciphering operation is performed by SA. The deciphered data packet is pushed into a queue by SA and popped by PDCP. The packet is further sent to NETCP to be forwarded to Gateway.

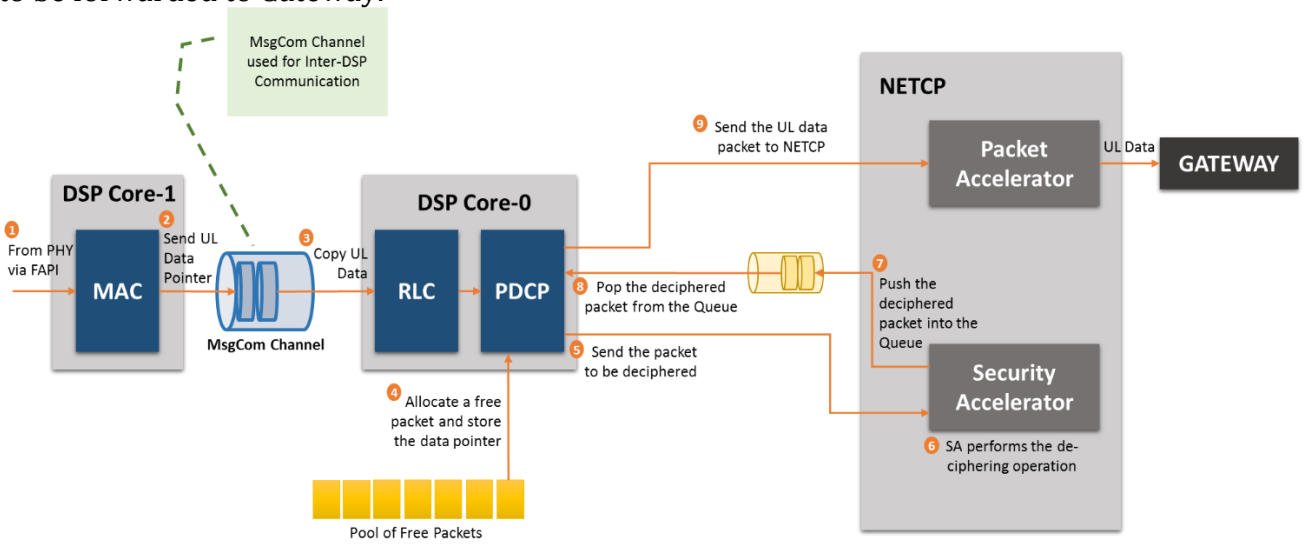


Figure 18 Uplink Data Flow

6.2.3 Task-Wise Flow

The flow of data through Layer-2 is distributed under certain tasks. Every task handles the sub-layer functionality for DL or UL. It is further described in following sub-sections.

6.2.3.1 Downlink Data

Following figure depicts the task wise flow of DL data.

