NC STATE UNIVERSITY

ECE 460/560: Class 02
A High-Level Look at Processes, HW and SW, Synchronization and Example Systems

A.G. Dean
agdean@ncsu.edu
https://sites.google.com/ncsu.edu/agdean/teaching

v2 9/2/2025

Class 02 Overview – Process Basics, HW and SW, Processes and Synchronization in Example Systems

- Review of **how** ES computers are different from GP, and **why**
  - Often demanding, complex **I/O timing requirements** drive **different design choices**
- Process relationships
  - Concurrent vs. sequential execution
  - HW vs. SW on single-core CPU vs. SW on multi-core
  - Free-running vs. synchronized
- Application example: Oscilloscope
  - Scope triggering: one kind of synchronization
  - How to implement with as little hardware as possible: busy-wait loop
- Hardware or software?
  - Software timing: hard to predict required time to execute (and its variability)
  - System response time for chain of processing steps
  - Application timing **requirements** vs. HW and SW **capabilities**
- Example applications
  - Scope
    - Key timing requirements:
      - Responsiveness to changing input signal. Detect trigger condition quickly.
      - Stable periodic timing for sampling input value.
    - Improving timing by moving key processing steps from software to hardware using peripherals, interrupts, direct memory access controller
  - ECE 306 line-following car
    - Inputs, processes, outputs
  - Motor position sensing and control
    - Input timing requirements for shaft position encoder. Missing deadline may give wrong direction or even miss pulses.
    - Output timing requirements for variable speed (pulse-width modulated) motor drive. Missing deadline affects motor speed proportionally
  - Waveform Generator
    - Stabilize output updates to regular periodic times with low jitter for accurate signal generation.

Computers for Embedded Systems vs. General-Purpose Systems

*"How slow can your CPU go and still be on time?" Embedded Systems have **concurrent compute processes** with **diverse I/O operations.** Often the I/O for a process has **challenging timing requirements,** so we decouple it from compute software (bad timing characteristics) by splitting it into two or more processes to make **input** or **output** operations **asynchronous** to the **compute** operations. These processes need to **synchronize** and **communicate** (data buffering). We may even move some processing to hardware. We use **interrupts, HW peripherals** and **DMA** to make a **low-cost** and **feasible** solution with a low-frequency CPU.*

Embedded (Computer) System enhances larger system: e.g. improves performance, adds safety protections, simplifies maintenance & diagnostics. Must monitor inputs and control outputs.

Range of **processing activities** needed to handle inputs, determine control actions, update outputs.

Inherently concurrent system. Often is most practical to **implement with multiple concurrent processes** (some SW, some HW).

System with concurrent processes requires **sync** & **comm**

Wide range of input and output signals. Digital, analog, differential, bit-dominance (wired-or), etc.

Some I/O operations step through a **sequence of I/O sub-operations** triggered by **events** or **time delays**, creating **new linked timing requirements**. UART RX operation, PWM, synchronous control of motor/SMPS, network with bit dominance, etc.

Wide range of **timing requirements** (absolute time, update rate & phase, synchronization (among signals, with clock, with system substate), response time, timing stability vs. jitter …) **for input signals, output signals, and between them (I->I, I->O, O->O)**.

Synchronous software I/O is bad fit for **time-critical I/O** requirements. **SW timing obscurity/ambiguity/non-determinism** clash with **I/O needs** (req'ts for **timing precision & stability)** and **SW<->I/O rate mismatches** (especially for burst activities)

Sources of software timing obscurity: inherent behavior of algorithm, arbitrary input event sequences, program compilation, performance variation/non-determinism (CPU, memory system), task scheduling

- Inherent behavior of algorithms (control flow variations)
- Disconnect between source code and object code timing: compilation, ISA features, optimizations
- CPU performance variations: data-dependent instruction timing, superscalar/dynamic execution, pipelines, predictors, prefetching
- Memory system (caches, VM, interference in multicore, …)
- Arbitrary input event sequences possible, complicating system timing behavior
- **Interrupts and Scheduling to share CPU core(s).**

*Mainstream computing just uses a **subset** of the **Async I/O design space.** Targets gen-purpose computers with a **few I/O devices** (user interface, storage, network) and **their use cases**. **Interrupts/exceptions** for timer tick, OS interface, faults, I/O events (Rx or Tx complete, error). **DMA** discussed if you dig deep enough into system design.*

Use **Async I/O** to bridge/tolerate timing mismatches (between I/O and SW) at low cost

Implementing Async I/O requires deciding **where to split** process, how those parts will **sync** and **communicate**.

- **Sync** for initial triggering (event generators/detectors)
- Supporting splits: **Communication** (esp. data buffering w/timing requirements), more **sync** to support comm (notifications, handshaking, overruns …)

Can implement process **functionality, sync and comm in SW, HW or both**. Should select based on strengths and weaknesses of SW, HW for given need.

Efficiently crossing between HW and SW to implement procs, sync and comm. Interrupts, DMA vs. prog I/O.

