Debug and Trace Design Specifications

Revision 0.51, 21-Sept-2020

Revision History

Revision Number	Description	Revision Date	Author
0.1	Initial draft Version	1-July-2020	Han Pengtu
0.2	Trace and Debug Architecture diagram updates	7-July-2020	Han Pengtu
0.3	Add low power feature and monitoring	27-July-2020	Han Pengtu
0.5	SW part added	12-Aug-2020	He Kun
0.51	PHY_SS Trace and debug architecture updated	21-Sept-202	Han Pengtu

Contents

Figures

Tables

1 Introduction

1.1 Scope

This document covers the baseband system debug and trace design details of Mariana project. The capabilities and features of debug and trace are included from both HW and SW perspective.

1.2 Purpose

This document will be used as reference for subsystem engineers to know how to make an effective debug in case of bandstand behaviors abnormal.

1.3 Audience

The document is intended to the subsystem designers, verification engineers, and system engineers.

1.4 Open Topics

Those topics under discussions are listed as below:

- PHY Control debug and trace
- DP debug and trace
- 2G/3G/Audio debug and trace
- Traces source and master ID allocation
- Monitoring elements allocation and ID assignments

2 Debug and Trace System Overview

Debug is a system engineering and becoming more and more important with increasing SoC complicity. An effective and flexible debug solution can help quick detection error in silicon test resulting either from HW or SW. In general, a complete debug system is consisting of SW solution, HW implementation and debug tools, as showing in figure 1 below.

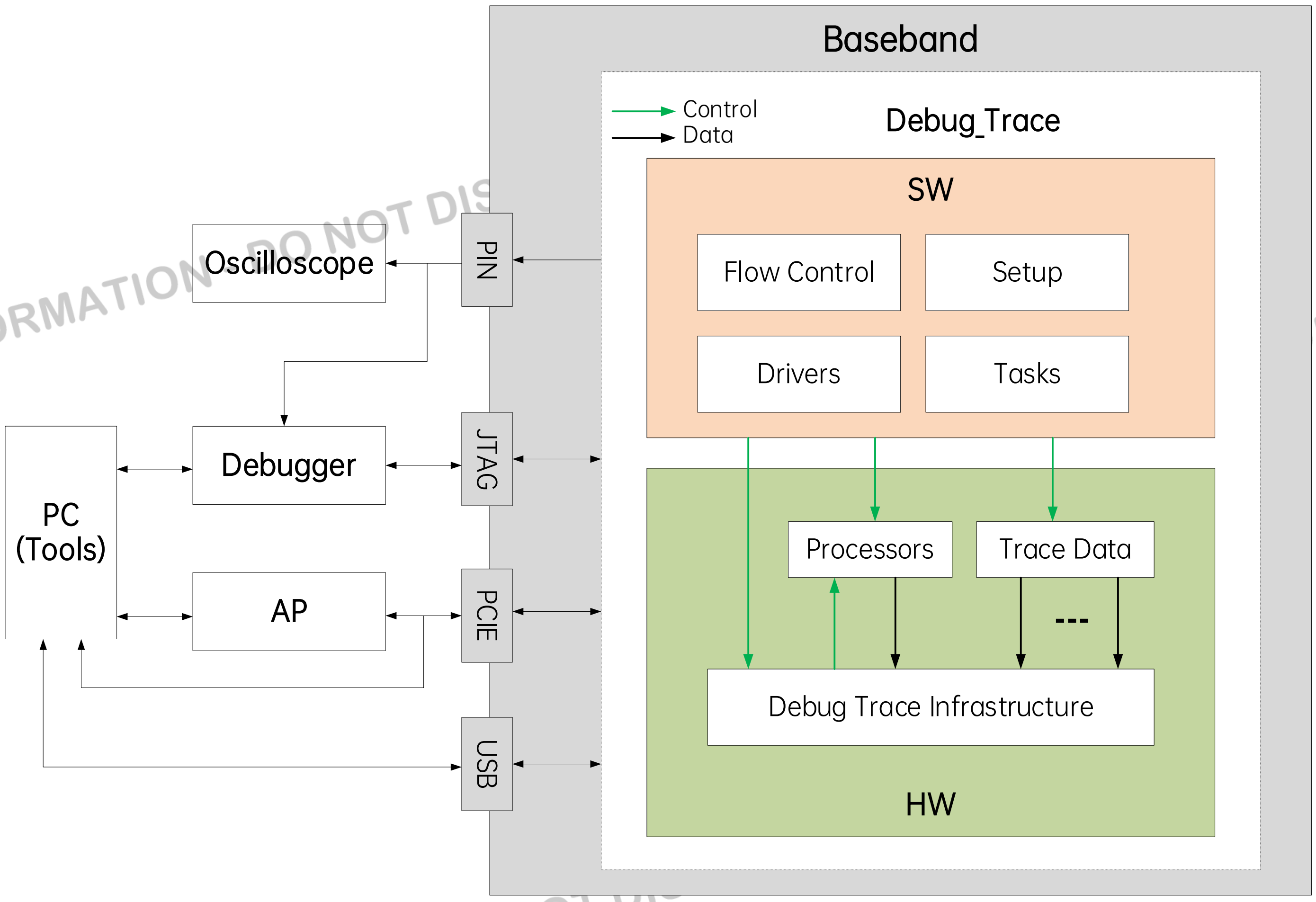

Figure 2-1 Debug and Trace System Overview

SW is responsible of overall trace settings and configurations:

- Trace source setup
- Trace path selection
- Trace sink selection
- Trace attributes
- Trigger timestamp assertion
- Trace module control: clock, power and reset

- Others

HW is consisting of all elements to provide on chip debug and message trace capabilities that is listed below, but not limited to.

- Debug access port
- Debug bus and decoder
- Multi-core debug break matrix
- Trace message formators
- Trace IF or bus
- Message on chip buffers or memories
- Timestamp generation
- On chip monitoring infrastructure
- Control registers
- Others

Debug tool include different devices, application SW and scripts. The common purpose and requirements for debug tools are listed in below.

- Provide debug command to make control of on chip cores or system
- Debug data process: collection, store, sort, filter, edition
- On chip signals or events observation
- Data or logic analysis
- Others

2.1 General Requirements

A debug system involves various functionalities and application requirements. It's mainly based on the specific projects demand. Based on the modem chip development and debug experiences, those features listed in table 1 should be take into consideration.

Table 2-1 Requirements List

Requirements	Descriptions
Debug IF	JTAG
Security	Debug and trace access should be under security control
Real time core debug	Individual or multi-core debug
Trace IF	USB, PCIe, TPIU or UART
Real time trace	SW or HWA generated data, or profiling
Big trace data	I/Q and LLR
Trace on chip store	In local buffer, RAM or DDR in case of no export connected
Trace message type	String, Internal/HMCC Message, Debug trace, Trace data etc.
Message filter	Dynamically or statically: single or multi-level
Monitoring	Signals or events observation, events counter or statistics
System data dump	Cores or RAMs data dump, or register values dump in case of system crash
Timestamp	Unified timestamp can be inserted to the message in trace
Time synchronization	Timers to be synchronized between different timer and different subsystems
Configuration	Dynamically or store in the NVM
Encode and decode	Message encoded in STP format, decoded by debugger or trace tools
Performance	High trace BW, and Fast: load, decoding, search, filter, run etc.

2.2 Debug Tools and Applications

For assistant the debug access, debug tools need to be available in different application cases.

In lab condition:

- Lauterbach Debugger
- Lauterbach Trace32

- Debug PC
- USB protocol analyzer
- Oscilloscope
- Scripts and others

In the field test condition:

- Congatec Trace Box
- Debug PC
- Scripts and others

2.3 HW Architecture Overview

The debug and trace HW architecture overview is showing in below figure 2. The modules highlighted in green are dedicated debug and trace elements. Others are SoC infrastructures.

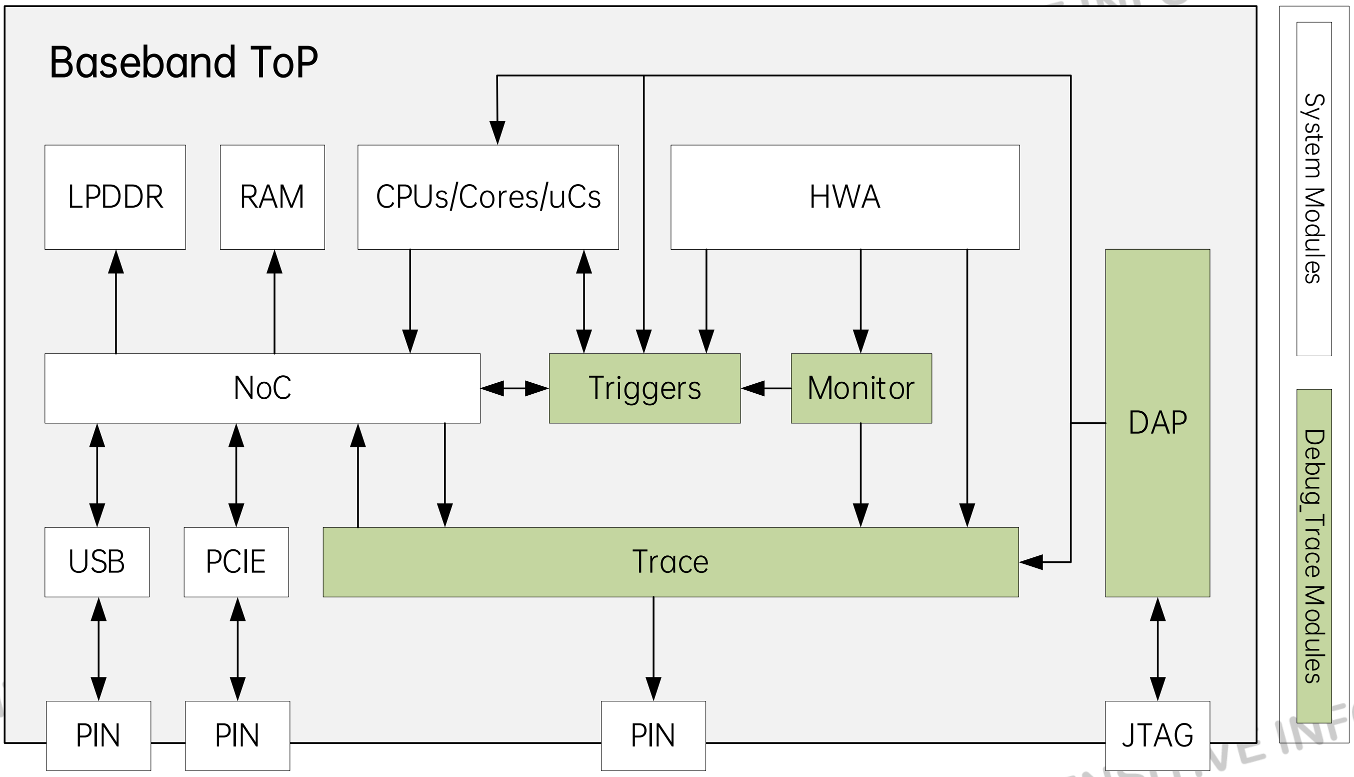

Figure 2-2 Debug and Trace HW Architecture Overview

DAP is the global debug access port to provided different debug IF to different cores via a debugger. It is also the central configuration module for all Coresight components.

On chip important signals or events are collected by the monitor modules. It's scalar able and flexible to connect and select any signals in theory. In practice, the signals interested and to be connected to the monitor system up to a module or subsystem designer's decision. The final selected signals can be feed into debug break triggers, or output from pad directly or after STM formation.

Triggers are those break in and break out signals from each core, from external request, and from selected on chip events. A specific trigger in can be assigned to trigger a selected core into debug mode via configuration.

Trace module is the function block to format the trace message into STP formation, timestamp assertion, and route the trace message to a selected sink.

Beside the dedicated debug and trace modules mentioned above, those SoC elements are essential to fulfill the trace functionality. Big payload data need to be stored in local SRAM or external DRAM first, then emptied by PCIE or USB to an external debugger or tool.

3 Debug HW Solution and Functionalities

All cores and DSP in the baseband will be connected to the debug system. They are accessible from external debuggers. The debug features are different from cores, but the general requirements are:

- Controllable via a debugger in debug mode
- Enter debug mode via a break point or break request
- Hosted debug capabilities
- Performance or instruction execution monitoring
- Debug access is under security control
- Debug access can be enabled/disabled from functional or test control registers

Debug capabilities of critical system abnormal behaviors need to be considered. It needs the effective HW solution to detection those error cases, listed as below.

- System stuck
- System crash
- Unexpected abortion
- Error or warning reported
- others

3.1 Debug Architecture Overview

Based on Mariana project requirements, debug capability and access need to be provided to several subsystems:

- Baseband BPP cluster
- Baseband BCP cluster
- PHY control subsystem
- DP subsystem
- PHY subsystem
- DFEC_BB subsystem
- Legacy RAT in potential: 2G/3G/4G/Audio

The overall debug HW architecture is showing in below figure 3. DAP is the universal debug access control to all cores and Coresight components. The Cross Triggers is the multi-core break matrix in which the triggers are from cores and system events.

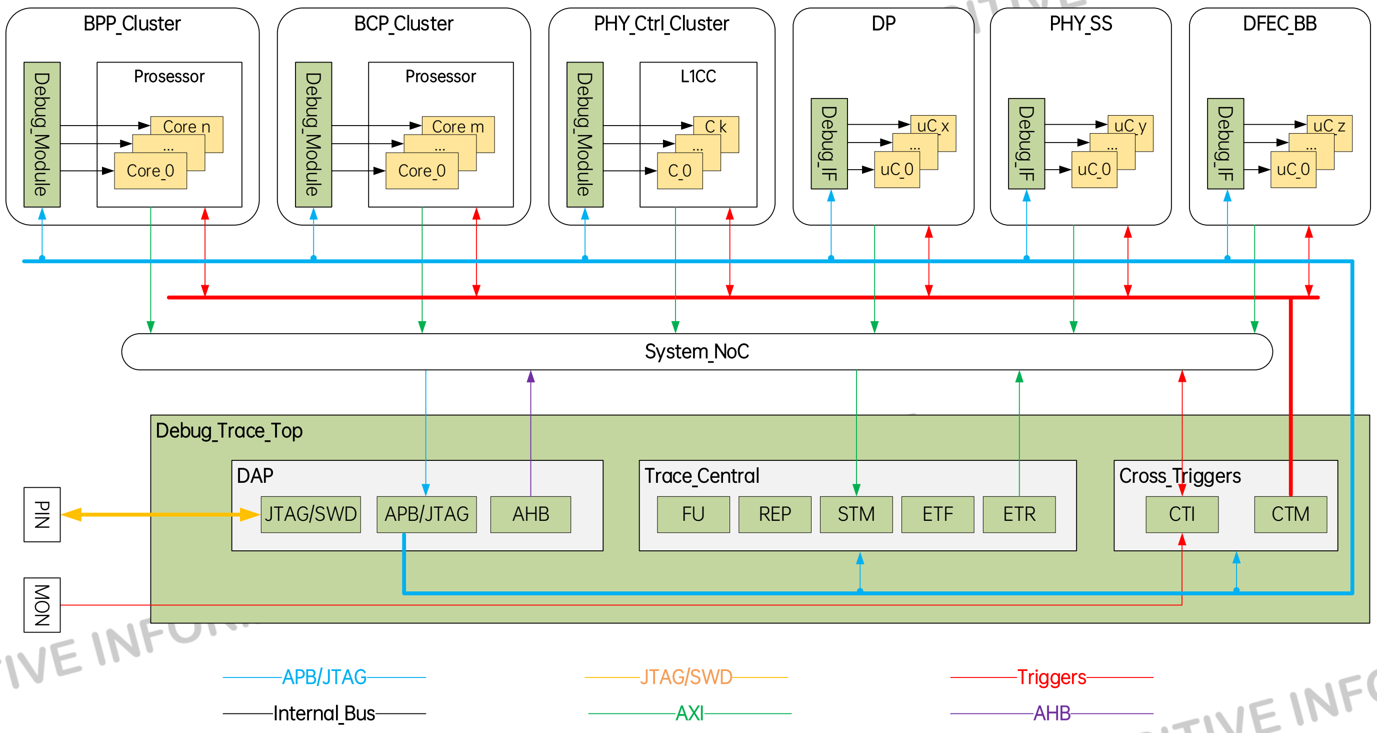

Figure 3-1 Debug HW Architecture

3.2 Debug Components

The primary debug components are ARM Coresight SoC 600 based, and they are described in the following section.

Besides the Coresight SoC 600 components, some system modules are also the parts of debug infrastructure.

- System NoC: events observation and triggers.
- Monitor: event collection and selection.
- Debug control units: debug access control and debug related control and status registers.

3.2.1 Debug Access Port DAP

With the Coresight SoC 600, DAP can be constructed from the generated and available Coresight components. Those access port generation based on the processors and cores need. It's the designers receptibility to make the right implementation. The general Dap architecture is showing in the figure 4 below.

