ECEN 3753: Real-Time Operating Systems

Introduction and Overview

whoami

Course Goals and Expectations

- See Syllabus (separate)
- Come to lecture to get the theory
- Read the pre-reading to enhance lectured content
- Do the lab assignment. If it's not making sense:
  - Make sure you've done the relevant pre-reading, as it may help fill a gap
  - THEN come to office hours, or seek a classmate's help to understand better.
- Occasionally, post-lecture links will be provided for further reading. The intention here is to enrich your education, not to add more content for which you'd be tested.

Attribution

This course's presentation material has been created for the University of Colorado, Boulder, with extensive support in the form of materials (slide content and ideas) from both our selected textbook (Real-Time Operating Systems, Book 1: The Theory, by Jim Cooling), and training materials provided by Silicon Labs.

Additionally, Silicon Labs has provided SDKs for the coursework developers. It is the hope of them and me that students gain an appreciation for being able to use a quality commercial tool for debug.

Appreciation

In addition to Silicon Lab's support of this class's development, a group of engineers at Western Digital volunteered to jointly develop most of the lecture and lab material for this course.

Adam Harper	Biping Wu	Chad Hahnlen
Ed Hoskins	Isaac Muse	Steve Love

This Week's Lab

- If you have not already done ECEN3360.Lab1 or are not comfortable with the SDK, this is your chance. If you already have and are, this'll be a virtual vacation of a lab week.
- You should have acquired the Pearl Gecko development board already
- Get ECEN3360.Lab1 from our git repo and follow the instructions therein (SiLabs account, Simplicity IDE, Segger SystemView...)

What You're Doing With This Lab

In general in this class's future labs, RTOS and BSP functions will help you to more easily do what you did in a "bare metal" manner previously.

Explore the tools

- git
- Boot (what level of analysis/tracing can you reasonably do?)
- Power Profiler
- Debug (single-step software breakpoints)
- Segger System View

Slide Format

Continuity with ECEN 3360:

- Information
  - *Example or Narrative to illustrate a point*
- Discussion point or Question to answer
- Don't miss these!

Baseline question for this semester

Shall silence be YES or NO?

Credit to Prof. Adam Norris, who made enough of an impact with this that I heard about it through a student

Reading Assignments

Value proposition:

- Will benefit your understanding/context of week's topics
- Required. (Some is not until later, but...)

Recurring Theme: Use of Real-life Analogies

How we start, run, and debug computing environments has parallels to our carbon-based life.

Some of our first topics:

- Bootstrapping
- Scheduling and Execution
- Debugging

Carbon-based Booting

https://en.wikipedia.org/wiki/Bootstrapping

Carbon-based Booting

https://en.wikipedia.org/wiki/Bootstrapping

How did you start your day? (Provide some invariant parts of your routine)

Carbon-based Booting

https://en.wikipedia.org/wiki/Bootstrapping

How did you start your day? (Provide some invariant parts of your routine)

- Woke up
- Got out of bed
- Put on selection of clothing
- Put on footwear
- Recalled your planning system
- Compared day and time against your system to determine first activity to perform
- Transported to first activity

Human-scale Scheduling and Execution

Review the plan and priorities, and start executing to the plan.

Plan/priorities:

- *I have to be at class at 10A*
- *I want to go climbing before the air gets hot*
- *It's 8AM, so I'm going to go climbing first.*

Execution:

- *Drive to climbing site*
- *Climb, enjoy the view and some snacks, climb back down, gather gear*
- *Drive back to campus*

Human-scale Scheduling and Execution

Review the plan and priorities, and start executing to the plan.

Plan/priorities:

- *I have to be at class at 10A*
- *I want to go climbing before the air gets hot*
- *It's 8AM, so I'm going to go climbing first.*

Execution:

- *Drive to climbing site*
- *Climb, enjoy the view and some snacks, climb back down, gather gear*
- *Drive back to campus*
- ***Arrive at class at 10:15A***.

Human-scale Logging/Debugging

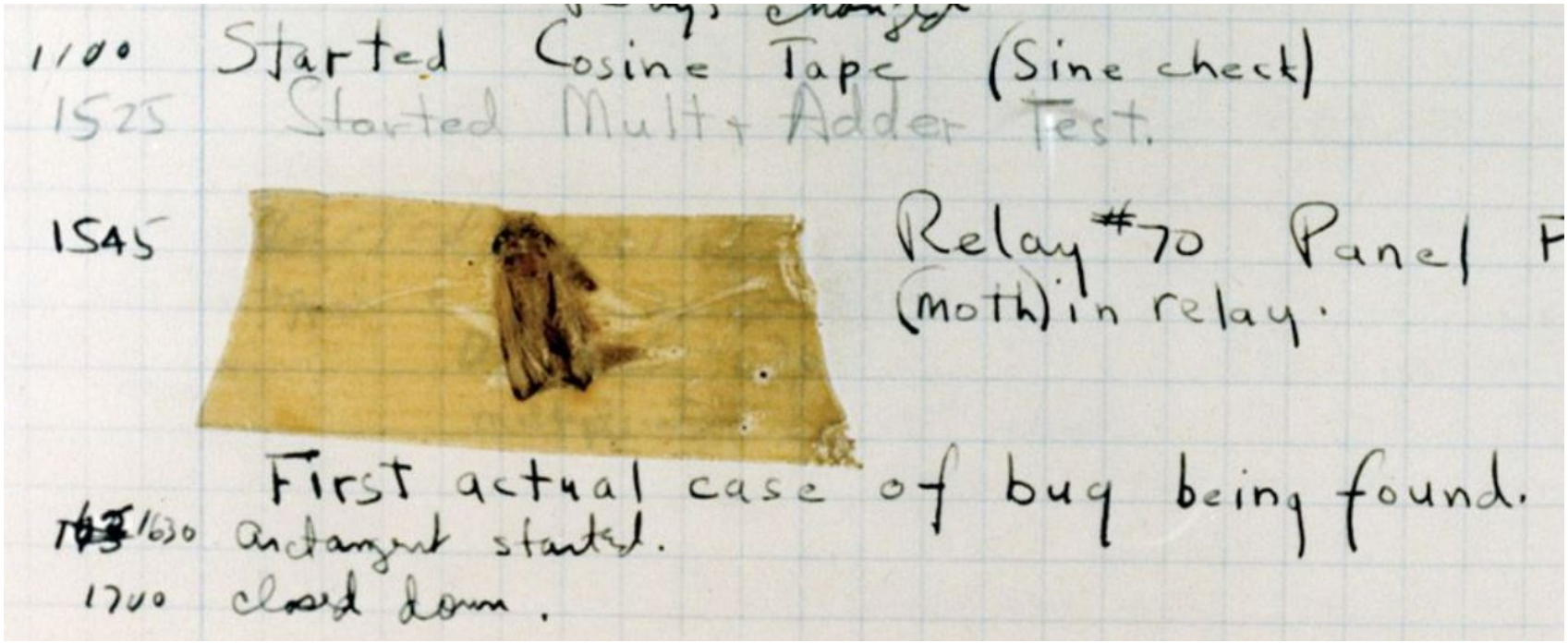
If/**when** things go off the rails and there's a train wreck before your eyes, think back to what happened BEFORE...

- *I didn't absorb what I needed to in class since I arrived late. Maybe going to go climbing at 8A was a poor plan, because there wasn't enough time to get back to campus afterwards.*

In the "Pre-Reading" video this week:

- Was the moment that we observed the derailment very relevant to learning why the train derailed?
- What early indicators were there?
- What other "debugging" info would the NTSB wish they had?
- Would it be worthwhile to plaster cameras and other recording equipment over every face of every car on a train? Why or why not?

In Memoriam: Grace Hopper (1906-1992)



Bring these 3 topics into the Embedded RT space

- Booting
- Scheduling/Execution
- Logging/Debugging

. . . over this course.

