Operating Systems

DMA Devices and the Ethernet Driver

Device Drivers and DMA

- We have seen the structure of device drivers, with upper and lower halves interfacing with shared data structures
- Next we examine devices that can transfer data to memory directly, and look at their device drivers
  - DMA – Direct Memory Access
- One necessity from the driver perspective is separation of request blocks and their buffers

DMA and Buffers

- The goal of DMA is to reduce interrupts and allow devices to transfer data to memory while the CPU is busy with other tasks
  - Reduction of overhead
  - Concurrent activities
- A bus can only transfer a bus-width of data at a time
  - One word or a double word
  - One byte a bit at a time for serial interfaces
- Block or frame oriented devices need to transfer multiple words for every transaction
- DMA allows the transfer of an entire chunk, or multiple chunks, before generating an interrupt

DMA and Buffers

- For a write operation, place data in a buffer and create a write request to pass to the device
  - The request contains a pointer to the buffer to be written
- For read, post a read request to the device, with a pointer to an empty buffer and
  - For a disk, specify the block to be read
  - For a network device, the read request is left pending until the device produces data

DMA and Buffers

- While system software is processing a data element, the device should be able to operate on others
  - Multiple data buffers posted
  - This gives rise to a
- Producer / consumer problem with a familiar solution
  - Have a logical ring of buffer elements
- Control access to the buffers based on whether they are empty or full

DMA Operations

- Post a linked list of operations
  - The device can follow the linked list of operation requests
- Generates an interrupt on completion of one operation, but continues to the next without waiting for the processor
- As long as requests remain on the list, the device can continue to operate
  - Multiple posted receive operations allow the network to continue to receive packets and place them in the ring buffer

DMA Operations

- Must have an EMPTY/FULL status bit that the hardware understands
- Hardware traverses the list, marking FULL or EMPTY
  - Depending on ingress or egress
- On input, if the hardware goes around the ring and all buffers are FULL, set an overflow bit
- On output, consume and transmit all FULL buffers

Ethernet Driver with DMA

- Separate DMA engines for input and output
- As in the serial case, create input and output ring buffers
- Separate input and output control registers get pointers to input and output request lists
- Mark all buffers as empty, post receive operations for input buffers
- The Galileo Ethernet device has a single interrupt vector, so as in the serial case, the handler must determine whether the interrupts indicates and input or output operation

