ENT-AN1051

Application Programming Guide

IEEE 1588 in PHY API

Preliminary

December, 2018

Revision History

Revision	Revision Date	Details of Change
1	July 2018	Revision 1.0 was published in July 2018. It was the first publication of this document.

Contents

Figures

1 Introduction

This programming guide describes programming different units of the 1588 time stamping block. This programming guide also covers different 1588 applications and Y.1731 OAM applications.

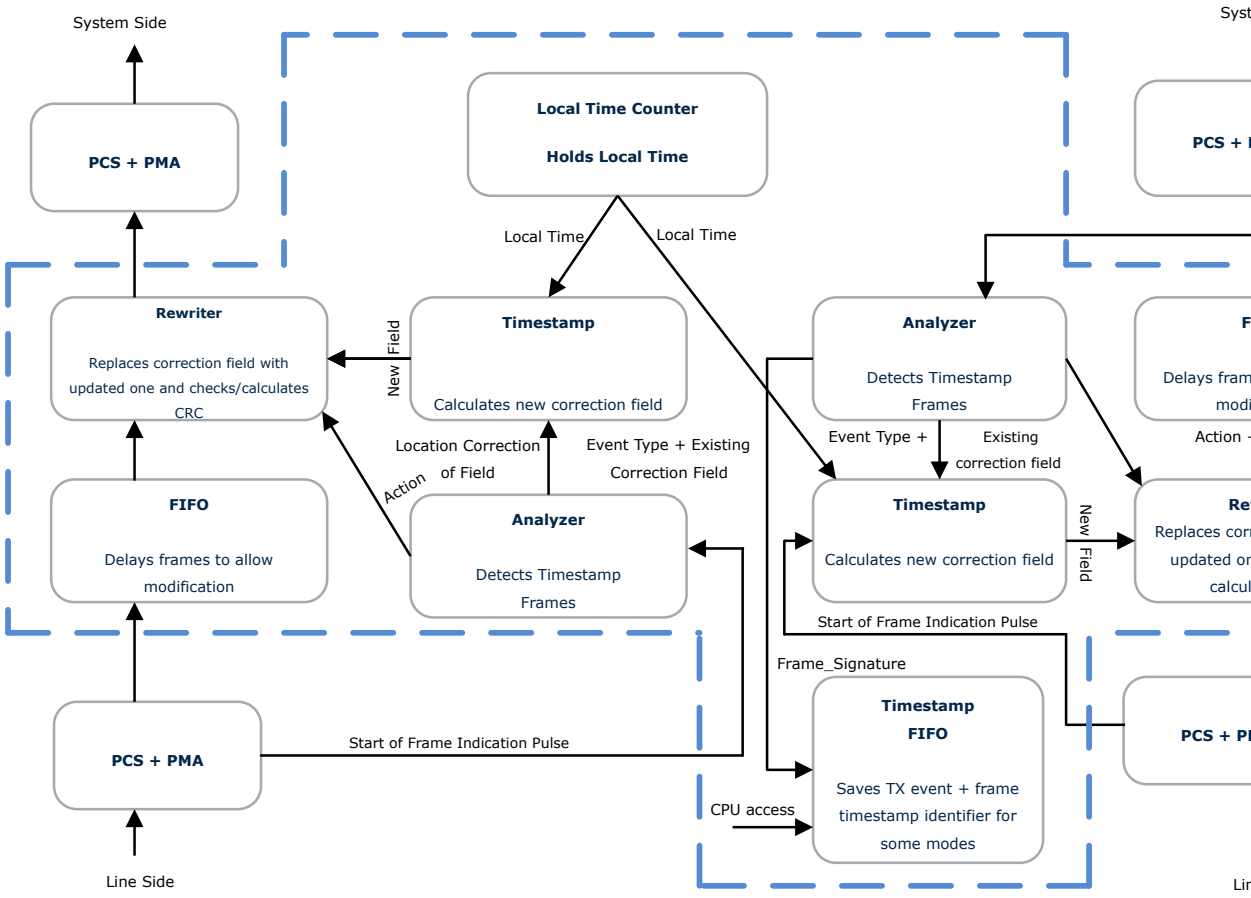
Audience

This will be relevant for 1588 API customers and 1588 application developers.

2 Overview

The following diagram illustrates the block level overview of the 1588 hardware architecture. The blocks within the blue dotted line are part of the time-stamping block and the PCS/PMA blocks are the normal parts of any PHY. Some PHYs might not have a PCS+PMA on the system side (for example, if the part has an xMII interface to the RS+MAC layer). Each block is used in both the ingress (RX) and egress (TX) side. The functionality will be the same but the blocks will be configured to operate slightly differently.

Figure 1 • 1588 Hardware Architecture



2.1 CPU Interface

The following are the serial and parallel CPU interfaces.

- MDIO serial interfaces on the VSC8572, VSC8574, and VSC8487/8-15 device.
- Parallel interface with separate address and data busses on the VSC8492/VSC8494 device.
- MDIO/SPI interface on the VSC8254, VSC8257, VSC8258, VSC8489, VSC8940, VSC8491, VSC8582, VSC8575, and VSC8584. The MDIO interface is a requirement for some devices.

For more information about the CPU interface, see the device datasheet.

2.2 Analyzer

The analyzer parses incoming and outgoing packets looking for PTP frames. The analyzer also determines the offset within the packet required for updating the timestamp—it can be correction field, reserved bytes, or origin timestamp for all PTP and Y.1731 OAM frames.

2.3 Timestamp

The primary function of the timestamp block is to generate a new timestamp_field and/or new correction_field (transparent clocks) for the rewriter block. The timestamp block generates a new_field output that is either a snapshot of the corrected local time (either timestamp structure or a 32-bit timestamp converted to nanoseconds) or a signed (two's complement) 64-bit correction_field.

In the ingress direction, the timestamp block calculates a new timestamp for the rewriter that indicates the earlier time when the corresponding PTP event frame entered the chip.

In the egress direction, the timestamp block calculates the new timestamp for the rewriter in time for the PCS block to transmit the new timestamp field in the frame. In this case, the timestamp field indicates when the corresponding PTP event frame will exit the chip.

2.4 Rewriter

The rewriter block recalculates the Ethernet CRC for the PTP message to modify the contents by writing a new timestamp or clearing certain bytes.

2.5 Timestamp FIFO

The timestamp FIFO is available in egress direction to store timestamps along with the frame signature information. This information can be read out by a CPU to get the packet egress time and can be used in the two-step processing mode to create follow-up messages.

The timestamp FIFO can be read either using the CPU interface (MDIO/SPI register interface) or the dedicated 3-wire push out SPI interface, not both at the same time. If the user configures to read the FIFO using the push out SPI interface, it cannot be read using the CPU interface.

2.6 Local Time Counter

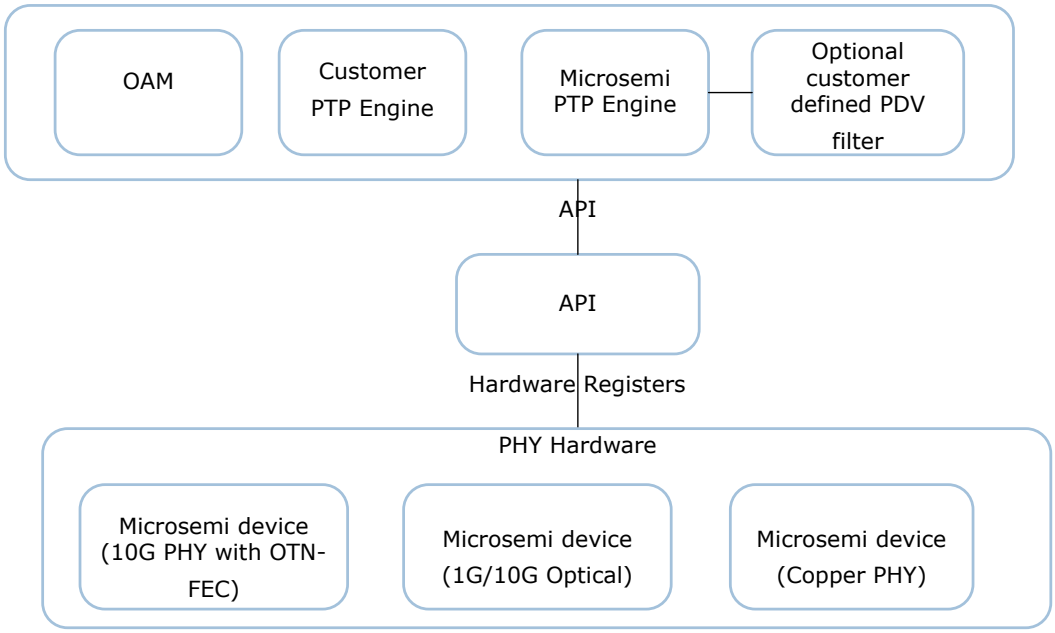
The local time counter (LTC) keeps the local time for the device and the time is monitored and synchronized to an external reference by the CPU. The source clock for the counter is selected externally to be 250 MHz, 156.25 MHz, or 125 MHz depending on the device used, and support for other frequencies may be added in the future. The clock may be a line clock or a dedicated LTC clock. An external clock MUX may be required to select the desired clock source.

2.7 API Overview

The 1588 PHY API implements the interface between different hardware components and the applications. It is a hardware abstraction layer. It is also responsible for setting up the hardware registers in the different 1588 PHY components.

The application can be either the Microsemi PTP engine or the customer's own.

Figure 2 • 1588 API Environment



2.8 Limitations

1588 timestamping is not supported in half-duplex mode.

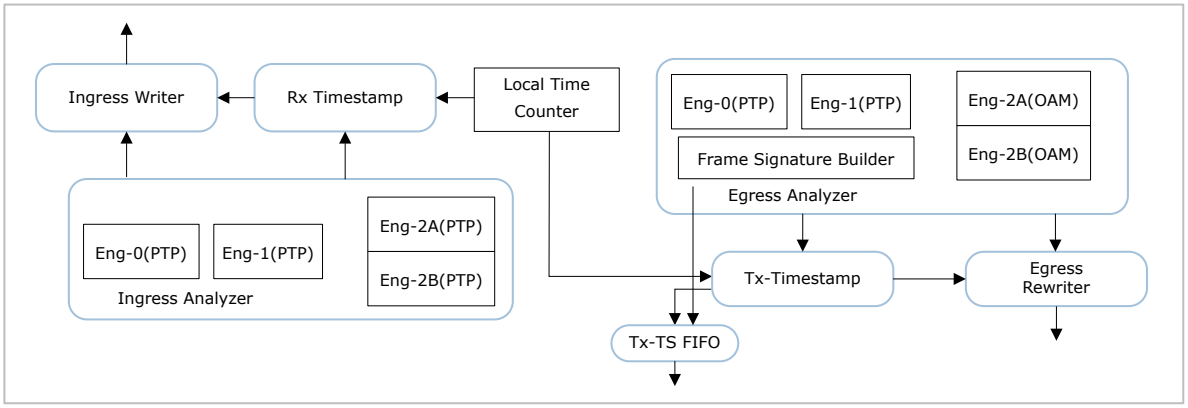
3 Programming the Timestamp Block

The timestamp (TS) block consist of three main components: analyzer, timestamp unit, and rewriter. These three components are present in both ingress and egress sides for each channel.