What is an OS?

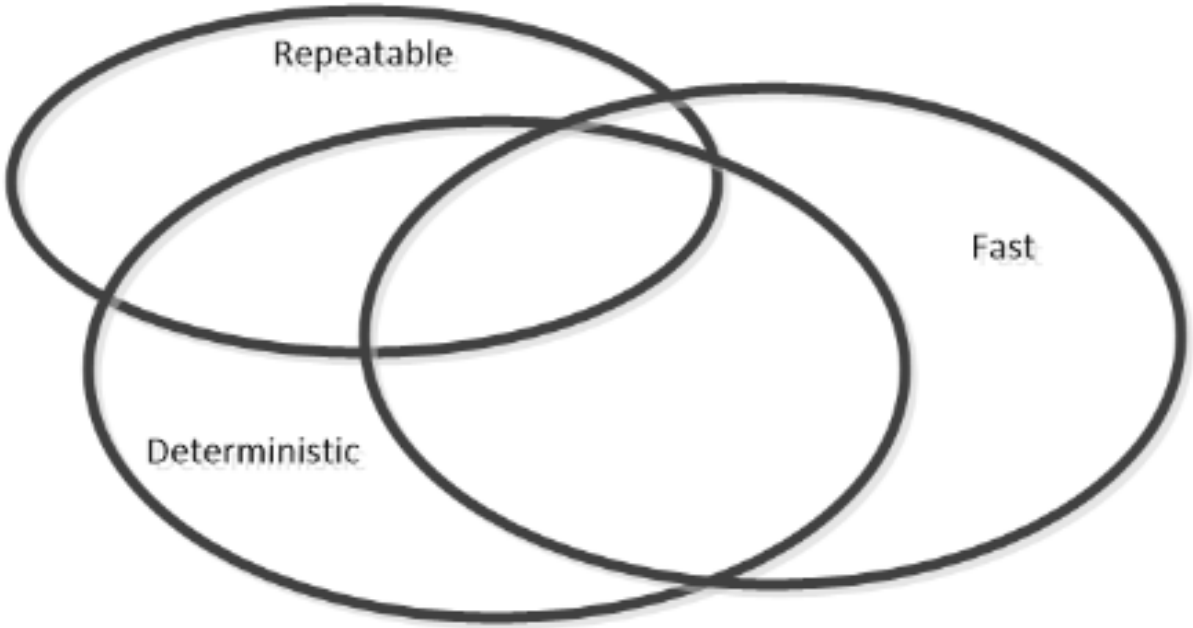
Software Providing:

- Encapsulated/standardized access to system resources
  - Memory
  - CPU
  - Storage (FileSystems, e.g. on HDD/SSD) & Networking
  - Boot
- Means to exclusively-use some resources
  - Virtual Memory
  - Abstract Processors (and scheduling algorithms to share real processors)
- Communication methods between requestors for sharing work and resources
- Security between data users when sharing is not appropriate
- Board Support Package (BSP) is not technically part of, but is usually provided with a RTOS, to encapsulate the hardware available in a particular embedded system

Generalized OS Boot Sequence – before main()

- Processor fetches code from ROM at boot address
  - May be slow
  - Interrupts are off, Exception table in ROM
- Initialize GPIO
  - Minimalist User Interface (UI) configured for reporting catastrophic boot errors before "full" UI (LEDs, beeps)
- Adjust clocks
  - Sometimes transitioning to faster execution
- Initialize / test memories
- Load any more code into RAM (from other places, possibly requiring additional initialization. . .)
  - Decompress if needed
    - Under what constraints would compression make sense?
  - Run from RAM

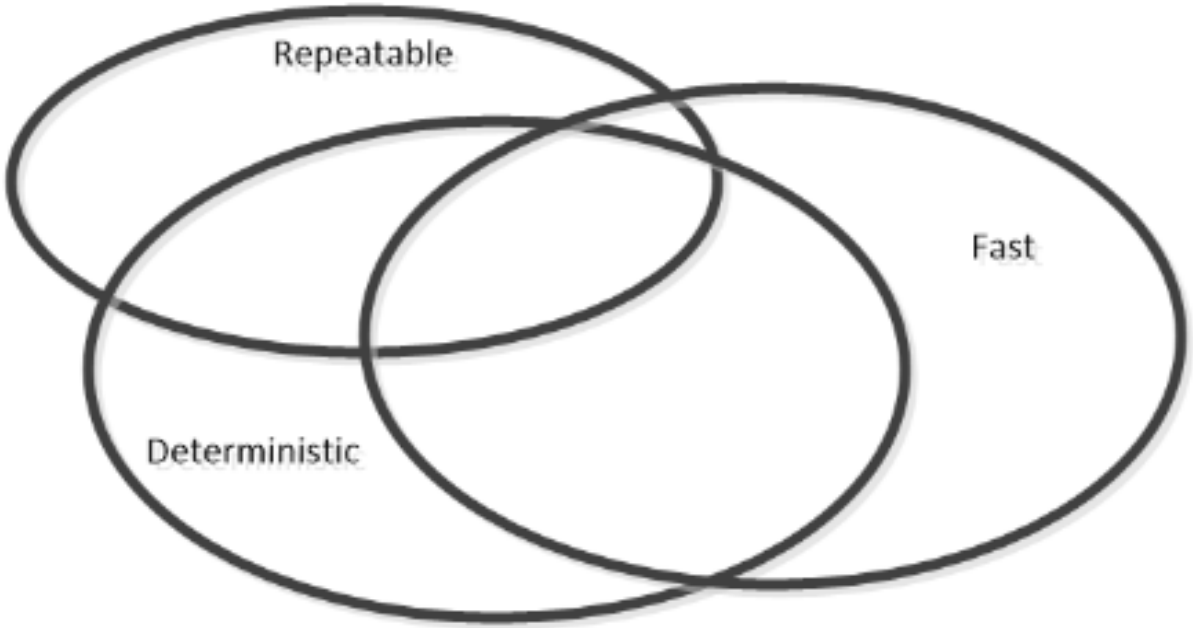
Is a "Real-Time"" system the same as "Fast"?



"Real-Time" means that a system is **deterministic**. That is, it will consistently meet all of the required time limits.

Is (Fast and Deterministic) better than (Slow and Deterministic)?

Is a "Real-Time"" system the same as "Fast"?



"Real-Time" means that a system is **deterministic**. That is, it will consistently meet all of the required time limits.

Is (Fast and Deterministic) better than (Slow and Deterministic)?

- It Depends.

What is a Non-RT OS missing?

- Time Control (Deadlines, determinate behavior)
- Tightly-Coupled Coordination of Work or Shared Resources
- Constrained Scale (more costly resources and support systems are often required)