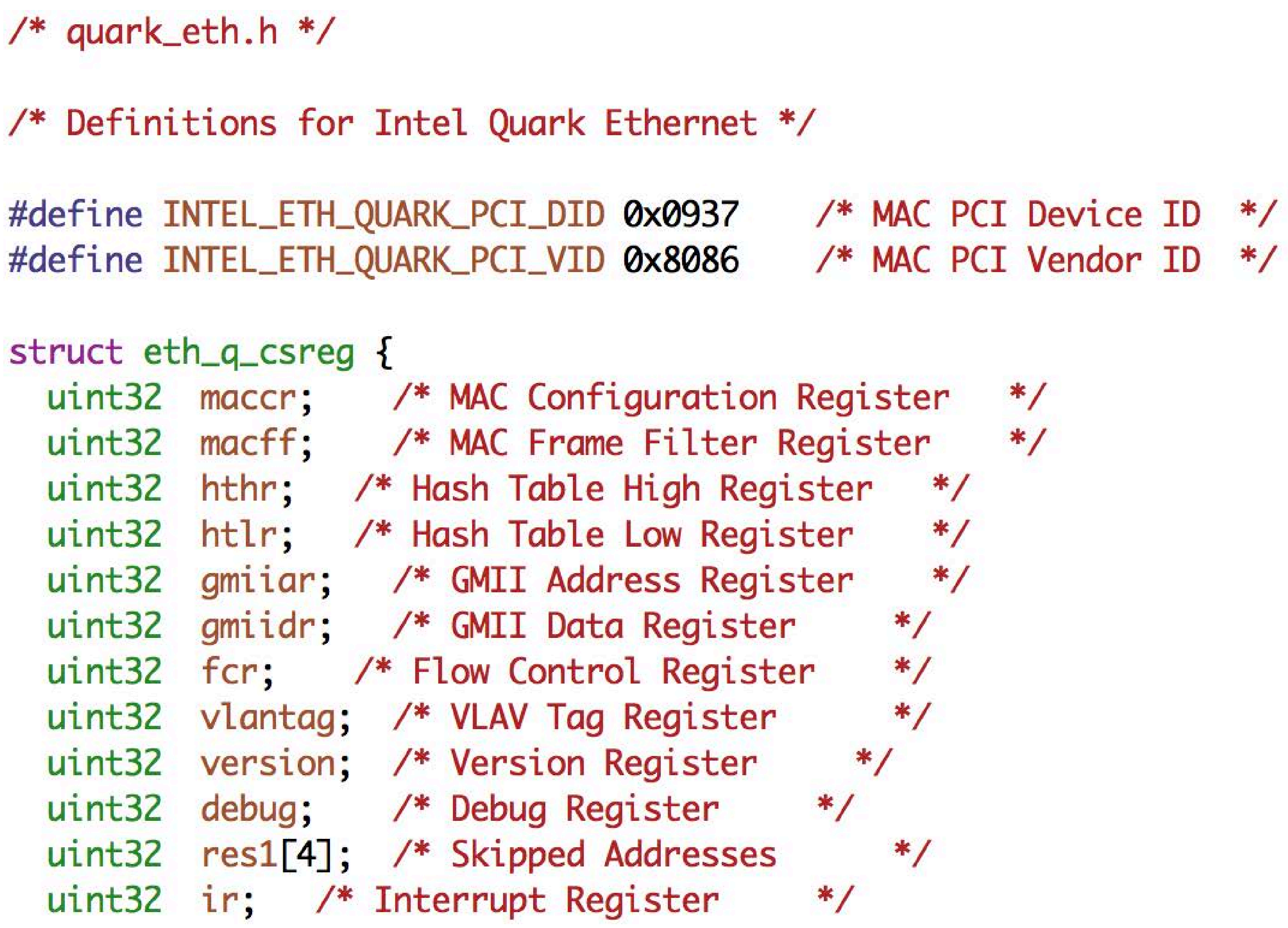
Ethernet Hardware

- Ethernet defines a Media Access Control (MAC) layer
  - The lower half of the “Data Link” layer in the OSI network model
  - The upper half is the Logical Link Control (LLC) layer
- Below the MAC is the Physical layer, or PHY
- The MAC and the PHY are decoupled in various ways so that the MAC operation is independent of e.g. copper or fiber optic physical layers

Ethernet Hardware

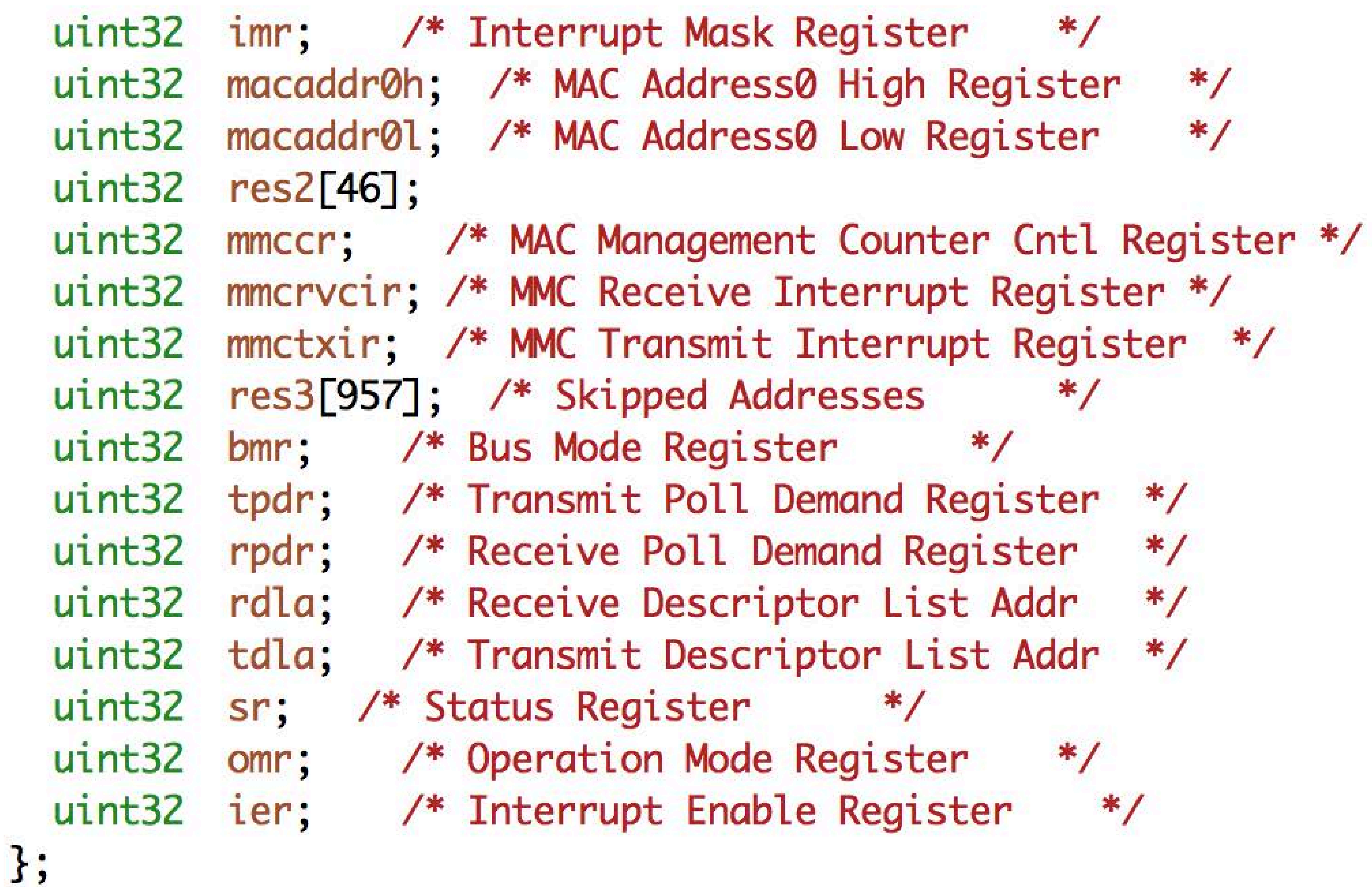
- The system interfaces with the MAC using the Media-Independent Interface (MII)
- The Intel Galileo (Quark SoC) exposes MII operations with the General MII (GMII) interface
  - Specifically a simplified subset known as the Reduced MII (RMII) interface
- The BeagleBone Black exposes a Management Data Input/Output (MDIO) interface
- Communication occurs from the Serial Management Interface (SMI) to the PHY chip
- These details matter in the driver, as the driver must understand the Control/Status registers (CSR) of the devices

quark_eth.h



```
/* quark_eth.h */

/* Definitions for Intel Quark Ethernet */

#define INTEL_ETH_QUARK_PCI_DID 0x0937    /* MAC PCI Device ID  */
#define INTEL_ETH_QUARK_PCI_VID 0x8086    /* MAC PCI Vendor ID  */

struct eth_q_csreg {
  uint32  maccr;    /* MAC Configuration Register   */
  uint32  macff;    /* MAC Frame Filter Register    */
  uint32  hthr;   /* Hash Table High Register   */
  uint32  htlr;   /* Hash Table Low Register    */
  uint32  gmiiar;   /* GMII Address Register    */
  uint32  gmiidr;   /* GMII Data Register     */
  uint32  fcr;    /* Flow Control Register    */
  uint32  vlantag;  /* VLAV Tag Register        */
  uint32  version;  /* Version Register       */
  uint32  debug;    /* Debug Register       */
  uint32  res1[4];  /* Skipped Addresses        */
  uint32  ir;   /* Interrupt Register     */
```