Sharing CPU: Interrupts, Scheduling, Real-Time System Concepts

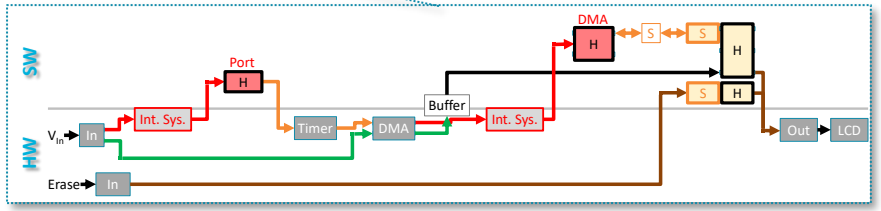
When you have only a hammer, everything looks like a nail. CS education doesn't do digital design (other than CPU, maybe memory system, AI accelerators, …).

Throw in another core

Programmable Coprocessors: TI PRU (prog. real-time unit), …

Use HW for some or all of func, sync, comm: **less SW** needed (if any), **easier SW deadlines** (fewer, looser).

Must **understand some digital design** to effectively recognize and assess HW implementation options

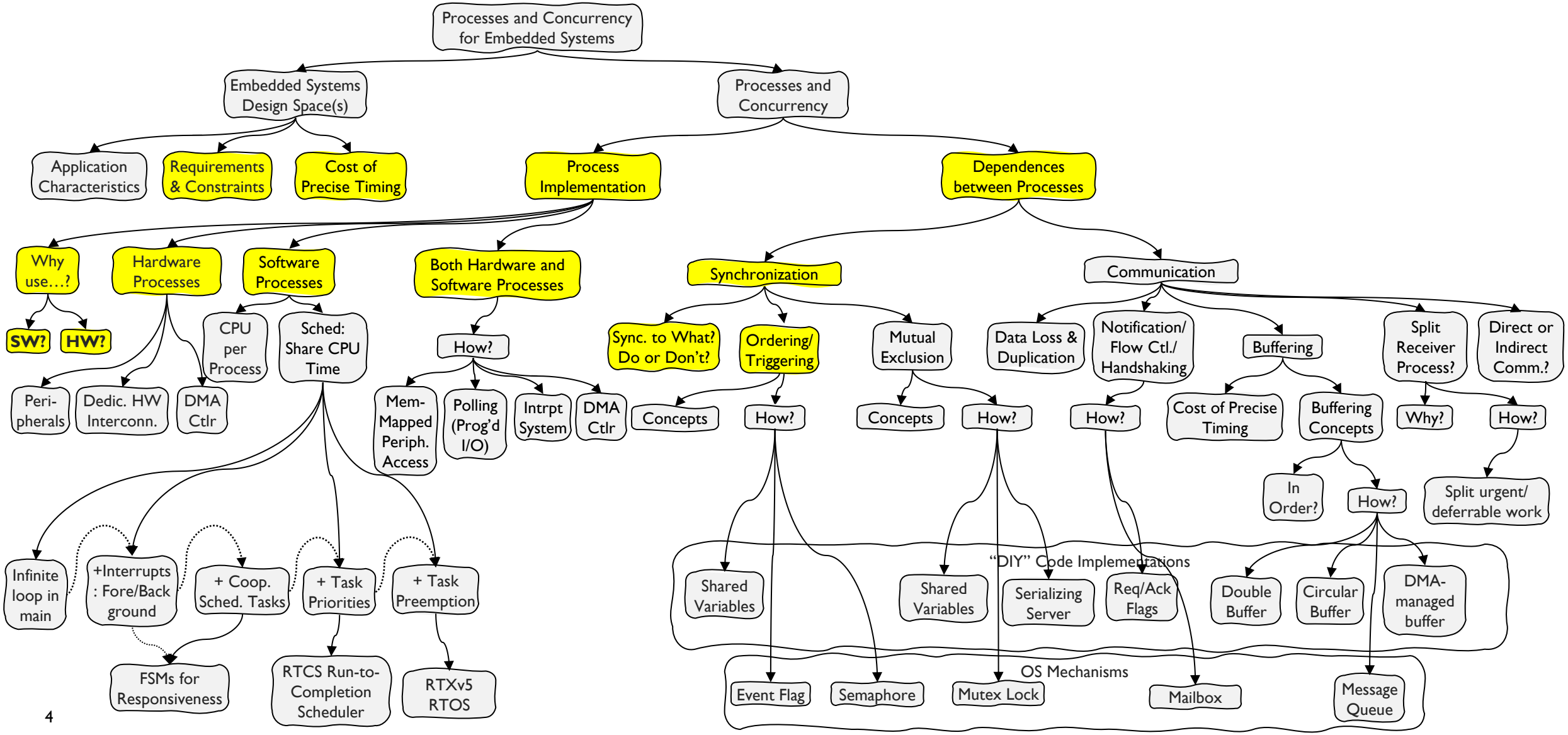


General Design Pattern: functionality, sync, comm (esp. buffering)

Implementations & Mechanisms (outside of CPU ISA)

- **General HW Peripherals**
- **Event/Sync Interconnect for Peripherals**
- **DMA**
- **HW Peripherals for Sync/Comm Support**
- **Programmable logic with custom FSM. CLB, FPGA. Pico Prog. I/O blocks (FSMs)**