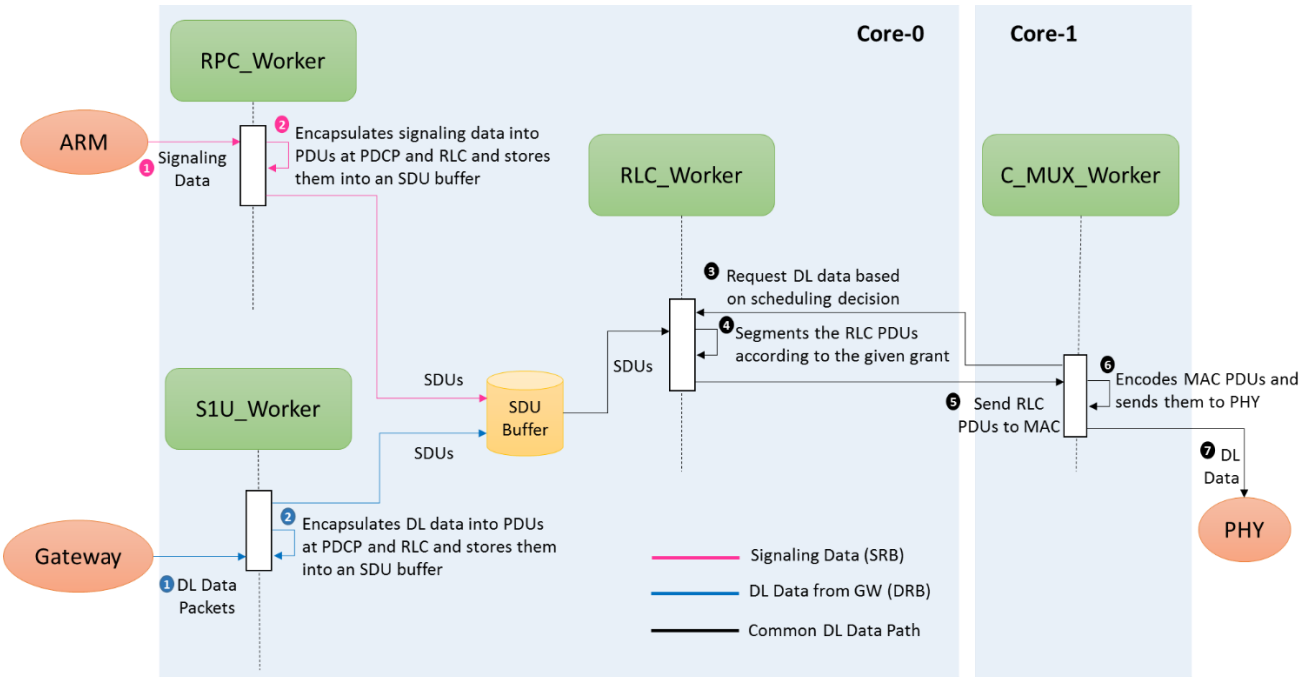


Figure 19 Task Wise Flow of DL Data

DL data received at PDCP can be both for SRB or DRB.

- Signaling data coming from RRC is received by *RPC Worker* on DSP Core-0. The functionality of PDCP DL and RLC DL for SRB data is handled by this task.
- Downlink data coming from Gateway is received by *S1U Worker* on DSP Core-0. The functionality of PDCP DL and RLC DL for DRB data runs under this task.

Any data when received at PDCP is first copied and then encapsulated into a PDU. This PDU is processed at PDCP and then passed to RLC. RLC SDU are stored in SDU buffer that is maintained separately for each UE present. The capacity of this buffer (buffer occupancy) is updated at MAC. When a scheduling decision is made at MAC, it requests the DL data from RLC. *RLC Worker* then pops the SDUs from the SDU buffer and segmentizes them into a single PDU of size granted by MAC. It handles the other half of the RLC DL functionality and sends the RLC PDUs to MAC. These RLC PDUs are received by *MUX Worker* that forms MAC PDUs, encodes them and sends them to PHY.

6.2.3.2 Uplink Data

Following figure describes the UL data flow on Layer-2 in detail and the tasks included in the process.

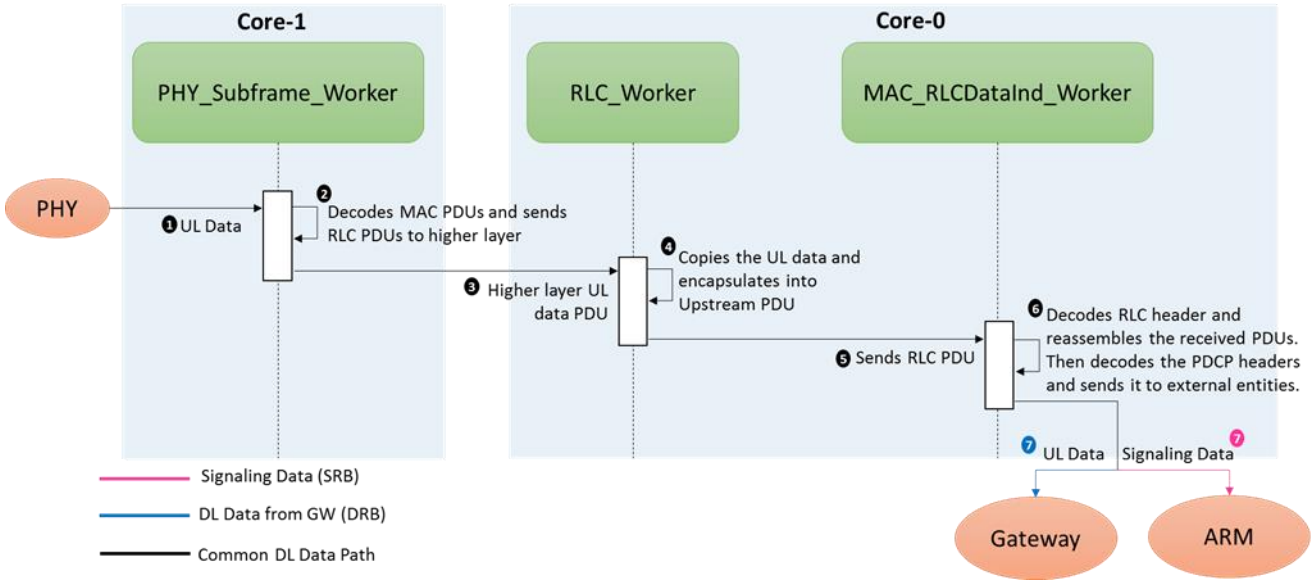


Figure 20 Task-Wise Flow of UL Data

The uplink data coming from UE is received at PHY and then passed to upper layers. *PHY Subframe Worker* at DSP Core-1 receives the UL data from PHY and handles the functionality of MAC UL. It decodes the MAC PDU received and sends the RLC PDU to the higher layers. This PDU is received by *RLC Worker* on DSP Core-0 which copies the data and encapsulates it into an Upstream PDU. This PDU is sent to *MAC RLC Data Ind Worker* that handles the functionality of RLC UL and PDCP UL. The UL data for both SRB and DRB is processed under this task and sent to the corresponding external entities, i.e. ARM or Gateway.

7 References

[1] TI SYS/BIOS user guide
[2] TI Multicore Navigator (CPPI) for Keystone Architecture user guide
[3] TI Syslib user guide