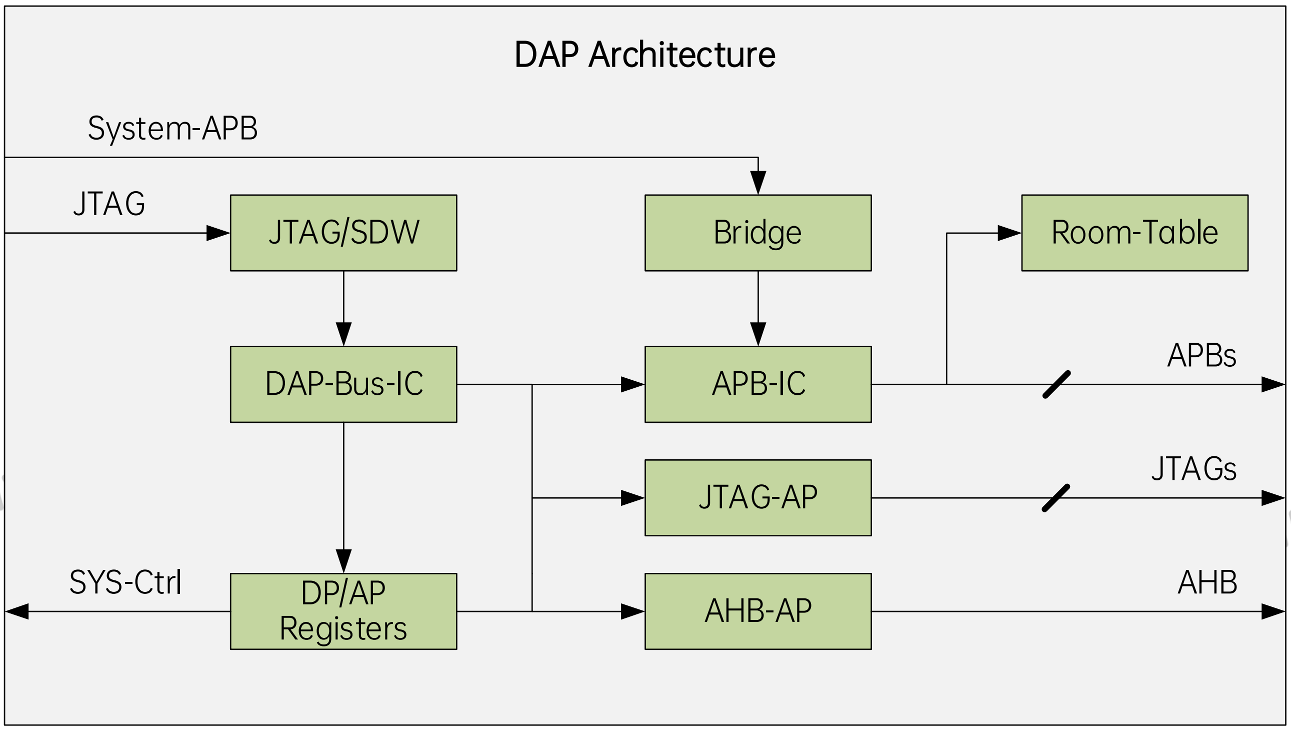

Figure 3-2 DAP Architecture Overview

Special care needs to be taken into consideration regard to the system control interaction with DAP, such system wakeup and debug power control.

3.2.2 Multicore Debug Break Trigger

The Cross Triggers for multi-core debug is built from Coresight components CTI and CTM. The triggers in and out are in pairs. The IF quantity of CTI and CTM is configurable which depends on the total number of triggers to be connected.

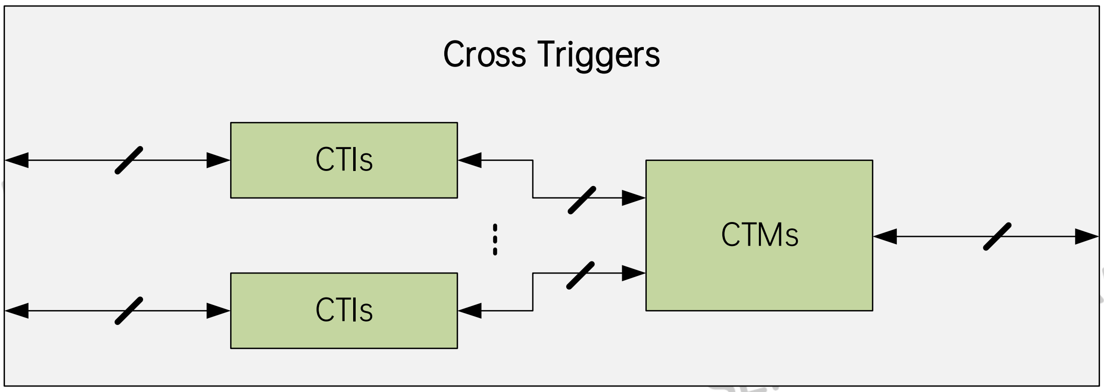

Figure 3-3 Cross Trigger Implementation

A selected trigger in event will be used to break a selected core into debug mode. Trigger in events are from various functional module or signals. In general, they are listed as below.

- System alarm event
- System error events
- Fetal error report
- Execution abort
- System crash interrupt
- Core break request
- External debug request
- SW debug request
- Others

3.3 Debug Address Memory Maps

Each core and Coresight components with APB IF will be assigned with a fixed global address range for external debugger or internal hosted debug access. The ROM table also need a dedicated address which usually start from the debug base address.

All components or submodules in a subsystem can shared one address chunk. APB address decoder can be introduced to decode each access.

Detailed debug address map, please refer to the Debug and Trace HD, or the global address mapping data sheet.

3.4 Debug Rom Table

The ROM table provided by the DAP is to record the debug components connected and its locations in the debug system. Besides the ROM table in the DAP, it is applicable to have local ROM table in a subsystem. Detailed ROM table implementation please refer the CoreSight Architecture Specification for more information.

3.5 Clock Power and Reset

Power domain allocation general guideline are:

- The central debug system is suggested to be in an always on or dedicated switch off domain. It should be independent from the any modules.

- The debug modules in a subsystem should be independent to the subsystem in which it is located.

There are several clock domains in the debug system:

- External debug clock domain: JTAG TCK.
- DAP clock: APB bus clock, JTAG-AP TCK (divided from DAP clock). All these clocks in the SoC should be synchronous.
- System bus clock: APB access from system, and AHB access to the system.

To prevent the debug subsystem or module to be reset carelessly or unexpected, only below reset are valid to a debug system or module:

- System power on reset.
- Debug reset: it's universal to all debug components and asserted by debugger or debug mode control registers.
- Debug subsystem or module power switch reset id applicable.

The debug clock: DAP_CLK, and debug reset DBG_RSTn should be the signal source shared by all of debug modules. The clock and reset distribution are illustrated in figure 6 below.

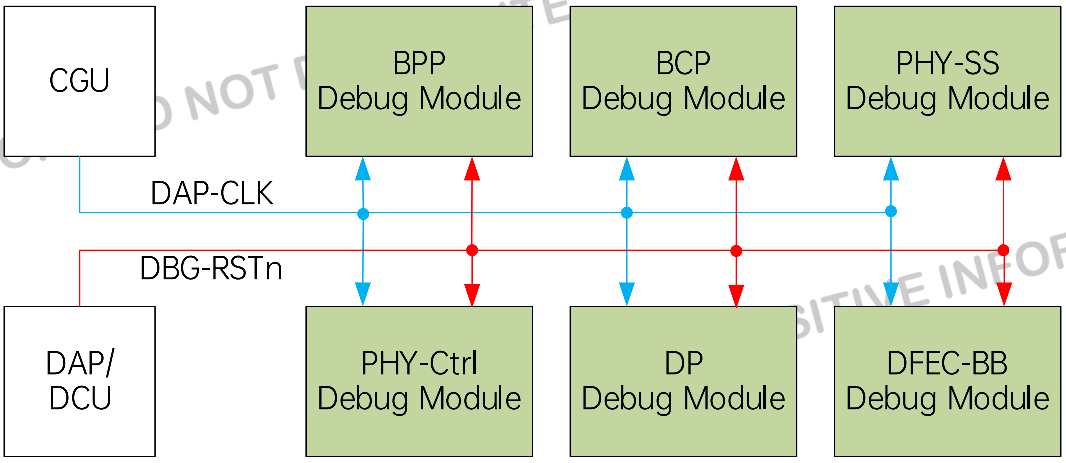

Figure 3-4 Debug Clock and Reset Distributions

3.6 Debug Access Control

To facilitate the debugger connection and access, those requirements listed below are mandatory in the condition of system low power mode, such as deep sleep mode.

- The debugger connection should be kept.
- Each Core power state status can be read from debug status registers.
- Debugger can wake up a core via DAP cSYSPWRUP_REQ.

In addition, these features listed below are very helpful in debug control:

- Debugger can keep a core power on via cDBGPWRUP_REQ. SW power off request will be override.
- Debug request to a core can be disabled/enabled by debug registers control bit.
- Debug request can be issues from debug registers control bit.
- Debug reset to a core can be disabled/enabled by debug registers control bit.

3.7 Security Debug

The debug access is sensitive to protected cores, memory or register contents. Therefore, the security feature implementation of debug should be taken from both debug access control, and debug area protections aspects. Please take below figure 7 as an example.

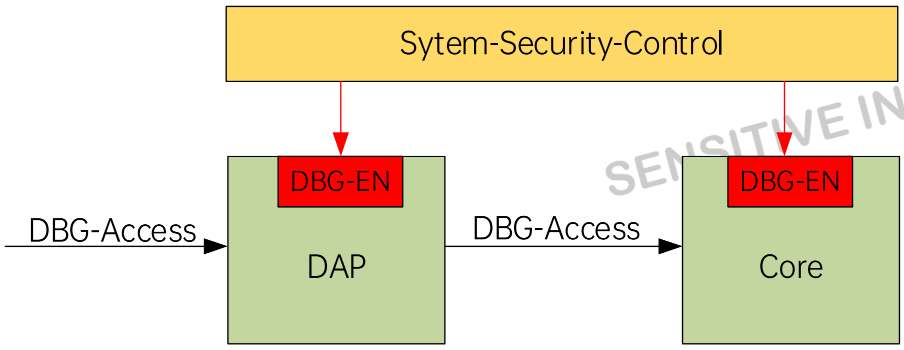

Figure 3-5 Debug Access Security Control

Debug access control or area protection can take achieved by mean of:

- Fuse protection
- Fused register bits control
- Debug mode register enable/disable
- Security policy control

The detailed debug security control please refer to System Security HD and Debug and Trace HD for more information.

4 Trace HW Solution and Functionalities

The overall trace HW solution based on the Coresight SoC600 implementation. Coresight components will be used as the trace infrastructure to meet below trace functionalities and requirements:

- Trace message packet into STP2.0 formation
- Timestamp insertion based on request
- Trace data buffering
- Trace data directed to different sink
- HW events trace

The SW trace requirements are use case specified, but the HW data trace path is shared with all trace sources. SW will take care of trace source data manage before feed into HW trace components. Details SW solution please refer to the SW related chapter.

HW will make the maximum capabilities to improve the trace bandwidth. For those very high payload trace, optimization between SW and HW should be evaluated. The right trace check point and precise capture of interested trace data is the practical way to improve HW bandwidth limitation.

4.1 Trace Architecture Overview

The proposed trace HW architecture is showing below as figure 8. Distributed architecture is applied for increase the flexibility and reduce the wiring and timing issues. A dedicated trace module will be integrated into those primary subsystems. The debug and trace central module will collect trace from different subsystem, and direct to selected sinks.

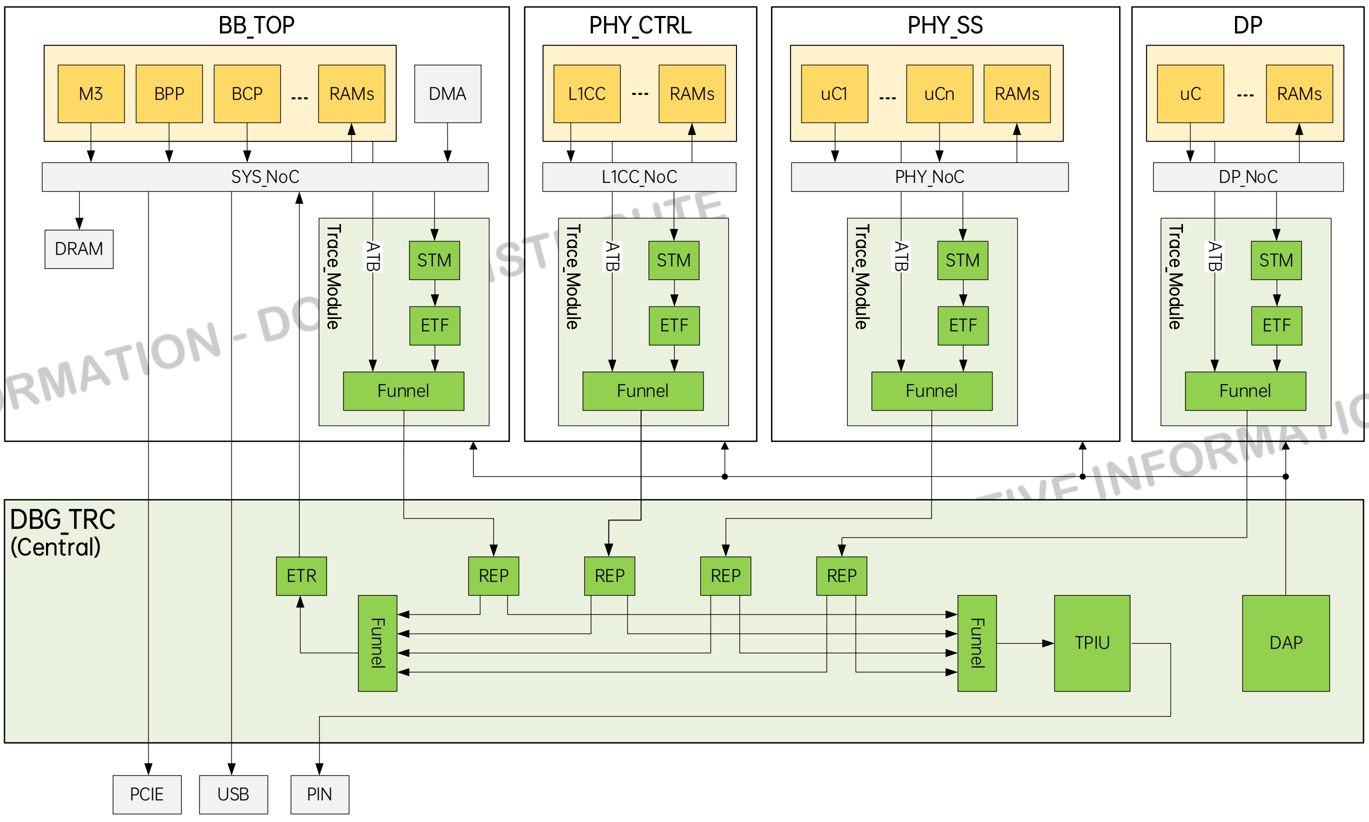

Figure 4-1 Trace HW Architecture

4.2 Trace Functionalities

The trace functionalities covered by the HW solution are listed as below:

- Concurrent trace capability from different subsystem
- Timestamp insertion on request
- Trace message formation with STM
- Trace message buffering with ETF
- Message filter in The Replicator
- Trace can be enabled/disabled in the funnel
- Trace priority configurable in the Funnel
- Two trace data output paths:
  - Off chip directly from TPIU
  - Store in DRAM from ETR, and off chip via PCIE or USB
- USB can empty trace data from ETR directly to off load DDR traffic
- Trace security control

4.3 Trace Components

To build the trace HW, the mandatory Coresight components are simply described in the following section. Please refer to the ARM specifications for detailed information.

