Open Source Used In IR1101 IOS-XE 16.9.1

Cisco Systems, Inc.
www.cisco.com

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco website at www.cisco.com/go/offices.

Text Part Number: 78EE117C99-142410159

This document contains licenses and notices for open source software used in this product. With respect to the free/open source software listed in this document, if you have any questions or wish to receive a copy of any source code to which you may be entitled under the applicable free/open source license(s) (such as the GNU Lesser/General Public License), please contact us at external-opensource-requests@cisco.com.

In your requests please include the following reference number 78EE117C99-142410159

Contents

1.1 netmap-for-armada 17.02.0

1.1.1 Available under license :

```
/*
* Copyright (C) 2012-2014 Luigi Rizzo. All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*/
```

```
/*
* $Id: if_igb_netmap.h 10878 2012-04-12 22:28:48Z luigi $
*
* netmap support for: igb (linux version)
* For details on netmap support please see ixgbe_netmap.h
*/


#include <bsd_glue.h>
#include <net/netmap.h>
#include <netmap/netmap_kern.h>

#define SOFTC_T igb_adapter

#define igb_driver_name netmap_igb_driver_name
char netmap_igb_driver_name[] = "igb" NETMAP_LINUX_DRIVER_SUFFIX;

/*
* Adapt to different versions of the driver.
* E1000_TX_DESC_ADV etc. have dropped the _ADV suffix at some point.
* Also the first argument is now a pointer not the object.
*/
#ifndef E1000_TX_DESC_ADV
#define E1000_TX_DESC_ADV(_r, _i) IGB_TX_DESC(&(_r), _i)
#define E1000_RX_DESC_ADV(_r, _i) IGB_RX_DESC(&(_r), _i)
#define READ_TDH(_txr)   ({struct e1000_hw *hw = &adapter->hw;rd32(E1000_TDH((_txr)->reg_idx));} )
#else /* up to 3.2, approximately */
#define igb_tx_buffer   igb_buffer
#define tx_buffer_info   buffer_info
#define igb_rx_buffer   igb_buffer
#define rx_buffer_info   buffer_info
#define READ_TDH(_txr)   readl((_txr)->head)
#endif


/*
* Register/unregister. We are already under netmap lock.
* Only called on the first register or the last unregister.
*/
static int
igb_netmap_reg(struct netmap_adapter *na, int onoff)
{
 struct ifnet *ifp = na->ifp;
 struct SOFTC_T *adapter = netdev_priv(ifp);

 /* protect against other reinit */
 while (test_and_set_bit(__IGB_RESETTING, &adapter->state))
  usleep_range(1000, 2000);
```

```
 if (netif_running(adapter->netdev))
  igb_down(adapter);

 /* enable or disable flags and callbacks in na and ifp */
 if (onoff) {
  nm_set_native_flags(na);
 } else {
  nm_clear_native_flags(na);
 }
 if (netif_running(adapter->netdev))
  igb_up(adapter);
 else
  igb_reset(adapter); // XXX is it needed ?

 clear_bit(__IGB_RESETTING, &adapter->state);
 return (0);
}


/*
* Reconcile kernel and user view of the transmit ring.
*/
static int
igb_netmap_txsync(struct netmap_kring *kring, int flags)
{
 struct netmap_adapter *na = kring->na;
 struct ifnet *ifp = na->ifp;
 struct netmap_ring *ring = kring->ring;
 u_int ring_nr = kring->ring_id;
 u_int nm_i; /* index into the netmap ring */
 u_int nic_i; /* index into the NIC ring */
 u_int n;
 u_int const lim = kring->nkr_num_slots - 1;
 u_int const head = kring->rhead;
 /* generate an interrupt approximately every half ring */
 u_int report_frequency = kring->nkr_num_slots >> 1;

 /* device-specific */
 struct SOFTC_T *adapter = netdev_priv(ifp);
 struct igb_ring* txr = adapter->tx_ring[ring_nr];

 rmb(); // XXX not in ixgbe ?

 /*
  * First part: process new packets to send.
  */
 if (!netif_carrier_ok(ifp)) {
```

```
 goto out;
}

nm_i = kring->nr_hwcur;
if (nm_i != head) { /* we have new packets to send */
 uint32_t olinfo_status=0;

 nic_i = netmap_idx_k2n(kring, nm_i);
 for (n = 0; nm_i != head; n++) {
  struct netmap_slot *slot = &ring->slot[nm_i];
  u_int len = slot->len;
  uint64_t paddr;
  void *addr = PNMB(na, slot, &paddr);

  /* device-specific */
  union e1000_adv_tx_desc *curr =
     E1000_TX_DESC_ADV(*txr, nic_i);
  int flags = (slot->flags & NS_REPORT ||
   nic_i == 0 || nic_i == report_frequency) ?
