Lecture 18
Input/Output Devices

CS213 – Intro to Computer Systems

Branden Ghena – Fall 2023

Slides adapted from:
Hester, Tarzia (Northwestern), Bryant, O'Hallaron (CMU), Dutta, Garcia, Weaver (UC Berkeley), Venkataraman (Wisconsin), Singh (Princeton)

Administrivia

- Homework 4 due today!
  - Don’t forget about it
  - Good practice for midterm 2

- SETI Lab due on Thursday!
  - Beware, it’ll take quite a while to get feedback close to the deadline
  - Run `seti-eval` as sparingly as possible
  - It will give you very similar results to `seti-perf`

Common SETI Lab Errors

- Straight line performance
  - Often better than 1.02x right away and graph does not have a curve shape
  - Doesn’t vary thread count per the program argument
- Stuck at 0.3x
  - Usually didn’t optimize
  - Or maybe just optimized `p_band_scan.c` but not anything it relies on
- No Carrier Match
  - Your code output didn’t match the original `band_scan`
- No Alien Match
  - You didn’t correctly determine which of your generated signals is alien

Administrivia

- Midterm 2 next week Wednesday
  - 12:00-1:20 pm in this classroom (Tech Auditorium)

  - Allowed **two sheets of standard paper**, front and back, for notes
    - You may reuse your notes from last time as the first sheet if you want or you can make entirely new notes sheets

  - Material from weeks 5 and onwards
    - x86-64 Assembly Procedures through I/O & Networks (Thursday)
    - Homeworks 3 and 4
    - Bomb Lab, Attack Lab, and SETI Lab

Today’s Goals

- Introduce Input and Output (I/O) in computer systems

- Consider application-level I/O details

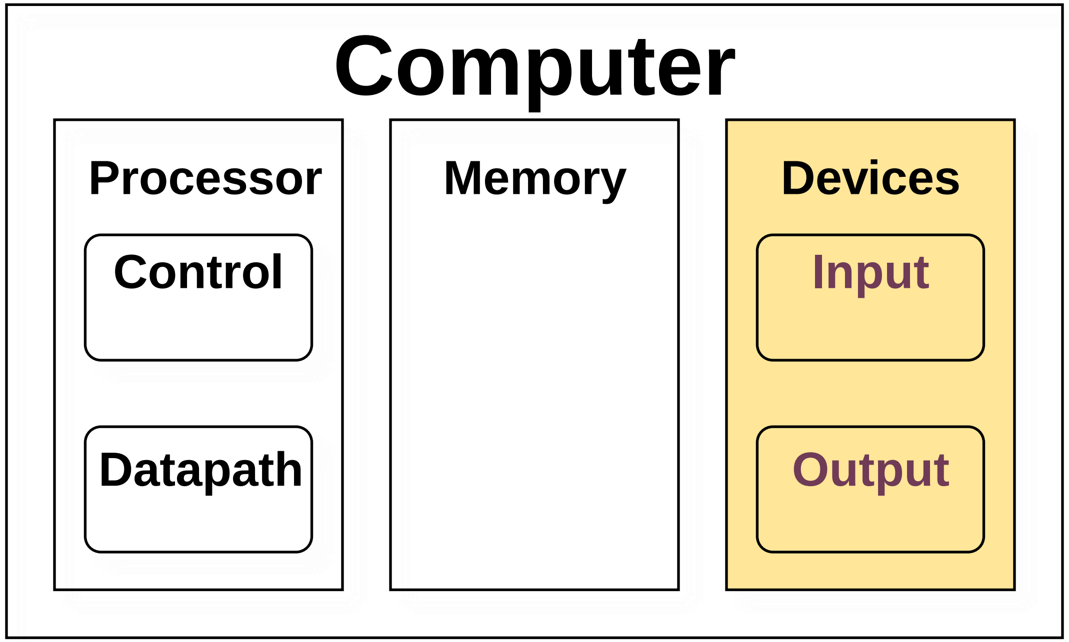
- Explore OS / microcontroller approaches to I/O
  - How to talk to devices
  - Interaction patterns with devices
  - Device drivers

Outline

- **Input/Output Motivation**
- Application-Level Input/Output
- Talking to Devices
  - MMIO Example
- Device Interaction Patterns
- Device Drivers

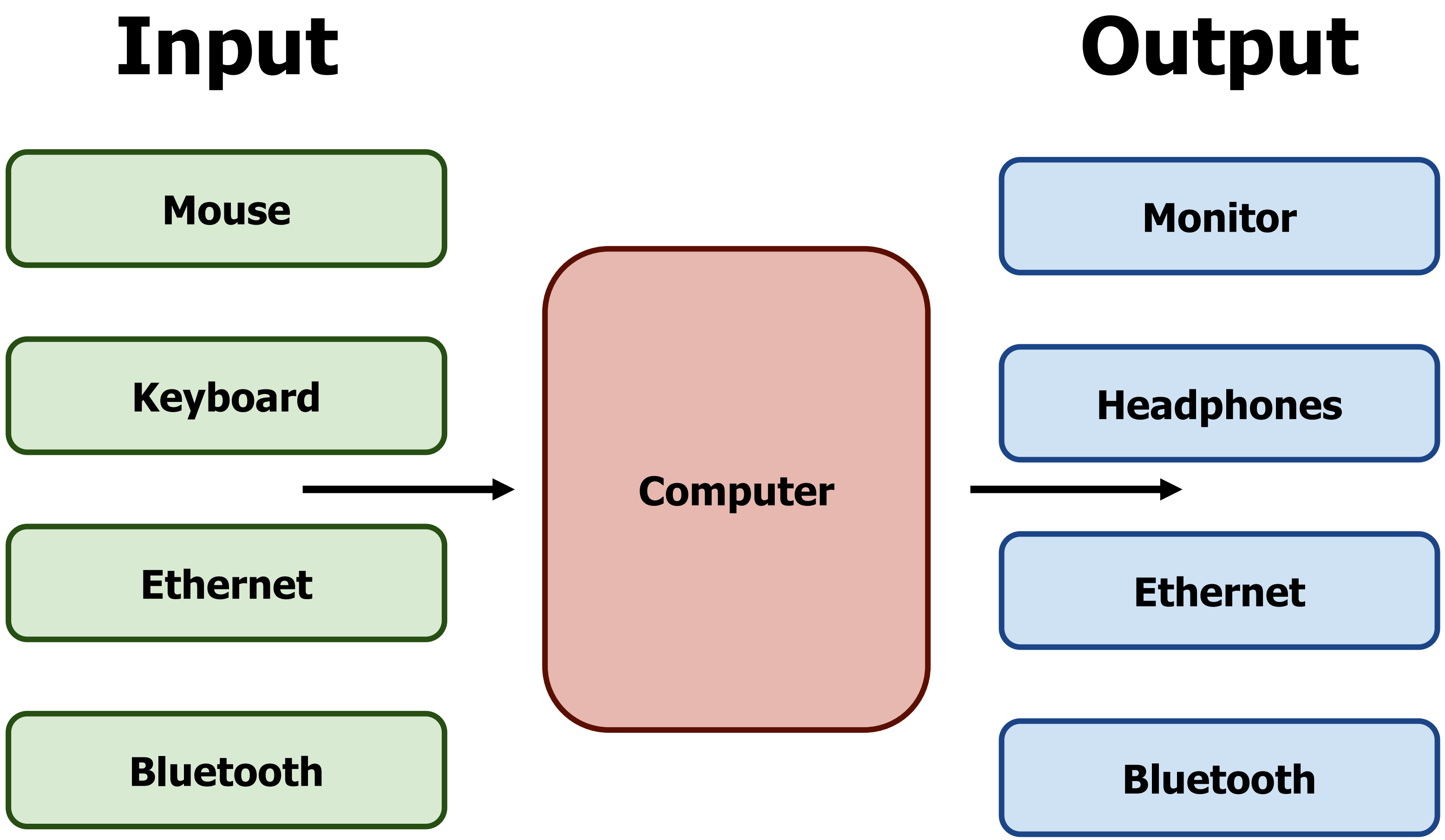
Devices are the point of computers

- Traditional systems need to receive input from users and output responses
  - Keyboard/mouse
  - Disk
  - Network
  - Graphics
  - Audio
  - Various USB devices

