Azcom eNodeB Protocol Stack Debugging and Tracing

User Guide

This document describes the facilities provided to inspect and log eNB software functioning and internal events.

Document Revision: 1.0
Date: January 28th, 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

Acronyms

UDP	User Datagram Protocol
ETB	Embedded Trace Buffer
eNB	eNodeB
RTOS	Real Time Operating System
gdb	GNU Debugger
CCS	Code Composer studio

References

[1] http://processors.wiki.ti.com/index.php/Embedded_Trace_Buffer#What_is_it.3F_What_does_it_do.3F

CONTENTS

LIST OF TABLES

1 Scope

This document describes the facilities provided to inspect and log eNB software functioning and internal events. This document is intended for the Azcom eNB Protocol Stack team, test team and customers.

2 Introduction

A number of facilities are provided to the eNB developer and the eNB testers to inspect and log eNB software functioning and internal events. The UDP-based tracing system, the Embedded Trace Buffer, the Error Handler and the Exception Handler.

The **UDP-based tracing system** is the foundation of the debugging facilities, as it is used to convey to the external world any system information. It is invoked on purpose by the application layer developer to trace some system event and is also automatically invoked by the DSP and ARM software when some information has to be traced.

The **Embedded Trace Buffer** is utilized to analyse the real-time behaviour of DSP software with respect to system timings, context switches, order of execution, various latencies and execution times of different functions.

The **Exception Handler** is a low-level tracing system which captures the basic system parameters when a SYS/BIOS RTOS exception happens.

3 UDP based Logging Facility

To trace special events or interesting data to an external log collector, dedicated functions are implemented and are made available to application layer developer. These functions will send the tracing data to the external world via UDP over IP connection, through the backhaul Ethernet port of the BBU.
These functions are able to trace two kinds of information, in text format and in binary format. To add logs in text format an API, TRACE_L2L3(), is provided whereas TRACE_L2L3_HEX() is used to trace data dumps in binary format.
The user has the option to selectively enable/disable traces pertaining to a channel or a group of channels by enabling/disabling the log mask corresponding to that channel. The user can also control the tracing message by controlling the tracing levels/severity in a dynamic manner.

3.1 Log Level/Severity

Depending upon the severity of logs, eNB software requires one of the five different logging levels to be set for each UDP log.

```
typedef enum
{
   TRACE_CRITICAL = 0, /**< Critical error, System needed to reboot. Mostly Platform errors */
   TRACE_ERROR,        /**< Application error occurred, System is recoverable*/
   TRACE_WARNING,      /**< Warning occurred, normal functioning path is not taken */
   TRACE_INFO,         /**< All Traces need to be logged*/
   TRACE_EVENT         /**< Some System event occurred*/
} tTraceLevel;
```

Traces will be sent if the logs' tracing level is at equal or higher severity than core's tracing level. This implies all logs with TRACE_INFO and TRACE_EVENT will not be communicated if core's tracing severity is TRACE_WARNING or higher. Such variables can be modified at runtime and the target subsystem would be periodically updating this value in a periodic interval of 1 second.

3.2 Log Masks

In addition to Log Levels, a 64-bit variable, the Logging Bitmask, is defined per subsystem to control the outflow of tracing logs. Logging bitmask provides the ability to filter logs of a desired channel/module[s]. e.g if someone wants to debug scheduler, logging mask specific to scheduler can be enabled at the run time which can be used to get only scheduler logs. Logging for multiple modules can be enabled simultaneously. Following filters are available for debugging.

