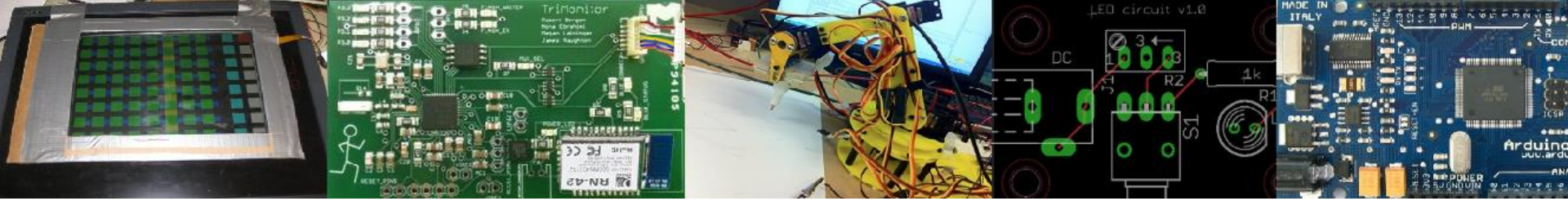


EECS 473
Lecture 2

More on interfacing

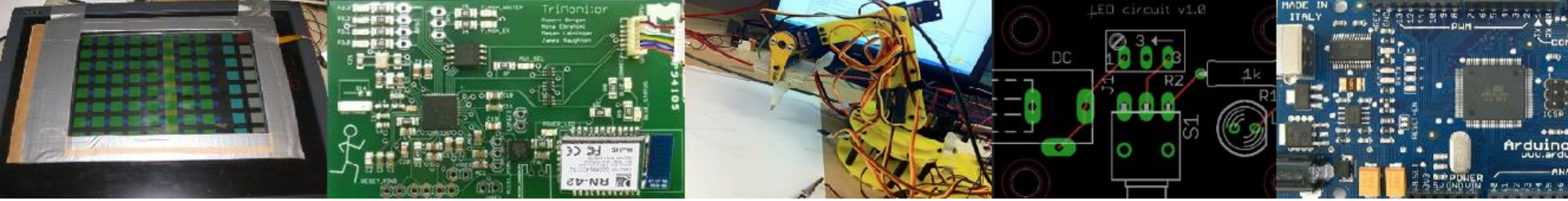
Project overview

(Maybe) Start on Real time and RM scheduling

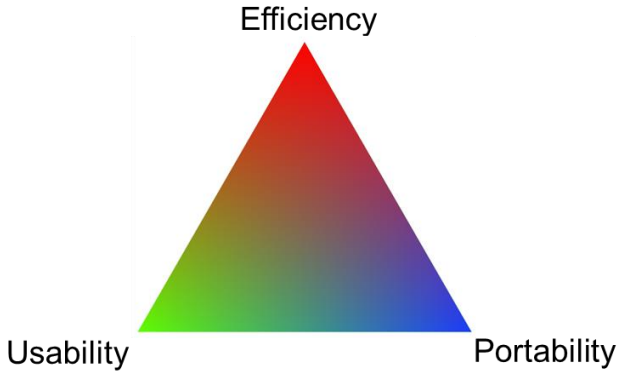
MICHIGAN

Today

- Review and expand on hardware interfacing
  - Some questions using different devices. Compare to Arduino.
  - Discuss top and bottom of interface diagram.
- Projects
  - What's doable, when things are expected.
- Introduction to real-time systems
  - I doubt we'll get to this today, but maybe a bit.



Hardware Interfacing



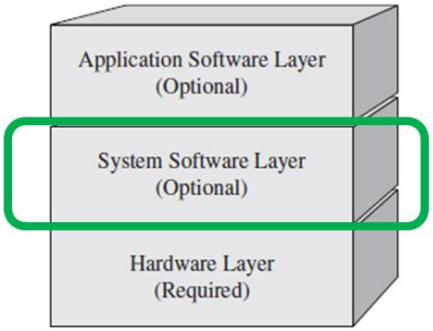
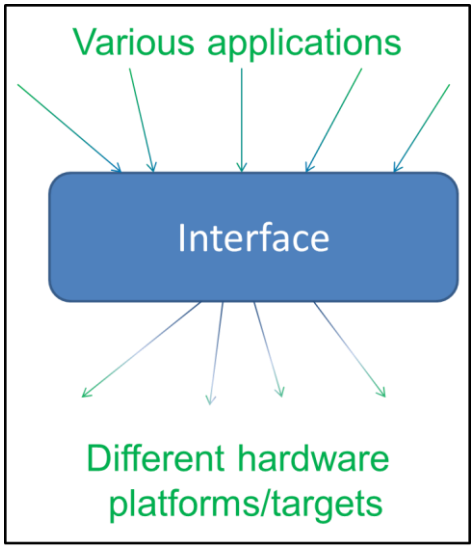


Figure 1-1
Embedded Systems Model.
Figure from *Embedded Systems Architecture* by Tammy Noergaard



MICHIGAN

What should the interface be for a servo?

- What I wanted you to do was to discuss:
  - What basic functions you want.
  - What the interface to those functions should be like.
  - Try to get a formal description of as much as you can.
  - You will have about 5 minutes.
  - Do it yourself—no web searches.

Discuss ideas

- Use OO (struct/class) for interface.
  - SetAngle
  - SetAngle (with speed)
  - Initialize servo (what info?)
  - GetAngle

Things change…

- Might get a new servo
  - So period and duty cycle might be different
- Might get a new processor
  - So timer configuration might change.
- Might need additional functionality
  - Perhaps want to include stepper motors