Extending the Topic Map



Take Multiple Passes, Getting More Details As Needed

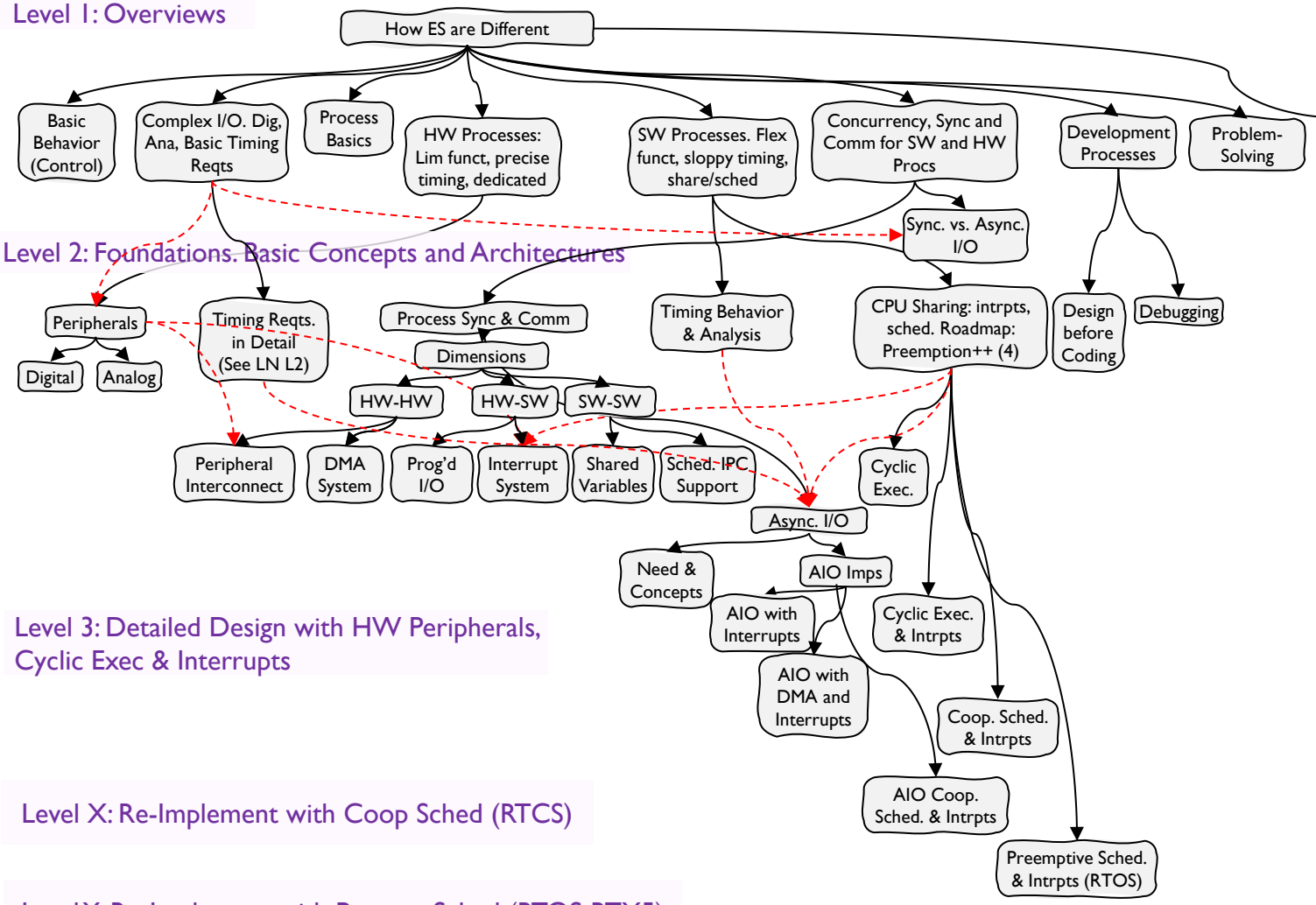
Level 1: Overviews

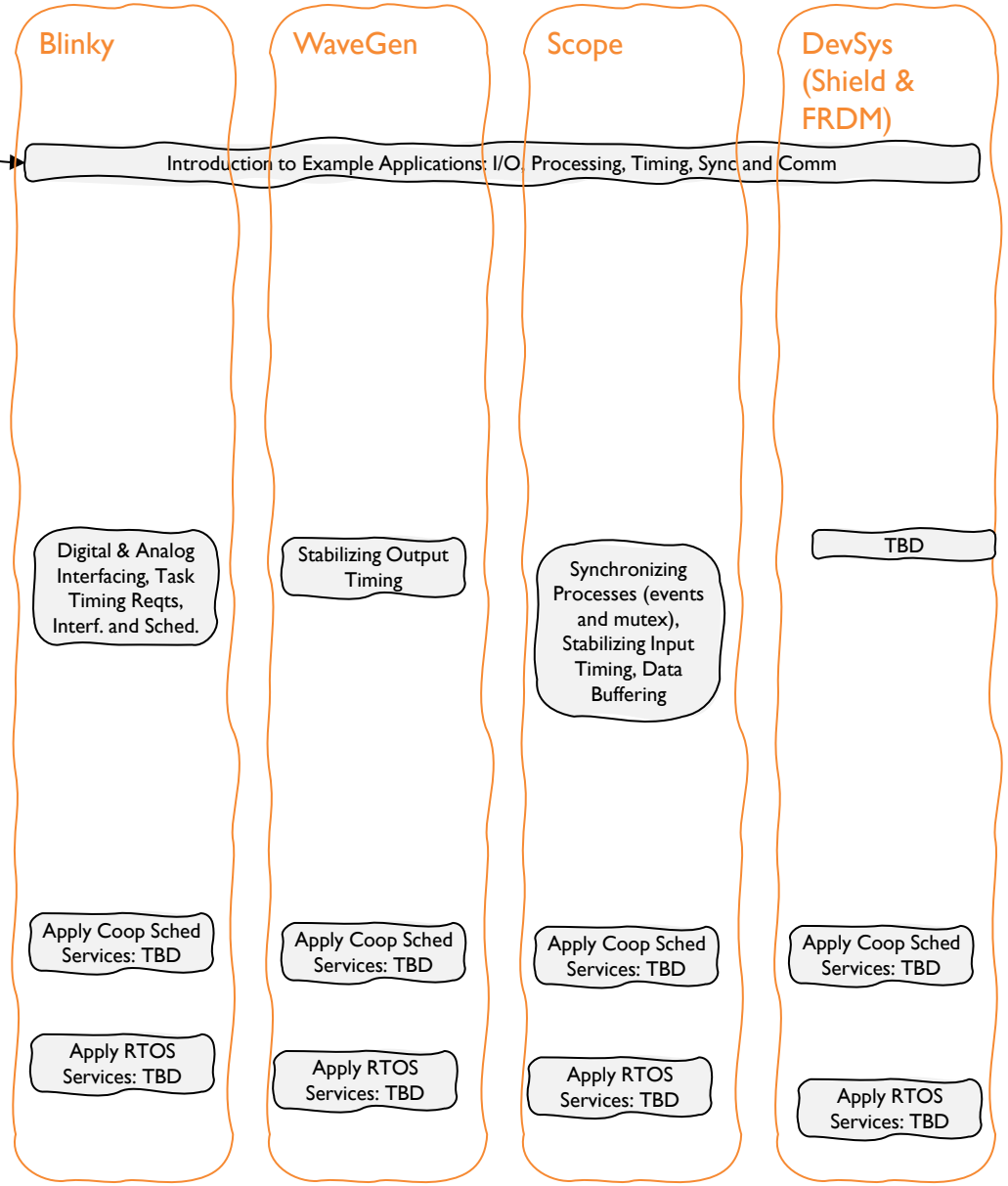
Level 2: Foundations. Basic Concepts and Architectures

Level 3: Detailed Design with HW Peripherals, Cyclic Exec & Interrupts

Level X: Re-Implement with Coop Sched (RTCS)

Level Y: Re-Implement with Preemp. Sched (RTOS RTX5)



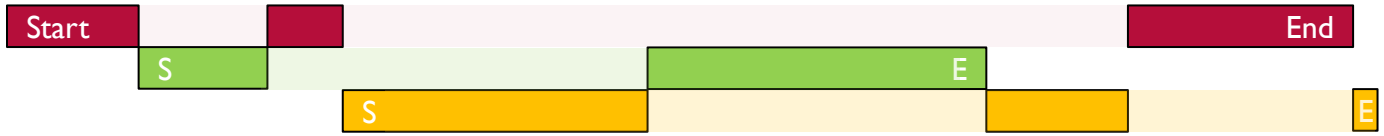


Process Relationships: Concurrency and Synchronization

Process Relationships

- Sequential: Finish current process before starting another
  - Finish red before starting any other process

- Concurrent: Process execution may overlap in time
  - Can start green, yellow before finishing red



- Execution of concurrent processes
  - Hardware: Dedicated circuit per process, so able to run at the same time

  - Software: depends on # of CPU cores
    - Each core works on one process at a specific point in time