4.3.1 System Trace Module STM

STM is the key component to format trace message into desired formation. The essential information included are:

- Master ID
- Message type
- Channel information
- Package synchronization
- Timestamp

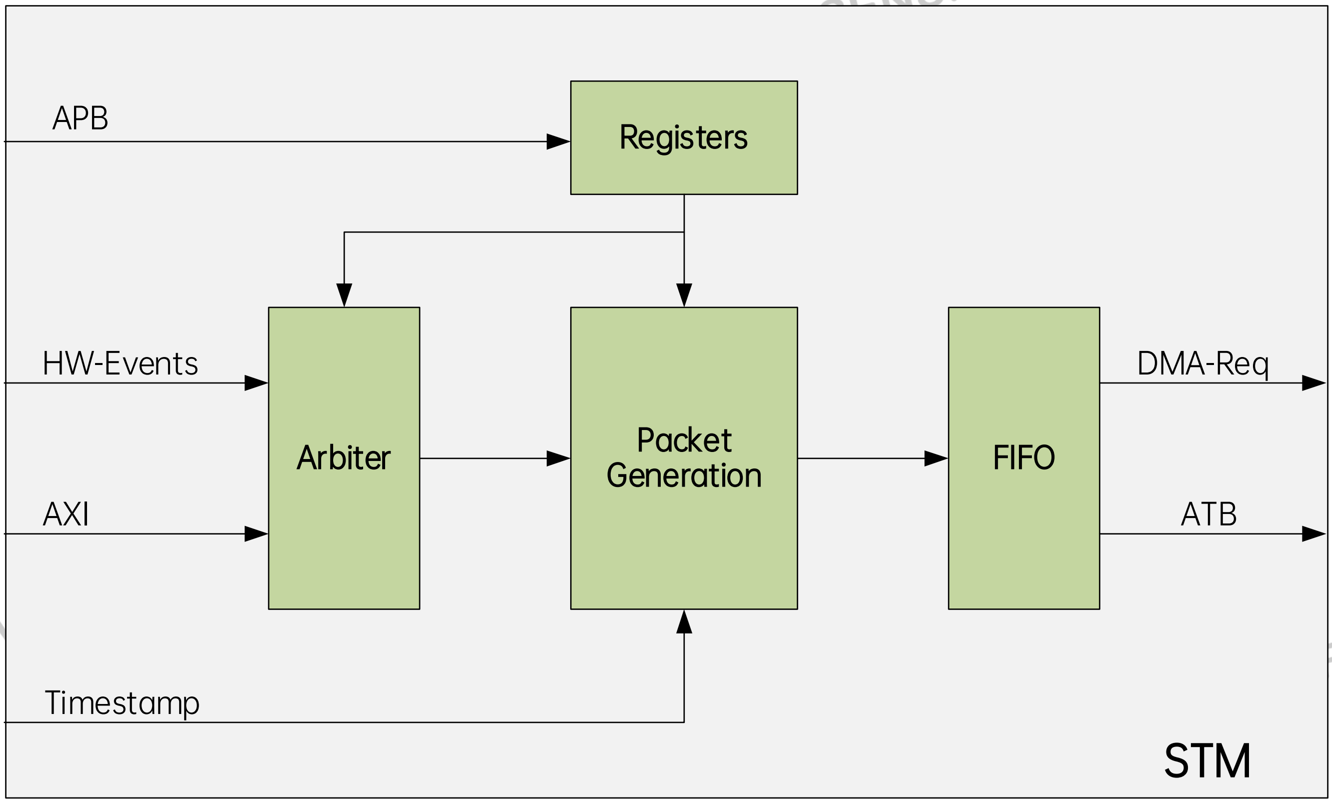

Figure 4-2 STM Architecture

The HW event IF is configurable and suggested only existing in the baseband top STM. Its input will be from the monitoring subsystem. Others subsystem don't support HW event trace.

DMA request IF can be used to move on chip memory content to the STM AXI input. It's compatible with ARM DMA330. A general DMA can also achieve the function if the DMA request IF is not used.

4.3.2 Embedded Trace FIFO

The ETF can be inserted anyway for the ATB bus to buffer the tracing data or relax the timing issues. It can be configured into FIFO mode or ring buffer mode. The RAM size is also configurable which will be define in the Debug and Trace HD. Figure 10 is a simple ETF diagram illustration.

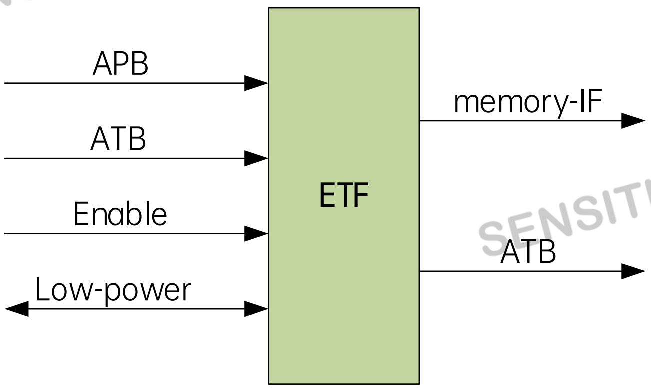

Figure 4-3 Embedded Trace FIFO

4.3.3 Embedded Trace Router

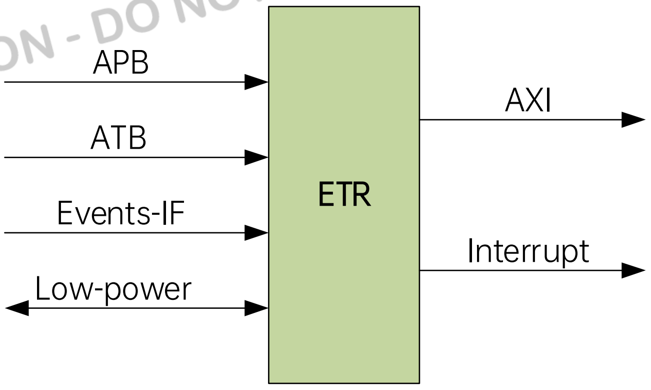

Figure 4-4 Embedded Trace Router

The ETR enables trace to be routed over an AXI interface to the system memory or to any other AXI slave. Only one ETR will be instantiated and shared by all trace sources. An ETR diagram is showing in figure 11 above.

To empty the trace data via USB from ETR directly is a nice to have feature to be evaluated. It will mitigate NoC and DRAM traffics greatly.

4.3.4 Trace Funnel

Trace funnel is the element to select one ATB bus out from several input. The number of input ATB slave IF is configurable from 2 to 8. Please refer to the drawing below.

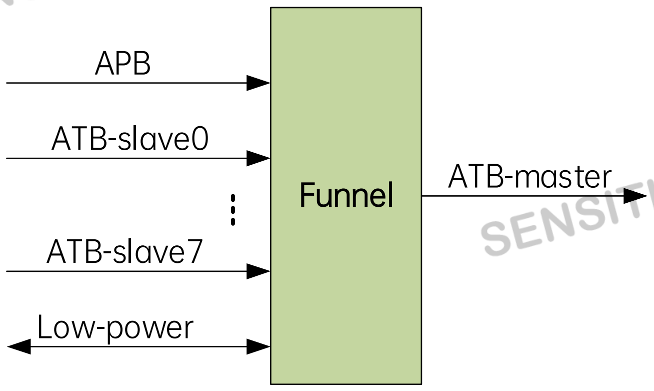

Figure 4-5 Funnel Architecture

The Funnel has programmable mode, or none programable mode. The programable mode should be used due to below desirable features. Referred from Arm® CoreSight™ System-on-Chip SoC-600 Technical Reference Manual.

- Independent enable control for each slave port.
- Independent priority setting for each slave port, so that higher priority ports are serviced ahead of lower priority ports.
- Programmable hold time to reduce input switching that is based on trace ID value.
- Registers to allow integration testing of the trace network.

4.3.5 Trace Replicator

A replicator is used to split an input ATB to two output that can go to different trace sinks. It supports programmable mode and none programmable mode. The programable mode should be used due to below desirable features. Referred from Arm® CoreSight™ System-on-Chip SoC-600 Technical Reference Manual.

- Filtering of trace IDs to allow some IDs to go to master port 0 and some to master port 1.
- Registers to allow integration testing of the trace network.

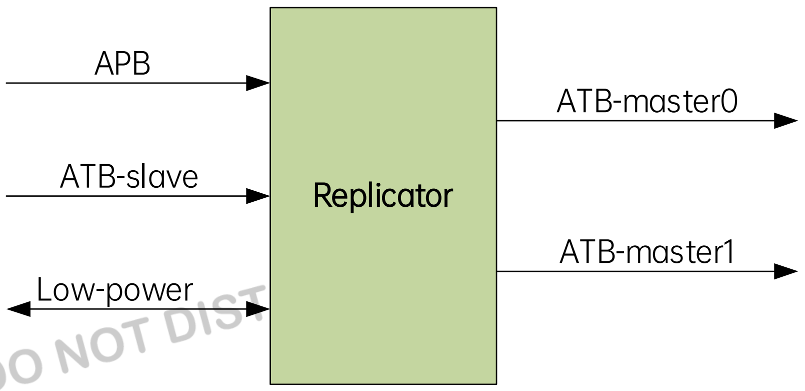

Figure 4-6 Replicator Architecture

4.3.6 Trace Port IF Unit TPIU

TPIU is the bridge between an ATB slave IF and the GPIO pins. The key features are listed as following:

- Trace capture and halt based on a trigger event.
- Insertion of source ID information into the trace stream so that trace data can be decoded.
- Outputs the trace data over trace port pins.
- Outputs patterns over the trace port so that a debugger can tune its capture logic, maximizing the speed at which trace can be captured.

Please refer the Arm® CoreSight™ System-on-Chip SoC-600 Technical Reference Manual for detailed information. On important implementation for maximum the pin capture speed is to make the clock edge at the middle of data to be captured.

4.4 Trace Data Path

One of the flexibilities provided by the HW is the trace path. Some examples are illustrated in below drawing as figure 14.

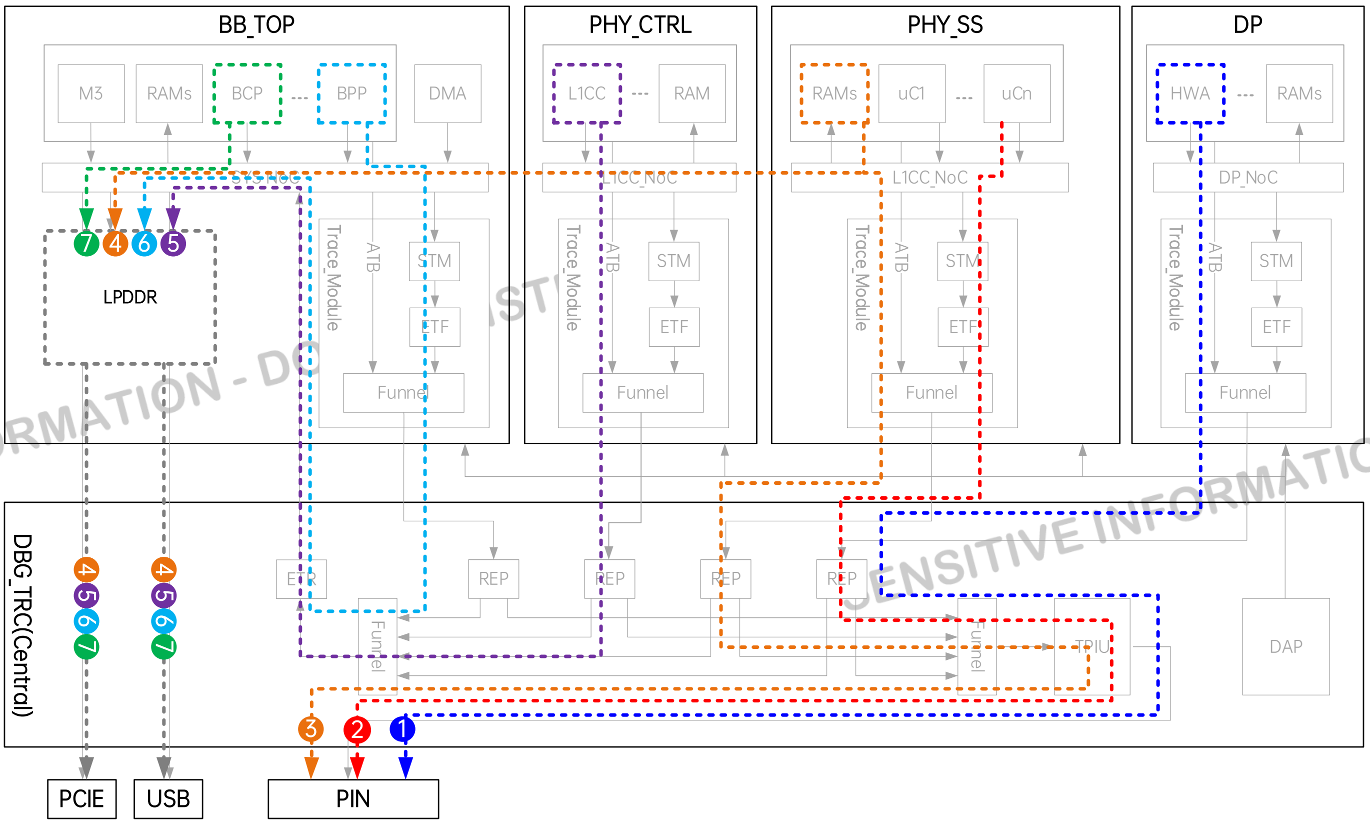

Figure 4-7 Trace Path Example

Example 1:
HW Accelerator data directly output to pin via TPIU without STP formation.

Example 2:
uC SW data pocketed into STP formation and output from pin via TPIU directly.

Example 3:
Data from RAM directly output to pin via TPIU without STP formation.

Example 4:
Data from local RAM moved to LPDDR then output via PCIE or USB.

Example 5:

ETM data moved to LPDDR then output via PCIE or USB.

Example 6:

BPP SW data pocketed into STP formation and move to LPDDR, then output via PCIE or USB.

Example 7:

BCP write data to LPDDR directly, then output via PCIE or USB.

4.5 Trace Source and ID Allocation (TBD)

Each trace source will be assigned with a unique global 8 bits master ID (MID) that will be used for debug tool to decode where the data is originated from. The MID information inserted in the STP protocol while the data formatted in STM.

Table 4-1 Trace Source MID Allocation

Trace Source	Master ID	Comments
BPP SW Trace	0x01	
BPP Profiling Trace	0x02	
BCP SW Trace	0x03	
BCP Profiling Trace	0x04	
L1CC SW Trace	0x05	
L1CC Profiling Trace	0x06	
ETM xx	0x10	
uC xx SW Trace	0x20	
BPP NoC Trace	0x60	
BCP NoC Trace	0x61	
PHY Data NoC Trace	0x62	
DP Data NoC Trace	0x63	
DP IP Header Trace	0x64	
2G Trace	0x65	
3G Trace	0x66	
4G Trace	0x67	
Audio Trace	0x68	

4.6 Trace Address Memory Maps (TBD)

Those Trace moved via NoC to STM or an ATB slave IF need assigned memory mapped address to decide which sink to go. Also, the MID information of SW traces is decoded from the AXI address bit. Please refer to STM specification for more information.

Table 4-2 Trace Address Mapping

Trace Source	Address Range	Comments
BPP SW Trace	0x0100_0000 - 0x0100_1000	
BPP Profiling Trace	0x0100_0000 - 0x0100_1001	
BCP SW Trace	0x0100_0000 - 0x0100_1002	
BCP Profiling Trace	0x0100_0000 - 0x0100_1003	
L1CC SW Trace	0x0100_0000 - 0x0100_1004	
L1CC Profiling Trace	0x0100_0000 - 0x0100_1005	
uC xx SW Trace	0x0100_0000 - 0x0100_1006	
BPP NoC Trace	0x0100_0000 - 0x0100_1007	
BCP NoC Trace	0x0100_0000 - 0x0100_1008	
PHY Data NoC Trace	0x0100_0000 - 0x0100_1009	
DP Data NoC Trace	0x0100_0000 - 0x0100_1010	
DP IP Header Trace	0x0100_0000 - 0x0100_1011	
2G Trace	0x0100_0000 - 0x0100_1012	
3G Trace	0x0100_0000 - 0x0100_1013	
4G Trace	0x0100_0000 - 0x0100_1014	
Audio Trace	0x0100_0000 - 0x0100_1015	

4.7 Trace Time Stamp

Timestamp used for trace message is generated from ARM timestamp generator. Please refer to the Arm® CoreSight™ System-on-Chip SoC-600 Technical Reference Manual for more details.

The considerations for timestamp implementation are listed as below:

- 64bits timestamp preferred.

- Timestamp counter value can be read by SW.
- Timestamp counter should continue count during system sleep mode.
- The timestamp counter can be synchronized with system time counters.

4.8 Clock Power and Reset

Power domain allocation general guideline are:

- The central trace system is suggested to be in an always on or dedicated switch off domain. It should be independent from the any other modules.

- The trace modules instantiated in a subsystem should be independent to the subsystem in which it is located.

There are several three clock domains for the trace system:

- Timestamp clock: suggesting 38.4MHz reference clock, and 32KHz in sleep mode.
- ATB bus clock: STM, Funnel, Replicator, ETF, ETR and TPIU.
- APB bus clock: for all Coresight components configurations.
- Trace clock: for TPIU.

Reset sources for trace module can be as following:

- System power on reset.
- Each trace module power switch reset if applicable.
- Each trace module SW reset
- Debug reset
- Others system warm reset should not reset any trace modules.

4.9 Low Power Trace Features

Low power IF should be generated for all trace modules. It is a safe way to guarantee no data lost on the request of system power, clock switch off, or reset active. The low power IF feature please refer to ARM Low Power Interface Specifications.

Besides the low power IF features, the power switch sequence should follow the right orders. General principals are:

- Power off: switch off trace source side first, then trace sink side. Take the Figure 15 below for instance:
  EMT → PHY_NoC →Trace_PHY → PHY_SS → Trace_Central → LPDDR → BPP_NoC → BPP_Top
- Power on: switch on sink side first, then the trace source side. Follow the reverse order as above example.