Table 1 Layer 2 log Mask

Channel	Log Mask
TOShexdump	0x000000001
TOSflow	0x000000002
TOS_config	0x000000004
TOS_ctrlsap	0x000000008
TOS_datasap	0x000000010
TOS_internal	0x000000020

TOSmac_config	0x000000040
TOSmac_ctrlsap	0x000000080
TOSmac_datasap	0x000000100
TOSmac_internal	0x000000200
TOSmac_sched	0x20000000000000
TOSmac_range	(TOSmac_config | TOSmac_ctrlsap |TOSmac_datasap | TOSmac_internal | TOSmac_sched)
TOSrlc_config	0x000000400
TOSrlc_ctrlsap	0x000000800
TOSrlc_datasap	0x000001000
TOSrrc_conn	0x000002000
TOSrlc_amdebug	(TOS_datasap | TOSmac_config | TOSrlc_config)
TOSrlc_internal	0x000000001
TOSrlc_range	(TOSrlc_config | TOSrlc_ctrlsap | TOSrlc_datasap | TOSrrc_conn)
TOSpdcp_config	0x000004000
TOSpdcp_ctrlsap	0x000008000
TOSpdcp_datasap	0x000010000
TOSpdcp_internal	0x000020000
TOSpdcp_range	(TOSpdcp_config | TOSpdcp_ctrlsap | TOSpdcp_datasap | TOSpdcp_internal)
TOSs1u_config	0x000040000
TOSs1u_ctrlsap	0x000080000
TOSs1u_datasap	0x000100000
TOSs1u_internal	0x000200000
TOSs1u_range	(TOSs1u_config | TOSs1u_ctrlsap | TOSs1u_datasap | TOSs1u_internal)
TOSsch_config	0x000400000
TOSsch_ctrlsap	0x000800000
TOSsch_datasap	0x001000000
TOSsch_internal	0x002000000
TOSsch_range	(TOSsch_config | TOSsch_ctrlsap | TOSsch_datasap | TOSsch_internal)
TOS_api	0x004000000
TOS_int	0x000000001
TOS_MIB_SIB_CCCH	0x008000000
TOSmac_debug	0x008000000
TOS_integ	0x000000001
TOSdump	0x020000000
TOSrtci	0x040000000
TOS_phy	0x080000000
TOSwarnings	0x100000000000
TOSwarning	0x000000001

TOS_fapi	0x000000001
TOSfapi_internal	0x000000001
TOSrrc	0x200000000000
TOS_profile	0x000000001
TOSharq_debug	0x80000000000000
TOSrach_debug	0x4000000000000000
TOS_pdu	0x40000000000
TOSarq_debug	0x80000000000
TOSipc	0x100000000000
TOSsat	0x200000000000
TOStime	0x400000000000
TOSstartup	0x800000000000
TOS_meas	0x2000000000000
ConsoleEnabled	0x2000000000000000
MscEnabled	0x4000000000000000
TOSProfiling	0x8000000000000000
TOSbits	(TOSrrm | ConsoleEnabled | MscEnabled | TOSProfiling)
TOSchannels	0x000000001
TOS_MUX	0x000000001
TOS_DLSHED	0x000000002
TOS_ULSHED	0x000000004
TOS_MACSAP	0x000000008
TOS_FAPI	0x000000010
TOS_MAC	(TOS_FAPI | TOS_MACSAP | TOS_ULSHED | TOS_DLSHED | TOS_MUX)
TOS_RLCDL	0x000000020
TOS_RLCUL	0x000000040
TOS_RLCSAP	0x000000080
TOS_RLC	(TOS_RLCSAP | TOS_RLCUL | TOS_RLCDL)
TOS_PDCP_DL	0x000000100
TOS_PDCP_UL	0x000000200
TOS_PDCP_SAP	0x000000400
TOS_PDCP	(TOS_PDCP_SAP | TOS_PDCP_DL | TOS_PDCP_UL)
TOS_DLSHED_DEBUG	0x200000000
TOS_ULSHED_DEBUG	0x400000000
TOS_MACSAP_DEBUG	0x800000000
TOS_MUX_DEBUG	0x100000000
TOS_FAPI_DEBUG	0x1000000000
TOS_MAC_DEBUG	(TOS_FAPI_DEBUG | TOS_MACSAP_DEBUG | TOS_ULSHED_DEBUG | TOS_DLSHED_DEBUG | TOS_MUX_DEBUG)
TOS_RLCDL_DEBUG	0x2000000000

