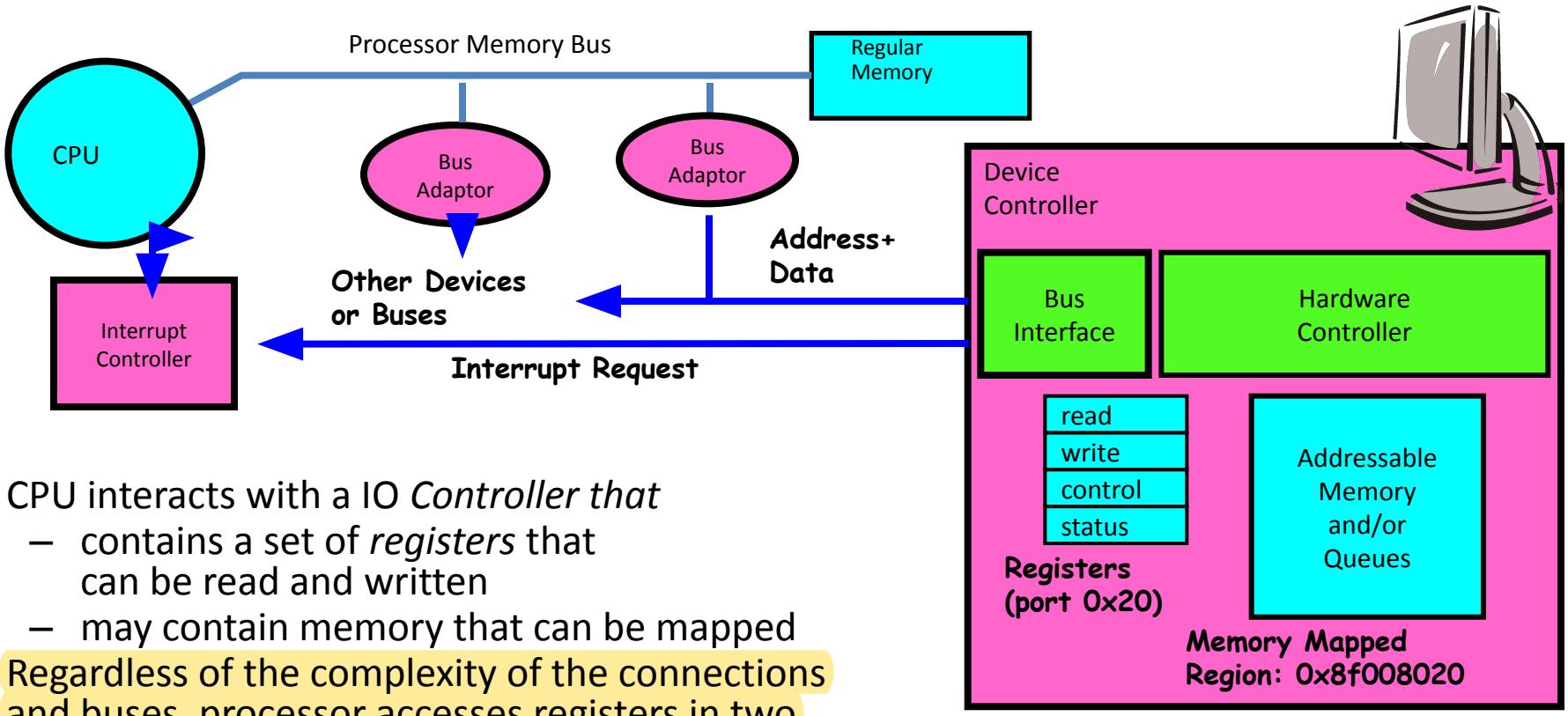
Module 4

Bus Independent IO and DMA

CPU and Device Interaction



- CPU interacts with a IO *Controller that*
  - contains a set of *registers* that can be read and written
  - may contain memory that can be mapped
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions
    - Intel architecture: `out 0x21,AL`
  - Memory mapped I/O: load/store instructions
    - Registers/memory mapped into physical address space
    - I/O accomplished to these registers via load and store instructions

Memory Addressing

Three ways of addressing physical memory in Linux:

- **CPU un-translated:** This is the "physical" address without going through MMU. Physical addresses are needed if you need to use memory mapping because some functions requires physical memory address.
- **CPU translated:** This is the "virtual" address mapped to the physical address via MMU, MMU (memory management unit) is part of cpu that translates physical addresses into Virtual Address. User applications, kernel and kernel drivers use virtual memory.
- **Bus address:** This is the address of memory as seen by devices, not the CPU. On x86/x64 architecture, the bus addresses are the same as the physical addresses considering the memory and devices share the same address space.

Getting physical address of virtual memory

- The kernel knows about physical memory and page tables
- unsigned long *virt_to_phys(void *addr)*
  - Given the virtual address addr, returns the backing location in physical memory
  - Not portable and full of other problems due to High Memory Area in x86. Works fine on x86_64 considering High Memory Area is empty.
- Physical devices uses physical addresses
- You need a ***virt_addr (bus_to_virt)*** when you need to get a pointer to buffer on the device memory. On the other hand, you need a ***bus_addr (virt_to_bus)*** when you have a buffer and needs to give it to hardware.
- Generally, you don't use ***phys_addr (virt_to_phys)*** because it cannot be used by cpu (use virtual addresses) or hardware (use bus addresses)

Device IO

- In Linux, device interface to the CPU is quite similar to memory. Typically, devices appear to the CPU as though they are memory devices.
- CPU transfers data to and from the devices. To output data to the device, the CPU simply stores data into a memory location and the data magically appears on the device connector that is mapped to this memory location.
- Similarly to input (receive) data from some external device, CPU simply transfers data from the memory location where the device connector or (device registers) are mapped into the CPU registers. This memory location normally holds the value on the pins of some external device connector.
- There are three basic forms of input and output that a typical computer system will use:
  - **Port Mapped I/O (PMIO):** I/O-mapped input/output uses special instructions to transfer data between the computer system and device
  - **Memory-mapped I/O (MMIO):** memory-mapped I/O uses special memory locations in the normal address space of the CPU to communicate with real-world devices
  - **Direct memory access (DMA):** DMA is a special form of memory-mapped I/O where the peripheral device reads and writes data in memory without going through the CPU.