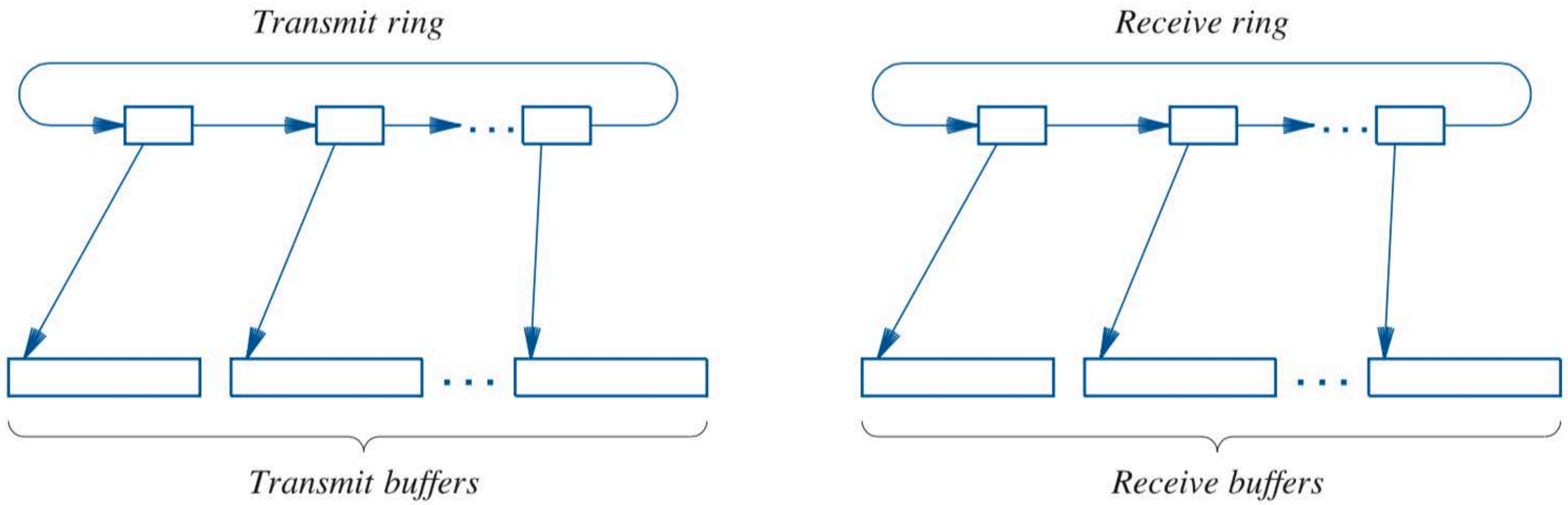
quark_eth.h



```
  uint32  imr;      /* Interrupt Mask Register    */
  uint32  macaddr0h;  /* MAC Address0 High Register   */
  uint32  macaddr0l;  /* MAC Address0 Low Register    */
  uint32  res2[46];
  uint32  mmccr;     /* MAC Management Counter Cntl Register */
  uint32  mmcrvcir; /* MMC Receive Interrupt Register */
  uint32  mmctxir;  /* MMC Transmit Interrupt Register  */
  uint32  res3[957];  /* Skipped Addresses         */
  uint32  bmr;      /* Bus Mode Register        */
  uint32  tpdr;     /* Transmit Poll Demand Register  */
  uint32  rpdr;     /* Receive Poll Demand Register   */
  uint32  rdla;     /* Receive Descriptor List Addr   */
  uint32  tdla;     /* Transmit Descriptor List Addr  */
  uint32  sr;    /* Status Register        */
  uint32  omr;      /* Operation Mode Register    */
  uint32  ier;      /* Interrupt Enable Register    */
};
```

