WR Switch Kernel Network Driver (wrnet) informal specification version 0.1

Tomasz Włostowski/CERN BE-Co-HT

Table of Contents

1.Introduction....2
2.wrnet driver features....2
  2.1.Endpoints....2
  2.2.The NIC....3
  2.3.PPS Generator....5
  2.4.TX Timestamping unit (TXTSU)....6
3.PTP-related features....6
  3.1.Generating TX timestamps....6
  3.2.Generating RX timestamps....7
  3.3.Timestamp postprocessing....7
  3.4.DMTD readout and calibration pattern controls....7

1. Introduction

The White Rabbit Switch Kernel Network Driver (called wrnet) implements a hybrid Linux device driver, providing multiple network interfaces with accurate PTP timestamping. The driver shall support the set of HDL peripherals depicted in **fig. 1**. The blocks which are drawn using dashed line are irrelevant for the wrnet driver.

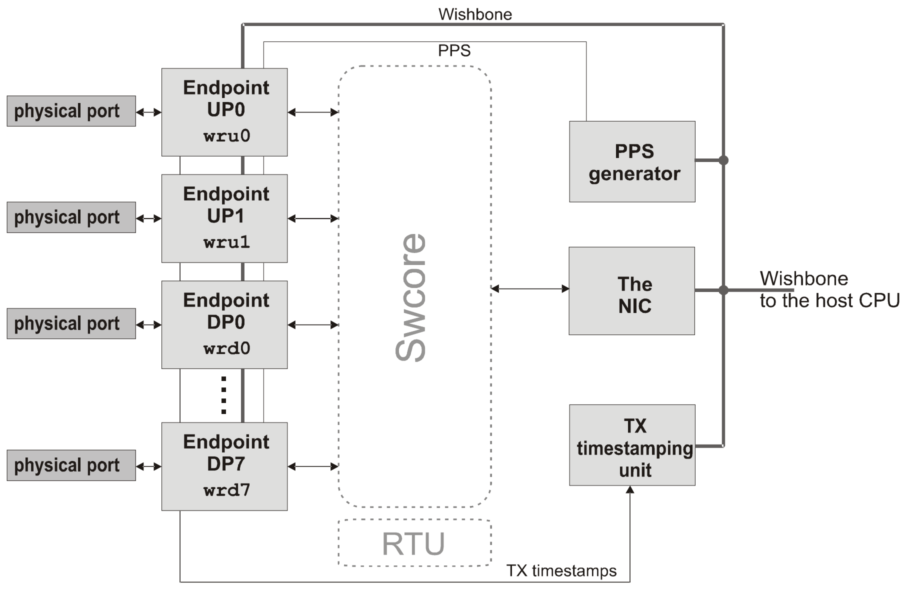

Figure 1. HDL modules supported by the wrnet driver (grey boxes only)

2. wrnet driver features

The wrnet driver shall provide the following features:

- Support for a configurable number of WR endpoints connected to a single NIC via the Swcore, each of the endpoints will be connected to a physical port (SFP or copper).
- Data coming from/to all the endpoints is passed via the Swcore to/from the NIC. The NIC implements the CPU packet queues, shared between the endpoints.
- Each Endpoint shall be identified by the kernel as a separate network interface (`wrux` for the uplinks, `wrdx` for the downlinks).
- The driver shall provide the WR PTP compatible timestamps.
- The driver shall provide calibration and DMTD readout ioctls.
- The driver shall register an adjustable timesource in the kernel which provides the PTP time.

2.1. Endpoints

The WR Endpoint comprises an WR-compatible Ethernet MAC and 1000Base-X or GMII PCS (Physical Coding Sublayer). A number of WR endpoints + Swcore + WR NIC make a multi-port WR network adapter.

