Timestamping

1. Control Interfaces

The interfaces for receiving network packages timestamps are:

SO_TIMESTAMP
: Generates a timestamp for each incoming packet in (not necessarily monotonic) system time. Reports the timestamp via recvmsg() in a control message in usec resolution. SO_TIMESTAMP is defined as SO_TIMESTAMP_NEW or SO_TIMESTAMP_OLD based on the architecture type and time_t representation of libc. Control message format is in struct __kernel_old_timeval for SO_TIMESTAMP_OLD and in struct __kernel_sock_timeval for SO_TIMESTAMP_NEW options respectively.

SO_TIMESTAMPNS
: Same timestamping mechanism as SO_TIMESTAMP, but reports the timestamp as struct timespec in nsec resolution. SO_TIMESTAMPNS is defined as SO_TIMESTAMPNS_NEW or SO_TIMESTAMPNS_OLD based on the architecture type and time_t representation of libc. Control message format is in struct timespec for SO_TIMESTAMPNS_OLD and in struct __kernel_timespec for SO_TIMESTAMPNS_NEW options respectively.

IP_MULTICAST_LOOP + SO_TIMESTAMP[NS]
: Only for multicast:approximate transmit timestamp obtained by reading the looped packet receive timestamp.

SO_TIMESTAMPING
: Generates timestamps on reception, transmission or both. Supports multiple timestamp sources, including hardware. Supports generating timestamps for stream sockets.

1.1 SO_TIMESTAMP (also SO_TIMESTAMP_OLD and SO_TIMESTAMP_NEW)

This socket option enables timestamping of datagrams on the reception path. Because the destination socket, if any, is not known early in the network stack, the feature has to be enabled for all packets. The same is true for all early receive timestamp options.

For interface details, see *man 7 socket*.

Always use SO_TIMESTAMP_NEW timestamp to always get timestamp in struct __kernel_sock_timeval format.

SO_TIMESTAMP_OLD returns incorrect timestamps after the year 2038 on 32 bit machines.

1.2 SO_TIMESTAMPNS (also SO_TIMESTAMPNS_OLD and SO_TIMESTAMPNS_NEW)

This option is identical to SO_TIMESTAMP except for the returned data type. Its struct timespec allows for higher resolution (ns) timestamps than the timeval of SO_TIMESTAMP (ms).

Always use SO_TIMESTAMPNS_NEW timestamp to always get timestamp in struct __kernel_timespec format.

SO_TIMESTAMPNS_OLD returns incorrect timestamps after the year 2038 on 32 bit machines.

1.3 SO_TIMESTAMPING (also SO_TIMESTAMPING_OLD and SO_TIMESTAMPING_NEW)

Supports multiple types of timestamp requests. As a result, this socket option takes a bitmap of flags, not a boolean. In:

```
err = setsockopt(fd, SOL_SOCKET, SO_TIMESTAMPING, &val, sizeof(val));
```

val is an integer with any of the following bits set. Setting other bit returns EINVAL and does not change the current state.

The socket option configures timestamp generation for individual sk_buffs (1.3.1), timestamp reporting to the socket's error queue (1.3.2) and options (1.3.3). Timestamp generation can also be enabled for individual sendmsg calls using cmsg (1.3.4).

1.3.1 Timestamp Generation

Some bits are requests to the stack to try to generate timestamps. Any combination of them is valid. Changes to these bits apply to newly created packets, not to packets already in the stack. As a result, it is possible to selectively request timestamps for a subset of packets (e.g., for sampling) by embedding an send() call within two setsockopt calls, one to enable timestamp generation and one to disable it. Timestamps may also be generated for reasons other than being requested by a particular socket, such as when receive timestamping is enabled system wide, as explained earlier.

SOF_TIMESTAMPING_RX_HARDWARE:
: Request rx timestamps generated by the network adapter.

SOF_TIMESTAMPING_RX_SOFTWARE:
: Request rx timestamps when data enters the kernel. These timestamps are generated just after a device driver hands a packet to the kernel receive stack.