Each I/O mechanism has its own set of advantages and disadvantages.

Port-mapped IO (PMIO)

- PMIO uses special CPU IN/OUT instructions to read and write 1-4bytes to an IO device.
  - Generally, a given peripheral device will use more than a single I/O port.
  - A typical PC parallel printer interface, uses three ports: a read/write port, an input port, and an output port. The read/write port is the data port .The input port returns control signals from the printer; these signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc. The output port transmits control information to the printer such as whether data is available to print.
- Linux allows device drivers to be written independent of the bus types by using APIs that abstracts IO operations across all buses.
  - Linux kernel has portable functions: ***in[b|w|l|sb|sl]()***, ***out[b|w|l|sb|sl]()*** for reading and writing to the IO ports
- PMIO is slower than MMIO.
  - Data transferred using PMIO is restricted specialized CPU register available to move data in/out of the device
  - MMIO can use all general purpose registers.
  - No 64-bit transfers available on PMIO
- PMIO Advantages:
  - No additional preparation is required to read/write to PMIO. In contrast, MMIO requires device memory be remapped first (using ***ioremap***) before IO can be performed to the device.
  - PMIO does not require device memory mapped and thus no kernel virtual address limitation on x86.

Linux init code disables caching on memory regions reserved for PMIO and MMIO. This to avoid reading a stale data from the cpu cache.

Memory-mapped IO (MMIO)

- Widely supported form of device IO. In this cpu's address space is interpreted not as accesses to memory, but as accesses to device.
- IO memory is not memory in ordinary sense, but rather ports or buffers mapped into reserved area that can only be used for device IO.
- Need to use **ioremap()** to map this into a virtual memory address for device driver to access. This function does not allocate memory, just setup page tables.
- Address returned by ***ioremap***() is not always directly usable by the CPU. Driver should not directly dereference the address, but rather use ***writeb|w|l/readb|w|l*** functions to access IO memory addresses
- MMIO is fast and supported by all modern hardware:
  - All cpu addressing modes are available for the I/O
  - Memory load and store instruction available for RAM can also be used for IO device registers
  - Support 64-bit transfers. AMD did not extend the PMIO instructions with x64 architecture.

Mapping Device Memory

- MMU doesn't know how to deal with device memory because it cannot use "bus addresses“.
  - At boot time, the page tables are only set up with an entry for each page of physical page (RAM). Thus to access device IO memory, we need some way to create additional page tables
- Conceptually, IO memory space is different than physical memory used by driver or kernel, so dereferencing a pointer should be avoided.
  - Exception: Dereferencing a pointer works on x86 platform because of IO space is part of the same memory address used by the kernel.
- In Linux there are utility functions available that allows device driver to map device memory to kernel or even user space:
  - ***ioremap*():**To create additional page tables for device IO memory range, *ioremap*() function is used. This function does not allocate any memory, instead it creates additional pages tables to provide access to device memory. Although, driver may reference this memory directly (x86/x64) to access device IO memory, it should be avoided for portability reasons. Once the device memory is mapped consider using bus independent IO routines: *writeb|w|l/readb|w|l*()
  - ***remap_pfn_range***() is mainly used to map device or kernel memory into user space. This function is used to implement driver's mmap method. Peripheral device memory and PCI IO memory regions can be mapped to user space with *remap_pfn_range()*.

Memory Mapped IO

MMIO routines:

ioreadb|w|l(), iowriteb|w|l()

void memset_io(void *addr, u8 value, unsigned int count);

void memcpy_fromio(void *dest, void *source, unsigned int count);

void memcpy_toio(void *dest, void *source, unsigned int count);

Memory Barriers

- Memory barriers are supplied to avoid reordering.
- Use Memory Barriers when operation must be visible to the hardware in a particular order.
- Memory Barrier functions insert hardware memory barriers in the compiled instruction flow
- This means load and stores preceding the memory barrier are committed to memory before any load and store following the memory barrier
- Barriers prevent compiler any optimizations that would have effect of reordering memory optimizations across the barriers
- Absence of barriers may cause device to malfunction.

Type	Detail
smp_mb()	order both load and store
smp_rmb()	order only load
smp_wmb()	order only stores

Reserving IO and Memory Region

Reserves specific *physical* address range for device I/O:

Request IO port (PMIO) region

struct resource *request_region(unsigned long start, unsigned long len, char *name);

Example: *request_region (0x0170, 8, "ide1");*

Driver can now access the device IO port by: *in[b|w|l|sb|sl]()* and *out[b|w|l|sb|sl]()*

Release IO port (PMIO) region:

void *release_region(unsigned long start, unsigned long len);*

Request IO mem (MMIO) region:

*struct resource *request_mem_region (unsigned long start, unsigned long len, char *name);*

Driver should call *ioremap*() before accessing device IO memory region by: *readb|w|l(), writeb|w|l(), memset_io(), memset_toio() and memset_fromio()*

Release IO mem (MMIO) region:

void *release_mem_region(unsigned long start, unsigned long len);*

What is DMA

- It is the capability to transfer data from a peripheral device to the main memory without the cpu intervention.
  - DMA capable devices generally have an interface to the CPU's bus so they can directly read and write memory without using the CPU as an intermediary
- It improves performance because it offload CPU from moving data
- PCI network cards and IDE/SCSI disks are examples of devices that use DMA for data transfer
- DMA is initiated by a DMA master. PC has a DMA controller that can master the IO bus for data transfer in Legacy ISA cards. PCI devices can master the bus and initiate DMA transfers on their own.
- Data travels directly between DMA controller and main memory, bypasses processor cache, that can cause processor to work with stale data in its cache. Hardware feature called bus **snooping** is used to keep the processor cache and main memory synchronized.

