Micrel 1588 PTP Developer Guide

Rev 1.3

August 8, 2012

Table of Contents

1 Revision History

Revision	Date	Summary of Changes
1.0	04/13/11	Initial revision.
1.1	06/06/11	Added TSM_CMD_GET_GPS_TS API.
1.2	08/04/11	Corrected clk_adjust_val algorithm.
1.3	08/08/12	Updated APIs.

2 Introduction

This document describes how to use the 1588 PTPv2 engine in KSZ846x device. Precision Time Protocol (PTP) is an application protocol that allows clock synchronization in sub-microsecond. The messages used in the protocol are normally UDP packets in IPv4 or IPv6, although they can be in 802.3 format. The packets are normally in multicast, but a unicast option is supported.

The PTPv2 messages are broken into 2 types: event and general. Event messages require hardware help to generate a timestamp when the message is received or sent, while general messages do not. Refer to PTPv2 Standard for how PTP messages are used.

2.1 Glossaries

Following are acronyms and terms used throughout the document:

● Audio/Video Bridge	AVB
● End-to-end	E2E
● Precision Time Protocol	PTP
● Peer-to-peer	P2P
● Pulse per second	PPS
● Inter-range Instrumentation Group	IRIG
● Network Time Protocol	NTP
● Rapid Spanning Tree Protocol	RSTP
● Master Clock	MC
● Ordinary Clock	OC
● Slave Clock	SC
● Transparent Clock	TC

Syntonization with master clock means the slave clock is running in step with the master clock. The time difference is called the offset from master. Transparent clocks are required to syntonize with the master to have accurate correction field adjustment. Synchronization means the device is completely in sync with the master, where the path delay is known.

3 Using PTP Hardware

The PTP driver is responsible for handling PTP hardware access. Currently only Linux driver is provided.

The driver provides two types of control access: file I/O and ioctl. For direct access where the information can be retrieved immediately ioctl is used as there is no waiting. For response that may take a while to arrive file I/O is used.

The PTP driver exposes two devices: `/dev/ptp_dev` and `/dev/ptp_event`. The first is used for standard clock adjustment using file I/O. The second is used strictly for retrieving events using file I/O. Ioctl can be used on both devices.

Commands sent to the driver using file I/O or ioctl have to be sent separately. For responses using file I/O the data have to be parsed to separate several responses. The data structure used for the response is `struct ptp_udp_msg`. The first field is `len`, which indicates how many bytes is in the response. If there are more data, the next buffer contains another response. See the `tsm_recv` code for the actual implementation.

It is up to the developers to implement code correctly to use the access control APIs provided by the driver. It is assumed a thread is used to read data from the device using file I/O so that reading can be blocked.

The driver maintains a PPS signal at GPIO 6. This can be measured with a scope to compare with another PPS.

There are 3 reserved fields in the PTP message header. The hardware and the driver use two of them to pass information to each other.

The hardware reports the timestamp of incoming PTP message in the 32-bit reserved field. The format is `(((second & 3) << 30) | nanosecond)`. This allows the application to get the timestamp directly from the PTP message payload without doing anything special. As the second part is not complete the application needs to get it from the driver. Alternately the application can maintain a system second and deduct the real second from the timestamp. The driver actually is doing this way instead of reading the hardware clock. Note the timestamp is stored in all the PTP messages, although it is not necessary for PTP general messages.

The application has control of which port to send PTP message. The 8-bit reserved field is used to specify the port destination. A value of 1 means port 1, 2 means port 2, and 3 means both ports. The hardware will not send the message if port destination is 0. For next revision of chip a port destination of 0 is interpreted by the switch engine and the packet will be sent either to both ports or one of them depending on the lookup table. Note in certain configurations the network driver associated with the PTP hardware can automatically insert the port destination in the PTP message header without the application doing anything, but it is expected the application specifies the correct port destination in the standard configuration.

There are 3 registers to hold the transmit timestamps for each port. These registers hold the Sync, Delay_Req/Pdelay_Req, and Pdelay_Resp transmit timestamps. It is necessary to retrieve these timestamps soon after sending the PTP messages, as subsequent sending the same type of message will override the previous result. This is not a problem in normal operation as those event messages are not sent very quickly and often. However, for a 2-step TC implementation it requires careful monitoring of sent event messages. The hardware is designed to run as 1-step TC so it may not be a concern.

The value held in the registers is the same as the incoming timestamp. As the application retrieves those timestamp information with API call the driver supplies the correct second part.

There are two special cases in sending Pdelay_Req and Pdelay_Resp messages. The application only needs to send one Pdelay_Req messages with the destination port 3. The hardware automatically replaces the source port number when the message is sent out. However the application needs to make sure the source port number is zero. Otherwise the hardware calculates a wrong UDP checksum. This will be fixed in next revision of chip.

The other case is for 1-step Pdelay_Resp message hardware needs to know the previous Pdelay_Req receive timestamp. This information needs to be inserted in the 32-bit reserved field of the PTP message header. The format is the same as the incoming timestamp. The hardware will calculate the difference of the Pdelay_Resp transmit time and Pdelay_Req receive time and put it in the correction field of the Pdelay_Resp message.

The hardware adjusts the clock continually for it to run in step with the master clock. The system clock runs in 25 MHz. The unit for adjustment is 2^{-32} ns. So the mathematical formula is `(2^32 * adjustment * 1000000000 / interval / 25000000)` = hardware adjustment unit. If the interval is normalized at 1 second the formula is reduced to `(2^32 * adjustment / 25000000)`. If floating point operation cannot be used and 64-bit division function is unavailable, then the following integer arithmetic code can be used to yield approximate value. The hardware register holds only 30-bit of adjustment, so the range of adjustment is under 6250 microseconds. In normal operation the adjustment should be under 60 microseconds. Putting a better crystal or tuning the load capacitors will yield a better clock.

```
#define MAX_DRIFT_CORR          6250000
#define LOW_DRIFT_CORR          2499981
#define MAX_U32_SHIFT           32
#define MAX_DIVIDER_SHIFT       31

static u32 drift_in_sec(u32 abs_offset, u64 interval64)
{
	u32 abs_drift;
	u32 divider;
	u32 interval;
	u64 drift64;

	/* 2^32 / 1000 */
	interval64 *= 4294967;
	interval64 += (u32)(1 << (MAX_U32_SHIFT - 1));
	interval64 >>= MAX_U32_SHIFT;

	/* interval64 / 1000 */
	interval = (u32) interval64;
	divider = (1 << MAX_DIVIDER_SHIFT);
	divider += interval / 2;
	divider /= interval;

	drift64 = abs_offset;

	/* NANOSEC_IN_SEC / 1000 */
	drift64 *= 1000000;
	drift64 *= divider;
	drift64 += (u32)(1 << (MAX_DIVIDER_SHIFT - 1));
	drift64 >>= MAX_DIVIDER_SHIFT;
	if (drift64 > 0xffffffff)
		abs_drift = 0xffffffff;
	else
		abs_drift = (u32) drift64;
	return abs_drift;
}

static u32 clk_adjust_val(int diff, u32 interval)
{
	u32 adjust;
	u64 adjust64;

	if (0 == diff)
		return 0;
	if (diff < 0)
		adjust = -diff;
	else
		adjust = diff;

	if (interval != NANOSEC_IN_SEC)
		adjust = drift_in_sec(adjust, interval);

	/* 2^32 * adjust * 1000000000 / interval / 25000000 */
	if (adjust >= MAX_DRIFT_CORR)
		adjust = 0x3fffffff;
	else {
		adjust64 = adjust;

		/* 2^32 * 10 / 25 */
		adjust64 *= 1717986918;

		/* 2^32 / 10000000 */
		if (adjust < LOW_DRIFT_CORR)
			adjust64 *= 4295;
		else
			adjust64 *= 429;
		adjust64 += (u32)(1 << (MAX_U32_SHIFT - 1));
		adjust64 >>= MAX_U32_SHIFT;
		if (adjust < LOW_DRIFT_CORR) {
			adjust = (u32) adjust64 + 5;
			adjust /= 10;
		} else {
			adjust = (u32) adjust64;
			if (adjust < 0x199998f3)
				adjust = 0x199998f3;
		}
	}
	if (diff < 0)
```

```
            adjust |= PTP_RATE_DIR << 16;
        return adjust;
}  /* clk_adjust_val */
```

The main 1588 PTP code is in files `ksz_ptp.c` and `ksz_ptp.h`. It is designed to be included in the main network driver. The network driver needs to declare data structures required and implement the following functions: `ptp_read`, `ptp_write`, `ptp_acquire`, and `ptp_release`. In addition, it needs to implement code to support the PTP operation.

The function `ptp_init` should be called during driver initialization. When the device is opened `ptp_start` can be called to initiate PPS. The function `ptp_close`, if implemented, should be called when the device is being closed. The function `ptp_exit` should be called when the driver is unloading.

The function `ptp_init_state` should be called when PTP operation is initiated. It normally is triggered by application doing PTP stack work. The function `ptp_exit_state` should be called when the application stops processing PTP messages.

The shared switch code is in files `ksz_sw.c` and `ksz_sw.h`. It is designed to be included in the main driver. That driver needs to declare data structures required and implement the following functions: `sw_acquire`, `sw_release`, and `exit_mib_read`. The following macros also need to be defined: `SW_R8`, `SW_W8`, `SW_R16`, `SW_W16`, `SW_R32`, `SW_W32`. In addition, the driver needs to implement code to support the switch operation.

Refer to ***Micrel 1588 PTP Application Notes*** to see how the PTP hardware registers are programmed to complete certain operations. Refer to ***KSZ8463 Switch Application Notes*** to see how the switch registers are used. More detailed instructions of using and porting these code are provided by Micrel.

4 Verifying PTP Hardware

Two sample programs are provided to test and verify the functionality of the PTP hardware. The first program, `ptp`, generates simple PTP messages. It can be compiled to run in PC to send PTP messages to the PTP clock. Capturing the messages from the clock can verify some timestamp capabilities. The other program, `ptp_cli`, provides a command line utility to access the PTP hardware. Refer to ***Micrel PTP Utilities User Guide*** on how to use these programs.

4.1 Verifying Clock Adjustment

To completely synchronize the slave clock to a grandmaster clock it requires a PTP stack, an application to process PTP messages and adjust the clock accordingly. OnTime Network provides a sample stack for demonstration only. All questions related to the stack should be forwarded to OnTime Network.

The driver has a testing mechanism to syntonize the clock internally. While the actual offset from master is not known, it can be verified from the displayed messages and the PPS generated that the slave clock follows the master clock closely.

To enable this feature the conditionals `PTP_MONITOR` and `PTP_PROCESS` have to be enabled. Verify syntonization is supported in the driver by checking the PTP syntonization feature is offered after running this command:

```
cat /sys/class/net/eth0/ptp/features
```

After that the following command is used to enable it:

```
echo 600000?? > /sys/class/net/eth0/ptp/features
```

This command is used to disable the feature:

```
echo 000000?? > /sys/class/net/eth0/ptp/features
```

The "??" values should be whatever in the original features.