SOF_TIMESTAMPING_TX_HARDWARE:
: Request tx timestamps generated by the network adapter. This flag can be enabled via both socket options and control

messages.

SOF_TIMESTAMPING_TX_SOFTWARE:
Request tx timestamps when data leaves the kernel. These timestamps are generated in the device driver as close as possible, but always prior to, passing the packet to the network interface. Hence, they require driver support and may not be available for all devices. This flag can be enabled via both socket options and control messages.

SOF_TIMESTAMPING_TX_SCHED:
Request tx timestamps prior to entering the packet scheduler. Kernel transmit latency is, if long, often dominated by queuing delay. The difference between this timestamp and one taken at SOF_TIMESTAMPING_TX_SOFTWARE will expose this latency independent of protocol processing. The latency incurred in protocol processing, if any, can be computed by subtracting a userspace timestamp taken immediately before send() from this timestamp. On machines with virtual devices where a transmitted packet travels through multiple devices and, hence, multiple packet schedulers, a timestamp is generated at each layer. This allows for fine grained measurement of queuing delay. This flag can be enabled via both socket options and control messages.

SOF_TIMESTAMPING_TX_ACK:
Request tx timestamps when all data in the send buffer has been acknowledged. This only makes sense for reliable protocols. It is currently only implemented for TCP. For that protocol, it may over-report measurement, because the timestamp is generated when all data up to and including the buffer at send() was acknowledged: the cumulative acknowledgment. The mechanism ignores SACK and FACK. This flag can be enabled via both socket options and control messages.

1.3.2 Timestamp Reporting

The other three bits control which timestamps will be reported in a generated control message. Changes to the bits take immediate effect at the timestamp reporting locations in the stack. Timestamps are only reported for packets that also have the relevant timestamp generation request set.

SOF_TIMESTAMPING_SOFTWARE:
Report any software timestamps when available.

SOF_TIMESTAMPING_SYS_HARDWARE:
This option is deprecated and ignored.

SOF_TIMESTAMPING_RAW_HARDWARE:
Report hardware timestamps as generated by SOF_TIMESTAMPING_TX_HARDWARE when available.

1.3.3 Timestamp Options

The interface supports the options

SOF_TIMESTAMPING_OPT_ID:

Generate a unique identifier along with each packet. A process can have multiple concurrent timestamping requests outstanding. Packets can be reordered in the transmit path, for instance in the packet scheduler. In that case timestamps will be queued onto the error queue out of order from the original send() calls. It is not always possible to uniquely match timestamps to the original send() calls based on timestamp order or payload inspection alone, then.

This option associates each packet at send() with a unique identifier and returns that along with the timestamp. The identifier is derived from a per-socket u32 counter (that wraps). For datagram sockets, the counter increments with each sent packet. For stream sockets, it increments with every byte.

The counter starts at zero. It is initialized the first time that the socket option is enabled. It is reset each time the option is enabled after having been disabled. Resetting the counter does not change the identifiers of existing packets in the system.

This option is implemented only for transmit timestamps. There, the timestamp is always looped along with a struct sock_extended_err. The option modifies field ee_data to pass an id that is unique among all possibly concurrently outstanding timestamp requests for that socket.

SOF_TIMESTAMPING_OPT_CMSG:

Support recv() cmsg for all timestamped packets. Control messages are already supported unconditionally on all packets with receive timestamps and on IPv6 packets with transmit timestamp. This option extends them to IPv4 packets with transmit timestamp. One use case is to correlate packets with their egress device, by enabling socket option IP_PKTINFO simultaneously.

SOF_TIMESTAMPING_OPT_TSONLY:

Applies to transmit timestamps only. Makes the kernel return the timestamp as a cmsg alongside an empty packet, as opposed to alongside the original packet. This reduces the amount of memory charged to the socket's receive budget (SO_RCVBUF) and delivers the timestamp even if sysctl net.core.tstamp_allow_data is 0. This option disables SOF_TIMESTAMPING_OPT_CMSG.