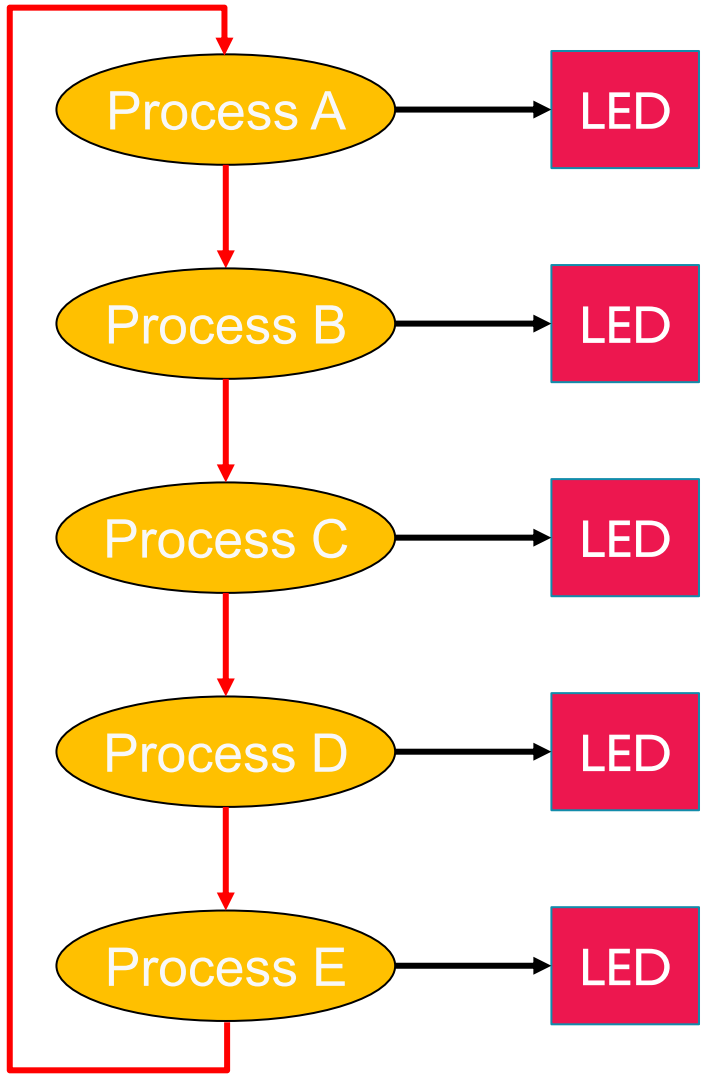
1 Core

2 Cores

3 Cores

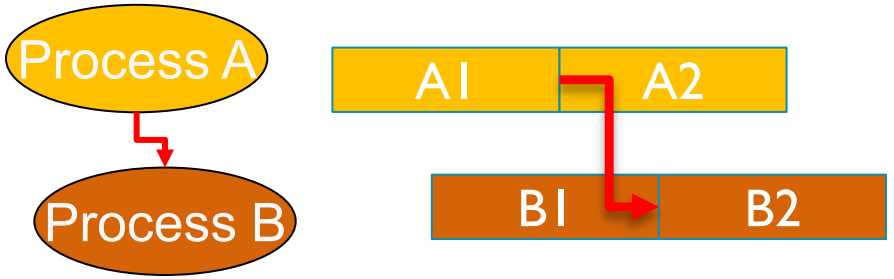
Synchronized or Free-Running Process Execution?

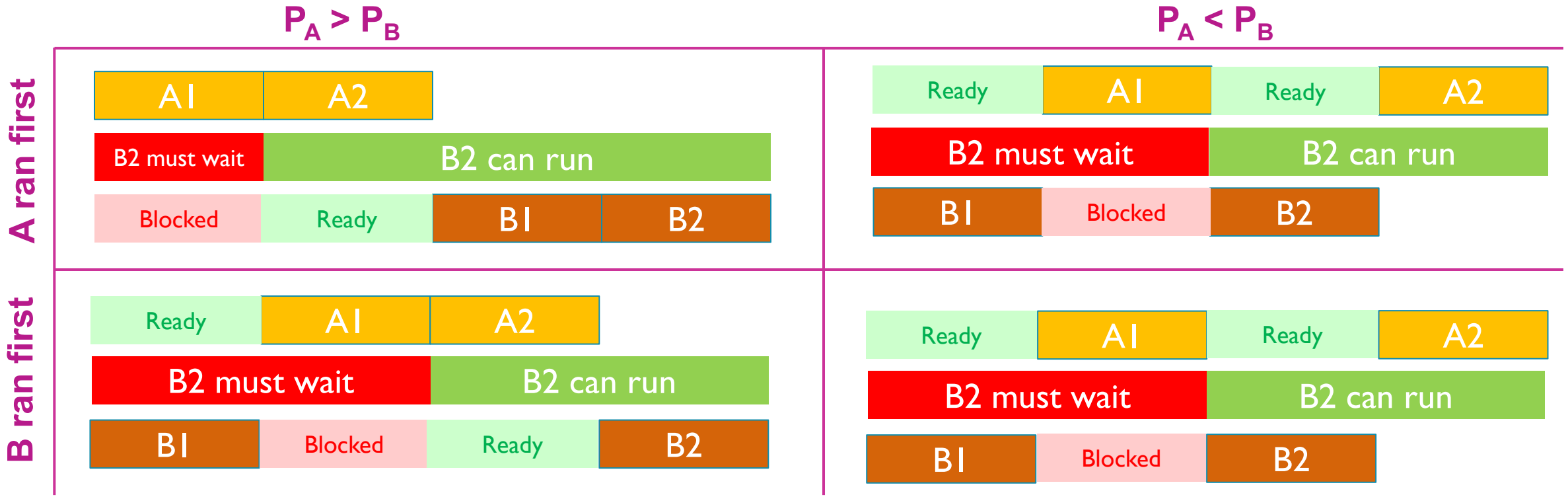
- Example: Five processes (A-E), each flashing an LED
- How to make LEDs flash in a scanning sequence?
  - Simple independent starter process doesn't do this
  - LEDs flash independently of each other. Changing one process doesn't affect the others
  - No synchronization between processes, are free-running
    - Hardware process runs non-stop
    - Software process runs whenever it can (CPU available)
- Processes need to synchronize with each other
  - After turning off its LED, process sends a synchronization signal to the next process.
  - A process doesn't turn on its LED until after it gets a signal from the previous process
    - Special case for start-up: Process A doesn't wait for signal on its first execution