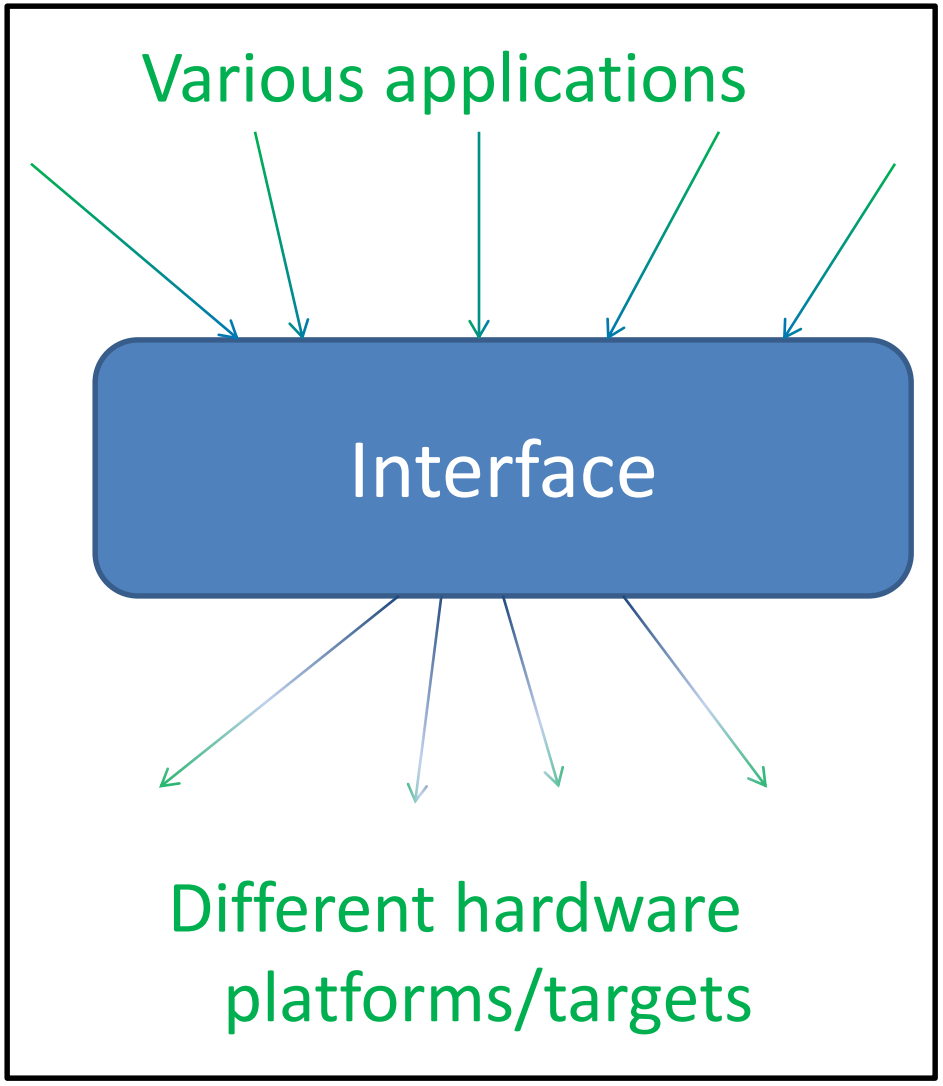
…but in 373 we didn't…

- Most of you didn't create any meaningful interfaces in 373*
  - Exposed the low-level details to the programmer
    - After all you were the programmer and interface design takes time.
      - Plus you often don't yet know what you're going to need.
  - This makes it easy to do boneheaded things.
    - Wrong MMIO address, lots of replicated code, etc.
  - It also makes it hard to write good code.
    - You are worrying about too many things at once.
    - Keep worrying about things like bounds checking when interface should do that.

*Though some did. Often the more successful groups had well-defined and reasonably well-documented interfaces.

Stepping back

- Can think of an interface as a single way to talk to a class of hardware devices.
  - Each application uses the interface.
  - Each target "just" needs to support the interface.
  - What is the alternative?
- Now that we have a sense of what an interface is, let us look at what makes a good one.



Creating interfaces to hardware

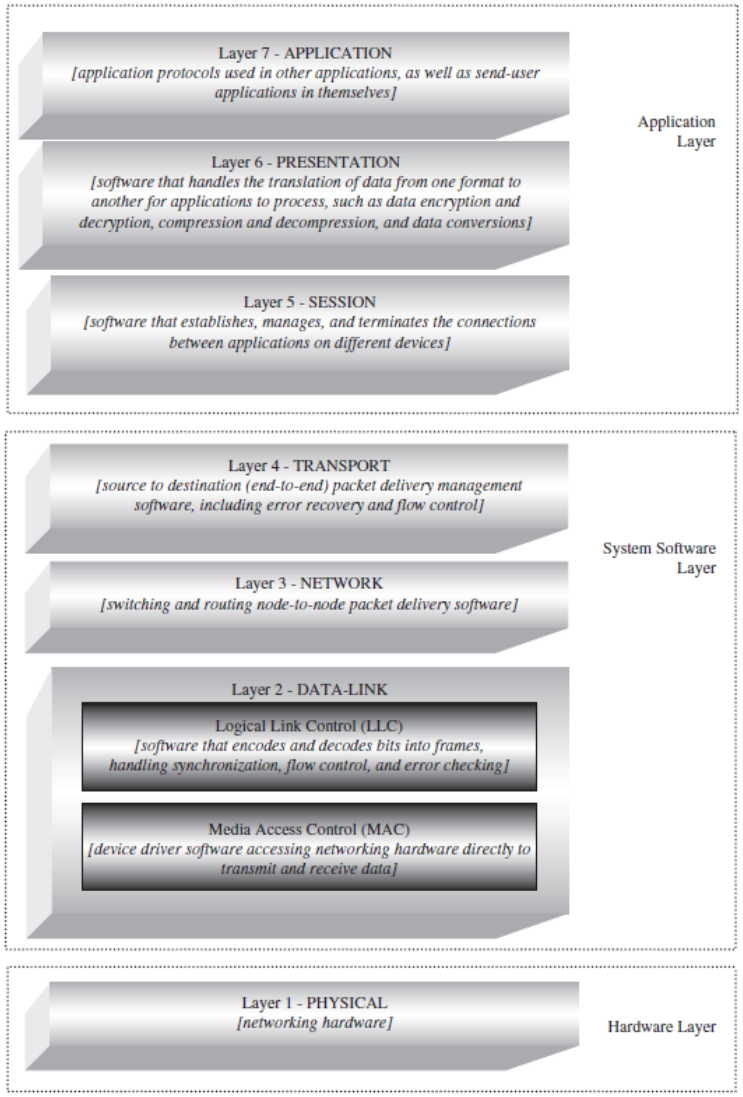
- A good hardware interface has three main goals:
  - It is easy to understand and use (useable)
  - It is efficient
  - It is portable to other hardware platforms.
- Those three things are often at odds.
  - And sometimes one matters a lot more than the others.
    - If the plan is to only use one hardware platform, portability matters little (though it matters a bit, as often plans change!)
      - Examples?
    - If the plan is that *you* are the only one who will use it, easy is less important.
      - *That* plan changes more than any IME
      - And easy is still powerful even if you are the only user.

What makes an interface easy to understand and use?

What does an interface have to do with efficiency?