The analyzer has two types of engines: one for PTP frames and one for OAM frames. The two engine types are mostly identical except that the OAM engine is optimized by removing IP comparators.

The following illustration shows a high level overview of the time stamping block.

Figure 3 • 1588 Timestamp Block



3.1 Timestamp Block Configuration

This section provides guidance on how to program the timestamping block's global configuration through initialization and enabling or disabling the 1588 macro.

The configuration starts with initialization of the timestamp block corresponding to the port/channel that describes the 1588 block reference clock, clock frequency, and so on.

3.1.1 Initialization

Initialization involves configuration of the reference clock source, clock frequency, Rx timestamp position, timestamp length, Tx TS FIFO access mode, and TSFIFO timestamp size.

The two options to add Rx timestamp in PTP frame are:

- Rx timestamp in reserved 4 bytes of PTP header.
- Shrink preamble by 4 bytes and append 4 bytes at the end of frame. In this case, Ethernet CRC will be overwritten by Rx timestamp and a new CRC will be appended after timestamp.

  Note:
  Rx timestamp position must be the same for all ports in the system; otherwise, an ingress timestamp is added in one position based on that port's configuration whereas egress extracts the time from a different position based on that port's configuration.

The PHY has two options for the format of the timestamp that is used internally in the reserved field: 30-bit mode and 32-bit mode.

- 30-bit mode: The value in the reserved field is simply the nanosecCounter, that is [0..999999999].
- 32-bit mode: The value in the reserved field is a 32 bit value and equals: (nanosecCounter + secCounter*10^9) mod 2^32.

The following API is used to initialize the timestamp block on each port.

```
vtss_rc
vtss_phy_ts_init( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_ts_init_conf_t *const conf);
```

3.1.2 Enabling/Disabling the 1588 Macro

The application needs to enable the 1588 macro to timestamp frames. Disabling the timestamp block will 'BYPASS' it.

The following API is used to enable/disable the timestamp block.

```
vtss_rc
vtss_phy_ts_init( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_ts_init_conf_t *const conf);
```

The application can get timestamp block status, which is either enabled/disabled, using the following API.

```
vtss_rc vtss_phy_ts_init_conf_get(const vtss_inst_t     inst,
                        const vtss_port_no_t           port_no,
                        BOOL  *const                   init_done,
                        vtss_phy_ts_init_conf_t        *const conf);
```

3.2 Programming the Analyzer Unit

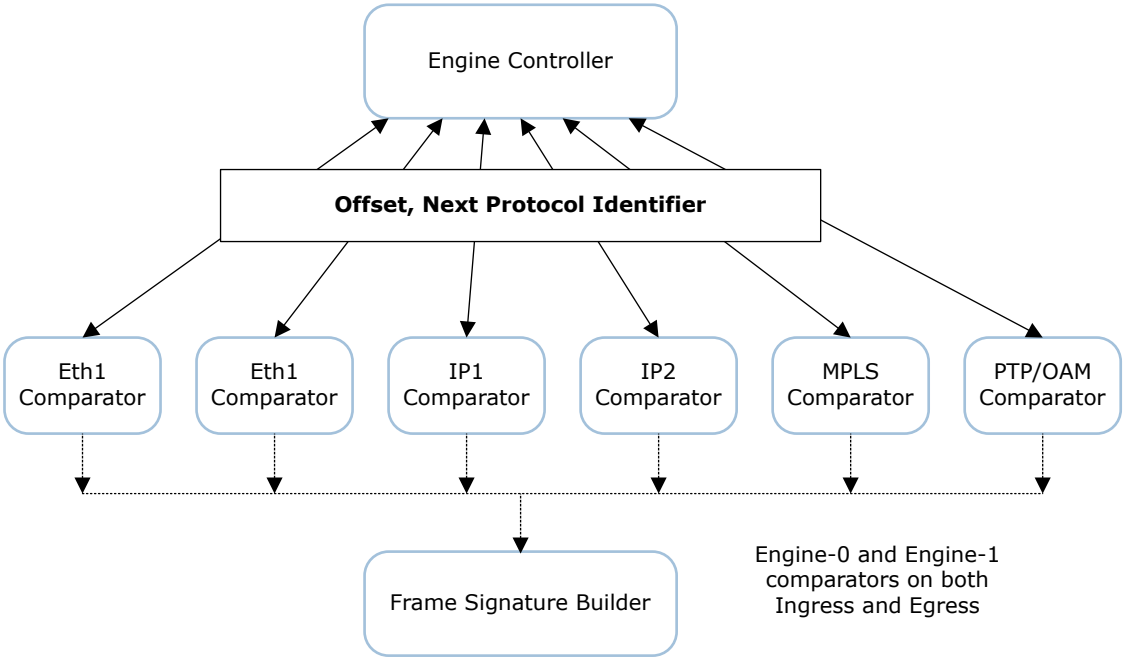
The analyzer unit has Eng-0, Eng-1, and Eng-2. Engines Eng-0, Eng-1, Eng-2A, and Eng-2B are numbered as 0, 1, 2, and 3, respectively. Eng-0 and Eng-1 has the same number of comparators and shares the same architecture. Both engines have two Ethernet comparators, two IP comparators, one MPLS comparator, and one PTP/OAM comparator. These two engines can be used for OAM applications.

Eng-2 is optimized for OAM flows. It is further divided into to two sub engines: Eng-2A and Eng-2B. Eng-2 has two Ethernet comparators, one MPLS comparator, and one PTP/OAM comparator. This engine cannot be used for PTP applications.

Each comparator has a common configuration that applies to all flows. In addition, each flow has its own configuration.

The following illustration shows Eng-0 and Eng-1.

Figure 4 • Eng-0 and Eng-1 Comparators



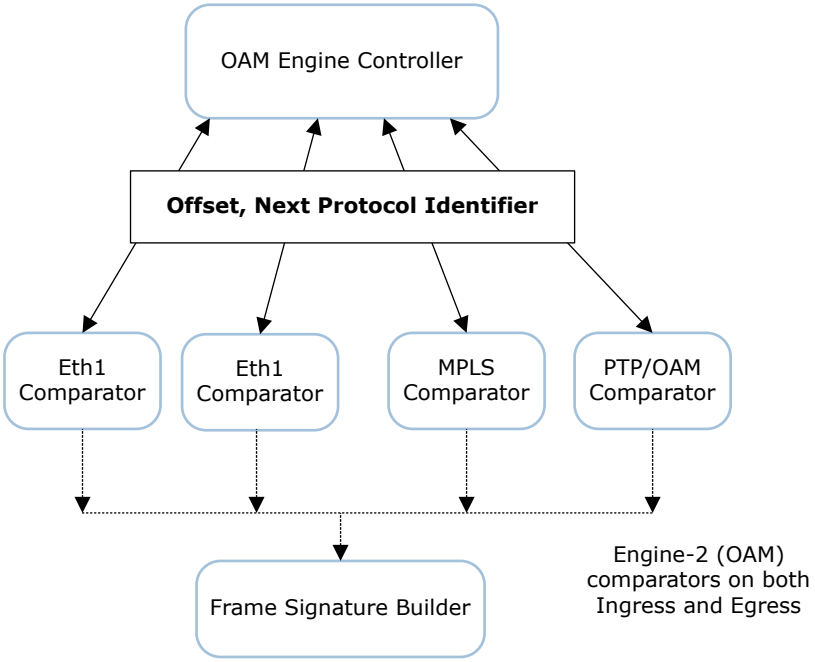
Note:
Dashed lines indicates the frame signature builder in egress analyzer that is absent in ingress analyzer.

PTP engine programming notes:

- Each PTP engine supports eight flows in ETH, IP, and MPLS comparators and six flows in the PTP/OAM comparator. The application has to configure flows to the PTP engines.
- Comparators are selected based on the encapsulation. ETH-1 is always used and for single IP, IP-1 is selected. The application needs to configure the comparators.

The following illustration shows Eng-2 comparators.

Figure 5 • Eng-2 Comparators



Note:
Dashed lines indicates the frame signature builder in egress analyzer that is absent in ingress analyzer.

OAM engine programming notes:

- OAM engines (2A and 2B) have total of eight flows for ETH-1, ETH-2, and MPLS, whereas PTP/OAM has six flows. The application has to associate these flows to OAM engines 2A and 2B.
- OAM engine 2B supports only Ethernet encapsulation, whereas OAM engine 2A can use all the comparators of OAM engine (that is, eth-1, eth-2, and MPLS).
- OAM flows can be shared between the two OAM engines.

For more information on programming the comparators, see 3.2.2 section.

3.2.1 Engine Initialization

The application has to initialize each engine individually. Engine initialization programs the hardware to identify the PTP encapsulation type, number of flows, and the flow match mode. Separate APIs program the ingress and egress engines.

Flow match mode can be either strict or non-strict. If the mode is strict, all comparators need to be matched on the same flow index.

The following API can be used to initialize the ingress engines.

```
vtss_rc
vtss_phy_ts_ingress_engine_init(
const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_ts_engine_t eng_id,
 const vtss_phy_ts_encap_t encap_type,
 const u8 flow_st_index,
 const u8 flow_end_index,
```

```
 const vtss_phy_ts_engine_flow_match_t flow_match_mode
);
```

This API programs the ingress engine encapsulation type, flow match mode, and number of flows.

The following API can be used to initialize the egress engines.

```
vtss_rc
vtss_phy_ts_egress_engine_init(
 const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_ts_engine_t eng_id,
 const vtss_phy_ts_encap_t encap_type,
 const u8 flow_st_index,
 const u8 flow_end_index,
 const vtss_phy_ts_engine_flow_match_t flow_match_mode
);
```

This API programs the egress engine encapsulation type, flow match mode, and number of flows.