The wrnet driver shall implement the following endpoint-related features:

- register initialization (see `void ep_enable()` in wr_minic driver),
- assigning an unique MAC address. The MAC address will be provided by the wrsw_hal daemon, based on the switch configuration. The interface can't be brought up (`ifconfig wrxx up`) without assigning it a MAC address first (`ifconfig wrxx hw ether xx:yy:zz....`)
- assigning an unique port ID (see the EP_ECR register documentation) which will be used by the NIC to recognize the source/destination port.
- DMTD phase readouts via a private ioctl (see the wr_minic driver)
- Calbration pattern TX/RX control via private ioctls (see the wr_minic driver)
- 802.3x autonegotiation (probably via kernel phylib) and initialization of the Endpoints' flow control logic.
- accurate PTP timestamps (using dual-edge TS counters) passed to the userland in the kernel's standard way (see wr_minic driver for the working implementation reference)
- synchronization of endpoints' local TS counters with the global PPS generator upon a request from the userspace.
- reading statistic counters (see the RMON section of the Endpoint documentation)

2.2. The NIC

The NIC is a packet queue capable of sending/receiving packets to/from multiple WR Endpoints via the WR Swcore. It has two descriptor tables (one for TX, one for RX packets) and a shared memory block for packet storage (currently without DMA access).

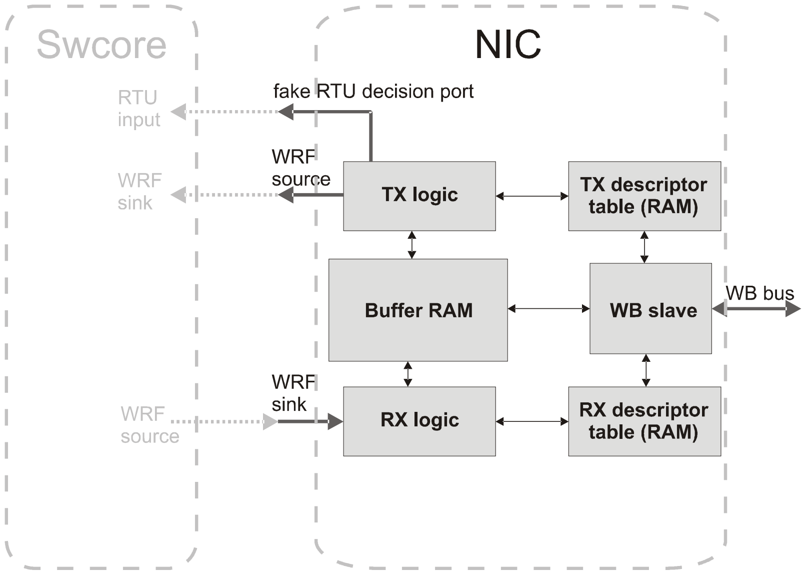

Figure 2. NIC block diagram and connections

The NIC has a pool of N TX descriptors and N RX descriptors. Each of the descriptors contains the base address and size of its assigned area in the shared packet buffer memory, READY/EMPTY flag, error indication flag and some timestamping-related paramaters.

Packet transmission

Below is the algorithm for transmission of a packet coming from a certain interface (`wrXX`):

1. Enable transmission in the NIC Control Register.
2. Reserve necessary space in the NIC packet RAM and store the packet contents in the reserved block. These include the packet header: valid destination MAC, empty source MAC (it will be embedded by the Endpoint transmitting the packet, ethertype/802.1q header and payload **without the CRC field** (generated in the Endpoint).
3. Fill the DPM field in the descriptor header, indicating to which physical port the packet should go. For example, if the userland sends a packet to interface `wrd1`, which is connected to Swcore port 3, we shall write `(1<<3) = 0x8` into DPM register.
4. If the packet has to be timestamped, set the TS_E bit and provide a unique value of TS_ID in the descriptor header.
5. Set the READY bit to 1. The NIC start transmitting the contents of the descriptor. If there are more TX descriptors with READY=1, they will be transmitted subsequently.
6. If an error occurred, the NIC:

   - stops transmitting the packet

   - disables transmission (TX_E bit in CR reg goes to 0)

   - clears READY and sets ERROR bit in the failed descriptor

   - triggers TX error interrupt. The driver can then retransmit or drop the packet. TX Error interrupt is generated by ORing ERROR bits in all TX descriptors and it's level-active.