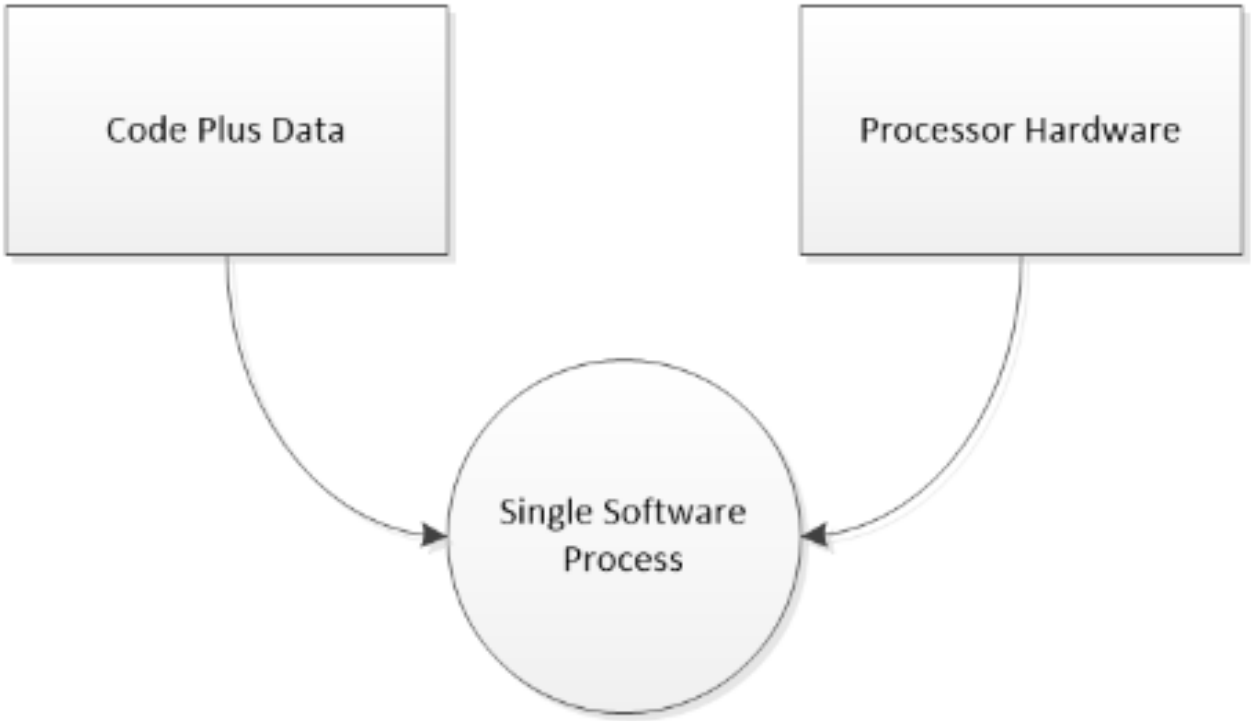
OS	RTOS	WinCE	Linux (server)	MSWin
Response	D, Fast	(N)D,Fast	ND, Very Fast	ND, *
Footprint	Tiny	Small	Small-Medium	Large

Communicating an Embedded, Real-Time Framework

We've got real (time) problems to solve...

How are we going to describe the solutions?

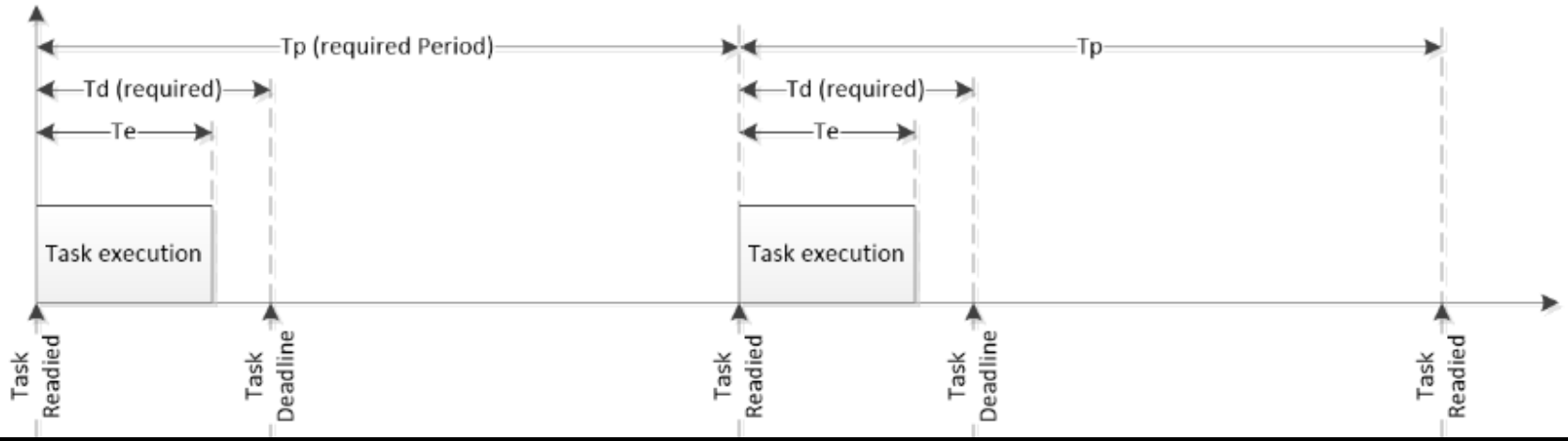
Single Software Process



A **Program** is the non-volatile storage of one or more processes' descriptions that will lead to **Process**(es) when it is brought into a real computing environment on physical processor(s), and set up properly.

Process components

- Multiple, separable computational efforts can be treated separately to allow simpler, more robust solutions to be applied to each
- Some parts will require periodic work, some will likely be purely reactive, non-periodic work
- If sufficient processing resources are available, each task can effectively run at a periodicity that's needed (T_p), and with reasonable fulfilment within a deadline (T_d).
- Responsiveness: ($T_e > T_d$) = Missed Time Limit!



Classification of Process Components

A Process can be broken down into a plurality of one of these lighter-weight constructs that share the process memory, and some resources.

- Threads are lightweight processes, aligned to separable flows
  - Can be used to help group work in parallel computing or asynchronous programming
- Tasks are lightweight process components, aligned to abstract work that must be done
  - May be used in a degenerate case for synchronous programming

Stack is a non-shared resource for these components.

Types of RealTime Systems

Classify by the impact if time limits are not met:

- **Hard RT**: If the time limit is not met, damage (sometimes catastrophic) can result
- **Firm RT**: An occasional missed time limit is still a bad thing, and could cause damage or out-of-design-goal performance if left unmitigated. Mitigation is implemented when this is a known possibility.
- **Soft RT**: Missed time limits will degrade performance, but in a non-damaging manner
- **Non-RT**: *Time limits? What time limits?*

Classify RT Sytems

What RT type do you think these are, and why?

(1:Hard, 2:Firm, 3:Soft, 4:Non)

- National Power Grid Protection Relay
- Refridgerator compressor control
- Microwave Oven Door Monitoring
- Solar Panel Grid-Tied Power Inverter
- Core Network Router
- Airplane Stall Mitigation
- Bitcoin mining

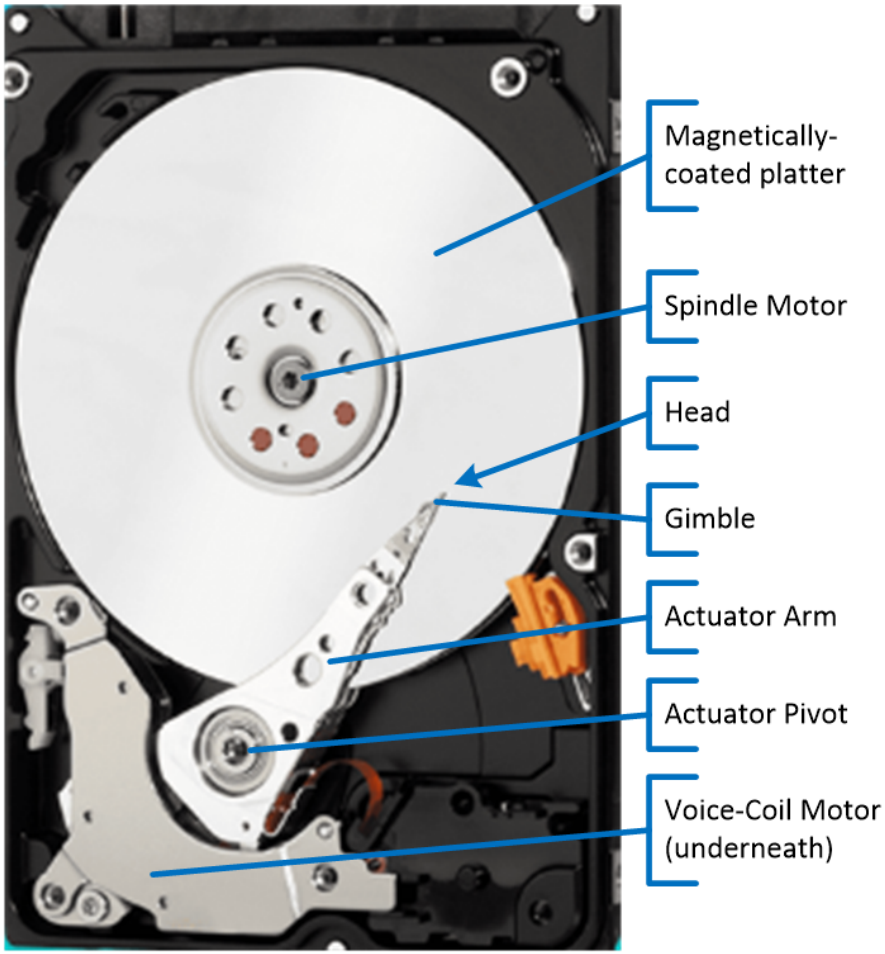
Approximately:

What RT type do you think these are, and why?

(1:Hard, 2:Firm, 3:Soft, 4:Non)

- National Power Grid Protection Relay (Hard)
- Refridgerator compressor control (Soft)
- Microwave Oven Door Monitoring (Hard*)
- Solar Panel Grid-Tied Power Inverter (Hard)
- Core Network Router (Firm)
- Airplane Stall Mitigation (Firm*)
- Bitcoin mining (Non)

Classify an HDD Storage System



Head: Radial Velocity o(100 in/s) max.
Servo position: sampled o(10KHz)

What kind of Real-time classification is the position-sampling system and its feedback control?

Angst!

Everything costs money

- ADC/DAC
- Processors
- Memory
- Power

Solution?

Angst!

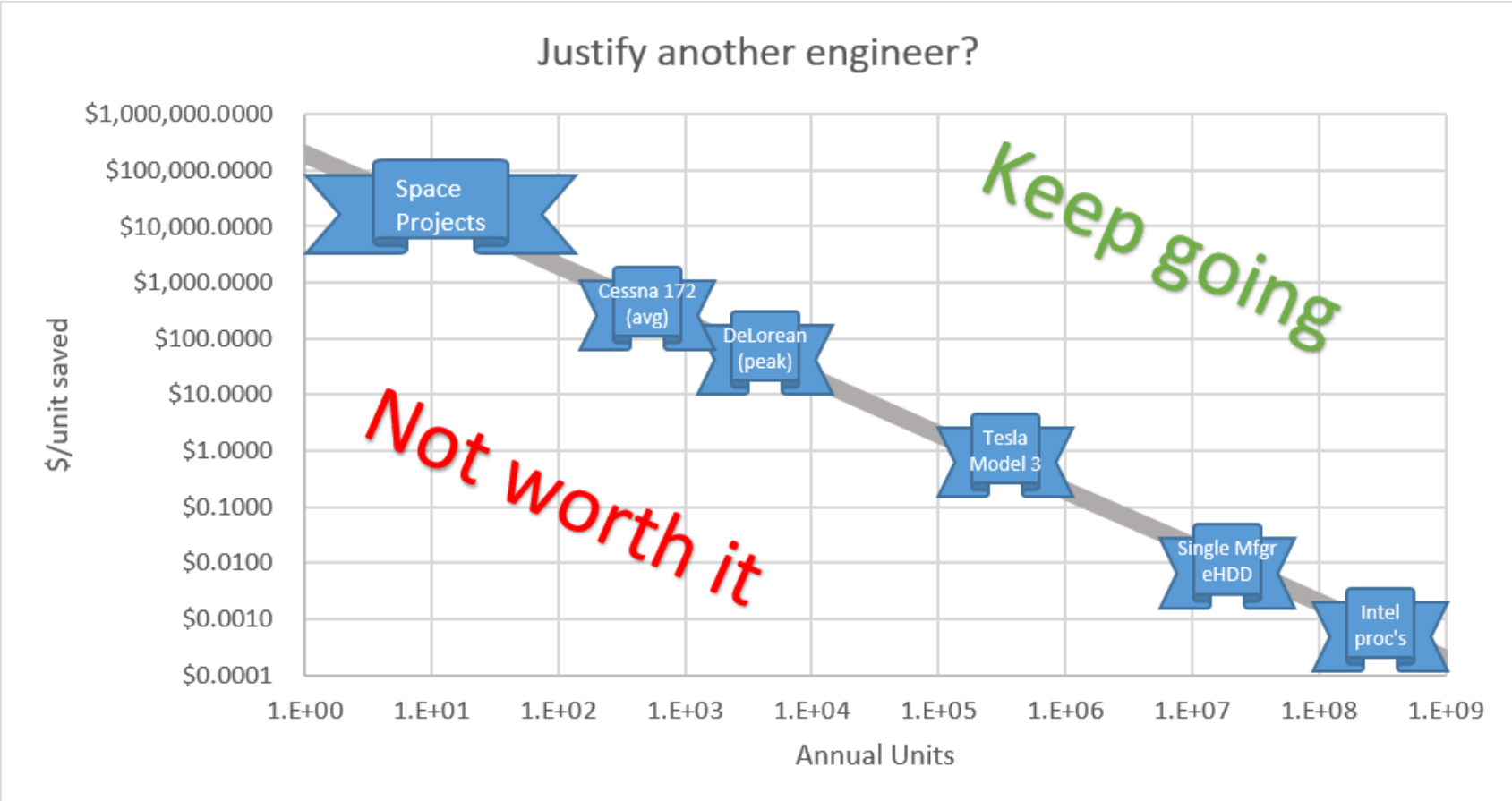
Everything costs money

- ADC/DAC
- Processors
- Memory
- Power

Solution?

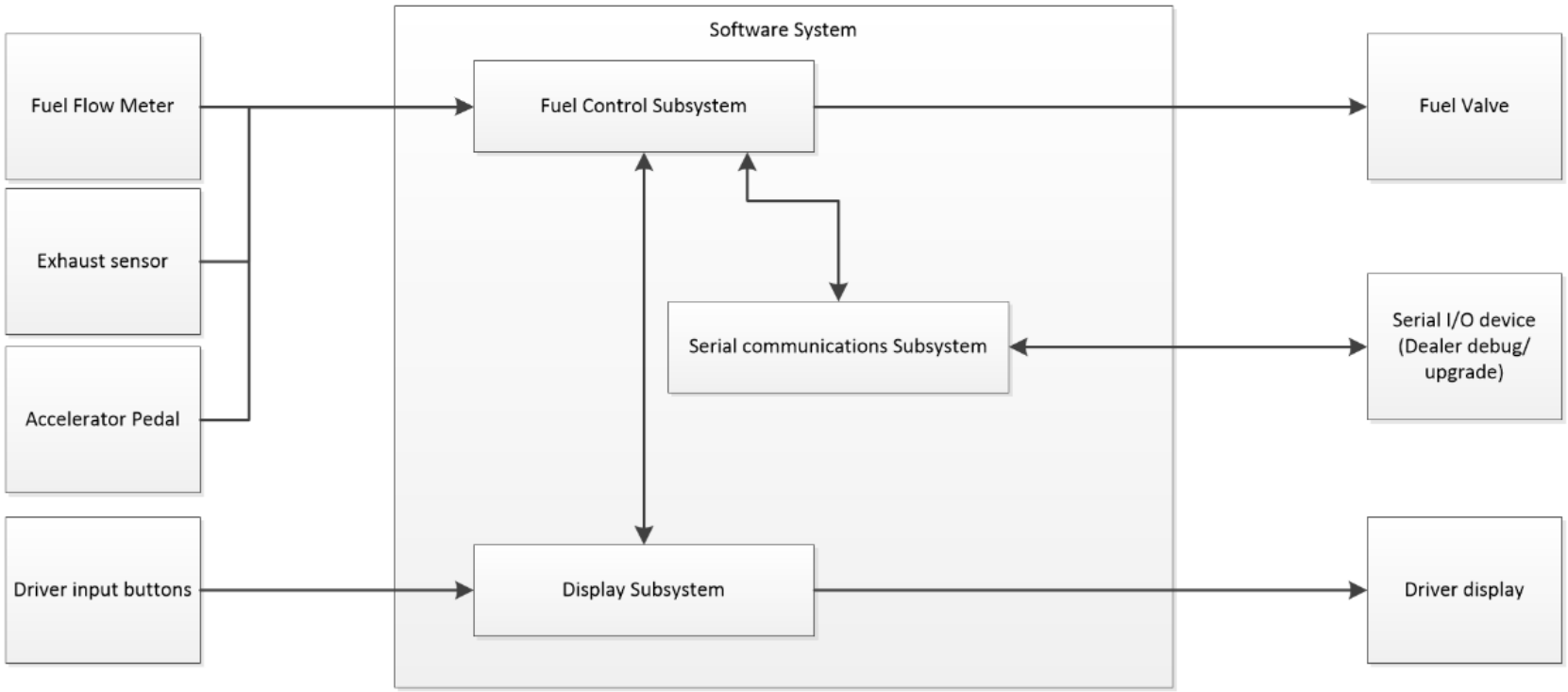
- Share resources.