   E1000_TXD_CMD_RS : 0;

  NM_CHECK_ADDR_LEN(na, addr, len);

  if (slot->flags & NS_BUF_CHANGED) {
   /* buffer has changed, reload map */
   // netmap_reload_map(pdev, DMA_TO_DEVICE, old_paddr, addr);
  }
  slot->flags &= ~(NS_REPORT | NS_BUF_CHANGED);

  /* Fill the slot in the NIC ring. */
  curr->read.buffer_addr = htole64(paddr);
  // XXX check olinfo and cmd_type_len
  curr->read.olinfo_status =
     htole32(olinfo_status |
                        (len<< E1000_ADVTXD_PAYLEN_SHIFT));
  curr->read.cmd_type_len = htole32(len | flags |
   E1000_ADVTXD_DTYP_DATA | E1000_ADVTXD_DCMD_DEXT |
   E1000_ADVTXD_DCMD_IFCS | E1000_TXD_CMD_EOP);
  nm_i = nm_next(nm_i, lim);
  nic_i = nm_next(nic_i, lim);
 }
 kring->nr_hwcur = head;

 wmb(); /* synchronize writes to the NIC ring */

 txr->next_to_use = nic_i; /* XXX what for ? */
 /* (re)start the tx unit up to slot nic_i (excluded) */
 writel(nic_i, txr->tail);
```

```
 mmiowb(); // XXX where do we need this ?
 }

 /*
  * Second part: reclaim buffers for completed transmissions.
  */
 if (flags & NAF_FORCE_RECLAIM || nm_kr_txempty(kring)) {
  /* record completed transmissions using TDH */
  nic_i = READ_TDH(txr);
  if (nic_i >= kring->nkr_num_slots) { /* XXX can it happen ? */
   D("TDH wrap %d", nic_i);
   nic_i -= kring->nkr_num_slots;
  }
  txr->next_to_use = nic_i;
  kring->nr_hwtail = nm_prev(netmap_idx_n2k(kring, nic_i), lim);
 }
out:

 return 0;
}


/*
* Reconcile kernel and user view of the receive ring.
*/
static int
igb_netmap_rxsync(struct netmap_kring *kring, int flags)
{
 struct netmap_adapter *na = kring->na;
 struct ifnet *ifp = na->ifp;
 struct netmap_ring *ring = kring->ring;
 u_int ring_nr = kring->ring_id;
 u_int nm_i; /* index into the netmap ring */
 u_int nic_i; /* index into the NIC ring */
 u_int n;
 u_int const lim = kring->nkr_num_slots - 1;
 u_int const head = kring->rhead;
 int force_update = (flags & NAF_FORCE_READ) || kring->nr_kflags & NKR_PENDINTR;

 /* device-specific */
 struct SOFTC_T *adapter = netdev_priv(ifp);
 struct igb_ring *rxr = adapter->rx_ring[ring_nr];

 if (!netif_carrier_ok(ifp))
  return 0;

 if (head > lim)
  return netmap_ring_reinit(kring);
```

```
rmb();

/*
 * First part: import newly received packets.
 */
if (netmap_no_pendintr || force_update) {
 uint16_t slot_flags = kring->nkr_slot_flags;

 nic_i = rxr->next_to_clean;
 nm_i = netmap_idx_n2k(kring, nic_i);

 for (n = 0; ; n++) {
  union e1000_adv_rx_desc *curr =
    E1000_RX_DESC_ADV(*rxr, nic_i);
  uint32_t staterr = le32toh(curr->wb.upper.status_error);

  if ((staterr & E1000_RXD_STAT_DD) == 0)
   break;
  ring->slot[nm_i].len = le16toh(curr->wb.upper.length);
  ring->slot[nm_i].flags = slot_flags;
  nm_i = nm_next(nm_i, lim);
  nic_i = nm_next(nic_i, lim);
 }
 if (n) { /* update the state variables */
  rxr->next_to_clean = nic_i;
  kring->nr_hwtail = nm_i;
 }
 kring->nr_kflags &= ~NKR_PENDINTR;
}

/*
 * Second part: skip past packets that userspace has released.