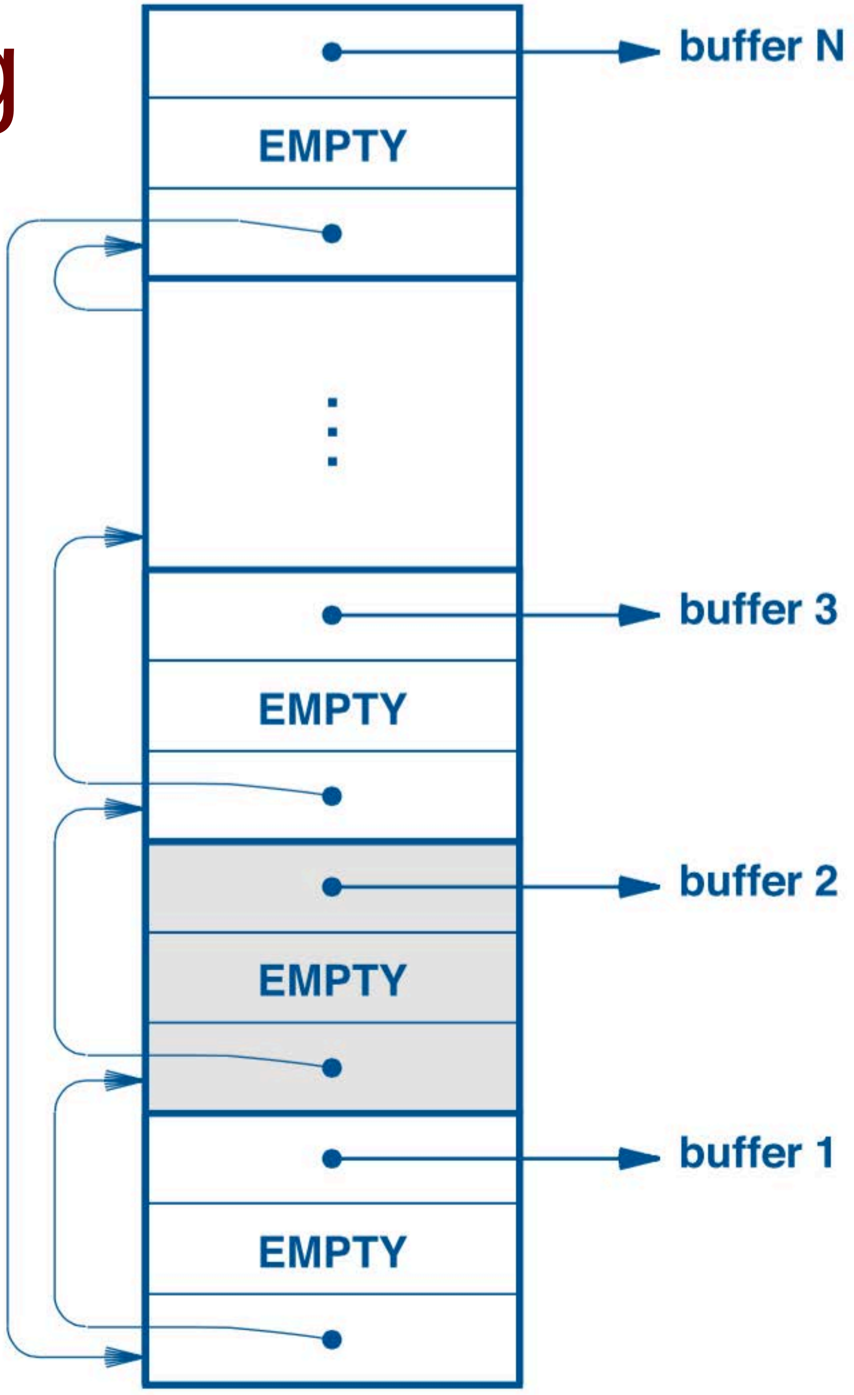
Rings and Buffers

- With separate request descriptors and buffers, the "rings" consist of the descriptors
  - The serial driver combined them
- The input and output rings are linked lists of descriptors with a status word indicating whether the associated buffer is empty or full



Descriptor Ring

- Each node contains
  - A pointer to the buffer
  - Status word
  - A pointer to the next node in the list
- The driver uses getmem() to allocate contiguous memory and links the nodes together



quark_eth.h

```
/* Quark Ethernet Transmit Descriptor */

struct eth_q_tx_desc {
  uint32  ctrlstat; /* Control and status */
  uint16  buf1size; /* Size of buffer 1 */
  uint16  buf2size; /* Size of buffer 2 */
  uint32  buffer1;  /* Address of buffer 1  */
  uint32  buffer2;  /* Address of buffer 2  */
};

#define ETH_QUARK_TDCS_OWN  0x80000000  /* Descrip. owned by DMA*/
#define ETH_QUARK_TDCS_IC 0x40000000  /* Int on Completion  */
#define ETH_QUARK_TDCS_LS 0x20000000  /* Last Segment    */
#define ETH_QUARK_TDCS_FS 0x10000000  /* First Segment   */
#define ETH_QUARK_TDCS_TER  0x00200000  /* Transmit End of Ring */
#define ETH_QUARK_TDCS_ES 0x00008000  /* Error Summary   */
```

quark_eth.h

```
/* Quark Ethernet Receive Descriptor */

struct eth_q_rx_desc {
  uint32  status;   /* Desc status word */
  uint16  buf1size; /* Size of buffer 1 */
  uint16  buf2size; /* Size of buffer 2 */
  uint32  buffer1;  /* Address of buffer 1  */
  uint32  buffer2;  /* Address of buffer 2  */
};
#define rdctl1  buf1size  /* Buffer 1 size field has control bits too */
#define rdctl2  buf2size  /* Buffer 2 size field has control bits too */

#define ETH_QUARK_RDST_OWN  0x80000000  /* Descrip. owned by DMA*/
#define ETH_QUARK_RDST_ES 0x00008000  /* Error Summary   */
#define ETH_QUARK_RDST_FS 0x00000200  /* First Segment   */
#define ETH_QUARK_RDST_LS 0x00000100  /* Last segment    */
#define ETH_QUARK_RDST_FTETH  0x00000020  /* Frame Type = Ethernet*/
```

Ethernet Driver Control Block (ether.h)

```
/* ether.h */

/* Ethernet packet format:

 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Dest. MAC (6)  |  Src. MAC (6)   |Type (2)|      Data (46-1500)...    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*/

#define ETH_ADDR_LEN  6   /* Len. of Ethernet (MAC) addr. */
typedef unsigned char Eaddr[ETH_ADDR_LEN];/* Physical Ethernet address*/

/* Ethernet packet header */

struct  etherPkt {
  byte  dst[ETH_ADDR_LEN];  /* Destination Mac address  */
  byte  src[ETH_ADDR_LEN];  /* Source Mac address    */
  uint16  type;      /* Ether type field   */
  byte  data[1];     /* Packet payload   */
};

#define ETH_HDR_LEN   14  /* Length of Ethernet packet  */
          /*   header      */
```