Cost Savings, at Scale



If you're going to make enough of something, extra engineering to reduce per-item costs can be compelling.

Resource Sharing – Software Design Model



Super-loop Construction

Using the text's simple "Super-Loop" method of doing each task to completion, we could code this system as:

```
{
  Measure Fuel Flow
  Measure Exhaust Characteristic
  Measure Accelerator Position
  Compute New Fuel Valve Setting
  If (Incoming Data on Diagnostic Port)
    Handle Diagnostic Command
  If (User has pressed a dashboard button)
    Handle User Command
    Update Display
}
```

Super-loops Incur Maintenance

```
{
  Measure Fuel Flow
  Measure Exhaust Characteristic
  Measure Accelerator Position
  Compute New Fuel Valve Setting
  If (Incoming Data on Diagnostic Port)
    Handle Diagnostic Command
  If (User has pressed a dashboard button)
    if (Part == 0)
      Handle User Command
    Update Display (Part = (Part++ % NUM_PARTS)) // 0..N-1
}
```

Simple enhancement: break display update into roughly-similar-sized parts at convenient points.

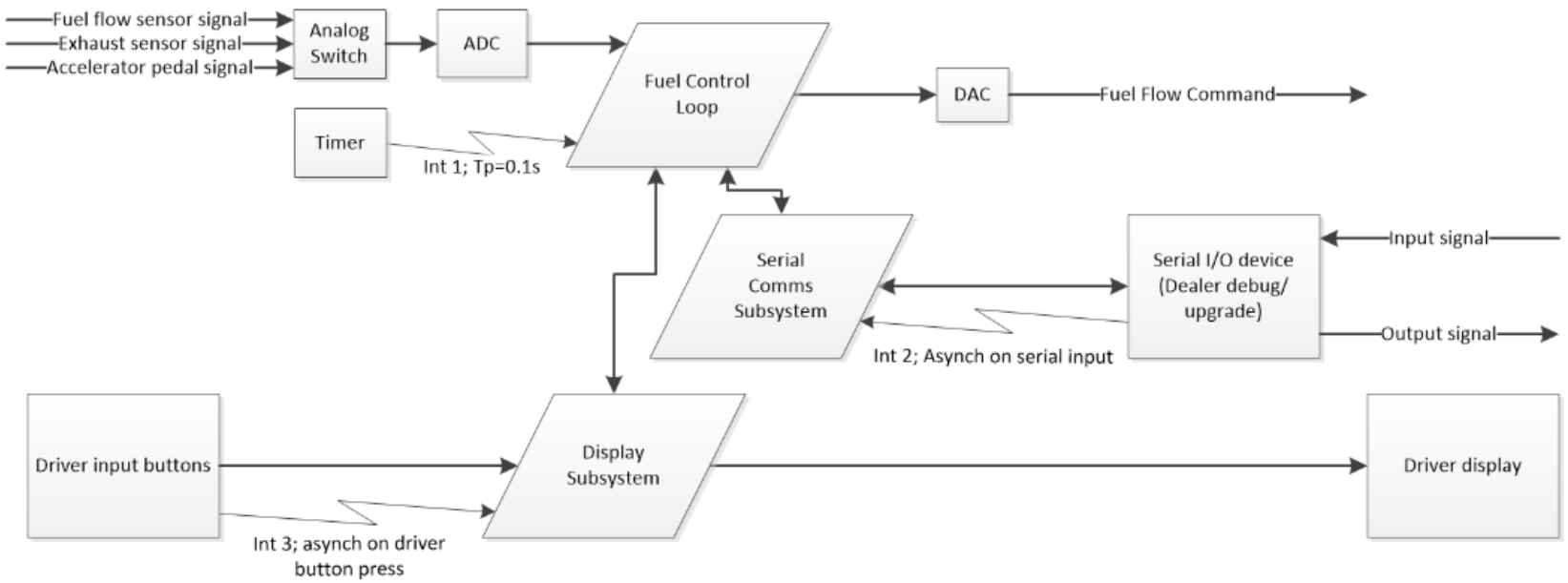
And introduce bugs by trying to plan ahead...

Super-loop with Yield()-ing

```
{
  Measure Fuel Flow (till yield)
  Measure Exhaust Characteristic (till yield)
  Measure Accelerator Position (till yield)
  Compute New Fuel Valve Setting (till yield)
  If (Incoming Data on Diagnostic Port)
    Handle Diagnostic Command (till yield)
  If (User has pressed a dashboard button)
    Handle User Command (till yield)
    Update Dashboard (till yield)
  Measure Fuel Flow (till yield--immediate if 1x)
  Compute New Fuel Valve Setting (till yield--immed if 1x)
}
```

Operating Systems can make this a bit simpler to program by allowing a task to yield() to the next (AKA sleep(0)).

Interrupt-driven Quasi-Concurrency



As long as there is sufficient processing capability in the shared physical processor, and there are no contentions for shared resources that interfere, could share a processor by reacting to interrupts for asynchronous events, and timers for periodic processing.

What would happen if one of the asynch $T_e > 0.1s$?

More that happens before main()...

- Compiler-directed initialization
  - Static construction / zero initialization
    - .text, .data, .bss
- Initialize/enable interrupts
- Additional HW initialization for the OS
- OS initialization
  - Tick timer
  - Idle task

What happens AFTER main()?

- Compiler related stuff
  - Static destruction
- Exit/terminate handling
  - Non-RT: return to prompt, close window, power-off, etc.
  - RT: There are no good choices. Recall: for Hard-RT, "physical damage will result"
    - Reboot
    - Hang
    - Crash / bluescreen

What happens AFTER main()?

- Compiler related stuff
  - Static destruction
- Exit/terminate handling
  - Non-RT: return to prompt, close window, power-off, etc.
  - RT: There are no good choices. Recall: for Hard-RT, "physical damage will result"
    - Reboot
    - Hang
    - Crash / bluescreen

A moving car with computer-controlled braking can become. . .

What happens AFTER main()?