- On the silly side:
  - One could imagine our servo interface having only a "turn 1 degree" function (direction specified)
    - Covers all functionality
    - But big turns require a lot of code to run.
- On the less silly side
  - If we use angles (in degrees) as the basis for the interface, that is going to require some math in the interface itself to convert to a register value.
    - Perhaps the programmer could skip a lot of that.
  - Other places we might see inefficiency for our servo?

Look at Open Systems Interconnect (OSI)



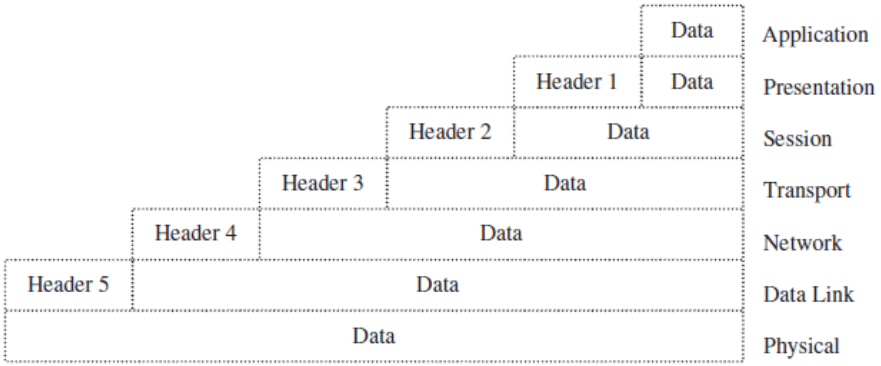


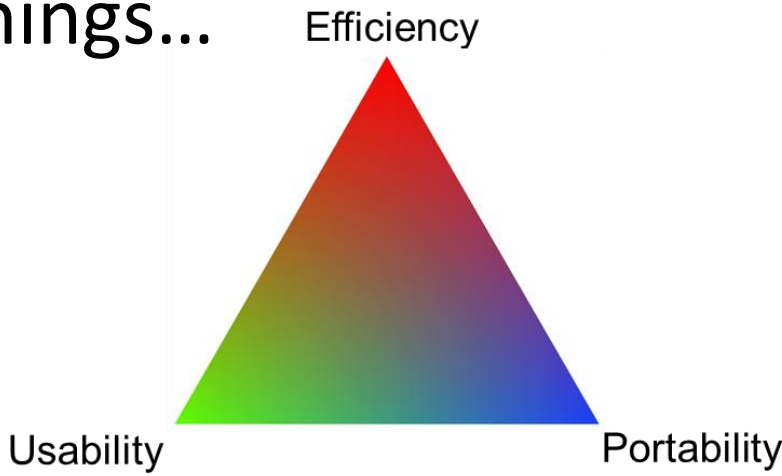
Figure 2-24
Header diagram.

This degree of layering is considered largely necessary, but it clearly creates a lot of overhead (inefficiencies)

Figures from *Embedded Systems Architecture* by Tammy Noergaard

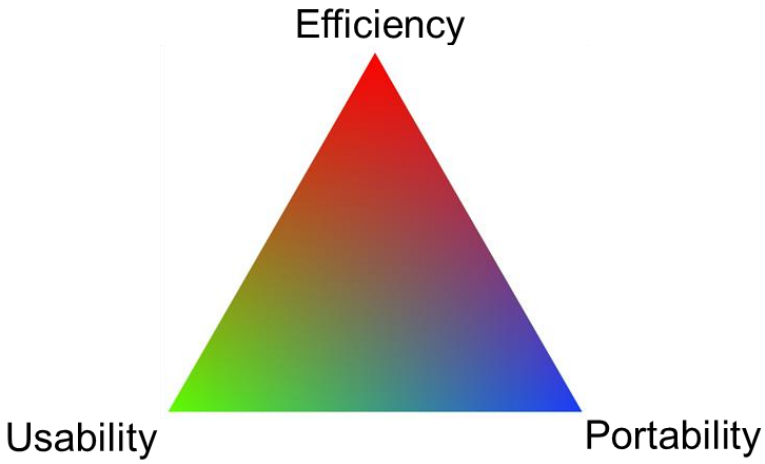
So what makes hardware-interface design difficult?

- Mainly the three competing requirements of usability, efficiency and portability.
  - As discussed, they often fight with each other.
- But there are a few other things…



Not a trivial task

- Trying to be able to work with all applications, targets, and platforms is hard.
  - Lots of coding effort
  - Can hurt efficiency and/or usability.



- Examples:
  - Some servos may need to be “linearized”.
    - That is their angle isn’t exactly proportional to the pulse width
  - Some servos do continuous rotation
    - Pulse width determines speed and direction, not angle.
  - Servos can (often?) “freewheel”
    - Apply no pulse, it provides no resistance to turning.

Arduino interface for servos

- attach() Attaches a servo motor to an i/o pin
- read() Gets the servo motor position
- write() Sets the servo motor position angle
- attached() Returns if there is a servo attached
- detach() Detaches a servo motor from an i/o pin

- writeMicroseconds()
  Sets the servo pulse width in microseconds
- readMicroseconds()
  Gets the last written servo pulse width in microseconds*

*Documentation doesn't mention this, but it exists?!

detach()

Description

Detach the Servo variable from its pin. If all Servo variables are detached, then pins 9 and 10 can be used for PWM output with analogWrite().

Syntax

- `servo.detach()`

Parameters

- servo: a variable of type Servo

So?

- Examples: (same as last page)
  - Some servos may need to be "linearized".
    - That is their angle isn't exactly proportional to the pulse width
  - Some servos do continuous rotation
    - Pulse width determines speed and direction, not angle
  - Servos can (often?) "freewheel"
    - Apply no pulse, it provides no resistance to turning.
      - Sometimes useful to an application.
  - Some servos provide angle they are *actually* at.
- How can supporting those things cause us difficulties?
  - Too many functions?
  - Too much computation?
- And that doesn't even account for having to support different processors and timers.
  - Yes, some processors have multiple types of timers.
    - Pretty common actually.

attach()

```
servo.attach(pin)
servo.attach(pin, min, max)
```

Parameters

servo	A variable of type Servo
pin	pin number servo attached to
min	min value in microseconds defaults to 544
max	max value in microseconds defaults to 2400

Example:

```
#include "Servo.h"
Servo myservo;
void setup() {
  // attaches a servo
  // connected to pin 2
  myservo.attach(2);
}

void loop() {
  // position the servo angle
  // at 90 degrees
  myservo.write(90);
}
```

