Lecture 23: More devices

Computer Architecture and
Systems Programming
(252-0061-00)

Timothy Roscoe

Herbstsemester 2012

Recall: I/O Devices

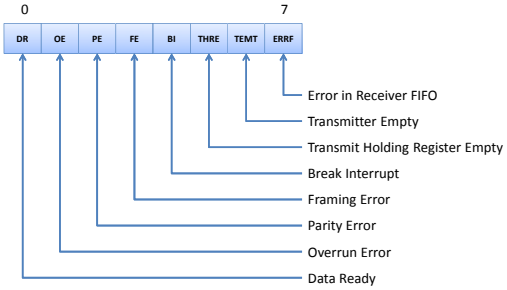
- What is a device?
- Registers
  - Example: NS16550 UART
- Interrupts
- Direct Memory Access (DMA)
- PCI (Peripheral Component Interconnect)
- Summary

ns16550 Registers (each 8 bits)

Addr.	Name	Description	Notes
0	RBR	Receive Buffer Register (read only)	DLAB=0
0	THR	Transmit Holding Register (write only)	DLAB=0
1	IER	Interrupt Enable Register	DLAB=0
2	IIR	Interrupt Identification Register (read only)	
2	FCR	FIFO Control Register (write only)	
3	LCR	Line Control Register	
4	MCR	MODEM Control Register	
5	LSR	Line Status Register	
6	MSR	MODEM Status Register	
7	SCR	Scratch Register	
0	DLL	Divisor Latch (LSB)	DLAB=1
1	DLM	Divisor Latch (MSB)	DLAB=1

DLAB = bit 7 of the LCR register

ns16550 LSR: Line Status Register



Very simple UART driver

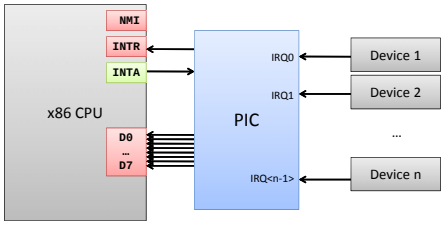
```
#define UART_BASE 0x3f8
#define UART_THR  (UART_BASE + 0)
#define UART_RBR  (UART_BASE + 0)
#define UART_LSR  (UART_BASE + 5)

void serial_putc(char c)
{
  // Wait until FIFO can hold more chars
  while( (inb(UART_LSR) & 0x20)== 0);
  // Write character to FIFO
  outb(UART_THR, c);
}

char serial_getc()
{
  // Wait until there is a char to read
  while( (inb(UART_LSR) & 0x01) == 0);
  // Read from the receive FIFO
  return inb(UART_RBR);
}
```

Register addresses from data sheet 0x3f8: location on a PC

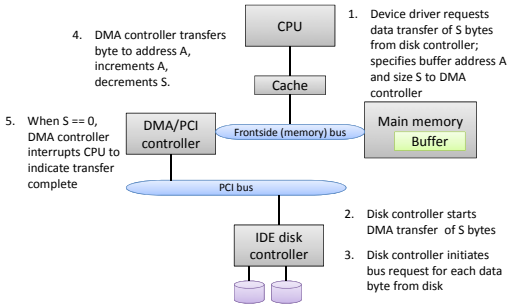
Send a character (wait until we can first)

Read a character (spin waiting until one is there to read)

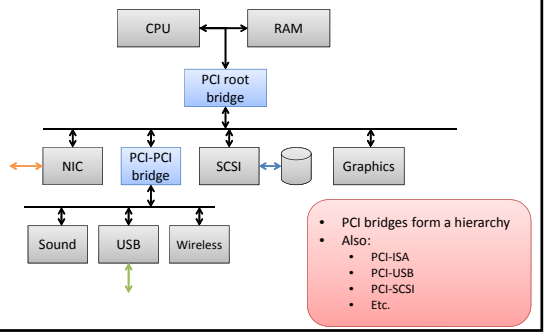
Programmable Interrupt Controllers



Old-style DMA transfer



Physical connections: PCI is a tree



Today: more complex devices

- Basic model: devices and device drivers
  - Software and hardware state machines
- Decoupling DMA and interrupts
  - Buffer rings
  - Descriptor rings
  - Descriptor protocols and states
- Example: DECchip 21140A "Tulip" Ethernet
  - Registers
  - Descriptors
  - Initialization
  - Send and receive state machines

