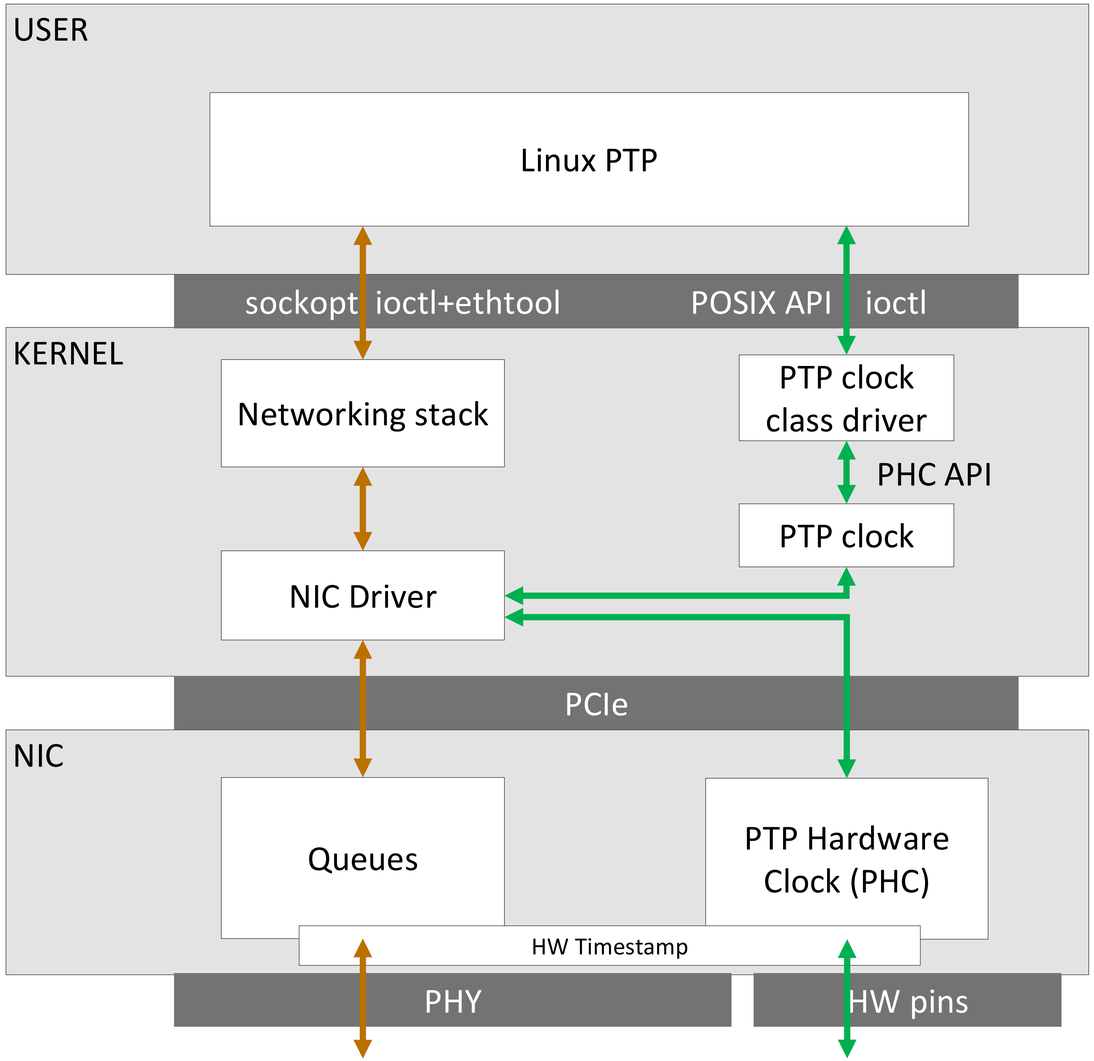
Introduction to PTP on Linux - APIs

Maciek Machnikowski | netDev 0x18

Agenda:

- Overview
- Netdev timestamp ioctl
- POSIX clock
- PHC ioctl
- Tx timestamping
- Rx timestamping

Overview


USER
Linux PTP
sockopt
ioctl+ethtool
POSIX API
ioctl
KERNEL
Networking stack
PTP clock
class driver
PHC API
PTP clock
NIC Driver
PCIe
NIC
Queues
PTP Hardware
Clock (PHC)
HW Timestamp
PHY
HW pins

Control/info path

Ethtool

- ethtool_ops. get_ts_info

- Returns an index of PHC that’s connected to a given netdev
- Informs userspace about timestamping capabilities
  - Tx modes
  - Supported Rx filters
- Used by ptp4l to resolve netdev to the PHC connection