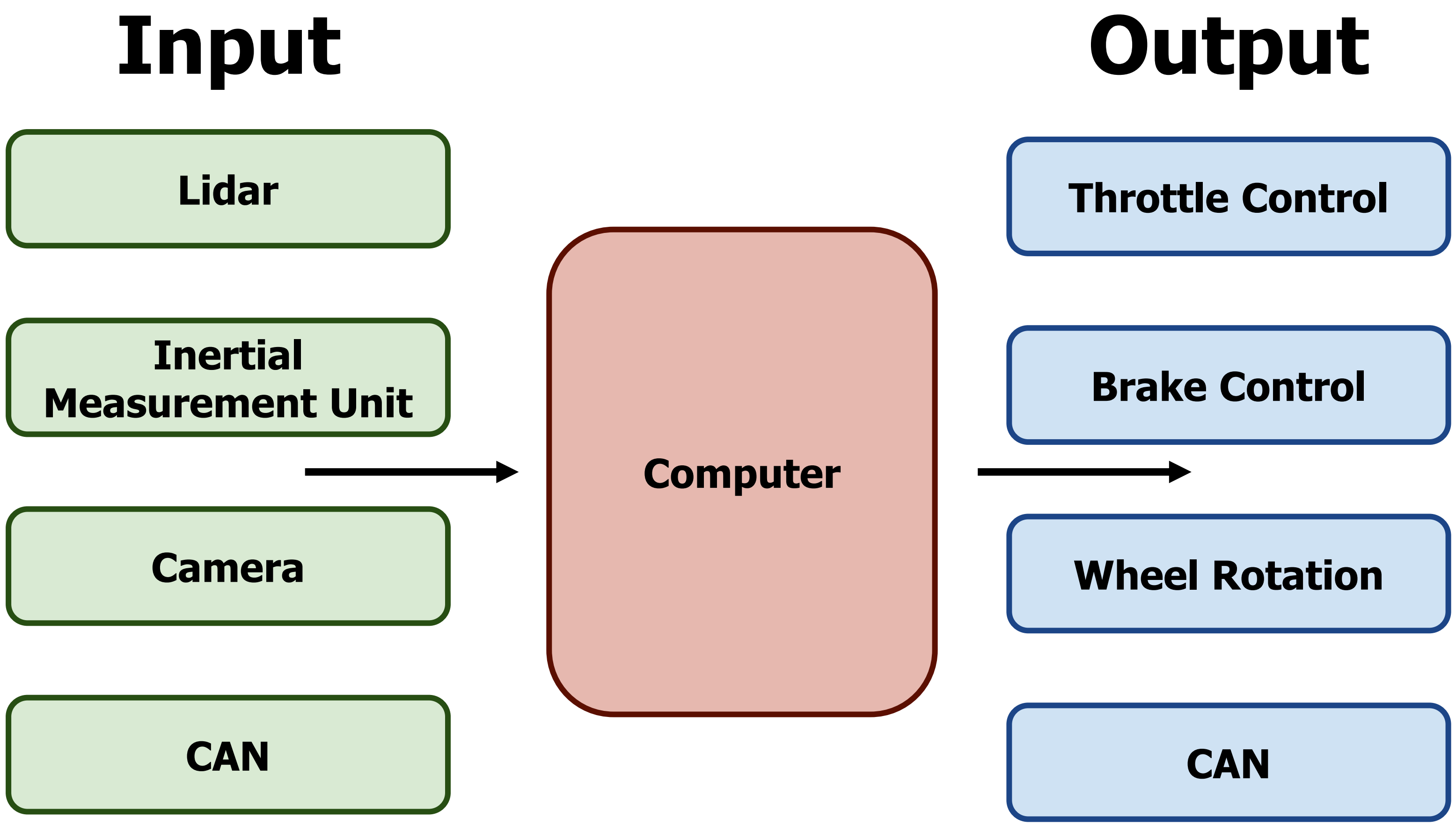


- Embedded systems have the same requirement, just more types of IO

Devices are core to useful general-purpose computing



Devices are essential to cyber-physical systems too



Device access rates vary by many orders of magnitude

- Rates in bit/sec
- System must be able to handle each of these
  - Sometimes needs low overhead
  - Sometimes needs to not wait around

Device	Behavior	Partner	Data Rate (Kb/s)
Keyboard	Input	Human	0.2
Mouse	Input	Human	0.4
Microphone	Output	Human	700.0
Bluetooth	Input or Output	Machine	20,000.0
Hard disk drive	Storage	Machine	100,000.0
Wireless network	Input or Output	Machine	300,000.0
Solid state drive	Storage	Machine	500,000.0
Wired LAN network	Input or Output	Machine	1,000,000.0
Graphics display	Output	Human	3,000,000.0

Outline

- Input/Output Motivation
- **Application-Level Input/Output**
- Talking to Devices
  - MMIO Example
- Device Interaction Patterns
- Device Drivers

Linux abstraction: everything is a file!

- Application-level: treat devices like files
  - They can be read and written
  - They may be created or destroyed (plugged/unplugged)

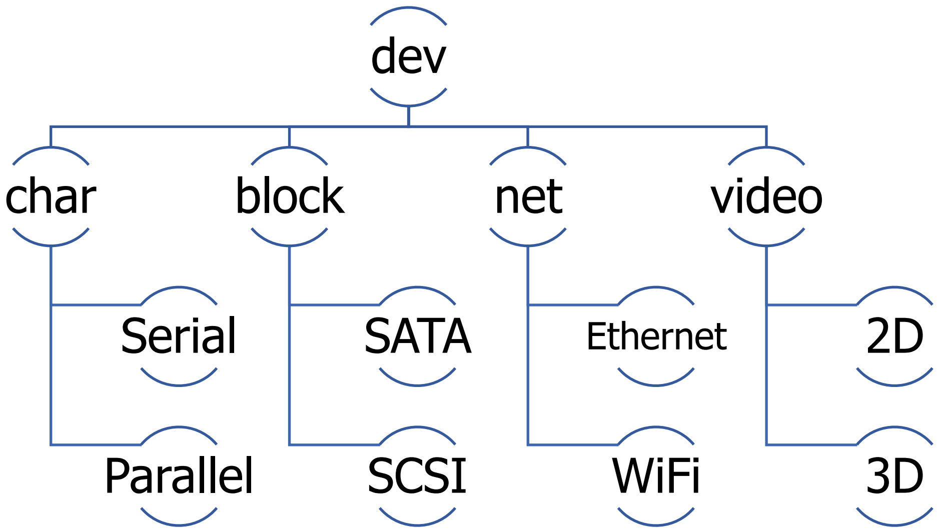
  - They can be created in hierarchies. Example:
    - SATA devices
      - SSD
    - USB devices
      - Webcam
      - Microphone

Linux device classes

- Character devices
  - Accessed as a stream of bytes (like a file)
  - Example: Webcam, Keyboard, Headphones

- Block devices
  - Accessed in blocks of data (like a disk)
  - Can hold entire filesystems
  - Example: Disks, Flash drives

- Network interfaces
  - See CS340 (Computer Networking)
  - Accessed through transfer of data packets



Communication with devices

- Must ask the OS to communicate with the device for us
  - This is a job for system calls!