SOF_TIMESTAMPING_OPT_STATS:

Optional stats that are obtained along with the transmit timestamps. It must be used together with SOF_TIMESTAMPING_OPT_TSONLY. When the transmit timestamp is available, the stats are available in a separate

control message of type SCM_TIMESTAMPING_OPT_STATS, as a list of TLVs (struct nlattr) of types. These stats allow the application to associate various transport layer stats with the transmit timestamps, such as how long a certain block of data was limited by peer's receiver window.

SOF_TIMESTAMPING_OPT_PKTINFO:

Enable the SCM_TIMESTAMPING_PKTINFO control message for incoming packets with hardware timestamps. The message contains struct scm_ts_pktinfo, which supplies the index of the real interface which received the packet and its length at layer 2. A valid (non-zero) interface index will be returned only if CONFIG_NET_RX_BUSY_POLL is enabled and the driver is using NAPI. The struct contains also two other fields, but they are reserved and undefined.

SOF_TIMESTAMPING_OPT_TX_SWHW:

Request both hardware and software timestamps for outgoing packets when SOF_TIMESTAMPING_TX_HARDWARE and SOF_TIMESTAMPING_TX_SOFTWARE are enabled at the same time. If both timestamps are generated, two separate messages will be looped to the socket's error queue, each containing just one timestamp.

New applications are encouraged to pass SOF_TIMESTAMPING_OPT_ID to disambiguate timestamps and SOF_TIMESTAMPING_OPT_TSONLY to operate regardless of the setting of sysctl net.core.tstamp_allow_data.

An exception is when a process needs additional cmsg data, for instance SOL_IP/IP_PKTINFO to detect the egress network interface. Then pass option SOF_TIMESTAMPING_OPT_CMSG. This option depends on having access to the contents of the original packet, so cannot be combined with SOF_TIMESTAMPING_OPT_TSONLY.

1.3.4. Enabling timestamps via control messages

In addition to socket options, timestamp generation can be requested per write via cmsg, only for SOF_TIMESTAMPING_TX_* (see Section 1.3.1). Using this feature, applications can sample timestamps per sendmsg() without paying the overhead of enabling and disabling timestamps via setsockopt:

```
struct msghdr *msg;
...
cmsg                        = CMSG_FIRSTHDR(msg);
cmsg->cmsg_level            = SOL_SOCKET;
cmsg->cmsg_type             = SO_TIMESTAMPING;
cmsg->cmsg_len              = CMSG_LEN(sizeof(__u32));
*((__u32 *) CMSG_DATA(cmsg)) = SOF_TIMESTAMPING_TX_SCHED |
                               SOF_TIMESTAMPING_TX_SOFTWARE |
                               SOF_TIMESTAMPING_TX_ACK;
err = sendmsg(fd, msg, 0);
```

The SOF_TIMESTAMPING_TX_* flags set via cmsg will override the SOF_TIMESTAMPING_TX_* flags set via setsockopt.

Moreover, applications must still enable timestamp reporting via setsockopt to receive timestamps:

```
__u32 val = SOF_TIMESTAMPING_SOFTWARE |
            SOF_TIMESTAMPING_OPT_ID /* or any other flag */;
err = setsockopt(fd, SOL_SOCKET, SO_TIMESTAMPING, &val, sizeof(val));
```

1.4 Bytestream Timestamps

The SO_TIMESTAMPING interface supports timestamping of bytes in a bytestream. Each request is interpreted as a request for when the entire contents of the buffer has passed a timestamping point. That is, for streams option SOF_TIMESTAMPING_TX_SOFTWARE will record when all bytes have reached the device driver, regardless of how many packets the data has been converted into.

In general, bytestreams have no natural delimiters and therefore correlating a timestamp with data is non-trivial. A range of bytes may be split across segments, any segments may be merged (possibly coalescing sections of previously segmented buffers associated with independent send() calls). Segments can be reordered and the same byte range can coexist in multiple segments for protocols that implement retransmissions.