ether.h

```
/* Ethernet DMA buffer sizes */

#define ETH_MTU      1500  /* Maximum transmission unit  */
#define ETH_VLAN_LEN    4 /* Length of Ethernet vlan tag  */
#define ETH_CRC_LEN   4 /* Length of CRC on Ethernet  */
          /*   frame       */

#define ETH_MAX_PKT_LEN ( ETH_HDR_LEN + ETH_VLAN_LEN + ETH_MTU )

#define ETH_BUF_SIZE    2048  /* A multiple of 16 greater   */
          /*   than the max packet  */
          /*   size (for cache alignment) */

/* State of the Ethernet interface */

#define ETH_STATE_FREE    0 /* Control block is unused  */
#define ETH_STATE_DOWN    1 /* Interface is inactive  */
#define ETH_STATE_UP    2 /* Interface is currently active*/

/* Ethernet device control functions */

#define ETH_CTRL_GET_MAC      1   /* Get the MAC for this device  */
#define ETH_CTRL_ADD_MCAST  2 /* Add a multicast address  */
#define ETH_CTRL_REMOVE_MCAST 3 /* Remove a multicast address */
```

Ethernet Driver Control Block (ether.h)

```
/* Control block for Ethernet device */

struct  ethcblk {
  byte  state;    /* ETH_STATE_... as defined above   */
  struct  dentry  *phy; /* physical eth device for Tx DMA   */
  byte  type;     /* NIC type_... as defined above  */

  /* Pointers to associated structures */

  struct  dentry  *dev; /* Address in device switch table */
  void  *csr;   /* Control and status regsiter address  */
  uint32  pcidev;   /* PCI device number      */
  uint32  iobase;   /* I/O base from config     */
  uint32  flashbase;      /* Flash base from config         */
      uint32  membase;  /* Memory base for device from config */

  void    *rxRing;  /* Ptr to array of recv ring descriptors*/
  void    *rxBufs;  /* Ptr to Rx packet buffers in memory */
  uint32  rxHead;   /* Index of current head of Rx ring */
  uint32  rxTail;   /* Index of current tail of Rx ring */
  uint32  rxRingSize; /* Size of Rx ring descriptor array */
  uint32  rxIrq;    /* Count of Rx interrupt requests       */
```

```
    void     *txRing;  /* Ptr to array of xmit ring descriptors*/
    void     *txBufs;  /* Ptr to Tx packet buffers in memory */
    uint32  txHead;    /* Index of current head of Tx ring */
    uint32  txTail;    /* Index of current tail of Tx ring */
    uint32  txRingSize; /* Size of Tx ring descriptor array */
    uint32  txIrq;     /* Count of Tx interrupt requests       */

    uint8 devAddress[ETH_ADDR_LEN];/* MAC address     */

    uint8 addrLen;   /* Hardware address length          */
    uint16  mtu;          /* Maximum transmission unit (payload)  */

    uint32  errors;    /* Number of Ethernet errors    */
    sid32 isem;    /* Semaphore for Ethernet input   */
    sid32 osem;      /* Semaphore for Ethernet output  */
    uint16  istart;    /* Index of next packet in the ring      */

    int16 inPool;    /* Buffer pool ID for input buffers    */
    int16 outPool;   /* Buffer pool ID for output buffers   */

    int16   proms;     /* Nonzero => promiscuous mode     */

    int16   ed_mcset;         /* Nonzero => multicast reception set   */
    int16   ed_mcc;    /* Count of multicast addresses    */
    Eaddr   ed_mca[ETH_NUM_MCAST];/* Array of multicast addrs    */
};
```

Reading from the Ethernet

- The DMA engine is configured to fill input buffers
- While the DMA engine can read the input descriptor ring, it cannot manipulate semaphores
- To read a packet, a process waits on a semaphore
  - Initial value is 0, so the first read will block
- When a packet is available, the interrupt handler will signal the semaphore and the process can read from the buffer

ethread.c

```
/*------------------------------------------------------------------------
 * ethread  -  Read an incoming packet on Intel Quark Ethernet
 *------------------------------------------------------------------------
 */
devcall ethread (
    struct dentry *devptr,  /* Entry in device switch table */
    char  *buf,        /* Buffer for the packet  */
    int32 len          /* Size of the buffer   */
  )
{
  struct  ethcblk *ethptr;  /* Ethertab entry pointer */
  struct  eth_q_rx_desc *rdescptr;/* Pointer to the descriptor  */
  struct  netpacket *pktptr;  /* Pointer to packet     */
  uint32  framelen = 0;    /* Length of the incoming frame */
  bool8 valid_addr;
  int32 i;

  ethptr = &ethertab[devptr->dvminor];

  while(1) {

    /* Wait until there is a packet in the receive queue */

    wait(ethptr->isem);
```

ethread.c

```
/* Point to the head of the descriptor list */

rdescptr = (struct eth_q_rx_desc *)ethptr->rxRing +
          ethptr->rxHead;
pktptr = (struct netpacket*)rdescptr->buffer1;

/* See if destination address is our unicast address */

if(!memcmp(pktptr->net_ethdst, ethptr->devAddress, 6)) {
  valid_addr = TRUE;

/* See if destination address is the broadcast address */

} else if(!memcmp(pktptr->net_ethdst,
                                NetData.ethbcast,6)) {
          valid_addr = TRUE;

/* For multicast addresses, see if we should accept */

    } else {
  valid_addr = FALSE;
  for(i = 0; i < (ethptr->ed_mcc); i++) {
    if(memcmp(pktptr->net_ethdst,
      ethptr->ed_mca[i], 6) == 0){
      valid_addr = TRUE;
      break;
    }
  }
}
```