- Which system calls? File I/O ones!
  - Open/Close
  - Read/Write
  - Seek, Flush
  - Ioctl
  - And various others

Accessing devices

- Open/Close
  - Inform device that something is using it (or not)
  - Argument is path to device (like path to file)
  - Get a file descriptor that the other operations act on

- "`/dev`" directory is populated with devices

```
[brghena@ubuntu code_examples] $ ls /dev/
agpgart          dri        lightnvm      mcelog  rtc0      tty0   tty22  tty36  tty5   tty63      ttyS18  ttyS31   vcs3   vcsu4
autofs           dvd        log           mem     sda       tty1   tty23  tty37  tty50  tty7       ttyS19  ttyS4    vcs4   vcsu5
block            ecryptfs   loop0         midi    sda1      tty10  tty24  tty38  tty51  tty8       ttyS2   ttyS5    vcs5   vcsu6
bsg              fb0        loop1         mqueue  sda2      tty11  tty25  tty39  tty52  tty9       ttyS20  ttyS6    vcs6   vfio
btrfs-control    fd         loop10        net     sda5      tty12  tty26  tty4   tty53  ttyprintk  ttyS21  ttyS7    vcsa   vga_arbiter
bus              full       loop2         null    sg0       tty13  tty27  tty40  tty54  ttyS0      ttyS22  ttyS8    vcsa1  vhci
cdrom            fuse       loop3         nvram   sg1       tty14  tty28  tty41  tty55  ttyS1      ttyS23  ttyS9    vcsa2  vhost-net
cdrw             hidraw0    loop4         port    shm       tty15  tty29  tty42  tty56  ttyS10     ttyS24  udmabuf  vcsa3  vhost-vsock
char             hpet       loop5         ppp     snapshot  tty16  tty3   tty43  tty57  ttyS11     ttyS25  uhid     vcsa4  vmci
console          hugepages  loop6         psaux   snd       tty17  tty30  tty44  tty58  ttyS12     ttyS26  uinput   vcsa5  vsock
core             hwrng      loop7         ptmx    sr0       tty18  tty31  tty45  tty59  ttyS13     ttyS27  urandom  vcsa6  zero
cpu_dma_latency  initctl    loop8         pts     stderr    tty19  tty32  tty46  tty6   ttyS14     ttyS28  userio   vcsu   zfs
cuse             input      loop9         random  stdin     tty2   tty33  tty47  tty60  ttyS15     ttyS29  vcs      vcsu1
disk             kmsg       loop-control  rfkill  stdout    tty20  tty34  tty48  tty61  ttyS16     ttyS3   vcs1     vcsu2
dmmidi           kvm        mapper        rtc     tty       tty21  tty35  tty49  tty62  ttyS17     ttyS30  vcs2     vcsu3
```

Interacting with devices

- Same read/write system calls you've seen before

- Read
  - `ssize_t read(int fd, void *buf, size_t count);`

- Write
  - `ssize_t write(int fd, const void *buf, size_t count);`

- Seek doesn't really make sense for most devices…

Arbitrary device interactions

- ioctl – I/O Control
  - `int ioctl(int fd, unsigned long request, ...);`

- Request number followed by an arbitrary list of arguments
  - "request" may be broken in fields: command, size, direction, etc.

- Catch-all for device operations that don't fit into file I/O model
  - Combine with "magic numbers" to form some special action
  - Reset device, Start action, Change setting, etc.
  - Read the device documentation to find these

ioctl example - sounds through console

```
void play_beep(unsigned int repeats, float frequency) {
  /* try to snag the console */
  int console_fd = open("/dev/console", O_WRONLY);

  for (unsigned int i=0; i<repeats; i++) {
    /* start beep */
    if(ioctl(console_fd, KIOCSOUND, (int)(CLOCK_TICK_RATE/frequency)) < 0) {
      perror("ioctl");
    }

    usleep(1000); /* wait... */

    ioctl(console_fd, KIOCSOUND, 0);  /* stop beep */
    usleep(1000); /* wait... */
  }

  close(console_fd);
}
```

Simplified from: http://www.johnath.com/beep/beep.c

Break + Open Question

- What are some downsides of “everything is a file”?

Break + Open Question

- What are some downsides of “everything is a file”?

  - Doesn’t allow devices to send data to us. Data needs to be requested

  - Slow turnaround time for detecting some action by a device
    - Like a mouse click

  - Not all devices map to files well
    - Microphone/Webcam work okay, just chunks of data to be read

    - Input devices not so much: keyboard, mouse, touchscreen
      - These are often handled directly by the OS instead

Outline

- Input/Output Motivation
- Application-Level Input/Output
- **Talking to Devices**
  - MMIO Example
- Device Interaction Patterns
- Device Drivers

Going deeper: how to talk to devices

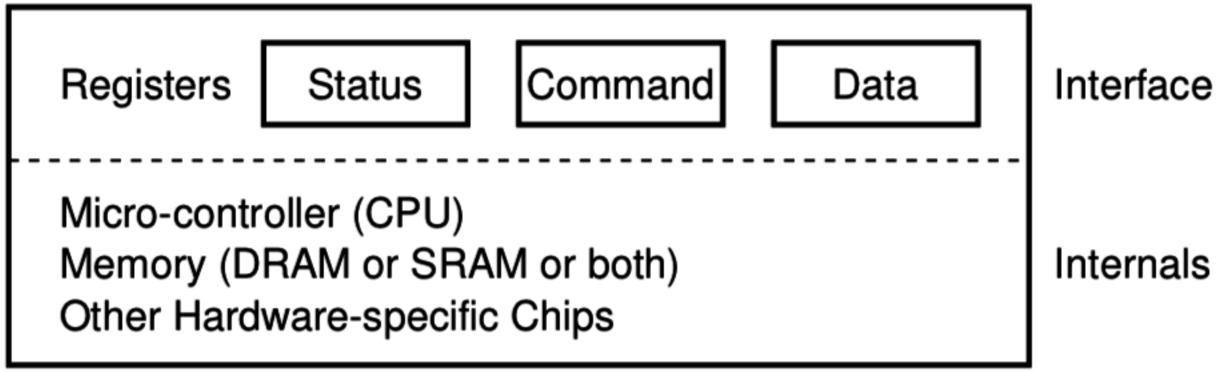
- What if you are writing the OS? How does it talk to devices?

- Or what if you don't have an OS at all and want to control devices directly?
  - Example: embedded systems

How to interact with I/O devices

- A device is really a miniature computer-within-the-computer
  - Has its own processing, memory, software

- We can mostly ignore that and deal with its interface
  - Called registers (actually are from EE perspective, but you can’t use them)
  - Read/Write like they’re data



Example powered device: Real Time Clock

- Battery-backed up clock on computer motherboard
- Keeps sense of time when computer is off
- Resynchronized when the computer is awake

Index	Contents	Range
0x00	Seconds	0-59
0x02	Minutes	0-59
0x04	Hours	0-23 in 24-hour mode, 1-12 in 12-hour mode, highest bit set if PM
0x06	Weekday	1-7, Sunday =1
0x07	Day of Month	1-31
0x08	Month	1-12
0x09	Year	0-99

Registers:

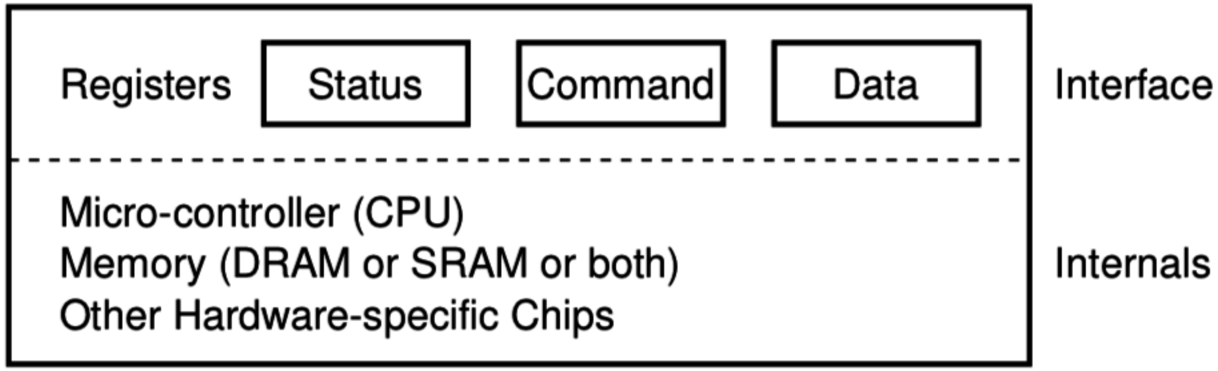
Index Register	Data Register

https://www.singlix.com/trdos/archive/pdf_archive/real-time-clock-nmi-enable-paper.pdf