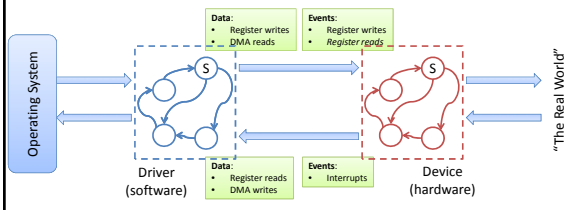
Today: more complex devices

- Basic model: devices and device drivers
  - Software and hardware state machines
- Decoupling DMA and interrupts
  - Buffer rings
  - Descriptor rings
  - Descriptor protocols and states
- Example: DECchip 21140A "Tulip" Ethernet
  - Registers
  - Descriptors
  - Initialization
  - Send and receive state machines

Evolution of device I/O

1. Programmed I/O (loads, stores).
   - Device state is polled by the processor
2. Polling is too slow (CPU cycles, response latency)
   ⇒ Interrupts notify CPU device needs attention
3. CPU spends too much time copying data
   ⇒ DMA allows CPU and device to operate in parallel
4. Too many interrupts (one per DMA)
   - CPU and device can't make much progress without resynchronizing
   ⇒ Use DMA for asynchronous buffering
5. Devices become complex enough to be "other processors"
   1. GPUs, NPUs, Channel Controllers, etc.

Basic model



- Driver and device are both *state machines*
- *Data* must be transferred between them
- *Events* signal state transitions

Device ↔ CPU communication

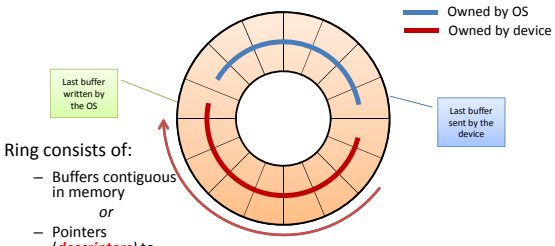
1. Writing a device register
   - CPU → device, synchronous
2. Reading a device register
   - CPU ↔ device, synchronous
3. Device requests interrupt
   - Device → CPU, synchronous
4. Shared memory
   - CPU writes to memory, DMA reads
   - DMA writes to memory, CPU reads
   - Asynchronous

Neither device nor software need to communicate simultaneously!

Today: more complex devices

- Basic model: devices and device drivers
  - Software and hardware state machines
- Decoupling DMA and interrupts
  - Buffer rings
  - Descriptor rings
  - Descriptor protocols and states
- Example: DECchip 21140A "Tulip" Ethernet
  - Registers
  - Descriptors
  - Initialization
  - Send and receive state machines

Buffer (or descriptor) rings (for transmit)

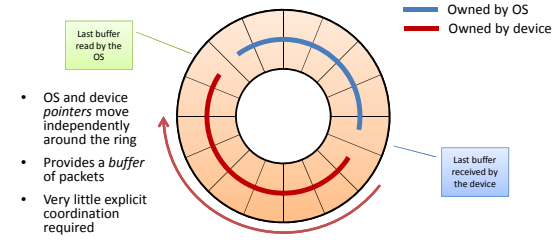


Ring consists of:
- Buffers contiguous in memory
  or
- Pointers (***descriptors***) to other bits of memory

Descriptors

- Most modern devices deal with *buffer descriptors*
  - Pointer to area(s) of memory – level of indirection
  - Other buffer *metadata*
- Advantages:
  - Allows software more flexibility in data placement
  - Buffers can be any size
  - Buffers can vary dynamically
  - Don't need to mix data and metadata

Buffer or descriptor rings (recieve)



- OS and device *pointers* move independently around the ring
- Provides a *buffer* of packets
- Very little explicit coordination required

What happens when one pointer catches up with the other?

