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)

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

*DLAB = bit 7 of the LCR register*
ns16550 LSR:
Line Status Register

- DR: Data Ready
- OE: Overrun Error
- PE: Parity Error
- FE: Framing Error
- BI: Break Interrupt
- THRE: Transmit Holding Register Empty
- TEMT: Transmitter Empty
- ERRF: Error in Receiver FIFO

Data Ready
# 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 char to read
    while( (inb(UART_LSR) & 0x01) == 0);
    // Read from the receive FIFO
    return inb(UART_RBR);
}
```

- **Register addresses**
  - 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

x86 CPU

NMI

INTR

INTA

D0

...

D7

PIC

IRQ0

IRQ1

IRQ<n-1>

Device 1

Device 2

...

Device n
Old-style DMA transfer

1. Device driver requests data transfer of $S$ bytes from disk controller; specifies buffer address $A$ and size $S$ to DMA controller

2. Disk controller starts DMA transfer of $S$ 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

- PCI bridges form a hierarchy
- Also:
  - PCI-ISA
  - PCI-USB
  - PCI-SCSI
  - Etc.
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

- Last buffer written by the OS

- Last buffer sent by the device

Owned by OS

Owned by device
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 (receive)

- 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)

NIC (network interface card)

- Control and status registers
- Bus interface and DMA engine
- Link interface

Received packets copied into memory
Transmitted packets copied from memory

PCIe interconnect

Network packets
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>0x08</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>0x18</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 MII management</td>
<td>0x48</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 3–23 CSR Mapping](Sheet 2 of 2)

<table>
<thead>
<tr>
<th>Register</th>
<th>Meaning</th>
<th>Offset from CSR Base Address (CBIO and CBMA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR11</td>
<td>General-purpose timer</td>
<td>58H</td>
</tr>
<tr>
<td>CSR12</td>
<td>General-purpose port</td>
<td>60H</td>
</tr>
<tr>
<td>CSR13</td>
<td>Reserved</td>
<td>68H</td>
</tr>
<tr>
<td>CSR14</td>
<td>Reserved</td>
<td>70H</td>
</tr>
<tr>
<td>CSR15</td>
<td>Watchdog timer</td>
<td>78H</td>
</tr>
</tbody>
</table>

**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 0
- Descriptor 1
- Descriptor 2
- ... (continues)
- Descriptor n

CSR3 or 4

Descriptor address in memory

Buffer 1
- Buffer 2

Buffer 1

Buffer 1
- Buffer 2

Buffer 1

Buffer 1
- Buffer 2

Buffer 1

Buffer 1
- Buffer 2
Descriptor rings – chain mode

CSR3 or 4 → Descriptor 0 → Buffer 1
   ↘    ↗
   |    |  
   Descriptor 1 → Buffer 1
   ↘    ↗
   |    |  
   Descriptor 2 → Buffer 1
   ↘    ↗
   |    |  
   Descriptor n → Buffer 1
   ↘    ↗
   |    |  
   Null (end of list)
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:

- Stopped
  - Driver writes CSR1 to start sending
  - Read next packet descriptor
  - Next descriptor owned by OS
  - Mark state stopped; request IRQ

- Running
  - Next descriptor owned by device
  - Send packet, get next descriptor

State of the hardware (simplified)
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)
DMA transactions

1. DMA Read: descriptor
2. If \texttt{descriptor.owner} == “OS” then enter state “stopped”
3. DMA Read: buffer
4. Send packet
5. DMA Write: \texttt{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
Sending packets

- **Software (OS driver) side:**

  - **Stopped**
    - Packet to send; Next descriptor owned by OS
      - $\Rightarrow$ Copy packet into descriptor, mark owned by OS
  - **Waiting**
    - IRQ from device; free descriptor
      - $\Rightarrow$ Copy packet into descriptor, mark owned by device
  - **Running**
    - Packet to send; Next descriptor owned by device
      - $\Rightarrow$ Write CSR1; Request IRQ when ready
    - Packet to send; Next descriptor owned by OS
      - $\Rightarrow$ Copy packet into descriptor, mark owned by device
    - No more packets to send
      - $\Rightarrow$ Request IRQ when ready

  **State of the driver (simplified)**
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:

- IRQ from device due to packet rx
  ⇒ Ack IRQ. Read packet, Mark descriptor owned by device; get next descriptor

- Next descriptor owned by device
  ⇒ Write CSR2 to request IRQ on next packet

- Next descriptor owned by OS
  ⇒ Read packet, Mark descriptor owned by device; get next descriptor

State of the software (simplified)
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!
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