It is essential that all timestamps implement the same semantics, regardless of these possible transformations, as otherwise they are incomparable. Handling "rare" corner cases differently from the simple case (a 1:1 mapping from buffer to skb) is insufficient because performance debugging often needs to focus on such outliers.

In practice, timestamps can be correlated with segments of a bytestream consistently, if both semantics of the timestamp and the timing of measurement are chosen correctly. This challenge is no different from deciding on a strategy for IP fragmentation. There, the definition is that only the first fragment is timestamped. For bytestreams, we chose that a timestamp is generated only when all bytes have passed a point. SOF_TIMESTAMPING_TX_ACK as defined is easy to implement and reason about. An implementation that has to take into account SACK would be more complex due to possible transmission holes and out of order arrival.

On the host, TCP can also break the simple 1:1 mapping from buffer to skbuff as a result of Nagle, cork, autocork, segmentation and GSO. The implementation ensures correctness in all cases by tracking the individual last byte passed to send(), even if it is no longer the last byte after an skbuff extend or merge operation. It stores the relevant sequence number in skb_shinfo(skb)->tskey. Because an skbuff has only one such field, only one timestamp can be generated.

In rare cases, a timestamp request can be missed if two requests are collapsed onto the same skb. A process can detect this situation

by enabling SOF_TIMESTAMPING_OPT_ID and comparing the byte offset at send time with the value returned for each timestamp. It can prevent the situation by always flushing the TCP stack in between requests, for instance by enabling TCP_NODELAY and disabling TCP_CORK and autocork.

These precautions ensure that the timestamp is generated only when all bytes have passed a timestamp point, assuming that the network stack itself does not reorder the segments. The stack indeed tries to avoid reordering. The one exception is under administrator control: it is possible to construct a packet scheduler configuration that delays segments from the same stream differently. Such a setup would be unusual.

2 Data Interfaces

Timestamps are read using the ancillary data feature of recvmsg(). See *man 3 cmsg* for details of this interface. The socket manual page (*man 7 socket*) describes how timestamps generated with SO_TIMESTAMP and SO_TIMESTAMPNS records can be retrieved.

2.1 SCM_TIMESTAMPING records

These timestamps are returned in a control message with cmsg_level SOL_SOCKET, cmsg_type SCM_TIMESTAMPING, and payload of type

For SO_TIMESTAMPING_OLD:

```
struct scm_timestamping {
        struct timespec ts[3];
};
```

For SO_TIMESTAMPING_NEW:

```
struct scm_timestamping64 {
        struct __kernel_timespec ts[3];
```

Always use SO_TIMESTAMPING_NEW timestamp to always get timestamp in struct scm_timestamping64 format.

SO_TIMESTAMPING_OLD returns incorrect timestamps after the year 2038 on 32 bit machines.

The structure can return up to three timestamps. This is a legacy feature. At least one field is non-zero at any time. Most timestamps are passed in ts[0]. Hardware timestamps are passed in ts[2].

ts[1] used to hold hardware timestamps converted to system time. Instead, expose the hardware clock device on the NIC directly as a HW PTP clock source, to allow time conversion in userspace and optionally synchronize system time with a userspace PTP stack such as linuxptp. For the PTP clock API, see Documentation/driver-api/ptp.rst.

Note that if the SO_TIMESTAMP or SO_TIMESTAMPNS option is enabled together with SO_TIMESTAMPING using SOF_TIMESTAMPING_SOFTWARE, a false software timestamp will be generated in the recvmsg() call and passed in ts[0] when a real software timestamp is missing. This happens also on hardware transmit timestamps.

2.1.1 Transmit timestamps with MSG_ERRQUEUE

For transmit timestamps the outgoing packet is looped back to the socket's error queue with the send timestamp(s) attached. A process receives the timestamps by calling recvmsg() with flag MSG_ERRQUEUE set and with a msg_control buffer sufficiently large to receive the relevant metadata structures. The recvmsg call returns the original outgoing data packet with two ancillary messages attached.