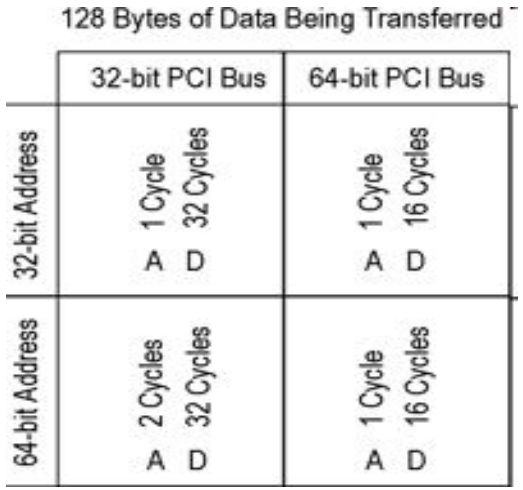
DMA Requirements

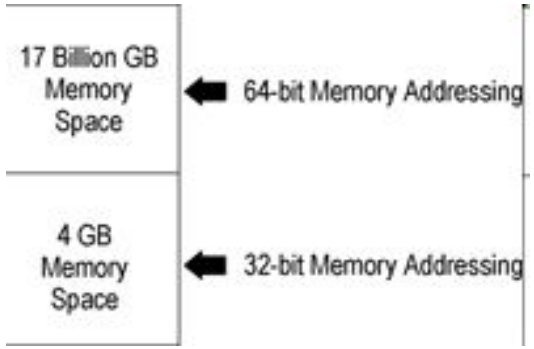
- DMA requires contiguous memory for source and destination of DMA transfers.
- DMA can be used on memory allocated using kmalloc (128k max) or __get_free_pages (8MB max).
- Block IO and skbuffs are also valid buffers to perform DMA
- vmalloc memory cannot be used because it is not physically contiguous.
- Addressing limit of bus interface would affect the memory range that can be used for DMA.
  - For example: DMA buffers suitable for 24-bit bus such as ISA can only live in the bottom 16MB of system memory, called ZONE_DMA. PCI buses are 32-64 bits and has no such limitation.

DMA
SAC and DAC

:

- Address and data transfers are multiplexed over the same lines on the PCI bus, the address is sent first and then the data.
- A 32-bit PCI bus has 32 data lines and can do 32-bit data transfers and both 32-bit and 64-bit memory addressing using one or two 32-bit PCI cycles (Dual Address Cycles or DAC)
- Two 32-bit cycles are used for DAC, with the first cycle sending the low address and the second cycle sending the high address
- PCI Burst transfer is one consisting of a single address phase followed by multiple data phases. Bus master only has to arbitrate for bus ownership one time. Both initiator and target should support burst mode





DMA

- To check if 32-bit or 64-bit DMA are supported on the platform use:

```
if (!(err=pci_set_dma_mask(pdev, DMA_64BIT_MASK))) {
   /* System supports 64-bit DMA */
     pci_using_dac=1;
} else {
   /* see if 32-bit DMA is supported */
 if ((err = pci_set_dma_mask(pdev, DMA_32BIT_MASK))){
     /* Not supported */
    printk("NO usable DMA configuration, aborting \n");
    return err;
}
  /* Using 32-bit DMA */
 pci_using_dac=0;
```

Consistent Or Coherent DMA

- This access method guarantees data coherency when DMA is performed considering both PCI device and CPU can operate on DMA buffer.
  - Trade-off is a degree of performance penalty
- These mappings usually exist for the life of the driver.
- A coherent buffer must be simultaneously available to both the CPU and the peripheral.
- Coherent mapping guarantees that both device and CPU can access data in parallel and able to see updates made by each other without requiring explicit software flushing.
  - Any cpu store to memory is immediately visible to the device, and vice versa.
- Coherent mappings can be bit expensive to set up and use
  - May require extra copy to/from DMA buffer to other kernel location
  - Coherent mappings has a long time span and thus can hold up device registers (used for DMA) even when not in use.

Consistent Or Coherent DMA (Cont.)

- Good examples of when to use consistent mappings for are:
  - Network card DMA ring descriptors.
  - SCSI adapter mailbox command data structures.
  - Device firmware microcode executed out of main memory.
- Allocate Consistent DMA buffer:

  void *vaddr = pci_alloc_consistent(pdev, size, &dma_handle);

  function generates a bus address pointed by dma_handle and returns associated kernel virtual address pointed by vaddr. First two arguments are PCI device structure and the size of the DMA buffer requested. dma_handle is the bus address that is given to device to perform DMA. Both addresses point to the same kernel memory location.
- Free consistent DMA buffer:

  DMA is typically allocated at device open and freed at close

  pci_free_consistent(pdev, size, vaddr, dma_handle);

Streaming DMA

- Since the cache-coherent mapping may be expensive and slow, performance sensitive devices use streaming DMA.
- Streaming DMA buffer is used for one-way communication, which means coherency is limited to flushing data from the cpu cache after a write finishes.
- It is required to have a buffer pre-allocated (e.g. kmalloc()) before buffer can be mapped (*dma_map_single*()) for DMA transfer.
- On DMA operation is completion:
  - Device notifies kernel for transfer completion or error via interrupt
  - Driver calls *dma_unmap_single*() to notify device
- Between map and unmap, the device is in control of the buffer.
  - Driver writes to DMA buffer before invoking *dma_map_single()*
  - Driver reads from DMA buffer after invoking *dma_unmap_single()*
- When the DMA mapping is active, only the device should access the buffer. Otherwise, potential cache inconsistency issues may result
- Thus driver should be accessing DMA buffer when it is not mapped.