 */
nm_i = kring->nr_hwcur;
if (nm_i != head) {
 nic_i = netmap_idx_k2n(kring, nm_i);
 for (n = 0; nm_i != head; n++) {
  struct netmap_slot *slot = &ring->slot[nm_i];
  uint64_t paddr;
  void *addr = PNMB(na, slot, &paddr);
  union e1000_adv_rx_desc *curr = E1000_RX_DESC_ADV(*rxr, nic_i);

  if (addr == NETMAP_BUF_BASE(na)) /* bad buf */
   goto ring_reset;

  if (slot->flags & NS_BUF_CHANGED) {
   // netmap_reload_map(pdev, DMA_FROM_DEVICE, old_paddr, addr);
```

```
   slot->flags &= ~NS_BUF_CHANGED;
  }
  curr->read.pkt_addr = htole64(paddr);
  curr->read.hdr_addr = 0;
  nm_i = nm_next(nm_i, lim);
  nic_i = nm_next(nic_i, lim);
 }
 kring->nr_hwcur = head;
 wmb();
 rxr->next_to_use = nic_i; // XXX not really used
 /*
  * IMPORTANT: we must leave one free slot in the ring,
  * so move nic_i back by one unit
  */
 nic_i = nm_prev(nic_i, lim);
 writel(nic_i, rxr->tail);
 }


 return 0;

ring_reset:
 return netmap_ring_reinit(kring);
}


static int
igb_netmap_configure_tx_ring(struct SOFTC_T *adapter, int ring_nr)
{
 struct ifnet *ifp = adapter->netdev;
 struct netmap_adapter* na = NA(ifp);
 struct netmap_slot* slot;
 struct igb_ring *txr = adapter->tx_ring[ring_nr];
 int i, si;
 void *addr;
 uint64_t paddr;

      slot = netmap_reset(na, NR_TX, ring_nr, 0);
 if (!slot)
  return 0;  // not in netmap native mode
 for (i = 0; i < na->num_tx_desc; i++) {
  union e1000_adv_tx_desc *tx_desc;
  si = netmap_idx_n2k(&na->tx_rings[ring_nr], i);
  addr = PNMB(na, slot + si, &paddr);
  tx_desc = E1000_TX_DESC_ADV(*txr, i);
  tx_desc->read.buffer_addr = htole64(paddr);
  /* actually we don't care to init the rings here */
 }
```

```
 return 1; // success
}


static int
igb_netmap_configure_rx_ring(struct igb_ring *rxr)
{
 struct ifnet *ifp = rxr->netdev;
 struct netmap_adapter* na = NA(ifp);
 int reg_idx = rxr->reg_idx;
 struct netmap_slot* slot;
 u_int i;

 /*
  * XXX watch out, the main driver must not use
  * split headers. The buffer len should be written
  * into wr32(E1000_SRRCTL(reg_idx), srrctl) with options
  * something like
  * srrctl = ALIGN(buffer_len, 1024) >>
  *  E1000_SRRCTL_BSIZEPKT_SHIFT;
  * srrctl |= E1000_SRRCTL_DESCTYPE_ADV_ONEBUF;
  * srrctl |= E1000_SRRCTL_DROP_EN;
  */
      slot = netmap_reset(na, NR_RX, reg_idx, 0);
 if (!slot)
  return 0; // not in native netmap mode

 for (i = 0; i < rxr->count; i++) {
  union e1000_adv_rx_desc *rx_desc;
  uint64_t paddr;
  int si = netmap_idx_n2k(&na->rx_rings[reg_idx], i);

#if 0
  // XXX the skb check can go away
  struct igb_rx_buffer *bi = &rxr->rx_buffer_info[i];
  if (bi->skb)
   D("rx buf %d was set", i);
  bi->skb = NULL; // XXX leak if set
#endif /* useless */

  PNMB(na, slot + si, &paddr);
  rx_desc = E1000_RX_DESC_ADV(*rxr, i);
  rx_desc->read.hdr_addr = 0;
  rx_desc->read.pkt_addr = htole64(paddr);
 }
 rxr->next_to_use = 0;
 /* preserve buffers already made available to clients */
 i = rxr->count - 1 - nm_kr_rxspace(&na->rx_rings[reg_idx]);
```

```
 wmb(); /* Force memory writes to complete */
 ND("%s rxr%d.tail %d", na->name, reg_idx, i);
 writel(i, rxr->tail);
 return 1; // success
}


static void
igb_netmap_attach(struct SOFTC_T *adapter)
{
 struct netmap_adapter na;

 bzero(&na, sizeof(na));

 na.ifp = adapter->netdev;
 na.pdev = &adapter->pdev->dev;
 na.num_tx_desc = adapter->tx_ring_count;
 na.num_rx_desc = adapter->rx_ring_count;
 na.nm_register = igb_netmap_reg;
 na.nm_txsync = igb_netmap_txsync;
 na.nm_rxsync = igb_netmap_rxsync;
 na.num_tx_rings = adapter->num_tx_queues;
 na.num_rx_rings = adapter->num_rx_queues;
 netmap_attach(&na);
}

/* end of file */
 /*
* ********************************************************************************
* Copyright (C) 2016 Marvell International Ltd.