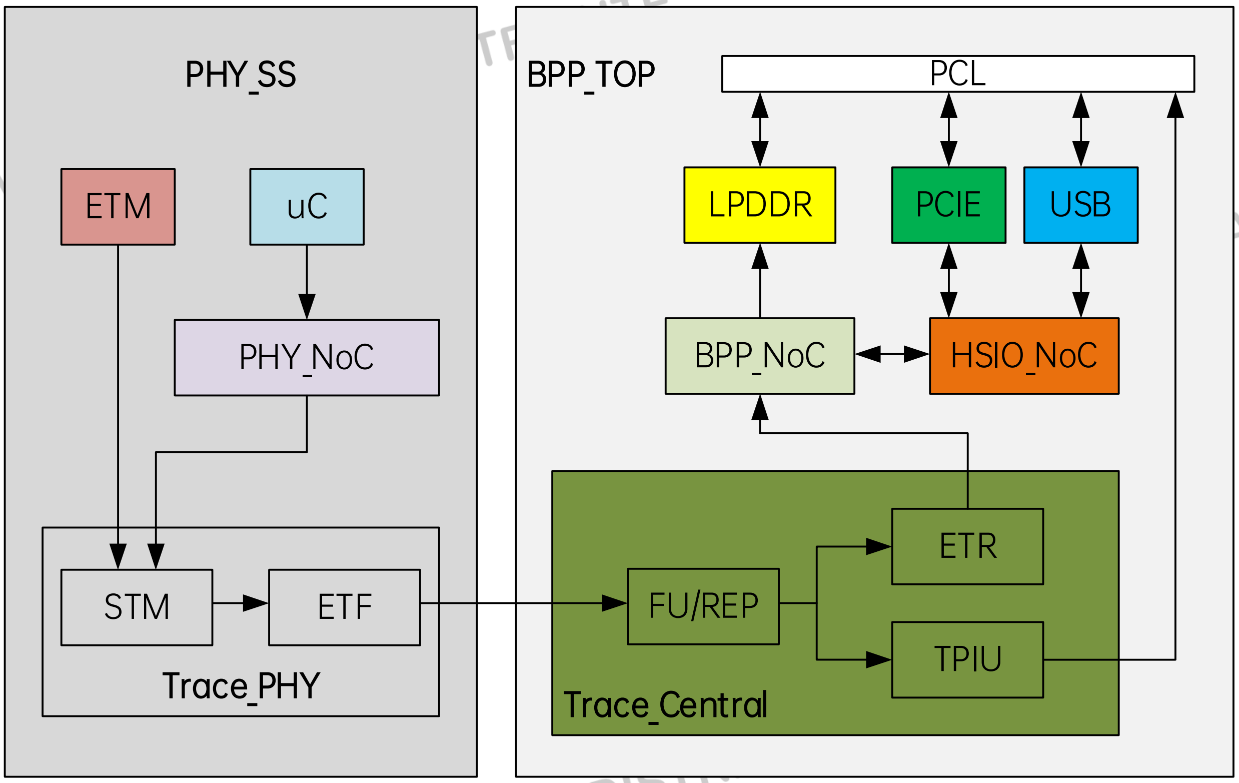

Figure 4-8 Trace Power Switch Example

4.10 PHY Subsystem Trace

PHY subsystem trace and debug is the most demanding customer due to its complexity and leading functionality in 5G modem. Beside big trace data requirements, there are many micro-controllers, vector DSPs and task sequencers that need high performance trace infrastructure to meet debug purpose. The solutions for each debug requirements will be detailed in following chapters.

4.10.1 PHY Subsystem Debug Trace Architecture Overview

PHY HWA generated data, including IQ and LLR, can be grouped based on the function and use cases. Each group will share one Funnel to wire the trace data to the trace sink. All register value and task sequencer execution trace can share one Funnel. Micro-controllers and vector DSPs trace data will be direct to the STM.

An APB decoder will be placed in the PHY debug and trace module to distribute the debug access to each micro-controllers and vector DSPs.

Monitoring signals collection is not show in the picture below. Designers should take care of monitoring implementation.

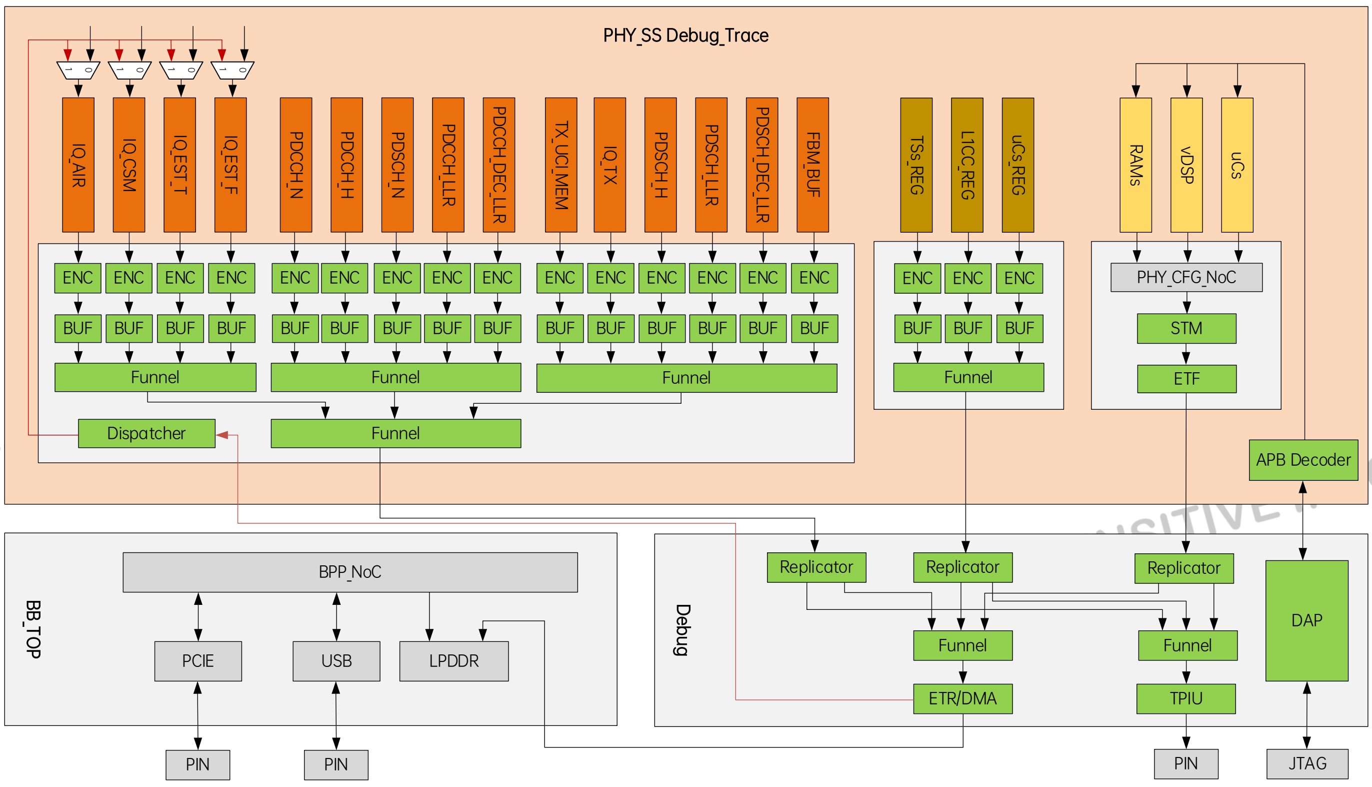

Figure 4-9 PHY Subsystem Trace and Debug Architecture Overview

4.10.2 HW Trace Source and BW Requirements

There are 15 HW check points identified from which the interested date needs to be output by trace infrastructure for debug purpose. Each check point and their BW requirement are listed as below.

Table 4-3 Check Points List for Sub 6G

Item	Description	Source Name	Property	Data rate(Mbps)
Check point 1	IQ data from DigRF	IQ_AIR	HW	11,796.48
Check point 2	IQ data & AGC info to Symbol Buffer of CSM	IQ_CSM	HW	2,834.88
Check point 3	Time domain IQ data to Symbol Buffer of EST	IQ_EST_T	HW	11,796.48
Check point 4	Frequency domain IQ data for EST	IQ_EST_F	MEM	8,306
Check point 5	Pdcch NoiseCov result	PDCCH_N	HW	86.5
Check point 6	Pdcch H info	PDCCH_H	HW	238
Check point 7	Pdsch NoiseCov result	PDSCH_N	HW	86.5
Check point 8	Pdsch H info	PDSCH_H	HW	16,796
Check point 9	Pdcch LLR data	PDCCH_LLR	MEM	158
Check point 10	Pdsch LLR data	PDSCH_LLR	MEM	21,500
Check point 11	LLR data for PDCCH decoder	PDCCH_DEC_LLR	HW	481
Check point 12	CBs LLR data for PDSCH decoder	PDSCH_DEC_LLR	HW	22,419
Check point 13	FBM buffer	FBM_BUF	MEM	5,242
Check point 14	TX UCI memory	TX_UCI_MEM	MEM	3.07
Check point 15	TX IQ data to DigRF	IQ_TX	HW	23,592
Total rate				113,539.43

Table 4-4 Check Points List for MMW

Item	Description	Source Name	Property	Data rate(Mbps)
Check point 1	IQ data from DigRF	IQ_AIR	HW	23,594
Check point 2	IQ data & AGC info to Symbol Buffer of CSM	IQ_CSM	HW	17,504
Check point 3	Time domain IQ data to Symbol Buffer of EST	IQ_EST_T	HW	23,594
Check point 4	Frequency domain IQ data for EST	IQ_EST_F	MEM	16,242
Check point 5	Pdcch NoiseCov result	PDCCH_N	HW	86.5
Check point 6	Pdcch H info	PDCCH_H	HW	238
Check point 7	Pdsch NoiseCov result	PDSCH_N	HW	86.5
Check point 8	Pdsch H info	PDSCH_H	HW	16,796
Check point 9	Pdcch LLR data	PDCCH_LLR	MEM	211
Check point 10	Pdsch LLR data	PDSCH_LLR	MEM	31,534
Check point 11	LLR data for PDCCH decoder	PDCCH_DEC_LLR	HW	712
Check point 12	CBs LLR data for PDSCH decoder	PDSCH_DEC_LLR	HW	34,242
Check point 13	FBM buffer	FBM_BUF	MEM	5,242
Check point 14	TX UCI memory	TX_UCI_MEM	MEM	81
Check point 15	TX IQ data to DigRF	IQ_TX	HW	23,592
Total rate				170,161

4.10.3 Big Data Trace

The IQ and LLR trace data rate are very high. Reasonable BW for 1CC in the use case of sub-6G and MMW is about 24Gbps and 35Gbps respectively. MMW trace will not be considered in the current project plan.

Due to the off chip IF BW limitation, the big payload trace data must be buffered in DRAM temporarily, and then drained by the high-speed IF, USB or PCIE. Please refer to figure below.

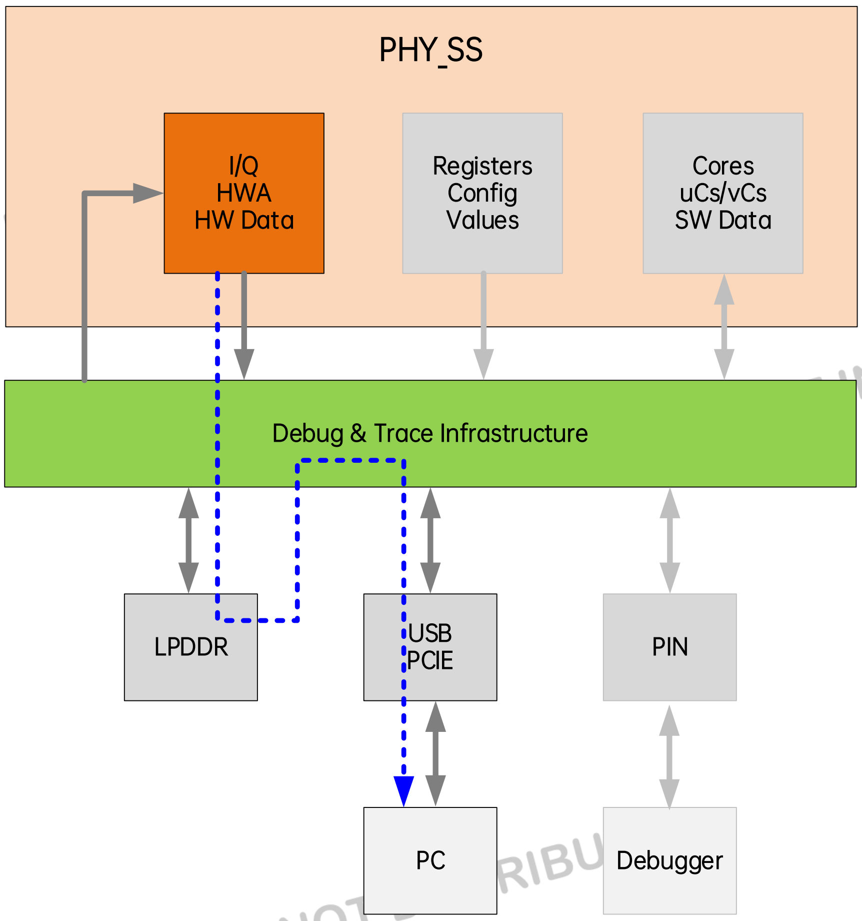

Figure 4-10 Big Data Trace Flow

Key points:

- several IQ or check points can be selected to trace in parallel, however, the total BW must be within the LPDDR allocated BW limitation.

4.10.4 HW Diagnosis Test

Due to the complex of PHY algorithm and design, self-diagnosis for HW and algorithm in silicon debug is preferred. Self-test data or pattern to be imported by USB or PCIE into the LPDDR, then moved out by on chip debug DMA and applied to the interested check points is proposed. Please refer to the picture below.

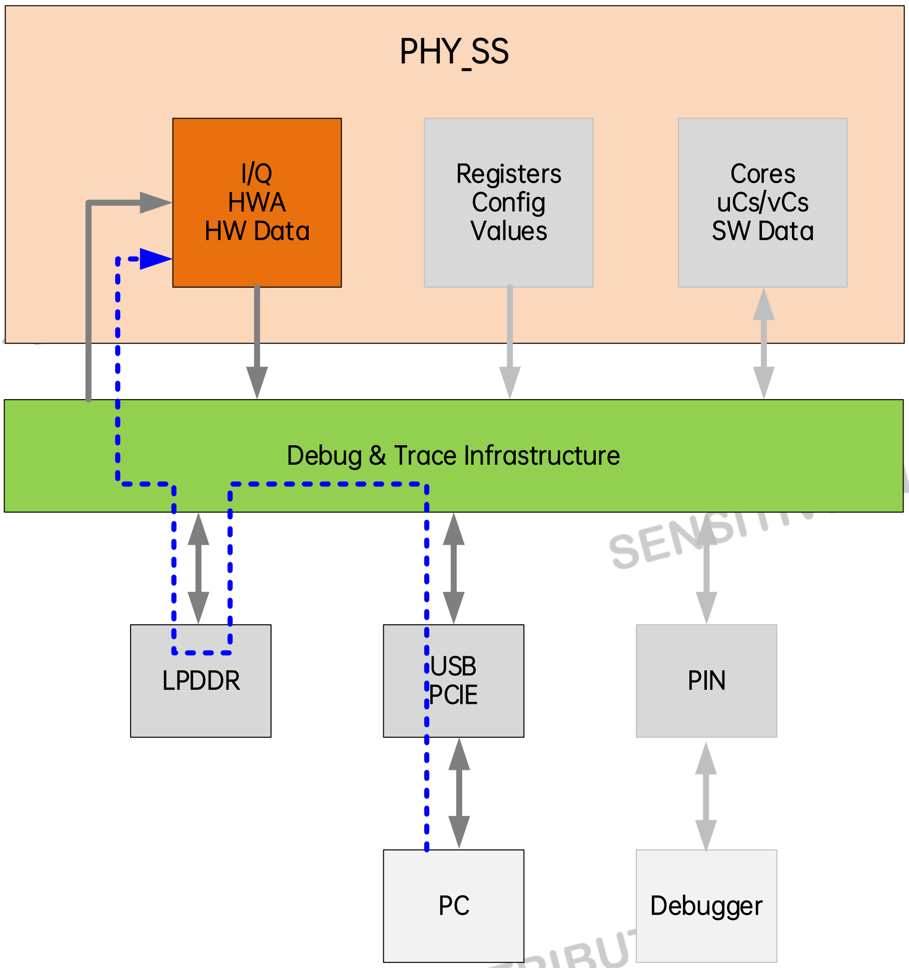

Figure 4-11 Self-Diagnosis Test Flow

4.10.5 Configuration Observation Trace

To facilitate the debug experience, all configuration information is quite interested during debug session. By doing so, user will clear know what is going on of SW or on chip cores or micro-controller, in real time. Those configuration information are listed as below.

- Control registers settings.
- Configuration or dispatcher executing by the task sequencer.
- Configuration or dispatcher executing by the L1 Control.

Configuration information can either be traced out from GPIO directly, or stored in LPDDR and drained by USB or PCIE. Please refer to the picture below.