Overruns and underruns (receive)

- Device has no buffers for received packets ⇒ starts discarding packets
  - Not as bad as it sounds
  - Will start copying them to memory when a buffer is free
  - Signals that it's lost some in a status register
- CPU reads all received packets ⇒ it must wait
  - Can spin polling, but inefficient
  - Signals device to interrupt it when a new packet has been received
  - Goes off to do something else

Overruns and underruns
(transmit)

- Device has no more packets to send ⇒ it must wait
  - Could continue to poll memory until next descriptor is owned by it
  - Could go to sleep and signal the software to wake it up
- CPU has no more slots to send packets ⇒ must wait
  - Can spin polling, but inefficient
  - Signals device to interrupt it when a packet has been sent i.e. a buffer slot is now free

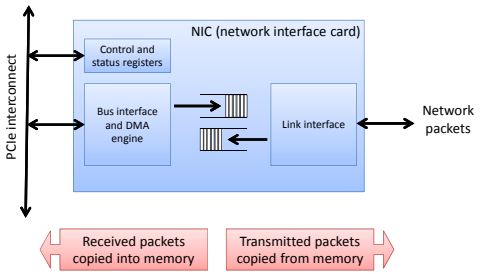
Observation: these are producer-consumer queues!

- Should be familiar from last year, except:
  - No mutexes or monitors available
  - No condition variables
  - No threads
- Instead, built using *messages*:
  - Register reads and writes
  - Interrupts

Today: more complex devices

- Basic model: devices and device drivers
  - Software and hardware state machines
- Decoupling DMA and interrupts
  - Buffer rings
  - Descriptor rings
  - Descriptor protocols and states
- Example: DECchip 21140A "Tulip" Ethernet
  - Registers
  - Descriptors
  - Initialization
  - Send and receive state machines

Network adaptors
(sneak preview of next semester)



Example: the DEC "Tulip" Fast Ethernet adaptor

digital

DIGITAL Semiconductor 21140A
PCI Fast Ethernet LAN Controller

Why? (it's old…)

- Very well documented:
  - datasheet on the web site
- Friendly card to write a driver for
- Illustrates *all the basic principles* of more complex devices

Registers (memory mapped)

Register	Description	Address offset
CSR0	Bus mode	0x00
CSR1	**Transmit poll demand**	0x08
CSR2	**Receive poll demand**	0x10
CSR3	**Receive list base address**	0x18
CSR4	**Transmit list base address**	0x28
CSR5	Status	0x28
CSR6	Operation mode	0x30
CSR7	**Interrupt enable**	0x38
CSR8	Missed frames and overflow counter	0x40
CSR9	Boot ROM, serial ROM, and MII management	0x48
CSR10	Boot ROM programming address	0x50
CSR11	General-purpose timer	0x58
CSR12	General-purpose port	0x60
CSR15	Watchdog timer	0x78

What about CSR13 and CSR14?

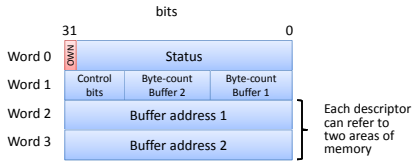
Table 3–23 CSR Mapping (Sheet 2 of 2)

Register	Meaning	Offset from CSR Base Address (CBIO and CBMA)
CSR11	General-purpose timer	58H
CSR12	General-purpose port	60H
CSR13	Reserved	68H
CSR14	Reserved	70H
CSR15	Watchdog timer	78H

Note: Writing to CSR14 may cause UNPREDICTABLE behavior.

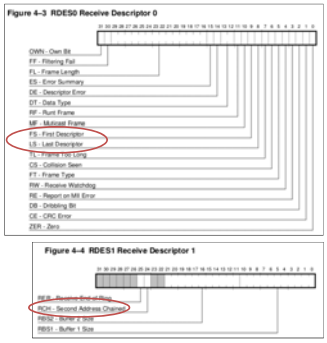
Literally anything can happen!
So why this register?