Netdev ioctl

- Ethernet-specific IOCTLS
  - SIOCGHWTSTAMP - gets timestamp configuration
  - SIOCSHWTSTAMP - sets timestamp configuration and reads back

ndo_hwtstamp_get
ndo_hwtstamp_set

- Introduced in Kernel 6.6
- Easier to implement than the legacy way
- Wrapper handles copying the config buffer data from/to user

```
*
* int (*ndo_hwtstamp_get)(struct net_device *dev,
*                         struct kernel_hwtstamp_config *kernel_config);
*      Get the currently configured hardware timestamping parameters for the
*      NIC device.
*
* int (*ndo_hwtstamp_set)(struct net_device *dev,
*                         struct kernel_hwtstamp_config *kernel_config,
*                         struct netlink_ext_ack *extack);
*      Change the hardware timestamping parameters for NIC device.
*/
```

ndo_eth_ioctl (legacy)

- Fallback for SIOCGHWTSTAMP/ SIOCSHWTSTAMP
- Requires manual handling of userspace buffer
- One entry point for both flags

Kernel_hwtstamp_config structure

- Conveys the timestamping configuration
- Contains the type of TX
  - Two-step
  - One-step
  - Off
- Configures custom RX filters
  - HW filters for specific packets
  - If a driver does not support a specific configuration - it may fall back to a more generic one and return it

```
/**
 * struct kernel_hwtstamp_config - Kernel copy of struct hwtstamp_config
 *
 * @flags: see struct hwtstamp_config
 * @tx_type: see struct hwtstamp_config
 * @rx_filter: see struct hwtstamp_config
 * @ifr: pointer to ifreq structure from the original ioctl request, to pass to
 *      a legacy implementation of a lower driver
 * @copied_to_user: request was passed to a legacy implementation which already
 *      copied the ioctl request back to user space
 * @source: indication whether timestamps should come from the netdev or from
 *      an attached phylib PHY
 *
 * Prefer using this structure for in-kernel processing of hardware
 * timestamping configuration, over the inextensible struct hwtstamp_config
 * exposed to the %SIOCGHWTSTAMP and %SIOCSHWTSTAMP ioctl UAPI.
 */
struct kernel_hwtstamp_config {
        int flags;
        int tx_type;
        int rx_filter;
        struct ifreq *ifr;
        bool copied_to_user;
        enum hwtstamp_source source;
};
```

POSIX clock

POSIX Clock API

- PTP clocks conform to POSIX clock standard
- clock_adjtime: Adjust the clock
- clock_gettime: Read the current time
- clock_settime: Set the current time
- ioctl: Optional IOCTL methods

POSIX Clock API

- PTP clocks are exposed as character devices (/dev/ptpX)
- The file can be opened by a process
- File descriptor may be converted into clock IDs
- This clock ID can be used by standard POSIX APIs

```
#define CLOCKFD 3
#define FD_TO_CLOCKID(fd)   ((~(clockid_t) (fd) << 3) | CLOCKFD)
#define CLOCKID_TO_FD(clk)  ((unsigned int) ~((clk) >> 3))

struct timeval tv;
clockid_t clkid;
int fd;

fd = open("/dev/ptp0", O_RDWR);
clkid = FD_TO_CLOCKID(fd);
clock_gettime(clkid, &tv);
```

Mapping POSIX API to PTP clock driver API

```
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

clock_adjtime

```
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

if (tx->modes & ADJ_FREQUENCY)

if (tx->modes & ADJ_OFFSET)

if (tx->modes & ADJ_SETOFFSET)

```
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

clock_adjtime

- Not part of IEEE POSIX specification
- Operates on the `timex` structure
- If called without any flags set – it returns info about the clock
  - For PTP clocks - information is limited to the current freq offset

```
struct __kernel_timex {
        unsigned int modes;     /* mode selector */
        int :32;            /* pad */
        long long offset;       /* time offset (usec) */
        long long freq; /* frequency offset (scaled ppm) */
        long long maxerror;/* maximum error (usec) */
        long long esterror;/* estimated error (usec) */
        int status;             /* clock command/status */
        int :32;            /* pad */
        long long constant;/* pll time constant */
        long long precision;/* clock precision (usec) (read only) */
        long long tolerance;/* clock frequency tolerance (ppm)
                               * (read only)
                               */
        struct __kernel_timex_timeval time;     /* (read only, except for ADJ_SETOFFSET) */
        long long tick; /* (modified) usecs between clock ticks */

        long long ppsfreq;/* pps frequency (scaled ppm) (ro) */
        long long jitter; /* pps jitter (us) (ro) */
        int shift;              /* interval duration (s) (shift) (ro) */
        int :32;            /* pad */
        long long stabil;           /* pps stability (scaled ppm) (ro) */
        long long jitcnt; /* jitter limit exceeded (ro) */
        long long calcnt; /* calibration intervals (ro) */
        long long errcnt; /* calibration errors (ro) */
        long long stbcnt; /* stability limit exceeded (ro) */

