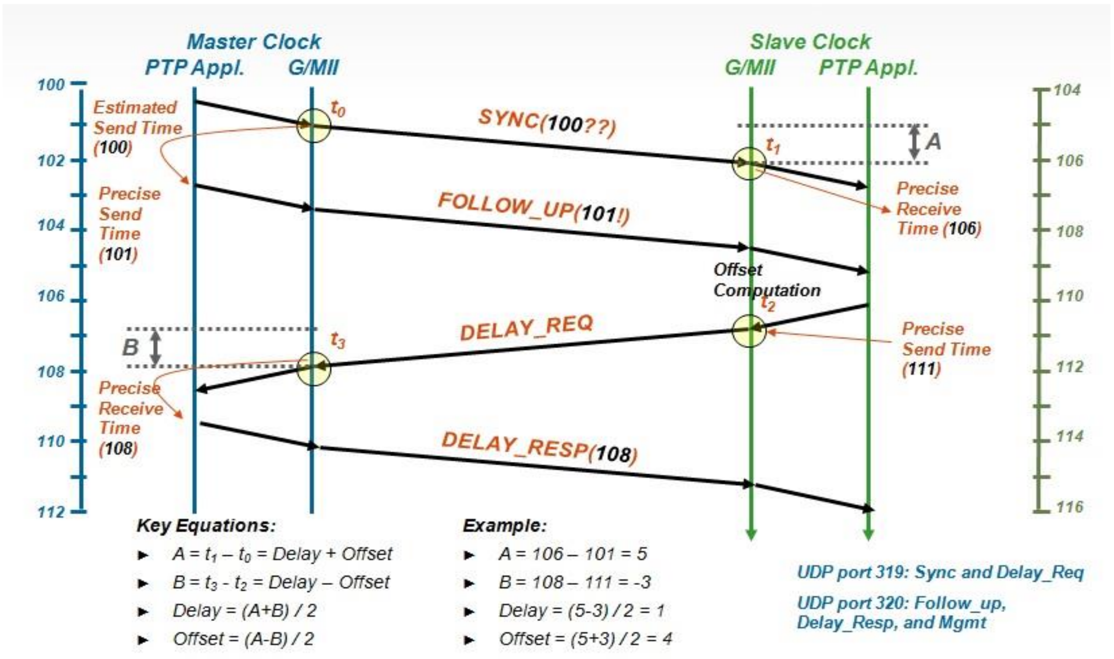
IEEE1588 Implementation in DPDK on DPAA2 Platforms

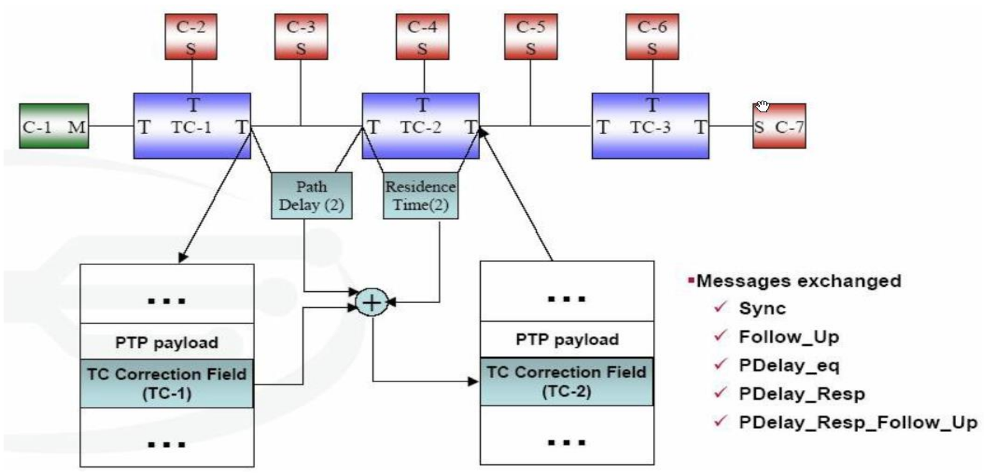
The Precision Time Protocol (PTP) is a protocol used to synchronize clock throughout a computer network. PTP was originally defined in IEEE1588-2002 standard.
To use PTP functionality in DPDK, users can use DPDK example application “ptpclient” present in DPDK source code, ptpclient application uses DPDK IEEE1588 API to communicate with a PTP master clock to synchronize the time on NIC and, optionally, on the Linux system.

IEEE1588 Introduction

IEEE Std 1588 standards for a precision clock synchronization protocol for networked measurement and control.
The standard defines a Precision Time Protocol (PTP) designed to synchronize real-time clocks in a distributed system
Targeted accuracy of microsecond to sub-microsecond with easy configuration and fast convergence between components.
IEEE588 Synchronization Message Sequence is as the following.