- Compiler related stuff
  - Static destruction
- Exit/terminate handling
  - Non-RT: return to prompt, close window, power-off, etc.
  - RT: There are no good choices. Recall: for Hard-RT, "physical damage will result"
    - Reboot
    - Hang
    - Crash / bluescreen

A moving car with computer-controlled braking can become. . .

A stationary car with brakes latched which is now safe to. . .

What happens AFTER main()?

- Compiler related stuff
  - Static destruction
- Exit/terminate handling
  - Non-RT: return to prompt, close window, power-off, etc.
  - RT: There are no good choices. Recall: for Hard-RT, "physical damage will result"
    - Reboot
    - Hang
    - Crash / bluescreen

A moving car with computer-controlled braking can become. . .

A stationary car with brakes latched which is now safe to. . .

Power off (including exiting the brake-control task).

Value of Frameworks

What are FRAMEWORKS for?

Value of Frameworks

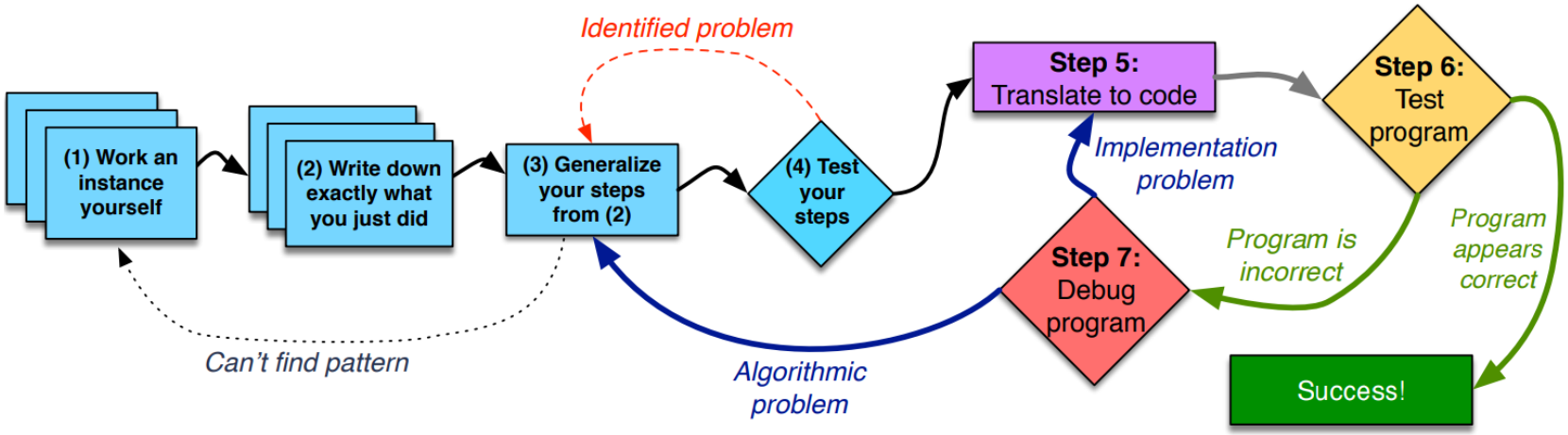
What are FRAMEWORKS for?

- *To guide inexperienced people to stay within bounds known to work?*
- *To ALWAYS force solutions to stay within the framework?*
- *To help provide a starting point for deconstruction of complexity? (. . . that has proven useful in the past, or by experts looking for how to help in the future)*

absolutes are rarely correct.

Very Useful Problem-Solving Method

This 7-Step Method was generated by Andrew Hilton, Genevieve Lipp, and Susan Rodger at Duke University.



Load/Store a redundant boot-time data set

<image from http://adhilton.pratt.duke.edu/sites/adhilton.pratt.duke.edu/files/u37/iticse-7steps.pdf>

GPIO, Interrupts, and Timers

Your connections to the real world, with a sense of time–

The enablers for an Embedded, Real-Time system!

We animals are amazing embedded *biological* systems!

Your connections to the real world, with a sense of time–

The enablers for an Embedded, Real-Time system!

We animals are amazing embedded *biological* systems!

In our carbon-based analogue, if we didn't have any of our 5 senses (Inputs), and the ability to act upon our environment (Outputs) with an awareness of time, we would be as unfit-for-use as an old PC on a dusty workbench, powered up and clocking a CPU, but without any useful effect (other than as a space-heater)!

General Purpose Input/Output (GPIO)

- Enabling
- Activity level (high/low)
- Electrical configuration
- Routing configuration

For our embedded systems' senses and muscles to be effective, we need to wire things up the right way...

General Purpose Input/Output (GPIO)

- Enabling
- Activity level (high/low)
- Electrical configuration
- Routing configuration

For our embedded systems' senses and muscles to be effective, we need to wire things up the right way...

- Choosing which inputs and outputs to use
- Setting them up to use/expect the right polarities
- ...with appropriate sensitivities/strengths
- Using the right neural pathways for the right activities

We're not yet talking about priorities or time sensitivity...but we'll get there soon.

General Purpose Input Electrical Configuration

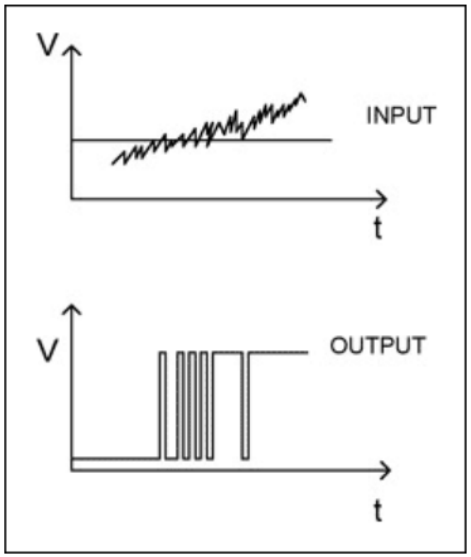
Options:

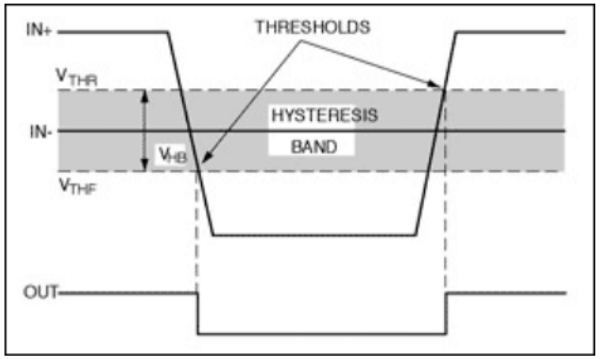
- Internal Pull-up resistor
- Internal Pull-down resistor
- High-impedence (no internal resistor)

Inputs are generally internally wired with hysteresis.

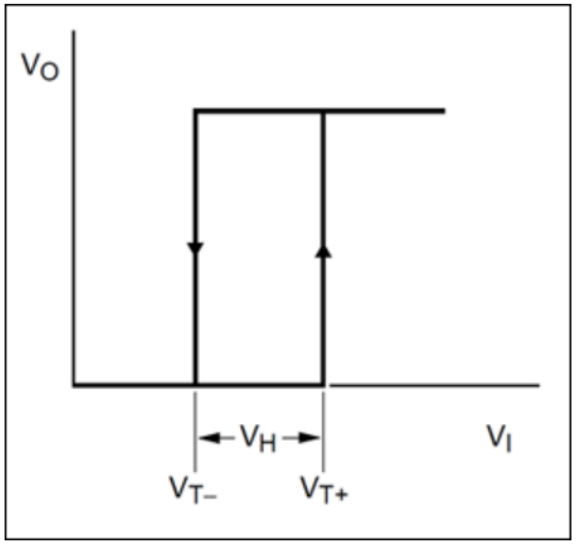
Hysteresis is helpful on inputs, to keep our real-time embedded system from being pestered by the equivalent of a flickering light bulb.