        int tai;                /* TAI offset (ro) */

        int  :32; int  :32; int  :32; int  :32;
        int  :32; int  :32; int  :32; int  :32;
        int  :32; int  :32; int  :32;
};
```

clock_adjtime

- When called with
- Adjfine (`tx->modes & ADJ_FREQUENCY`)
  - Adjusts the frequency of the hardware clock
- Adjphase (`tx->modes & ADJ_OFFSET`)
  - Adjusts the PHC by a given number of nanoseconds
  - PHC should use an internal servo algorithm to consume the phase offset
- Adjtime (`tx->modes & ADJ_SETOFFSET`)
  - Shifts the time of the hardware clock by the desired change in nanoseconds
- Without flags
  - Returns current clock state

clock_gettime

```
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function_func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

gettime64: Reads the current time from the hardware clock

clock_gettime

- Return the current value of time
- Calls:
  - `gettimex64` (if implemented)
    - Reads the current time from the hardware clock and optionally the system clock
  - `gettime64` (otherwise)
    - Reads the current time from the hardware clock

clock_getres

```
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function_func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

```
tp->tv_sec = 0;
tp->tv_nsec = 1;
return 0;
```

clock_getres

- Return the resolution of a clock
- Currently statically terminated in the ptp_clock class driver
- Is the clock resolution, not the clock tick rate

```
static int ptp_clock_getres(struct posix_clock *pc, struct timespec64 *tp)
{
        tp->tv_sec = 0;
        tp->tv_nsec = 1;
        return 0;
}
```

clock_settime

```
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function_func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

settime64:

clock_settime

- Set the specified clock
- Check if the clock is freerunning
- If not – call settime64

IOCTLs

IOCTL swiss army knife

- Allows advanced operation on PTP clocks
- Read clock capabilities
- Control external timestamp source
- Control periodic outputs
- Control PPS subsystem
- Get hw-accelerated cross-timestamps
- Capture offset between the clock and system time (in 2 ways)
- Control auxiliary pins
- (IOCTLs with 2 do strict flags/args checking)

PTP_CLOCK_GETCAPS
PTP_CLOCK_GETCAPS2

- Returns PTP clock capabilities
- Generic ptp_clock class driver function
- No extra implementation is needed

```
struct ptp_clock_caps {
        int max_adj;   /* Maximum frequency adjustment in parts per billon. */
        int n_alarm;   /* Number of programmable alarms. */
        int n_ext_ts;  /* Number of external time stamp channels. */
        int n_per_out; /* Number of programmable periodic signals. */
        int pps;       /* Whether the clock supports a PPS callback. */
        int n_pins;    /* Number of input/output pins. */
        /* Whether the clock supports precise system-device cross timestamps */
        int cross_timestamping;
        /* Whether the clock supports adjust phase */
        int adjust_phase;
        int max_phase_adj; /* Maximum phase adjustment in nanoseconds. */
        int rsv[11];       /* Reserved for future use. */
};
```

PTP_EXTTS_REQUEST
PTP_EXTTS_REQUEST2

```c
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```c
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

n_ext_ts: number of external time stamp channels
enable: request driver to enable or disable an ancillary feature

PTP_EXTTS_REQUEST
PTP_EXTTS_REQUEST2

- Controls external timestamp sources
  - PPS input pin
- Is the PPS input that reports PHC timestamps via the PTP clock
- Verifies the request against `n_ext_ts`
- Calls `enable` function

PTP_PEROUT_REQUEST
PTP_PEROUT_REQUEST2

```c
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```c
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function_func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

n_ext_ts: number of programmable periodic signals
enable: request driver to enable or disable an ancillary feature

PTP_PEROUT_REQUEST
PTP_PEROUT_REQUEST2

- Controls external periodic signals outputs
  - PPS output pin
- Verifies the request against `n_per_out`
- Calls `enable` function

PTP_ENABLE_PPS
PTP_ENABLE_PPS2

```c
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```c
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function_func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

pps: Indicates whether the clock supports a PPS callback
enable: request driver to enable or disable an ancillary feature

PTP_ENABLE_PPS
PTP_ENABLE_PPS2

- Controls PPS subsystem callback of the clock
- Is NOT PPS input that reports PHC timestamps via the PTP clock
- Calls `enable` function

PTP_SYS_OFFSET_PRECISE
PTP_SYS_OFFSET_PRECISE2

```c
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```c
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function_func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