Tulip descriptors

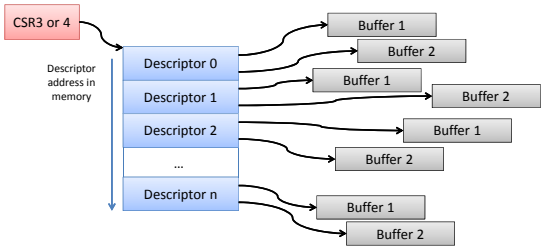


The Tulip has 2 rings of descriptors in main memory - One for transmit, one for receive

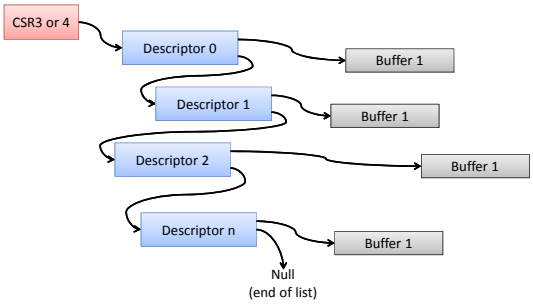
Gory details



Descriptor rings



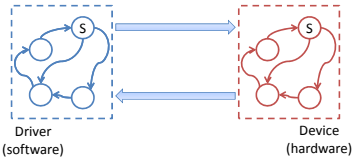
Descriptor rings – chain mode



Today: more complex devices

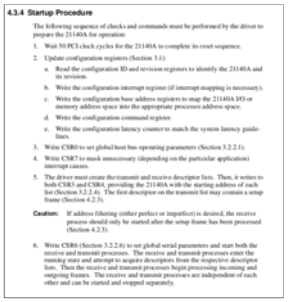
- Basic model: devices and device drivers
  - Software and hardware state machines
- Decoupling DMA and interrupts
  - Buffer rings
  - Descriptor rings
  - Descriptor protocols and states
- Example: DECchip 21140A “Tulip” Ethernet
  - Registers
  - Descriptors
  - Initialization
  - Send and receive state machines

Initialization



- Events ensure state transitions are synchronized
  - Register reads/writes, interrupts
- How to ensure common starting states?

Goal: put device in a "known state"



4.3.4 Startup Procedure

The following sequence of checks and commands must be performed by the driver to prepare the 21140A for operation:

1. Wait 50 PCI clock cycles for the 21140A to complete its reset sequence.
2. Update configuration registers (Section 3.1):
   a. Read the configuration ID and revision registers to identify the 21140A and its revision.
   b. Write the configuration interrupt register (if interrupt mapping is necessary).
   c. Write the configuration base address registers to map the 21140A I/O or memory address space into the appropriate processor address space.
   d. Write the configuration command register.
   e. Write the configuration latency counter to match the system latency guidelines.
3. Write CSR0 to set global host bus operating parameters (Section 3.2.2.1).
4. Write CSR7 to mask unnecessary (depending on the particular application) interrupt causes.
5. The driver must create the transmit and receive descriptor lists. Then, it writes to both CSR3 and CSR4, providing the 21140A with the starting address of each list (Section 3.2.2.4). The first descriptor on the transmit list may contain a setup frame (Section 4.2.3).

Caution: If address filtering (either perfect or imperfect) is desired, the receive process should only be started after the setup frame has been processed (Section 4.2.3).

6. Write CSR6 (Section 3.2.2.6) to set global serial parameters and start both the receive and transmit processes. The receive and transmit processes enter the running state and attempt to acquire descriptors from the respective descriptor lists. Then the receive and transmit processes begin processing incoming and outgoing frames. The receive and transmit processes are independent of each other and can be started and stopped separately.

What it all means for software

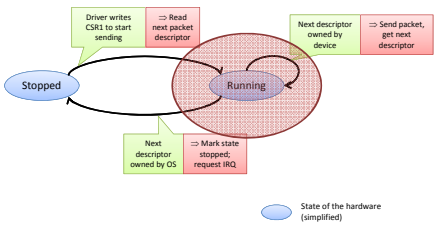
1. Wait for the hardware to settle down
2. Stop the device doing anything, just to be sure
   - No interrupts
   - No DMA
   - No sending packets
3. Create shared data structures
   - I.e. descriptor rings
   - Must tell device where they are!
4. Write registers to start device running

Today: more complex devices

- Basic model: devices and device drivers
  - Software and hardware state machines
- Decoupling DMA and interrupts
  - Buffer rings
  - Descriptor rings
  - Descriptor protocols and states
- Example: DECchip 21140A "Tulip" Ethernet
  - Registers
  - Descriptors
  - Initialization
  - Send and receive state machines

Sending packets

- Hardware (device) side:



DMA transactions

1. DMA Read: descriptor
2. If *descriptor.owner* == "OS" then enter state "stopped"
3. DMA Read: buffer
4. Send packet
5. DMA Write: *descriptor.owned* ← "OS"
6. Calculate next descriptor address
   - Next in memory (for unchained)
   - Value of buffer field 2 (for chained mode)
7. Goto 1.