Input Hysteresis





From: electronics.stackexchange.com,
in: ./questions/12312/what-is-input-hysteresis
by: ./users/2064/stevevh



- What you DON'T want to happen with a noisy input is shown on the left,
- A plot of behavior we DO want (with a clean input) is in the center.

- Alternatively you can view this as a transfer function, on the right.

General Purpose Output Electrical Configuration

Options:

- Push/Pull (MOSFET above and below, can actively source or sink current)
- Open Drain(FET)/Collector(BJT): can sink current. *Wired-OR*
  - With internal resistor (often insufficient)
  - Without internal resistor (What's needed?)
- "Speed", "Slew-rate", "Frequency", or a "current-limit" setting
  - Helps drive the output *transition* more aggressively
  - May generate more noise
  - May use more power Lab1: Did it significantly? Reason?

When setting up the "muscles"" of our embedded system, we need to configure the electrical properties to ensure that we can drive the eventual states the way we need. There is often an option about whether to put the muscles on steroids–if we do, there may be side effects.

GPIO Routing

On-chip and off-chip (de)multiplexors and analog switches allow general designs of chips to be useful in more solutions.

Designs likely need to optimize across several factors:

- Switching I to/from O requires all of the rest of the path to be resilient to this
  - Can you think of any I/O usage that switches back-and-forth?
- Keeping settings static (current/speed, multiplexors) especially on high-usage paths
- Switching settings more on slower-sampled signals may be acceptable
- Desire for dedicated pins (debug, board connects, PCB optimizations)

GPIO Routing

On-chip and off-chip (de)multiplexors and analog switches allow general designs of chips to be useful in more solutions.

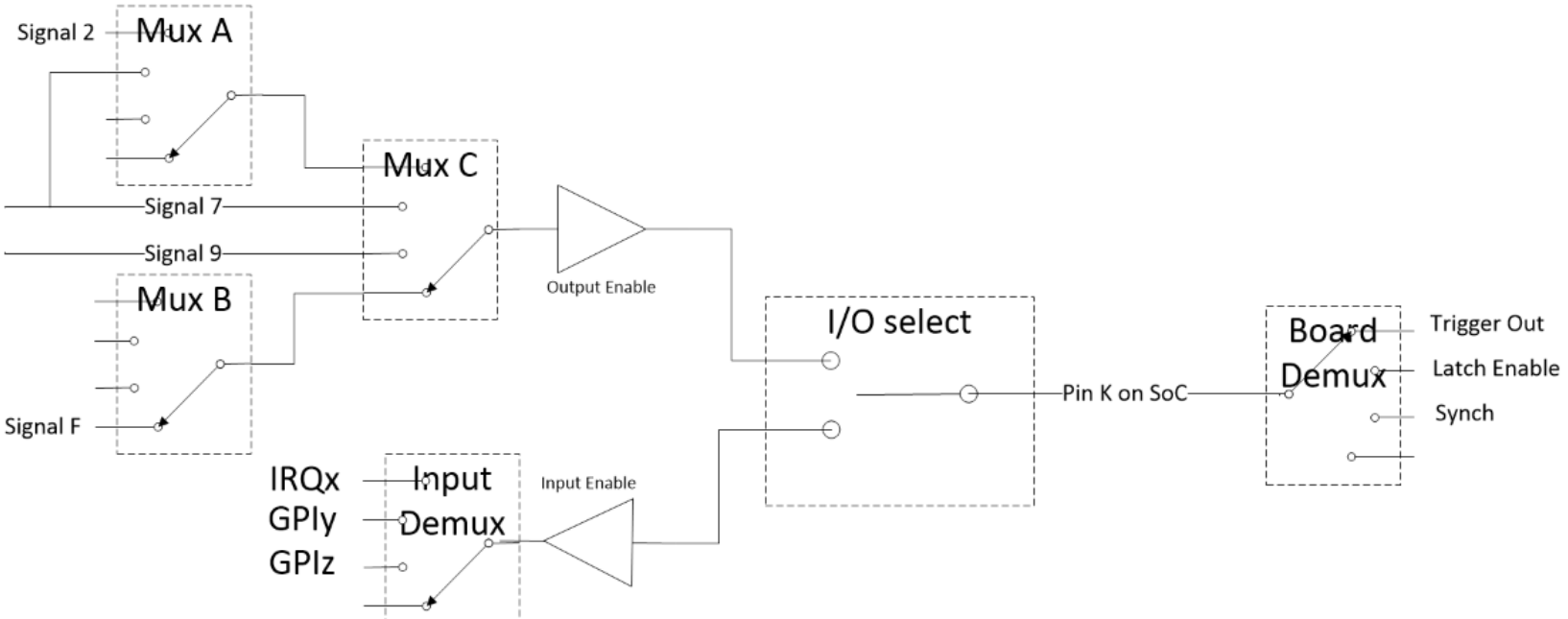
Designs likely need to optimize across several factors:

- Switching I to/from O requires all of the rest of the path to be resilient to this
  - Can you think of any I/O usage that switches back-and-forth?
- Keeping settings static (current/speed, multiplexors) especially on high-usage paths
- Switching settings more on slower-sampled signals may be acceptable
- Desire for dedicated pins (debug, board connects, PCB optimizations)

I2C : What I/O flipping does SerialDAta enjoy?

GPIO Routing example

- Signal 9 is needed for Synch:
  - MuxC=10b; I/O:O; BoardDemux:10b
- Signal 7 needs to route to TriggerOut (2 ways!):
  - MuxC:01b; I/O:0; BoardDemux:00b
  - MuxA:01b; MuxC:00b; I/O:O; BoardDemux:00b
- More complex when multiple pins are available, with different routing options. Why would these exist?



Interrupts

Historically, an "interrupt" was an external event requiring action. However, terminology is now ambiguous. (Internally-generated exceptions are sometimes called interrupts too (SWI, NMI), and the M4 manual refers only to "exception handlers".)

- Setup required in advance:
  - Configuration of Interrupt Controller (incl. ISR location setup)
  - Enablement of IRQ at CPU (sometimes an additional step)
  - Level/latched-edge configuration
- Required at time-of-interrupt:
  - *Shared resources are needed for the interrupt to be serviced.*
  - *ISR code is at a known location.*
  - *Data may be pre-fetched from known locations for ISR operation.*

What Shared resources are needed?

What Needs to Happen when Execution is Interrupted?

The non-ISR code execution **context** must be saved so that the non-ISR work can pick up where it left off, later.

- What, Where, and How much?
  - In addition to the CPU itself, CPU registers (including the CPSR, PC, LR) are shared resources, but the values in these registers at any point in time are specific only to the currently running code, not JOINTLY between the interrupted code and ISR. So if the ISR is going to use any particular registers, their pre-interrupt content must be saved at start-of-ISR, and restored at end-of-ISR.
  - The stack seems like a reasonable default location to store the registers' prior contents...
  - Why is the amount not straightforward?

Why the sizeof(context) isn't obvious

- Simpler encapsulation by providing more freedom to use the full register set with abandon, but
- Requires more:
  - time, if using stack
  - hardware gates, if using dedicated register banking

In custom-designed hardware, banking the full register set may not be cost-prohibitive. Example (c.2001) was a $1kUSD core internet router chip that was designed to fully pre-fetch an execution context for any one of hundreds-of-thousands of tasks and then to flush it back out after only getting about 40 cycles of the CPU in which to execute routing logic. Clearly, this was not a "general purpose" embedded controller–the cost was well-justified for its market.