* ********************************************************************************
* This program is free software: you can redistribute it and/or modify it
* under the terms of the GNU General Public License as published by the Free
* Software Foundation, either version 2 of the License, or any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
* ********************************************************************************
*/

/* mv_pp2x_netmap.h */
```

```
#ifndef __MV_PP2X_NETMAP_H__
#define __MV_PP2X_NETMAP_H__

#include <bsd_glue.h>
#include <net/netmap.h>
#include <dev/netmap/netmap_kern.h>

#define SOFTC_T mv_pp2x_port
#define MVPP2_BM_NETMAP_PKT_SIZE 2048
#define MVPP2_NETMAP_MAX_QUEUES_NUM (MVPP2_MAX_CELLS * MVPP2_MAX_PORTS * \
    MVPP2_MAX_RXQ)

#define MVPP2_NETMAP_TXD_L3INFO_SHIFT  26
#define MVPP2_NETMAP_TXD_L4INFO_SHIFT  24
#define MVPP2_NETMAP_TXD_L3_CSUM_EN_SHIFT 14
#define MVPP2_NETMAP_TXD_L4_CSUM_EN_SHIFT 13

struct mvpp2_ntmp_buf_idx {
 int rx;
 int tx;
};

struct mvpp2_ntmp_cell_params {
 int bm_pool_num;
 int active_if;
};

struct mvpp2_ntmp_params {
 struct mvpp2_ntmp_buf_idx buf_idx[MVPP2_NETMAP_MAX_QUEUES_NUM];
 struct mvpp2_ntmp_cell_params cell_params[MVPP2_MAX_CELLS];
};

static struct mvpp2_ntmp_params *ntmp_params;

/*
* Register/unregister
* adapter is pointer to eth_port
*/
/*
* Register/unregister. We are already under netmap lock.
*/
static int
mv_pp2x_netmap_reg(struct netmap_adapter *na, int onoff)
{
 struct ifnet *ifp = na->ifp;
 struct SOFTC_T *adapter = netdev_priv(ifp);
 struct mvpp2_ntmp_cell_params *cell_params;
 u_int rxq, queue, si, cell_id;
```

```
if (!na)
 return -EINVAL;

/* stop current interface */
if (!netif_running(adapter->dev))
 return -EINVAL;

mv_pp2x_stop(adapter->dev);

pr_debug("%s: stopping interface\n", ifp->name);

cell_id = adapter->priv->pp2_cfg.cell_index;
cell_params = &ntmp_params->cell_params[cell_id];

/* enable or disable flags and callbacks in na and ifp */
if (onoff) {
 u_int pool;

 pr_info("Netmap started\n");

 nm_set_native_flags(na);
 adapter->flags |= MVPP2_F_IFCAP_NETMAP;
 cell_params->active_if++;
 if (cell_params->bm_pool_num == 0) {
  if (mv_pp2x_bm_pool_ext_add(ifp->dev.parent,
   adapter->priv, &pool,
   MVPP2_BM_NETMAP_PKT_SIZE) != 0) {
   pr_err("Unable to allocate a new pool\n");
   return -EINVAL;
  }
  cell_params->bm_pool_num = pool;
 }
} else {
 u_int i, idx;

 pr_info("Netmap stop\n");

 nm_clear_native_flags(na);

 idx = cell_id * MVPP2_MAX_PORTS * MVPP2_MAX_RXQ
      + adapter->id * MVPP2_MAX_RXQ;
 /* restore buf_idx to original place in ring
  * Netmap releases buffers according to buf_idx, so
  * they need to be in place.
  */
 for (queue = 0; queue < na->num_rx_rings;
     queue++) {
```

```
 struct netmap_kring *kring =
  &na->rx_rings[queue];
 struct netmap_ring *ring = kring->ring;
 struct netmap_slot *slot = &ring->slot[0];

 if (ntmp_params->buf_idx[idx + queue].rx != 0){
  for (i = 0; i < na->num_rx_desc; i++) {
   si = netmap_idx_n2k(
    &na->rx_rings[queue], i);
   (slot + si)->buf_idx =
       ntmp_params->buf_idx[idx + queue].rx + i;
  }
  ntmp_params->buf_idx[idx + queue].rx = 0;
 }
}
for (queue = 0; queue < na->num_tx_rings;
     queue++) {
 struct netmap_kring *kring =
  &na->tx_rings[queue];
 struct netmap_ring *ring = kring->ring;
 struct netmap_slot *slot = &ring->slot[0];

 if (ntmp_params->buf_idx[idx + queue].tx != 0){
  for (i = 0; i < na->num_tx_desc; i++) {
   si = netmap_idx_n2k(
    &na->tx_rings[queue], i);
   (slot + si)->buf_idx =
       ntmp_params->buf_idx[idx + queue].tx + i;
  }
  ntmp_params->buf_idx[idx + queue].tx = 0;
 }
}

if (--cell_params->active_if < 0) {
 pr_err("Error in active interfaces\n");
 return -EINVAL;
}
if (cell_params->active_if == 0) {
 pr_info("removig netmap BM pool %d from cell %d\n",
  cell_params->bm_pool_num, cell_id);
 for (rxq = 0; rxq < na->num_rx_rings; rxq++) {
  mv_pp2x_swf_bm_pool_assign(adapter, rxq,
    adapter->pool_long->id,
    adapter->pool_short->id);
 }

 mv_pp2x_bm_pool_destroy(ifp->dev.parent, adapter->priv,
    &adapter->priv->bm_pools[cell_params->bm_pool_num]);
```

```
  cell_params->bm_pool_num = 0;
 }

 adapter->flags &= ~MVPP2_F_IFCAP_NETMAP;
}

if (netif_running(adapter->dev)) {
 mv_pp2x_open(adapter->dev);
 pr_debug("%s: starting interface\n", ifp->name);
}
return 0;
}

/*
* Reconcile kernel and user view of the transmit ring.