Interfaces: Servo example

```
typedef struct  {
  uint8_t nbr        :6 ;             // a pin number from 0 to 63
  uint8_t isActive   :1 ;             // true if this channel is enabled, pin not pulsed if false
} ServoPin_t   ;

typedef struct {
  ServoPin_t Pin;
  unsigned int ticks;
} servo_t;

class Servo
{
public:
  Servo();
  uint8_t attach(int pin);           // attach the given pin to the next free channel,
                                     // sets pinMode, returns channel number or 0 if failure
  uint8_t attach(int pin, int min, int max); // as above but also sets min and max values for writes.
  void detach();
  void write(int value);             // if value is < 200 its treated as an angle, otherwise as pulse width in microseconds
  void writeMicroseconds(int value); // Write pulse width in microseconds
  int read();                        // returns current pulse width as an angle between 0 and 180 degrees
  int readMicroseconds();            // returns current pulse width in microseconds for this servo
  bool attached();                   // return true if this servo is attached, otherwise false
private:
   uint8_t servoIndex;               // index into the channel data for this servo
   int8_t min;                       // minimum is this value times 4 added to MIN_PULSE_WIDTH
   int8_t max;                       // maximum is this value times 4 added to MAX_PULSE_WIDTH
};
```

Interfaces: Servo example

```
void Servo::write(int value)
{
  if(value < MIN_PULSE_WIDTH)
  {  // treat values less than 544 as angles in degrees
     //(valid values in microseconds are handled as microseconds)
    if(value < 0) value = 0;
    if(value > 180) value = 180;
    value = map(value, 0, 180, SERVO_MIN(),  SERVO_MAX());
  }
  this->writeMicroseconds(value);
}
```

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

Best I can find— integer division is about 15 microseconds on an Arduino.

Code to keep it portable
Initializing the ISR

```
static void initISR(timer16_Sequence_t timer)
{
#if defined (_useTimer1)
  if(timer == _timer1) {
    TCCR1A = 0;             // normal counting mode
    TCCR1B = _BV(CS11);     // set prescaler of 8
    TCNT1 = 0;              // clear the timer count
#if defined(__AVR_ATmega8__)|| defined(__AVR_ATmega128__)
    TIFR |= _BV(OCF1A);      // clear any pending interrupts;
    TIMSK |=  _BV(OCIE1A) ;  // enable the output compare interrupt
#else
    // here if not ATmega8 or ATmega128
    TIFR1 |= _BV(OCF1A);     // clear any pending interrupts;
    TIMSK1 |=  _BV(OCIE1A) ; // enable the output compare interrupt
#endif
#if defined(WIRING)
    timerAttach(TIMER1OUTCOMPAREA_INT, Timer1Service);
#endif
  }
#endif

#if defined (_useTimer3)
  if(timer == _timer3) {
    TCCR3A = 0;             // normal counting mode
    TCCR3B = _BV(CS31);     // set prescaler of 8
    TCNT3 = 0;              // clear the timer count
#if defined(__AVR_ATmega128__)
    TIFR |= _BV(OCF3A);     // clear any pending interrupts;
            ETIMSK |= _BV(OCIE3A);  // enable the output compare
                                    // interrupt
#else
    TIFR3 = _BV(OCF3A);     // clear any pending interrupts;
    TIMSK3 =  _BV(OCIE3A) ; // enable the output compare interrupt
#endif
#if defined(WIRING)
    timerAttach(TIMER3OUTCOMPAREA_INT, Timer3Service);
                                // for Wiring platform only
#endif
  }
#endif

#if defined (_useTimer4)
  if(timer == _timer4) {
    TCCR4A = 0;             // normal counting mode
    TCCR4B = _BV(CS41);     // set prescaler of 8
    TCNT4 = 0;              // clear the timer count
    TIFR4 = _BV(OCF4A);     // clear any pending interrupts;
    TIMSK4 =  _BV(OCIE4A) ; // enable the output compare interrupt
  }
#endif

#if defined (_useTimer5)
  if(timer == _timer5) {
    TCCR5A = 0;             // normal counting mode
    TCCR5B = _BV(CS51);     // set prescaler of 8
    TCNT5 = 0;              // clear the timer count
    TIFR5 = _BV(OCF5A);     // clear any pending interrupts;
    TIMSK5 =  _BV(OCIE5A) ; // enable the output compare interrupt
  }
#endif
}
```

Entire library is 337 lines (including comments)

Conclusions?

- Hardware interfacing isn't trivial
  - Having to deal with multiple platforms can at the least make the code more complex.
- Efficiency issues exist
  - Uses a significant amount of CPU time if specify things in degrees.
    - Might not matter—if CPU is otherwise fairly idle who cares?
  - Efficiency issue hidden from application writer.
    - That's a common problem and happens here.
- OSI model really gets at just how complex things can get.

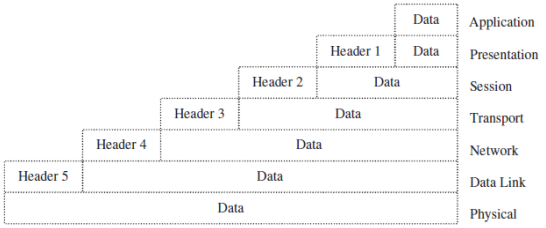


Figure 2-24
Header diagram.

Terminology

- There is a lot of terminology wrapped around hardware interfacing.
  - Terms like: HAL (hardware abstraction layer), Middleware, and Device Driver are all related to hardware interfacing.
    - And it's not unusual to see different people use those terms differently.
  - We'll try to take a look at that terminology later on.
    - I'll generally use "device driver" or "system software layer"

Hardware interfacing summary

- Hardware Interfaces:
  - Create a standard "driver" interface where you can abstract the interface to a device from the tasks the processor needs to perform.
  - Ideally covers most use cases (applications)
  - Ideally a single interface covers a wide variety of parts/targets and platforms.
- Advantages:
  - Ideally application software can be written independent of platform & target.
  - Application programmer can think at the application level and not the device/MMIO level.
    - Abstraction is good and makes for code that is easier to write, debug and maintain.