Peer-To-Peer Transparent Clock
Forwards and Corrects Sync and Follow Up messages only.
Correction achieved by addition of bridge residence time plus the peer to peer link delay into a Correction filed within the header of message.



Compile the Application

To enable IEEE1588, set 'CONFIG_RTE_LIBRTE_IEEE1588=y' in config/defconfig_arm64-dpaa2-linuxapp-gcc.

```
flex-builder -c openssl -a arm64 # to resolve the dependency on OpenSSL package
flex-builder -c linux -a arm64 # to resolve the dependency of KNI module
flex-builder -c dpdk -a arm64 # build dpdk application
flex-builder -c pktgen_dpdk -a arm64 # to generate dpdk pktgen application
flex-builder -i merge-component -a arm64
```

Running the Application

Start ptp server on tester machine. This will act as PTP Master. Suppose eth1 is tester_port which is connect to DUT.

```
#./ptp4l -i eth1 -m -2
ptp4l[581.659]: selected /dev/ptp1 as PTP clock
ptp4l[581.718]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[581.718]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[587.813]: port 1: LISTENING to MASTER on ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES
ptp4l[587.813]: selected local clock b26433.fffe.beb68c as best master
ptp4l[587.813]: assuming the grand master role
```

DUT machine: This machine will act as ptp slave. Create DPRTC instance and attach DPAA2 port to DPDK using dynamic_dpl script

To synchronize DUT PTP Hardware clock with Tester Machine PTP Hardware clock and additionally update system kernel clock

```
#./ptpclient -l 1 -n 1 -- -p 0x1 -T 1
```

```
root@localhost:~# export DPRTC_COUNT=1
root@localhost:~# source ./dynamic_dpl.sh dpmac.5
parent - dprc.1
Creating Non nested DPRC
NEW DPRCs
dprc.1
dprc.2
Using board type as 2088
Using High Performance Buffers
#################### Container dprc.2 is created ####################
                 Container dprc.2 have following resources :=>
                                * 1 DPMCP
                               * 16 DPBP
                               * 8 DPCON
                               * 8 DPSECI
                               * 1 DPNI
                               * 18 DPIO
                               * 2 DPCI
                               * 2 DPDMAI
                               * 1 DPRTC
########################## Configured Interfaces ##########################
Interface Name              Endpoint              Mac Address
==============         ========             ==================
dpni.1                    dpmac.5                   -Dynamic
root@localhost:~# date Mon Jul 1 21:41:26 UTC 2019
root@localhost:~# ./ptpclient -l 1 -n 1 -- -p 0x1 -T 0
EAL: Detected 8 lcore(s) EAL: Detected 1 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
fslmc: Skipping invalid device (power)
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: PCI device 0000:01:00.0 on NUMA socket -1
EAL: Invalid NUMA socket, default to 0
EAL: probe driver: 8086:10d3 net_e1000_em
PMD: dpni.1: netdev created
dpaa2_net: Rx offloads non configurable - requested 0x0 ignored 0x2000
dpaa2_net: Tx offloads non configurable - requested 0x18000 ignored 0x1c000
          Core 1 Waiting for SYNC packets. [Ctrl+C to quit]
          Master Clock id: 32:70:3e:ff:fe:ff:a6:59
          T2 - Slave Clock. 207s 560468378ns
          T1 - Master Clock. 19324s 999662036ns
          T3 - Slave Clock. 0s 0ns
          T4 - Master Clock. 19324s 999702684ns
          Delta between master and slave clocks:19221219448171ns
```

```
Comparison between Linux kernel Time and PTP:
Current PTP Time: Thu Jan 1 05:23:48 1970 780202685 ns
Current SYS Time: Mon Jul 1 21:42:10 2019 317847 ns
Delta between PTP and Linux Kernel time:-1561997901537542450ns
[Ctrl+C to quit]
root@localhost:~# date
Mon Jul 1 21:42:18 UTC 2019
root@localhost:~# ./ptpclient -l 1 -n 1 -- -p 0x1 -T 1
EAL: Detected 8 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
fslmc: Skipping invalid device (power)
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: PCI device 0000:01:00.0 on NUMA socket -1
EAL: Invalid NUMA socket, default to 0
EAL: probe driver: 8086:10d3 net_e1000_em
PMD: dpni.1: netdev created
 dpaa2_net: Rx offloads non configurable - requested 0x0 ignored 0x2000
 dpaa2_net: Tx offloads non configurable - requested 0x18000 ignored 0x1c000
 Core 1 Waiting for SYNC packets. [Ctrl+C to quit]
 Master Clock id: 32:70:3e:ff:fe:ff:a6:59
 T2 - Slave Clock. 20845s 385135978ns
 T1 - Master Clock. 19339s 999998152ns
 T3 - Slave Clock. 0s 0ns
 T4 - Master Clock. 19340s 23532ns
 Delta between master and slave clocks:8917307442853ns
 Comparison between Linux kernel Time and PTP:
 Current PTP Time: Thu Jan 1 08:16:02 1970 692874689 ns
 Current SYS Time: Thu Jan 1 08:16:02 1970 692915 ns
 Delta between PTP and Linux Kernel time:52105ns
 [Ctrl+C to quit]
 root@localhost:~# date
 Thu Jan 1 05:16:17 UTC 1970
 root@localhost:~#
```