Streaming DMA (cont.)

- Streaming DMA mappings are usually mapped for one DMA transfer operation, unmapped right after it
- Streaming DMA mapping allows platform to make performance optimizations in the hardware to avoid extra copy
- To Allocate single streaming DMA buffer, driver specify the intended direction of the buffer.

  dma_addr_t pci_map_single(struct pci_dev *pdev, void *ptr, size_t size, int direction);

  Where pdev represents pci device and ptr is the address of the kernel buffer already been allocated. Direction indicates if the buffer is for read or write.

  – *PCI_DMA_TODEVICE*
  – *PCI_DMA_FROMDEVICE*
- Good examples of when to use streaming mappings are:
  - Networking buffers transmitted/received by a device.
  - Filesystem buffers written/read by a SCSI device.

Scatter-Gather DMA

- Scatter/gather mappings are a special type of streaming DMA mapping that allows multiple buffers to be transferred to the device in one shot.
- This type of mapping allows the system to perform DMA operations on scattered buffers in memory.
  - This situation can come about in several ways, including from a *readv* or *writev* system call, a clustered disk I/O request, or a list of pages in a mapped kernel I/O buffer.
  - With scatter-gather operation one can map the whole list of buffers and can transferred them as one.
- Many devices can accept a *scatterlist* of array pointers and lengths, and transfer them in one DMA operation.
  - Advantages:
    - "zero-copy" networking is easier if packets can be built in multiple pieces.
    - IOMMU: IOMMU works similar to CPU MMU. IOMMU allows devices to see contiguous DMA buffer (transferred by driver) even though physical memory is scattered.
    - IOMMU requires mapping registers in the bus hardware, that allows physically discontiguous pages to be assembled into a single contiguous array from the device's point of view. This can turn multiple DMA operations into a single DMA, and speed things up considerably.

Scatter-Gather DMA (cont.)

- Buffer to be used for scatter/gather DMA operation is represented by an array of one or more *scatterlist* structures.
- Driver loop through list of buffer that need transfer and sets one *scatterlist* entry pointing to each buffer.
- Driver initialize the scatterlist entry with: *page, offset, and length* fields matching each buffer to be transferred.
- Once the scatterlist is setup, driver maps the whole list for DMA transfer in one shot:

  *int dma_map_sg(struct device *dev, struct scatterlist *sg, int* ***nents****, enum dma_data_direction direction)*

  where ***nents*** *is the number of scatterlist entries passed in. The return value is the number of DMA buffers to transfer; it may be less than nents due to coalescing of buffers performed by platform behind the scene when buffers are adjacent to each other.*

- If the hardware supports IOMMU, DMA to the device is done as a single contiguous buffer (faster).

Scatter-Gather DMA (cont.)

- *dma_map_sg* determines the proper bus address to give to the device. After mapping list of buffers (scatterlist) driver programs the hardware with bus addresses buffer length:
  dma_addr_t sg_dma_address(struct scatterlist *sg);
  int sg_dma_len(struct scatterlist *sg);
- Once the transfer is complete, a scatter/gather mapping is unmapped:

void dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents, enum dma_data_direction direction);

Note that *nents* must be the number of entries that you originally passed to dma_map_sg and not the number of buffers the function returned to you.

- Same access rules apply as for Streaming DMA. Before accessing a mapped scatter/gather list, driver must synchronize it first:
  - *void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);*
  - *void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);*

- scatterlist can be initialized using:
  *sg_init_table(struct scattlist *sg, unsigned int nents)*
  *sg_set_page(struct scatterlist *sg, struct page *page, unsigned int len, unsigned int offset);*
  *sg_set_buf(struct scattlist *sg, const void *buf, unsigned int buflen);*
  *sg_mark_end(struct scattlist *sg, unsigned int nents);*

DMA Comparison

Coherent mappings	Streaming mappings
Buffer can simultaneously be accessed by the cpu and device	Use a buffer already been allocated (e.g: kmalloc()) by the driver
Kernel allocates a suitable buffer and sets the mapping for the driver	Kernel just sets the mapping for a buffer provided by the driver
Can be expensive to setup and use	Can be optimized. Recommended!
Usually allocated when the device is opened and freed when device is closed or module unload. Easier!	Setup mapping for each transfer. Keep DMA registers free on the hardware. zero copy.

RTL8139 Network Card Programming

Hardware Registers

Register offsets are the memory addresses/locations in the device memory region. To read or write to a specific register, you need to provide a correct offset.

•Transmit descriptors contain status of transmit packet. There are four such descriptors: TSD0 – TSD3

#define TSD0

• Transmit descriptors contain physical address of a packet in memory. There are four such descriptor: TSAD0 – TSAD3

#define TSAD0 0x20

•Receive buffer start address. Points to the start of ring buffer (first packet)

#define RBSTART 0x30

• This register is used for sending different commands to chipset such as: Reset, Enable transmit and receive mechanism. Commands are discussed later.