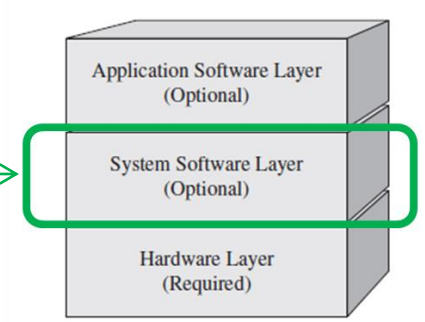
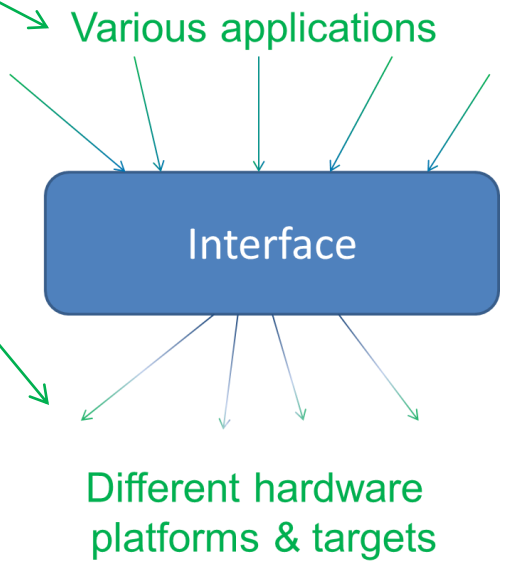
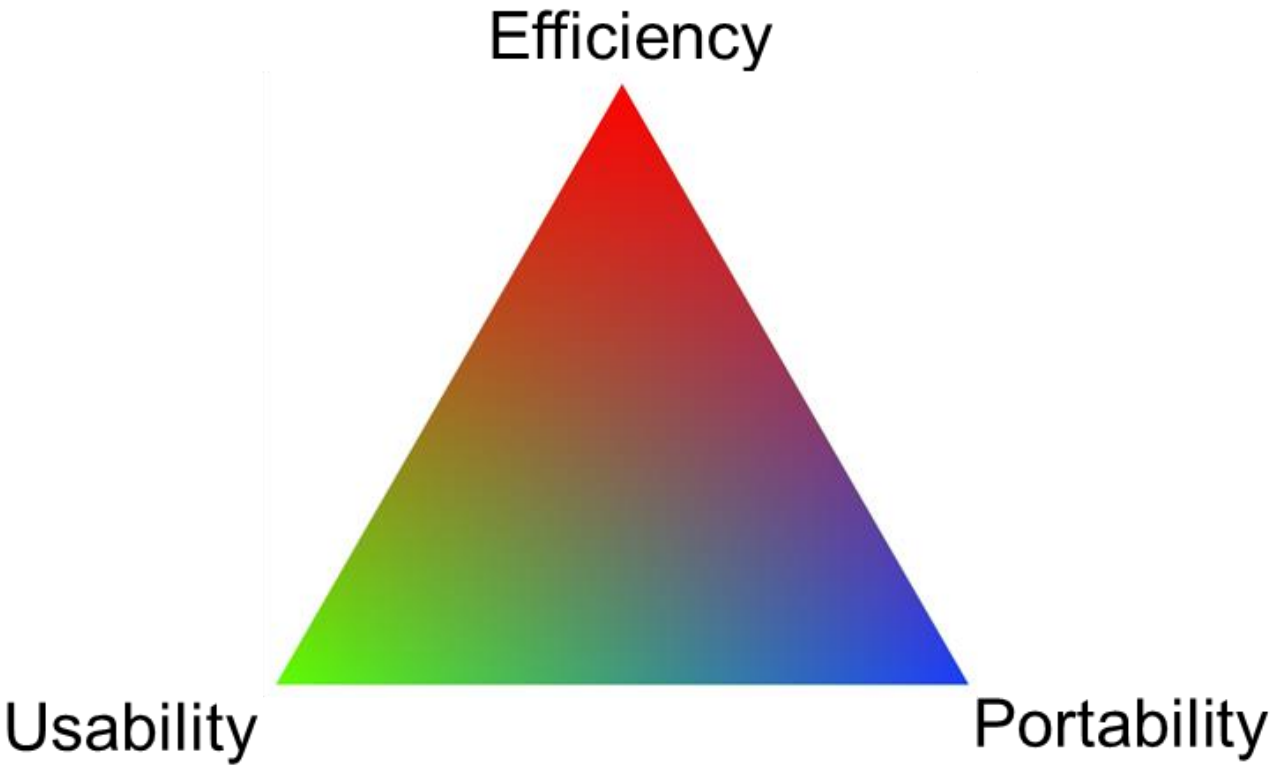
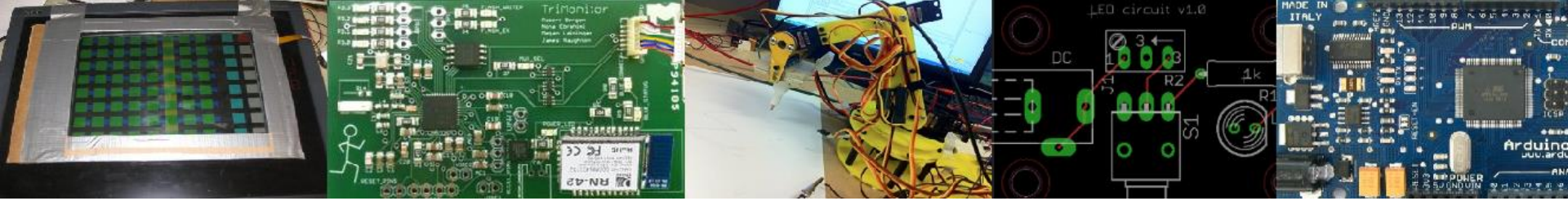


Figure 1-1
Embedded Systems Model.
Figure from *Embedded Systems Architecture* by Tammy Noergaard



Interfaces: wrap-up





Projects

What's reasonable

What's expected when

MICHIGAN

What is a reasonable project?

- **Required**
  - Entire embedded system with a realistic use
    - Design allows for that realistic use (not an unusable prototype)
  - Doable in the semester
  - Design and population of a PCB with a processor (or maybe FPGA).
  - Design and implementation of hardware interfaces
- **Suggested**
  - Not “barebones” programming
    - Use RTOS, Linux, or something else
    - Barebones might make sense if software fairly trivial
  - Stay away from switching power supplies

Projects in the past:
Baby Monitor

- What it does
  - Watches for SIDS
  - Sets alarm on board and wirelessly to base station if problem detected.
- Main issues
  - Easy to use
    - Easy to place on baby correctly, easy to recharge, no complex interface
  - Resilient
    - Water-resistant, can be dropped, easily cleaned.
  - Cheap
    - Very low-cost target.
  - Testing that it works
    - Need baby?

Projects in the past:
Triathlon monitor

- What it does:
  - Monitor location, heart rate, and strides
  - Display on watch, record to dump to PC later
- Issues
  - Weight
  - Waterproof
  - Getting data in (GPS, heart rate, strides)
  - Getting data out (to watch, stored and to PC)

Projects in the past:
Bike helmet (Hail-met)

- What it does:
  - Bluetooth
    - Voice, etc. Can make calls or listen to music
  - Turn signals and tail light
  - Basic heads-up display (signals, call info)
  - Solar recharge
- Issues
  - Weight
  - Waterproof
  - Safety

Hail-Met!