PTP_SYS_OFFSET_PRECISE
PTP_SYS_OFFSET_PRECISE2

- Require hardware support
- Uses cross timestamping hardware features (such as PTM)
- Capture the system and the PHC time simultaneously.
- Provides exact offset measurement between system clock and PHC time

```
struct ptp_sys_offset_precise {
        struct ptp_clock_time device;
        struct ptp_clock_time sys_realtime;
        struct ptp_clock_time sys_monoraw;
        unsigned int rsv[4];    /* Reserved for future use. */
};
```

PTP_SYS_OFFSET_EXTENDED
PTP_SYS_OFFSET_EXTENDED2

```c
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```c
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function_func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);
};
```

PTP_SYS_OFFSET_EXTENDED
PTP_SYS_OFFSET_EXTENDED2

- Calls gettimex64
- Requests system offset measurement
- Reads system time, PHC time, and then system time in the driver
- Can capture multiple samples to allow averaging

```
struct ptp_sys_offset_extended {
        unsigned int n_samples; /* Desired number of measurements. */
        unsigned int rsv[3];    /* Reserved for future use. */
        /*
         * Array of [system, phc, system] time stamps. The kernel will provide
         * 3*n_samples time stamps.
         */
        struct ptp_clock_time ts[PTP_MAX_SAMPLES][3];
};
```

PTP_SYS_OFFSET
PTP_SYS_OFFSET2

```c
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```c
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function_func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);

};
```

gettime64: Reads the current time from the hardware clock
gettimex64: Reads the current time from the hardware clock and, optionally, the system clock.

PTP_SYS_OFFSET
PTP_SYS_OFFSET2

- Simplified legacy function for reading the system offset
- Reads the system time in `ptp_chardev`
- Calls to either `gettimex64` or `gettime64`
- Reads the system time again
- Can capture multiple samples to allow averaging

PTP_PIN_GETFUNC
PTP_PIN_GETFUNC2

- Retrieve pin config of a requested pin index

```
struct ptp_pin_desc {
        /*
         * Hardware specific human readable pin name. This field is
         * set by the kernel during the PTP_PIN_GETFUNC ioctl and is
         * ignored for the PTP_PIN_SETFUNC ioctl.
         */
        char name[64];
        /*
         * Pin index in the range of zero to ptp_clock_caps.n_pins - 1.
         */
        unsigned int index;
        /*
         * Which of the PTP_PF_xxx functions to use on this pin.
         */
        unsigned int func;
        /*
         * The specific channel to use for this function.
         * This corresponds to the 'index' field of the
         * PTP_EXTTS_REQUEST and PTP_PEROUT_REQUEST ioctls.
         */
        unsigned int chan;
        /*
         * Reserved for future use.
         */
        unsigned int rsv[5];
};
```

PTP_PIN_SETFUNC
PTP_PIN_SETFUNC2

```
    case PTP_PIN_SETFUNC:
    case PTP_PIN_SETFUNC2:
```

```
static struct posix_clock_operations ptp_clock_ops = {
        .owner          = THIS_MODULE,
        .clock_adjtime  = ptp_clock_adjtime,
        .clock_gettime  = ptp_clock_gettime,
        .clock_getres   = ptp_clock_getres,
        .clock_settime  = ptp_clock_settime,
        .ioctl          = ptp_ioctl,
        .open           = ptp_open,
        .release        = ptp_release,
        .poll           = ptp_poll,
        .read           = ptp_read,
};
```

```
struct ptp_clock_info {
        struct module *owner;
        char name[PTP_CLOCK_NAME_LEN];
        s32 max_adj;
        int n_alarm;
        int n_ext_ts;
        int n_per_out;
        int n_pins;
        int pps;
        struct ptp_pin_desc *pin_config;
        int (*adjfine)(struct ptp_clock_info *ptp, long scaled_ppm);
        int (*adjphase)(struct ptp_clock_info *ptp, s32 phase);
        s32 (*getmaxphase)(struct ptp_clock_info *ptp);
        int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
        int (*gettime64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*gettimex64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                          struct ptp_system_timestamp *sts);
        int (*getcrosststamp)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*settime64)(struct ptp_clock_info *p, const struct timespec64 *ts);
        int (*getcycles64)(struct ptp_clock_info *ptp, struct timespec64 *ts);
        int (*getcyclesx64)(struct ptp_clock_info *ptp, struct timespec64 *ts,
                            struct ptp_system_timestamp *sts);
        int (*getcrosscycles)(struct ptp_clock_info *ptp,
                              struct system_device_crosststamp *cts);
        int (*enable)(struct ptp_clock_info *ptp,
                      struct ptp_clock_request *request, int on);
        int (*verify)(struct ptp_clock_info *ptp, unsigned int pin,
                      enum ptp_pin_function func, unsigned int chan);
        long (*do_aux_work)(struct ptp_clock_info *ptp);

};
```

verify: Confirm that a pin can perform a given function. This hook gives
drivers a way of telling the PHC core about limitations on specific pins.
parameter pin: index of the pin in question.
parameter func: the desired function to use.
parameter chan: the function channel index to use.