The utility `ptp_cli` has simple commands to set and adjust the clock.

4.2 Verifying Message Filtering

Normal users should not concern about PTP message filtering. It is documented here for debug purpose only.

The driver also needs to be compiled with the `PTP_MONITOR` conditional. In addition, the `VERIFY_PTP_MSG` conditional should also be defined.

The following command is used to enable message monitoring:

```
echo 400000?? > /sys/class/net/eth0/ptp/features
```

Send PTP messages using the `ptp` program. The driver will display those messages as received. Use the `ptp_cli` utility to turn on and off certain hardware settings to see whether the messages are passed or dropped. For verification of forwarding to the other port connect another PTP clock to the second port and verify the messages received.

4.3 Verifying Trigger Output and Event Detection

The utility `ptp_cli` has commands to enable trigger output and event detection. The GPIO pins can be connected to a scope so the generated waveform can be seen. Without a scope event detection should be enabled to capture all events. From those events a waveform can be constructed. Matching the generated waveform with the output waveform should verify the trigger output operation.

5 Using Linux Drivers

There are 2 versions of the Micrel 1588 PTP evaluation kit: KSZ8463MLI and KSZ8462HLI. Both chips are connected to a Micrel KSZ9692PB SoC for general operation. The KSZ8462HLI chip contains a network engine so its driver is independent and can be easily ported to other systems. The KSZ8463MLI chip contains only a switch engine and so requires other MAC such as the KSZ9692PB network engine. In this case the driver model is more complicated. There is a KSZ8463 driver which uses SPI to access the hardware. It provides a pseudo PHY device interface so that KSZ9692PB network driver can hook on the KSZ8463 driver to program switch registers. Both KSZ9692PB network driver and KSZ8463 driver are independent, but the switch data structure can be shared between them. It is possible to include the KSZ8463 driver code inside the KSZ9692PB network driver so that the pseudo PHY device model is not used to provide a little performance boost.

All KSZ8462, KSZ8463, and KSZ9692PB drivers use the shared switch code in `ksz_sw.c`. Both KSZ8462 and KSZ9692PB network drivers use the shared 1588 PTP code in `ksz_ptp.c`.

5.1 Multiple Network Device Interfaces

The KSZ8463MLI and KSZ8462HLI chips contain a switch with 2 ports. For an Ordinary Clock running in E2E mechansim the PTP stack can treat the device as having one port and run without problem. But for other clock types such as Transparent Clock and Boundary Clock and in P2P mechansim the PTP stack needs to take into consideration the different ports and handle them properly. The PTP stack may like to deal with actual network devices rather than manipulating the PTP message header for port operation. There are ways to accommodate that.

5.1.1 One Simple Network Interface for Each Port

The driver can create another network device for the other port. Suppose the original network device name is `eth0`. The new configuration will allow `eth0` to link with port 1 and `eth1` to port 2. The default hardware configuration is the ports will be independent; there are no forwarding between the ports. This is not very helpful, and more switch setup need to be done to allow forwarding. The idea is to run a RSTP daemon on the switch so that the ports can be blocked and the network traffic be forwarded between the ports properly.

This driver configuration is enabled by specifying the driver variable `multi_dev` to 1. This can be accomplished in 3 ways: change the variable directly in the driver, specify the variable in the Linux command line while building the Linux kernel, and specify a new Linux command line when booting the kernel.

The `multi_dev` variable in the KSZ9692PB network driver is named `lan_multi_dev`. To specify this variable in the Linux command line it has to be in the right format. For KSZ9692PB it is “ksz8692.lan_multi_dev=1.” For KSZ8462 it is “ksz846x_h.multi_dev=1.” The Linux command line in the Linux configuration file is named `CONFIG_CMDLINE`.

To specify the Linux command line during boot it is required to change the `bootarg` environment variable in U-Boot. The whole Linux command line should be put in that variable. The command to set the variable in normal configuration is

```
     set bootargs console=ttyS0,115200 init=/etc/preinit
ksz846x.spi_bus=2 ksz8692.lan_multi_dev=1 ksz846x_h.multi_dev=1
```

5.1.2 One Additional VLAN Interface for Each Port

Another way to use virtual network devices for different ports is through VLAN devices. The Linux kernel allows creating VLAN virtual network devices in which it automatically inserts a correct VLAN tag when sending packets through a VLAN device and forwards VLAN tagged packets to that device before stripping out the VLAN tag. This allows the driver to simulate port forwarding using VLAN id. When a packet with a special VLAN id indicating a certain port is sent the driver will remove the VLAN tag and sends the packet through that port. When a packet is received the driver will insert a proper VLAN tag so that the packet can be passed to the specific network device. The main network device `eth0` is still available and used normally. The command to create a new VLAN device for port 1 is

```
     vconfig add eth0 200
     ifconfig eth0.200 up
```

This driver configuration is enabled by setting `multi_dev` to 2.

5.1.3 One Additional Network Interface for Each Port

The third way is similar to the second but VLAN is not used as the application would like to know the link status of the port but the VLAN device is not actually accessible by the driver.

This driver configuration is an extension of the first one in which after the main device `eth0` is created there will be additional devices named `eth0.200` and `eth0.201` to indicate port 1 and port 2. There are no forwarding between the ports, so more switch setup need to be done.

This driver configuration is enabled by setting `multi_dev` to 3.

5.2 RSTP Support

RSTP hardware support is planned so that network forwarding is done inside the switch rather than by software.

5.3 Sysfs Support

It is possible to access switch registers for switch operation in user space by reading and writing the Sysfs files exposed by the drivers. For KSZ9692PB and KSZ8462 network drivers the files are located at `/sys/class/net/eth0/`. For KSZ89463 SPI driver the location is `/sys/bus/spi/devices/spi2.0/`.

The subdirectory `sw` is for the whole switch. The subdirectories `sw0`, `sw1`, and `sw2` are for individual ports. The subdirectories `mac?` are for static MAC table. The subdirectories `vlan?` are for VLAN table.

Use the command “cat” to read the file to get the value. Use the command “echo” to write to the file to set the value.

```
cat sw1/1_rx
echo 0 > sw1/1_rx
```

There are some special files which require the “echo” command to read. An example is the `reg` file for reading and writing hardware registers:

```
echo 600 > sw/reg
echo 600=22 > sw/reg
```

5.4 PTP Access Inside Network Device

As PTP stack code deal mainly with regular network devices it is a bit inconvenient to open a separate PTP device to access PTP hardware. It is possible to access PTP hardware through network device ioctl calls. The whole PTP APIs can be passed to the PTP driver for processing by using the network device ioctl call number `(SIOCDEVPRIVATE + 15)`. As the file I/O of network device is interpreted as sending or receiving network traffic it is not possible to use that mechanism to communicate with the PTP driver. If timestamp input event is used it is required to open a PTP device to receive the event notifications.

5.5 Getting RX and TX Timestamps From Network Device

From Linux kernel 2.6.38 and up it is possible to get the receive and transmit timestamps directly from the network device interface instead of using PTP APIs to retrieve them. First the hardware timestamp support needs to be turned on inside the driver by using the ioctl call `SIOCSHWTSTAMP`. Next the application needs to read the error packet from the socket error queue by specifying the `MSG_ERRQUEUE` flag in the `recvmsg` call. The packet returned is the whole Ethernet frame so the application needs to parse it to get to the PTP message header. This allows the application to get the PTP message code and sequence id for bookkeeping. The actual timestamp is retrieved using the `cmsghdr` structure inside the `msghdr` structure. The name of that `cmsghdr` structure is `SCM_TIMESTAMPING` and the PTP hardware timestamp is in the third buffer of hardware timestamp. Application needs to provide a bigger data buffer to retrieve all these information. This code is used in the modified PTPd provided by Micrel.

6 PTP APIs

6.1 Data Structures and Definitions

The PTP APIs use the following defined data types and data structures:

Data types	
`u8`	8-bit unsigned value.
`S16`	16-bit signed value.
`u16`	16-bit unsigned value.
`u32`	32-bit unsigned value.
`s64`	64-bit signed value.
`u64`	64-bit unsigned value.

The device ioctl call structure and returned error values:

Device Ioctl Returned Values	
DEV_IOC_OK	Operation is successful.
DEV_IOC_INVALID_SIZE	Parameter size is not correct.
DEV_IOC_INVALID_CMD	Command not recognized.
DEV_IOC_INVALID_LEN	
DEV_IOC_UNIT_UNAVAILABLE	No more units available for use.
DEV_IOC_UNIT_USED	Unit is being used.
DEV_IOC_ERROR	Command error.

struct ksz_request	
int size	Size of the request.
int cmd	Main command.
int subcmd	Subcommand.
int output	Output of the request if one exists.
int result	Result of the request.
union param	Parameters of the request.

struct ptp_clock_identity	
u8 addr[8]	8-byte array of 8-bit unsigned value.

struct ptp_utime	
u32 sec	The clock's second value.
u32 nsec	The clock's nanosecond value.

struct tsm_cfg	
u8 cmd	
u8 port	Port number.
u8 enable	Configurations to set.
u8 gmp	Grandmaster port number.
u32 ingress_delay	Ingress delay value.
u16 egress_delay	Egress delay value.

struct tsm_clock_set	
`u8 cmd`	
`u32 timestamp`	The clock's eventual nanosecond value.
`u32 nsec`	The clock's second value.
`u32 sec`	The clock's nanosecond value.
`u8 reserved[5]`	

struct tsm_clock_correct	
`u8 cmd`	
`u8 add`	Used to indicate adding or subtracting continual clock adjustment.
`u32 sec`	Not used.
`u32 nsec`	Used to indicate nanosecond adjustment.
`u32 drift`	The drift value related to master clock.
`u32 offset`	Used to indicate adding or subtracting nanosecond adjustment.

struct tsm_db	
`u8 cmd`	
`u8 index`	Used to indicate which timestamp to retrieve.
`u16 seqid`	The sequence id.
`u8 mac[2]`	
`u32 sec`	The timestamp's second value.
`u32 nsec`	The timestamp's nanosecond value.
`u32 timestamp`	

struct tsm_get_time	
`u8 cmd`	
`u16 seqid`	The sequence id.
`u8 msg`	The PTP message code.
`u32 sec`	The clock's second value.

`u32 nsec`	The clock's nanosecond value.

`struct tsm_get_gps`	
`u8 cmd`	
`u8 reserved[7]`	
`u16 seqid`	The sequence id.
`u32 sec`	The GPS clock's second value.
`u32 nsec`	The GPS clock's nanosecond value.