An all-in-one multi-purpose helmet for the modern commuter.

Intro

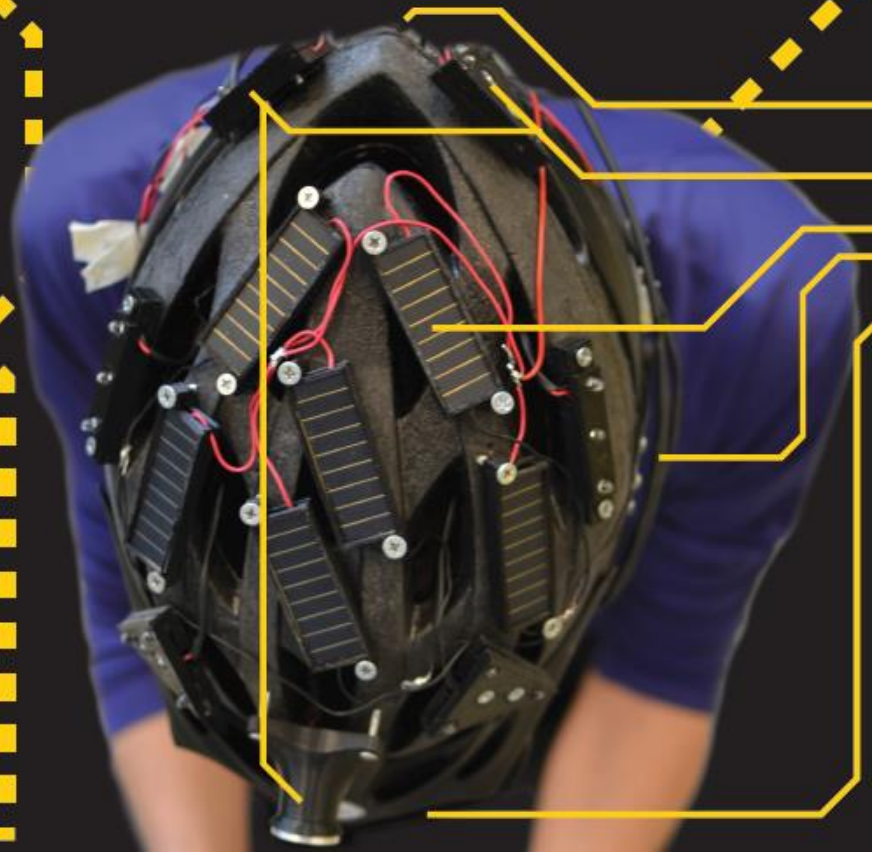
Besides protecting your head from direct contact with hard surfaces, traditional bike helmets don't offer many benefits to consumers. Advanced communication, entertainment, and night-time safety features come standard with every new car, but don't yet exist for cyclists.

Problem

Advanced communication, entertainment, and night-time safety features come standard with every new car, but don't yet exist for cyclists.

More modular solutions (like stand-alone detachable bike head lights) do exist but are easily stolen if they are not detached when the bike is in public.

Bicycle helmets are exposed to the elements (moisture, dust, etc.), so embedding electronic devices into a helmet is no easy feat.



Hardware

- PCB with Bluetooth, LiPo Battery, Charging, and LED Driver
- Turn Signals, Headlight and Taillight
- Solar panels
- Speakers
- Heads up display (HUD)

Custom PCB

Interfacing & Software Design

The Hail-met runs on an

Project dates next 30 days

- Sept. 5th:
  - Project ideas pdf *turned in* by **11pm** (on Gradescope shortly)
- Sept 6th:
  - All ideas posted, probably not until late in the day.
- Sept. 7thth:
  - Forming groups 6:30-8:30pm **(Room TBA)**
  - Each person will have a chance to speak for about 30 seconds about what they'd like to do
  - Wander around and form groups.
- Sept. 15th:
  - Draft proposal due (3% of project grade)
- Sept 21nd
  - Required meeting with staff to discuss proposal*
    - No class that day.
  - After meeting most groups can (and should) start ordering parts!
- Sept 28th
  - Formal proposal due. (12% of project grade).

*You should have met with various 473 staff members well before this!

Milestones: What are they? What's due?

- Report to us about how things are going and some demonstrations.
- What's due:
  - A short report (1-2 pages)
    - Due a few days before the meeting
  - Demonstrate to GSIs that you've meet your "objectively demonstrable" deliverables.
    - Needs to be done the day before the meeting at the latest.
- In general, these should take 30-60 minutes to write unless things are in really bad shape.
  - There is a docx template you are to use.

Milestone 1

(Can start ordering parts on Sept 21nd after meeting)

- Oct 12th
  - Device interfaces designed and working
    - Might need some refinement, but in good shape
  - Prototype largely working
    - Probably devboard (maybe breadboard, but avoid if possible!)
    - Can talk to all devices via interfaces
    - Can more-or-less do the task
  - Have identified solutions to each major issue.
- Started on PCB
  - Perhaps just barely, but at least have patterns done

Milestone 1 Example: Baby

- Have interface to GPS, flash (for storage), and variable resistor (for checking breathing) written useable.
- Not yet sure on chest cavity movement of infant.
  - Flash still a bit buggy—erasing not always working.
    - Otherwise demonstrated
  - GPS association working, (time is 3 minutes, but worst case probably acceptable for application)
    - Demonstrated

Milestone 1 Example: Baby

- Prototype on Arduino
  - Each interface works
  - When done together GPS (serial) interrupt messing with timer interrupt for resistor monitoring

PCB deadline

- All PCBs must be ordered by Oct 31st
  - **Worst case**, you need to respin the PCB later but we want them all out by this date.
    - Depending on budget for PCB, should have back in 7-10 days.
  - Should target earlier—hopefully most groups have PCB out by Oct 21st

Milestone 2

- November 7th
  - PCB back and assembled, still may be debugging last issue or two
  - Final plans for all “extras”
    - Enclosure (if needed)
    - Special documentation
    - ?????

November 20th

- Everything should be done, just final testing and debug of a largely working system.
- Start on design expo things and final report by 11/26.
  - Setup, poster, etc.
  - Discussion in class about posters etc. on 11/21
  - Ideally get much of the report written before the expo.