ethread.c

```c
if(valid_addr == TRUE){ /* Accept this packet */

  /* Get the length of the frame */

  framelen = (rdescptr->status >> 16) & 0x00003FFF;

  /* Only return len characters to caller */

  if(framelen > len) {
    framelen = len;
  }

  /* Copy the packet into the caller's buffer */

  memcpy(buf, (void*)rdescptr->buffer1, framelen);
}

/* Increment the head of the descriptor list */

ethptr->rxHead += 1;
if(ethptr->rxHead >= ETH_QUARK_RX_RING_SIZE) {
  ethptr->rxHead = 0;
}
```

ethread.c

```c
    /* Reset the descriptor to max possible frame len */

    rdescptr->buf1size = sizeof(struct netpacket);

    /* If we reach the end of the ring, mark the descriptor */

    if(ethptr->rxHead == 0) {
      rdescptr->rdctl1 |= (ETH_QUARK_RDCTL1_RER);
    }

    /* Indicate that the descriptor is ready for DMA input */

    rdescptr->status = ETH_QUARK_RDST_OWN;

    if(valid_addr == TRUE) {
      break;
    }
  }

  /* Return the number of bytes returned from the packet */

  return framelen;

}
```

ethwrite.c

```
/*------------------------------------------------------------------------
 * ethwrite  -  enqueue packet for transmission on Intel Quark Ethernet
 *------------------------------------------------------------------------
 */
devcall ethwrite  (
    struct dentry *devptr,  /* Entry in device switch table */
    char  *buf,      /* Buffer that hols a packet  */
    int32 len        /* Length of the packet   */
  )
{
  struct  ethcblk *ethptr;  /* Pointer to control block */
  struct  eth_q_csreg *csrptr;  /* Address of device CSRs */
  volatile struct eth_q_tx_desc *descptr; /* Ptr to descriptor  */
  uint32 i;     /* Counts bytes during copy */

  ethptr = &ethertab[devptr->dvminor];

  csrptr = (struct eth_q_csreg *)ethptr->csr;

  /* Wait for an empty slot in the transmit descriptor ring */

  wait(ethptr->osem);
```

ethwrite.c

```c
/* Point to the tail of the descriptor ring */

descptr = (struct eth_q_tx_desc *)ethptr->txRing + ethptr->txTail;

/* Increment the tail index and wrap, if needed */

ethptr->txTail += 1;
if(ethptr->txTail >= ethptr->txRingSize) {
  ethptr->txTail = 0;
}

/* Add packet length to the descriptor */

descptr->buf1size = len;

/* Copy packet into the buffer associated with the descriptor */

for(i = 0; i < len; i++) {
  *((char *)descptr->buffer1 + i) = *((char *)buf + i);
}
```

ethwrite.c

```
  /* Mark the descriptor if we are at the end of the ring */

  if(ethptr->txTail == 0) {
    descptr->ctrlstat = ETH_QUARK_TDCS_TER;
  } else {
    descptr->ctrlstat = 0;
  }

  /* Initialize the descriptor */

  descptr->ctrlstat |=
    (ETH_QUARK_TDCS_OWN | /* The desc is owned by DMA */
     ETH_QUARK_TDCS_IC  | /* Interrupt after transfer */
     ETH_QUARK_TDCS_LS  | /* Last segment of packet   */
     ETH_QUARK_TDCS_FS);  /* First segment of packet  */

  /* Un-suspend DMA on the device */

  csrptr->tpdr = 1;

  return OK;
}
```

ethhandler.c

```
/*------------------------------------------------------------------------
 * ethhandler  -  Interrupt handler for Intel Quark Ethernet
 *------------------------------------------------------------------------
 */
interrupt ethhandler(void)
{
  struct  ethcblk *ethptr;  /* Ethertab entry pointer */
  struct  eth_q_csreg *csrptr;  /* Pointer to Ethernet CRSs */
  struct  eth_q_tx_desc *tdescptr;/* Pointer to tx descriptor */
  struct  eth_q_rx_desc *rdescptr;/* Pointer to rx descriptor */
  volatile uint32 sr;   /* Copy of status register  */
  uint32  count;      /* Variable used to count pkts  */

  ethptr = &ethertab[devtab[ETHER0].dvminor];

  csrptr = (struct eth_q_csreg *)ethptr->csr;

  /* Copy the status register into a local variable */

  sr = csrptr->sr;

  /* If there is no interrupt pending, return */

  if((csrptr->sr & ETH_QUARK_SR_NIS) == 0) {
    return;
  }
```

ethhandler.c

```
/* Acknowledge the interrupt */

csrptr->sr = sr;

/* Check status register to figure out the source of interrupt */

if (sr & ETH_QUARK_SR_TI) { /* Transmit interrupt */

  /* Pointer to the head of transmit desc ring */

  tdescptr = (struct eth_q_tx_desc *)ethptr->txRing +
            ethptr->txHead;

  /* Start packet count at zero */

  count = 0;

  /* Repeat until we process all the descriptor slots */

  while(ethptr->txHead != ethptr->txTail) {

    /* If the descriptor is owned by DMA, stop here */

    if(tdescptr->ctrlstat & ETH_QUARK_TDCS_OWN) {
      break;
    }
```

ethhandler.c

```
    /* Descriptor was processed; increment count  */

    count++;

    /* Go to the next descriptor */

    tdescptr += 1;

    /* Increment the head of the transmit desc ring */

    ethptr->txHead += 1;
    if(ethptr->txHead >= ethptr->txRingSize) {
      ethptr->txHead = 0;
      tdescptr = (struct eth_q_tx_desc *)
            ethptr->txRing;
    }
  }

  /* 'count' packets were processed by DMA, and slots are */
  /* now free; signal the semaphore accordingly   */

  signaln(ethptr->osem, count);

}
```