`struct ptp_cfg_options`	
`u8 master:1`	Hardware master setting.
`u8 two_step:1`	Hardware 2-step mode setting.
`u8 p2p:1`	Hardware P2P mode setting.
`u8 as:1`	Hardware 802.1AS setting.
`u8 domain_check:1`	Hardware domain check setting.
`u8 udp_csum:1`	Hardware UDP checksum setting.
`u8 unicast:1`	Hardware unicast setting.
`u8 alternate:1`	Hardware alternate master setting.
`u8 delay_assoc:1`	Hardware Delay_Req association setting.
`u8 pdelay_assoc:1`	Hardware Pdelay_Req association setting.
`u8 sync_assoc:1`	Hardware Sync association setting.
`u8 drop_sync:1`	Hardware drop Sync setting.
`u8 priority:1`	Hardware message priority setting.
`u8 reserved1:3`	
`u8 master_set:1`	Master setting changed.
`u8 two_step_set:1`	2-step mode setting changed.
`u8 p2p_set:1`	P2P mode setting changed.
`u8 as_set:1`	802.1AS setting changed.
`u8 domain_check_set:1`	domain check setting changed.
`u8 udp_csum_set:1`	UDP checksum setting changed.
`u8 unicast_set:1`	Unicast setting changed.
`u8 alternate_set:1`	Alternate master setting changed.

u8 delay_assoc_set:1	Delay_Req association setting changed.
u8 pdelay_assoc_set:1	Pdelay_Req association setting changed.
u8 sync_assoc_set:1	Sync association setting changed.
u8 drop_sync_set:1	Drop Sync setting changed.
u8 priority_set:1	Priority setting changed.
u8 reserved_set:2	
u8 domain_set:1	Domain changed.
u8 domain	Domain number.
u8 reserved3	
u32 access_delay	Hardware access delay value.

struct ptp_tsi_info	
u8 cmd	Type of information returned.
u8 unit	The event unit.
u8 signal	Detect signal specified.
u8 num	Number of `struct ptp_utime` followed.
u32 edge	Edges detected in events. Least significant bit is of the first event.
struct ptp_utime t[0]	Time in second and nanosecond.

struct ptp_tsi_options	
u8 tsi	Event unit.
u8 gpi	General purpose input pin.
u8 event	Input event.
u8 flags	Operation flags.
u8 total	Number of event units.
u8 reserved[3]	
u32 timeout	Timeout value in millisecond.

struct ptp_tso_options	
u8 tso	Output unit.
u8 gpo	General purpose output pin.

u8 event	Output event.
u8 flags	Operation flags.
u8 total	Number of output units.
u8 reserved	
u16 cnt	Repeat count.
u32 pulse	Pulse time in nanosecond.
u32 cycle	Cycle time in nanosecond.
u32 sec	Starting time in second.
u32 nsec	Starting time in nanosecond.
u32 iterate	Iteration time in nanosecond.

struct ptp_clk_options	
u32 sec	The clock's second value.
u32 nsec	The clock's nanosecond value.
int drift	The clock's continual adjustment value.
u32 interval	Interval value is normally nanoseconds in 1 second.

struct ptp_ts_options	
u32 timestamp	The hardware timestamp value.
u32 sec	The timestamp's second value.
u32 nsec	The timestamp's nanosecond value.
u8 msg	PTP event message code.
u8 port	The port index.
u16 seqid	Sequence id of PTP event message. Used for delayed response.
u8 mac[2]	Last 2 bytes of source clock identity. Used for delayed response.

struct ptp_delay_values	
u16 rx_latency	The receive latency value.
u16 tx_latency	The transmit latency value.

`short asym_delay`	The asymmetric delay value.
`u16 reserved`	Used to store the peer delay value.

`struct ptp_udp_msg`	
`u16 len`	Length of message.
`u8 data[]`	Message data.

`struct dev_info`	
`int fd`	Device handle.
`u8 *udp_buf`	Data buffer.
`int udp_len`	Buffer length.
`int index`	For data tracking.
`int left`	For data tracking.

6.2 Initialization Functions

6.2.1 tsm_init

```
#define DATA_BUFFER_SIZE  2000

static struct dev_info dev[2];

int tsm_init ( void );
```

Parameters		None.
Return	`int`	0 if successful.
Description		This function opens the devices and does some initialization.

It is assumed two `struct dev_info` are declared for devices `ptp_dev` and `ptp_event`. All other functions use these structures to access the devices. Data buffers will be allocated with the declared size. If the function returns an error code, then there is something wrong with the PTP driver and the application should exit.

6.2.2 tsm_cleanup

```
void tsm_cleanup ( void );
```

Parameters		None.
Return		None.
Description		This function frees the data buffers used by the devices.

6.2.3 tsm_exit

```
int tsm_exit ( void );
```

Parameters		None.
Return	`int`	0 if successful.
Description		This function closes the devices.

Before closing the devices this function sends a command to the driver to indicate the devices are not used anymore. The driver sends something back using file I/O so that the threads reading the device responses can be unblocked and exit properly.

6.3 File I/O

6.3.1 tsm_recv

```
int tsm_recv ( struct dev_info *info, u8 data[], int len );
```

Parameters	`struct dev_info *info`	The device to retrieve data.
	`u8 data[]`	Buffer to store data received.
	`int len`	Size of buffer.
Return	`int`	Number of bytes received.
Description		This function retrieves available data from the device.

If there is no available data this function will be blocked, so it is better used inside a thread.

As the device data buffer can contain several responses, calling this function immediately again can yield another response.

For a response to a command using file I/O the first few bytes of data are exactly the same as the command except the response bit is set. The table below describes all available commands.

Command Code	**No.**	**Meaning**
`TSM_CMD_RESP`	`0x04`	This is a command response.
`TSM_CMD_GET_TIME_RESP`	`0x08`	This is a database get time command response.
`TSM_CMD_CLOCK_SET`	`0x10`	Clock set command.
`TSM_CMD_CLOCK_CORRECT`	`0x20`	Clock correct command.
`TSM_CMD_DB_SET`	`0x30`	Database set command. Not used.
`TSM_CMD_DB_GET`	`0x40`	Database get command.
`TSM_CMD_STAT_CLEAR`	`0x50`	Statistics clear command. Not used.
`TSM_CMD_STAT_GET`	`0x60`	Statistics get command. Not used.
`TSM_CMD_CNF_SET`	`0x70`	Configuration set command.
`TSM_CMD_CNF_GET`	`0x80`	Configuration get command. Not used.
`TSM_CMD_GPIO_SET`	`0x90`	GPIO set command. Not used.
`TSM_CMD_GPIO_GET`	`0xA0`	GPIO get command. Not used.
`TSM_CMD_SET_SECONDS`	`0xB0`	Second set command. Not used.
`TSM_CMD_GET_GPS_TS`	`0xE0`	GPS timestamp get command.
`TSM_CMD_DB_GET_RESRV1`	`0xB0`	Database get reserved1 command. Not used.
`TSM_CMD_DB_GET_RESRV2`	`0xC0`	Database get reserved2 command. Not used.
`TSM_CMD_DB_GET_TIME`	`0xD0`	Database get time command.
`TSM_CMD_DB_SET_TIME`	`0xF0`	Database set time command. Not used.

The `ptp_event` device uses the following command responses to indicate events to application.

Command Code	**No.**	**Meaning**
`PTP_CMD_RESP`	`0x01`	This is a command response.
`PTP_CMD_GET_MSG`	`0x00`	Get raw Ethernet PTP message.
`PTP_CMD_GET_OUTPUT`	`0xE0`	Get output result.

`PTP_CMD_GET_EVENT`	`0xF0`	Get input event.

6.3.2 tsm_send

```
int tsm_send ( u8 data[], int len );
```

Parameters	`u8 data[]`	Data to send.
	`int len`	Number of bytes to send.
Return	`int`	Number of bytes sent.
Description		This function sends command in data format.

This function sends a command to device `ptp_dev` using file I/O as device `ptp_event` does not accept commands using file I/O. The data are one of the `struct tsm_*` data types and the `cmd` field contains the command code in the above table. As the command and response are processed separately it is not guaranteed that every command that expects a response will get one. The first few bytes of the response match the command and that is how the application differentiate responses with same command code.

The following TSM functions are the simple ones that demonstrate how the driver act on the commands. The OnTime application versions are the complete ones, with more parameters. However, those parameters are only used by the application and not the driver. The driver just echos those parameters in the responses.

TSM functions use network format so 32-bit value needs to be converted with htonl and ntohl macros and 16-bit value with htons and ntohs macros.

6.3.3 tsm_cmd_get

```
int tsm_cmd_get ( u8 msg, u8 port, u16 seqid, u8 mac[2] );
```

Parameters	`u8 msg`	PTP message code.
	`u8 port`	Port where message arrives.
Return	`int`	A positive value to indicate success.
Description		This function asks device to return the timestamp of PTP message.

Data		`struct tsm_db`
	`cmd`	`TSM_CMD_DB_GET | msg`
	`index`	`1 << port`
	`seqid`	`seqid`
	`mac[2]`	`mac[2]`
Resp	`cur_sec`	time in second.
	`cur_nsec`	time in nanosecond.
	`timestamp`	same as `cur_nsec`.

The response bit of this command is `TSM_CMD_RESP`.

Depending on the index position the driver will return either the receive or transmit timestamp of the PTP message. Right now only the transmit timestamp is supported. The message can be Sync, Delay_Req, Pdelay_Req, and Pdelay_Resp.

The sequence id are echoed back for use by the application for bookkeeping.

6.3.4 tsm_cmd_get_time

```
int tsm_cmd_get_time ( u32 timestamp );
```

Parameters	`u32 timestamp`	32-bit timestamp value generated by hardware.
Return	`int`	A positive value to indicate success.
Description		This function asks device to return the time based on the timestamp.

Data		`struct tsm_cmd_get_time`
	`cmd`	`TSM_CMD_DB_GET_TIME`
	`nsec`	`timestamp`
Resp	`sec`	time in second.
	`nsec`	time in nanosecond.

The response bit of this command is `TSM_CMD_GET_TIME_RESP`.

The format of the timestamp is `(((sec & 3) << 30) | nanosec)`. Driver will return the actual second close to the timestamp. Obviously if this function is called later than 3 seconds the result will not be accurate.

Application can keep track of the system second so that calling this function is not necessary.

Driver reads the hardware clock when the timestamp is zero.

6.3.5 tsm_cmd_set_cfg

```
int tsm_cmd_set_cfg ( u8 port, u8 enable, u8 gmp, u32 ingress,
u16 egress );
```

Parameters	`u8 port`	Port number.
	`u8 enable`	Hardware configurations to set.
	`u8 gmp`	Grandmaster port number. Not used.
	`u32 ingress`	Ingress delay value.
	`u32 egress`	Egress delay value. Not used.
Return	`int`	A positive value to indicate success.
Description		This function sets some hardware configurations.

