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)

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

ns16550 LSR: Line Status Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Error in Receiver FIFO</td>
</tr>
<tr>
<td>1</td>
<td>Transmitter Empty</td>
</tr>
<tr>
<td>2</td>
<td>Transmit Holding Register Empty</td>
</tr>
<tr>
<td>3</td>
<td>Break Interrupt</td>
</tr>
<tr>
<td>4</td>
<td>Framing Error</td>
</tr>
<tr>
<td>5</td>
<td>Parity Error</td>
</tr>
<tr>
<td>6</td>
<td>Overrun Error</td>
</tr>
<tr>
<td>7</td>
<td>Data Ready</td>
</tr>
</tbody>
</table>

Very simple UART driver

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

void serial_putchar(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_getchar()
{
    // Wait until there is a character to read
    while( (inb(UART_LSR) & 0x01) == 0);
    // Read from receive FIFO
    return inb(UART_RBR);
}
```

Programmable Interrupt Controllers

![Diagram of Programmable Interrupt Controllers]
Old-style DMA transfer

1. Device driver requests data transfer of 5 bytes from disk controller; specifies buffer address A and use S to DMA controller
2. Disk controller starts DMA transfer of 5 bytes
3. Disk controller initiates bus request for each data byte from disk
4. DMA controller transfers byte to address A, increments A, decrements S.
5. When S == 0, DMA controller interrupts CPU to indicate transfer complete

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, NIPUs, 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

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

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
  - 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

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)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Address offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR0</td>
<td>Bus mode</td>
<td>0x00</td>
</tr>
<tr>
<td>CSR1</td>
<td>Transmit poll demand</td>
<td>0x10</td>
</tr>
<tr>
<td>CSR2</td>
<td>Receive poll demand</td>
<td>0x10</td>
</tr>
<tr>
<td>CSR3</td>
<td>Receive list base address</td>
<td>0x28</td>
</tr>
<tr>
<td>CSR4</td>
<td>Transmit list base address</td>
<td>0x28</td>
</tr>
<tr>
<td>CSR5</td>
<td>Status</td>
<td>0x28</td>
</tr>
<tr>
<td>CSR6</td>
<td>Operation mode</td>
<td>0x30</td>
</tr>
<tr>
<td>CSR7</td>
<td>Interrupt enable</td>
<td>0x38</td>
</tr>
<tr>
<td>CSR8</td>
<td>Missed frames and overflow counter</td>
<td>0x40</td>
</tr>
<tr>
<td>CSR9</td>
<td>Boot ROM, serial ROM, and MH management</td>
<td>0x40</td>
</tr>
<tr>
<td>CSR10</td>
<td>Boot ROM programming address</td>
<td>0x50</td>
</tr>
<tr>
<td>CSR11</td>
<td>General-purpose timer</td>
<td>0x58</td>
</tr>
<tr>
<td>CSR12</td>
<td>General-purpose port</td>
<td>0x60</td>
</tr>
<tr>
<td>CSR15</td>
<td>Watchdog timer</td>
<td>0x78</td>
</tr>
</tbody>
</table>
What about CSR13 and CSR14?

Table S-23 CSR Mapping (Sheet 1 of 6)

<table>
<thead>
<tr>
<th>Register</th>
<th>Meaning</th>
<th>Offset from CSR Base Address (CSR0 and CSR4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR0</td>
<td>General purpose status</td>
<td>797B</td>
</tr>
<tr>
<td>CSR2</td>
<td>General purpose port</td>
<td>6984</td>
</tr>
<tr>
<td>CSR6</td>
<td>Reserved</td>
<td>6080</td>
</tr>
<tr>
<td>CSR8</td>
<td>Reserved</td>
<td>7084</td>
</tr>
<tr>
<td>CSR9</td>
<td>No accessing</td>
<td>7085</td>
</tr>
</tbody>
</table>

Note: Writing to CSR4 may cause UNPREDICTABLE behavior.

Literally anything can happen! So why this register?

Tulip descriptors

<table>
<thead>
<tr>
<th>Word 0</th>
<th>31</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Control bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte count</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>Buffer address 1</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>Buffer address 2</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>Buffer address 1</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>Buffer address 2</td>
</tr>
</tbody>
</table>

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

Gory details

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?

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

Goal: put device in a “known state”

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.owner ← “OS”
6. Calculate next descriptor address
   - Next in memory (for unchained)
   - Value of buffer field 2 (for chained mode)
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)

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

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

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 receive)
  - 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!

Sending packets

- Software (OS driver) side:

Receiving packets

- Software (OS) side:

Receiving packets

- Software (OS) side:
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
- Haching of packet headers to queues
- Directing interrupts to different cores
- Packet checksumming in hardware
- etc.

Next time:
Multiprocessors

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