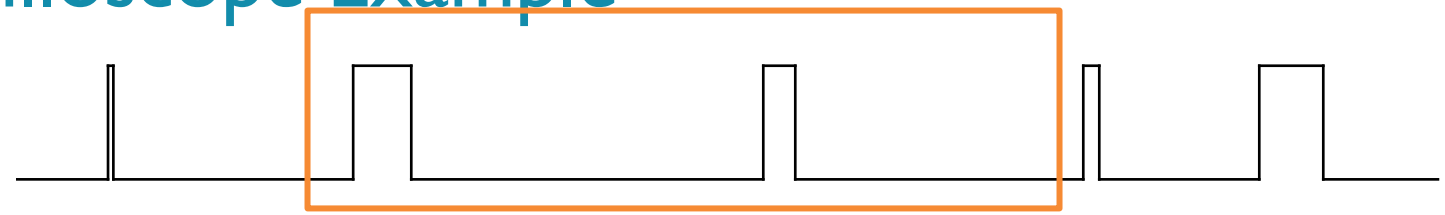
Synchronized Process Execution

- Don't let Process B start to execute section B2 until Process A has completed section A1
  - Includes case where each thread has only one section
- Multiple cases possible based on initial process execution order and priority (if sharing a CPU)

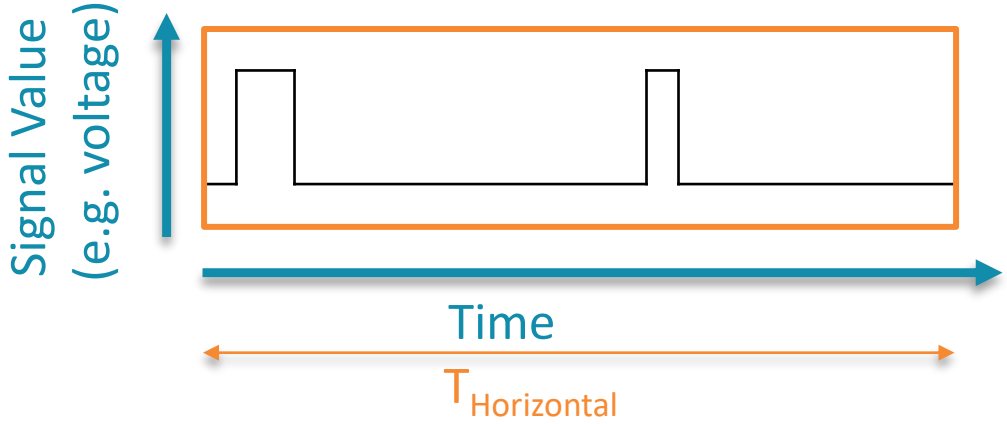




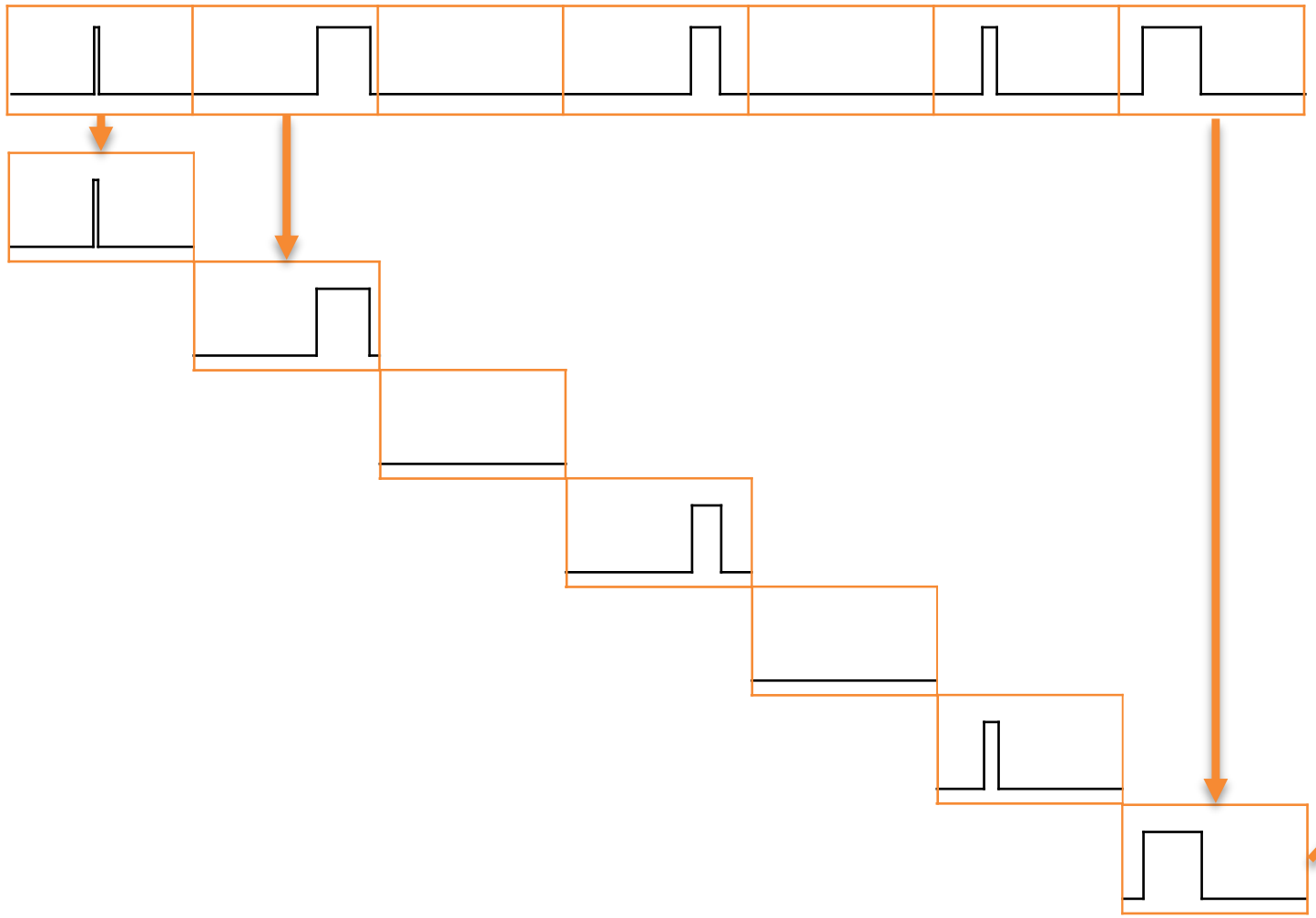
Synchronization: Simple Oscilloscope Example



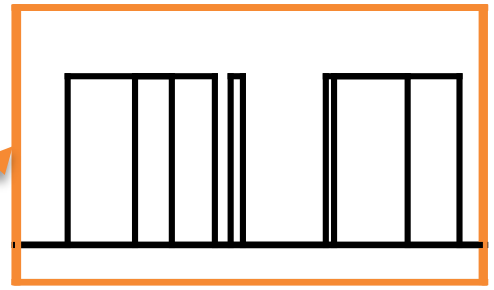
- Input signal
  - Start with simple one-bit digital signal (do analog later)
  - Pulses have irregular start times, changing pulse widths
- Displaying the signal
  - Oscilloscope ("scope") plots signal value (e.g. voltage) vertically vs. time horizontally
  - Horizontal time base determines amount of time (T_{Horiz}) represented on scope display
  - Display stability depends **timing relationship** between when scope starts displaying the signal, and when the signal changes
  - "Infinite persistence" accumulates all acquired traces on display until erase button is pressed