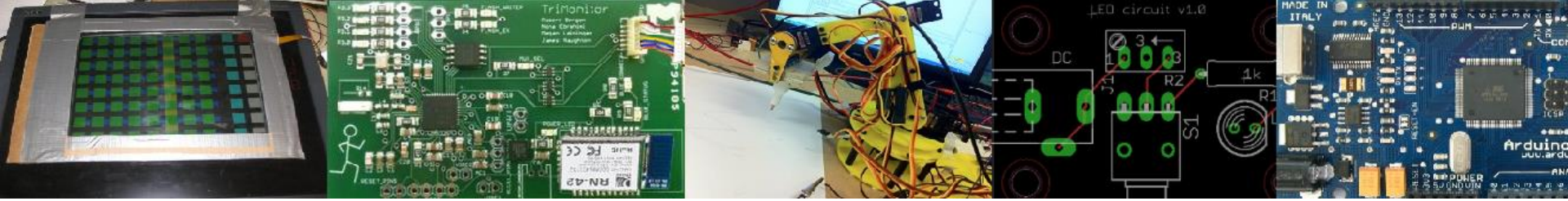
- Testing, testing, testing

Design Expo

- December November 30th
  - Working and clearly presented demo
  - Hopefully exciting
    - Not all projects will be and that's okay.
  - Shirt and tie or equivalent required.

December 3rd

- Final report due.
  - This report and the associated documentation is 15% of of your project grade.
  - With the proposal document done, you have a solid starting point.



EECS 473
Advanced Embedded Systems

An introduction to real time systems and scheduling

MICHIGAN

Chunks adapted from work by
Dr. Fred Kuhns of Washington University
and Farhan Hormasji

What is a Real-Time System?

- Real-time systems have been defined as: "those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced";
  - J. Stankovic, "Misconceptions About Real-Time Computing," ***IEEE Computer,*** 21(10), October 1988.

Soft, Firm and Hard deadlines

- The instant at which a result is needed is called a deadline.
  - If the result has utility even after the deadline has passed, the deadline is classified as **soft**, otherwise it is **firm**.
  - If a catastrophe ***could*** result if a firm deadline is missed, the deadline is **hard**.
- Examples?

Definitions taken from a paper by Kanaka Juvva, not sure who originated them.

Why is this hard?

Three major issues

1. We want to use as cheap ($$, power) a processor as possible.
   – Don't want to overpay
2. There are non-CPU resources to worry about.
   – Say two devices both on an SPI bus.
   – So often can't treat tasks as independent
3. Validation is hard
   – You've got deadlines you ***must*** meet.
     • How do you ***know*** you will?

Let's discuss that last one a bit more

Validating a RTS is hard

- Validation is simply the ability to be able to prove that you will meet your constraints
  - Or for a non-hard time system, prove failure is rare.

- This is a hard problem just with timing restrictions
  - How do you ***know*** that you will meet all deadlines?
    - A task that needs 0.05s of CPU time and has a deadline 0.1s after release is “easy”.
    - But what if three such jobs show up at the same time?
  - And how do you know the worst-case for all these applications?
    - Sure you can measure a billion instances of the program running, but could something make it worse?
      - Caches are a pain here.

- And all that ***ignores*** non-CPU resource constraints!

We need some formal definitions to make progress here…

Real-Time Characteristics

- Pretty much your typical embedded system
  - Sensors & actuators all controlled by a processor.
  - The big difference is **timing constraints** (deadlines).

- Those tasks can be broken into two categories[1]
  - **Periodic Tasks**: Time-driven and recurring at regular intervals.
    - A car checking for a wall every 0.1 seconds;
    - An air monitoring system grabbing an air sample every 10 seconds.
  - **Aperiodic**: event-driven
    - That car having to react to a wall it found
    - The loss of network connectivity.

[1]Sporadic tasks are sometimes also discussed as a third category. They are tasks similar to aperiodic tasks but activated with some known bounded rate. The bounded rate is characterized by a minimum interval of time between two successive activations.

Some Definitions

- **Timing constraint:**
  - Constraint imposed on timing behavior of a job: hard, firm, or soft.
- **Release Time**:
  - Instant of time job becomes available for execution.
- **Deadline**:
  - Instant of time a job's execution is required to be completed.
- **Response time**:
  - Length of time from release time to instant job completes.

Example Using the Definitions: Periodic jobs

- Consider a processor where you have two different tasks.
  - We'll call them τ_1 and τ_2.
  - It is commonly the case that the period of a task is the same as the time you have to complete the task.
    - So if τ_1 needs to run every 5 seconds, you just need to finish a given instance of τ_1 before the next one is to start.

- Consider the following:

	Period
τ_1	7
τ_2	16
τ_3	31

- Assuming all tasks are first released at time 0:
  - What is the deadline of the first instance of τ_1?
  - What is the release time of the second instance of τ_1?
  - What is the deadline of the second instance of τ_1?

Properties for Scheduling tasks

- **Priority**
  - If two tasks are both waiting to run at the same time, one will be selected. That one is said to have the higher priority.
- **Fixed/Dynamic priority tasks**
  - In priority driven scheduling, assigning the priorities can be done statically or dynamically while the system is running
- **Preemptive/Non-preemptive tasks**
  - Execution of a non-preemptive task is to be completed without interruption once it is started
  - Otherwise a task can be preempted if another task of higher priority becomes ready
- **Multiprocessor/Single processor systems**
  - In multiprocessor real-time systems, the scheduling algorithms should prevent simultaneous access to shared resources and devices.

Preemption

What it is and how it helps

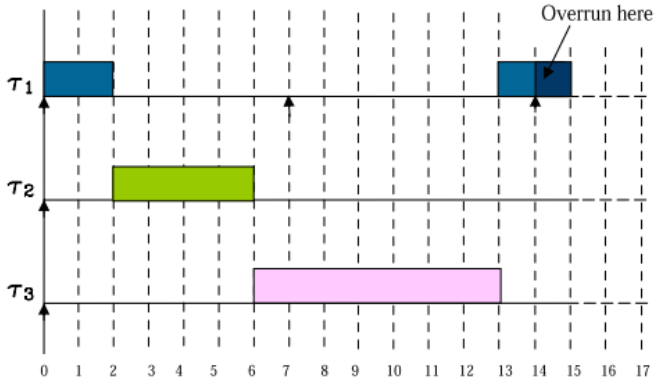


Figure 1: Priorities without preemption

Assume all tasks are released at time 0.