TOS_RLCUL_DEBUG	0x4000000000
TOS_RLCSAP_DEBUG	0x8000000000
TOS_RLC_DEBUG	(TOS_RLCSAP_DEBUG | TOS_RLCUL_DEBUG | TOS_RLCDL_DEBUG)
TOS_PDCP_DL_DEBUG	0x10000000000
TOS_PDCP_UL_DEBUG	0x20000000000
TOS_PDCP_SAP_DEBUG	0x40000000000
TOS_PDCP_DEBUG	(TOS_PDCP_SAP_DEBUG | TOS_PDCP_DL_DEBUG | TOS_PDCP_UL_DEBUG)
TOS_MEMORY_DEBUG	0x80000000000
TOS_RPC	0x100000000000

Table 2 Layer 3 Log Masks

Channel	Log Mask
TOShexdump	0x000000001
TOSflow	0x000000002
TOS_config	0x000000004
TOS_ctrlsap	0x000000008
TOS_internal	0x000000020
TOSmac_config	0x000000040
TOSmac_ctrlsap	0x000000080
TOSmac_datasap	0x000000100
TOSmac_internal	0x000000200
TOSmac_sched	0x20000000000000
TOSmac_range	(TOSmac_config | TOSmac_ctrlsap | TOSmac_datasap | TOSmac_internal | TOSmac_sched)
TOSrrc_conn	0x000002000
TOSs1u_config	0x000040000
TOSs1u_ctrlsap	0x000080000
TOSs1u_datasap	0x000100000
TOSs1u_internal	0x000200000
TOSs1u_range	(TOSs1u_config | TOSs1u_ctrlsap | TOSs1u_datasap | TOSs1u_internal)
TOS_api	0x004000000
TOS_int	0x000000001
TOSdump	0x020000000
TOS_phy	0x080000000
TOSwarnings	0x100000000000
TOSwarning	0x000000001
TOSrrc	0x200000000000
TOSstartup	0x800000000000
TOSasn1c	0x1000000000000

TOSrrc_internal	0x1000000000000
TOS_meas	0x2000000000000
TOSrrc_si	0x4000000000000
TOSrpc	0x4000000000000
TOSrrc_decode	0x8000000000000
TOSrrc_encode	0x8000000000000
TOSrrc_range	(TOSrrc | TOSrrc_internal | TOSrrc_si | TOSrrc_decode | TOSrrc_encode)
TOSrrc_flow	0x10000000000000
TOSrrc_db	0x20000000000000
TOSrrcErroneousMsg	0x40000000000000
TOSrrc_range_cont	(TOSrrc_flow | TOSrrc_db | TOSrrcErroneousMsg)
TOSs1_ctrlsap	0x100000000000000
TOSx2_ctrlsap	0x200000000000000
TOSs1_internal	0x400000000000000
TOSx2_internal	0x800000000000000
TOSs1_x2_range	(TOSs1u_range | TOSs1_ctrlsap | TOSs1_internal | TOSx2_ctrlsap | TOSx2_internal)
TOSrrm	0x1000000000000000
ConsoleEnabled	0x2000000000000000
MscEnabled	0x4000000000000000
TOSProfiling	0x8000000000000000
TOSbits	(TOSrrm | ConsoleEnabled | MscEnabled | TOSProfiling)
TOSchannels	0x000000001
TOS_RRC	0x000000001
TOS_RRC_SAP	0x000000002
TOS_RRM	0x000000004
TOS_RRM_SAP	0x000000008
TOS_S1AP	0x000000010
TOS_S1AP_SAP	0x000000020
TOS_OAM	0x000000040
TOS_OAM_SAP	0x000000080
TOS_RRC_DEBUG	0x100000000
TOS_RRC_SAP_DEBUG	0x200000000
TOS_RRM_DEBUG	0x400000000
TOS_RRM_SAP_DEBUG	0x800000000
TOS_S1AP_DEBUG	0x1000000000
TOS_S1AP_SAP_DEBUG	0x2000000000
TOS_OAM_DEBUG	0x4000000000
TOS_OAM_SAP_DEBUG	0x8000000000
TOS_PLT_DEBUG	0x000000100