Two options for reading/writing device registers

1. Special assembly instructions

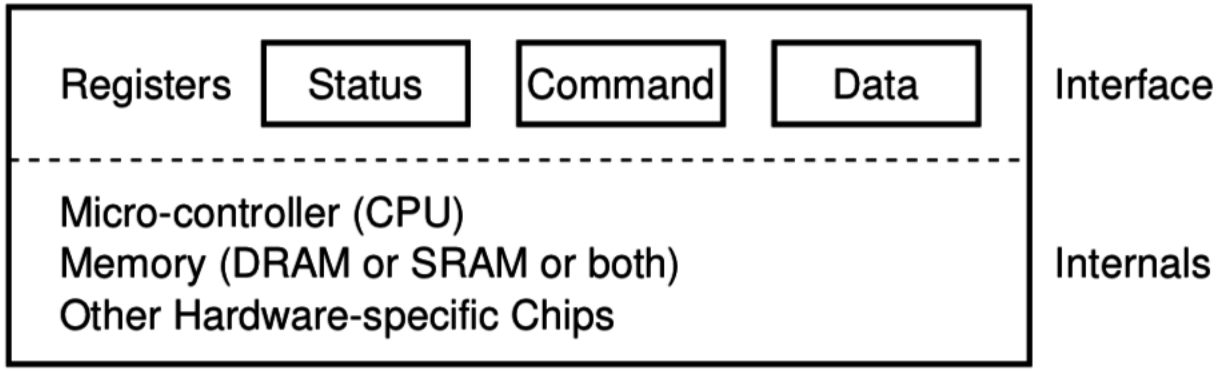
2. Treat like normal memory



Two options for reading/writing device registers

1. **Special assembly instructions**

2. Treat like normal memory



Port-Mapped I/O (PMIO): special assembly instructions

- x86 IN and OUT instructions
  - Privileged instructions (kernel mode only)
  - Two arguments: destination and data register

- Each device is mapped to some port address
  - IN and OUT instructions interact with interface

  - IN <PORT NUMBER>, <REGISTER>
  - OUT <REGISTER>, <PORT NUMBER>

Example powered device: Real Time Clock

- Example: read current value from real-time clock

```
// read seconds

mov $0, %al
out %al, $0x70
in $0x71, %al
```

Index	Contents	Range
0x00	Seconds	0-59
0x02	Minutes	0-59
0x04	Hours	0-23 in 24-hour mode, 1-12 in 12-hour mode, highest bit set if PM
0x06	Weekday	1-7, Sunday =1
0x07	Day of Month	1-31
0x08	Month	1-12
0x09	Year	0-99

Index Register	Data Register	
0x70	0x71	← Port Address

https://www.singlix.com/trdos/archive/pdf_archive/real-time-clock-nmi-enable-paper.pdf

Example I/O port map

This isn't standardized, but these are some typical values.

https://wiki.osdev.org/Can_I_have_a_list_of_IO_Ports

Port range	Summary
0x0000-0x001F	The first legacy DMA controller, often used for transfers to floppies.
0x0020-0x0021	The first Programmable Interrupt Controller
0x0022-0x0023	Access to the Model-Specific Registers of Cyrix processors.
0x0040-0x0047	The PIT (Programmable Interval Timer)
0x0060-0x0064	The "8042" PS/2 Controller or its predecessors, dealing with keyboards and mice.
0x0070-0x0071	The CMOS and RTC registers
0x0080-0x008F	The DMA (Page registers)
0x0092	The location of the fast A20 gate register
0x00A0-0x00A1	The second PIC
0x00C0-0x00DF	The second DMA controller, often used for soundblasters
0x00E9	Home of the Port E9 Hack. Used on some emulators to directly send text to the hosts' console.
0x0170-0x0177	The secondary ATA harddisk controller.
0x01F0-0x01F7	The primary ATA harddisk controller.
0x0278-0x027A	Parallel port
0x02F8-0x02FF	Second serial port
0x03B0-0x03DF	The range used for the IBM VGA, its direct predecessors, as well as any modern video card in legacy mode.
0x03F0-0x03F7	Floppy disk controller
0x03F8-0x03FF	First serial port

Check your understanding – PMIO in C

- How would you access PMIO from a C program?

Check your understanding – PMIO in C

- How would you access PMIO from a C program?
  - Need to use assembly!
  - Hopefully with C function wrapper, like System Calls

Annoying parts of Port-Mapped I/O

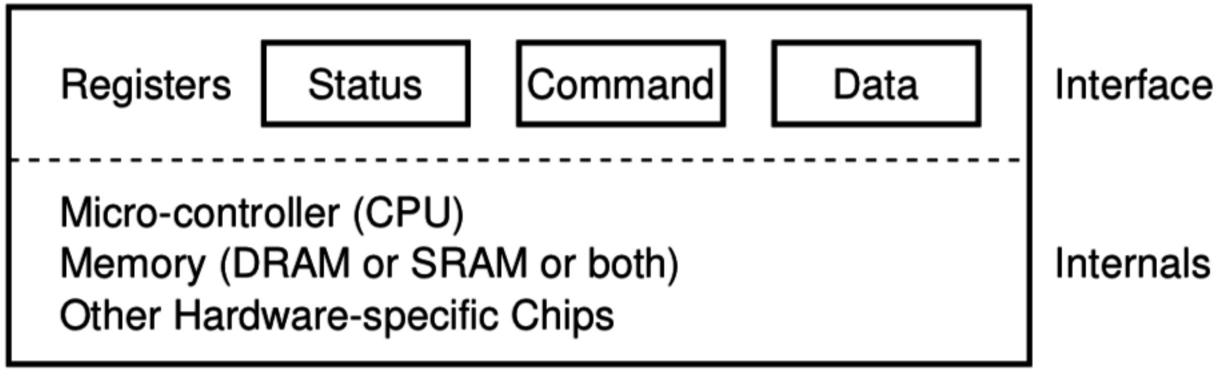
- Special assembly instructions are hard to write in C
  - Need some wrapper function that actually calls them
  - Not really that big of an issue, but a little weird

- Feels sort of like memory read/write, but isn't
  - Why not?
  - Can we just put the "port address space" somewhere in memory?
    - Could be a problem if we don't have enough memory
    - But today we have tons of extra physical address space laying around

Two options for reading/writing device registers

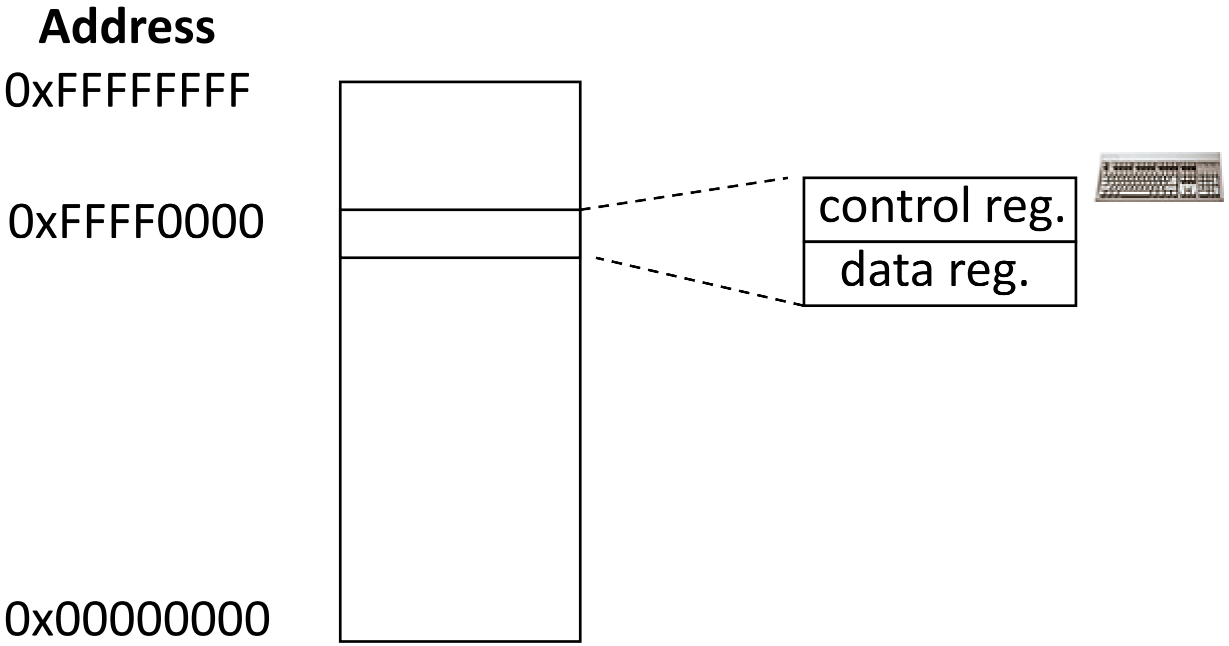
1. Special assembly instructions

2. Treat like normal memory



Memory-mapped I/O (MMIO): treat devices like normal memory

- Certain physical addresses do not actually go to RAM
- Instead, they correspond to I/O devices
  - And any instruction that accesses memory can access them too!