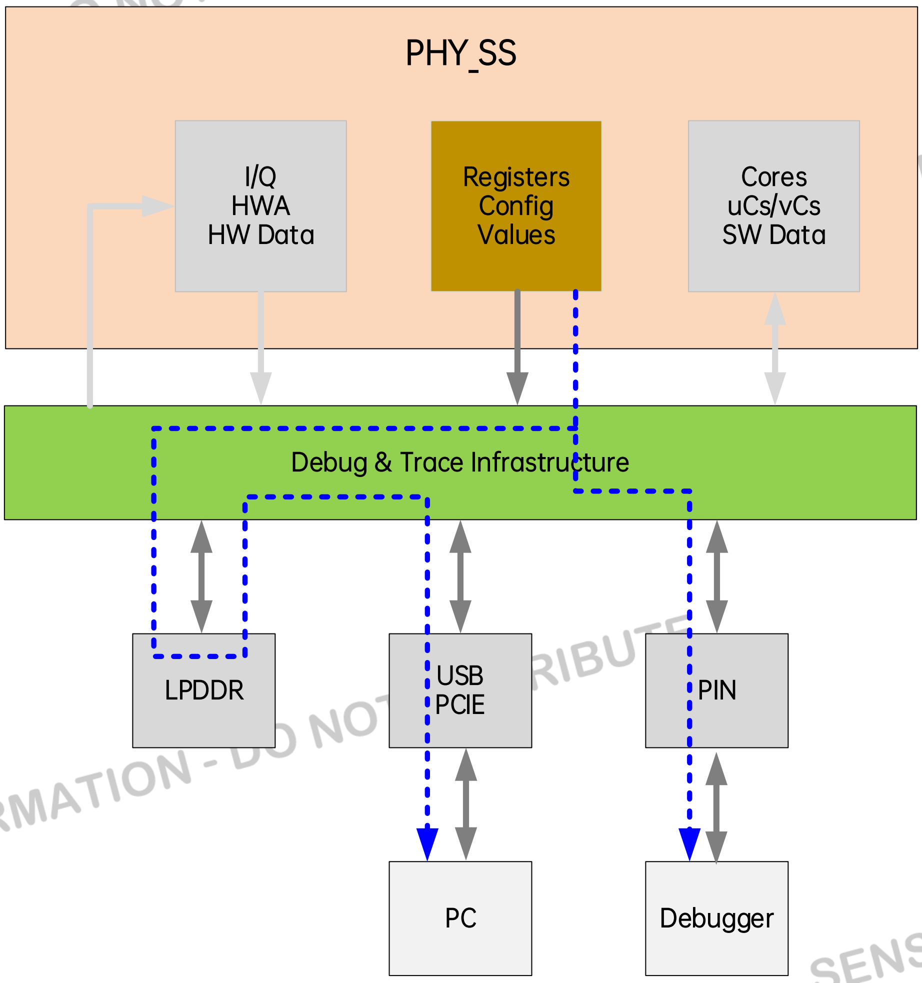

Figure 4-12 Configuration Trace Flow

4.10.6 SW Data Trace

Micro-controller and vDSP generated SW data trace flow is showing in figure 4-12. Detailed SW trace information please refer to SW design specifications.

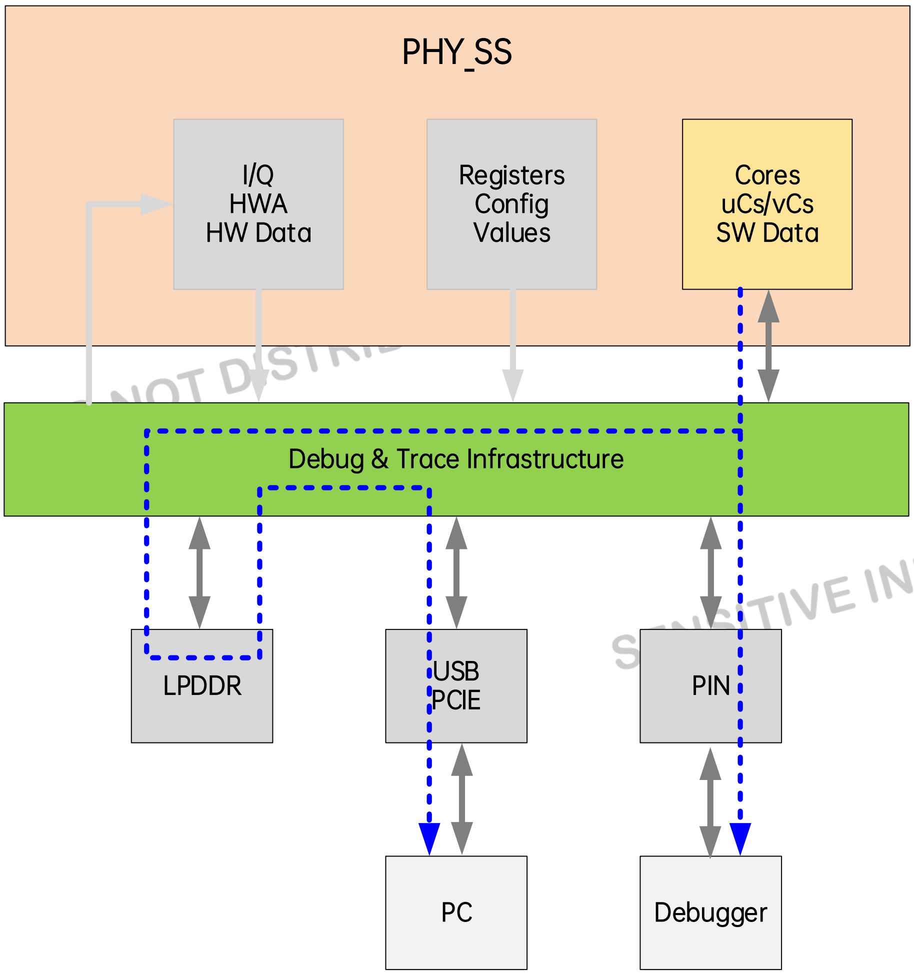

Figure 4-13 SW Data Trace Flow

4.11 PHY Control Trace (TBD)

4.12 DP Subsystem Trace (TBD)

4.13 DFEC_BB Trace (TBD)

5 HW Monitoring

A simple and desired way of silicon debug is to observe interest signals from an oscilloscope. This can be achieved by the signals monitoring mechanism. Detailed concept of monitoring is explained in the following sections.

5.1 Monitoring Architecture Overview

The monitoring system can be built from multi-multiplexer that distributed in different subsystem in the SoC. A three level of monitoring system is illustrated in figure 16 below. The first and second level on the left are the multiplexers distributed in different subsystem based on the number of signals to be monitored. The final level multiplexer on the right will select 32 bits of signals out from different fist level multiplexer.

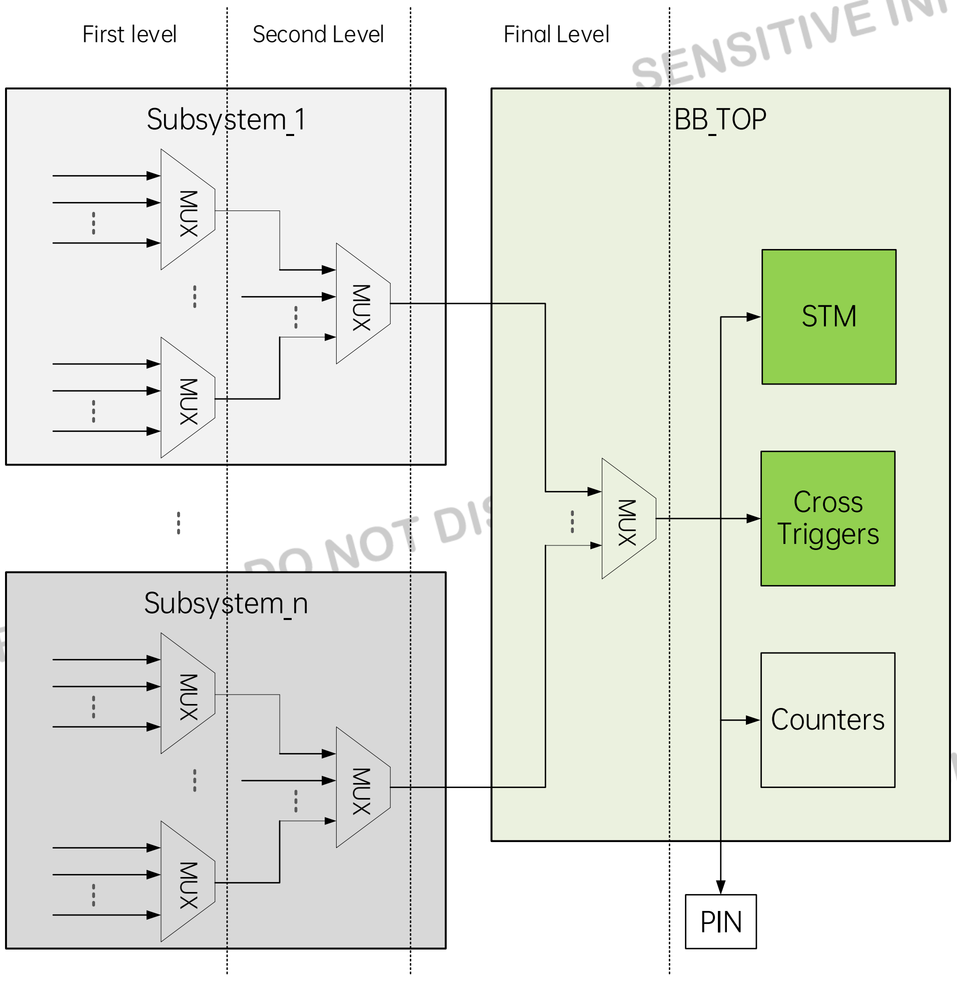

Figure 5-1 Monitoring Architecture Overview

5.2 Monitoring Functionalities

The selected signals can be used as various means of debug purpose:

- Trace out from STM for analysis
- As the multi-core debug break triggers
- As the event counter input to enable/disable/reload a counter
- Output from pin to oscilloscope for observatoin

5.3 Monitoring Components

Multiplexers are the key components for monitoring system and their specifications are described in following sections. And their implementation details please refer to the Debug and Trace HD.

5.3.1 Block Level Monitor Multiplexer BLMM

The fist level multiplexer is call BLMM. Its main features are specified as below:

- Each BLMM will select 16 signals out from 256 input signals.
- Totally 128bits control signals.
- Bit [7:0] will select out the sig4mon_o [0], …, and bit [127:120] will select out the sig4mon_o [15] from any 256 input signals.

5.3.2 Subsystem Level Monitor Multiplexer SLMM

The second level of multiplexer is call SLMM. Its main features are specified as below:

- Each SLMM can host max 16 BLMM input, and the number of inputs is generic, for example G.
- Totally G*8bits configuration signals. Bit [7:0] will select out the sig4mon_o [0], …, and bit [127:120] will select out the sig4mon_o [15], from any bit of any BLMM input.
- In each 8bits, bit [7:4] will sect which BLMM input, and bit [3:0] will sect which signals in the selected BLMM input.

5.3.3 Top Level Monitor Multiplexer TLMM

The final level of multiplexer is called TLMM. Its main features are specified as below:

- Only one TLMM in a chip, and it will host all SLMM input from the chip.
- The number of inputs for a CLMM is generic, for example G.
- Totally G*(M+4) bits configuration signals for a TLMM. M will sect which SLMM input, and the 4 bits will select which signals in the selected SLMM input.

5.4 Monitoring Element ID Allocation (TBD)

Each multiplexer will be assigned with a unique global ID number for control decode. The assignments please refer to the table 4 below.

Multiplexer	ID	Comments
PHY_SS_BLMM1	0x01	
PHY_SS_BLMM2	0x02	
PHY_SS_BLMM3	0x03	
PHY_SS_BLMM4	0x04	
PHY_SS_BLMM5	0x05	
PHY_SS_BLMM6	0x06	
PHY_SS_SLMM	0x10	

Table 5-1 Multiplexer ID Allocation

5.5 Monitoring Control

All multiplexers are central controlled from TLMM central control unit. The control signals are propagated from TLMM to each BLMM and SLMM multiplexers. Detailed control protocol please refer to the Debug and Trace HD.

- All BLMM, PLMM, CLMM configuration registers are in the TLMM with AHB access.
- A single bit line will be used to convey the configuration of each BLMM and SLMM to their respective internal registers.
- A 2bits protocol will be defined for the BLMM and SLMM configuration.

- All BLMM and SLMM will be assigned with a unified global ID. Please refer to the table 4 above.
- A FSM need to be implemented in the TLMM, each BLLM and SLMM to control the register write and read access.

5.6 Bus Monitoring (TBD)

5.7 Events Observations and Triggers (TBD)

5.8 Clock Power and Reset

Clock:

- All BLMM, SLMM and TLMM use the same clock, system reference clock, i.e., 38.4MHz.

Reset:

- All BLMM, SLMM and TLMM can share the same reset from TLMM.
- Subsystem reset will be applied if a multiplexer is in subsystem PSD.

Power:

- All BLMM, SLMM and TLMM are suggested in AON domain. They should be independent from each subsystem power domain.

6 Trace SW Solution and Functionalities

System Trace is necessary to do post mortem analysis on issues and improve performance. It provides an ongoing record of hardware events and software events that occurs in the system. In Mariana, Trace data can come from BPP, BCP, DP, PHY, SYS_CU and DFEC-BB.

SW and HW trace stream are routed to Trace HW and combined into one stream. Then the combined stream will be sent to PC or AP.

6.1 Key Features

Trace export interface related features:

- Shall support real time trace exported to AP or PC
- Shall support trace exported to PC over USB
- Shall support trace exported to PC over TPIU
- Shall support formatted string exported to PC over UART
- Shall support trace exported to PC or AP over PCIe
- Shall support trace storage in circular buffer in DDR in case no connection to PC is not available

Trace protocol related features:

- Trace from all source (including HW and SW) must be synchronized with same time source, so debugger knows the true order of events
- Each trace source shall have its master and channel ID
- Trace exported to PC or AP shall be in accordance with MIPI STP
- All Software trace shall be in accordance with MIPI SyS-T protocol
- All Software components in a subsystem shall use the same methodology of message definition
- Different severity levels shall be supported for trace source, like debug, info, warning, error, etc.

Trace functionality related features:

- Shall generate an error trace message in case of trace loss
- Trace shall be available as part of system initialization
- The trace message names and the associated definition shall be managed at a central place for all modem platform versions and software versions.

- The system trace shall be dynamically configurable at run-time
- Trace configuration shall be effective in real time.
- Trace configurations shall be stored in NVM which can survive a reset
- Shall support transferring PHY trace to Trace HW by DMA
- Shall support real time big data trace for PHY, but trace rate cannot exceed trace HW bandwidth limitation
- Shall support temporary storage of high speed PHY trace in DDR, and output as user requests
- Shall support low power trace output to on chip memory or DDR

Trace security related features:

- PS trace functionality shall be gated according to EFUSE
- PHY trace functionality shall be gated according to EFUSE
- DP trace functionality shall be gated according to EFUSE
- SYS_CU trace functionality shall be gated according to EFUSE

6.2 Software and Hardware Boundary Definition

This part describes the functions and features that are to be implemented in SW. Below figure shows an overview of SW and HW boundary definition. HW is the basis of System Trace, and SW is running on it. Trace SW configures and manages HW and provides trace relevant functionalities to upper applications.

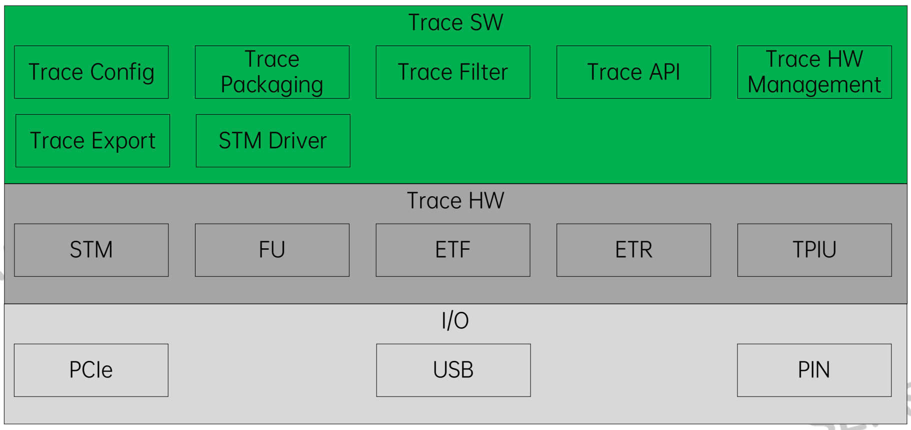

Figure 6-1 SW and HW Boundary Definition Overview

Detailed Trace software scope are as below:

- Trace hardware configuration and management

- System Trace configurations management.
- System trace message packaging
- System Trace API provided to trace sources
- Write Trace to STM
- System Trace filtering
- System Trace output

6.2.1 System Trace Configuration Management

Trace configuration request originally comes from Trace Tool on PC or application on AP. These configurations are set by users according to different test scenario. Trace task decodes these configurations and store them in NVM.

System Trace configurations describe what and where trace is output. Generally, they consist of below:

- Trace Sources
- Trace filter rule
- Trace export interface.