   The driver should clear the encountered ERROR flag and eventually retransmit or drop the packet.
7. When there are no remaining ready TX descriptors, the NIC triggers a TX_DONE interrupt.
8. If the packets were meant to be timestamped, the TX timestamps can be retreived from the TX timestamping unit (See Timestamping section).

WARNING: The NIC is treated like a virtual network card connected to a virtual switch port – that implies that packet loss may occur between the NIC and its destination endpoint. You can never assume that the physical transmission was successful – even if the packet appears to be have been sent successfully by the NIC (ERROR = 0 in the descriptor header).

Packet reception

The reception procedure goes as follows:

1. Enable reception by setting a bit in CR register.
2. Initialize the RX descriptors. Reserve memory in the packet RAM for each of them.
3. Mark the descriptors as EMPTY.
4. A frame arrives and it's written into the first available RX descriptor. Immediately after enabling reception, it's descriptor 0.
5. When a packet has been successfully received (and written by the NIC to the first available RX descriptor), the descriptor's EMPTY flag will be set to 0 and you will get an interrupt. The ISR should receive all allocated RX descriptors which aren't empty. The driver must check the source port ID in the descriptor header and put the received packet in the queue belonging to

corresponding network interface.

6. If the received packet has got a timestamp (GOT_TS = 1), it should be read and passed to the userspace.
7. If an error occurred, RX_ERROR interrupt will be triggered. The ISR shall clear the faulty descriptor and continue receiving packets.

2.3. *PPS Generator*

The PPS generator produces the system-wide PPS signal and UTC time. Since the PTP timestamping must be done using identical time base in every endpoint (which requires a pretty tight cooperation between the PPSG and the endpoints), the PPSG support should be integrated within the wrnet driver.

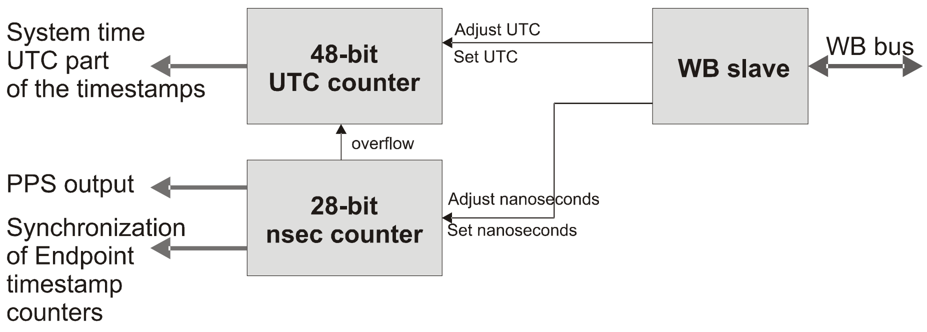

Figure 3. PPS generator diagram

PPSG produces the following:

– System time (a second counter)
– PPS pulses of programmable width
– sync signal for synchronization of the TS units in the Endpoints (so every endpoint uses the same time base for producing the timestamps).

PPSG should be seen by the OS as a real-time clock and provide the two following operations:

– setting the seconds and nanoseconds counters
– adjusting the seconds and nanoseconds counters by adding/subtracting a specified value (this operation must be atomic)

WARNING: Adjustment of PPSG counters causes the adjustment of the TS counters in all the Endpoints via the synchronization mechanism. Therefore, all timestamps taken during the PPSG adjust operation **must be discarded**.

NOTE: Please don't use `adjtime/adjtimex` for adjusting the counters. I would prefer to have a simple ioctl for that to be sure that no application except for the PTP daemon will be messing around with the PPSG.