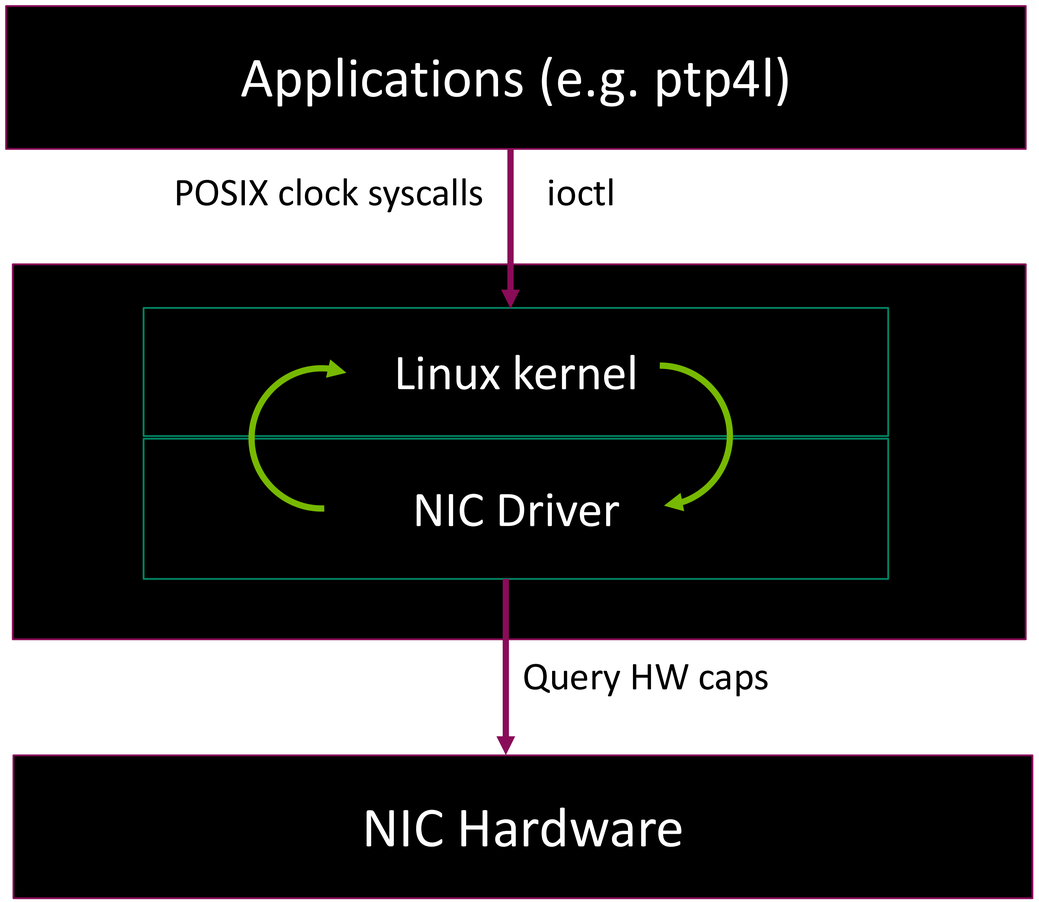
PTP_PIN_SETFUNC
PTP_PIN_SETFUNC2

- Controls advanced pin assignment.
  - Such as bi-dir pins
- Verifies if a given pin configuration is supported
- Stores configuration in a local table
- **ptp_find_pin** function can retrieve a pin index assigned to a given function/channel configuration
  - inside the driver (enable)