Trace filter rule is optional which is used to further define the trace to output. For example, severity level can be used for filtering. Trace task typically provides a "less than or equal to" comparison on the value of severity level that allows through all Messages from Severity 0 up to a selected level. Setting this filter to the highest numerical level results in all messages being allowed through without any filtering. Setting this filter to the lowest numerical level, results in only Messages with this severity level being allowed through the filter.

As defined in MIPI SyS-T protocol, the Severity levels and their typical uses are:

- MAX – never filtered.
- FATAL – a condition from which the system cannot recover.
- ERROR – an operation failure or action that the system is not designed to handle.
- WARNING – an undesirable, but recoverable, situation that might result in problems if no actions were taken.
- INFO – reporting component-specific information about actions and progress.
- USER1 – similar to INFO but with greater detail.
- USER2 – similar to USER1 but with even greater detail.
- DEBUG – maximum detail of system behavior, e.g., during verification and debug phases.

6.2.2 Trace Hardware Management

Trace is a distributed system whose components are located in different sub-system. In Mariana Trace system, 4 separate STM paths are designed, which serve different sub-system. Trace HW needs to be configured prior to working. The configuration is got from Trace Configuration Management, then Trace HW can be configured accordingly.

Distributed and centralized management are two modes to do this. In order to simplify the management, centralized mode is chosen. A global control task of Trace SW is designed which is running on BCP. Trace HW is initialized in this task. Additionally, it receives trace configurations from Trace Tool, then configure Trace HW correspondingly.

6.2.3 System Trace Message Packaging

SyS-T protocol is defined by MIPI. It stands for System Software Trace, which is a universal data format for transmitting software debug and trace information. SyS-T protocol is not applicable for HW trace. Every HW trace has its specific trace protocol. The latest version of SyS-T is V1.0.

SyS-T message is sent to host using a transport layer. System trace data flow is shown as below.

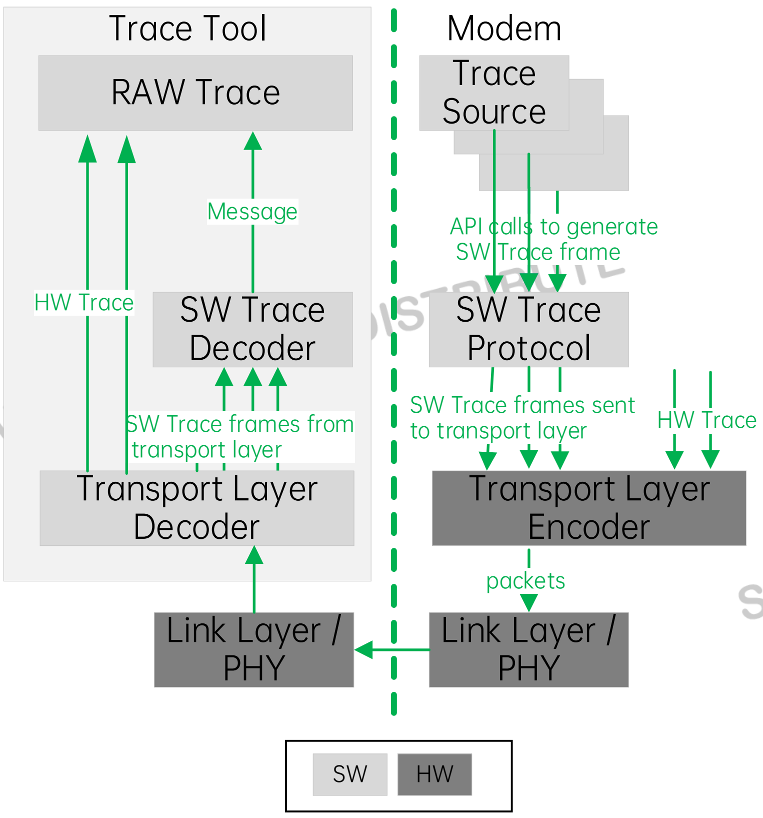

Figure 6-2 System Trace Data Flow

SyS-T normal trace message structure is shown in below figure.

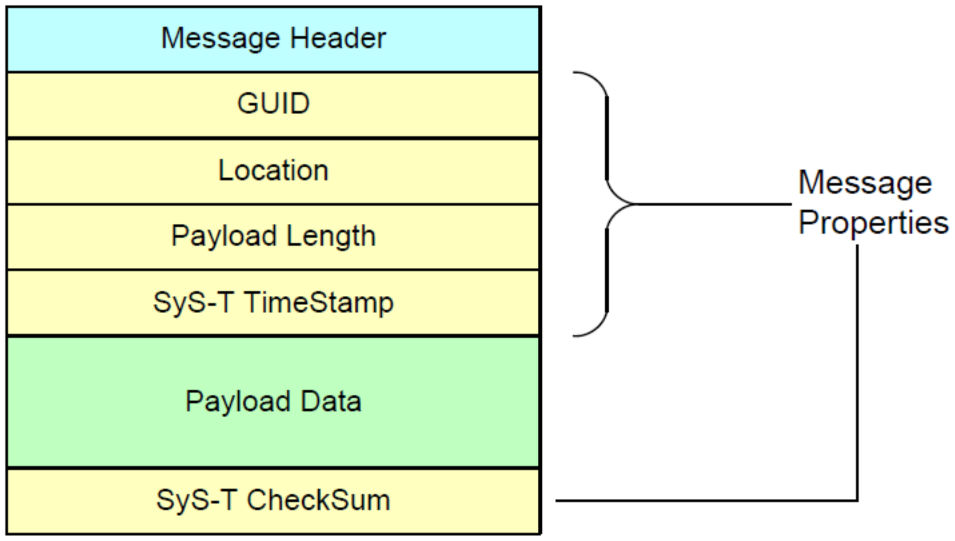

Figure 6-3 SyS-T Normal Message Structure

The fixed length Message Header (colored blue) contains several basic fields that are common to all SyS-T Normal messages. These include the Type field which indicates the meaning of the Payload Data (colored green). Each of optional Message Property fields (colored yellow) is present or absent dependent upon the value of an indicator bit in the Message Header. The set of Property fields that are present in a particular Message is dependent on every trace source.

6.2.4 Write Trace to STM

The STM enables tracing of system activity from various sources:

- SW trace.
- HW trace

The activity observed by the STM is packaged into a trace stream, for output to trace capture device such as USB.

The STM supports the following:

- Multiple software masters writing software trace independently. Each master can use multiple stimulus ports.
- Timestamping of the system activity. The timestamp is a global timestamp which can be shared with other trace sources in the system, to enable correlation of activity from multiple trace sources.
- Interaction with DMA controllers, to manage the flow of data in the system.

- Indicating that specific events have occurred, such as the occurrence of a particular hardware event or a particular piece of software instrumentation. These events are known as triggers and can be indicated in the trace stream, or through signals to other system components.

There are 3 types of writing to STM.

1) Direct write in calling context. Trace is immediately written to STM in the calling task or ISR, however only short trace is suggested in ISR, otherwise there will be a timing issue. This type of transfer is applicable for a metal bare sub-system.
2) Write by DMA. DMA is used to transfer trace to STM, so CPU is not occupied. This type of transfer is applicable for large block of data or CPU timing critical sub-system.
3) Deferred write in a task. Trace source sends message with trace data information to a dedicate task which is with low priority, then this task writes trace to STM. This type of transfer is applicable for a RTOS based sub-system.

Deferred write may cause late timestamp. Timestamp is generated in STM, so the Trace is not with a timestamp until it is sent to STM by the low priority Trace task. Though this could not be an issue, because the order of Trace does not change, but the timestamp delta between them is different from the real value. A gap should be expected.

Deferred write may also cause low utilization of STM. If the Trace task is not scheduled to run, no writing to STM would happen, while there are Trace pending in the task. If the Trace task can be scheduled based on SMP, it is unlikely to have this task starving.

6.2.5 System Trace Filtering

Filtering enables us output the specific trace that we need for post-mortem analysis. If trace filter rule is configured in system trace configuration management, trace needs to be filtered according to the rule before export. Trace filtering can be implemented in HW or SW. if it is HW filtering, it can offload CPU time, but it is too late because STM writing has happened.

Software filtering gives below two great advantages:

- Earliest filtering can be supported. Trace software can provide API for querying current trace filtering settings. If the trace is configured to be filtered, no next step will be taken for tracing.
- Flexible filtering can be supported. Trace software can support flexible filtering rules, and these rules are dynamically configurable.

Filtering is performed for each trace, so the efficiency of software filtering is very critical. Low efficient filtering code may cause timing issue, especially for PHY SS and PHY CTRL.

6.2.6 System Trace Output

The last step is to output Trace to PC or AP where Trace is stored. The supported export interfaces include USB3.2 Gen2 x 2, PCIe Gen4 and TPIU. USB3.2 Gen2 x 2 supports transfer data rate of 20 Gbps. PCIe Gen4 supports transfer data rate of 16 Gbps. TPIU supports transfer data rate of 6.4 Gbps in case of 16 pins and 400 MHz clock.

Multi export interfaces provide flexibility to Trace system. Different export interface is chosen according to different user case.

Multi export interfaces provide robustness to Trace system. The coupling level of these export interfaces are very low. If one interface is not working, it is easy to switch to another one, so Trace will not be blocked.

Multi export interfaces provide parallel capability to Trace system. Trace can be divided into USB, PCIe and TPIU, and output in parallel. Therefore, system trace bandwidth is increased significantly.

USB is the primary export interface thanks to its easy connection and high bandwidth. On the contrary, a small customized PC is necessary for PCIe connection. However, if it is AP and CP solution, PCIe will become the primary export interface for Trace, and the customized PC is not required any more. TPIU is the most stable export interface. Less system intervention than USB and PCIe, and Trace is transferred automatically in hardware. TPIU Trace is widely applied in early bring up phase on development board.

6.3 Software Architecture Proposal

Trace hardware is a distributed system, which is determined by the subsystem-based system architecture. In order to improve concurrent performance of trace, CoreSight based trace elements are distributed in different subsystems.

4 concurrent STM paths are designed for below reasons:

- Each STM path has comparable trace traffic. It is not necessary for every subsystem to have its dedicated trace path. Several subsystems may share one trace path, while some specific subsystem has its own trace path.
- Each STM path has guaranteed invasion. To ensure no trace is lost, guaranteed transaction mode is set in STM. This means that current write to STM will block trace from another CPU, so the other CPU has to get stuck until STM is released from previous write.

Correspondingly, trace software is designed as per the feature of subsystem and allocated trace path. Trace hardware is distributed, but the management is centralized. Detailed trace Software Architecture is shown in below figure.

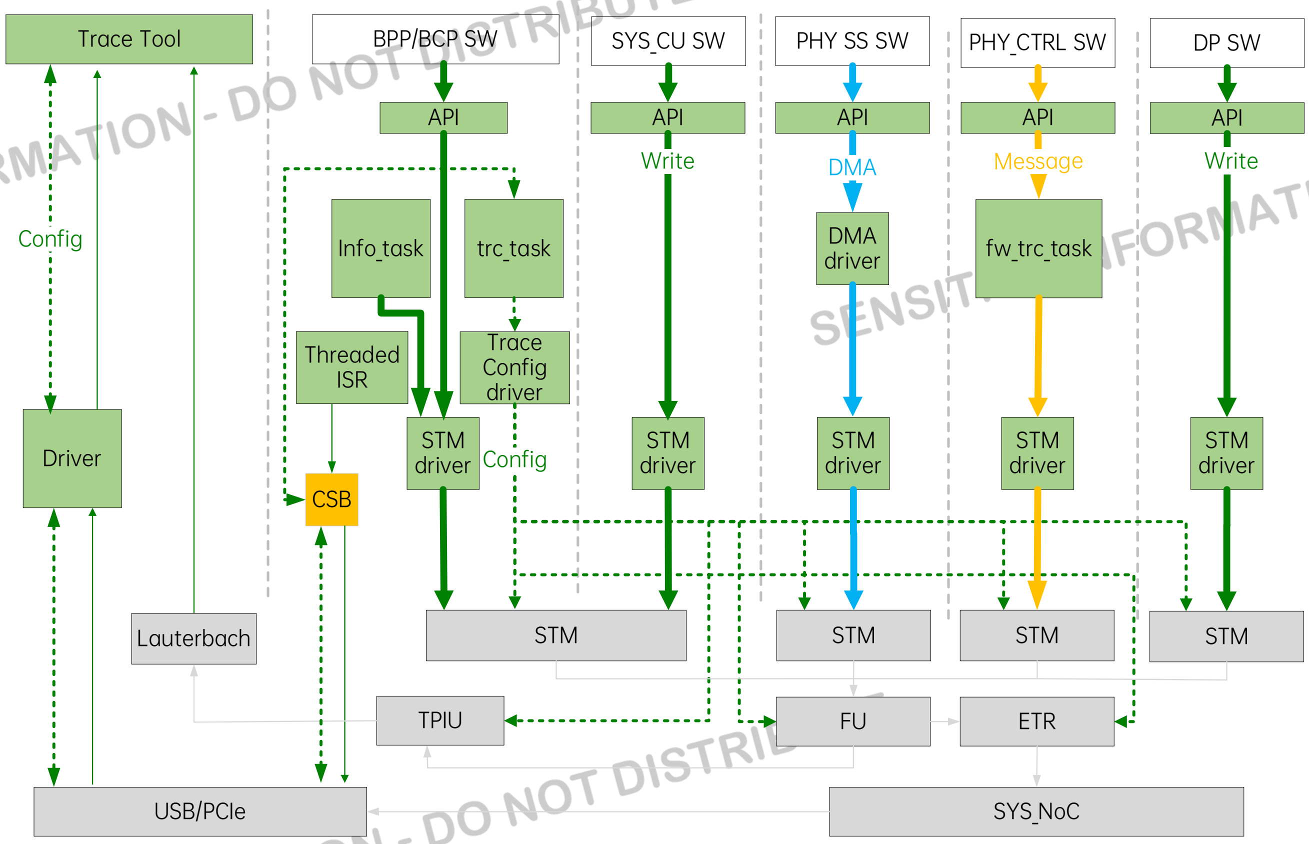

Figure 6-4 Trace Software Architecture

Trace functionality is provided to each subsystem through API. These API functions mainly consist of initialization, write, send and read, details as below table.

API	Description
Init	This API initializes Trace hardware and get it ready for tracing. It also can be used to reset Trace hardware in case of hardware error.

write	This API constructs a SyS-T message with payload sent by each trace source, and directly writes the SyS-T message to CoreSight STM and returns only if write is finished.
send	This API constructs a SyS-T message with payload sent by each trace source, and send a message to notify trace task, then returns immediately.
read	This API reads current trace settings. 1. Returns enablement status for specified trace source 2. Returns trace level for specified trace source

Table 6-1 Trace API List

For write and send API, both formatted String or object trace are supported. Different trace attributes could be defined by user, including trace channel and trace level (FATAL, ERROR, WARNING, INFO, DEBUG and etc.).

info_task: This is a periodic task. Periodic information is handled and sent out in this task. The information includes RTC, decoder list, SW/HW version, current Trace configurations and etc.

trc_task: This is the control task of Trace SW. Initialization of Trace hardware is done in this task during system boot up. After it receives trace configurations from Trace Tool or read them from NVM, it will configure Trace hardware correspondingly.

threaded ISR: When a trace level interrupt is issued, the ISR is called. This interrupt means the expected amount of trace is available in the memory, and it is time to read them out. To reduce the time in interrupt mode, a thread is started to process the time-consuming part for the interrupt, while simple process is done in the ISR. Threaded ISR is deployed on BPP, so it is easy to call CSB driver to send out trace over USB or PCIe.

fw_trc_task: This is the task for PHY CTRL trace. Low enough priority is assigned to it in order not to interfere with the timing critical tasks. It receives trace message from different source on PHY CTRL. When fw_tra_task gets time to run, those trace received previously will be sent to STM, where a unique timestamp generated by the global Trace Timestamp is added in the trace message.

DMA DRV: PHY SS is a CPU timing critical and bare metal sub-system, so DMA is used to transfer trace to STM.

STM DRV: This driver provides interface to trace API for defining trace channel, sending trace message and reading trace filter settings. It runs on different sub systems. Due to this, It should be implemented in C or C++ as requested by each subsystem.

After trace message is processed in STM, a STP v2 stream is output to an on-chip memory or DDR, and an interrupt is issued toward BCP. Then a threaded ISR responds to the interrupt and call CSB API to transfer trace data over USB or PCIe. If trace output interface is set to TPIU, trace will be sent out in hardware without software intervention.

Finally, all sub trace streams are combined into one stream in Funnel, then it is routed to USB or PCIe or TPIU.

6.3.1 System Trace HW Setup

Before Trace can work, Trace HW should be setup. This is implemented in trc_task which is running on BCP.

A centralized management mode is designed in trc_task. It retrieves trace configurations from Trace Tool, then setups every Trace HW instance accordingly. trc_task is the unique task which is responsible for configuring the HW, and other Trace SW are not involved during setup. USB and PCIe does not belong to Trace HW, and they are initiated and configured by CSB. HW setup flow is shown in below figure. Config path is drawn in green dotted line.