#define CR 0x37

Device registers or memory offsets TSAD0-3 are updated by driver with DMA memory addresses used for transmitting packets

Device register or memory offset RBSTART is updated by driver with DMA memory address used for receiving packets

Hardware Registers (cont.)

•It contains buffer read pointer. It is the address of buffer that driver has read. Need to reset it after processing the receive packet to point to the next available space in ring buffer

#define CAPR 0x38

• This register is set to enable interrupts

#define IMR 0x3c

• This register is checked to get the interrupt status

#define ISR 0x3e

• Needs to initialize this register before transmit to support DMA burst size.

#define TCR 0x40

• Needs to be initialized before receive

#define RCR 0x44

Needs to initialize for missed packet counter

#define MPC 0x4c

•Initialize it to prevent early receive interrupts

#define MULINT 0x5c

Interrupt Status Register - ISR

Interrupt routine reads the ISR register to identify the type of interrupt:

```
#define RxOK 0x01    /* Packet receive successfully */
#define RxErr 0x02   /* Receive error */

#define TxOK 0x04    /* Packet transmitted successfully */
#define TxErr 0x08   /*  Transmit error */

/* Error conditions */
#define RxOverFlow 0x10
#define RxUnderrun 0x20
#define RxFIFOOver 0x40
#define CableLen 0x2000
#define TimeOut 0x4000
#define SysErr 0x8000
```

Interrupt Status Register - ISR

•Enable all interrupts by setting the mask and writing the mask to IMR register

#define **INT_MASK** (RxOK | RxErr | TxOK | TxErr | RxOverFlow | RxUnderrun | RxFIFOOver | CableLen | TimeOut | SysErr)

• Enable interrupts by ORing to value in IMR register. All requested interrupts are now enabled

writew(readw(ioaddr + IMR) | INT_MASK, ioaddr + IMR);

•Disable all interrupts by clearing the interrupt mask

writew(0x0000, ioaddr + IMR);

• Check the interrupt and save it in variable for later reference and then clear all interrupts by writing oxffff in the ISR register

isr = readw(ioaddr + ISR);

writew(0xffff, ioaddr + ISR);

ioaddr: address returned by ioremap()

Hardware Commands

These commands are used with TSD registers. TSD0-3 registers are used to check status of Transmit packets.

#define TxHostOwns 0x2000
#define TxUnderrun 0x4000
#define TxStatOK 0x8000
#define TxOutOfWindow 0x20000000
#define TxAborted 0x40000000
#define TxCarrierLost 0x80000000

Example:

```
txstatus = readl(ioaddr + TSD0)
 if(!(txstatus & (TxStatOK | TxAborted | TxUnderrun)
          break;
 if(txstatus & TxStatOK)
          printk("\n Packet is transmitted, TxStatOK bit is set\n");
```

Hardware Commands

These commands are used with CR registers. These commands enable transmit and receive mechanism or reset the chip:

```
#define RxBufEmpty 0x01
#define CmdTxEnb   0x04
#define CmdRxEnb   0x08
#define CmdReset   0x10
#define ChipCmdClear 0xe2
```

- Reset the chip

```
writeb(CmdReset, ioaddr + CR);
udelay (100);              /* wait for chip to finish the reset. */
for (i = 1000; i > 0; i--) {
  if ((readb(ioaddr + CR) & CmdReset) == 0)  /* Chip is reset successfully */
      break; }
```

- Enable Tx/Rx (Transmit and Receive)

```
writeb(CmdTxEnb | CmdRxEnb, ioaddr + CR);
```

Transmit Packets

- RTL8139 device uses 4 transmit descriptors:
  - **–TSD0-3 – Contains Status of the transmitted packet**
  - **–TSADO-3 – Contains physical location (bus address) in kernel memory (DMA buffer) where the packet is placed by driver when ready for transmit**
- These descriptors are located at a fixed offset in device memory:
  - **– First (TSD0) descriptor is at offset 0x10**
  - **– First (TSAD0) descriptor is at offset 0x20**

• Physical memory address in TSADO-3 should be contiguous address. It is the memory used for DMA transfer. TSAD0-3 contain dma bus addresses.

•DMA bus address is returned by DMA routine such as **dma_alloc_consistent**(). Bus address can be written to TSADO register to tell the device location of DMA buffer in memory.

writel(tx_bufs_dma, ioaddr + TSAD0);

Transmit Packets

Transmitting packet to hardware requires:

• DMA packet to physically contiguous memory that is referenced by bus address in TSAD0-3 (active transmit descriptor)

•Fill in the size of the packet in the status register ***TSD0-3***. That also clears the bit 13 "own", which triggers the xmit. "own" bit is set by the hardware when trasmit is completed.

•When the packet is transmitted successfully, interrupt is generated and TSD0-3 register is set to **TxStatOK.**

•Bits 0-12 (See Datasheet) of TSD0-3 contains total bytes in this descriptor. Masking 0x1fff (0-12) on txstatus gives transmit bytes

txstatus = readl(ioaddr + TSD0)

priv->stats.tx_bytes += (txstatus & 0x1fff);

• Clear the interrupt by setting the ISR register to 0xfff. Reading the ISR register should clear the interrupt, but some time it does not.

Transmit buffers

•Min/Max supported Ethernet frame size.

#define MAX_ETH_FRAME_SIZE 1536
#define ETH_MIN_LEN 60
#define TX_BUF_SIZE MAX_ETH_FRAME_SIZE

•Number of transmit Tx descriptor registers.