Data		`struct tsm_cfg`
	`cmd`	`TSM_CMD_CNF_SET`
	`port`	`port`
	`enable`	`enable`
	`gmp`	`gmp`
	`ingress_delay`	`ingress`
	`egress_delay`	`egress`

OnTime application uses this function to set some hardware configurations like P2P and peer delays. When the `port` parameter is 0xFF, meaning all ports, setting bit 2 (0x4) of the `enable` parameter means the clock is running in P2P mode. When the port is set to value other than zero, it means the configuration only applies to that port, number 1 for port 1 and number 2 for port 2. Generally this is used to the specify the peer delay of the port under P2P. The value is supplied in the `ingress` parameter. The `gmp` parameter can be set to tell the hardware which port is the grandmaster port, i.e., which port the Announce and Sync messages come from. For now hardware does not do anything special with this information.

The peer delay of each port is used by hardware to compensate the correct field correctly when running in 1-step P2P mode.

6.3.6 tsm_cmd_set_clock

```
int tsm_cmd_set_clock ( u32 sec, u32 nsec, u32 timestamp );
```

Parameters	`u32 sec`	Clock's second value.
	`u32 nsec`	Clock's nanosecond value.
	`u32 timestamp`	Clock's eventual nanosecond value.
Return	`int`	A positive value to indicate success.
Description		This function sets the clock as close as possible to the master clock.

Data		`struct tsm_clock_set`
	`cmd`	`TSM_CMD_CLOCK_SET`
	`timestamp`	`timestamp`
	`nsec`	`nsec`
	`sec`	`sec`

This function tries to set the clock as close as possible to the master clock after receiving the Sync message from the master. When the Sync message is received the incoming nanosecond timestamp is generated. With this information application knows the difference between the master clock and local clock. The difference is added to the nanosecond value to generate the timestamp value.

However due to hardware limitation the resulting time may not match the intent of this function. It will require a few more clock adjustments to bring the local clock close to the master clock.

6.3.7 tsm_cmd_correct_clock

```
int tsm_cmd_correct_clock ( int drift, int nsec );
```

Parameters	`int drift`	The drift value to adjust the clock continually.
	`int nsec`	Zero to indicate no nanosecond adjustment.
Return	`int`	A positive value to indicate success.
Description		This function adjusts the clock continually for synchronization.

Data		`struct tsm_clock_correct`

	`cmd`	`TSM_CMD_CLOCK_CORRECT`
	`add`	Used to indicate adding or subtracting clock adjustment.
	`sec`	Not used.
	`nsec`	Used to indicate nanosecond adjustment.
	`drift`	The drift value related to master clock.
	`offset`	Used to indicate adding or subtracting nanosecond adjustment.

The drift value is calculated as the nanosecond difference related to the master clock under 1 second. If the slave clock is found to be running 100 ns slower than the master clock in 1 second interval, then the clock adjustment should be increased by 100 ns.

The nanosecond adjustment is used to quickly close the nanosecond difference gap is it is big enough. Generally it is preferable to adjust the drift value to close the gap slowly.

OnTime application specifies the add value to be `(2 << 4)` when calling this function the very first time after reset. The driver ignores the command.

6.3.8 tsm_cmd_get_gps

```
int tsm_cmd_get_gpos ( void );
```

Parameters		None.
Return	`int`	A positive value to indicate success.
Description		This function requests the GPS timestamp to be sent.

Data		`struct tsm_get_gps`
	`cmd`	`TSM_CMD_GET_GPS_TS`
Resp	`seqid`	The sequence id.
	`sec`	Timestamp in second.
	`nsec`	Timestamp in nanosecond.

This function is used to request GPS timestamp to be sent to the application. The PPS of the GPS is supposedly connected to one the GPIO pins. OnTime application uses the GPS information to adjust the master clock.

6.4 Ioctl

The following functions use ioctl to send commands. Immediate responses are stored in the ioctl parameters, while delayed responses will be sent using file I/O.

6.4.1 Hardware Settings

These functions affect hardware operation related to PTP messages.

6.4.1.1 get_global_cfg

```
int get_global_cfg ( int *master, int *two_step, int *p2p, int
*as, int *csum, int *unicast, int *alternate, int *check, int
*delay_assoc, int *pdelay_assoc, int *sync_assoc, int
*drop_sync, int *priority, u8 *domain, u32 *delay );
```

Parameters	`int *master`	Buffer to store the master clock setting.
	`int *two_step`	Buffer to store the 2-step clock setting.
	`int *p2p`	Buffer to store the P2P clock setting.
	`int *as`	Buffer to store the 802.1AS setting.
	`int *csum`	Buffer to store the UDP checksum setting.
	`int *unicast`	Buffer to store the unicast setting.
	`int *alternate`	Buffer to store the alternate master setting.
	`int *check`	Buffer to store the domain check setting.
	`int *delay_assoc`	Buffer to store the Delay_Req association setting.
	`int *pdelay_assoc`	Buffer to store the Pdelay_Req association setting.
	`int *sync_assoc`	Buffer to store the Sync association setting.
	`int *drop_sync`	Buffer to store the drop Sync setting.
	`int *priority`	Buffer to store the message priority setting.
	`u8 *domain`	Buffer to store the domain number.
	`u32 *delay`	Buffer to store the access delay value.
Return	`int`	0 if successful.
Description		This function retrieves the hardware settings in use.

Ioctl	size	sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)
	cmd	DEV_CMD_GET
	subcmd	DEV_PTP_CFG
Param		struct ptp_cfg_options
	master	*master
	two_step	*two_step
	p2p	*p2p
	as	*as
	udp_csum	*csum
	unicast	*unicast
	alternate	*alternate
	domain_check	*check
	delay_assoc	*delay_assoc
	pdelay_assoc	*pdelay_assoc
	sync_assoc	*sync_assoc
	drop_sync	*drop_sync
	priority	*priority
	domain	*domain
	access_delay	*delay

This function is generally called during application initialization to get the current hardware settings in use. The most important value is the hardware access delay, as the hardware can run in generic bus or SPI, where the register access is very slow. In that case the clock correct command cannot be called too frequently as the result will not be correct. The application needs to adjust correspondingly as generally the clock correction is done when a Sync message is received and the Sync frequency can be very high.

The hardware access delay in generic bus is about 5 microseconds. The hardware can handle up to 100 Sync per second. Hardware access delay in SPI bus is about 15 milliseconds. The hardware can handle up to 16 Sync per second.

The other values are for information only as generally the application will override them as it setup the hardware.

6.4.1.2 set_hw_domain

```
int set_hw_domain ( u8 domain )
```

Parameters	`u8 domain`	The domain number to set.
Return	`int`	0 if successful.
Description		This function sets the domain number for domain checking.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`domain_set`	`1`
	`domain`	`domain`

6.4.1.3 set_hw_master

```
int set_hw_master ( int master );
```

Parameters	`int master`	Zero to run as a slave; others to run as a master.
Return	`int`	0 if successful.
Description		This function configures the clock to run as a master or a slave.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`master_set`	`1`
	`master`	`master`

6.4.1.4 set_hw_2_step

```
int set_hw_2_step ( int two_step );
```

Parameters	`int two_step`	Zero to run in 1-step mode; others to run in 2-step mode.
Return	`int`	0 if successful.
Description		This function configures the clock to run in 2-step or 1-step clock mode.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`two_step_set`	`1`
	`two_step`	`two_step`

6.4.1.5 set_hw_p2p

```
int set_hw_p2p ( int p2p );
```

Parameters	`int p2p`	Zero to run in E2E mode; others to run in P2P mode.
Return	`int`	0 if successful.
Description		This function configures the clock to run in P2P or E2E mode.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`p2p_set`	`1`
	`p2p`	`p2p`

6.4.1.6 set_hw_as

```
int set_hw_as ( int as );
```

Parameters	`int as`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware 802.1AS feature, which acts as a debug tool to forward all PTP messages to host CPU.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`as_set`	`1`
	`as`	`as`

6.4.1.7 set_hw_csum

```
int set_hw_csum ( int csum );
```

Parameters	`int csum`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware IPv4 UDP checksum feature. It has no effect on IPv6 PTP messages.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`udp_csum_set`	`1`
	`udp_csum`	`csum`

6.4.1.8 set_hw_unicast

```
int set_hw_unicast ( int unicast );
```

Parameters	`int unicast`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware receive unicast feature.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`unicast_set`	`1`
	`unicast`	`unicast`

6.4.1.9 set_hw_domain_check

```
int set_hw_domain_check ( int check );
```

Parameters	`int check`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware domain check feature.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`domain_check_set`	`1`
	`domain_check`	`check`

6.4.1.10 set_hw_alternate

```
int set_hw_alternate ( int alternate );
```

Parameters	`int alternate`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware alternate master feature, which allows Sync forwarding when running as a master.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`alternate_set`	`1`
	`alternate`	`alternate`

6.4.1.11 set_hw_delay_assoc

```
int set_hw_delay_assoc ( int assoc );
```

Parameters	`int assoc`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware Delay_Req association feature.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`delay_assoc_set`	`1`
	`delay_assoc`	`assoc`

6.4.1.12 set_hw_pdelay_assoc

```
int set_hw_pdelay_assoc ( int assoc );
```

Parameters	`int assoc`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware Pdelay_Req association feature.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`pdelay_assoc_set`	`1`
	`pdelay_assoc`	`assoc`

6.4.1.13 set_hw_sync_assoc

```
int set_hw_sync_assoc ( int assoc );
```

Parameters	`int assoc`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware Sync association feature.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`sync_assoc_set`	`1`
	`sync_assoc`	`assoc`

6.4.1.14 set_hw_drop_sync

```
int set_hw_drop_sync ( int drop );
```

Parameters	`int drop`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware drop Sync feature.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`drop_sync_set`	`1`
	`drop_sync`	`drop`

6.4.1.15 set_hw_priority

```
int set_hw_priority ( int priority );
```

Parameters	`int priority`	Zero to disable; others to enable.
Return	`int`	0 if successful.
Description		This function disables/enables the hardware all PTP message high priority feature.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`priority_set`	`1`
	`priority`	`priority`

6.4.1.16 set_global_cfg

```
int set_global_cfg ( int master, int two_step, int p2p, int as );
```

Parameters	`int master`	Zero to run as a slave; others to run as a master.
	`int two_step`	Zero to run in 1-step mode; others to run in 2-step mode.
	`int p2p`	Zero to run in E2E mode; others to run in P2P mode.
	`int as`	Non-zero to run in 802.1AS mode.
Return	`int`	0 if successful.
Description		This function configures the hardware settings frequently used.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_cfg_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CFG`
Param		`struct ptp_cfg_options`
	`master_set`	`1`
	`master`	`master`
	`two_step_set`	`1`
	`two_step`	`two_step`
	`p2p_set`	`1`
	`p2p`	`p2p`
	`as_set`	`1`
	`as`	`as`

The `set_hw_*` functions are used to turn on/off certain hardware features. Most of them are related to message filtering and called once to set a hardware setting that will stay the same through the life of the application. The `set_global_cfg` function combines the functions `set_hw_master`, `set_hw_2_step`, `set_hw_p2p`, and `set_hw_as` as they are called frequently inside the application.