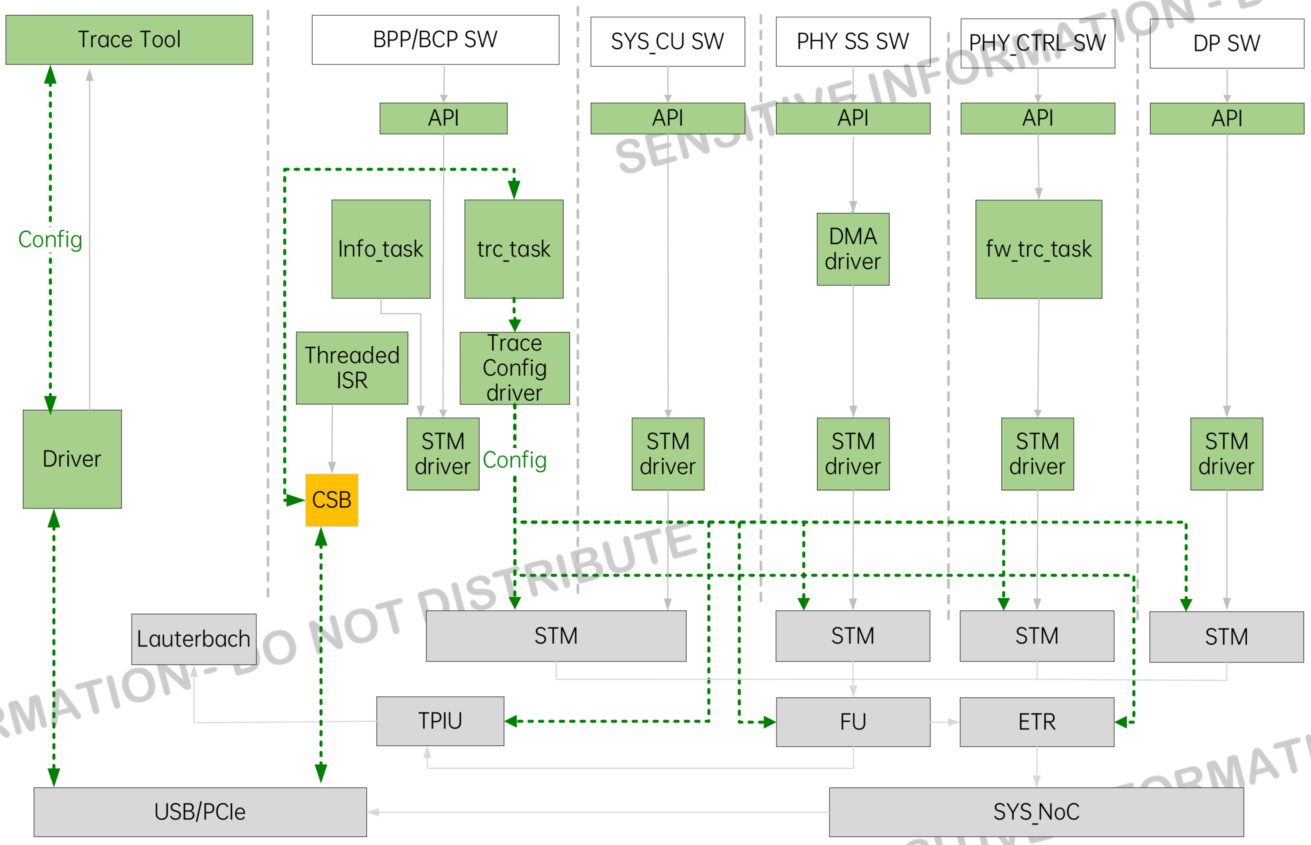

Figure 6-5 Trace HW Setup Flow

Trace config driver accesses registers in trace hardware and provides interfaces to trc_task. In order to do this, trace config driver receives request from trc_task, and translates it into register operations. Not only the registers of STM, but also those of FU, ETR, FU and TPIU should be configured, thus a trace path can be setup.

6.3.2 System Trace Data Flow

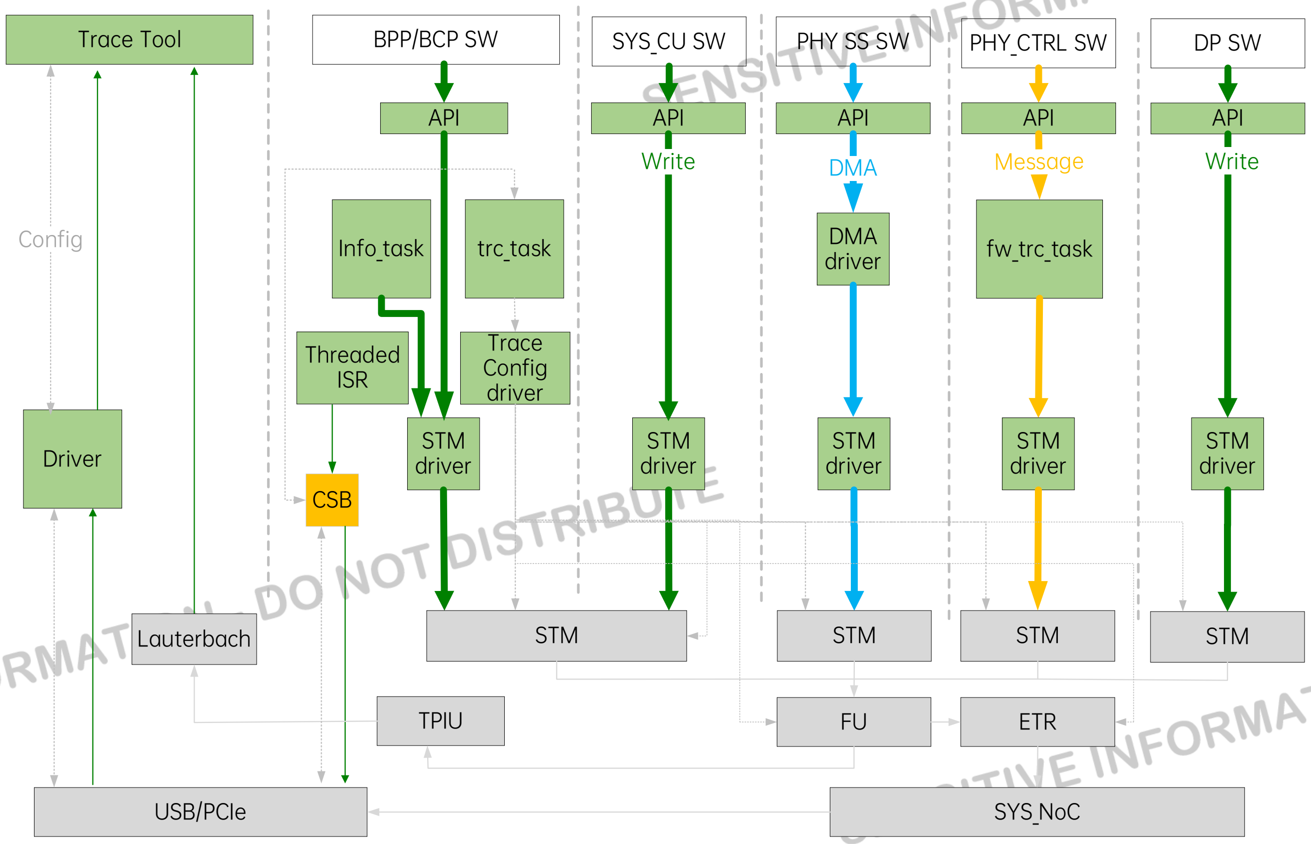

Figure 6-6 Trace Data Flow

Trace is sent by source, and steams though export interface, finally reaches to sink. Each subsystem is the trace source, and Trace Tool is trace sink. The export interfaces include USB3.2 Gen2 x 2, PCIe Gen4 and TPIU.

Each source calling API to write trace independently, then trace is written to STM. Specific transfer type is designed for each subsystem. Each subsystem can have only one transfer type, otherwise there will be trace disordering issue, because transfer in different types can not be synced. For example, let us assume that BCP supports direct both write and deferred write. The 1st trace is written to STM in deferred mode, and the 2nd trace is written to STM in direct mode. The bad result is that the 1st race could be sent to STM after the 2nd trace.

Subsystem	Transfer Type
BPP	Direct write
BCP	Direct write
SYS_CU	Direct write

DP uC	Direct write
PHY_SS	DMA write
PHY_CTRL	Deferred write in fw_trc_task

Table 6-2 Transfer Type of Subsystem

The transfer type is transparent to trace user. Trace user does not need to choose it and it will be automatically determined in Trace API.

6.3.3 Big Trace Data

Big trace data means too large trace which is generated at a super high speed. In general, the request of big trace data comes from PHY SS, like LLR and I/Q data. This super high-speed trace challenges the capability of trace system. To solve this problem, some specific solution is necessary compared to normal trace. There are 2 potential solutions.

Coredump Solution:

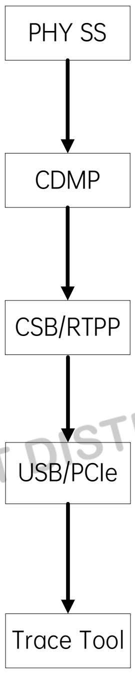

Figure 6-7 Coredump for Big Trace Data

PHY_SS triggers a crash when LLR/IQ data is required, for example, in case of high CRC error rate. Then all related on-chip or DDR memories are dumped by CDMP software. These memories are processed and concatenated to generate complete LLR/IQ data in Trace Tool.

In this way, PHY_SS works as normal, and no extra resource is required for trace. But the LLR/IQ data of interest in memory could be overwritten by new data at crashing time.

Circular Buffer Solution:

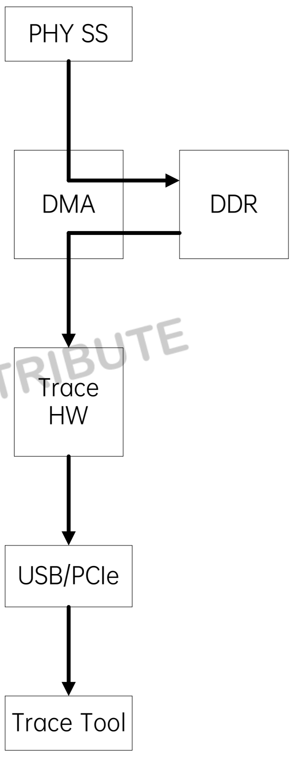

Figure 6-8 Circular Buffer for Big Trace Data

PHY_SS uses DMA to send LLR/IQ data to a circular buffer in DRR, and stops moving once trigger condition is met. This means that the required LLR/IQ trace is now in the circular buffer. Then PHY_SS requests Trace module to output LLR/IQ data. The LLR/IQ data is collected and concatenated to generate complete LLR/IQ data in Trace Tool.

In this way, the space and BW of DDR are required. But much more LLR/IQ data could be stored in DDR.

6.3.4 Trace Output Interface

Trace output interface can be configured to USB or PCIE or TPIU. This output is shared by all trace source in the system, so the available bandwidth of the output interface for trace is limited. Especially for USB and PCIE, trace has to share the bandwidth with IP data. Detailed trace BW is shown in below table.

Output	Total BW (Gbps)	BW for Trace (Gbps)	Comment
USB	20	1.8	excluding CSB overhead ~25% and trace protocol overhead: ~20%
PCIe	16	1.8	excluding CSB overhead ~25% and trace protocol overhead: ~20%
TPIU	6.4	5.12	excluding trace protocol overhead: ~20%

Table 6-3 Trace BW for USB PCIE and TPIU

Every raw trace message from trace source is encapsulated into SyS-T message, and further into STP v2 trace stream. We assume the overhead for SyS-T and STP v2 protocol is 20%. However, the actual overhead should be less than 20%, as both SyS-T and STP v2.2 are quite efficient in packaging the trace message/stream.

For trace over PCIe, 25% overhead of CSB for trace channel is estimated, so the final valid bandwidth for trace is around 1.8 Gbps.

For trace over TPIU, after trace path is setup by software, STP 2 trace stream is automatically sent out by HW without any software involvement. This is the basic trace output interface thanks to its high bandwidth and easy software flow. It is the first trace path for hardware bring-up.

7 System Monitor and Functionalities

System Monitor records the system event which depict system run-time behaviors of interest, including task scheduling and interrupt execution, task timing, task priority, CPU load, memory usage, message queue, semaphore and mutex. The settings for system event trace are configurable.

ThreadX provides a built-in System Monitor. It stores system event trace in a circular buffer in DDR, which enables the most recent “N” events to be available for inspection. The size of each event trace item is 32 bytes, so if this circular buffer is set to 20 MB, it may contain roughly 655360 events.

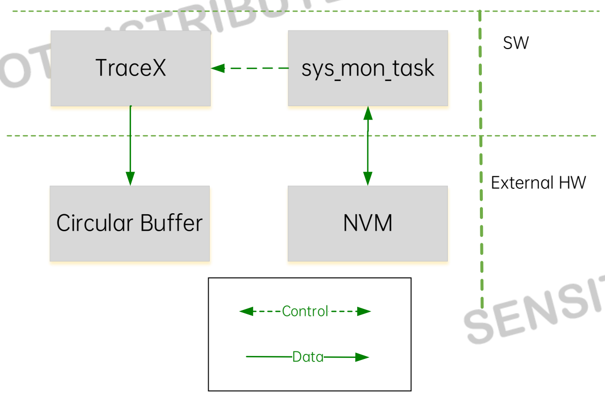

Figure 7-1 System Monitor Software Architecture

sys_mon_task: In this task, Activation/deactivation of TraceX is handled. In addition, the settings for system event trace are received by sys_mon_task, then them will be stored in NVM. The size of circular buffer is also managed here.

System Monitor supports below system event trace.

- Internal thread resume
- Internal thread suspend
- Interrupt Service Routine (ISR) Enter
- Interrupt Service Routine (ISR) Exit
- Internal time-slice
- User-Defined Event
- Block pool allocate
- Block pool create
- Block pool delete
- Block pool information get

- Block pool performance information get
- Block pool system performance information get
- Block pool prioritize
- Block release to pool
- Byte pool allocate memory
- Byte pool create
- Byte pool delete
- Byte pool information get
- Byte pool performance information get
- Byte pool system performance information get
- Byte pool prioritize
- Byte memory release to pool
- Event flags create
- Event flags delete
- Event flags get
- Event flags information get
- Event flags performance information get
- Event flags system performance information get
- Event flags set
- Event flags set notify
- Interrupt enable/disable
- Mutex create
- Mutex delete
- Mutex get
- Mutex information get
- Mutex performance information get
- Mutex system performance information get
- Mutex prioritize
- Mutex put
- Queue create
- Queue delete
- Queue flush
- Queue front send
- Queue information get
- Queue performance information get
- Queue system performance information get
- Queue prioritize
- Queue receive message
- Queue send message

- Queue send notify
- Semaphore ceiling put
- Semaphore create
- Semaphore delete
- Semaphore get
- Semaphore information get
- Semaphore performance information get
- Semaphore system performance information get
- Semaphore prioritize
- Semaphore put
- Semaphore put notify
- Thread create
- Thread delete
- Thread exit/entry notify
- Thread identify
- Thread information get
- Thread performance information get
- Thread performance system information get
- Thread preemption change
- Thread priority change
- Thread relinquish
- Thread reset
- Thread resume
- Thread Sleep
- Thread stack error notify
- Thread suspend
- Thread terminate
- Thread time-slice change
- Thread wait abort
- Time get Time set
- Timer activate
- Timer change
- Timer create
- Timer deactivate
- Timer delete
- Timer information get
- Timer performance information get
- Timer performance system information get

If ThreadX is not chosen for Mariana, TraceX in above figure needs to be implemented by ourselves, and above system event trace shall be supported, though this SW architecture is still applicable for System Monitor.

The circular buffer may be uploaded to the host for analysis at any time, either post mortem or upon a breakpoint. We can use host-based visual tool, like Tracealyzer from Percepio or GTKWave, to decode these system events trace and show a clear view of system running. This way, we can better understand the system behaviors, debug real time issue and optimize system performance.

8 CDMP SW Solution and Functionalities

CDMP is a mechanism to collect each core's memory and registers when the system is in trap. CDMP may be produced on-demand, or automatically upon system exception. These memory and registers are saved in files, then they are passed on to developers upon request where it can be invaluable as a post-mortem snapshot of the system's state at the time of the crash, especially if the fault is hard to reliably reproduce.

8.1 Key Features

CDMP functionality related features:

- Support crash notification among different cores. Halt each core at crashing time triggered by any core.
- If the 2nd crash occurs after 1st crash, it shall be dumped as well
- Critical memory and registers shall be dumped at crashing time.
- Each component shall support generation of unified crash signature
- Crash information shall be logged in the CDMP file, including crash_id, file_id, component_id, cpu_id, etc. This will be easy for crash filtering.
- Support better dump than simple memory dump without context. For example, first offending core info.
- HW & SW info shall be logged in the CDMP file.
- CDMP shall support pre-warn of watchdog timeout
- CDMP shall work after a hardware watchdog reset
- The remaining trace data in trace buffer shall be dumped
- Crash information shall be stored in a non-cleared RAM memory which survives the reset
- CDMP shall be dynamically configurable at run-time by AT commands
- CDMP configurations shall be effective right after AT commands.
- CDMP configurations shall be stored in NVM which can survive a reset

CDMP output related features:

- CDMP files shall be available via USB
- CDMP files shall be available via PCIe
- CDMP files shall be available via JTAG

8.2 CDMP Software and Hardware Boundary Definition

There is not much to do with hardware with respect to CDMP flow. The key is that CTM has to interrupt every subsystem immediately after crash.