A message of cm_level SOL_IP(V6) and cm_type IP(V6)_RECVERR embeds a struct sock_extended_err. This defines the error type. For timestamps, the ee_errno field is ENOMSG. The other ancillary message will have cm_level SOL_SOCKET and cm_type SCM_TIMESTAMPING. This embeds the struct scm_timestamping.

2.1.1.2 Timestamp types

The semantics of the three struct timespec are defined by field ee_info in the extended error structure. It contains a value of type SCM_TSTAMP_* to define the actual timestamp passed in scm_timestamping.

The SCM_TSTAMP_* types are 1:1 matches to the SOF_TIMESTAMPING_* control fields discussed previously, with one exception. For legacy reasons, SCM_TSTAMP_SND is equal to zero and can be set for both SOF_TIMESTAMPING_TX_HARDWARE and SOF_TIMESTAMPING_TX_SOFTWARE. It is the first if ts[2] is non-zero, the second otherwise, in which case the timestamp is stored in ts[0].

2.1.1.3 Fragmentation

Fragmentation of outgoing datagrams is rare, but is possible, e.g., by explicitly disabling PMTU discovery. If an outgoing packet is fragmented, then only the first fragment is timestamped and returned to the sending socket.

2.1.1.4 Packet Payload

The calling application is often not interested in receiving the whole packet payload that it passed to the stack originally: the socket

error queue mechanism is just a method to piggyback the timestamp on. In this case, the application can choose to read datagrams with a smaller buffer, possibly even of length 0. The payload is truncated accordingly. Until the process calls recvmsg() on the error queue, however, the full packet is queued, taking up budget from SO_RCVBUF.

2.1.1.5 Blocking Read

Reading from the error queue is always a non-blocking operation. To block waiting on a timestamp, use poll or select. poll() will return POLLERR in pollfd.revents if any data is ready on the error queue. There is no need to pass this flag in pollfd.events. This flag is ignored on request. See also *man 2 poll*.

2.1.2 Receive timestamps

On reception, there is no reason to read from the socket error queue. The SCM_TIMESTAMPING ancillary data is sent along with the packet data on a normal recvmsg(). Since this is not a socket error, it is not accompanied by a message SOL_IP(V6)/IP(V6)_RECVERROR. In this case, the meaning of the three fields in struct scm_timestamping is implicitly defined. ts[0] holds a software timestamp if set, ts[1] is again deprecated and ts[2] holds a hardware timestamp if set.

3. Hardware Timestamping configuration: SIOCSHWTSTAMP and SIOCGHWTSTAMP

Hardware time stamping must also be initialized for each device driver that is expected to do hardware time stamping. The parameter is defined in include/uapi/linux/net_tstamp.h as:

```
struct hwtstamp_config {
        int flags;      /* no flags defined right now, must be zero */
        int tx_type;    /* HWTSTAMP_TX_* */
        int rx_filter;  /* HWTSTAMP_FILTER_* */
};
```

Desired behavior is passed into the kernel and to a specific device by calling ioctl(SIOCSHWTSTAMP) with a pointer to a struct ifreq whose ifr_data points to a struct hwtstamp_config. The tx_type and rx_filter are hints to the driver what it is expected to do. If the requested fine-grained filtering for incoming packets is not supported, the driver may time stamp more than just the requested types of packets.

Drivers are free to use a more permissive configuration than the requested configuration. It is expected that drivers should only implement directly the most generic mode that can be supported. For example if the hardware can support HWTSTAMP_FILTER_PTP_V2_EVENT, then it should generally always upscale HWTSTAMP_FILTER_PTP_V2_L2_SYNC, and so forth, as HWTSTAMP_FILTER_PTP_V2_EVENT is more generic (and more useful to applications).

A driver which supports hardware time stamping shall update the struct with the actual, possibly more permissive configuration. If the requested packets cannot be time stamped, then nothing should be changed and ERANGE shall be returned (in contrast to EINVAL, which indicates that SIOCSHWTSTAMP is not supported at all).