NOTE: The PPSG driver is already present in the `libswitchhw`. You can use this as a reference.

2.4. TX Timestamping unit (TXTSU)

The role of the TXTSU is to receive the TX timestamps from all Endpoints and provide a FIFO interface for the wrnet driver for retreiving them. It's a simple 32-entry FIFO in which each entry contains:

- the value of the TX timestamp
- the ID of the timestamped frame
- the ID of the Endpoint which produced the timestamp.

TXTSU produces a level-active interrupt when the FIFO is not empty. This interrupt must be handled by the driver during the timestamping process.

3. PTP-related features

3.1. Generating TX timestamps

The process of generating TX timestamps is depicted on **figure 4**.

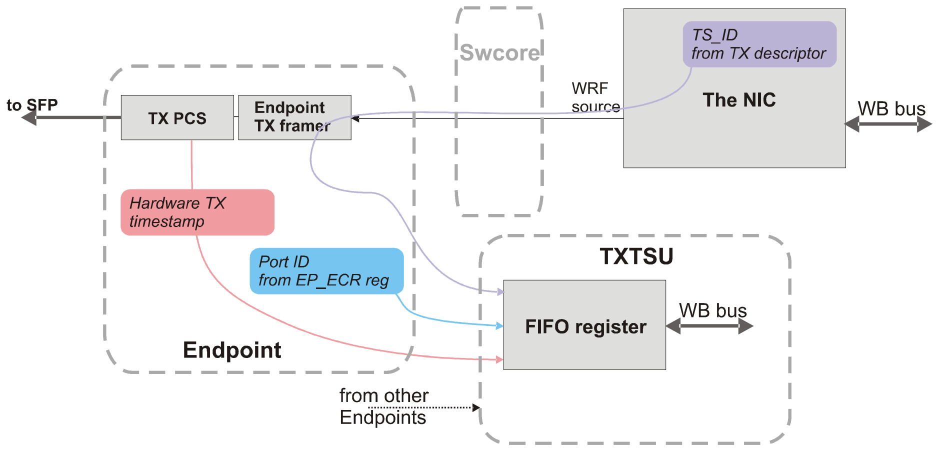

Figure 4. TX timestamping data flow.

Below is an explanation of the TX timestamping process:

- the driver assigns a 16-bit timestamp ID for the packet and enables timestamping in the TX descriptor header.
- the NIC will send the packet with associated TS_ID in the TX OOB block
- the packet reaches the Endpoint. The Endpoint detects the TX OOB block and generates the TX timestamp
- the TX timestamp value, TS_ID value and the identifier of the endpoint which produced the timestamp are passed together to the TXTSU in which they are stored in a FIFO register
- the driver gets the TXTSU interrupt and reads out the timestamp value from the TXTSU FIFO. Because we have the frame and endpoint ID, we can assign the retreived timestamp to the correct frame.

3.2. Generating RX timestamps

RX timestamps are automatically generated and appended to WRF transfers when RX timestamping is enabled in the Endpoint. The value of RX timestamp can be simply read from the RX descriptor.

3.3. Timestamp postprocessing

The raw timestamp values produced by the Endpoints need to be postprocessed. Postprocessing includes:

- merging the nanoseconds part (produced by the Endpoint) with the seconds part (read from the PPSG)
- checking if the falling edge counter isn't ahead of the rising edge counter to detect metastabilities
- converting the timestamps into `timespec` format. The MSB of the `tv_sec` value should indicate if the counters are not equal

NOTE: See the `minic_rx_frame()` function in the miNIC driver for the implementation of timestamp postprocessing. It will be identical in the wrnet driver.

3.4. DMTD readout and calibration pattern controls

The driver must provide two following private ioctls for handling WR calibration and DMTD readout:

- `PRIV_IOCGCALIBRATE`: Calibration pattern TX/RX control,
- `PRIV_IOCGGETPHASE`: DMTD phase shift readout.

The implementation of these is already done in the miNIC driver.