Note:
Depending on the chip type, an analyzer configuration is shared by two ports, and therefore, does not initialize separate engines for each port. If the same encapsulation is used by the two ports, both can use the same configuration to configure the engine associated with the port. To share the configuration by two ports, either of the ports can be passed to this API and it can use flow_map within the engine to map flows to the ports.

3.2.2 Programming the Comparators/Flow of the Engine

When the engine is initialized, the comparators need to be configured to identify the PTP/OAM frames. There are separate APIs to configure ingress and egress engines.

Note:
Based on the encapsulation method selected, minimum required comparators are configured by default by the API. These default configurations can be changed or can be configured with new parameters.

The following API programs the specified comparators of the specified ingress engine.

```
vtss_rc
vtss_phy_ts_ingress_engine_conf_set(
const vtss_inst_t inst,
const vtss_port_no_t port_no,
const vtss_phy_ts_engine_t eng_id,
const vtss_phy_ts_engine_flow_conf_t *const flow_conf);
```

The following API programs the specified comparators of the specified egress engine.

```
vtss_rc
vtss_phy_ts_egress_engine_conf_set(
const vtss_inst_t inst,
const vtss_port_no_t port_no,
const vtss_phy_ts_engine_t eng_id,
const vtss_phy_ts_engine_flow_conf_t *const flow_conf);
```

There are two APIs to get the engine configuration, one for ingress and one egress.

The following API gets the existing configuration of the specified engress engine.

```
vtss_rc
vtss_phy_ts_ingress_engine_conf_get(
const vtss_inst_t inst,
const vtss_port_no_t port_no,
const vtss_phy_ts_engine_t eng_id,
const vtss_phy_ts_engine_flow_conf_t *const flow_conf);
```

The following gets the existing configuration of the specified egress engine.

```
vtss_rc
vtss_phy_ts_egress_engine_conf_get(
```

```
const vtss_inst_t inst,
const vtss_port_no_t port_no,
const vtss_phy_ts_engine_t eng_id,
const vtss_phy_ts_engine_flow_conf_t *const flow_conf);
```

3.2.3 Programming the Engine Action

The action conf API internally configures PTP/OAM comparator's flows. PTP/OAM comparators are used to identify the action required for the flows identified by the other comparators. Programming the PTP/OAM comparators sets the actions.

The engine takes the action for configuration based on the operation mode. The operation mode consists of two parts: clock mode and delay mode.

An application can set the ingress engine action using the following API.

```
vtss_rc
vtss_phy_ts_ingress_engine_action_set(
const vtss_inst_t inst,
const vtss_port_no_t port_no,
 const vtss_phy_ts_engine_t eng_id,
 vtss_phy_ts_engine_action_t *const action_conf);
```

An application can set the egress engine action using the following API.

```
vtss_rc
vtss_phy_ts_egress_engine_action_set(
const vtss_inst_t inst,
const vtss_port_no_t port_no,
 const vtss_phy_ts_engine_t eng_id,
 vtss_phy_ts_engine_action_t *const action_conf);
```

3.2.4 Programming the Frame Signature Builder

The frame signature builder generates the signature of the packet and stores it in the timestamp FIFO along with the timestamp to help match frames. This information is used by the CPU to match timestamps in the timestamp FIFO with the actual frames.

The frame signature is up to 16 bytes extracted from programmable locations in the frame. The frame signature can be composed of PTP message type, PTP domain number, source port identity, PTP frame sequence ID, destination IP address, and source IP address fields of the frame.

Ports sharing the timestamp block use a common analyzer register to configure the signature. All the ports within the timestamp block have the same fields in signature in TSFIFO, they are composed based on configurable parameter signature mask. In other words, to configure the signature mask, any of the ports within the timestamp block can be used.

Note:

1. It is possible to configure two different TSFIFO signature masks for two 1588 IP blocks available in the VSC8574/VSC8258 family of devices.
2. The frame signature builder is available on egress only.
3. The frame signature order is the actual field order in a packet. For multiple fields, the first field appearing in the packet will be the first field in the frame signature.

The following are the definitions for the signature mask generation.

```
VTSS_PHY_TS_FIFO_SIG_SRC_IP 0x01 /* Src IP address: inner IP for
 IP-over-IP */
VTSS_PHY_TS_FIFO_SIG_DEST_IP 0x02 /* Dest IP address */
VTSS_PHY_TS_FIFO_SIG_MSG_TYPE 0x04 /* Message type */
VTSS_PHY_TS_FIFO_SIG_DOMAIN_NUM 0x08 /* Domain number */
VTSS_PHY_TS_FIFO_SIG_SOURCE_PORT_ID 0x10 /* Source port identity */
VTSS_PHY_TS_FIFO_SIG_SEQ_ID 0x20 /* PTP frame Sequence ID */
vtss_phy_ts_fifo_sig_mask_t
```

The following API is used to set the signature mask.

```
vtss_rc
vtss_phy_ts_fifo_sig_set( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_ts_fifo_sig_mask_t sig_mask);
```

The following API is used to get the signature mask.

```
vtss_rc
vtss_phy_ts_fifo_sig_get( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 vtss_phy_ts_fifo_sig_mask_t *const sig_mask);
```

3.3 Programming the Timestamp Unit

The following sections provide information required for programming the timestamp unit. There are two time stamping units, one for ingress and one for egress.

3.3.1 Local Latency

Local latency is taken as time interval, which is 48 bit nanoseconds + 16 bit sub-nanoseconds; although the hardware supports nanoseconds within the range 0–2^16.

3.3.1.1 Ingress Latency

Applications can get or set the ingress latency that is used to generate ingress timestamp by the hardware.

<ingressTimestamp> = <ingressMeasuredTimestamp> – ingressLatency

The following API can be used to set the ingress latency from applications.

```
vtss_rc
vtss_phy_ts_ingress_latency_set(const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const TimeInterval *const latency);
```

Applications can get ingress latency using the following API.

```
vtss_rc
vtss_phy_ts_ingress_latency_get(const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const TimeInterval *const latency);
```

3.3.1.2 Egress Latency

Applications can get or set the egress latency that is used to generate the egress timestamp by the hardware.

<egressTimestamp> = <egressMeasuredTimestamp> + egressLatency

The following API can be used to set the egress latency from applications.

```
vtss_rc
vtss_phy_ts_egress_latency_set(const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const TimeInterval *const latency);
```

Applications can get egress latency using the following API.

```
vtss_rc
vtss_phy_ts_egress_latency_get(const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const TimeInterval *const latency);
```

3.3.2 Path Delay

Path delay is an estimate of P2P path delay. The application can use the following API to set or get an estimate of P2P delay for the ingress side.

Note:
Path delay is taken as time interval that is 48 bit nanoseconds + 16 bit sub-nanoseconds though hardware supports nanoseconds in the range 0–2^32.

The following API can be used to set the path delay.

```
vtss_rc
vtss_phy_ts_path_delay_set( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const TimeInterval *const path_delay);
```

The following API can be used to get the path delay.

```
vtss_rc
vtss_phy_ts_path_delay_get( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 TimeInterval *const path_delay);
```

3.3.3 Delay Asymmetry

The PTP engine can use the PHY API to get or set the delay asymmetry when the corresponding events occurred.

Note:
Asymmetry is taken as timeinterval, that is 48-bit nanoseconds + 16-bit sub-nanoseconds; although, the hardware supports scaled nanoseconds, that is 16-bit nanoseconds + 16-bit sub-nanoseconds. In other words, the range of timeinterval is –2^15 to 2^15–2^–16 sub-nanoseconds.

The following API can be used to set the delay asymmetry.

```
vtss_rc
vtss_phy_ts_delay_asymmetry_set( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const TimeInterval *const delay_asym );
```

The following API can be used to get the delay asymmetry.

```
vtss_rc
vtss_phy_ts_delay_asymmetry_get( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const TimeInterval *const delay_asym );
```

3.4 Programming the LTC Unit

The following sections provide information required for programming the LTC unit.

3.4.1 PTP Time

The PTP time has to be synchronized on all modules in a node, therefore the time is loaded into an intermediate register, and the time is applied to the hardware when a hardware trigger is set from a central module (the “load” command).

Setting the timestamp should only be done in situations where the node becomes locked to a new master and the clock offset is greater than “a configurable value”. In this situation, timestamping may result in erroneous residence time and path delay, so timestamping is suspended within the time setting period. When a node is locked to a master, the clock rate adjustment is used to minimize the clock offset.

The following API will be used to program the PTP time.

```
vtss_rc
vtss_phy_ts_ptptime_set( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_timestamp_t *const ts);
```

Note:
While programming the time into the hardware, the application needs to pass the time at next PPS.

The following API programs the PTP time.

```
vtss_rc
vtss_phy_ts_ptptime_get( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 vtss_phy_timestamp_t *const ts);
```

3.4.2 Local Clock Increment/Decrement

The following API provides the increment/decrement operation on the LTC clock value by one nanoseconds.

```
vtss_rc
vtss_phy_ts_ptptime_adj1ns( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const BOOL incr);
```

The application may use this API to adjust the local LTC, which is few nanoseconds behind or forward from the master clock. If the local clock continually deviates from master, use the clock rate adjustment API.

3.4.3 Adjust Local Clock Frequency

The frequency of the internal clock can be adjusted in units of scaled parts per billion, which is defined as the rate in units of ppb and multiplied by 2^16 and contained in a signed 64-bit value. For example, 2.5 ppb is expressed as 0000 0000 0002 8000.

The application can adjust the clock rate/frequency using the following API.

```
vtss_rc
vtss_phy_ts_clock_rateadj_set( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_ts_scaled_ppb_t *const adj);
```

The application can get the frequency/clock rate using the following API.

```
vtss_rc
vtss_phy_ts_clock_rateadj_get( const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 vtss_phy_ts_scaled_ppb_t *const adj);
```

3.5 Programming the TX-Timestamp FIFO

Tx TSFIFO has two interfaces to access the timestamps: the push-out SPI and the CPU interface.

- Push-out SPI interface: An application has to read the timestamp through the serial port.
- CPU interface: Timestamps are gathered into a FIFO and the application can register a callback function to read the FIFO.

  Note:
  Only one of the two interfaces can be used at a time.

3.5.1 Tx TSFIFO Interface Configuration

The egress Tx TSFIFO can store the timestamps of the egress frames. These timestamps can be accessed by the software either by using the SPI push out interface (if the hardware supports it) or by using the CPU register interface of MDIO or SPI.