Only a processes with admin rights may change the configuration. User space is responsible to ensure that multiple processes don't interfere with each other and that the settings are reset.

Any process can read the actual configuration by passing this structure to ioctl(SIOCGHWTSTAMP) in the same way. However, this has not been implemented in all drivers.

```
/* possible values for hwtstamp_config->tx_type */
enum {
        /*
        * no outgoing packet will need hardware time stamping;
        * should a packet arrive which asks for it, no hardware
        * time stamping will be done
        */
        HWTSTAMP_TX_OFF,

        /*
        * enables hardware time stamping for outgoing packets;
        * the sender of the packet decides which are to be
        * time stamped by setting SOF_TIMESTAMPING_TX_SOFTWARE
        * before sending the packet
        */
        HWTSTAMP_TX_ON,
};

/* possible values for hwtstamp_config->rx_filter */
enum {
        /* time stamp no incoming packet at all */
        HWTSTAMP_FILTER_NONE,

        /* time stamp any incoming packet */
        HWTSTAMP_FILTER_ALL,
```

```
        /* return value: time stamp all packets requested plus some others */
        HWTSTAMP_FILTER_SOME,

        /* PTP v1, UDP, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V1_L4_EVENT,

        /* for the complete list of values, please check
        * the include file include/uapi/linux/net_tstamp.h
        */
};
```

3.1 Hardware Timestamping Implementation: Device Drivers

A driver which supports hardware time stamping must support the SIOCSHWTSTAMP ioctl and update the supplied struct hwtstamp_config with the actual values as described in the section on SIOCSHWTSTAMP. It should also support SIOCGHWTSTAMP.

Time stamps for received packets must be stored in the skb. To get a pointer to the shared time stamp structure of the skb call skb_hwtstamps(). Then set the time stamps in the structure:

```
struct skb_shared_hwtstamps {
        /* hardware time stamp transformed into duration
        * since arbitrary point in time
        */
        ktime_t     hwtstamp;
};
```

Time stamps for outgoing packets are to be generated as follows:

- In hard_start_xmit(), check if (skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP) is set no-zero. If yes, then the driver is expected to do hardware time stamping.
- If this is possible for the skb and requested, then declare that the driver is doing the time stamping by setting the flag SKBTX_IN_PROGRESS in skb_shinfo(skb)->tx_flags , e.g. with:

  ```
  skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS;
  ```

  You might want to keep a pointer to the associated skb for the next step and not free the skb. A driver not supporting hardware time stamping doesn't do that. A driver must never touch sk_buff::tstamp! It is used to store software generated time stamps by the network subsystem.
- Driver should call skb_tx_timestamp() as close to passing sk_buff to hardware as possible. skb_tx_timestamp() provides a software time stamp if requested and hardware timestamping is not possible (SKBTX_IN_PROGRESS not set).
- As soon as the driver has sent the packet and/or obtained a hardware time stamp for it, it passes the time stamp back by calling skb_tstamp_tx() with the original skb, the raw hardware time stamp. skb_tstamp_tx() clones the original skb and adds the timestamps, therefore the original skb has to be freed now. If obtaining the hardware time stamp somehow fails, then the driver should not fall back to software time stamping. The rationale is that this would occur at a later time in the processing pipeline than other software time stamping and therefore could lead to unexpected deltas between time stamps.

3.2 Special considerations for stacked PTP Hardware Clocks

There are situations when there may be more than one PHC (PTP Hardware Clock) in the data path of a packet. The kernel has no explicit mechanism to allow the user to select which PHC to use for timestamping Ethernet frames. Instead, the assumption is that the outermost PHC is always the most preferable, and that kernel drivers collaborate towards achieving that goal. Currently there are 3 cases of stacked PHCs, detailed below:

3.2.1 DSA (Distributed Switch Architecture) switches

These are Ethernet switches which have one of their ports connected to an (otherwise completely unaware) host Ethernet interface, and perform the role of a port multiplier with optional forwarding acceleration features. Each DSA switch port is visible to the user as a standalone (virtual) network interface, and its network I/O is performed, under the hood, indirectly through the host interface (redirecting to the host port on TX, and intercepting frames on RX).