#define NUM_TX_DESC 4

•Total transmit buffer size is 4 x 1536

#define TOTAL_TX_BUF_SIZE (TX_BUF_SIZE * NUM_TX_DESC)

Receive Packets

- Receive path of the device is designed as ring buffer

- Ring buffer should be contiguous memory

- Similar to transmit, receive DMA bus address is written in the register RBSTART that contains receive buffer start address

  writel(priv->rx_ring_dma, ioaddr + RBSTART)

- Flush all receive interrupts before enabling device interrupts

  writew((readw(ioaddr + MULINT) & 0xF000), ioaddr + MULINT);

Receive Packets

- In the interrupt routine check ISR register. If set to **RxOK**, that means packet is received successfully.

•Receive status and packet length are saved in the beginning of the packet. It should be converted into host (big/little) endian byte order. First two bytes are received status and next two bytes are frame length.

rx_status = priv->rx_ring ;

rx_size = rx_status >> 16 # shift bits to right 16 times or divide by 2^16.

•Frame length also contain 4 CRC bytes (discard them). Therefore one can calculate receive packet size:

pkt_size = rx_size - 4;

•Allocate the sk_buff, copy the packet into it and call **netif_rx(sk_buff)** to dispatch packet to network or protocol stack for further processing

•Update the ring buffer pointer to next available buffer and update **CAPR** register that keeps track of data driver has read, start of next packet.

writew(priv->cur_rx, ioaddr + CAPR);

Receive buffers

Size of the receive ring. Where: 0==8K, 1==16K, 2==32K, 3==64K:

```
#define RX_BUF_LEN_IDX 2

#define RX_BUF_LEN     (8192 << RX_BUF_LEN_IDX)  # multiply by 2^2 => 32K
```

- See 11th and 12th bit of RCR (Receive Configuration Register) at offset 0x44

```
#define RX_BUF_PAD     16
```

- Spare padding to handle packet wrap (additional memory allocated)

```
#define RX_BUF_WRAP_PAD 2048

#define TOTAL_RX_BUF_SIZE  (RX_BUF_LEN+ RX_BUF_PAD+ RX_BUF_WRAP_PAD)
```

Device Initialization

- Reset the chip

- Enable Transmit and Receive

- Update transmit configuration register TCR to set the maximum DMA burst size (bits:8-16). Recommended value is 1024. Setting it to 6, sets the DMA burst size for transmit DMA to 1024 bytes

  writel(0x00000600, ioaddr + TCR);

- Write DMA bus addresses in registers TSAD0-3 and RBSTART for transmit and receive DMA respectively

- update receive configuration register RCR

- Initialize missed packet counter

- Flush receive interrupts to prevent early interrupts

- Enable all interrupts

MISC

PCIe NIC

RTL 8168/8169 PCI Express NIC Programming

Hardware Registers

Register offsets are the memory addresses/locations in the device memory region. To read or write to a specific register, you need to provide a correct offset.

Start address of Normal Priority Descriptor. Physical or dma_t address of Transmit Desc. It is 64-bit wide and 256 bytes aligned (See page 13-22, rtl8169 datasheet)

```
#define TNPDS                0x20
#define TNPDSLowBits         0x20     // Low 4-bytes offset TNPDS
#define  TNPDSHighBits       0x24     // High 4-bytes offset TNPDS
```

Start address of Receive Descriptor. Physical or dma_t address of Receive Desc. It is 64-bit wide and 256 bytes aligned, called RDS in datasheet.

```
#define   RDSAR              0xe4
#define   RDSARLowBits       0xe4     // Low 4-bytes offset RDSAR
#define   RDSARHighBits      0xe8     // High 4-bytes offset RDSAR
```

```
#define   CR          0x37     // Command Register to reset the device
#define   IMR         0x3c     // Set this to enable interrupts
#define   ISR         0x3e     // Interrupt status register
#define   TCR         0x40     // Transmit Configuration register for setting: DMA Burst, Inter Frame Gap, etc.
#define   RCR         0x44     // Receive Configuration register for setting: DMA Burst, promiscuous mode etc.
#define   MPC         0x4c     // Needs to initialize this missed packet counter
#define   MULINT      0x5c     // Initialize to avoid early receive interrupts
#define   IntrMitigate 0xe2    // NIC Interrupt Mitigation. Similar to NAPI but in hardware
…
```

Interrupt Status Register - ISR

Interrupt routine reads the ISR register to identify the type of interrupt:

```
#define RxOK 0x01          //Packet received successfully
#define RxErr 0x02         //Receive error
#define TxOK 0x04          //Packet transmitted successfully
#define TxErr 0x08         //Transmit error
```

Errors:

```
#define RxOverFlow 0x10
#define RxUnderrun 0x20
#define RxFIFOOver 0x40
#define CableLen 0x2000
#define TimeOut 0x4000
#define SysErr 0x8000
#define TxDescUnavail 0x0080
```

Interrupt Status Register - ISR

•Enable all interrupts by setting the mask and writing the mask to IMR register

#define INT_MASK (RxOK | RxErr | TxOK | TxErr | SysErr | RxOverFlow | RxFIFOOver| RxUnderrun | TimeOut | TxDescUnavail)

• Write the mask to IMR register. All interrupts are now enabled

iowrite32(INT_MASK, ioaddr + IMR);

•Disable all interrupts by clearing the interrupt mask

iowrite32(0, ioaddr + IMR);

*NOTE: **RxFIFOOver** applies to RTL8169/8101 but not RTL8168*

Hardware Commands

These commands are used with CR registers. These commands enable transmit and receive mechanism or reset the chip:

```
#define RxBufEmpty 0x01
#define CmdTxEnb   0x04
#define CmdRxEnb   0x08
#define CmdReset   0x10
#define StopReq    0x80
```