Tx/Rx Timestamping

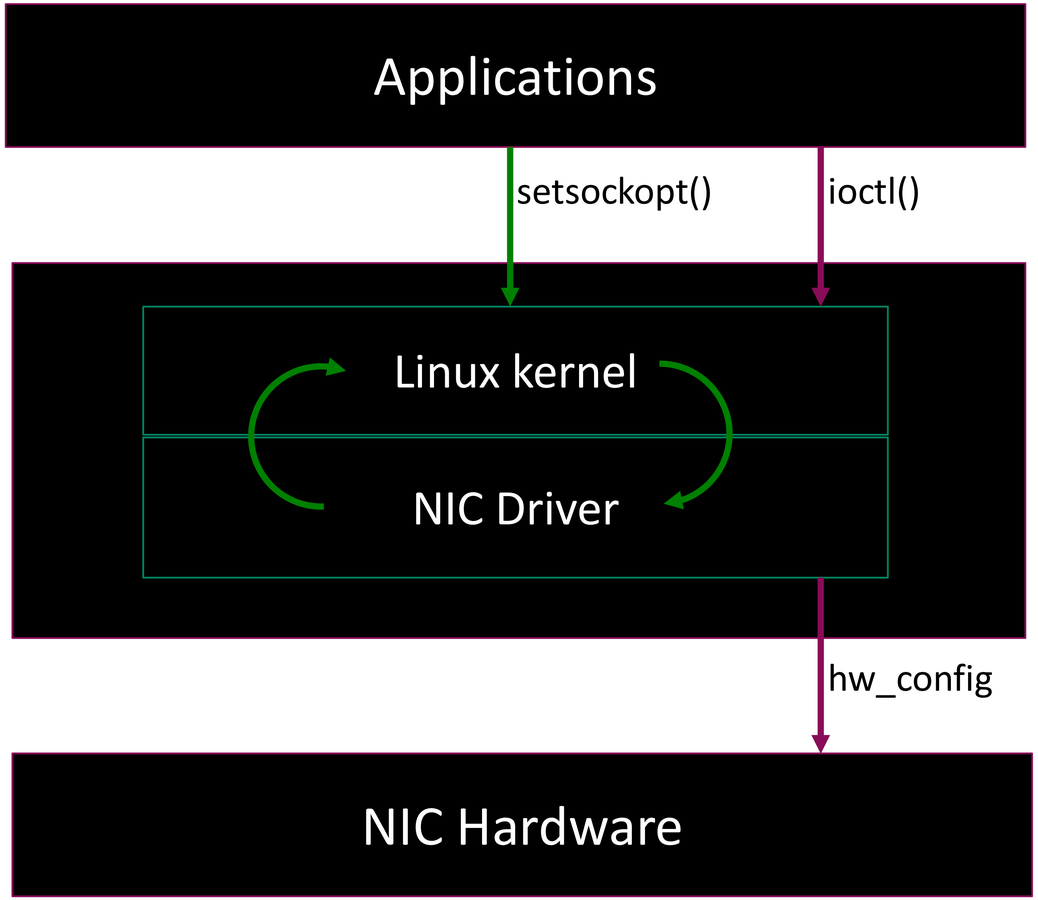
HW Time - control

- Driver creates `ptp_clock_info`
  - `pps, n_per_out, max_adj, etc.`
  - `adjfreq/adjtime/gettime/settime/enable`
- registers clock via `ptp_clock_register()`
- Applications utilize POSIX clock system calls and ioctls to control the PHC