*/
static int
mv_pp2x_netmap_txsync(struct netmap_kring *kring, int flags)
{
 struct netmap_adapter *na = kring->na;
 struct ifnet *ifp = na->ifp;
 struct netmap_ring *ring = kring->ring;
 u_int ring_nr = kring->ring_id;
 u_int nm_i;  /* index into the netmap ring */
 u_int nic_i = 0; /* Number of sent packets from NIC  */
 u_int n;
 u_int const lim = kring->nkr_num_slots - 1;
 u_int count = 0;
 u_int tx_count = 0, tx_bytes = 0;
 u_int const head = kring->rhead;
 struct mv_pp2x_tx_desc *tx_desc;
 struct mv_pp2x_aggr_tx_queue *aggr_txq = NULL;
 struct mv_pp2x_txq_pcpu *txq_pcpu;
 struct mv_pp2x_tx_queue *txq;
 u_int tx_sent;
 u8 first_addr_space;
 u_int num_cpus = num_active_cpus();
 u_int rsv_chunk;
 int cpu = get_cpu();

 /* generate an interrupt approximately every half ring */
 /*u_int report_frequency = kring->nkr_num_slots >> 1;*/

 /* device-specific */
 /* take a copy of ring->cur now, and never read it again */

 struct SOFTC_T *adapter = netdev_priv(ifp);
```

```
txq = adapter->txqs[ring_nr % adapter->num_tx_queues];
txq_pcpu = this_cpu_ptr(txq->pcpu);
aggr_txq = &adapter->priv->aggr_txqs[cpu];
first_addr_space = adapter->priv->pp2_cfg.first_sw_thread;

rsv_chunk = (num_cpus > 2) ?
    (txq_pcpu->size >> 2) -1 :
    (txq_pcpu->size >> (num_cpus >> 1)) -1;

/*
 * Process new packets to send. j is the current index in the
 * netmap ring, l is the corresponding index in the NIC ring.
 */

if (!netif_carrier_ok(ifp))
 goto out;

nm_i = kring->nr_hwcur;
if (nm_i != head) { /* we have new packets to send */
 for (n = 0; nm_i != head; n++) {
  /* slot is the current slot in the netmap ring */
  struct netmap_slot *slot = &ring->slot[nm_i];
  u_int len = slot->len;
  u64 paddr;
  void *addr = PNMB(na, slot, &paddr);

  /* device-specific */
  NM_CHECK_ADDR_LEN(na, addr, len);
  slot->flags &= ~NS_REPORT;

  /* check aggregated TXQ resource */
  if (unlikely((aggr_txq->count + 1) > aggr_txq->size)) {
   if (mv_pp2x_aggr_desc_num_check(adapter->priv, aggr_txq, 1, cpu))
    break;
  }

  if (unlikely(txq_pcpu->reserved_num == 0)){
   txq_pcpu->reserved_num += mv_pp2x_txq_alloc_reserved_desc(adapter->priv,
       txq, rsv_chunk, cpu);
   if (txq_pcpu->reserved_num == 0)
    break;
  }

  tx_desc = mv_pp2x_txq_next_desc_get(aggr_txq);
  tx_desc->phys_txq = txq->id; /* Destination Queue ID */
  mv_pp2x_txdesc_phys_addr_set(adapter->priv->pp2_version,
   (uint32_t)(paddr) & ~MVPP2_TX_DESC_ALIGN,
   tx_desc);
```

```
 tx_desc->data_size = len;
 tx_desc->packet_offset = slot->data_offs;

 if (slot->csum_offload.l3_offset) {
  tx_desc->command = (slot->csum_offload.l3_offset << MVPP2_TXD_L3_OFF_SHIFT) |
      (slot->csum_offload.ip_hdr_len << MVPP2_TXD_IP_HLEN_SHIFT) |
      ((slot->csum_offload.l3_type & 0x1) << MVPP2_NETMAP_TXD_L3INFO_SHIFT) |
      ((~slot->csum_offload.l3_type & 0x2) << MVPP2_NETMAP_TXD_L3_CSUM_EN_SHIFT) |
      ((slot->csum_offload.l4_type & 0x1) << MVPP2_NETMAP_TXD_L4INFO_SHIFT) |