Code Explanation

`Ptpclient:`

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
        rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
              MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
```

Mbufs are the packet buffer structure used by DPDK.

The `main()` function also initializes all the ports using the user defined `port_init()` function with portmask provided by user:

```
for (portid = 0; portid < nb_ports; portid++)
        if ((ptp_enabled_port_mask & (1 << portid)) != 0) {

                if (port_init(portid, mbuf_pool) == 0) {
                        ptp_enabled_ports[ptp_enabled_port_nb] = portid;
                        ptp_enabled_port_nb++;
                } else {
                        rte_exit(EXIT_FAILURE, "Cannot init port %"PRIu8 "\n",
                                        portid);
                }
        }
```

The main work of the application is done within the loop:

```
for (portid = 0; portid < ptp_enabled_port_nb; portid++) {

        portid = ptp_enabled_ports[portid];
        nb_rx = rte_eth_rx_burst(portid, 0, &m, 1);

        if (likely(nb_rx == 0))
                continue;

        if (m->ol_flags & PKT_RX_IEEE1588_PTP)
                parse_ptp_frames(portid, m);

        rte_pktmbuf_free(m);
}
```

Packets are received one by one on the RX ports and, if required, PTP response packets are transmitted on the TX ports.

The parse_ptp_frames() function processes PTP packets, implementing slave PTP IEEE1588 L2 functionality.

```
void
parse_ptp_frames(uint16_t portid, struct rte_mbuf *m) {
        struct ptp_header *ptp_hdr;
        struct rte_ether_hdr *eth_hdr;
        uint16_t eth_type;

        eth_hdr = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);
        eth_type = rte_be_to_cpu_16(eth_hdr->ether_type);

        if (eth_type == PTP_PROTOCOL) {
                ptp_data.m = m;
                ptp_data.portid = portid;
                ptp_hdr = (struct ptp_header *)(rte_pktmbuf_mtod(m, char *)
                                        + sizeof(struct rte_ether_hdr));

                switch (ptp_hdr->msgtype) {
                case SYNC:
                        parse_sync(&ptp_data);
                        break;
                case FOLLOW_UP:
                        parse_fup(&ptp_data);
                        break;
                case DELAY_RESP:
                        parse_drsp(&ptp_data);
                        print_clock_info(&ptp_data);
                        break;
                default:
                        break;
                }
        }
}
```

There are 3 types of packets on the RX path which we must parse to create a minimal implementation of the PTP slave client:

- SYNC packet.
- FOLLOW UP packet
- DELAY RESPONSE packet.

The MC exports the DPRTC object to allow GPP software to control the physical IEEE-1588 Real Time Clock. A single DPRTC object is needed to control the IEEE-1588 RTC, and this object is expected to serve the PTP stack running in GPP.

drivers/net/dpaa2/mc/dprtc.c
dprtc_open() - Open a control session for the specified object. This function can be used to open a control session for an already created object; an object may have been declared in the DPL or by calling the dprtc_create function. This function returns a unique authentication token, associated with the specific object ID and the specific MC portal; this token must be used in all subsequent commands for this specific object.

dprtc_close() - Close the control session of the object. After this function is called, no further operations are allowed on the object without opening a new control session.

dprtc_create() - Create the DPRTC object.
Create the DPRTC object, allocate required resources and perform required initialization.
The function accepts an authentication token of a parent container that this object should be assigned to. The token can be '0' so the object will be assigned to the default container. The newly created object can be opened with the returned object id and using the container's associated tokens and MC portals.

dprtc_destroy() - Destroy the DPRTC object and release all its resources. The function accepts the authentication token of the parent container that created the object (not the one that currently owns the object). The object is searched within parent using the provided 'object_id'. All tokens to the object must be closed before calling destroy.

dprtc_enable() - Enable the DPRTC.
dprtc_disable() - Disable the DPRTC.
dprtc_reset() - Reset the DPRTC, returns the object to initial state.
dprtc_get_attributes - Retrieve DPRTC attributes.
dprtc_set_clock_offset() - Sets the clock's offset (usually relative to another clock).
dprtc_set_freq_compensation() - Sets a new frequency compensation value.
dprtc_get_freq_compensation() - Retrieves the frequency compensation value
dprtc_get_time() - Returns the current RTC time.
dprtc_set_time() - Updates current RTC time.
dprtc_set_alarm() - Defines and sets alarm.

drivers/net/dpaa2/dpaa2_ethdev.h
```
struct dpaa2_dev_priv {
…
#if defined(RTE_LIBRTE_IEEE1588)
		/*stores timestamp of last received packet on dev*/
		uint64_t rx_timestamp;
		/*stores timestamp of last received tx confirmation packet on dev*/
```