When a DSA switch is attached to a host port, PTP synchronization has to suffer, since the switch's variable queuing delay introduces a path delay jitter between the host port and its PTP partner. For this reason, some DSA switches include a timestamping clock of their own, and have the ability to perform network timestamping on their own MAC, such that path delays only measure wire and PHY propagation latencies. Timestamping DSA switches are supported in Linux and expose the same ABI as any other network interface (save for the fact that the DSA interfaces are in fact virtual in terms of network I/O, they do have their own PHC). It is typical, but not mandatory, for all interfaces of a DSA switch to share the same PHC.

By design, PTP timestamping with a DSA switch does not need any special handling in the driver for the host port it is attached to. However, when the host port also supports PTP timestamping, DSA will take care of intercepting the `.ndo_eth_ioctl` calls towards the host port, and block attempts to enable hardware timestamping on it. This is because the SO_TIMESTAMPING API does not allow the delivery of multiple hardware timestamps for the same packet, so anybody else except for the DSA switch port

must be prevented from doing so.

In the generic layer, DSA provides the following infrastructure for PTP timestamping:

- `.port_txtstamp()`: a hook called prior to the transmission of packets with a hardware TX timestamping request from user space. This is required for two-step timestamping, since the hardware timestamp becomes available after the actual MAC transmission, so the driver must be prepared to correlate the timestamp with the original packet so that it can re-enqueue the packet back into the socket's error queue. To save the packet for when the timestamp becomes available, the driver can call `skb_clone_sk`, save the clone pointer in skb->cb and enqueue a tx skb queue. Typically, a switch will have a PTP TX timestamp register (or sometimes a FIFO) where the timestamp becomes available. In case of a FIFO, the hardware might store key-value pairs of PTP sequence ID/message type/domain number and the actual timestamp. To perform the correlation correctly between the packets in a queue waiting for timestamping and the actual timestamps, drivers can use a BPF classifier (`ptp_classify_raw`) to identify the PTP transport type, and `ptp_parse_header` to interpret the PTP header fields. There may be an IRQ that is raised upon this timestamp's availability, or the driver might have to poll after invoking `dev_queue_xmit()` towards the host interface. One-step TX timestamping do not require packet cloning, since there is no follow-up message required by the PTP protocol (because the TX timestamp is embedded into the packet by the MAC), and therefore user space does not expect the packet annotated with the TX timestamp to be re-enqueued into its socket's error queue.
- `.port_rxtstamp()`: On RX, the BPF classifier is run by DSA to identify PTP event messages (any other packets, including PTP general messages, are not timestamped). The original (and only) timestampable skb is provided to the driver, for it to annotate it with a timestamp, if that is immediately available, or defer to later. On reception, timestamps might either be available in-band (through metadata in the DSA header, or attached in other ways to the packet), or out-of-band (through another RX timestamping FIFO). Deferral on RX is typically necessary when retrieving the timestamp needs a sleepable context. In that case, it is the responsibility of the DSA driver to call `netif_rx()` on the freshly timestamped skb.

3.2.2 Ethernet PHYs

These are devices that typically fulfill a Layer 1 role in the network stack, hence they do not have a representation in terms of a network interface as DSA switches do. However, PHYs may be able to detect and timestamp PTP packets, for performance reasons: timestamps taken as close as possible to the wire have the potential to yield a more stable and precise synchronization.

A PHY driver that supports PTP timestamping must create a `struct mii_timestamper` and add a pointer to it in `phydev->mii_ts`. The presence of this pointer will be checked by the networking stack.