Reset the chip:

```
iowrite8(CmdReset, ioaddr + CR);
udelay (100);                 // wait for chip to finish the reset.
for (i = 1000; i > 0; i--) {
  if ((ioread8(ioaddr + CR) & CmdReset) == 0)  // Chip is reset successfully
      break; }
```

Enable Tx/Rx (Transmit and Receive)

```
writeb(CmdTxEnb | CmdRxEnb, ioaddr + CR);
```

Tx and Rx Descriptors

- RTL8169 Chip is descriptor based. There can be 1024 Descriptor available for Tx and Rx.
- Additional 1024 descriptors available for high priority Tx. It is not required to use all of them.
- Each Transmit and Receive descriptor is 16-byte wide:
  - First 4 bytes are used for sending command or reading status
  - Next 4 bytes are unused. It can be used for vlan tagging
  - Next 8 bytes are used for DMA address for Tx and Rx buffer. This address is used by NIC to read packets ready to transmit and write packets received over the wire

```
struct TxDesc {
    __le32 command;   // Use for setting command and getting status
    __le32 vlan;          // unused
    __le64 addr;        //DMA buffer address
};
struct RxDesc {
    __le32 command;   // Use for command and status
    __le32 vlan;     // unused
    __le64 addr;     // DMA buffer address
};
```

See Page 50-58, rtl8169 Datasheet

RTL8169 programming guide: http://www.singlix.com/trdos/runix/OSDev_Wiki/RTL8169.pdf

Setting up DMA Descriptors

- We need to allocate DMA buffers for both Tx and Rx descriptors

```
#define TOTAL_TX_BUF_SIZE    (NUM_TX_DESC * sizeof(struct TxDesc))
#define TOTAL_RX_BUF_SIZE    (NUM_RX_DESC * sizeof(struct RxDesc))

priv->tx_bufs = pci_alloc_consistent(priv->pci_dev, TOTAL_TX_BUF_SIZE, &priv->tx_bufs_dma);
priv->rx_bufs = pci_alloc_consistent(priv->pci_dev, TOTAL_RX_BUF_SIZE, &priv->rx_bufs_dma);
```

- Update NIC registers TNPDS and RDSAR with DMA address so that device can read and write to DMA buffers

```
iowrite32(TNPDSHighBits, ioaddr + ((u64) priv->tx_bufs_dma >> 32));
iowrite32(TNPDSLowBits, ioaddr + ((u64) priv->tx_bufs_dma) & DMA_BIT_MASK(32));
iowrite32(RDSARHighBits, ioaddr + ((u64) priv->rx_bufs_dma >> 32));
iowrite32(RDSARHighBits, ioaddr + ((u64) priv->rx_bufs_dma) & DMA_BIT_MASK(32));
```

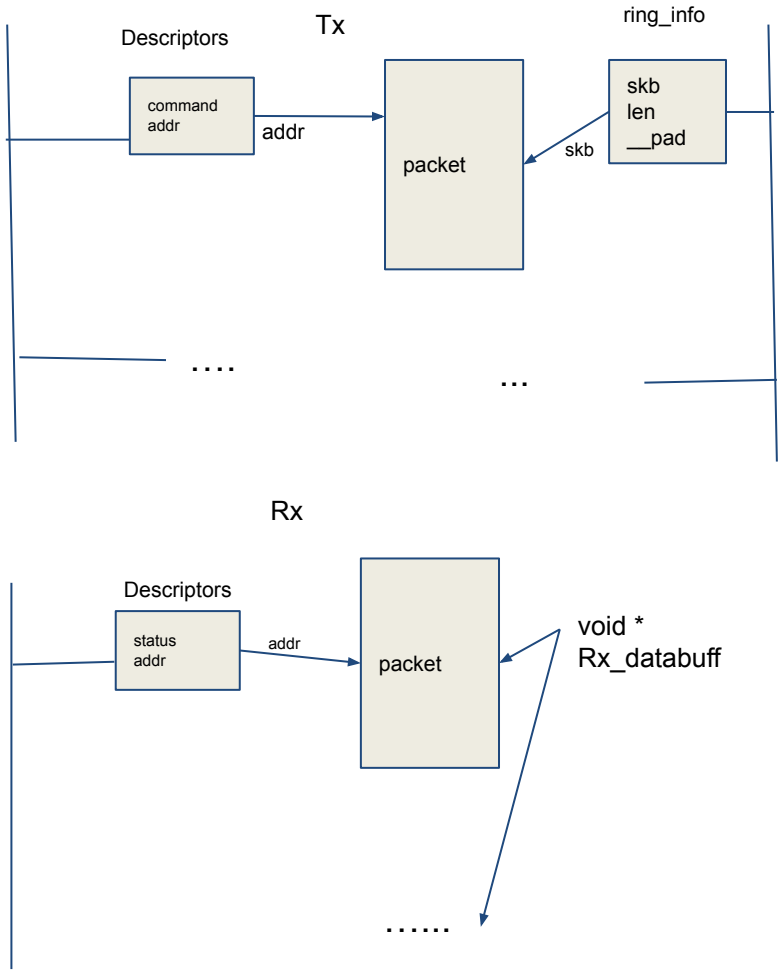
Tx and Rx DMA buffers

- Tx and Rx buffers are mapped to device via streaming DMA. Addresses of these buffers are saved into Rx and Tx descriptors.
- There can be several Tx buffers mapped at any given time, we need some data structure to keep track of Tx DMA buffers

```
struct ring_info {
      struct sk_buff *skb;
      u32    len;
      u8     __pad[sizeof(void *) - sizeof(u32)];
};
```