6.4.2 Trigger Output and Event Detection

The PTP hardware can generate trigger outputs and capture incoming events using general-purpose I/O pins. Up to 12 GPIO pins can be used for trigger output and event detection depending on chip configurations. However, most of the time some of the GPIO pins are not available and at least one is reserved for hardware interrupt. There are 12 output units and 12 events units available for use in event generation and detection. They are labeled from 0 to 11. The last units (11) in output and event units are special and the last output unit is reserved for use by the system. One of the event units, 10, is reserved for use in the driver.

Input Event	Value	Input Signal
`DETECT_FALL`	0	Detect falling edge
`DETECT_RISE`	1	Detect rising edge
`DETECT_BOTH`	2	Detect both falling and rising edges.

The following trigger output events are supported:

Output Event	Value	Output Signal
`TRIG_NEG_EDGE`	0	Negative edge—a falling edge from high to low
`TRIG_POS_EDGE`	1	Positive edge—a rising edge from low to high
`TRIG_NEG_PULSE`	2	Negative pulse—falling edge then rising edge after pulse time
`TRIG_POS_PULSE`	3	Positive pulse—rising edge then falling edge after pulse time
`TRIG_NEG_CYCLE`	4	Negative cycle—falling edge then rising edge after pulse time and stay high through cycle time
`TRIG_POS_CYCLE`	5	Positive cycle—rising edge then falling edge after pulse time and stay low through cycle time
`TRIG_REG_OUTPUT`	6	Register bit pattern output—0 to indicate low and 1 to indicate high

For `TRIG_NEG_CYCLE` and `TRIG_POS_CYCLE` outputs that require cycle time the output can be repeated. For the others the output is one-shot. To repeat those outputs or generate more complex signals cascade mode can be used. In cascade mode several output units are grouped to perform an output signal in sequence. The cascaded output itself can be repeated.

The output units in cascade are separated by an iteratation time. When these iteratation times in output units are the same doing the cascaded output repeatedly has no concern. As this iteratation time is configured separately for each output unit and can be different, care must be exercised to not repeat the cascade output too much lest the individual outputs overlap each other.

For event detection the hardware can only detect falling and rising edges. Each event unit can only store 2 events, except the last one, which can store 8 events. To detect an event signal that has more than 2 events cascade mode has to be used. The API automatically takes care of this and so the users do not need to concern whether cascade mode is being used or not.

Command Flags	**Values**	**Meaning**
`PTP_CMD_INTR_OPER`	`0x01`	Enable interrupt.
`PTP_CMD_SILENT_OPER`	`0x02`	Silent output.
`PTP_CMD_ON_TIME`	`0x04`	Operate exactly on trigger time.
`PTP_CMD_REL_TIME`	`0x08`	Trigger time is relative.
`PTP_CMD_CLK_OPT`	`0x10`	Enable clock option.
`PTP_CMD_CASCADE_RESET_OPER`	`0x40`	Used for testing only.
`PTP_CMD_CANCEL_OPER`	`0x80`	Cancel operation.

The flags above are used in trigger output and timestamp event operations to influence the operation results.

6.4.2.1 rx_event

```
int rx_event ( u8 tsi, u8 gpi, u8 event, u8 total, u8 flags, u32
timeout, int *unit );
```

Parameters	`u8 tsi`	Event unit to use.
	`u8 gpi`	General purpose input pin to use.
	`u8 event`	0 for falling edge and other for rising edge.
	`u8 total`	Number of event units to use.
	`u8 flags`	Flags to modify operation.
	`u32 timeout`	Timeout value in millisecond.
	`int *unit`	Event unit assigned.
Return	`int`	0 if successful.
Description		This function starts the event detection.

Ioctl	`size`	`sizeof(struct ksz_request) +`

		sizeof(struct ptp_tsi_options)
	cmd	DEV_CMD_PUT
	subcmd	DEV_PTP_TEVT
Resp	output	The event unit assigned if successful.
Param		**struct ptp_tsi_options**
	tsi	tsi
	gpi	gpi
	event	event
	flags	flags
	total	total
	timeout	timeout

This function starts an event unit to detect either a falling or rising edge signal from a GPIO input pin.

The event unit can be specified directly. If a value equal to or greater than `MAX_TIMESTAMP_UNIT` is used the driver will choose an available one and returns that in the `unit` variable. If there is no unit available for use the function will fail.

The GPIO input pin should be less than the `MAX_GPIO` value.

Event can only be 0 for falling edge detection, 1 for rising edge, and other for both edge detection.

The total units can be specified if more event captures are needed. All event units are assigned consecutively starting from the first one. If there are not enough available units the function will fail.

Note the units can loop to the beginning, so it is not necessary to make sure the last unit is greater than the first unit.

Setting the `PTP_CMD_INTR_OPER` bit in the flags indicates interrupt should be enabled for event units. When event is detected the driver immediately notifies the application. Setting the `PTP_CMD_CANCEL_OPER` bit stops the unit and allows the unit to be used in other way. This is necessary if the event does not happen at all.

Setting the `PTP_CMD_SILENT_OPER` bit in the flags causes the driver not to notify applications about the event. This is used internally by the driver.

A timeout value other than zero indicates the driver can stop and free the unit after that timeout passes since the first event happens. Application does not need to call this function specifically to free the unit. A zero timeout value indicates the event is repeatable. The driver automatically restarts the unit after the event.

A timeout value is normally needed as the hardware only notifies the driver via interrupt when the first event happens, but not the second. The driver needs to poll the hardware to detect the second event. As applications likely care only the first event, this provides a means to quickly reuse the event unit.

The timeout is checked using system ticks, which normally is updated every 10 milliseconds. Therefore, the minimum timeout is 10 milliseconds.

The unit returned should be the same unit specified if it is less than the `MAX_TIMESTAMP_UNIT` value.

In addition to normal error codes returned by ioctl calls, the error can be `DEV_IOC_UNIT_USED` for unit in used and `DEV_IOC_UNIT_UNAVAILABLE` for unit not available.

6.4.2.2 tx_event

```
int tx_event ( u8 tso, u8 gpo, u8 event, u32 pulse, u32 cycle
u16 cnt, u32 iterate, u32 sec, u32 nsec, u8 flags, int *unit );
```

Parameters	`u8 tso`	Output unit to use.
	`u8 gpo`	General-purpose output pin to use.
	`u8 event`	Trigger output event to use.
	`u32 pulse`	Pulse time in 8 nanoseconds unit.
	`u32 cycle`	Cycle time in nanosecond.
	`u16 cnt`	Repeat count.
	`u32 iterate`	Iteration time in nanosecond. Used in cascade mode.
	`u32 sec`	Starting time in second.
	`u32 nsec`	Starting time in nanosecond.
	`u8 flags`	Operation flags.
	`int *unit`	Output unit assigned.
Return	`int`	0 if successful.
Description		This function starts the trigger output in normal mode. It is used to setup individual output unit in cascade mode.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_tso_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_TOUT`

Resp	output	The starting output unit assigned if successful.
Param		**struct ptp_tso_options**
	tso	tso
	gpo	gpo
	event	event
	flags	flags
	pulse	pulse
	cycle	cycle
	cnt	cnt
	sec	sec
	nsec	nsec
	iterate	iterate

This function starts an output unit to generate a signal in normal mode. It is also used to setup an output unit in cascade mode.

The output unit can be specified directly. If a value equal to or greater than MAX_TRIG_UNIT is used the driver will choose the first available one and returns that in the unit variable. If there is no available unit the function will fail.

The GPIO output pin should be less than the MAX_GPIO value.

The output event is one of the codes displayed in the Output Event table.

Setting the PTP_CMD_INTR_OPER bit in the flags indicates interrupt should be enabled for the output unit so that after the operation is completed the unit can immediately be stopped and freed for other use. Setting the PTP_CMD_CANCEL_OPER bit stops the output unit and allows it to be used again.

Setting the PTP_CMD_SILENT_OPER bit in the flags causes the driver not to notify applications after the command is completed. It is used when the application does not care about that information.

The pulse time is required for pulse signals. It is in unit of 8 ns and the value is 16-bit, so the minimum pulse time is 8 ns, and the maximum is 524280 ns.

The last unit can use a 24-bit value as pulse time, so it is reserved to generate PPS as 20 ms pulse time is required.

The cycle time is required for cycle signals.

The repeat count can be more than 1 for repeated signals. A count of zero means infinite.

The second and nanosecond are used to specify when the output starts. They are absolute time. For ease of testing a PTP_CMD_REL_TIME bit can be set in the flags to make them relative from the current time, i.e., they act like a delay value.

Setting the `PTP_CMD_ON_TIME` bit in the flags means the operation has to start right on the target time. If that time is past the operation returns in error. If that bit is not set the operation will always run to completion.

The unit returned should be the same output unit specified if it is less than the `MAX_TRIG_UNIT` value.

In addition to normal error codes returned by ioctl calls, the error can be `DEV_IOC_UNIT_USED` for unit in use or `DEV_IOC_UNIT_UNAVAILABLE` for unit not available. An error of `DEV_IOC_ERROR` will be returned if `PTP_CMD_ON_TIME` bit is set and the target time is past.

In cascade mode the units allowed to be operated are the ones pre-allocated in the `tx_cascade_init` function.

6.4.2.3 tx_cascade_init

```
int tx_cascade_init ( u8 tso, u8 gpo, u8 total, u8 flags, int
*unit );
```

Parameters	`u8 tso`	Starting output unit requested for use in cascade mode.
	`u8 gpo`	GPIO output pin operated on.
	`u8 total`	Number of output units requested for use.
	`u8 flags`	Operation flags.
	`int *unit`	Starting output unit assigned.
Return	`int`	0 if successful.
Description		This function reserves output units for use in cascade mode.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_tso_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CASCADE`
	`output`	0
Resp	`output`	The starting output unit assigned if successful.
Param		**`struct ptp_tso_options`**
	`tso`	`tso`

	gpo	gpo
	total	total
	flags	flags

The function pre-allocates output units for use in cascade mode.

The first starting output unit can be specified. If a value equal to or greater than `MAX_TRIG_UNIT` is used the driver will choose the first available unit and returns that in the `unit` variable.

The total units is specified to indicate how many output units are needed. Units are assigned consecutively and if there are not enough available units the function will fail.

Setting the `PTP_CMD_CANCEL_OPER` bit in the flags frees the assigned units and allows them to be used again.

6.4.2.4 tx_cascade

```
int tx_cascade ( u8 tso, u8 gpo, u8 total, u16 cnt, u8 flags );
```

Parameters	u8 tso	Starting output unit.
	u8 gpo	GPIO output pin operated on.
	u8 total	Number of output units.
	u16 cnt	Repeat count.
	u8 flags	Operation flags.
Return	int	0 if successful.
Description		This function starts the trigger output in cascade mode.