The parameter `tx_fifo_mode` can be set to one of the values in the following enum that is passed to the API `vtss_phy_ts_init`.

```
typedef enum {
    VTSS_PHY_TS_FIFO_MODE_NORMAL, /**< in this mode, timestamp can be read from normal
 CPU interface */
    VTSS_PHY_TS_FIFO_MODE_SPI,    /**< Timestamps are pushed out on the SPI interface
*/
} vtss_phy_ts_fifo_mode_t;
```

If the option is VTSS_PHY_TS_FIFO_MODE_SPI the CPU has to read the timestamps from the push out SPI interface. When timestamp is captured into the hardware Tx TSFIFO, the CPU is notified. To notify the CPU, the application has to enable the interrupt VTSS_PHY_TS_EGR_TIMESTAMP_CAPTURED using the API vtss_phy_ts_event_enable_set.

For applications accessing timestamps using the CPU register interface, the application is responsible for reading the timestamp from the Tx TSFIFO before the FIFO overflows.

The following API will be used to read/empty the TSFIFO timestamp.

```
vtss_rc
vtss_phy_ts_fifo_empty(const vtss_inst_t       inst,
                       const vtss_port_no_t    port_no);
```

Application should install a callback function for the API to return the timestamp to the application.

Tx TSFIFO read the callback function prototype.

```
typedef void (*vtss_phy_ts_fifo_read)(
 const vtss_inst_t              inst,
           const vtss_port_no_t           port_no,
           const vtss_phy_timestamp_t     *const ts,
           const vtss_phy_ts_fifo_sig_t   *const sig,
           void                           *cntxt,
           const vtss_phy_ts_fifo_status_t status);
```

Install the callback to read data (signature + timestamp) from Tx TSFIFO.

```
 vtss_rc
vtss_phy_ts_fifo_read_install(const vtss_inst_t      inst,
                              vtss_phy_ts_fifo_read  rd_cb,
                                  void                  *cntxt);
```

The registered callback is called for each entry in TSFIFO whenever the vtss_phy_ts_fifo_empty API is called by the user application.

3.6 Interrupt Handling

Enable event generation for a specific event type or group of events.

```
vtss_rc
vtss_phy_ts_event_enable(const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const BOOL enable,
 const vtss_phy_ts_event_t ev_mask);
```

The polling function is called at interrupt or periodically. Interrupt status will be cleared on read.

This will give the event status. Bit set indicates the corresponding event/interrupt has detected.

```
vtss_rc
vtss_phy_ts_event_poll(const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 vtss_phy_ts_event_t *const status);
```

3.7 Statistics Handling

The following are the timestamp statistics gathered by the TS block.

- Number of frames with preambles too short to shrink
- Number of frames received at the timestamp block with FCS error in ingress
- Number of frames received at the timestamp block with FCS error in egress
- Number of frames modified by the timestamp block (rewritter) in ingress
- Number of frames modified by the timestamp block (rewritter) in egress
- Number of frames transmitted to the interface
- Number of timestamped frames dropped without en-queueing to the Tx TSFIFO

The following API gets the counters.

```
vtss_rc
vtss_phy_ts_stats_get(const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 vtss_phy_ts_stats_t *const statistics);
```

4 1588v2 (PTP) Applications

This section describes how to use the API to create different 1588v2 applications and explains the different clocks, delay methods, and encapsulation types supported.

PTP (1588v2) Clocks

- Boundary/ordinary clock
- Transparent clock

PTP Delay Methods

- Peer-to-peer delay measurement method
- End-to-end delay measurement method

PTP Encapsulation Types

- ETH/PTP
- ETH/IP/PTP
- ETH/IP/IP/PTP
- ETH/ETH_PTP
- ETH/ETH/IP/PTP
- ETH/MPLS/IP/PTP
- ETH/MPLS/ETH/PTP
- ETH/MPLS/ETH/IP/PTP
- ETH/MPLS/ACH/PTP

Before configuring any application, there are some steps for initializing and enabling the timestamp block.

4.1 One-step Boundary Clock/Ordinary Clock

The following steps create a one-step boundary clock or one-step ordinary clock.

This example creates a one-step boundary clock for the following key parameters.

- Delay method: end-to-end
- Encapsulation type: ETH/IP/UDP/PTP
- Number of flows: 4
- Number of domains: 4 (multi-domain).

4.1.1 Ingress

Use the following steps to initialize and configure the ingress engine.

1. Initialize an analyzer ingress engine.
2. Configure flows for ingress PTP engine.
3. Configure ingress engine action.

4.1.1.1 Initialize an Analyzer Ingress Engine

The `vtss_phy_ts_ingress_engine_init` API initializes the ingress engine. The following is an example API call to initialize port 1 of API instance 0 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict pattern matching for flows indexed from 0 to 3 on the ingress side.

```
vtss_phy_ts_ingress_engine_init(
 0, /* VTSS API instance 0 */
 1, /* VTSS Port 1 of API instance 0 */
```

```
 VTSS_PHY_TS_PTP_ENGINE_ID_0, /*ingress engine 0 */
 VTSS_PHY_TS_ENCAP_ETH_IP_PTP, /* Encap type Eth/IP/PTP */
 0, /* Flow ID min */
 3, /* Flow ID max., i.e., 4 flows with IDs 0, 1, 2 and 3 will get time-stamped */
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT /* Need to match Eth and IP addresses incase of ETH/IP
 encapsulation type */
);
```

The ingress encapsulation indicates the packet type to the analyzer.

4.1.1.2 Configure Flows for Ingress Engine

For a multi-port timestamp block, either of the ports can be used to configure the ingress engine. The port to which the frame matches is specified in the flow-map parameter.

During the process of updating an engine configuration, the engine mode will be disabled and it will start processing PTP frames after configuration is complete.

If the local time counter is out of sync, and has to be set, then the received PTP packets must be ignored and no PTP packets should be transmitted, but this should be controlled by the application.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* engine enable */
flow_conf.eng_mode = TRUE;
/* ether type is IP for ETH/IP/PTP encapsulation */
flow_conf.flow_conf.ptp.eth1_opt.comm_opt.etype = 0x0800;
/* configure the four flows mentioned in ingress engine initialization, i.e., flow IDs
 0, 1, 2 and 3 for ETH/IP/PTP encapsulation */
For each flow (flow_id from 0 to 3 configured in previous step)
/* Map each flow to the channel which already mapped to the port */
flow_conf.channel_map[flow_id] = VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0 ;
/* Enable the MAC flow */
flow_conf.flow_conf.ptp.eth1_opt.flow_opt[flow_id].flow_en = TRUE;
/* */
flow_conf.flow_conf.ptp.eth1_opt.flow_opt[flow_id].addr_match_mode =
TSS_PHY_TS_ETH_ADDR_MATCH_ANY_MULTICAST;
/* Notify the engine which mac address needs to be matched(dest/src) */
flow_conf.flow_conf.ptp.eth1_opt.flow_opt[flow_id].addr_match_select =
 VTSS_PHY_TS_ETH_MATCH_DEST_ADDR;
/* Notify the engine if it is VLAN flow */
flow_conf.flow_conf.ptp.eth1_opt.flow_opt[flow_id].vlan_check = FALSE;
/* Configure the MAC address of the flow, needs to be time stamped */
flow_conf.flow_conf.ptp.eth1_opt.flow_opt[flow_id].mac_addr = {0x01, 0x00, 0x5e, 0x0,
ox1, 0x81}
 /* Enable the IP flow */ flow_conf.flow_conf.ptp.ip1_opt.flow_opt[flow_id].flow_en =
TRUE;
 /* match src, dest or either IP address */
flow_conf.flow_conf.ptp.ip1_opt.flow_opt[flow_id].match_mode = VTSS_PHY_TS_IP_MATCH_DEST;
/* match any IP address */
flow_conf.flow_conf.ptp.ip1_opt.flow_opt[flow_id].ip_addr.ipv4.mask = 0;
flow_conf.flow_conf.ptp.ip1_opt.flow_opt[flow_id].ip_addr.ipv4.addr = 0;
 /* Set Ip Version to IPv4 */
flow_conf.flow_conf.ptp.ip1_opt.comm_opt.ip_mode = VTSS_PHY_TS_IP_VER_4;
/* Set dest port to 319 to receive PTP event messages */
flow_conf.flow_conf.ptp.ip1_opt.comm_opt.dport_val = 319;
/* Set dest port mask 0 means any port 0xFFFF means exact match to given port */
 flow_conf.flow_conf.ptp.ip1_opt.comm_opt.dport_mask = 0xffff;
vtss_phy_ts_ingress_engine_conf_set( API_INST_DEFAULT,
 1,
VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.1.1.3 Configure Action for Ingress Engine

For a multi-port timestamp block, either of the ports can be used to configure the ingress engine.

While updating an engine configuration, the engine mode will be disabled and the engine will start processing the frames only after the configuration is complete.

```
vtss_phy_ts_engine_action_t ptp_action;
/* ether type is IP for ETH/IP/PTP encapsulation */
ptp_action.action_ptp = TRUE;
/* ether type is IP for ETH/IP/PTP encapsulation */
ptp_action.action.ptp_conf[0].enable = TRUE;
```

```
/* ether type is IP for ETH/IP/PTP encapsulation */
ptp_action.action.ptp_conf[0].channel_map = VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0;
/* ether type is IP for ETH/IP/PTP encapsulation */
ptp_action.action.ptp_conf[0].ptp_conf.range_en = TRUE;
/* ether type is IP for ETH/IP/PTP encapsulation */
ptp_action.action.ptp_conf[0].ptp_conf.domain.range.upper = 3;
/* ether type is IP for ETH/IP/PTP encapsulation */
ptp_action.action.ptp_conf[0].ptp_conf.domain.range.lower = 0;
/* ether type is IP for ETH/IP/PTP encapsulation */
ptp_action.action.ptp_conf[0].clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_BC1STEP;
/* ether type is IP for ETH/IP/PTP encapsulation */
ptp_action.action.ptp_conf[0].delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E;
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.1.2 Egress

Use the following steps to initialize and configure the egress engine.

1. Initialize an analyzer egress engine.
2. Configure the PTP engine for egress.
3. Configure the engine egress action.

4.1.2.1 Initialize an Analyzer Egress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on egress side.

Ingress encapsulation type indicates the packet type to the analyzer.

```
vtss_phy_ts_egress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

4.1.2.2 Configure PTP Egress Engine

The following is an example used to configure the PTP egress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* can use same flow configuration used to configure ingress engine */
vtss_phy_ts_egress_engine_conf_set(API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.1.2.3 Configure Action for Egress Engine

The following is an example used to configure action for egress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
/* can use same action configuration used to configure ingress engine */
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT, 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.1.3 Events and Actions

This section describes what actions the API sets up in the PHY chip in a onestep ordinary/boundary clock (that is, `clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_BC1STEP and delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E`).