      ((~slot->csum_offload.l3_type & 0x2) << MVPP2_NETMAP_TXD_L4_CSUM_EN_SHIFT) |
      MVPP2_TXD_F_DESC | MVPP2_TXD_L_DESC;
 } else
  tx_desc->command = MVPP2_TXD_L4_CSUM_NOT |
   MVPP2_TXD_IP_CSUM_DISABLE | MVPP2_TXD_F_DESC |
   MVPP2_TXD_L_DESC;

 mv_pp2x_txq_inc_put(adapter->priv->pp2_version,
      txq_pcpu, addr, tx_desc);

 txq_pcpu->count += 1;
 txq_pcpu->reserved_num -= 1;

 if (++count >= (aggr_txq->size >> 2)) {
  wmb(); /* synchronize writes to the NIC ring */
  mv_pp2x_aggr_txq_pend_desc_add(adapter, count);
  aggr_txq->count += count;
  count = 0;
 }
 tx_bytes += len;
 tx_count++;

 if (slot->flags & NS_BUF_CHANGED)
  slot->flags &= ~NS_BUF_CHANGED;

 nm_i = nm_next(nm_i, lim);
}

/* Enable transmit */
if (count > 0) {
 wmb(); /* synchronize writes to the NIC ring */
 mv_pp2x_aggr_txq_pend_desc_add(adapter, count);
 aggr_txq->count += count;
}
/* Update tx counters */
if (tx_count) {
 struct mv_pp2x_pcpu_stats *stats = this_cpu_ptr(adapter->stats);
 u64_stats_update_begin(&stats->syncp);
 stats->tx_packets += tx_count;
```

```
  stats->tx_bytes += tx_bytes;
  u64_stats_update_end(&stats->syncp);
 }
 kring->nr_hwcur = head; /* the saved ring->cur */
}

/*
 * Second part: reclaim buffers for completed transmissions.
 */
nic_i = netmap_idx_k2n(kring, kring->nr_hwtail);
tx_sent = mv_pp2x_txq_sent_desc_proc(adapter,
   (first_addr_space + txq_pcpu->cpu), txq->id);
txq_pcpu->count -= tx_sent;

if (tx_sent >= kring->nkr_num_slots) {
 pr_warn("tx_sent: %d, nkr_num_slots: %d\n", tx_sent,
  kring->nkr_num_slots);
 tx_sent = tx_sent % kring->nkr_num_slots;
}

nic_i = ((tx_sent + nic_i) % kring->nkr_num_slots);
kring->nr_hwtail = netmap_idx_n2k(kring, nic_i);

out:
 put_cpu();
 return 0;
}

/*
* Reconcile kernel and user view of the receive ring.
*/
static int
mv_pp2x_netmap_rxsync(struct netmap_kring *kring, int flags)
{
 struct netmap_adapter *na = kring->na;
 struct ifnet *ifp = na->ifp;
 struct netmap_ring *ring = kring->ring;
 u_int ring_nr = kring->ring_id;
 u_int nm_i; /* index into the netmap ring */
 u_int nic_i; /* index into the NIC ring */
 u_int n = 0, m = 0;
 u_int rx_count = 0, rx_bytes = 0;
 u_int const lim = kring->nkr_num_slots - 1;
 u_int const head = kring->rhead;
 int force_update = (flags & NAF_FORCE_READ) || kring->nr_kflags &
     NKR_PENDINTR;
 u_int rx_done = 0;
 u_int cell_id, bm_pool;
```

```
/* device-specific */
struct SOFTC_T *adapter = netdev_priv(ifp);
struct mv_pp2x_rx_queue *rxq;
struct mv_pp2x_pcpu_stats *stats;
int cpu;

if (!netif_carrier_ok(ifp))
 return 0;

rxq = adapter->rxqs[ring_nr];

if (head > lim)
 return netmap_ring_reinit(kring);

cell_id = adapter->priv->pp2_cfg.cell_index;
bm_pool = ntmp_params->cell_params[cell_id].bm_pool_num;

cpu = get_cpu();
stats = this_cpu_ptr(adapter->stats);

/* hardware memory barrier that prevents any memory read access from
*  being moved and executed on the other side of the barrier rmb();
*/
rmb();

/* First part: import newly received packets into the netmap ring */
/* netmap_no_pendintr = 1, see netmap.c */