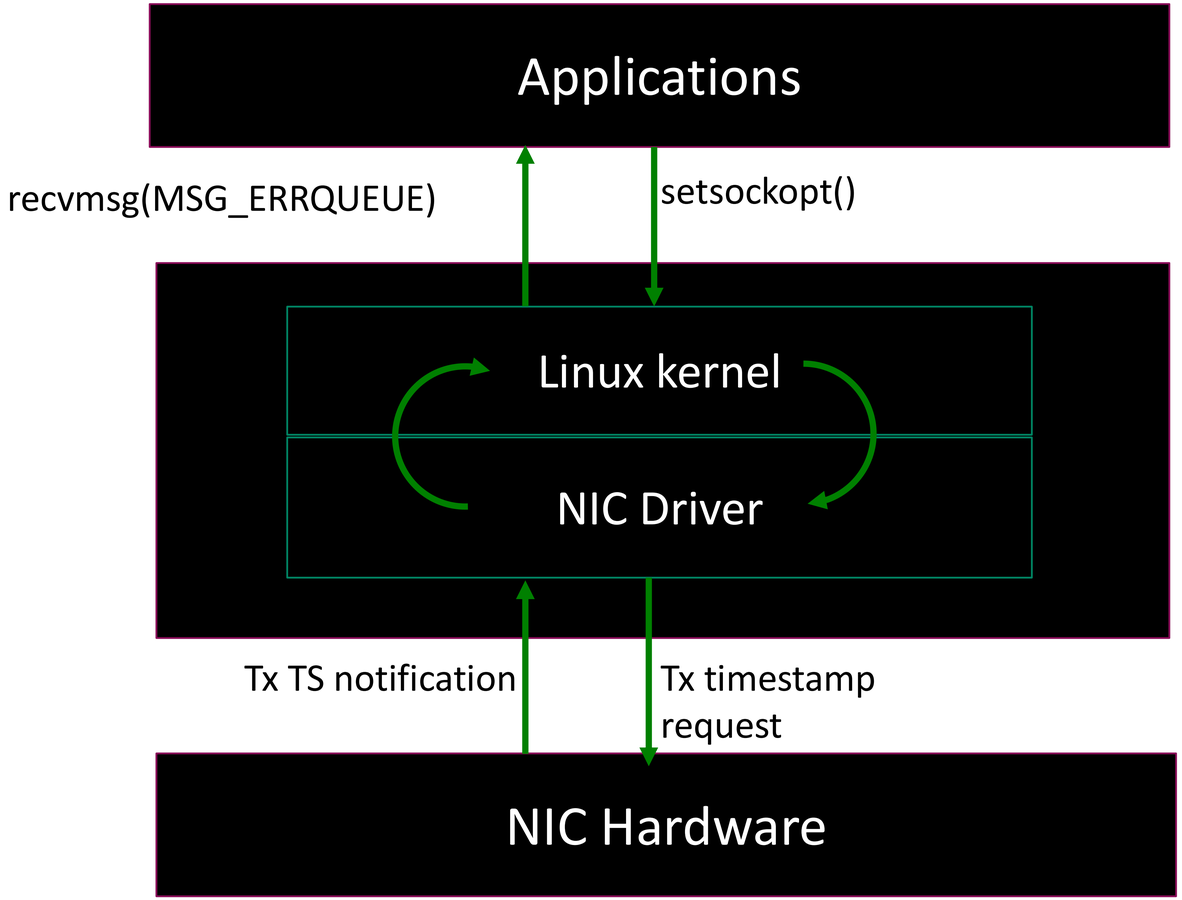
HW Tx Timestamping – control

- App sets Tx timestamping using SIOCSHWTSTAMP
- Driver configures timestamp in hardware
  - Rx/Tx Timestamping disabled by default
- App sets socket options for a created socket to enable Tx timestamps