Ingress

- `message_type = Sync(0), versionPTP = 2, domainNumber = mydomain`
  - `write(RX_timestamp, reserved); add(Asymmetry, correctionField).`

- `message_type = DelayReq(1), versionPTP = 2, domainNumber = mydomain`
  - `write(RX_timestamp, reserved).`
- `message_type > 1 -> packet is P2P event or a PTP general message`
  - No modification is done.

Egress

- `message_type = Sync(0), versionPTP = 2, domainNumber = mydomain`
  - `write(TX_timestamp, originTimestamp).`
- `message_type = DelayReq(1), versionPTP = 2, domainNumber = 'mydomain'`
  - `save (TX_timestamp, TXFiFo);sub((Asymmetry, correctionField).`
- `message_type > 1 -> packet is P2P event or a PTP general message`
  - No modification is done.

4.2 One-step Peer-to-Peer Transparent Clock

The following steps create a one-step peer-to-peer transparent clock.

4.2.1 Ingress

Use the following steps to initialize and configure the ingress engine.

1. Initialize an analyzer ingress engine.
2. Configure the PTP engine for ingress.
3. Configure the engine ingress action.

4.2.1.1 Initialize an Analyzer Ingress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and a strict matching pattern for flows indexed from 0 to 3 on ingress side.

```
vtss_phy_ts_ingress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 VTSS_PHY_TS_ENCAP_ETH_IP_PTP,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

The ingress encapsulation type indicates packet type to the analyzer.

4.2.1.2 Configure Flows for Ingress Engine

The following is an example used to configure the flows for ingress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* flow configuration used as in sec:4.2.1.2*/
vtss_phy_ts_ingress_engine_conf_set( API_INST_DEFAULT,
 1,
VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.2.1.3 Configure Action for Ingress Engine

The following is an example used to configure action for ingress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
ptp_action.action.ptp_conf[0].clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_TC1STEP;
ptp_action.action.ptp_conf[0].delaym_type = VTSS_PHY_TS_PTP_DELAYM_P2P;
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
```

```
VTSS_PHY_TS_PTP_ENGINE_ID_0,
&ptp_action);
```

4.2.2 Egress

Use the following steps to initialize and configure the egress engine.

1. Initialize an analyzer egress engine.
2. Configure the PTP engine for egress.
3. Configure the engine ingress action.

4.2.2.1 Initialize an Analyzer Egress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on the egress side.

The ingress encapsulation type will tell the packet type to the analyzer.

```
vtss_phy_ts_egress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0 ,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

4.2.2.2 Configure PTP Egress Engine

The following is an example used to configure the PTP egress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* flow configuration used as in sec:4.2.2.1*/
vtss_phy_ts_egress_engine_conf_set(API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.2.2.3 Configure Action for Egress Engine

The following is an example used to configure action for egress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
/* can use same action configuration used to configure ingress engine */
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.2.3 Events and Actions

This section describes what actions the API sets up in the PHY in a one-step P2P transparent clock (that is, `clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_TC1STEP and delaym_type = VTSS_PHY_TS_PTP_DELAYM_P2P`).

Ingress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’,`
  - `write(RX_timestamp, reserved);`
  - `add(PathDelay + Asymmetry, correctionField).`
- `message_type = Pdelay_Req, versionPTP = 2, domainNumber = ‘mydomain’`
  - `write(RX_timestamp, reserved).`
- `message_type > Pdelay_Resp(3), versionPTP = 2, domainNumber = ‘mydomain’`
  - `write(RX_timestamp, reserved).`
  - `add (Asymmetry, correctionField);`
- `message_type > Delay_Req or message_type > 3`

- no modification is done.

Egress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’`
  - `Subtract_add(TX_timestamp, Reserved, correctionField)`
- `message_type = Pdelay_Resp(3), versionPTP = 2, domainNumber = ‘mydomain’`
  - `Subtract_add(TX_timestamp, Reserved, correctionField)`
- `message_type = Pdelay_Req(2), versionPTP = 2, domainNumber = ‘mydomain’`
  - `save (TX_timestamp - Asymmetry, TXFiFo);`

4.3 One-step End-to-End Transparent Clock

The following steps create a one-step end-to-end transparent clock.

4.3.1 Ingress

Use the following steps to initialize and configure the ingress engine.

1. Initialize an analyzer ingress engine.
2. Configure the PTP engine for ingress.
3. Configure the engine ingress action.

4.3.1.1 Initialize an Analyzer Ingress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on ingress side.

```
vtss_phy_ts_ingress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 VTSS_PHY_TS_ENCAP_ETH_IP_PTP,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

The ingress encapsulation type will tell the packet type to the analyzer.

4.3.1.2 Configure Flows for Ingress Engine

The following is an example used to configure the flows for ingress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
vtss_phy_ts_ingress_engine_conf_set( API_INST_DEFAULT,
 1,
VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.3.1.3 Configure Action for Ingress Engine

The following is an example used to configure action for ingress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
ptp_action.action.ptp_conf[0].clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_TC1STEP;
ptp_action.action.ptp_conf[0].delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E;
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.3.2 Egress

Use the following steps to initialize and configure the egress engine.

1. Initialize an analyzer egress engine.
2. Configure the PTP engine for egress.
3. Configure the engine ingress action.

4.3.2.1 Initialize an Analyzer Egress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on egress side.

The egress encapsulation type indicates the packet type to the analyzer.

```
vtss_phy_ts_egress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0 ,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

4.3.2.2 Configure PTP Egress Engine

The following is an example used to configure the PTP egress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* flow configuration used as in sec:4.2.2.1*/
vtss_phy_ts_egress_engine_conf_set(API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.3.2.3 Configure Action for Egress Engine

The following is an example used to configure action for egress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
/* action parameters are same as in sec.: 4.2.1.3 */
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.3.3 Events and Actions

This section describes what actions the API sets up in the PHY in a one-step E2E transparent clock (that is, `clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_TC1STEP` and `delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E`).

Ingress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’`
  - `write(RX_timestamp, reserved);`
  - `add(Asymmetry, correctionField);`
- `message_type = DelayReq(1), versionPTP = 2, domainNumber = ‘mydomain’`
  - `write(RX_timestamp, reserved).`
- `message_type = Pdelay_Req(2), versionPTP = 2, domainNumber = ‘mydomain’`
  - `write(RX_timestamp, reserved).`
- `message_type > Pdelay_Resp(3),versionPTP = 2, domainNumber = ‘mydomain’`
  - `write(RX_timestamp, reserved).`
  - `add (Asymmetry, correctionField);`

Egress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’,`

- Subtract_add(TX_timestamp, Reserved, correctionField);

- message_type = DelayReq(1), versionPTP = 2, domainNumber = ‘mydomain’
  - Subtract_add(TX_timestamp - Asymmetry, Reserved, correctionField);
- message_type = Pdelay_Req, versionPTP = 2, domainNumber = ‘mydomain’
  - Subtract_add(TX_timestamp - Asymmetry, Reserved, correctionField);
- message_type > Pdelay_Resp
  - Subtract_add(TX_timestamp, Reserved, correctionField);

4.4 Two-step Boundary/Ordinary Clock

The following steps create two-step boundary/ordinary clock.

4.4.1 Ingress

Use the following steps to initialize and configure the ingress engine.

1. Initialize an analyzer ingress engine.
2. Configure the PTP engine for ingress.
3. Configure the engine ingress action.

4.4.1.1 Initialize an Analyzer Ingress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on ingress side.

```
vtss_phy_ts_ingress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 VTSS_PHY_TS_ENCAP_ETH_IP_PTP,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

The ingress encapsulation type indicates the packet type to the analyzer.

4.4.1.2 Configure Flows for Ingress Engine

The following is an example used to configure the flows for ingress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* flow conf parameters are same as in sec.: 4.2.1.2 */
vtss_phy_ts_ingress_engine_conf_set( API_INST_DEFAULT,
 1,
VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.4.1.3 Configure Action for Ingress Engine

The following is an example used to configure action for ingress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
ptp_action.action.ptp_conf[0].clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_BC2STEP;
ptp_action.action.ptp_conf[0].delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E;
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.4.2 Egress

Use the following steps to initialize and configure the egress engine.

1. Initialize an analyzer egress engine.

2. Configure the PTP engine for egress.
3. Configure the engine ingress action.

4.4.2.1 Initialize an Analyzer Egress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on egress side.

The ingress encapsulation type indicates the packet type to the analyzer.

```
vtss_phy_ts_egress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

4.4.2.2 Configure PTP Egress Engine

The following is an example used to configure the PTP egress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* flow conf parameters are same as in sec.: 4.2.1.2 */
vtss_phy_ts_egress_engine_conf_set(API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.4.2.3 Configure Action for Egress Engine

The following is an example used to configure action for egress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
/* action parameters are same as in sec.: 4.2.1.3 */
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.4.3 Events and Actions

This section describes what actions the API sets up in the PHY in a two-step E2E boundary clock (that is, `clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_BC2STEP` and `delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E`).

Ingress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’,`
  - `write(RX_timestamp, reserved);`
  - `add(Asymmetry, correctionField);`
- `message_type = DelayReq(1), versionPTP = 2, domainNumber = ‘mydomain’`
  - `write(RX_timestamp, reserved).`

Egress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’,`
  - `save(TX_timestamp, TxFIFO);`
- `message_type = DelayReq(1), versionPTP = 2, domainNumber = ‘mydomain’`
  - `save(TX_timestamp - Asymmetry, TxFIFO);`

4.5 Two-step Peer-to-Peer Transparent Clock

The following steps create a two-step boundary/ordinary clock.

4.5.1 Ingress

Use the following steps to initialize and configure the ingress engine.

1. Initialize an analyzer ingress engine.
2. Configure the PTP engine for ingress.
3. Configure the engine ingress action.

4.5.1.1 Initialize an Analyzer Ingress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on ingress side.

```
vtss_phy_ts_ingress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 VTSS_PHY_TS_ENCAP_ETH_IP_PTP,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

The ingress encapsulation type indicates the packet type to the analyzer.

4.5.1.2 Configure Flows for Ingress Engine

The following is an example used to configure the flows for ingress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
vtss_phy_ts_ingress_engine_conf_set( API_INST_DEFAULT,
 1,
VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.5.1.3 Configure Action for Ingress Engine

The following is an example used to configure action for ingress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
ptp_action.action.ptp_conf[0].clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_TC2STEP;
ptp_action.action.ptp_conf[0].delaym_type = VTSS_PHY_TS_PTP_DELAYM_P2P;
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.5.2 Egress

Use the following steps to initialize and configure the egress engine.