if (netmap_no_pendintr || force_update) {
 struct mv_pp2x_rx_desc *curr;
 struct netmap_slot *slot;
 /* Get number of received packets */
 u16 slot_flags = kring->nkr_slot_flags;
 /* TBD :: remove CRC or not */
 u32 strip_crc = (0) ? 4 : 0;

 rx_done = mv_pp2x_rxq_received(adapter, rxq->id);
 rx_done = (rx_done >= lim) ? lim - 1 : rx_done;
 nic_i = rxq->next_desc_to_proc;
 nm_i = kring->nr_hwtail;

 for (n = 0; n < rx_done; n++) {
  if (unlikely(nm_next(nm_i, lim) == kring->nr_hwcur)) break;
  curr = MVPP2_QUEUE_DESC_PTR(rxq, nic_i);

#if defined(__BIG_ENDIAN)
  if (adapter->priv->pp2_version == PPV21)
   mv_pp21_rx_desc_swap(curr);
```

```
  else
   mv_pp22_rx_desc_swap(curr);
#endif /* __BIG_ENDIAN */

  /* TBD : check for ERRORs */
  if (unlikely(curr->status & MVPP2_RXD_ERR_SUMMARY)) {
   dma_addr_t paddr = mv_pp22_rxdesc_phys_addr_get(curr);
   mv_pp2x_bm_pool_put_virtual(&adapter->priv->hw, bm_pool, paddr,
     mv_pp22_rxdesc_cookie_get(curr), cpu);
   nic_i = nm_next(nic_i, lim);
   /* update rx error counters */
   u64_stats_update_begin(&stats->syncp);
   adapter->dev->stats.rx_errors++;
   u64_stats_update_end(&stats->syncp);
   continue;
  }

  slot = &ring->slot[nm_i];
  slot->len = (curr->data_size) -
      strip_crc - MVPP2_MH_SIZE;
  slot->data_offs = NET_SKB_PAD + MVPP2_MH_SIZE;
  slot->buf_idx = (uintptr_t)
    mv_pp22_rxdesc_cookie_get(curr);
  slot->flags = slot_flags;
  nm_i = nm_next(nm_i, lim);
  nic_i = nm_next(nic_i, lim);
  rx_count++;
  rx_bytes += slot->len;
 }
 if (n) { /* update the state variables and rx counters */

  rxq->next_desc_to_proc = nic_i;
  kring->nr_hwtail = nm_i;
  mv_pp2x_rxq_status_update(adapter, rxq->id, n, n);

  u64_stats_update_begin(&stats->syncp);
  stats->rx_packets += rx_count;
  stats->rx_bytes += rx_bytes;
  u64_stats_update_end(&stats->syncp);
 }
 kring->nr_kflags &= ~NKR_PENDINTR;
}

/*
 * Second part: skip past packets that userspace has released.
 */
nm_i  = kring->nr_hwcur; /* netmap ring index */
```

```
 if (nm_i != head) { /* userspace has released some packets. */
  nic_i = netmap_idx_k2n(kring, nm_i); /* NIC ring index */
  for (m = 0; nm_i != head; m++) {
   struct netmap_slot *slot = &ring->slot[nm_i];
   dma_addr_t paddr;

   void *addr = PNMB(na, slot, &paddr);

   if (addr == NETMAP_BUF_BASE(na)) {    /* bad buf */
    goto ring_reset;
   }

   /* In big endian mode: no need to swap descriptor here,
   *  already swapped before
   */
   mv_pp2x_bm_pool_put_virtual(&adapter->priv->hw, bm_pool, paddr,
        (u8 *)(uintptr_t)slot->buf_idx, cpu);
   /* mark this slot as invalid */
   slot->buf_idx = 0;

   if (slot->flags & NS_BUF_CHANGED)
    slot->flags &= ~NS_BUF_CHANGED;

   nm_i = nm_next(nm_i, lim);
   nic_i = nm_next(nic_i, lim);
  }
  kring->nr_hwcur = head;
  /*mv_pp2x_rxq_status_update(adapter, rxq->id, 0, m)*/;
 }
 put_cpu();
 return 0;

ring_reset:
 return netmap_ring_reinit(kring);
}

/*
* Make the rx ring point to the netmap buffers.