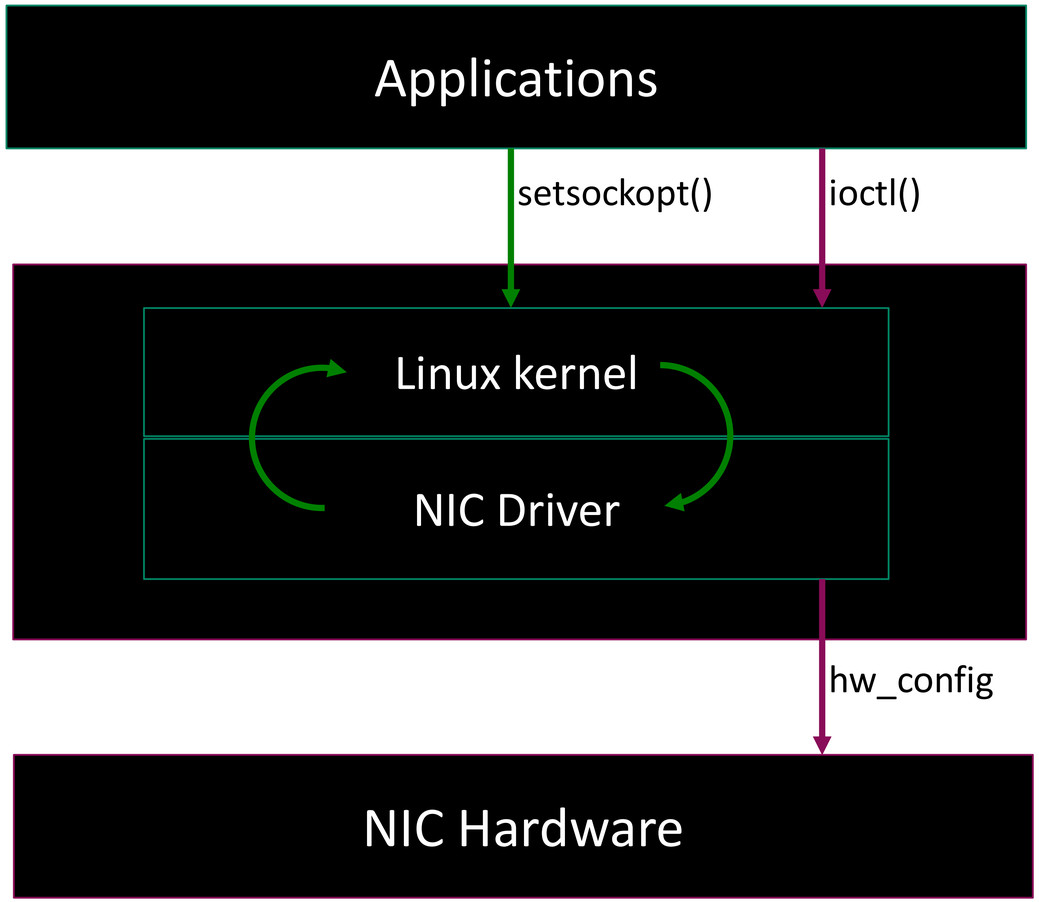
NIC HW Tx Timestamping – Data

- Application uses setsockopt to set SOF_TIMESTAMPING_TX_HARDWARE flag
- Driver requests HW Tx timestamps for packets with SKBTX_HW_TSTAMP flag
- Driver sets SKBTX_IN_PROGRESS flag to notify it is waiting for a hw tstamp and sends the packet
- NIC Hardware notifies the driver once Timestamp is collected
- NIC Driver
  - Sets shhwtstamps.hwtstamp
  - Calls skb_tstamp_tx with the original skb and shwtstamps struct
- Application informed via MSG_ERRQUEUE



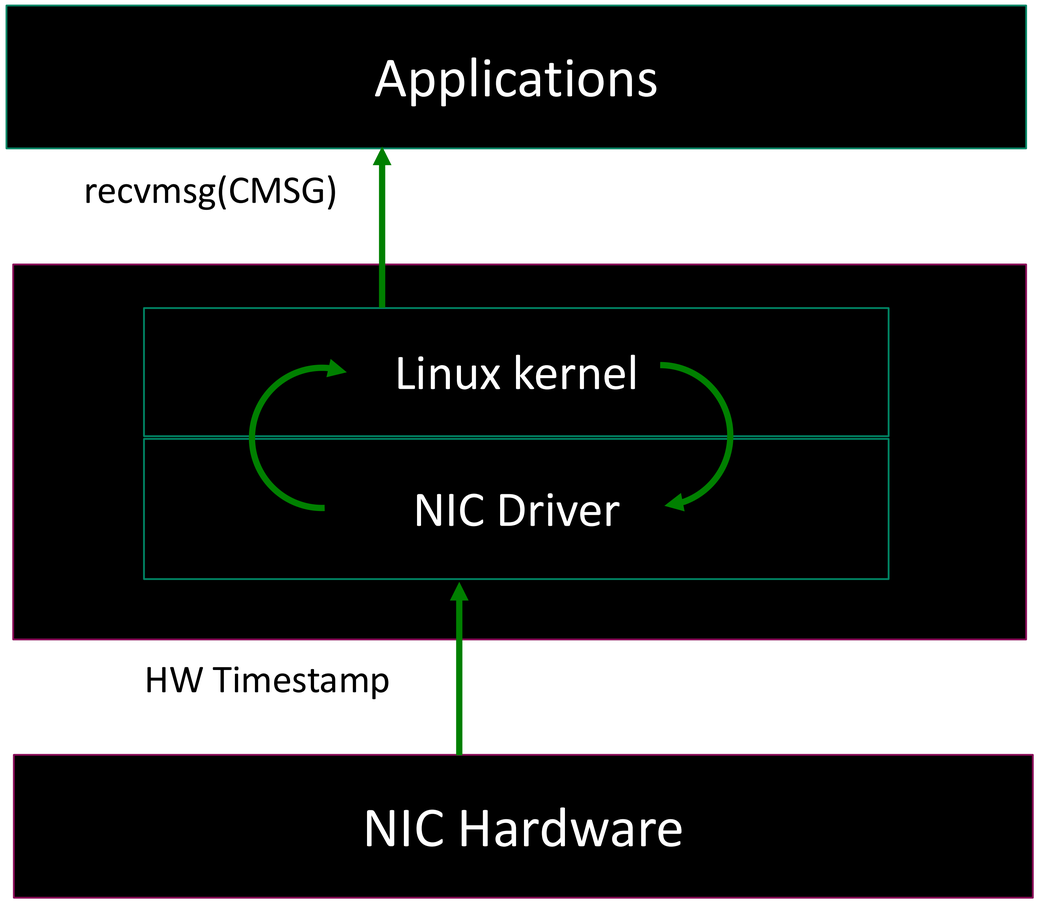
HW Rx Timestamping – control

- App sets Tx timestamping using SIOCSHWTSTAMP
- Driver sets up HW to generate timestamps
- Application uses setsockopt to set SOF_TIMESTAMPING_RX_HARDWARE flag



HW Rx Timestamping – data

- Hardware timestamps Rx packets
- And notifies the driver
- Driver reads Rx descriptor, Rx timestamp
- And sets the skb_hwtstamps(skb)->hwtstamp
- Application informed via CMSG



Useful links

- https://elixir.bootlin.com/linux/latest/source/Documentation/driver-api/ptp.rst
- https://elixir.bootlin.com/linux/latest/source/Documentation/timers/timekeeping.rst
- https://elixir.bootlin.com/linux/latest/source/Documentation/networking/timestamping.rst