	Priority	*Period*	*Computation time*
τ_1	1	7	2
τ_2	2	16	4
τ_3	3	31	7

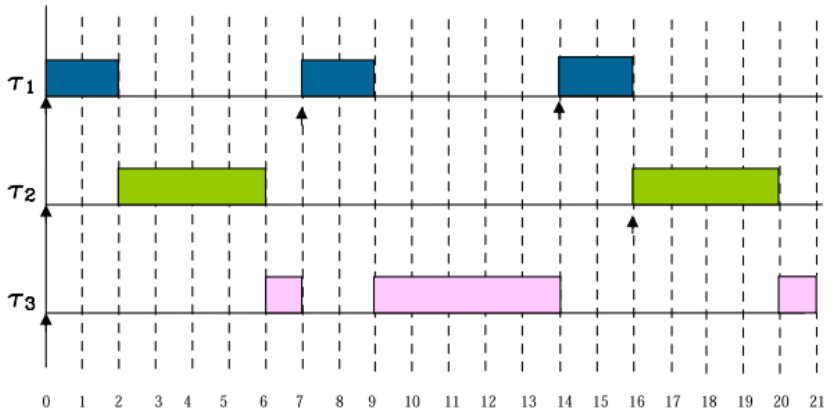


Figure 2: Priorities with preemption

Mohammadi, Arezou, and Selim G. Akl. "Scheduling Algorithms for Real-Time Systems." (2005)

Scheduling algorithms

- A scheduling algorithm is a scheme that selects what job to run next.
  - Can be preemptive or non-preemptive.
  - Dynamic or static priorities
  - Etc.

Two scheduling schemes

- **Rate monotonic (RM)**
  - Static priority scheme
  - Simple to implement
  - Nice properties
- **Earliest deadline first (EDF)**
  - Dynamic priority scheme
  - Harder to implement
  - Very nice properties

RM and EDF *assumptions*

- No task has any non-preemptable section and the cost of preemption is negligible.
- Only processing requirements are significant; memory, I/O, and other resource requirements are negligible.
- All tasks are independent; there are no precedence constraints.

Terms and definitions

- **Execution time of a task** - time it takes for a task to run to completion
- **Period of a task** - how often a task is being called to execute; can generally assume tasks are periodic although this may not be the case in real-world systems
- **CPU utilization** - the percentage of time that the processor is being used to execute a specific scheduled task

$$U = \frac{e_i}{P_i}$$

  - where e_i is the execution time of task i, and P_i is its period
- **Total CPU utilization** - the summation of the utilization of all tasks to be scheduled

$$\sum_{i=1}^{n} \frac{e_i}{P_i}$$

RM Scheduling

- It is a **static**-priority preemptive scheme involving periodic tasks only.
  - Well, it mumbles about non-periodic tasks, but…
- Basic idea:
  - Priority goes to the task with the lowest period.

Mohammadi, Arezou, and Selim G. Akl. "Scheduling Algorithms for Real-Time Systems." (2005)

How well does RMS work?

- Surprisingly well actually.
  - Let *n* be the number of tasks.
  - If the total utilization is less than $n(2^{1/n}-1)$, the tasks are schedulable.
    - That's pretty cool.
      - At n=2 that's ~83.3%
      - At n=∞ that's about 69.3%
  - This means that our (extremely) simple algorithm will work if the total CPU utilization is less than 2/3!
    - Still, don't forget our assumptions (periodic being the big one)
- Also note, this is a sufficient, but not necessary condition
  - Tasks may still be schedulable even if this value is exceeded.
    - But not if utilization is over 100% of course…

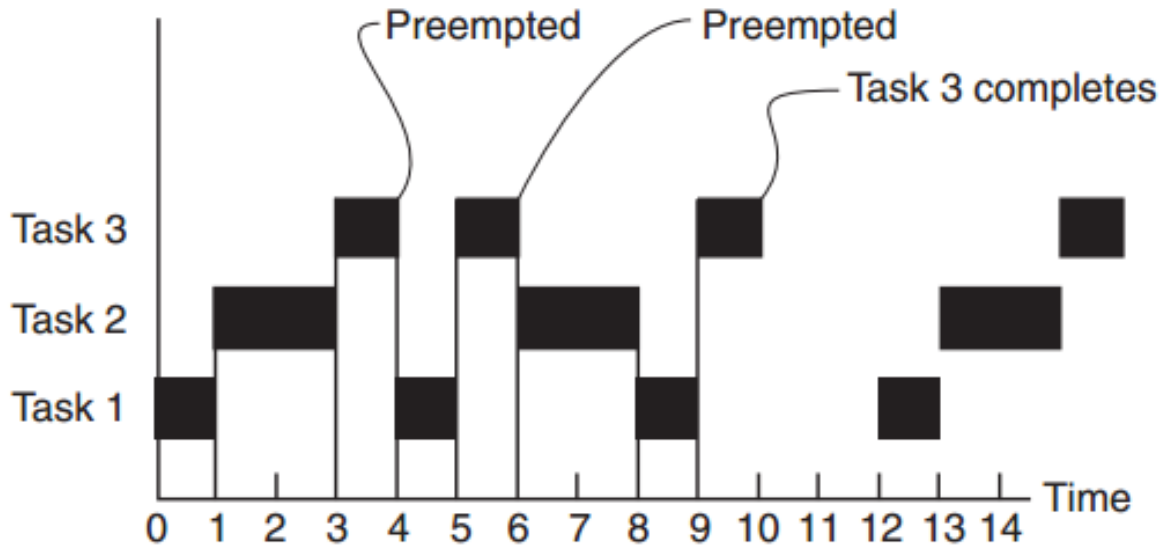
http://cn.el.yuntech.edu.tw/course/95/real_time_os/present%20paper/Scheduling%20Algorithms%20for%20Multiprogramming%20in%20a%20Hard-.pdf

What if the sufficiency bound is not met?

- Critical instant analysis
  - The worst case for RMS is that all tasks happen to start at the exact same time.
    - If RM can schedule the first instance of each such task, the tasks are schedulable.
- With RM scheduling we can always jump to doing the critical instant analysis

Example #1

Task	Execution Time	Period	Priority
T1	1	4	High
T2	2	6	Medium
T3	3	12	Low



http://www.idsc.ethz.ch/Courses/embedded_control_systems/Exercises/SWArchitecture08.pdf

Example #2

Task	Execution Time	Period	Priority
T1	1	3	High
T2	1.5	4	Low

Example #3

Task	Execution Time	Period	Priority
T1	1	3	High
T2	2.1	4	Low

Example #4

Task	Execution Time	Period	Priority
T1	1	2	High
T2	2	5	Low

Easy?

- Fixed priorities are pretty easy.
  - Scheduler easy to write.
  - Could, in theory, use interrupt priorities from a timer to do this
    - But that's probably dumb (why?)