- x86-64 being the historical amalgamation that it is, uses both PMIO or MMIO depending on the device

Other details about MMIO

- Devices are mapped into physical memory
  - Usually only accessible by the operating system
  - But could be directly placed in virtual memory for a process in very special cases

- Devices are NOT memory though
  - Need to be careful not to cache them
    - Values being read could change, or reading could have an effect

  - Cannot let compiler mess with our reads/writes either
    - *volatile* keyword in C

- Conceptually not really very different from PMIO
  - Both just read/write to specific addresses the device is mapped to

Outline

- Input/Output Motivation
- Application-Level Input/Output
- **Talking to Devices**
  - **MMIO Example**
- Device Interaction Patterns
- Device Drivers

Simpler Memory-Mapped IO example: a microcontroller

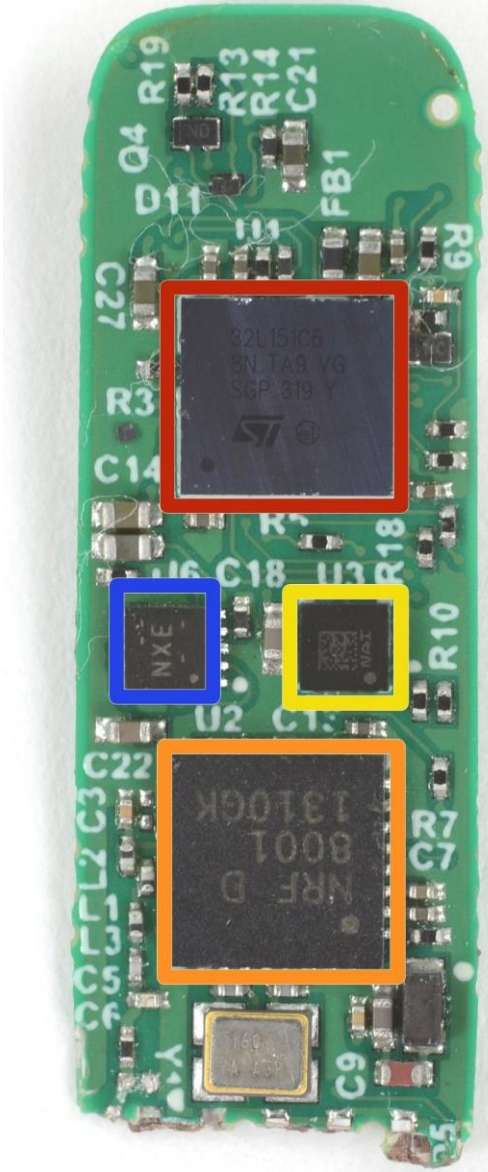
- A microcontroller is a single chip with a processor and memory
  - Used for simple devices such as embedded systems
  - Example use case: a Fitbit

- Fitbit flex (circa 2013)

- Features
  - Counts your steps
  - Reports via Bluetooth Low Energy
  - Lights up some LEDs based on your goals
  - Vibrates when its battery is low

Fitbit teardown

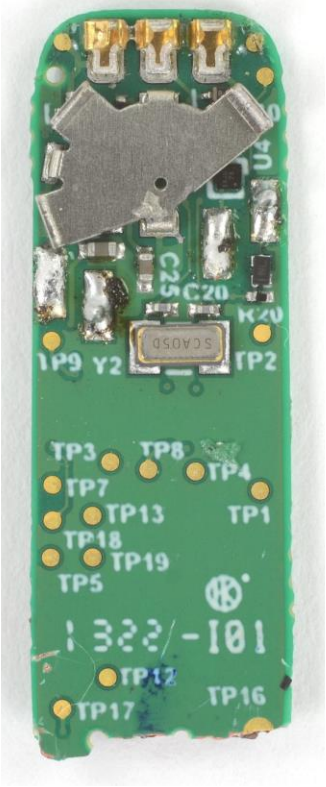
https://www.ifixit.com/Teardown/Fitbit+Flex+Teardown/16050



Fitbit circuit board front

- Red (top)
  - STMicro 32L151C6 Microcontroller
- Blue (left)
  - TI BQ24040 Battery Charger
- Yellow (right)
  - STMicro LIS2DH Accelerometer
- Orange (bottom)
  - Nordic nRF8001 Bluetooth Low Energy Radio

The back is uninteresting



Fitbit as a computer

- Computers *usually* need
  - Processor
  - Memory (RAM)
  - Storage (Flash/SSD)
  - External communication
    - USB, Thunderbolt, SATA, HDMI, WiFi
  - Power management
    - Maybe batteries and charging
  - Something to connect it all: motherboard

- Microcontroller
- Bluetooth radio
- Vibratory motor
- Battery and power management
- Circuit board

Example microcontroller memory map

- 0x1000 bytes is plenty of space for each device (a.k.a. peripheral)
  - 1024 registers, each 32 bits
  - No reason to pack them tighter than that

nRF52833 microcontroller

5	0x40005000	NFCT	NFCT	Near field communication tag
6	0x40006000	GPIOTE	GPIOTE	GPIO tasks and events
7	0x40007000	SAADC	SAADC	Analog to digital converter
8	0x40008000	TIMER	TIMER0	Timer 0
9	0x40009000	TIMER	TIMER1	Timer 1
10	0x4000A000	TIMER	TIMER2	Timer 2
11	0x4000B000	RTC	RTC0	Real-time counter 0
12	0x4000C000	TEMP	TEMP	Temperature sensor
13	0x4000D000	RNG	RNG	Random number generator
14	0x4000E000	ECB	ECB	AES electronic code book (ECB) mode block encryption
15	0x4000F000	AAR	AAR	Accelerated address resolver

Example: TEMP device on nRF52833 microcontroller

- Internal temperature sensor
  - 0.25° C resolution
  - Range equivalent to microcontroller chip (-40° to 105° C)
  - Various configurations for the temperature conversion (ignoring)

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

Table 110: Instances

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

MMIO addresses for TEMP

- What addresses do we need? (ignore interrupts for now)
  - 0x4000C000 – TASKS_START
  - 0x4000C100 – EVENTS_DATARDY
  - 0x4000C508 - TEMP

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

Table 110: Instances

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

Accessing addresses in C

- What does this C code do?

```
*(uint32_t*)(0x4000C000) = 1;
```

Accessing addresses in C

- What does this C code do?

```
*(uint32_t*)(0x4000C000) = 1;
```

  - 0x4000C000 is cast to an `uint32_t*` (a 32-bit unsigned integer pointer)
  - Then dereferenced
  - And we write 1 to it

  - "There are 32-bits of memory at 0x4000C000. Write a 1 there."

Example code

- To the terminal!

- Let’s write it from scratch

Example code (`temp_mmio` app)

```
// loop forever
while (1) {

  // start a measurement
  *(uint32_t*)(0x4000C000) = 1;

  // wait until ready
  volatile uint32_t ready = *(uint32_t*)(0x4000C100);
  while (!ready) {
    ready = *(uint32_t*)(0x4000C100);
  }

  /* WARNING: we can't write the code this way!
   *  Without `volatile`, the compiler optimizes out the memory access
  while (!*(uint32_t*)(0x4000C100));
  */

  // read data and print it
  volatile int32_t value = *(int32_t*)(0x4000C508);
  float temperature = ((float)value)/4.0;
  printf("Temperature=%f degrees C\n", temperature);

  nrf_delay_ms(1000);
}
```

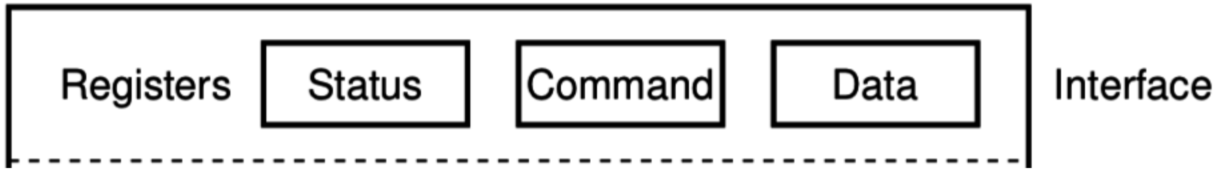
Break + relevant xkcd

https://xkcd.com/138/

Outline

- Input/Output Motivation
- Application-Level Input/Output
- Talking to Devices
  - MMIO Example
- **Device Interaction Patterns**
- Device Drivers

What do interactions with devices look like?