CDMP software scope:

1. CDMP configurations management
2. Crash interrupt service routine
3. Collect CDMP sections in all subsystems
4. Generate CDMP files in ELF format with CDMP sections
5. Transfer CDMP files to AP/PC

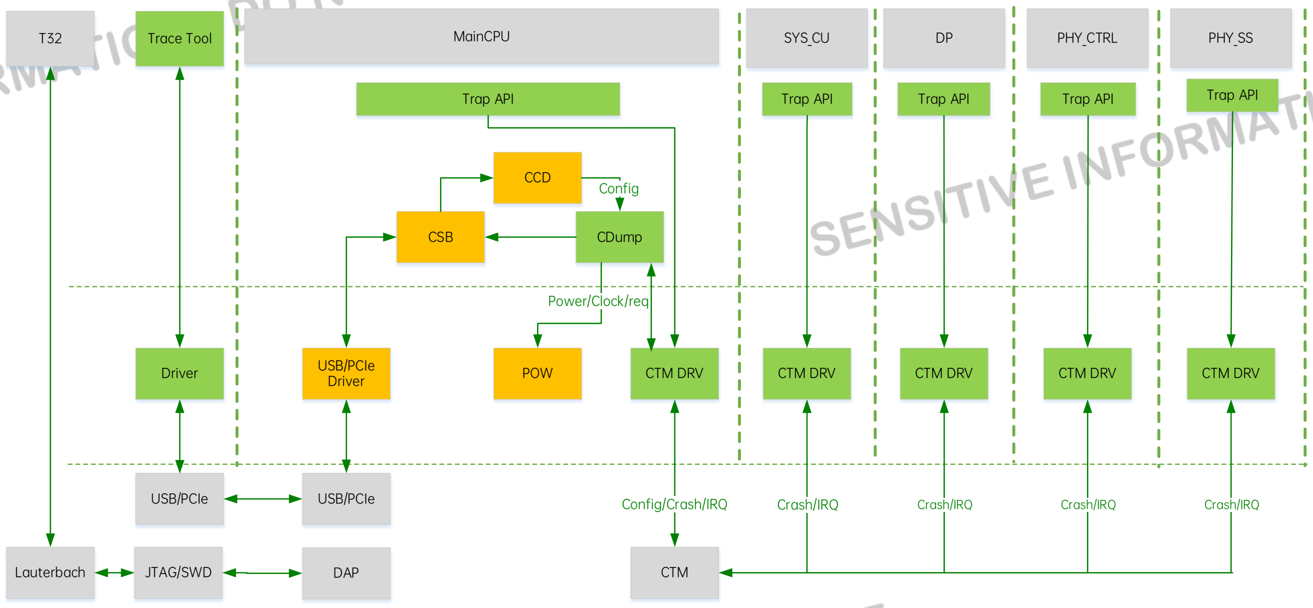

Figure 8-1 CDMP System Overview

CDMP software consists of CDump, CTM driver and trap API.

CDump: It is the control section of CDMP. It is not a task, but broken down to 2 parts which are running in different phases. All CDMP configuration commands are handled in this task. Once a crash is triggered in the system, CDump will take control of CDMP flow.

Trap API: Crash functionality is provided to each subsystem through API. This API forces the whole system into crash mode.

CTM driver: It is running in every subsystem. It is responsible for receiving crash interrupt which is triggered by CTM at crashing time. Then Crash ISR will do the necessary to hold the subsystem and sync with CDump. It is also responsible for triggering a crash interrupt to the whole system after receiving a crash request from API, which forces the whole system into crash mode.

Specially, the CTM drv running on MainCPU provides is in charge of configuring CTM. So other subsystem does not need to take care of the configuration.

8.3 CDMP Flow

Basic CDMP flow is shown as below. There are 2 phases for CDMP process. In the first phase, all CDMP sections are identified, and some of them are copied to DDR for temporary storage, thus all CDMP sections can survive reset. In the second phase, all CDMP sections are collected, and converted to an ELF file. In the ELF file header, some basic notes like crash information and build information will be added. Then this ELF will be sent to AP/PC over USB or PCIe or JTAG. Watchdog reset is an exception, as it only has the second phase. We can see it in below figure.

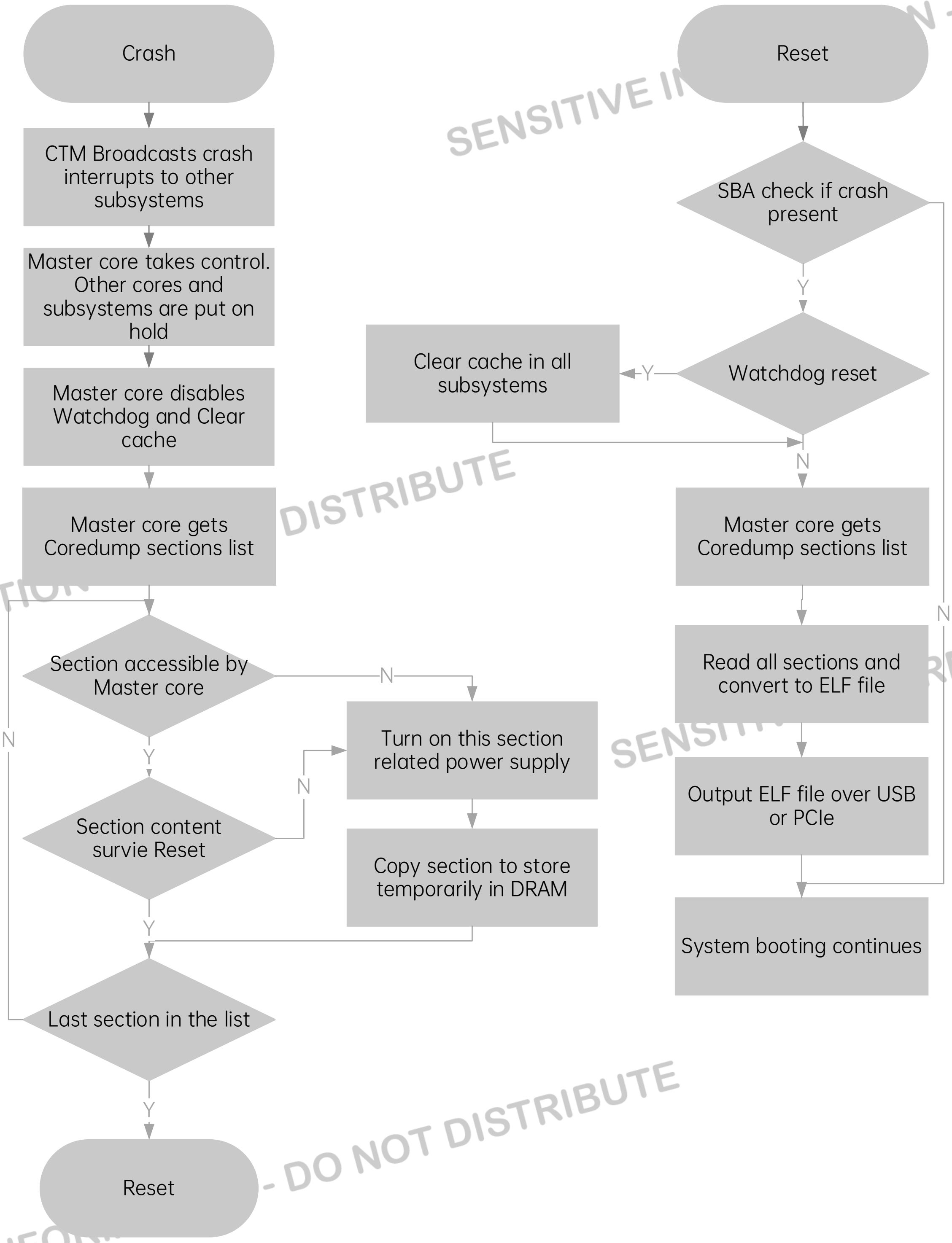
Crash
CTM Broadcasts crash interrupts to other subsystems
Master core takes control. Other cores and subsystems are put on hold
Master core disables Watchdog and Clear cache
Master core gets Coredump sections list
Section accessible by Master core
N
Y
Turn on this section related power supply
Section content survie Reset
N
Y
Copy section to store temporarily in DRAM
N
Last section in the list
Y
Reset
Reset
SBA check if crash present
Y
Watchdog reset
Y
Clear cache in all subsystems
N
Master core gets Coredump sections list
N
Read all sections and convert to ELF file
Output ELF file over USB or PCIe
System booting continues

Figure 8-2 Basic CDMP Flow

Once a crash happens on some subsystem, CTM will broadcasts it to other subsystems in the system. Then master takes control of CDMP flow, and put other cores in the same subsystem and other subsystems on hold. This master core resides in MainCPU, and it is the same core which is booting up the system. There is a pre-built CDMP section list which shows what memory sections are going to be dumped. It also contains other related information, shown as below. Sections are sorted by subsystem attribute for easy process.

	Name	Subsystem	Source Address	Target Address	Length	Master Core Accessible	Reset Survive
Section_0							
Section_1							
…							
Section_n							

Table 8-1 CDMP Section List

Source Address: it is the source address of a CDMP section. If a section is accessible by master core, then it is the address in master core's view, otherwise it is in the view of its subsystem.

Target Address: it is the target address in DDR where this section shall be copied to. If a section is accessible by master core, then this field is NULL.

Master Core Accessible: if a section is accessible by master core, then this field will be set to TRUE.

Reset Survive: if a section can survive a reset, then this field will be set to TRUE.

If a section is not accessible by master core or cannot survive a reset, master core will store it temporarily in DDR. After all sections are handled, a reset is triggered. At bootup, SBA collects all sections, and convert them into ELF file. This ELF file is going to be sent out to AP/PC over USB or PCIe or JTAG.

ELF stands for Executable and Linkable Format. It is a common standard file format for executable files, object code, shared libraries, and core dumps. There are three main types of object files. Executable file is one of them and used to hold core dumps.

Executable file structure is shown in Table 3-2. An ELF header resides at the beginning and holds a "road map" describing the file's organization. A program header table comes after ELF header. Program header table is an array of structures, each describing the attributes of a segment. A segment holds data or stack or register or other specific information. It maps to a section in the CDMP section list.

ELF Header
Program Header Table
Segment 0
Segment 1
…
Segment _n

Table 8-2 Executable File Structure

ELF header structure is defined as below.

```
#define EI_NIDENT       16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

Figure 8-3 ELF Header Structure

e_type: This member identifies the object file type. Only Executable file is used here.

e_machine: This member's value specifies the required architecture for an individual file.

e_version： This member identifies the object file version.

e_entry: This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

e_phoff: This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

e_shoff: This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

e_flags: This member holds processor-specific flags associated with the file.

e_ehsize: This member holds the ELF header's size in bytes.

e_phentsize: This member holds the size in bytes of one entry in the file's program header table; all entries are the same size.

e_phnum: This member holds the number of entries in the program header table. Thus, the product of e_phentsize and e_phnum gives the table's size in bytes. If a file has no program header table, e_phnum holds the value zero.

e_shentsize: This member holds a section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

e_shnum: This member holds the number of entries in the section header table. Thus, the product of e_shentsize and e_shnum gives the section header table's size in bytes. If a file has no section header table, e_shnum holds the value zero.

e_shstrndx: This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value SHN_UNDEF.

Program header structure is defined as below.

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
```

Figure 8-4 Program Header Structure

ELF32: This means the data is 4 bytes long.

p_type: This member tells what kind of segment this array element describes. Only PT_NOTE and PT_LOAD are used here.

p_offset: This member gives the offset from the beginning of the file at which the first byte of the segment resides.

p_vaddr: This member gives the virtual address at which the first byte of the segment resides in memory.

p_paddr: On systems for which physical addressing is relevant, this member is reserved for the segment's physical address.

p_filesz: This member gives the number of bytes in the CDMP file of the segment.

p_memsz: This member gives the number of bytes in the memory of the segment.

p_flags: This member gives flags relevant to the segment.

p_align: This member gives the value to which the segments are aligned in memory and in the file.

A subsystem maps to one ELF header, so there are several ELF headers in the CDMP File for Mariana, shown as below.

MainCPU	ELF Header
	Program Header Table
	PT_NOTE: crash info, SW/HW version, CDump log
	PT_LOAD_0
	…
	PT_LOAD_n
SYS_CU Subsystem	ELF Header
	Program Header Table
	PT_LOAD_0
	…
	PT_LOAD_n
DP Subsystem	ELF Header
	Program Header Table
	PT_LOAD_0
	…
	PT_LOAD_n
PHY Subsystem	ELF Header
	Program Header Table
	PT_LOAD_0
	…
	PT_LOAD_n
…	
ELF Header: ET_NONE	

Table 8-3 CDMP File Structure

There is only one segment of PT_NOTE in the CDMP file, which provides special information about the crash and SW/HW version. CDump log can also be included in this segment, which is used to debug CDump issue. PT_LOAD maps to the segment in the executable file structure, which specifies a dumped section in CDMP phase 1.

8.4 CDMP Software Architecture

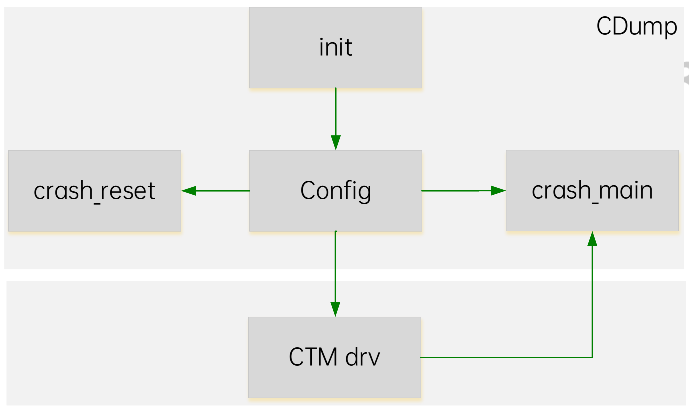

Figure 8-5 CDMP Software Overview

CDump is the control task for CDMP. It is comprised of crash_reset, config, init and crash_main.

init: Initialization of CDUMP is called during system boot up. State machine and related variables in CDUMP are initialed. CDMP related parameters are read to local from NVM.

config: All CDMP related commands are parsed and handled in config, then the configured parameters are stored in NVM. These parameters include enable/disable, output interface and so on.

crash_main: This is the handler of crash. It is in charge of phase 1 of CDMP flow. Once done, a reset request will be triggered.

crash_reset: This is the handler of crash in SBA. It is in charge of phase 2 of CDMP flow. Once done, it will control back to SBA to continue with system booting.

8.5 CDMP Post-mortem Analysis

CDMP tool parses the ELF file, and extracts PT_NOTE part from it. PT_NOTE gives an overview of the crash, including crash signature, file id, line number and so on. SW/HW version is also available in PT_NOTE, so it is easy to find the corresponding software to get decoders and debug symbol files.

In addition, CDMP tool can extract every subsystem from the ELF file. A subsystem may consist of memory and register dumps, which are specified in the CDMP section list. These dumps can be used to recover the context at crashing time.

Further, we can debug the crash in Trace32. CDMP tool can load all dumps for a subsystem into Trace32 Simulator. Then stacks, tasks, variables, registers and etc. can be investigated.

9 Abbreviations

Abbreviations	Descriptions
BLMM	Block level monitoring multiplexer
BM	Break Matrix
CDMP	Core dump
CLMM	Chip level monitoring multiplexer
CTI	Cross trigger input
CTM	Cross trigger matrix
DAP	Debug access port
DBUF	Data buffer
DSRAM	Defined SRAM
DTCU	Debug Test contro unit
ETB	Embedded trace buffer
GTS	Gloable timestamp
MCD	Multi-core debug
OSC	Oscillescope
OCT	On Chip Trace
PTI	Paralell trace interface
SLMM	Subssytem level monitoring multiplexer
STP	Sytem trace protocol
STM	System Trace Module
T32	Lauterbach Trace32
TBB	Trace back bone
TPIU	Trace port interface unit

10 References

1. *coresight_ela_600_technical_reference_manual_101088_0100_04_en.pdf*
2. *coresight_soc600_technical_reference_manual_100806_0400_00_en.pdf*
3. *DDI0314H_coresight_components_trm.pdf*
4. *DDI0424D_dma330_r1p2_trm.pdf*
5. *DDI0528B_coresight_system_trace_macrocell_r0p1_trm.pdf*
6. *embedded_trace_router_architecture_specification_IHI0081A.pdf*
7. *IHI0054B_stm_architecture_specification_v1_1.pdf*
8. *IHI0068C_low_power_interface_spec.pdf*
9. *DGI0012D_coresight_dk_sdg.pdf*
10. *DDI0317-TPIU.pdf*
11. *DDI0316D_dap_lite_trm.pdf*