Ioctl	size	sizeof(struct ksz_request) + sizeof(struct ptp_tso_options)
	cmd	DEV_CMD_PUT
	subcmd	DEV_PTP_CASCADE
	output	1
Param		**struct ptp_tso_options**
	tso	tso
	gpo	gpo
	total	total
	flags	flags

	cnt	cnt

This function starts the cascade outputs.

The staring output unit and total units should match the ones used in `tx_cascade_init` function.

The repeat count can be more than 1 for repeated signals.

The `PTP_CMD_INTR_OPER` bit in the flags can be set to indicate interrupt should be enabled for the operation. When the operation is completed driver can stop and free the output units for other use. The `PTP_CMD_CANCEL_OPER` bit can be set to manually stop the operation.

For ease of testing the `PTP_CMD_REL_TIME` bit can be set in the flags to indicate all the times specified in `tx_event` setup are relative to the time when this function is executed.

6.4.2.5 get_rx_event

```
int get_rx_event ( u8 tsi );
```

Parameters	u8 tsi	Event unit to retrieve the events.
Return	int	0 if successful.
Description		This function retrieves events from the event unit and puts them in the data buffer.

Ioctl	size	sizeof(struct ksz_request) + sizeof(struct ptp_tsi_info)
	cmd	DEV_CMD_GET
	subcmd	DEV_PTP_TEVT
	output	0
Param		**struct ptp_tsi_info**
	cmd	PTP_CMD_GET_EVENT
	unit	tsi

This function requests the driver to return the events from the specified unit in `struct ptp_tsi_info` using file I/O. The events returned are those stored in the driver, so calling this function several times will yield the same information. The returned code can be `DEV_IOC_UNIT_UNAVAILABLE` when there are no events.

6.4.2.6 poll_rx_event

```
int poll_rx_event ( u8 tsi );
```

Parameters	`u8 tsi`	Event unit to poll.
Return	`int`	0 if successful.
Description		This functions polls the event unit to see if there are new events.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_tsi_info)`
	`cmd`	`DEV_CMD_GET`
	`subcmd`	`DEV_PTP_TEVT`
	`output`	`1`
Param		**`struct ptp_tsi_info`**
	`cmd`	`PTP_CMD_GET_EVENT`
	`unit`	`tsi`

This function polls the hardware to read any new events from the event unit. The returned information is the same as from `ptp_get_event`. Calling this function when the unit is disabled will always give `DEV_IOC_UNIT_UNAVALABLE` error.

This function is necessary if the application wants to read the second event in the event unit in a timely fashion as the driver only updates that information when there is an interrupt and after a detection timeout. That period can be very short or long depending on PTP operation. The maximum time is 1 second for the PPS generated by the hardware and detected by the driver.

6.4.3 Hardware Clock Controls

These functions manipulate the hardware clock.

6.4.3.1 get_clock

```
int get_clock ( u32 *sec, u32 *nsec );
```

Parameters	`u32 *sec`	Buffer to store the second.

	`u32 *nsec`	Buffer to store the nanosecond.
Return	`int`	0 if successful.
Description		This function gets the current clock and store the second and nanosecond in the provided buffers.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_clk_options)`
	`cmd`	`DEV_CMD_GET`
	`subcmd`	`DEV_PTP_CLK`
	`output`	`0`
Param		`struct ptp_clk_options`
	`sec`	`*sec`
	`nsec`	`*nsec`

This function returns the current hardware clock.

6.4.3.2 set_clock

```
int set_clock ( u32 sec, u32 nsec );
```

Parameters	`u32 sec`	The second to set.
	`u32 nsec`	The nanosecond to set.
Return	`int`	0 if successful.
Description		This function sets the clock with supplied second and nanosecond.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_clk_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CLK`
	`output`	`0`
Param		`struct ptp_clk_options`
	`sec`	`sec`
	`nsec`	`nsec`

This function sets the hardware clock. It is used mostly in testing as the clock can not be set accurately using this way.

6.4.3.3 adj_clock

```
int adj_clock ( int add, u32 sec, u32 nsec, int drift, u32
interval );
```

Parameters	`int add`	Zero to subtract; others to add.
	`u32 nsec`	The number of seconds to change.
	`u32 nsec`	The number of nanoseconds to change.
	`int drift`	The continual clock adjustment drift value.
	`u32 interval`	The interval for continual clock adjustment.
Return	`int`	0 if successful.
Description		This functions adjusts the clock by adding or subtracting nanoseconds and/or setting the clock drift value.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_clk_options)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_CLK`
	`output`	`1` to subtract; `2` to add
Param		`struct ptp_clk_options`
	`sec`	`sec`
	`nsec`	`nsec`
	`drift`	`drift`
	`interval`	`interval`

This function adjusts the hardware clock by nanoseconds. Used this way the clock can be set more accurately. Note the time should be less than 1 second. Next revision of chip allows adjustment by second. For now driver also simulates second adjustment by adjusting the nanoseconds several times if the second is small enough. Otherwise, a set clock command is used instead.

This function also sets the continual clock adjustment value by calculating the drift value over the interval. The drift value is the offset from master over the interval. Normally this value is normalized over an interval of 1 second.

To set the continual clock adjustment value but not adjusting the clock, set the `sec` and nanosecond values to zero. To adjust the clock but not setting the continual clock adjustment value, set the interval value to zero.

6.4.4 Hardware Configurations

6.4.4.1 get_delay

```
int get_delay ( int port, u32 *rx, u32 *tx, int *asym );
```

Parameters	`int port`	`0` for port 1 or `1` for port 2.
	`u32 *rx`	Buffer to store the receive delay in nanoseconds.
	`u32 *tx`	Buffer to store the transmit delay in nanoseconds.
	`int *asym`	Buffer to store the asymmetric delay in nanoseconds.
Return	`int`	0 if successful.
Description		This function gets the port receive, transmit, and asymmetric delays in hardware.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_delay_values)`
	`cmd`	`DEV_CMD_GET`
	`subcmd`	`DEV_PTP_DELAY`
	`output`	`port`
Param		`struct ptp_delay_values`
	`rx_latency`	`*rx`
	`tx_latency`	`*tx`
	`asym_delay`	`*asym`

This function returns the receive latency, transmit latency, and asymmetric delay values stored in hardware. They are for information only as applications normally will override them.

6.4.4.2 set_delay

```
int set_delay ( int port, u32 rx, u32 tx, int asym );
```

Parameters	`int port`	`0` for port 1 or `1` for port 2.
	`u32 rx`	Receive delay in nanoseconds.
	`u32 tx`	Transmit delay in nanoseconds.
	`int asym`	Asymmetric delay in nanoseconds.
Return	`int`	0 if successful.
Description		This function sets the port receive, transmit, and asymmetric delays in hardware.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_delay_values)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_DELAY`
	`output`	`port`
Param		`struct ptp_delay_values`
	`rx_latency`	`rx`
	`tx_latency`	`tx`
	`asym_delay`	`asym`

This function is used to set the receive latency, transmit latency, and asymmetric delay values in the hardware. Applications normally retrieve those values from configuration. The values are used by hardware to adjust the correction field of PTP event messages when they are forwarded to the other port.

6.4.4.3 get_clock_ident

```
int get_clock_ident ( struct ptp_clock_identity *id );
```

Parameters	`struct ptp_clock_identity *id`	Buffer to store the clock identity.
Return		0 if successful.
Description		This function gets the clock id of the device. It is primarily used to retrieve the MAC address of the device.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_clock_identity)`

	cmd	DEV_CMD_GET
	subcmd	DEV_PTP_CLOCK_IDENT
Param		struct ptp_clock_identity

This function is used for debug purpose.

6.4.4.4 set_clock_ident

```
int set_clock_ident ( struct ptp_clock_identity *id );
```

Parameters	struct ptp_clock_identity *id	Clock identity.
Return		0 if successful.
Description		This function sets the clock id of the device. It is primarily used in debugging.

Ioctl	size	sizeof(struct ksz_request) + sizeof(struct ptp_clock_identity)
	cmd	DEV_CMD_PUT
	subcmd	DEV_PTP_CLOCK_IDENT
Param		struct ptp_clock_identity

This function is used for debug purpose.

6.4.4.5 get_reg

```
int get_reg ( int reg, int *val );
```

Parameters	int reg	Register.
	int *val	Buffer to store the value.
Return		0 if successful.
Description		This function reads register to get its value.

Ioctl	size	sizeof(struct ksz_request)

	`cmd`	`DEV_CMD_GET`
	`subcmd`	`DEV_PTP_REG`
	`output`	`reg`
Resp	`output`	`*val`

This function allows reading the hardware register. It is used for debug purpose.

6.4.4.6 set_reg

```
int set_reg ( int reg, int val );
```

Parameters	`int reg`	Register.
	`u16 val`	Value.
Return		0 if successful.
Description		This function programs value to the register.

Ioctl	`size`	`sizeof(struct ksz_request)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_REG`
	`output`	`reg | (val << 16)`

This function allows writing to hardware register. It is used for debug purpose only as application should use the supplied APIs to program any necessary hardware registers.

6.4.4.7 get_peer_delay

```
int get_peer_delay ( int port, u32 *delay );
```

Parameters	`int port`	`0` for port 1 or `1` for port 2.
	`u32 *delay`	Buffer to store the peer delay in nanoseconds.
Return	`int`	0 if successful.
Description		This function gets the port peer delay in hardware.

Ioctl	`size`	`sizeof(struct ksz_request) + sizeof(struct ptp_delay_values)`

	cmd	DEV_CMD_GET
	subcmd	DEV_PTP_PEER_DELAY
	output	port
Param		struct ptp_delay_values
	reserved	*delay

This function returns the peer delay values stored in hardware. They are for information only as applications normally will override them.

6.4.4.8 set_peer_delay

```
int set_peer_delay ( int port, u32 delay );
```

Parameters	int port	0 for port 1 or 1 for port 2.
	u32 delay	Peer delay in nanoseconds.
Return	int	0 if successful.
Description		This function sets the port peer delay in hardware.

Ioctl	size	sizeof(struct ksz_request) + sizeof(struct ptp_delay_values)
	cmd	DEV_CMD_PUT
	subcmd	DEV_PTP_PEER_DELAY
	output	port
Param		struct ptp_delay_values
	reserved	delay

This function is used to set the peer delay value of each port in the hardware. Applications calculate peer delay using P2P mechanism. Hardware uses peer delay value to adjust the correction field of Sync messages when they are forwarded to the other port.

6.4.4.9 get_utc_offset

```
int get_utc_offset ( int *offset );
```

Parameters	int *offset	Buffer to store the UTC offset.

Return		0 if successful.
Description		This function gets the UTC offset stored in the driver.