1. while STATUS==BUSY; Wait
   - (Need to make sure device is ready for a command)

2. Write value(s) to DATA

3. Write command(s) to COMMAND

4. while STATUS==BUSY; Wait
   - (Need to make sure device has completed the request)

5. Read value(s) from Data

This is the "polling" model of I/O.

"Poll" the peripheral in software repeatedly to see if it's ready yet.

Waiting can be a waste of CPU time

1. **while STATUS==BUSY; Wait**
   - **(Need to make sure device is ready for a command)**
2. Write value(s) to DATA
3. Write command(s) to COMMAND
4. **while STATUS==BUSY; Wait**
   - **(Need to make sure device has completed the request)**
5. Read value(s) from Data

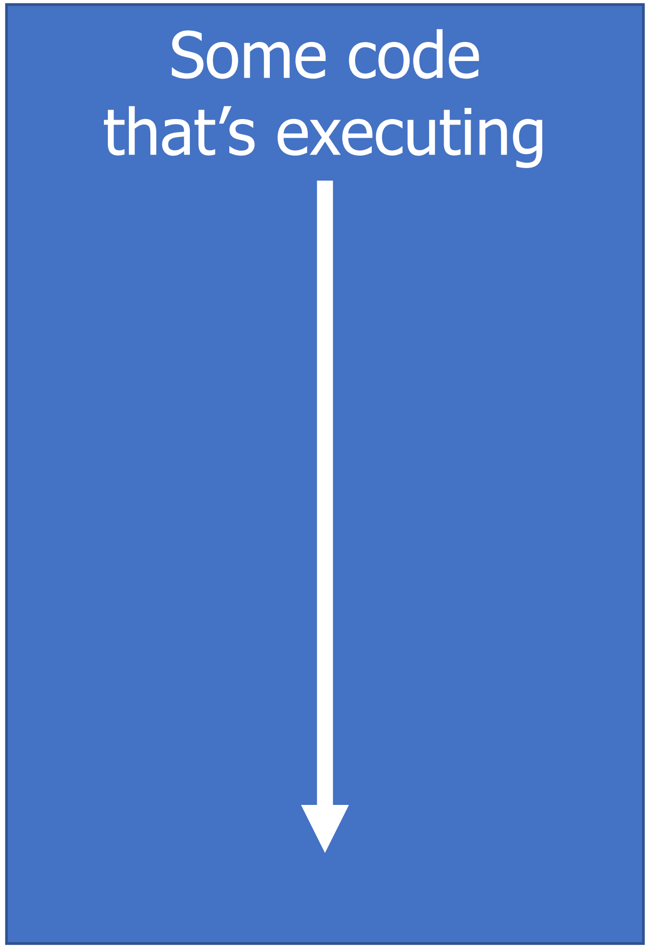
- Imagine a keyboard device
  - CPU could be waiting for minutes before data arrives
  - Need a way to notify CPU when an event occurs
    - Interrupts!

Interrupts

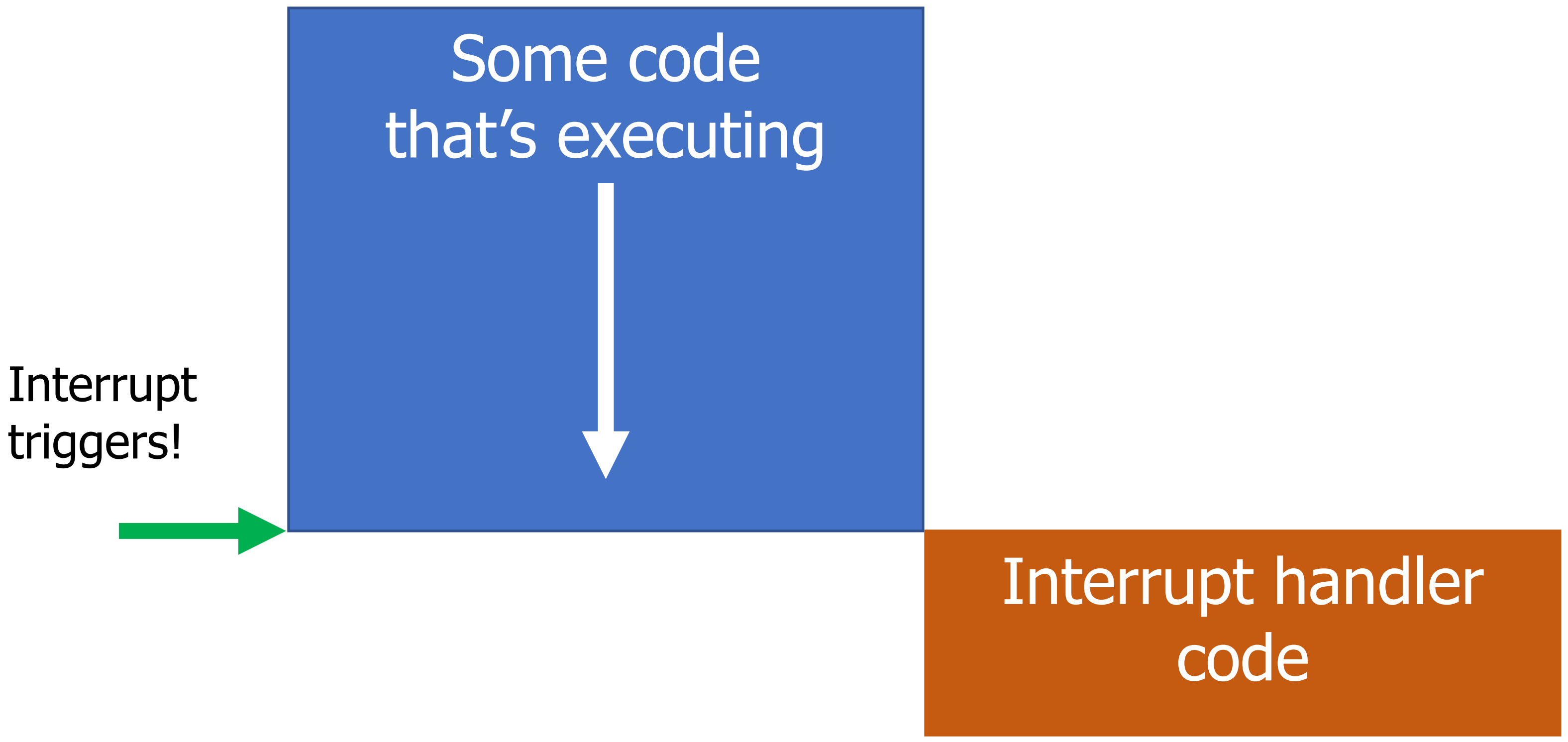
- What is an interrupt?
  - Some event which causes the processor to stop normal execution
  - The processor instead jumps to a software “handler” for that event
    - Then returns back to what it was doing afterwards

- What causes interrupts?
  - Hardware exceptions
    - Divide by zero, Undefined Instruction, Memory bus error
  - Software
    - Syscall, Software Interrupt (SWI)
  - External hardware
    - Input pin, Timer, various “Data Ready”