3.3 Log messages format

3.3.1 Text Message Format

This format is used by the developer to trace unique and atomic events that should clearly appear in the log file. This kind of trace monitors the regularly happening events in the eNB software and gives a clear indication that the basic functioning of the system is happening.
The format of the text log sent out of the system for textual data is:

[CoreId] [Trace Level] [64 Bit Timestamp] [File ID and Line Number] String ID

The string ID is mapped to a string by the log capturing application running on an external PC. The log capturing application requires a coreX_traceId.dat (X represents the DSP Core ID) file corresponding to each core, which keeps all the strings that were present in the code, before the build for the core was initiated. This coreX_traceId.dat file is created as a pre-build step for each core.
Similar to string ID, file ID is mapped to the file name by the capturing application, using a file named file_id_string.dat.
Once the capturing application, processes each log, the log format looks as follows:

[CoreId] [Trace Level] [64 Bit Timestamp] [File name and Line Number] Text String

For e.g, Following is a UDP trace from core 1

[C66xx_1] [INFO] [03045 996610585] [e_utra_mac_layer.cc : 1040] timeout, tti 470963 ul_buffer_info_queue 0 rach_msg2_queue 0

From the log, it is clear that

- Log has been generated by Core 1.
- Trace Level for this Log is INFO
- At timestamp of "03045 996610585" for this core.
- File generating the Log is e_utra_mac_layer.cc at line number is 1040
- The following phrase indicates a key periodic event in the software functioning, and has been added by the developer for such purpose.

3.3.2 Binary Message Format

This format is used by the developer to trace special events that deserve a deeper inspection of the related parameters and variables. This kind of trace monitors only hex data, which is compacted in a binary string. They are not of intuitive interpretation and usually need some specific information to be understood.
The format of the binary log is:

[CoreId] [64 Bit Timestamp] [Trace Level] Binary string

For e.g, Following is a UDP trace from core 1

[C66xx_1][00141 4130013587] [CRITICAL] 0x3b 0x3d 0x21 0x4 0x1f

From the log, it is clear that

- Log has been generated by Core 1.

- 00141 4130013587 is the timestamp for this core.
- Trace Level for this Log is CRITICAL.
- The following binary string is to be interpreted depending on the developer purpose.

4 Embedded Trace Buffer (ETB) Capturing

ETB is an on-chip circular memory buffer where the compressed trace information is stored. The size of the buffer is 4K bytes. Because of the compression, the user will get roughly 10k to 30k lines of program trace. This buffer operates as a circular buffer, continuously capturing trace information until the halted. Trace provides a detailed, historical account of application code execution, timing, and data accesses. This information is useful for finding bugs and performance analysis. Trace works in real-time and does not impact the execution of the system.
When the processing of a DSP is halted, the buffer is dump into DDR memory. This memory is then read by ARM, and sent out of the system in the form of UDP logs. The ETB dump can be written into a .txt file, converted to a .bin and eventually to a .tdf (trace data format) and displayed using CCS [1].

5 Exception generation and handling

5.1 DSP exception handling

Exception handling has been incorporated in L2 software to dump vital parameters on UDP traces whenever an exception is detected by SYSBIOS. This has been realised by hooking a function (PLT_checkSystemStatus) to exception module of SYSBIOS.

5.2 ARM (L3) exception handling

L3 is a multithreaded application developed in C++ which runs on ARM Linux. Exception handling on L3 is done by generating core dump file which can be used for analysis on the host PC. For this following compiler options are added in L3 makefiles.
'-g -ggdb'
This compiler option will help the binary generate a core dump file whenever an exception is raised. The core file can be debugged using gdb.