ethhandler.c

```
if(sr & ETH_QUARK_SR_RI) { /* Receive interrupt */

  /* Get the pointer to the tail of the receive desc list */

  rdescptr = (struct eth_q_rx_desc *)ethptr->rxRing +
            ethptr->rxTail;

  count = 0;  /* Start packet count at zero */

  /* Repeat until we have received     */
  /* maximum no. packets that can fit in queue  */

  while(count <= ethptr->rxRingSize) {

    /* If the descriptor is owned by the DMA, stop */

    if(rdescptr->status & ETH_QUARK_RDST_OWN) {
      break;
    }
```

ethhandler.c

```
      /* Descriptor was processed; increment count  */
      count++;

      /* Go to the next descriptor */

      rdescptr += 1;

      /* Increment the tail index of the rx desc ring */

      ethptr->rxTail += 1;
      if(ethptr->rxTail >= ethptr->rxRingSize) {
        ethptr->rxTail = 0;
        rdescptr = (struct eth_q_rx_desc *)
              ethptr->rxRing;
      }
    }

    /* 'count' packets were received and are available, */
    /*   so signal the semaphore accordingly     */

    signaln(ethptr->isem, count);
  }

  return;
}
```

ethhandler.c (BBB)

```
/*------------------------------------------------------------------------
 * ethhandler - TI AM335X Ethernet Interrupt Handler
 *------------------------------------------------------------------------
 */
interrupt ethhandler (
    uint32  xnum   /* IRQ number */
  )
{
  struct  eth_a_csreg *csrptr;     /* Ethernet CSR pointer */
  struct  eth_a_tx_desc *tdescptr;  /* Tx desc pointer   */
  struct  eth_a_rx_desc *rdescptr;  /* Rx desc pointer   */
  struct  ethcblk *ethptr = &ethertab[0]; /* Ethernet ctl blk ptr */

  csrptr = (struct eth_a_csreg *)ethptr->csr;

  if(xnum == ETH_AM335X_TXINT) {  /* Transmit interrupt */

    /* Get pointer to first desc in queue */

    tdescptr = (struct eth_a_tx_desc *)ethptr->txRing +
               ethptr->txHead;

    /* Defer scheduling until all descs are processed */

    resched_cntl(DEFER_START);
```

ethhandler.c (BBB)

```c
while(semcount(ethptr->osem) < (int32)ethptr->txRingSize) {

  /* If desc owned by DMA, check if we need to  */
  /* Restart the transmission     */

  if(tdescptr->stat & ETH_AM335X_TDS_OWN) {
    if(csrptr->stateram->tx_hdp[0] == 0) {
      csrptr->stateram->tx_hdp[0] =
          (uint32)tdescptr;
    }
    break;
  }

  /* Acknowledge the interrupt  */

  csrptr->stateram->tx_cp[0] = (uint32)tdescptr;

  /* Increment the head index of the queue  */
  /* And go to the next descriptor in queue */

  ethptr->txHead++;
  tdescptr++;
  if(ethptr->txHead >= ethptr->txRingSize) {
    ethptr->txHead = 0;
    tdescptr = (struct eth_a_tx_desc *)
          ethptr->txRing;
  }
```

ethhandler.c (BBB)

```
    /* Signal the output semaphore */

    signal(ethptr->osem);
  }

  /* Acknowledge the transmit interrupt */

  csrptr->cpdma->eoi_vector = 0x2;

  /* Resume rescheduling  */

  resched_cntl(DEFER_STOP);
}
```

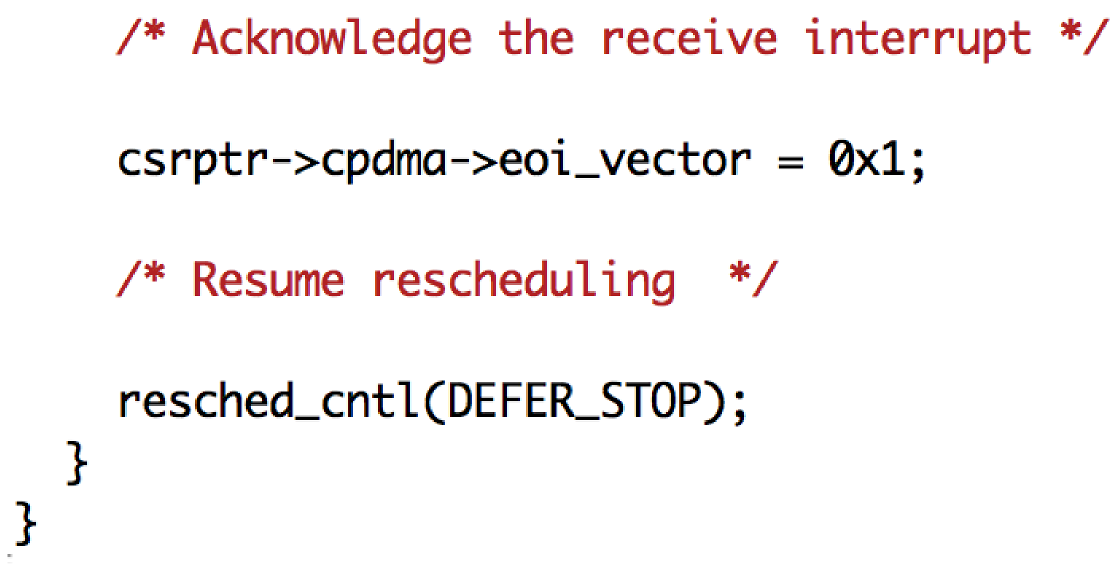
ethhandler.c (BBB)

```
else if(xnum == ETH_AM335X_RXINT) { /* Receive interrupt */

  /* Get the pointer to last desc in the queue  */

  rdescptr = (struct eth_a_rx_desc *)ethptr->rxRing +
          ethptr->rxTail;

  /* Defer scheduling until all descriptors are processed */

  resched_cntl(DEFER_START);

  while(semcount(ethptr->isem) < (int32)ethptr->rxRingSize) {

    /* Check if we need to restart the DMA  */

    if(rdescptr->stat & ETH_AM335X_RDS_OWN) {
      if(csrptr->stateram->rx_hdp[0] == 0) {
        csrptr->stateram->rx_hdp[0] =
          (uint32)rdescptr;
      }
      break;
    }
```