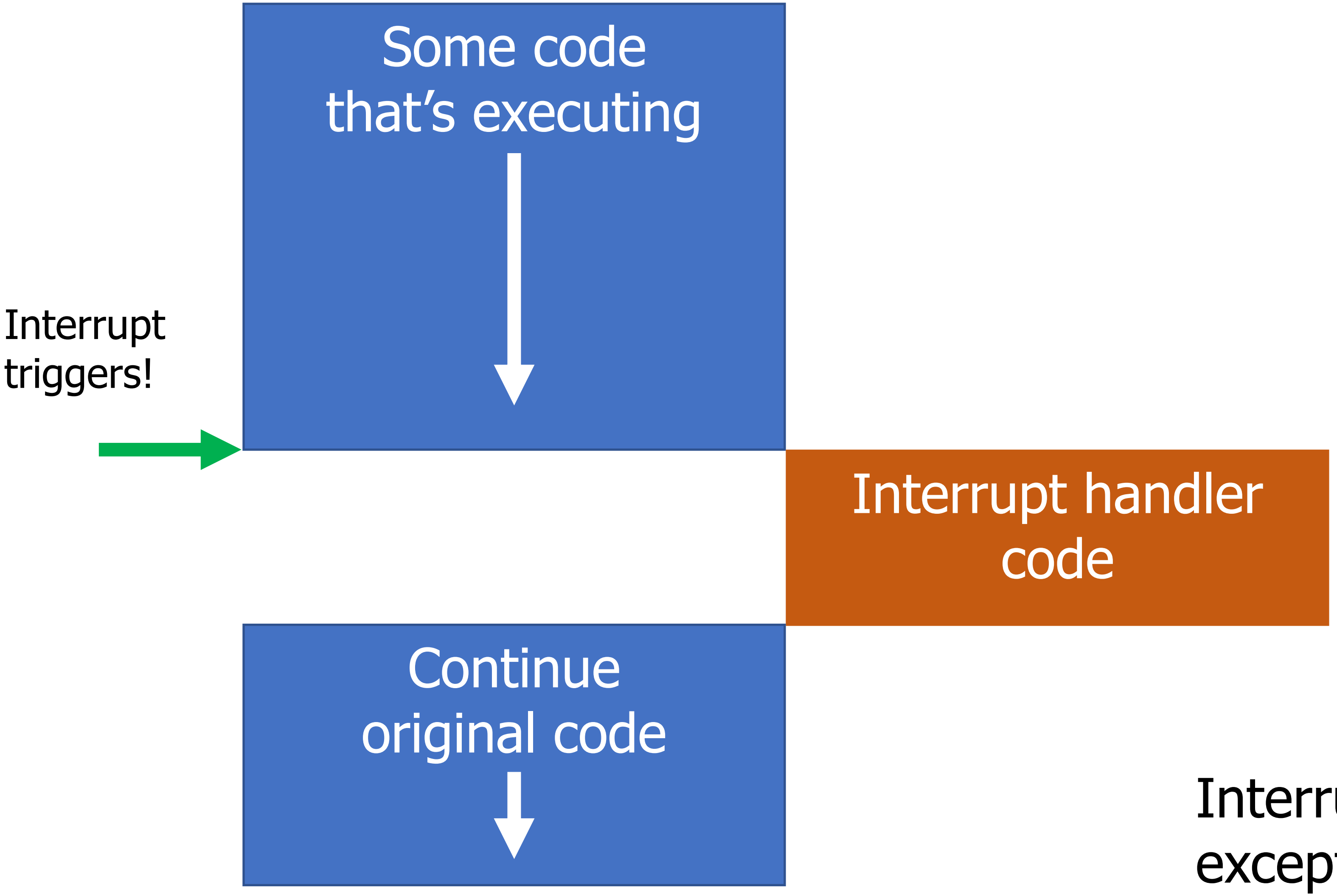
Interrupts, visually



Interrupts, visually



Interrupts, visually



Interrupts are a form of exceptional control flow

Hardware devices can generate interrupts

- Each device maps to some number of hardware interrupts
- Done at system boot time for x86-64
  - Discover devices
  - Map devices into address space
  - Map interrupts for devices
- Hardcoded into hardware for microcontrollers

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Interrupts allow waiting to happen asynchronously

- Prior code example was *synchronous*
  - Nothing else continued on the processor until access was complete
  - Good for very fast devices (like the real-time clock, that just returns data)
  - We call this "Polling"

- With interrupts, device handling is now *asynchronous*
  - Access occurs in the background and processor can do something else
  - Good for very slow devices (Disk)
  - Comes with all the downsides of concurrency though…

Microcontroller TEMP device supports interrupts!

- Can either loop while checking the EVENTS_DATARDY register
- Or could enable an interrupt from the device
  - And only bother reading data when it is ready

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

Table 110: Instances

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

When should a system use polling versus interrupts?

- Polling
  - Great if the device is going to respond immediately (like 1 cycle)
  - Important if we need to respond very quick (less than a microsecond)

- Interrupts
  - Great if we'll need to wait a long time for status to change
  - Still responds pretty quickly, but not *immediately*
    - Needs to context switch from running code to interrupt handler

Check your understanding – writing to GPU

- Let’s say that a GPU has MMIO registers for an entire 4 KB page
  - Takes 100 ns to write each word (8 bytes) of memory

- Assuming that we’re just writing all zeros (ignore reading from memory), how long does it take to write a page to MMIO?

Check your understanding – writing to GPU

- Let's say that a GPU has MMIO registers for an entire 4 KB page
  - Takes 100 ns to write each word (8 bytes) of memory

- Assuming that we're just writing all zeros (ignore reading from memory), how long does it take to write a page to MMIO?

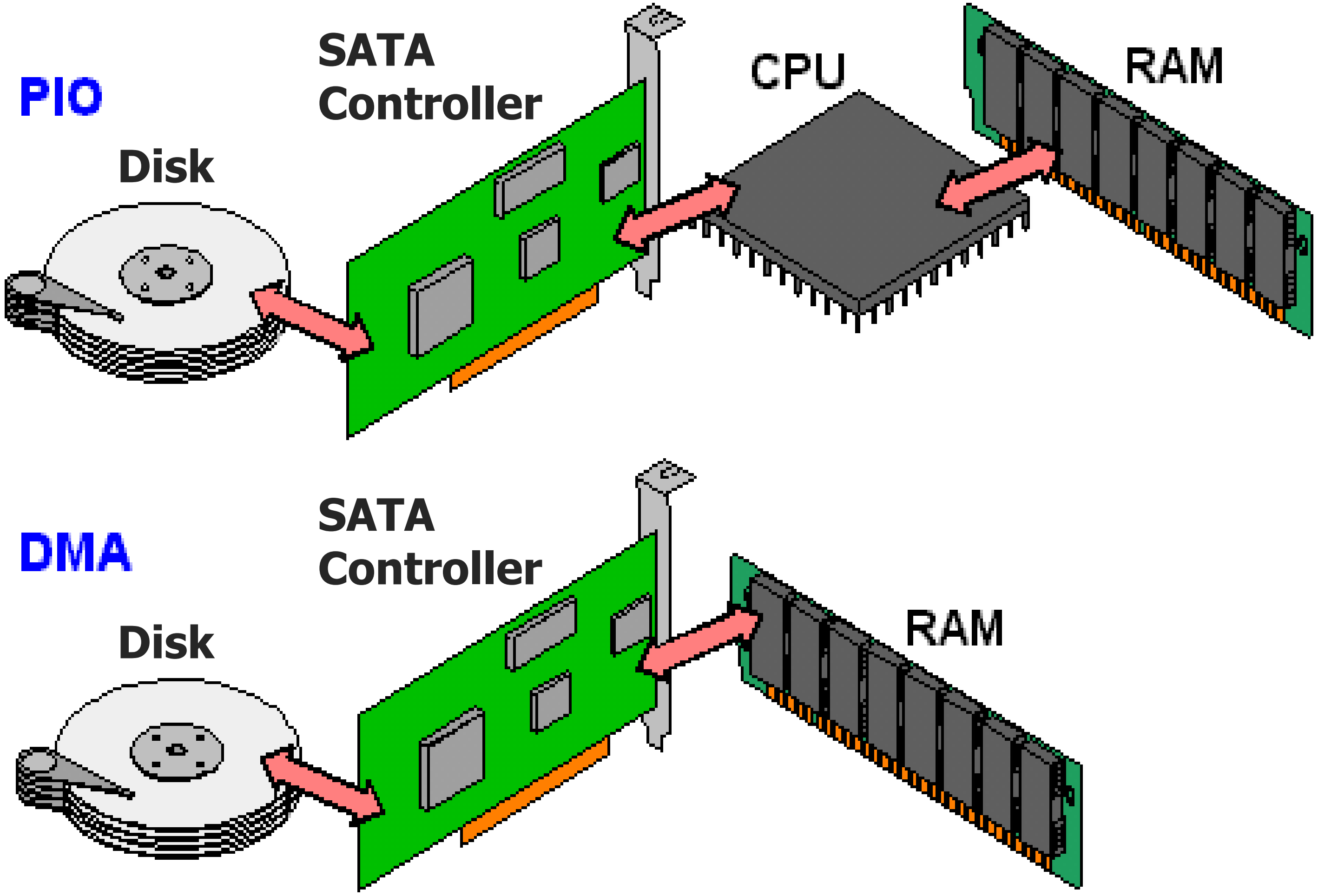
  - 4 KB / 8 B = 512 writes * 100 ns / write = 51 μs
    - (For a 3 GHz processor, that's ~150,000 cycles)