- Similarly to keep track of Rx buffers mapped at any given time, we us void ***Rx_databuff*** pointer
- Both ***ring_info*** and ***Rx_databuff*** are embedded inside the device private structure
- Initialize these structure to the size of maximum number of Descriptors

```
memset(tp->tx_skb, 0x0, NUM_TX_DESC * sizeof(struct ring_info));
memset(tp->Rx_databuff, 0x0, NUM_RX_DESC * sizeof(void *));
```



Rx DMA Buffers

- We need to allocate and map Rx DMA buffers of type ***Rx_databuff*** in advance so that device can dump the packet received over the wire to dma buffers.

```
for (i = 0; i < NUM_RX_DESC; i++) {
        void *data;
        RxDesc *desc = priv->rx_bufs + i;
        data = kmalloc(rx_buf_sz);
        (void *)ALIGN((long(data)), 16);  // align the buffer
         priv->Rx_databuff[i] = data;
         mapping = dma_map_single(priv->pci_dev->dev, rx_buf_sz, DMA_FROM_DEVICE);
         desc->addr = mapping;
         desc->command = (OWN | rx_buf_sz) ;
      }
/ Set the EOR bits in  last descriptor so that device know it is the end of descriptor list

      RxDesc lastdesc =  priv->rx_bufs + (NUM_RX_DESC - 1)
      lastdesc->command |= EOR;
```

NIC Transmit

- Linux protocol stack queues packets (sk_buf) by calling netdevice *start_xmit* routine.
- NIC driver keeps track of active transmit descriptors via *priv->cur_tx* counter. To know what descriptor to use, one can do

  unsigned int entry = priv->cur_tx % NUM_TX_DESC;
- To avoid extra copy and optimize performance, NIC driver uses streaming DMA.

  *dma_addr **mapping** = dma_map_single(d, skb->data, len, DMA_TO_DEVICE);*
- Update *ring_info* structure to keep track of active mapped DMA buffers to device

  priv->tx_skb[entry].len = len;

  priv->tx_skb[entry].skb = skb;
- Update transmit descriptor "*addr*" field to reference mapped dma buffer

  *txdesc->addr = cpu_to_le64(**mapping**);*
- Update transmit descriptor "*command*" field. Check if it is the last descriptor, then set EOR (End Of Ring) flag. Also write owner and packet (skb) length.

  *command = OWN | len | (EOR * !((entry + 1) % NUM_TX_DESC));*

  txdesc->command = cpu_to_le32(command);
- Notify NIC there is a packet waiting for transmit by writing to *TxPoll* register NPQ (0x40) value

  iowrite8(TxPoll, NPQ);

NIC Interrupt - Transmit

- Read *ISR* register to find the type of interrupt and then clear the ISR register

  isr = read16(ioaddr + ISR);

  iowrite16(0xffff, ioaddr + ISR);
- Check isr variable to see the type of interrupt:
  - For example: If (*isr & **TxOK**)* returns True, we received Transmit Interrupt etc..
- For TxOK interrupt:
  - Process all transmit descriptors. Break if the descriptor is still owned by device or any of the following conditions is true:
    - reached the current descriptor *"cur_tx"* descriptor
    - protocol stack is flow controlled

  while((priv->dirty_tx != priv->cur_tx) || netif_queue_stopped(netdev))
- At every iteration, we perform following tasks:
  - Unmap the dma buffer so that we can safely access it.

    dma_unmap_single(priv->pci_dev->dev, desc->addr, tx_skb->len, DMA_TO_DEVICE)
  - Update NIC stats

    priv->stats.tx_bytes += tx_skb->skb->len;

    priv->stats.tx_packets++;
  - initialize the tx descriptor and *ring_info* structure (tx_skb)

    desc->command = 0; desc->addr = 0; tx_skb->len = 0;
  - Update dirty_tx to next descriptor

    priv->dirty_tx = (priv->dirty_tx + 1) % NUM_TX_DESC;

NIC Interrupt - Receive

- For **RxOK** interrupt, we process all receive descriptors that have packet waiting. Similar to transmit, break if the descriptor is still owned by device, or we are done with all Rx descriptors or protocol stack is flow controlled

  while((priv->cur_rx < NUM_RX_DESC) || netif_queue_stopped(netdev))

- At Every iteration:
  - Check *"command"* field of rx descriptor to find the packet size. Also subtract 4 bytes CRC

    pkt_size = (rx_status & 0x00003fff) - 4
  - *Copy DMA buffer into skb. Receive DMA buffer is of type void *, we first need to align it to 16-bytes boundary.*

    *(void *)ALIGN((long(data)), 16);*
  - *Since we are copying DMA region that is still mapped to device, we first need to take ownership of the buffer to avoid any synchronization issues*

    dma_sync_single_for_cpu(priv->pci_dev->dev, addr, pkt_size, DMA_FROM_DEVICE)
  - Allocate skb buffer and copy the buffer into it.

    skb = dev_alloc_skb (pkt_size + 2);

    skb_reserve (skb, 2);

    memcpy(skb->data, data, pkt_size);

    skb_put(skb, pkt_size);

    skb->protocol = eth_type_trans(skb, netdev);

    netif_rx(skb); // hand skb to the protocol layer

NIC Interrupt - Receive cont.

Every iteration:

- Once copy is finished return the ownership back to Device

  dma_sync_single_for_device(priv->pci_dev->dev, addr, pkt_size, DMA_FROM_DEVICE);
- Update NIC stats

  priv->stats.rx_bytes += pkt_size;

  priv->stats.rx_packets++
- initialize the rx descriptor and increment the count priv->cur_rx.

  desc->command= (OWN | EOR| rx_buf_sz);

  priv->cur_rx++;