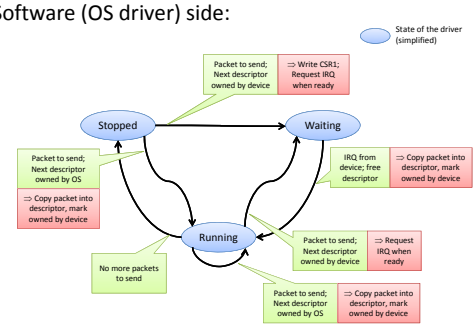
DMA transactions

1. DMA Read: descriptor
2. If *descriptor.owner* == "OS" then enter state "stopped"
3. DMA Read: buffer
4. Send packet
5. DMA Write: *descriptor.owned* ← "OS"
6. Calculate next descriptor address
   - Next in memory (for unchained)
   - Value of buffer field 2 (for chained mode)
7. Goto 1.

Key point:
DMA used for both:
1. Data transfer
2. Control information

Sending packets

- Software (OS driver) side:



Avoiding cache problems

- On x86 PC hardware, PCI-based DMA transfers are coherent with CPU caches ☺
- On anything else; they're not ☹

Hence:

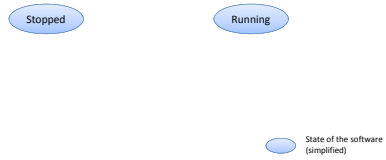
- DMA reads:
  Before: CPU should **flush** the cache for that address
  ⇒ main memory is up to date
  After: CPU should **invalidate** cache for that address
  ⇒ cache doesn't hold old value
- DMA writes:
  Before: CPU should **flush** or **invalidate** cache
  ⇒ No dirty lines left to write to memory
  After: CPU **invalidates** cache
  ⇒ Cache doesn't hold old value

Flush: write all dirty lines to memory

Invalidate: discard all lines from the cache

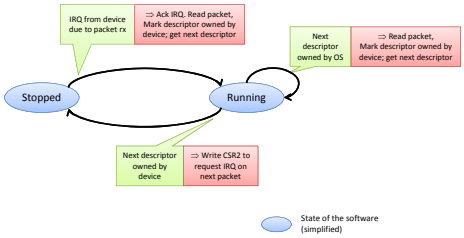
Receiving packets

- Software (OS) side:



Receiving packets

- Software (OS) side:



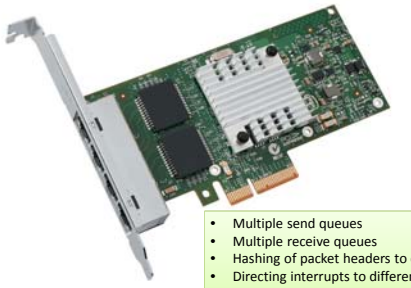
Observation: devices are more complex

- Can think of the descriptor lists as a *program*
  - List of instructions to perform
  - Tulip has two "processors" (transmit and recieve)
  - More sophisticated devices have more
- Programs can be more complex
  - Branching, conditionals
  - Calculations (checksums, offsets, etc.)
- Devices begin to look like other processors
  - *Frequently, they are!*

Intelligent devices

- Bus mastering, plus plenty of address space now
- Devices can now autonomously access any:
  - Main memory
  - Other devices
- Allows complex protocols for CPU↔Device interaction
  - Try to keep both CPU and device busy during high load
  - Extensive in-RAM buffering
  - "Descriptor rings" exchange requests and responses

Example: Intel e1000 PCI-Express Ethernet card



- Multiple send queues
- Multiple receive queues
- Hashing of packet headers to queues
- Directing interrupts to different cores
- Packet checksuming in hardware
- etc.

Next time: Multiprocessors

- Symmetric Multiprocessing (SMP)
- Consistency and Coherence
- Sequential Consistency
- Snoopy Caches