1. Initialize an analyzer egress engine.
2. Configure the PTP engine for egress.
3. Configure the engine ingress action.

4.5.2.1 Initialize an Analyzer Egress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on egress side.

The ingress encapsulation type indicates the packet type to the analyzer.

```
vtss_phy_ts_egress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

4.5.2.2 Configure PTP Egress Engine

The following is an example used to configure the PTP egress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* flow conf parameters are same as in sec.: 4.2.1.2 */
vtss_phy_ts_egress_engine_conf_set(API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.5.2.3 Configure Action for Egress Engine

The following is an example used to configure action for egress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
/* action parameters are same as in sec.: 4.2.1.3 */
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.5.3 Events and Actions

If `clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_TC2STEP and delaym_type = VTSS_PHY_TS_PTP_DELAYM_P2P` then the API sets up the PHY analyzer to do the operations described here.

Ingress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’,`
  - `write(RX_timestamp, reserved);`
- `message_type = Pdelay_Req,`
  - `write(RX_timestamp, reserved);`
- `message_type = Pdelay_Resp,`
  - `write(RX_timestamp, reserved);`

Egress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’,`
  - `save(TX_timestamp, TXFiFo);`
- `message_type = Pdelay_Resp,`
  - `save(TX_timestamp, TXFiFo);`
- `message_type = Pdelay_Req,`
  - `save(TX_timestamp, TXFiFo);`

4.6 Two-step End-to-End Transparent Clock

The following steps create a two-step boundary/ordinary clock.

4.6.1 Ingress

Use the following steps to initialize and configure the ingress engine.

1. Initialize an analyzer ingress engine.
2. Configure the PTP engine for ingress.
3. Configure the engine ingress action.

4.6.1.1 Initialize an Analyzer Ingress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on ingress side.

```
vtss_phy_ts_ingress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 VTSS_PHY_TS_PTP_DELAYM_E2E,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

The ingress encapsulation type will tell the packet type to the analyzer.

4.6.1.2 Configure Flows for Ingress Engine

The following is an example used to configure the flows for ingress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
vtss_phy_ts_ingress_engine_conf_set( API_INST_DEFAULT,
 1,
VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.6.1.3 Configure Action for Ingress Engine

The following is an example used to configure action for ingress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
ptp_action.action.ptp_conf[0].clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_TC2STEP;
ptp_action.action.ptp_conf[0].delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E;
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.6.2 Egress

Use the following steps to initialize and configure the egress engine.

1. Initialize an analyzer egress engine.
2. Configure the PTP engine for egress.
3. Configure the engine ingress action.

4.6.2.1 Initialize an Analyzer Egress Engine

The following is an example to initialize port 1 to use PTP engine 0 with the ETH/IP/UDP/PTP encapsulation type and strict matching pattern for flows indexed from 0 to 3 on egress side.

The ingress encapsulation type indicates the packet type to the analyzer.

```
vtss_phy_ts_egress_engine_init(0,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 0,
 3,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

4.6.2.2 Configure PTP Egress Engine

The following is an example used to configure the PTP egress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
/* flow conf parameters are same as in sec.: 4.2.1.2 */
vtss_phy_ts_egress_engine_conf_set(API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &flow_conf);
```

4.6.2.3 Configure Action for Egress Engine

The following is an example used to configure action for egress engine.

```
vtss_phy_ts_engine_action_t ptp_action;
/* action parameters are same as in sec.: 4.2.1.3 */
vtss_phy_ts_ingress_engine_action_set( API_INST_DEFAULT,
 1,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 &ptp_action);
```

4.6.3 Events and Actions

If `clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_TC2STEP and delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E` then the API sets up the PHY analyzer to do the operations described in the following section.

Ingress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’,`
  - `write(RX_timestamp, reserved);`
- `message_type = DelayReq,`
  - `write(RX_timestamp, reserved);`
- `message_type = Pdelay_Req,`
  - `write(RX_timestamp, reserved);`
- `message_type = Pdelay_Resp,`
  - `write(RX_timestamp, reserved);`

Egress

- `message_type = Sync(0), versionPTP = 2, domainNumber = ‘mydomain’,`
  - `save(TX_timestamp, TXFiFo);`
- `message_type = DelayReq,`
  - `save(TX_timestamp, TXFiFo);`
- `message_type = Pdelay_Resp,`
  - `save(TX_timestamp, TXFiFo);`
- `message_type = Pdelay_Req,`
  - `save(TX_timestamp, TXFiFo);`

5 Y.1731 OAM Applications

ETH-DM frames are used to measure frame delays and delay variation between two maintenance endpoints (MEPs). A MEP sends a ETH-DM request to a peer MEP with timestamp information of the transmission time and the peer MEP that can be used to calculate the delay in the following way:

Frame Delay = RxTimeStampb – TxTimeStamp

This is the one-way frame delay and requires the two MEPs to be time synchronized by an appropriate protocol, (for example, the IEEE 1588 PTP protocol).

If it is not possible to have the two MEPs synchronized, a two-way delay measurement can be used. Here, the MEP sends the ETH-DM request with TxTimeStampf to the peer MEP, which replies with the time of the request arrival (RxTimeStampf) and the transmission time of the reply (TxTimeStampb). With the recording of the arrival time of the reply, the frame delay is calculated as:

Frame Delay = (RxTimeStampb–TxTimeStampf)–(TxTimeStampb–RxTimeStampf)

These two steps (TS block initialization and enable) are common for all applications.

OAM encapsulation types:

- ETH/OAM
- ETH/ETH/OAM
- ETH/MPLS/ETH/OAM
- ETH/MPLS/ACH/OAM

5.1 Initialize Timestamp Block

Initialize the timestamp block whenever the port state is up if the port has a 1588 PHY support.

```
vtss_phy_ts_init_conf_t phy_conf;
/* Specify the tx time stamp length, 4 byte or 10byte */
phy_conf.clk_freq = VTSS_PHY_TS_CLOCK_FREQ_15625M;
/* Specify the tx time stamp length, 4 byte or 10byte */
phy_conf.clk_src = VTSS_PHY_TS_CLOCK_SRC_CLIENT_RX;
 /* Specify the rx timestamp position, in Reserved 4 bytes of PTP header */
 phy_conf.rx_ts_pos = VTSS_PHY_TS_RX_TIMESTAMP_POS_IN_PTP;
/* Specify the rx timestamp length, i.e 30bit or 32bit Rx timestamp */
 phy_conf.rx_ts_len = VTSS_PHY_TS_RX_TIMESTAMP_LEN_30BIT;
/* Enable the software fifo to capture the tx Timestamp, we can enable SPI
 also the another option */
 phy_conf.tx_fifo_mode = VTSS_PHY_TS_FIFO_MODE_NORMAL;
/* Specify the tx time stamp length, 4 byte or 10byte */
 phy_conf.tx_ts_len = VTSS_PHY_TS_FIFO_TIMESTAMP_LEN_4BYTE;
 rc = vtss_phy_ts_init(API_INST_DEFAULT, port_no, &phy_conf);
```

5.2 Enable Timestamp Block

The default mode of the 1588 IP block is bypassed; we need to switch to data mode to process data. The `vtss_phy_ts_mode_set` API is used to change the mode.

```
/* Set the TS block mode to un bypassed */
vtss_phy_ts_mode_set(0, port_no ,TRUE);
```

5.3 One-way Delay Measurements

The encapsulation type for one-way delay measurements is: OAM over Ethernet.

5.3.1 Ingress

Use the following steps to initialize and configure the ingress engine.

1. Initialize an analyzer ingress engine.

2. Configure the PTP engine for ingress.
3. Configure the engine ingress action.

5.3.1.1 Initialize an Analyzer Ingress Engine

The following is an example to initialize port 1 to use OAM engine 2A with the simple ETH/OAM encapsulation type and strict matching pattern for flows indexed from 0 to 3 on ingress side.

```
 vtss_phy_ts_ingress_engine_init(
API_INST_DEFAULT, /* Use default instance */
port, /* The port has PTP PHY */
 /* Use OAM engine 2A, has no PTP support */
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
/* Specify the encapsulation type Ex. OAM over ETH */
 VTSS_PHY_TS_ENCAP_ETH_OAM,
 0, 3, /* 4 flows from 0 to 3 */
/* strict flow match */
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

5.3.1.2 Configure Flows for Ingress Engine

For a multi-port timestamp block, either of the ports can be used to configure the egress engine, port to which frame will be matched is specified in the flow-map parameter.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
 /* modify flow configuration */
 flow_conf.eng_mode = TRUE;
 flow_conf.flow_conf.oam.eth1_opt.comm_opt.etype = 0x8902;
/* configure all four flows mentioned in init */
 for (flow_id = 0; flow_id < 4; flow_id++) {
 flow_conf.channel_map[flow_id] = VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0;
flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].flow_en = TRUE;
flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].addr_match_mode =
VTSS_PHY_TS_ETH_ADDR_MATCH_48BIT;
flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].addr_match_select =
VTSS_PHY_TS_ETH_MATCH_DEST_ADDR;
/* match DA unicast address */
memcpy(flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].mac_addr,
mep_phy_ts_port[port].ing_mac[flow_id-mep_phy_ts_port[port].flow_id_low],
sizeof(mac_addr_t));
flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].vlan_check = FALSE;
}
vtss_phy_ts_ingress_engine_conf_set( API_INST_DEFAULT,
 port,
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
 &flow_conf);
```

5.3.1.3 Configure Action for Ingress Engine

For a multi-port timestamp block, either of the ports can be used to configure the ingress engine.

During the process of updating an engine configuration, the engine mode is disabled and it starts processing frames after completing the configuration.

```
vtss_phy_ts_engine_action_t oam_act;
oam_action.action_ptp = FALSE;
 /* Maximum of 6 actions are possible for each engine */
 oam_act.action.oam_conf[0].enable = TRUE;
 oam_act.action.oam_conf[0].y1731_en = TRUE;
 oam_act.action.oam_conf[0].channel_map =
 VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0;
oam_act.action.oam_conf[0].oam_conf.y1731_oam_conf.range_en = FALSE;
oam_act.action.oam_conf[0].oam_conf.y1731_oam_conf.meg_level.value.val= 0;
oam_act.action.oam_conf[0].oam_conf.y1731_oam_conf.meg_level.value.mask = 0; /* The
action can be set to 1DM or DMM */
oam_act.action.oam_conf[0].oam_conf.y1731_oam_conf.delaym_type =
VTSS_PHY_TS_Y1731_OAM_DELAYM_1DM;
 oam_act.action.oam_conf[1].enable = FALSE;
rc = vtss_phy_ts_ingress_engine_action_set(API_INST_DEFAULT,
 port,
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
 &oam_action);
```

5.3.2 Egress

Use the following steps to initialize and configure the egress engine.