Direct Memory Access (DMA)

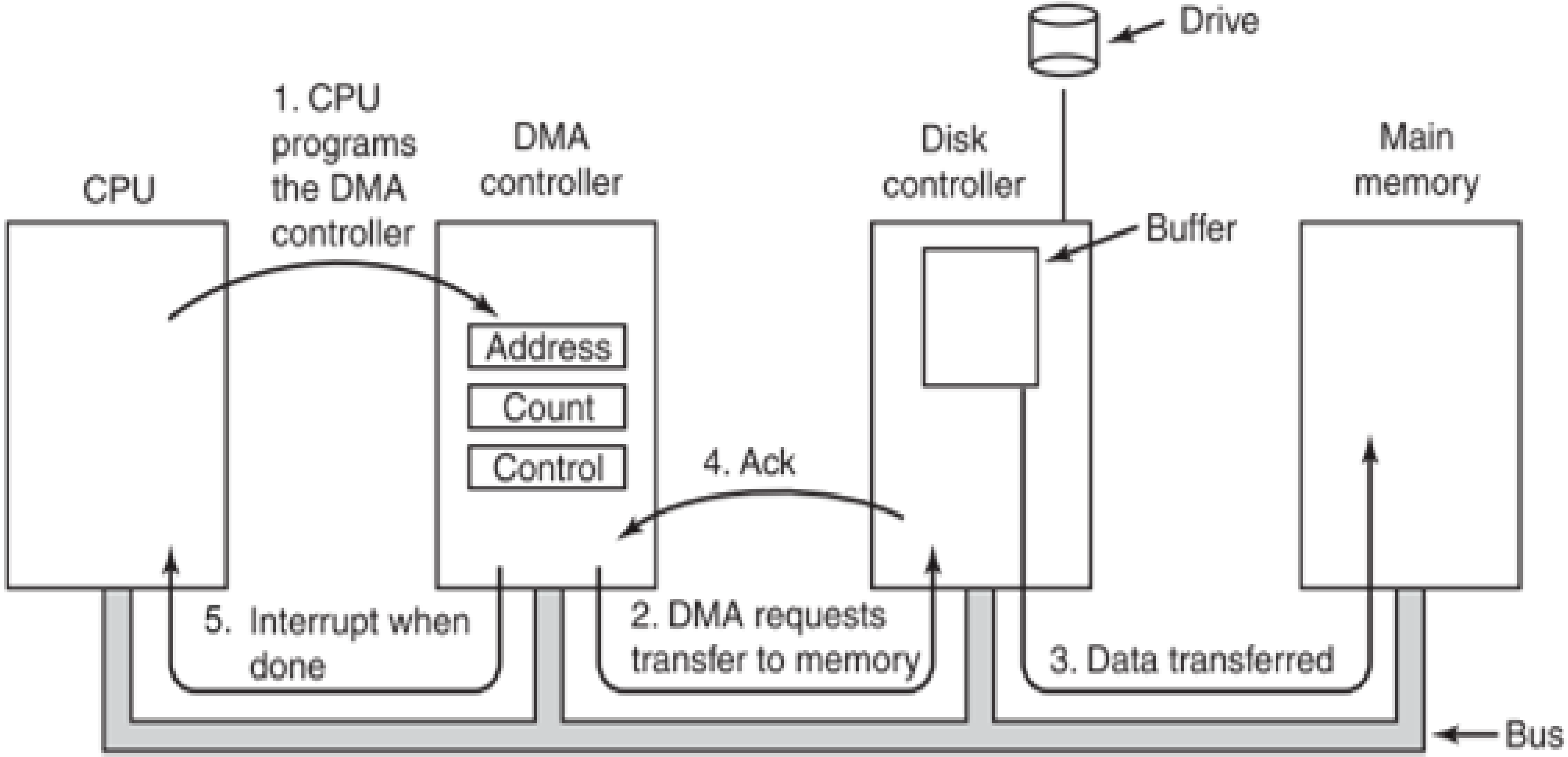
- Even with interrupts, providing data to the peripheral is time consuming for transferring lots of data
  - Need to be interrupted every byte, to copy the next byte over

- DMA is an alternative method that uses hardware to do the memory transfers for the processor
  - Software writes address of the data and the size to the peripheral
  - Peripheral reads data directly from memory
  - Processor can go do other things while read/write is occurring

Programmed I/O versus Direct Memory Access



General-purpose DMA



Full peripheral interaction pattern

1. Configure the peripheral
2. Enable peripheral interrupts
3. Set up peripheral DMA transfer
4. Start peripheral

Continue on to other code

5. Interrupt occurs, signaling DMA transfer complete
6. Set up next DMA transfer

Continue on to other code, and repeat

Break + Open Question

- What kinds of peripherals/devices should you use the DMA for?

Break + Open Question

- What kinds of peripherals/devices should you use the DMA for?

  - Anything where there is a lot of data coming in over a period of time
    - Either a big buffer of lots of data, like a radio message

    - Or a bunch of individual samples, coming in quickly

  - Devices
    - Canonical example from general computing: disks (HDD/SSD)
    - Messages to/from other devices (radios, wired busses)
    - Sensor readings (if read quickly)

Outline

- Input/Output Motivation
- Application-Level Input/Output
- Talking to Devices
  - MMIO Example
- Device Interaction Patterns
- **Device Drivers**

What is a device driver?

- A device driver is the software in the Operating System that manages a specific device
  - Modern computers come with MANY of these pre-installed
  - And have the ability to automatically find many others if needed

- Can be generic “keyboard driver”
  or very specific “Ricoh IM C3000 printer driver”

- Source of many bugs in the in Operating System
  - Due to amount and variety of them

How should we write driver software?

- There are various knobs available to us from hardware
  - Polling, Interrupts, DMA

- There are also various software interface design
  - Synchronous
  - Asynchronous Callbacks

Synchronous device drivers

- Synchronous functions
  - Function call issues a command
  - Does not return until action is complete and result is ready

- Example: most functions we’re used to
  - `sqrt()` for example
  - `printf()` also usually works this way (with some exceptions)

- For microcontrollers: Arduino interfaces are usually like this!
  - Easy to get started with and understand

Downside of synchronous code: the waiting

- How long will it take until the function returns?
  - Immediately, seconds, minutes?

- What if there’s an error and the device never responds?
  - More advanced interface could include a timeout option

- Synchronous designs require other synchronous designs
  - We can build synchronous interfaces from asynchronous ones
  - But we can’t go the other way

Asynchronous drivers

- Goal: let the hardware run on its own and have the code get back to it later
- Challenge: programmers don’t think that way
- Other challenge: how do we “get back to it later”?
    - One solution: Callbacks

Callbacks

- Callbacks reuse a similar idea to interrupts
  - When the event occurs, call this function

- General pattern
  - Call driver function with one argument being a function pointer
  - Driver sets up interaction and returns immediately
  - Later the event happens and the driver calls the function pointer

Callback functions

- ```
  uint32_t timer_start(
                uint32_t microseconds,
                void (*callback_fn)(void*),
                void* context
                );
  ```

- ```
  timer_start(duration, my_timer_handler, context);
  ```

- "Context" is often provided as well (void*)
  - Ability for caller to pass an argument for the callback function
  - Often a pointer to a position in a structure or a shared variable to modify
  - Similar to idea of closures in other languages

Example with callbacks (could be temp driver)

```
void temp_callback(float temp, void* _unused) {
  printf("Temperature: %f degrees C\n", temp);
}

int main(void) {
  printf("Board started!\n");

  // Get temperature without blocking
  get_temperature_nonblocking(temp_callback, NULL);
  nrf_delay_ms(1000); // should have printed before delay is complete
  ...
```

CE346 – Microprocessor System Design

- Embedded Systems
  - How does hardware work?
  - How do we write drivers to interact with hardware from software?
  - Sensing and sensors

- Big open-ended project: anything with inputs and outputs

- Project demos: Tuesday of exam week (12/5) from 11-5 in Mudd 3514
  - Public! Anyone is welcome to stop in
  - Not formal presentations, just demos you can play with

Outline

- Input/Output Motivation
- Application-Level Input/Output
- Talking to Devices
  - MMIO Example
- Device Interaction Patterns
- Device Drivers