ethhandler.c (BBB)

```
    /* Acknowledge the interrupt  */

    csrptr->stateram->rx_cp[0] = (uint32)rdescptr;

    /* Increment the tail index of the queue  */
    /* And go to the next descriptor in the queue */

    ethptr->rxTail++;
    rdescptr++;
    if(ethptr->rxTail >= ethptr->rxRingSize) {
        ethptr->rxTail = 0;
        rdescptr = (struct eth_a_rx_desc *)
                ethptr->rxRing;
    }

    /* Signal the input semaphore */

    signal(ethptr->isem);
}
```

ethhandler.c (BBB)

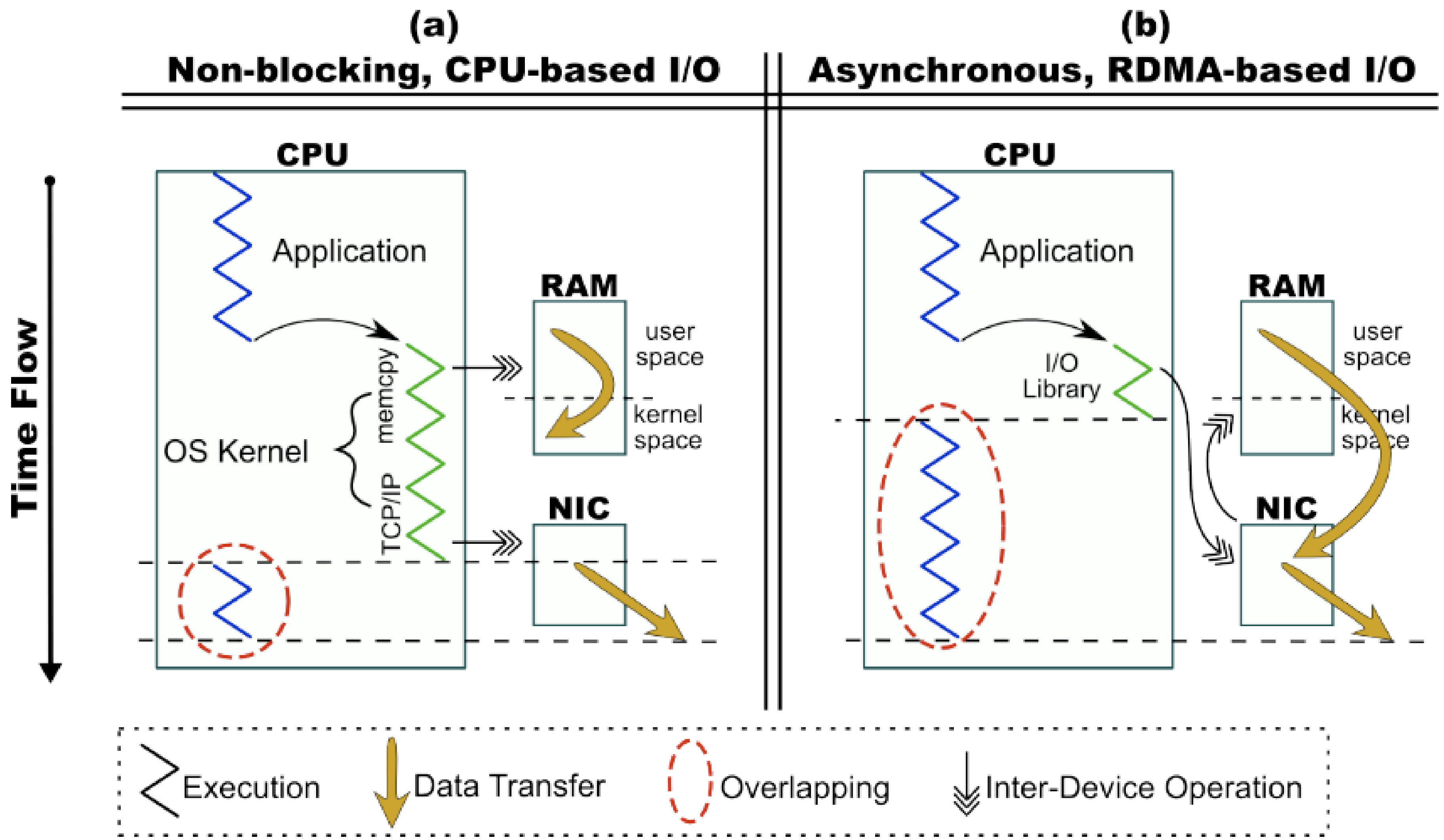


```
        /* Acknowledge the receive interrupt */

        csrptr->cpdma->eoi_vector = 0x1;

        /* Resume rescheduling  */

        resched_cntl(DEFER_STOP);
    }
}
```

Improving Performance

- Interrupt Coalescing
- Interrupt Affinity
- Large Send Offload
- Reduce Data Copying

Improving Performance



(a)
Non-blocking, CPU-based I/O
(b)
Asynchronous, RDMA-based I/O
Time Flow
CPU
Application
OS Kernel
memcpy
TCP/IP
RAM
user space
kernel space
NIC
CPU
Application
I/O Library
RAM
user space
kernel space
NIC
Execution
Data Transfer
Overlapping
Inter-Device Operation