1. Initialize an analyzer egress engine.
2. Configure the PTP engine for egress.
3. Configure the engine engress action.

5.3.2.1 Initialize an Analyzer Egress Engine

The following is an example to initialize any port that supports TS to use OAM engine 2A with the simple ETH/OAM encapsulation type and strict matching pattern for flows indexed from 0 to 3 on egress side.

```
 vtss_phy_ts_egress_engine_init(
API_INST_DEFAULT, /* Use default instance */
port, /* The port has PTP PHY */
 /* Use OAM engine 2A, has no PTP support */
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
/* Specify the encapsulation type Ex. OAM over ETH */
 VTSS_PHY_TS_ENCAP_ETH_OAM,
 0, 3, /* 4 flows from 0 to 3 */
/* strict flow match */
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

5.3.2.2 Configure Flows for Egress Engine

The following is an example used to configure the PTP egress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
 /* the same configuration can be applied to both ingress and egress */
 rc = vtss_phy_ts_egress_engine_conf_get(API_INST_DEFAULT,
 port,
 mep_phy_ts_port[port].engine_id,
 &flow_conf);
```

5.3.2.3 Configure Action for Egress Engine

The following is an example used to configure action for egress engine.

```
vtss_phy_ts_engine_action_t oam_act;
/* the same configuration can be applied to both ingress and egress */
vtss_phy_ts_egress_engine_action_set(API_INST_DEFAULT,
 port,
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
 &oam_action);
```

5.4 Two-way Delay Measurements

This example will show the use of the VTSS_PHY_TS_ENCAP_ETH_ETH_OAM encapsulation type and two-way delay method.

5.4.1 Ingress

Use the following steps to initialize and configure the ingress engine.

1. Initialize an analyzer ingress engine.
2. Configure the PTP engine for ingress.
3. Configure the engine ingress action.

5.4.1.1 Initialize an Analyzer Ingress Engine

The following is an example to initialize port 1 to use OAM engine 2A with the simple the ETH/ETH/OAM encapsulation type and strict matching pattern for flows indexed from 0 to 3 on ingress side.

```
 vtss_phy_ts_ingress_engine_init(
API_INST_DEFAULT, /* Use default instance */
port, /* The port has PTP PHY */
```

```
 /* Use OAM engine 2A, has no PTP support */
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
/* Specify the encapsulation type Ex. OAM over ETH */
VTSS_PHY_TS_ENCAP_ETH_ETH_OAM,
 0, 3, /* 4 flows from 0 to 3 */
/* strict flow match */
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

5.4.1.2 Configure Flows for Ingress Engine

For a multi-port timestamp block, either of the ports can be used to configure the egress engine. The port to which frames will be matched is specified in the flow-map parameter.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
 /* modify flow configuration */
 flow_conf.eng_mode = TRUE;
 flow_conf.flow_conf.oam.eth2_opt.comm_opt.etype = 0x8902;
/* configure all four flows mentioned in init */
 for (flow_id = 0; flow_id < 4; flow_id++) {
 flow_conf.channel_map[flow_id] = VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0;
flow_conf.flow_conf.oam.eth2_opt.comm_opt.etype = 0x88E7;
flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].flow_en = TRUE;
flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].addr_match_mode =
VTSS_PHY_TS_ETH_ADDR_MATCH_48BIT;
flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].addr_match_select =
VTSS_PHY_TS_ETH_MATCH_DEST_ADDR;
/* match DA unicast address */
memcpy(flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].mac_addr,
mep_phy_ts_port[port].ing_mac[flow_id-mep_phy_ts_port[port].flow_id_low],
sizeof(mac_addr_t));
flow_conf.flow_conf.oam.eth1_opt.flow_opt[flow_id].vlan_check = FALSE;
flow_conf.flow_conf.oam.eth2_opt.flow_opt[flow_id].flow_en = TRUE;
flow_conf.flow_conf.oam.eth2_opt.flow_opt[flow_id].addr_match_mode =
VTSS_PHY_TS_ETH_ADDR_MATCH_48BIT;
flow_conf.flow_conf.oam.eth2_opt.flow_opt[flow_id].addr_match_select =
VTSS_PHY_TS_ETH_MATCH_DEST_ADDR;
/* match DA unicast address */
memcpy(flow_conf.flow_conf.oam.eth2_opt.flow_opt[flow_id].mac_addr,
mep_phy_ts_port[port].ing_mac[flow_id-mep_phy_ts_port[port].flow_id_low],
sizeof(mac_addr_t));
flow_conf.flow_conf.oam.eth2_opt.flow_opt[flow_id].vlan_check = FALSE;
}
vtss_phy_ts_ingress_engine_conf_set( API_INST_DEFAULT,
 port,
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
 &flow_conf);
```

5.4.1.3 Configure Action for Ingress Engine

The following is an example used to configure action for ingress engine.

```
vtss_phy_ts_engine_action_t oam_act;
oam_action.action_ptp = FALSE;
 /* Maximum of 6 actions are possible for each engine */
 oam_act.action.oam_conf[0].enable = TRUE;
 oam_act.action.oam_conf[0].y1731_en = TRUE;
 oam_act.action.oam_conf[0].channel_map =
 VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0;
oam_act.action.oam_conf[0].oam_conf.y1731_oam_conf.range_en = FALSE;
oam_act.action.oam_conf[0].oam_conf.y1731_oam_conf.meg_level.value.val= 0;
oam_act.action.oam_conf[0].oam_conf.y1731_oam_conf.meg_level.value.mask = 0; /* The
action can be set to 1DM or DMM */
oam_act.action.oam_conf[0].oam_conf.y1731_oam_conf.delaym_type =
VTSS_PHY_TS_Y1731_OAM_DELAYM_DMM;
 oam_act.action.oam_conf[1].enable = FALSE;
vtss_phy_ts_ingress_engine_action_set(API_INST_DEFAULT,
 port,
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
 &oam_action);
```

5.4.2 Egress

Use the following steps to initialize and configure the egress engine.

1. Initialize an analyzer egress engine.

2. Configure the PTP engine for egress.
3. Configure engine ingress action.

5.4.2.1 Initialize an Analyzer Egress Engine

The following is an example to initialize any port that supports TS to use OAM engine 2A with the simple ETH/OAM encapsulation type and strict matching pattern for flows indexed from 0 to 3 on egress side.

```
 vtss_phy_ts_egress_engine_init(
API_INST_DEFAULT, /* Use default instance */
port, /* The port has PTP PHY */
 /* Use OAM engine 2A, has no PTP support */
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
/* Specify the encapsulation type Ex. OAM over ETH */
 VTSS_PHY_TS_ENCAP_ETH_OAM,
 0, 3, /* 4 flows from 0 to 3 */
/* strict flow match */
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
```

5.4.2.2 Configure PTP Egress Engine

The following is an example used to configure the PTP egress engine.

```
vtss_phy_ts_engine_flow_conf_t flow_conf;
 /* the same configuration can be applied to both ingress and egress */
 rc = vtss_phy_ts_egress_engine_conf_get(API_INST_DEFAULT,
 port,
 mep_phy_ts_port[port].engine_id,
 &flow_conf);
```

5.4.2.3 Configure Action for Egress Engine

The following is an example used to configure action for egress engine.

```
vtss_phy_ts_engine_action_t oam_act;
/* the same configuration can be applied to both ingress and egress */
vtss_phy_ts_egress_engine_action_set(API_INST_DEFAULT,
 port,
 VTSS_PHY_TS_OAM_ENGINE_ID_2A,
 &oam_action);
```

6 Appendix A

6.1 Sample Code

Tx-Timestamp Read Example

```
vtss_phy_ts_fifo_read_install(inst, my_phy_ts_fifo_read, NULL);
/* A sample callback function to maintain timestamp table at application level */
static void my_phy_ts_fifo_read(
const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_timestamp_t *const fifo_ts,
 const vtss_phy_ts_fifo_sig_t *const sig,
 void *cntxt,
 const vtss_phy_ts_fifo_status_t status)
{
 u64 port_mask;
 int ts_idx;
vtss_ts_timestamp_t ts;
 for (ts_idx = 0; ts_idx < PHY_TS_TABLE_SIZE; ++ts_idx) {
/* Handle the time stamps */
 }
}
```