Minimalist Interrupt Handling Hardware

Restating, when an interrupt is enabled and arrives with sufficient priority, at a minimum the hardware needs to:

- push register content that would otherwise be in the way (CPSR, Link Register at least) onto the stack
- copy the PC that was in use to the Link Register
- set the PC appropriately to handle the interrupt (start executing the ISR)

When we discuss shared resources and task switching, this same concept of the **context** will apply, as will the optimizing concerns about saving-and-restoring the "right amount".

ARM Interrupt controllers

The Generic Interrupt Controller (for the Cyclone V SoC with a pair of A9 cores) or the Nested Vectored Interrupt Controller (for our M4) provide support for:

- Edge/Pulse interrupt latching (Synchronous)
- Level interrupt handling
- Prioritization
  - Programmable with groups [GIC and NVIC]
  - From humble origins c.1976: Fixed / Rotating [Intel 8259]
- Processor/Operating Modes (select: Stack & HW accel.)
  - Thread/Handler [NVIC]
  - System/Supervisor/User/IRQ/FIQ/Abort/Undefined [GIC]
- Routing/"distribution" (to one or more of the cores) [GIC]

We will circle back to these in a few weeks, to look at ways in which hardware support can shorten latencies and control priorities for interrupts to make our system more responsive for real-time apps.

Interrupts: almost as good as another CPU?

Reflecting–an interrupt essentially:

- Is giving you the same benefit as a dedicated polling CPU that then hands off the "real work" to the main CPU (and ISR code) once "something is ready to be done".
- Is very cheap/easy to do with concurrent hardware, rather than CPU+PollingCode.

Rather than a super-loop's polling code executing as frequently as the loop allows, the pseudo-polling "code" is ALWAYS running in interrupt hardware, until the Interrupt happens.

When might CPU cycles make sense instead?

- on the main CPU
- on a baby core (e.g. M0)

Interrupts: almost as good as another CPU?

Reflecting–an interrupt essentially:

- Is giving you the same benefit as a dedicated polling CPU that then hands off the "real work" to the main CPU (and ISR code) once "something is ready to be done".
- Is very cheap/easy to do with concurrent hardware, rather than CPU+PollingCode.

Rather than a super-loop's polling code executing as frequently as the loop allows, the pseudo-polling "code" is ALWAYS running in interrupt hardware, until the Interrupt happens.

When might CPU cycles make sense instead?

- on the main CPU
- on a baby core (e.g. M0)

When Jitter is acceptable, requirements are fuzzy, algorithms half-baked.

Timer Services

What are timers used for?

Commonly 3 things:

Timer Services

What are timers used for?

Commonly 3 things:

- Stopwatch
- Holdoff
- Repeating

Micrium OS Timer Services

Software Timers used for:

- Duration Measurement (Stopwatch)
- Single Shot Timer (Need to do something *then* (watchdog), or have nothing to do until *then*)
- Periodic Timer (with optional delay, to trigger recurring time-based work)

What is the first thing you need to know about ANY timer for it to be useful?

Micrium OS Timer Services

Software Timers used for:

- Duration Measurement (Stopwatch)
- Single Shot Timer (Need to do something *then* (watchdog), or have nothing to do until *then*)
- Periodic Timer (with optional delay, to trigger recurring time-based work)

What is the first thing you need to know about ANY timer for it to be useful?

ClockTick: Timer Granularity (and inversely proportional: Time-to-Aliasing)

Hardware Timer Implementations of ClockTick

- Hardware Timer
  - Frequency Determination (that granularity question. . .)
    - Input clock source selection (frequency, power, cost, stability)
    - Configurable divider (e.g. TIMERn_CTRL.PRESC)
  - Reload methods
    - None (free-running)
    - Software (set single-shot ISR)
    - Hardware reload register
  - Up/Down variations

Limitation:

Hardware Timer Implementations of ClockTick

- Hardware Timer
  - Frequency Determination (that granularity question. . .)
    - Input clock source selection (frequency, power, cost, stability)
    - Configurable divider (e.g. TIMERn_CTRL.PRESC)
  - Reload methods
    - None (free-running)
    - Software (set single-shot ISR)
    - Hardware reload register
  - Up/Down variations

Limitation:

- Frequency scaling is often available only at values of 2 to an integer exponent (e.g. Pearl Gecko)

Example: Free-running counter, with SW Maintenance

Assume:

- We want a periodic timer of 10ms
  - The input clock is 1MHz.
  - Divider: Desired is 1MHz * 10ms = 10,000.
    - 8192 is closest, leading to an actual frequency of 1e6/8192 = approx 122Hz.
- Interrupt generated by timer on increment; ISR increments the OS "TICKS", OS looks for tasks to unblock

Can you imagine any problems with other timer services from this configuration? (hint: stopwatch...)

Example: Free-running counter, with SW Maintenance

Assume:

- We want a periodic timer of 10ms
  - The input clock is 1MHz.
  - Divider: Desired is 1MHz * 10ms = 10,000.
    - 8192 is closest, leading to an actual frequency of 1e6/8192 = approx 122Hz.
- Interrupt generated by timer on increment; ISR increments the OS "TICKS", OS looks for tasks to unblock

Can you imagine any problems with other timer services from this configuration? (hint: stopwatch. . .) Is it a time-to-aliasing, or a granularity problem?

Evaluation of this free-running solution

Granularity:

- Stopwatch samples just before (N) and after (N+1)
  - Perceived time = 2 ticks * 8.192ms/tick
- Stopwatch samples just after (N) and before (N+1)
  - Perceived time = 0 ticks.

Anti-aliasing:

- 32b register: (2^32)*8.192ms = 48 day anti-aliasing period.

Example: Software Reload of One-Shot

Counter initially loaded with the number of higher-frequency clocks needed for the desired clock tick.

- Interrupt is generated by underflow of count-down counter
- ISR reloads the value, restarts counter by reloading, increments timer ticks accumulation, looks for tasks to unblock, etc.
- Benefits?
- Disadvantages?

Example: Software Reload of One-Shot (cont.)

- Benefits:
  - stopwatch resolution greatly improved (can use the counter register as more bits)
  - if counter continues after underflow, SW can estimate/correct clock drift
- Disadvantages:
  - the extra bits will need to be normalized by the value pre-loaded to convert into tick units
  - The time that it takes to reload the counter to get it counting down again is "LOST". (Can remove the bias, but not necessarily all variation.)
  - Most Risky: what if a critical section held off the timer interrupt? "LOST TIME!" (at least if counter stops upon underflow)

Example: Hardware Reload Register

Counter loaded with the number of higher-frequency clocks needed for the desired clock tick.

- Counter restarted BY HARDWARE by reloading value from a register designated for this.
- Interrupt is generated for underflow of the count-down counter.
- ISR increments timer ticks accumulation, looks for tasks to unblock, etc.
- What's better now?
- Are you guaranteed exactly the desired clock tick now?

Example: Hardware Reload Register

Counter loaded with the number of higher-frequency clocks needed for the desired clock tick.

- Counter restarted BY HARDWARE by reloading value from a register designated for this.
- Interrupt is generated for underflow of the count-down counter.
- ISR increments timer ticks accumulation, looks for tasks to unblock, etc.
- What's better now?
- Are you guaranteed exactly the desired clock tick now?

No Jitter! But the desired and fundamental clocks may not have much in common.

Time Jitter

Aside from the jitter that could be introduced by reloading counter values from software, how can we reduce jitter in stopwatch measurements?

- Poll a counter value
  - What additional steps are needed?
- Optimize OS clock tick
  - What optimization might you force on the tick selection?

Time Jitter

Aside from the jitter that could be introduced by reloading counter values from software, how can we reduce jitter in stopwatch measurements?

- Poll a counter value
  - What additional steps are needed?
- Optimize OS clock tick
  - What optimization might you force on the tick selection?

- Polling in a spin-loop: disable interrupts (critical section) to guarantee no lost counter observations
- Clock tick: make it smaller. (Performance disadvantages coming in future lecture)