```
        uint64_t tx_timestamp;
        /* stores pointer to next tx_conf queue that should be processed,
         * it corresponds to last packet transmitted
         */
        struct dpaa2_queue *next_tx_conf_queue;
#endif
…
}

#if defined(RTE_LIBRTE_IEEE1588)
int dpaa2_timesync_read_time(struct rte_eth_dev *dev,
                                        struct timespec *timestamp);
int dpaa2_timesync_write_time(struct rte_eth_dev *dev,
                                        const struct timespec *timestamp);
int dpaa2_timesync_adjust_time(struct rte_eth_dev *dev, int64_t delta);
int dpaa2_timesync_read_rx_timestamp(struct rte_eth_dev *dev,
                                                struct timespec *timestamp,
                                                uint32_t flags __rte_unused);
int dpaa2_timesync_read_tx_timestamp(struct rte_eth_dev *dev,
                                          struct timespec *timestamp);
#endif
```

drivers/net/dpaa2/dpaa2_ethdev.c

```
static struct eth_dev_ops dpaa2_ethdev_ops = {
…
#if defined(RTE_LIBRTE_IEEE1588)
        .timesync_read_time     = dpaa2_timesync_read_time,
        .timesync_write_time    = dpaa2_timesync_write_time,
        .timesync_adjust_time = dpaa2_timesync_adjust_time,
        .timesync_read_rx_timestamp = dpaa2_timesync_read_rx_timestamp,
        .timesync_read_tx_timestamp = dpaa2_timesync_read_tx_timestamp,
#endif
…
}
```

Callback to handle sending packets through WRIOP based interface

drivers/net/dpaa2/dpaa2_rxtx.c

```
dpaa2_dev_tx
{
…
#ifdef RTE_LIBRTE_IEEE1588
        /* IEEE1588 driver need pointer to tx confirmation queue
```

```
                * corresponding to last packet transmitted for reading
                * the timestamp
                */
              priv->next_tx_conf_queue = dpaa2_q->tx_conf_queue;
              dpaa2_dev_tx_conf(dpaa2_q->tx_conf_queue);
#endif
…

#ifdef RTE_LIBRTE_IEEE1588
                                        enable_tx_tstamp(&fd_arr[loop]);
#endif
…
}
```

drivers/net/dpaa2/dpaa2_ptp.c

```
int dpaa2_timesync_read_time(struct rte_eth_dev *dev,
                                        struct timespec *timestamp)
{
…
ret = dprtc_get_time(&dprtc_dev->dprtc, CMD_PRI_LOW,
                                dprtc_dev->token, &ns);
…
}

int dpaa2_timesync_write_time(struct rte_eth_dev *dev,
                                        const struct timespec *ts)
{
…
ret = dprtc_set_time(&dprtc_dev->dprtc, CMD_PRI_LOW,
                                dprtc_dev->token, ns);
…
}

int dpaa2_timesync_adjust_time(struct rte_eth_dev *dev, int64_t delta)
{
…
ret = dprtc_get_time(&dprtc_dev->dprtc, CMD_PRI_LOW,
                                dprtc_dev->token, &ns);
…
ret = dprtc_set_time(&dprtc_dev->dprtc, CMD_PRI_LOW,
                                dprtc_dev->token, ns);
…
}
```

```
static int
dpaa2_create_dprtc_device(int vdev_fd __rte_unused,
                          struct vfio_device_info *obj_info __rte_unused,
                          int dprtc_id)
{
/* Allocate DPAA2 dprtc handle */
        dprtc_dev = rte_malloc(NULL, sizeof(struct dpaa2_dprtc_dev), 0);
        if (!dprtc_dev) {
                DPAA2_PMD_ERR("Memory allocation failed for DPRTC Device");
                return -1;
        }

        /* Open the dprtc object */
        dprtc_dev->dprtc.regs = rte_mcp_ptr_list[MC_PORTAL_INDEX];
        ret = dprtc_open(&dprtc_dev->dprtc, CMD_PRI_LOW, dprtc_id,
                         &dprtc_dev->token);
…
}
```