*/
static int mv_pp2x_netmap_rxq_init_buffers(struct SOFTC_T *adapter)
{
 struct ifnet *ifp = adapter->dev; /* struct net_devive */
 struct netmap_adapter *na = NA(ifp);
 struct netmap_slot *slot;
 dma_addr_t paddr;
 u32 *addr;
 struct mv_pp2x_rx_queue *rxq;
 u_int queue, idx, cell_id, bm_pool;
```

```
u_int i = 0, si;
struct mv_pp2x_bm_pool *bm_pool_netmap;
struct netmap_kring *kring;
int cpu;

cell_id = adapter->priv->pp2_cfg.cell_index;
idx = cell_id * MVPP2_MAX_PORTS * MVPP2_MAX_RXQ
     + adapter->id * MVPP2_MAX_RXQ;
bm_pool = ntmp_params->cell_params[cell_id].bm_pool_num;
bm_pool_netmap = &adapter->priv->bm_pools[bm_pool];

pr_debug("Cell ID: %d, Port %d\n", cell_id, adapter->id);
pr_debug("--------------------\n");

for_each_online_cpu(cpu) {
 /* Reset the BM virtual and physical address high register
   (not in use in netmap) */
 mv_pp2x_relaxed_write(&adapter->priv->hw, MVPP22_BM_PHY_VIRT_HIGH_RLS_REG,
   0, cpu);
}

for (queue = 0; queue < na->num_rx_rings; queue++) {
 rxq = adapter->rxqs[queue];

 kring = na->rx_rings + queue;
 if (kring->nr_mode == NKR_NETMAP_ON){
  pr_debug("rx queue %d, ring already initialized %p\n",
   queue, (void *)kring);
  continue;
 }

 /* initialize the rx ring */
 slot = netmap_reset(na, NR_RX, queue, 0);
 if (!slot){
  pr_debug("rx queue %d, ring is still null\n", queue);
  continue;
 }

 ntmp_params->buf_idx[idx + queue].rx = slot->buf_idx;
 mv_pp2x_swf_bm_pool_assign(adapter, queue, bm_pool, bm_pool);
 cpu = get_cpu();
 for (i = 0; i < na->num_rx_desc; i++) {
  si = netmap_idx_n2k(&na->rx_rings[queue], i);
  addr = PNMB(na, slot + si, &paddr);
  mv_pp2x_bm_pool_put_virtual(&adapter->priv->hw, bm_pool, paddr,
       (u8 *)(uint64_t)(slot + si)->buf_idx, cpu);
  bm_pool_netmap->buf_num++;
  if (bm_pool_netmap->buf_num >= MVPP2_BM_POOL_SIZE_MAX){
```

```
   pr_info("Max size of BM pool reached %d\n",
    bm_pool_netmap->buf_num);
   break;
  }
 }
 put_cpu();
 pr_debug("rx queue %d, buf_idx[%d].rx %d, new ring %p, BM pool buffers %d\n",
  queue, idx + queue,
  ntmp_params->buf_idx[idx + queue].rx, (void *)kring,
  bm_pool_netmap->buf_num);
 rxq->next_desc_to_proc = 0;
 }
 return 1;
}

/*
* Make the tx ring point to the netmap buffers.
*/
static int mv_pp2x_netmap_txq_init_buffers(struct SOFTC_T *adapter)
{
 struct ifnet *ifp = adapter->dev;
 struct netmap_adapter *na = NA(ifp);
 struct netmap_slot *slot;
 struct mv_pp2x_tx_queue *txq;
 u_int queue, idx;
 struct netmap_kring *kring;

 idx = adapter->priv->pp2_cfg.cell_index *
      MVPP2_MAX_PORTS * MVPP2_MAX_RXQ + adapter->id * MVPP2_MAX_RXQ;

 /* initialize the tx ring */
 for (queue = 0; queue < na->num_tx_rings; queue++) {
  txq = adapter->txqs[queue];

  kring = na->tx_rings + queue;
  if (kring->nr_mode == NKR_NETMAP_ON){
   pr_debug("tx queue %d, ring already initialized %p\n",
    queue, (void *)kring);
   continue;
  }

  /* initialize the rx ring */
  slot = netmap_reset(na, NR_TX, queue, 0);
  if (!slot){
   pr_debug("tx queue %d, ring is still null\n", queue);
   continue;
  }
```

```
  ntmp_params->buf_idx[idx + queue].tx = slot->buf_idx;
  pr_debug("tx queue %d, buf_idx[%d].tx %d, new ring %p\n",
   queue, idx + queue,
   ntmp_params->buf_idx[idx + queue].tx, (void *)kring);
 }
 return 1;
}

/* Update the mvpp2x-net device configurations. Number of queues can
* change dinamically
*/

static int
mv_pp2x_netmap_config(struct netmap_adapter *na, u_int *txr, u_int *txd,
        u_int *rxr, u_int *rxd)
{
 struct ifnet *ifp = na->ifp;
 struct SOFTC_T *adapter = netdev_priv(ifp);

 *txr = adapter->num_tx_queues * num_active_cpus();
 *rxr = adapter->num_rx_queues;
 *rxd = adapter->rx_ring_size;
 *txd = adapter->tx_ring_size;

 return 0;
}

static void
mv_pp2x_netmap_attach(struct SOFTC_T *adapter)
{
 struct netmap_adapter na;

 bzero(&na, sizeof(na));

 if (!ntmp_params)
  ntmp_params = kmalloc(sizeof(*ntmp_params), GFP_KERNEL);

 na.ifp = adapter->dev; /* struct net_device */
 na.num_tx_desc = adapter->tx_ring_size;
 na.num_rx_desc = adapter->rx_ring_size;
 na.nm_register = mv_pp2x_netmap_reg;
 na.nm_config = mv_pp2x_netmap_config;
 na.nm_txsync = mv_pp2x_netmap_txsync;
 na.nm_rxsync = mv_pp2x_netmap_rxsync;
 na.num_tx_rings = adapter->num_tx_queues * num_active_cpus();
 na.num_rx_rings = adapter->num_rx_queues;
 netmap_attach(&na);
}
```

```
#endif  /* __MV_PP2X_NETMAP_H__ */
```