Since PHYs do not have network interface representations, the timestamping and ethtool ioctl operations for them need to be mediated by their respective MAC driver. Therefore, as opposed to DSA switches, modifications need to be done to each individual MAC driver for PHY timestamping support. This entails:

- Checking, in `.ndo_eth_ioctl`, whether `phy_has_hwtstamp(netdev->phydev)` is true or not. If it is, then the MAC driver should not process this request but instead pass it on to the PHY using `phy_mii_ioctl()`.
- On RX, special intervention may or may not be needed, depending on the function used to deliver skb's up the network stack. In the case of plain `netif_rx()` and similar, MAC drivers must check whether `skb_defer_rx_timestamp(skb)` is necessary or not - and if it is, don't call `netif_rx()` at all. If `CONFIG_NETWORK_PHY_TIMESTAMPING` is enabled, and `skb->dev->phydev->mii_ts` exists, its `.rxtstamp()` hook will be called now, to determine, using logic very similar to DSA, whether deferral for RX timestamping is necessary. Again like DSA, it becomes the responsibility of the PHY driver to send the packet up the stack when the timestamp is available.

  For other skb receive functions, such as `napi_gro_receive` and `netif_receive_skb`, the stack automatically checks whether `skb_defer_rx_timestamp()` is necessary, so this check is not needed inside the driver.
- On TX, again, special intervention might or might not be needed. The function that calls the `mii_ts->txtstamp()` hook is named `skb_clone_tx_timestamp()`. This function can either be called directly (case in which explicit MAC driver support is indeed needed), but the function also piggybacks from the `skb_tx_timestamp()` call, which many MAC drivers already perform for software timestamping purposes. Therefore, if a MAC supports software timestamping, it does not need to do anything further at this stage.

3.2.3 MII bus snooping devices

These perform the same role as timestamping Ethernet PHYs, save for the fact that they are discrete devices and can therefore be used in conjunction with any PHY even if it doesn't support timestamping. In Linux, they are discoverable and attachable to a `struct phy_device` through Device Tree, and for the rest, they use the same mii_ts infrastructure as those. See Documentation/devicetree/bindings/ptp/timestamper.txt for more details.

3.2.4 Other caveats for MAC drivers

Stacked PHCs, especially DSA (but not only) - since that doesn't require any modification to MAC drivers, so it is more difficult to ensure correctness of all possible code paths - is that they uncover bugs which were impossible to trigger before the existence of stacked PTP clocks. One example has to do with this line of code, already presented earlier:

```
skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS;
```

Any TX timestamping logic, be it a plain MAC driver, a DSA switch driver, a PHY driver or a MII bus snooping device driver, should set this flag. But a MAC driver that is unaware of PHC stacking might get tripped up by somebody other than itself setting this flag, and deliver a duplicate timestamp. For example, a typical driver design for TX timestamping might be to split the transmission part into 2 portions:

1. "TX": checks whether PTP timestamping has been previously enabled through the `.ndo_eth_ioctl` ("`priv->hwtstamp_tx_enabled == true`") and the current skb requires a TX timestamp ("`skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP`"). If this is true, it sets the "`skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS`" flag. Note: as described above, in the case of a stacked PHC system, this condition should never trigger, as this MAC is certainly not the outermost PHC. But this is not where the typical issue is. Transmission proceeds with this packet.
2. "TX confirmation": Transmission has finished. The driver checks whether it is necessary to collect any TX timestamp for it. Here is where the typical issues are: the MAC driver takes a shortcut and only checks whether "`skb_shinfo(skb)->tx_flags & SKBTX_IN_PROGRESS`" was set. With a stacked PHC system, this is incorrect because this MAC driver is not the only entity in the TX data path who could have enabled SKBTX_IN_PROGRESS in the first place.

The correct solution for this problem is for MAC drivers to have a compound check in their "TX confirmation" portion, not only for "`skb_shinfo(skb)->tx_flags & SKBTX_IN_PROGRESS`", but also for "`priv->hwtstamp_tx_enabled == true`". Because the rest of the system ensures that PTP timestamping is not enabled for anything other than the outermost PHC, this enhanced check will avoid delivering a duplicated TX timestamp to user space.