Ioctl	`size`	`sizeof(struct ksz_request)`
	`cmd`	`DEV_CMD_GET`
	`subcmd`	`DEV_PTP_UTC_OFFSET`

This function retrieve the UTC offset from the driver. The UTC offset, or leap second, is used to offset the system time in UTC from the PTP hardware time in TAI.

6.4.4.10 set_utc_offset

```
int set_utc_offset ( int offset );
```

Parameters	`int offset`	UTC offset.
Return		0 if successful.
Description		This function sets the UTC offset.

Ioctl	`size`	`sizeof(struct ksz_request)`
	`cmd`	`DEV_CMD_PUT`
	`subcmd`	`DEV_PTP_UTC_OFFSET`

This function sets the UTC offset to the driver. The UTC offset is reported from the grandmaster or NTP server.

6.4.4.11 get_rx_timestamp

```
int get_rx_timestamp ( u32 timestamp, u32 *sec, u32 *nsec );
```

Parameters	`u32 timestamp`	Timestamp supplied by the hardware.
	`u32 *sec`	Buffer to store the second value.
	`u32 *nsec`	Buffer to store the nanosecond value.
Return		0 if successful.
Description		This function retrieves the receive time with the receive timestamp.

Ioctl	size	sizeof(struct ksz_request) + sizeof(struct ptp_ts_options)
	cmd	DEV_CMD_GET
	subcmd	DEV_PTP_TIMESTAMP
Param		struct ptp_ts_options
	timestamp	timestamp
Resp	sec	*sec
	nsec	*nsec

The function retrives the receive time of a PTP event message. The receive timestamp supplied should be the one reported from hardware. This is stored in the 32-bit reserved field of the PTP message header. It should be swapped from big-endian to little-endian.

As explained before about the hardware timestamp, the second contained in the timestamp is limited to 3 seconds, so the call to retrieve the receive time should be made within 3 seconds after the event message is received; otherwise, the second information will not be correct.

6.4.4.12 get_tx_timestamp

```
int get_tx_timestamp ( u8 msg, u8 port, u16 seqid, u8 mac[2],
u32 *sec, u32 *nsec );
```

Parameters	u8 msg	One of the PTP event messages.
	u8 port	0 for port 1 or 1 for port 2.
	u16 seqid	The sequence id of the PTP event message.
	u8 mac[2]	The last 2 bytes of the source clock identity.
	u32 *sec	Buffer to store the second value.
	u32 *nsec	Buffer to store the nanosecond value.
Return		0 if successful. Error code -EAGAIN if not available.
Description		This function retrieves the transmit timestamp of the PTP event message with supplied information.

Ioctl	size	sizeof(struct ksz_request) + sizeof(struct ptp_ts_options)
	cmd	DEV_CMD_GET
	subcmd	DEV_PTP_TIMESTAMP

Param		`struct ptp_ts_options`
	`msg`	`msg`
	`port`	`port`
	`seqid`	`seqid`
	`mac[2]`	`mac[2]`
Resp	`sec`	`*sec`
	`nsec`	`*nsec`

This function retrieves the transmit timestamp of a PTP event message, which is Sync, Delay_Req, Pdelay_Req, or Pdelay_Resp. If the timestamp is available, all the driver needs is the message code and the port index, which is 0 for port 1 and 1 for port 2. If the timestamp is not available and the PTP device is used to send this call, driver uses the sequence id and MAC address information to later send a **`tsm_cmd_get`** response to the application. The application has the option of keep sending this call or waiting for that response. With the echoed sequence id and clock identity information application can easily find and match the sent event message. See **6.3.3 tsm_cmd_get** for more information.

7 PTP Message Filtering

The hardware is designed to filter PTP messages to help software handle less messages as those dropped messages are not used by software at all. The following tables display how the hardware forwards or drops messages depending on different hardware settings. They assume the message is coming from port 1 and forwarded to host port 3 and the other port 2.

Legends used in the tables:

`+`	Forwarded as should be
`-`	Not forwarded and dropped as should be
`P`	Forwarded
`D`	Not forwarded and dropped
`M`	Domain number matched and forwarded; otherwise dropped
`U`	Domain number not matched and forwarded; otherwise dropped
`A`	Delay_Resp message matched Delay_Req and forwarded; otherwise dropped
`B`	Delay_Resp message not matched Delay_Req and forwarded; otherwise dropped
`C`	Pdelay_Resp/Pdelay_Resp_Follow_Up messages matched Pdelay_Req and forwarded
`S`	Follow_Up message matched Sync and forwarded; otherwise dropped
`P`	Result different from previous table.

PTP Message	1-step								2-step							
	Master				Slave				Master				Slave			
	E2E		P2P		E2E		P2P		E2E		P2P		E2E		P2P	
	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2
Sync	−	D	−	D	+	+	+	+	D	−	D	−	+	−	+	−
Follow_Up	−	D	−	D	+	+	+	+	D	−	D	−	+	−	+	−
Delay_Req	+	D	−	−	−	+	−	−	+	−	−	−	+	−	−	−
Delay_Resp	−	D	−	−	+	+	−	−	D	−	−	−	+	−	−	−
Pdelay_Req	***P***	D	+	−	−	+	+	−	***+***	−	+	−	+	−	+	−
Pdelay_Resp	−	D	+	−	***P***	+	+	−	D	−	+	−	+	−	+	−
Pdelay_Resp_Follow_Up	−	D	+	−	***P***	+	+	−	D	−	+	−	+	−	+	−
Announce	+	+	+	+	+	+	+	+	+	***P***	+	***P***	+	***P***	+	***P***
Management	+	+	+	+	+	+	+	+	+	***P***	+	***P***	+	***P***	+	***P***
Signaling	+	+	+	+	+	+	+	+	+	***P***	+	***P***	+	***P***	+	***P***

Table 1: All Filtering Functions Disabled

In the default mode where all filtering functions are disabled it is assumed the PTP system has only one grandmaster and one domain. The PTP 1-step slave clock in E2E mode only needs to process Sync, Follow_Up, and Delay_Resp messages, so only those messages are forwarded to port 3. All PTP messages are forwarded to port 2. P2P mode is not compatible to E2E, so all Delay_Req and Delay_Resp messages are dropped. Pdelay_Req, Pdelay_Resp, and Pdelay_Resp_Follow_Up messages are forwarded to port 3 and will never be forwarded to port 2 as those messages are stopped in the received device.

PTP 2-step clock never forwards messages to port 2 as forwarding is done in software. Slave clock needs to receive all messages to determine which one to forward. As in previous case it is assumed there are no Delay_Req and Delay_Resp messages in a P2P system and so they are dropped.

The PTP 1-step master clock in E2E mode only needs to process Delay_Resp message, and so only that message is forwarded to port 3. It is assumed there is one master and so the master clock does not forward Sync, Follow_Up, Delay_Req, and Delay_Resp messages to port 2. Theoretically it should forward Pdelay_Req, Pdelay_Resp, and Pdelay_Resp_Follow_Up messages. But as it does not forward Sync and Follow_Up messages it does not really help any P2P device connected to port 2 in actual use.

PTP 2-step master clock only forwards Delay_Req and Pdelay_Req messages to port 3, so software cannot do any forwarding. The end result is the same as running in 1-step clock mode.

There are some errors in hardware implementation. In E2E mode master clock forwards

Pdelay_Req message to port 3, and slave clock forwards not Pdelay_Req message but Pdelay_Resp and Pdelay_Resp_Follow_Up messages. As those messages are ignored by software, it does not impact PTP operation.

The only mode for hardware to forward all messages to port 2 is 1-step E2E slave mode. When software is not running it is required to set this mode to not drop any message. It is the default hardware mode after reset.

The E2E slave mode is the mode to use in 2-step clock setting to receive all messages so software can determine which to forward.

PTP Message	**1-step**								**2-step**							
	Master				**Slave**				**Master**				**Slave**			
	E2E		**P2P**		**E2E**		**P2P**		**E2E**		**P2P**		**E2E**		**P2P**	
	P3	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**
Sync	–	D	–	D	+	+	+	+	D	–	D	–	+	–	+	–
Follow_Up	–	D	–	D	+	+	+	+	D	–	D	–	+	–	+	–
Delay_Req	+	***U***	–	–	–	+	–	–	+	–	–	–	+	–	–	–
Delay_Resp	–	D	–	–	+	+	–	–	D	–	–	–	+	–	–	–
Pdelay_Req	P	***U***	***M***	–	–	+	***M***	–	+	–	***M***	–	+	–	***M***	–
Pdelay_Resp	–	D	+	–	P	+	+	–	D	–	+	–	+	–	+	–
Pdelay_Resp_Follow_Up	–	D	+	–	P	+	+	–	D	–	+	–	+	–	+	–
Announce	+	+	+	+	+	+	+	+	+	P	+	P	+	P	+	P
Management	+	+	+	+	+	+	+	+	+	P	+	P	+	P	+	P
Signaling	+	+	+	+	+	+	+	+	+	P	+	P	+	P	+	P

*Table 2: **Domain Check** Disabled With Different Domain Number*

There is a filtering feature called Domain Check in which hardware drops the message if the domain number in the message does not match the domain number set in the hardware. It turns out the hardware also has some implementation issues.

When Domain Check is disabled hardware is supposed to ignore the domain number. It seems the hardware is still doing some checking on certain messages. In P2P mode the Pdelay_Req message is only forwarded when the domain number matches. In E2E mode 1-step master clock also passes Delay_Req and Pdelay_Req messages to port 2 when the domain number is not matched.

As these problems do not impact PTP operation, it is not required to fix them. It is documented here for regression testing.

This can be verified by changing the domain number with `set_hw_domain` function.

PTP Message	**1-step**								**2-step**							
	Master				**Slave**				**Master**				**Slave**			
	E2E		**P2P**		**E2E**		**P2P**		**E2E**		**P2P**		**E2E**		**P2P**	
	P3	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**
Sync	−	**+**	−	**+**	+	+	+	+	**+**	−	**+**	−	+	−	+	−
Follow_Up	−	**+**	−	**+**	+	+	+	+	**+**	−	**+**	−	+	−	+	−
Delay_Req	+	**+**	−	−	−	+	−	−	+	−	−	−	+	−	−	−
Delay_Resp	−	**+**	−	−	+	+	−	−	**+**	−	−	−	+	−	−	−
Pdelay_Req	P	D	+	−	−	+	+	−	+	−	+	−	+	−	+	−
Pdelay_Resp	−	D	+	−	P	+	+	−	D	−	+	−	+	−	+	−
Pdelay_Resp_Follow_Up	−	D	+	−	P	+	+	−	D	−	+	−	+	−	+	−
Announce	+	+	+	+	+	+	+	+	+	P	+	P	+	P	+	P
Management	+	+	+	+	+	+	+	+	+	P	+	P	+	P	+	P
Signaling	+	+	+	+	+	+	+	+	+	P	+	P	+	P	+	P

*Table 3: **Alternate Master** Enabled*

PTP standard has a feature called Alternate Master in which slave clocks can synchronize with more than one master in case the main master fails. For this to work master clock needs to pass Sync, Follow_Up, Delay_Req, and Delay_Resp messages to port 2. This can be done by enabling Alternate Master setting.