BC One-step E2E Clock Sample Application

```
/* Vtss Sample Application for BC
If we pass same ing_port_no and egr_port_no to this application that port will be
configured for both ing and egr functionalities of BC App,
and needs be called on both switches.
Application: Ordinary/Boundary clock, 1 step
Delay method: End-to-End delay measurement method
*/
vtss_rc vtss_1588_sample_bc_app(const vtss_inst_t inst,
 const vtss_port_no_t ing_port_no,
 const vtss_port_no_t egr_port_no)
{
 vtss_phy_ts_init_conf_t conf = {VTSS_PHY_TS_CLOCK_FREQ_250M,
VTSS_PHY_TS_CLOCK_SRC_LINE_TX, VTSS_PHY_TS_RX_TIMESTAMP_POS_IN_PTP,
VTSS_PHY_TS_RX_TIMESTAMP_LEN_30BIT, VTSS_PHY_TS_FIFO_MODE_SPI,
VTSS_PHY_TS_FIFO_TIMESTAMP_LEN_10BYTE, 0};
 vtss_phy_ts_engine_flow_conf_t *flow_conf;
 vtss_phy_ts_engine_action_t *ptp_action;
 vtss_rc rc;
 if ((flow_conf =
(vtss_phy_ts_engine_flow_conf_t*)malloc(sizeof(vtss_phy_ts_engine_flow_conf_t))) ==
NULL) {
 printf("Engine flow_conf memory allocation Failed!\n");
 return -1;
 }
 if ((ptp_action =
(vtss_phy_ts_engine_action_t*)malloc(sizeof(vtss_phy_ts_engine_action_t))) == NULL) {
 printf("Engine ptp_action memory allocation Failed!\n");
 return -1;
 }
 memset(flow_conf, 0, sizeof(vtss_phy_ts_engine_flow_conf_t));
 memset(ptp_action, 0, sizeof(vtss_phy_ts_engine_action_t));
 do {
 conf.clk_freq = VTSS_PHY_TS_CLOCK_FREQ_15625M;
 conf.clk_src = VTSS_PHY_TS_CLOCK_SRC_EXTERNAL;
 conf.tx_ts_len = VTSS_PHY_TS_FIFO_TIMESTAMP_LEN_10BYTE;
 conf.tx_fifo_mode = VTSS_PHY_TS_FIFO_MODE_SPI;
 /* Ingress port Configuration for TC operation */
 /* This is for channel 0 */
 rc = vtss_phy_ts_init(*(NULL, ing_port_no, &conf);
 if (rc == VTSS_RC_OK) {
 printf("PHY TS Init Success for channel 0\n");
 } else {
 printf("PHY TS Init Failed for channel 0!\n");
```

```
 break;
 }
 rc = vtss_phy_ts_mode_set (NULL, ing_port_no, TRUE);
 if (rc != VTSS_RC_OK) {
 printf("PHY TS Block Enable Failed for channel 0!\n");
 break;
 }
 rc = vtss_phy_ts_ingress_engine_init(NULL, ing_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 VTSS_PHY_TS_ENCAP_ETH_IP_PTP,
 0,
 0,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
 if (rc == VTSS_RC_OK) {
 printf("PHY TS Engine Init Success\n");
 } else {
 printf("PHY TS Engine Init Failed!\n");
 break;
 }
 rc = vtss_phy_ts_ingress_engine_conf_get(NULL, ing_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 flow_conf);
 if (rc != VTSS_RC_OK) {
 printf("PHY TS Engine Conf_get Failed!\n");
 break;
 }

 /* engine enable */
 flow_conf->eng_mode = TRUE;
 /* Map each flow to the channel which already mapped to the port */
 flow_conf->channel_map[0] = VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0 |
VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH1;
 /* Enable the MAC flow */
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].flow_en = TRUE;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].addr_match_mode =
VTSS_PHY_TS_ETH_ADDR_MATCH_ANY_MULTICAST;
 /* Notify the engine which mac address needs to be matched(dest/src) */
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].addr_match_select =
VTSS_PHY_TS_ETH_MATCH_DEST_ADDR;
 /* Notify the engine if it is VLAN flow */
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].vlan_check = FALSE;
 /* Configure the MAC address of the flow, needs to be time stamped */
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[0] = 0x01;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[1] = 0x00;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[2] = 0x5e;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[3] = 0x00;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[4] = 0x01;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[5] = 0x81;
 /* Enable the IP flow */
 flow_conf->flow_conf.ptp.ip1_opt.flow_opt[0].flow_en = TRUE;
 /* match src, dest or either IP address */
 flow_conf->flow_conf.ptp.ip1_opt.flow_opt[0].match_mode = VTSS_PHY_TS_IP_MATCH_DEST;
 /* match any IP address */
 flow_conf->flow_conf.ptp.ip1_opt.flow_opt[0].ip_addr.ipv4.mask = 0;
 flow_conf->flow_conf.ptp.ip1_opt.flow_opt[0].ip_addr.ipv4.addr = 0;
 /* Set IP Version to IPv4 */
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.ip_mode = VTSS_PHY_TS_IP_VER_4;
 /* Set dest port to 319 to receive PTP event messages */
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.dport_val = 319;
 /* Set dest port mask 0 means any port 0xFFFF means exact match to given port */
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.dport_mask = 0xffff;
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.sport_val = 0;
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.sport_mask = 0;
 rc = vtss_phy_ts_ingress_engine_conf_set(NULL, ing_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 flow_conf);
 if (rc == VTSS_RC_OK) {
 printf("PHY TS Engine Conf_set Success\n");
 } else {
 printf("PHY TS Engine Conf_set Failed!\n");
 break;
 }
 rc = vtss_phy_ts_ingress_engine_action_get(NULL, ing_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 ptp_action);
 if (rc != VTSS_RC_OK) {
 printf("PHY TS Engine Action_get Failed!\n");
```

```
 break;
 }
 ptp_action->action_ptp = TRUE;
 ptp_action->action.ptp_conf[0].enable = TRUE;
 ptp_action->action.ptp_conf[0].channel_map = VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0 |
VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH1;
 ptp_action->action.ptp_conf[0].ptp_conf.range_en = TRUE;
 ptp_action->action.ptp_conf[0].ptp_conf.domain.range.upper = 3;
 ptp_action->action.ptp_conf[0].ptp_conf.domain.range.lower = 0;
 ptp_action->action.ptp_conf[0].clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_BC1STEP;
 ptp_action->action.ptp_conf[0].delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E;
 rc = vtss_phy_ts_ingress_engine_action_set(NULL, ing_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 ptp_action);
 if (rc == VTSS_RC_OK) {
 printf("PHY TS Engine Action_set Success\n");
 } else {
 printf("PHY TS Engine Action_set Failed!\n");
 break;
 }
 /* Egress port Configuration for TC operation */
 /* this is for channel 1 */
 rc = vtss_phy_ts_init(NULL, egr_port_no, &conf);
 if (rc == VTSS_RC_OK) {
 printf("PHY TS Init Success for channel 1\n");
 } else {
 printf("PHY TS Init Failed for channel 1!\n");
 break;
 }
 rc = vtss_phy_ts_mode_set (NULL, egr_port_no, TRUE);
 if (rc != VTSS_RC_OK) {
 printf("PHY TS Block Enable Failed for channel 1!\n");
 break;
 }
 rc = vtss_phy_ts_egress_engine_init(NULL, egr_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 VTSS_PHY_TS_ENCAP_ETH_IP_PTP,
 0,
 0,
 VTSS_PHY_TS_ENG_FLOW_MATCH_STRICT);
 if (rc == VTSS_RC_OK) {
 printf("PHY TS Engine Init Success\n");
 } else {
 printf("PHY TS Engine Init Failed!\n");
 break;
 }
 rc = vtss_phy_ts_egress_engine_conf_get(NULL, egr_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 flow_conf);
 if (rc != VTSS_RC_OK) {
 printf("PHY TS Engine Conf_get Failed!\n");
 break;
 }


/* engine enable */
 flow_conf->eng_mode = TRUE;
 /* Map each flow to the channel which already mapped to the port */
 flow_conf->channel_map[0] = VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0 |
VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH1;
 /* Enable the MAC flow */
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].flow_en = TRUE;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].addr_match_mode =
VTSS_PHY_TS_ETH_ADDR_MATCH_ANY_MULTICAST;
 /* Notify the engine which mac address needs to be matched(dest/src) */
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].addr_match_select =
VTSS_PHY_TS_ETH_MATCH_DEST_ADDR;
 /* Notify the engine if it is VLAN flow */
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].vlan_check = FALSE;
 /* Configure the MAC address of the flow, needs to be time stamped */
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[0] = 0x01;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[1] = 0x00;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[2] = 0x5e;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[3] = 0x00;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[4] = 0x01;
 flow_conf->flow_conf.ptp.eth1_opt.flow_opt[0].mac_addr[5] = 0x81;
 /* Enable the IP flow */
```

```
 flow_conf->flow_conf.ptp.ip1_opt.flow_opt[0].flow_en = TRUE;
 /* match src, dest or either IP address */
 flow_conf->flow_conf.ptp.ip1_opt.flow_opt[0].match_mode = VTSS_PHY_TS_IP_MATCH_DEST;
 /* match any IP address */
 flow_conf->flow_conf.ptp.ip1_opt.flow_opt[0].ip_addr.ipv4.mask = 0;
 flow_conf->flow_conf.ptp.ip1_opt.flow_opt[0].ip_addr.ipv4.addr = 0;
 /* Set IP Version to IPv4 */
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.ip_mode = VTSS_PHY_TS_IP_VER_4;
 /* Set dest port to 319 to receive PTP event messages */
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.dport_val = 319;
 /* Set dest port mask 0 means any port 0xFFFF means exact match to given port */
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.dport_mask = 0xffff;
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.sport_val = 0;
 flow_conf->flow_conf.ptp.ip1_opt.comm_opt.sport_mask = 0;
 rc = vtss_phy_ts_egress_engine_conf_set(NULL, egr_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 flow_conf);
 if (rc == VTSS_RC_OK) {
 printf("PHY TS Engine Conf_set Success\n");
 } else {
 printf("PHY TS Engine Conf_set Failed!\n");
 break;
 }
 rc = vtss_phy_ts_egress_engine_action_get(NULL, egr_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 ptp_action);
 if (rc != VTSS_RC_OK) {
 printf("PHY TS Engine Action_get Failed!\n");
 break;
 }
 ptp_action->action_ptp = TRUE;
 ptp_action->action.ptp_conf[0].enable = TRUE;
 ptp_action->action.ptp_conf[0].channel_map = VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH0 |
VTSS_PHY_TS_ENG_FLOW_VALID_FOR_CH1;
 ptp_action->action.ptp_conf[0].ptp_conf.range_en = TRUE;
 ptp_action->action.ptp_conf[0].ptp_conf.domain.range.upper = 3;
 ptp_action->action.ptp_conf[0].ptp_conf.domain.range.lower = 0;
 ptp_action->action.ptp_conf[0].clk_mode = VTSS_PHY_TS_PTP_CLOCK_MODE_BC1STEP;
 ptp_action->action.ptp_conf[0].delaym_type = VTSS_PHY_TS_PTP_DELAYM_E2E;
 rc = vtss_phy_ts_egress_engine_action_set(NULL, egr_port_no,
 VTSS_PHY_TS_PTP_ENGINE_ID_0,
 ptp_action);
 if (rc == VTSS_RC_OK) {
 printf("PHY TS Engine Action_set Success\n");
 } else {
 printf("PHY TS Engine Action_set Failed!\n");
 break;
 }
 } while (0);
 free(flow_conf);
 free(ptp_action);
 return VTSS_RC_OK;
}
```

6.1.1 Sample Code

```
vtss_phy_ts_fifo_read_install(inst, my_phy_ts_fifo_read, NULL);
/* A sample callback function to maintain timestamp table at application level */
static void my_phy_ts_fifo_read(
const vtss_inst_t inst,
 const vtss_port_no_t port_no,
 const vtss_phy_timestamp_t *const fifo_ts,
 const vtss_phy_ts_fifo_sig_t *const sig,
 void *cntxt,
 const vtss_phy_ts_fifo_status_t status)
{
 u64 port_mask;
 int ts_idx;
vtss_ts_timestamp_t ts;
 for (ts_idx = 0; ts_idx < PHY_TS_TABLE_SIZE; ++ts_idx) {
/* Handle the time stamps */
 }
}
```

Microsemi Headquarters
One Enterprise, Aliso Viejo,
CA 92656 USA

Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Fax: +1 (949) 215-4996
Email: sales.support@microsemi.com
www.microsemi.com

Microsemi, a wholly owned subsidiary of Microchip Technology Inc. (Nasdaq: MCHP), offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions; security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Learn more at **www.microsemi.com**.

VPPD-04655