This setting is disabled/enabled by `set_hw_alternate` function.

PTP Message	1-step								2-step							
	Master				Slave				Master				Slave			
	E2E		P2P		E2E		P2P		E2E		P2P		E2E		P2P	
	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2
Sync	–	***U***	–	***U***	***M***	+	***M***	+	***U***	–	***U***	–	+	–	+	–
Follow_Up	–	***U***	–	***U***	***M***	+	***M***	+	***U***	–	***U***	–	+	–	+	–
Delay_Req	***M***	***U***	–	–	–	+	–	–	+	–	–	–	+	–	–	–
Delay_Resp	–	***U***	–	–	***M***	+	–	–	***U***	–	–	–	+	–	–	–
Pdelay_Req	***M***	***U***	***M***	–	–	+	***M***	–	+	–	***M***	–	+	–	***M***	–
Pdelay_Resp	–	***U***	***M***	–	P	+	***M***	–	***U***	–	***M***	–	+	–	***M***	–
Pdelay_Resp_Follow_Up	–	***U***	***M***	–	P	+	***M***	–	***U***	–	***M***	–	+	–	***M***	–
Announce	***P***	+	***P***	+	***P***	+	***P***	+	+	P	+	P	+	P	+	P
Management	***P***	+	***P***	+	***P***	+	***P***	+	+	P	+	P	+	P	+	P
Signaling	***P***	+	***P***	+	***P***	+	***P***	+	+	P	+	P	+	P	+	P

*Table 4: **Domain Check** Enabled*

To support multiple domains master clock needs to pass Sync and Follow_Up messages to port 2 in P2P mode, and all messages in E2E mode. This is accomplished by enabling Domain Check setting.

Messages with different domain number received by master clock are passed to port 2 in 1-step clock mode or forwarded to port 3 in 2-step clock mode so that software can forward them as necessary. Otherwise, they are dropped as in Table 1.

Only Sync and Follow_Up messages with same domain number are forwarded to port 3 in slave clock.

In E2E mode only Delay_Req message with same domain number is forwarded to port 3 in master clock, and Delay_Resp message with same domain number in slave clock.

In P2P mode only Pdelay_Req, Pdelay_Resp, and Pdelay_Resp_Follow_Up messages with same domain number are forwarded to port 3.

This is the setting software should use so that messages are properly forwarded. As a result, it will be enabled in the driver.

It is noted that this does not mean Micrel PTP devices support multiple domains as specified in PTP standard, in which a device can syntonize with master clocks in 2 or more domains.

This setting is disabled/enabled by `set_hw_domain_check` function.

PTP Message	1-step								2-step							
	Master				Slave				Master				Slave			
	E2E		P2P		E2E		P2P		E2E		P2P		E2E		P2P	
	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2
Sync	−	**+**	−	**+**	M	+	M	+	**+**	−	**+**	−	+	−	+	−
Follow_Up	−	**+**	−	**+**	M	+	M	+	**+**	−	**+**	−	+	−	+	−
Delay_Req	M	**+**	−	−	−	+	−	−	+	−	−	−	+	−	−	−
Delay_Resp	−	**+**	−	−	M	+	−	−	**+**	−	−	−	+	−	−	−
Pdelay_Req	M	U	M	−	−	+	M	−	+	−	M	−	+	−	M	−
Pdelay_Resp	−	U	M	−	P	+	M	−	U	−	M	−	+	−	M	−
Pdelay_Resp_Follow_Up	−	U	M	−	P	+	M	−	U	−	M	−	+	−	M	−
Announce	P	+	P	+	P	+	P	+	+	P	+	P	+	P	+	P
Management	P	+	P	+	P	+	P	+	+	P	+	P	+	P	+	P
Signaling	P	+	P	+	P	+	P	+	+	P	+	P	+	P	+	P

*Table 5: **Domain Check** and **Alternate Master** Enabled*

If Alternate Master feature is used, the hardware Alternate Master setting can be enabled to forward Sync, Follow_Up, Delay_Req, and Delay_Resp messages in master clock without checking the domain number.

PTP Message	1-step								2-step							
	Master				Slave				Master				Slave			
	E2E		P2P		E2E		P2P		E2E		P2P		E2E		P2P	
	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2
Sync	−	+	−	+	M	+	M	+	+	−	+	−	+	−	+	−
Follow_Up	−	+	−	+	***S***	+	***S***	+	+	−	+	−	+	−	+	−
Delay_Req	M	+	−	−	−	+	−	−	+	−	−	−	+	−	−	−
Delay_Resp	−	+	−	−	***A***	***B***	−	−	+	−	−	−	+	−	−	−
Pdelay_Req	M	U	M	−	−	+	M	−	+	−	M	−	+	−	M	−
Pdelay_Resp	−	U	***C***	−	***C***	+	***C***	−	U	−	***C***	−	+	−	***C***	−
Pdelay_Resp_Follow_Up	−	U	***C***	−	***C***	+	***C***	−	U	−	***C***	−	+	−	***C***	−

*Table 6: **Delay_Req**, **Pdelay_Req**, and **Sync Associations** Enabled*

There is an additional filtering feature called message association. A Delay_Resp message needs to be associated with a certain Delay_Req message for software to accept the response. Likewise, Pdelay_Resp, and Pdelay_Resp_Follow_Up messages are associated with certain Pdelay_Req message. And Follow_Up message is associated with a 2-step Sync message. The response is considered associated with a request if the domain number, sequence id, and source port id match.

The Sync/Follow_Up Association setting checks the received Follow_Up message match the previous 2-step Sync message and drops it if not matched.

This setting is disabled/enabled by `set_hw_sync_assoc` function.

The Delay_Req/Delay_Resp Association setting checks the received Delay_Resp message match the Delay_Req message sent by the host. The message is forwarded to the host if matched and dropped if not. Additionally the message is forwarded to port 2 if not matched and dropped if matched. Messages with different sequence id are forwarded to the host.

This setting is disabled/enabled by `set_hw_delay_assoc` function.

The Pdelay_Req/Pdelay_Resp Association setting checks the received Pdelay_Resp and Pdelay_Resp_Follow_Up messages match the Pdelay_Req sent by the host. The messages are forwarded to the host if matched and dropped if not.

Note that this works even in 1-step E2E slave clock although software is not supposed to send out Pdelay_Req messages.

This setting is disabled/enabled by `set_hw_pdelay_assoc` function.

PTP Message	**1-step**								**2-step**							
	Master				**Slave**				**Master**				**Slave**			
	E2E		**P2P**		**E2E**		**P2P**		**E2E**		**P2P**		**E2E**		**P2P**	
	P3	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**	**P3**	**P2**
Sync	–	***D***	–	***D***	***D***	***D***	***D***	***D***	***D***	–	***D***	–	***D***	–	***D***	–
Follow_Up	–	***D***	–	***D***	***D***	***D***	***D***	***D***	***D***	–	***D***	–	***D***	–	***D***	–
Delay_Req	***D***	***D***	–	–	***D***	***D***	–	–	***D***	–	–	–	***D***	–	–	–
Delay_Resp	–	+	–	–	M	+	–	–	+	–	–	–	+	–	–	–
Pdelay_Req	M	U	M	–	–	+	M	–	+	–	M	–	+	–	M	–
Pdelay_Resp	–	U	M	–	P	+	M	–	U	–	M	–	+	–	M	–
Pdelay_Resp_Follow_Up	–	U	M	–	P	+	M	–	U	–	M	–	+	–	M	–

*Table 7: **Drop Sync and Delay_Req** Enabled*

There is a hardware setting called Drop Sync and Delay_Req, which drop all Sync, Follow_Up, and Delay_Req messages after receiving them. The idea was not to pass those messages to clocks connected to port 2 while the slave clock is setting up and so the system clock is not accurate. But the slave clock needs to receive Sync and Follow_Up messages to properly syntonize with the master. So this feature has no practical use.

The practice is to enable 2-step clock mode first to not pass any message to port 2, then switches back to 1-step clock mode after the system clock is syntonized with the master.

This setting is disabled/enabled by `set_hw_drop_sync` function.

PTP Message	1-step								2-step							
	Master				Slave				Master				Slave			
	E2E		P2P		E2E		P2P		E2E		P2P		E2E		P2P	
	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2	P3	P2
Sync	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
Follow_Up	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
Delay_Req	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
Delay_Resp	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
Pdelay_Req	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
Pdelay_Resp	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
Pdelay_Resp_Follow_Up	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
Announce	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***
Management	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***
Signaling	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***	+	***P***

*Table 8: **802.1AS** Enabled*

The 802.1AS setting is supposed to support running the clock in AVB. It forwards all messages to port 3 and does not forward any to port 2. Software may not actually use this setting when running in AVB as it may be desirable for hardware to forward some messages. For now this setting is used as a fallback in case the filtering mechanism mentioned above fails in some ways.

The current chip always forwards Announce, Management, and Signaling messages to the other port, which actually breaks the requirement of AVB and prevents the clock from being used as a Boundary Clock. The next revision of chip will correct that.

This setting is disabled/enabled by `set_hw_as` function.

8 Hardware Limitations

There are some limitations in current hardware revision that software needs to be aware of:

- Continual clock adjustment needs to be disabled first before doing nanosecond clock adjustment. Otherwise, operation may not be successful. This will be fixed in next revision.
- Infinite running trigger outputs like PPS need to be restarted after the clock changes significantly. Otherwise, the outputs may stop.

- Hardware expects the reserved fields of incoming PTP message header to be zero. Otherwise, wrong UDP checksum may be generated when the message is forwarded. Workaround is to disable UDP checksum generation. This will be fixed in next revision.
- Hardware expects the source port of transmitted Pdelay_Req message to be zero when running in P2P mode. Otherwise, wrong UDP checksum may be generated when the message is sent out from each port. This will be fixed in next revision.
- Hardware expects the originTimestamp of 1-step Sync message to be zero. Otherwise, wrong UDP checksum may be generated when the message is sent out from each port. This will be fixed in next revision.
- When calculated UDP checksum is zero hardware does not change it to 0xffff. As a result it is interpreted as no UDP checksum under IPv4 and rejected under IPv6. This will be fixed in next revision.
- There are issues in trigger outputs when the time interval is very small. As a result it is advised the minimum time interval be set at 80 ns.
- There are filtering errors when Domain Check and Sync Association are used. As a result Sync Association has to be disabled. This will be fixed in next revision.
- When VLAN Tag Insertion/Tag Removal features of the switch are used hardware does not modify the PTP message payload properly. Resolution is not to use those Tag Insertion/Tag Removal features. This will be fixed in next revision.
- In rare occasion under heavy network traffic hardware will calculate wrong correction field. This will be fixed in next revision.

Alphabetical Index