Simple Method: Display Signal Continuously

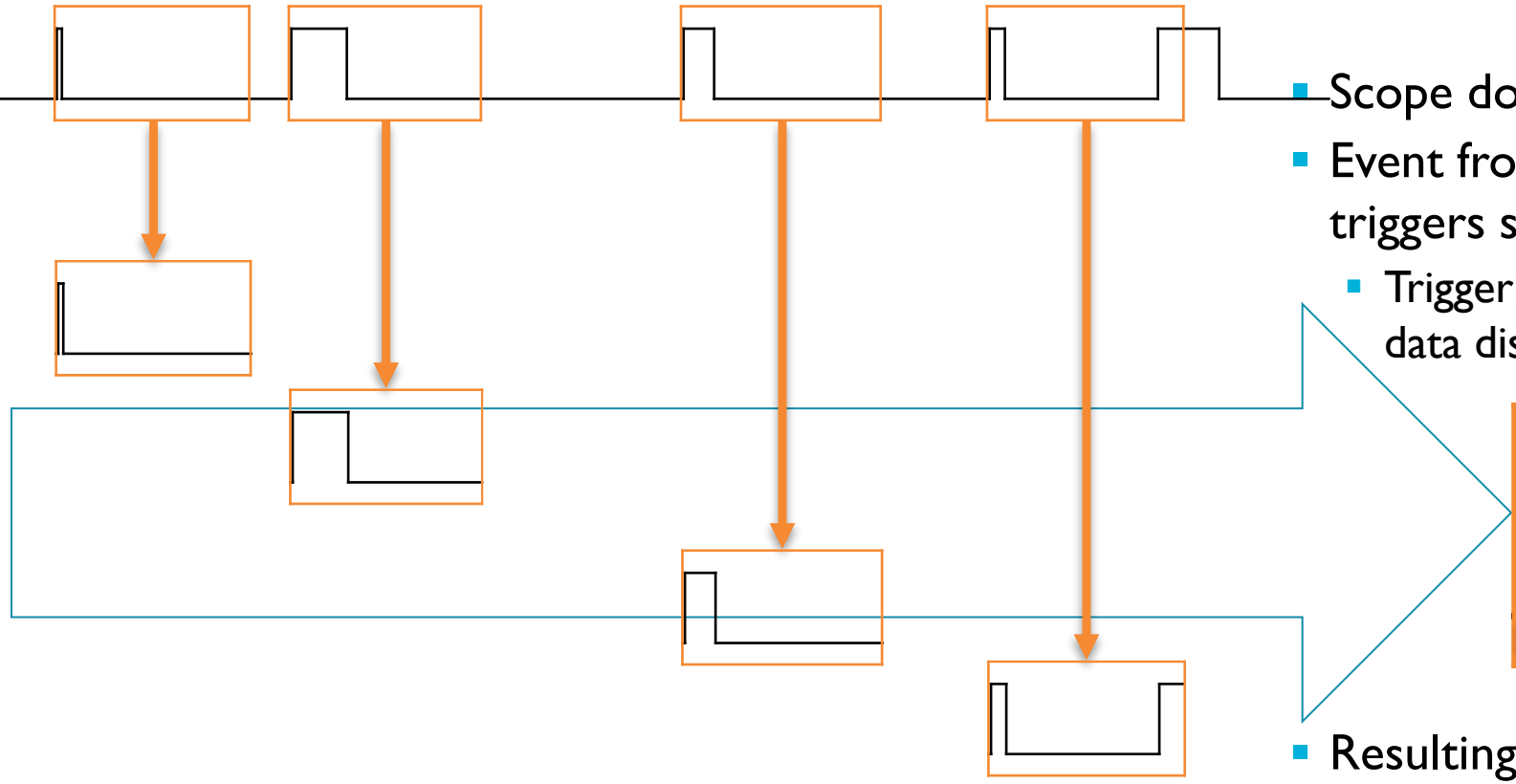


- Sequence
  - Display signal from 0 to T_{Horiz}
  - Display signal from T_{Horiz} to $2*T_{Horiz}$
  - Display signal from $2*T_{Horiz}$ to $3*T_{Horiz}$
  - Display signal from $3*T_{Horiz}$ to $4*T_{Horiz}$
  - Display signal from $4*T_{Horiz}$ to $5*T_{Horiz}$
  - Display signal from $5*T_{Horiz}$ to $6*T_{Horiz}$
  - Display signal from $6*T_{Horiz}$ to $7*T_{Horiz}$
  - etc.

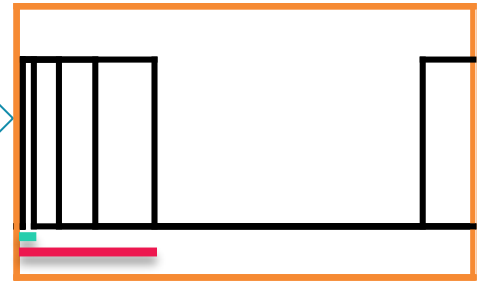


- What is range of pulse widths? Can't see.
- Resulting display is unstable, jumps around over time.

Stabilize Display with Triggering

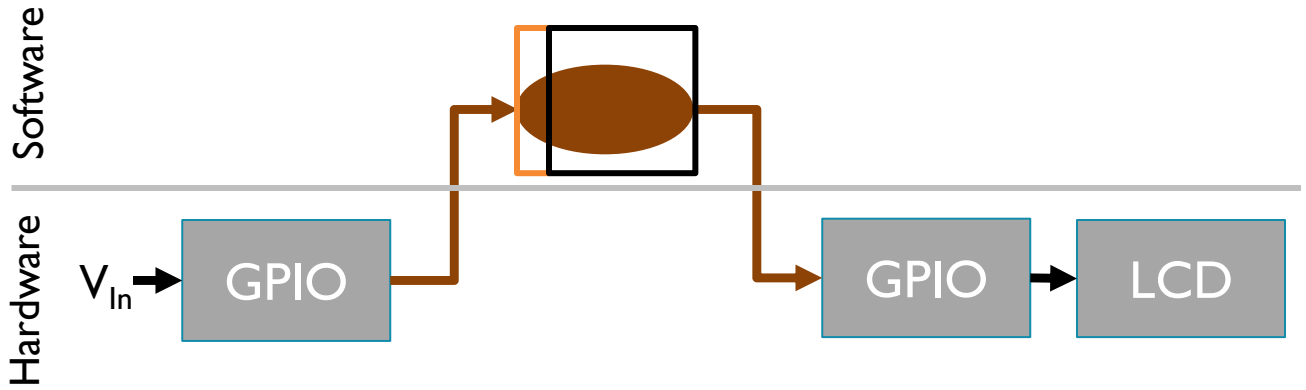


- Scope does nothing until triggered
- Event from input signal (e.g. 0 to 1 edge) triggers scope to start displaying signal
  - Triggering **synchronizes** the scope's start of data display to input signal event



- Resulting display is much more stable
  - Range of pulse widths is easy to see.
  - Rising edge of signal is stable
    - Except for last acquisition, where time between rising edges < T_{horiz}
  - Falling edge unstable since pulse width varies

Simple Busy-Wait Loop

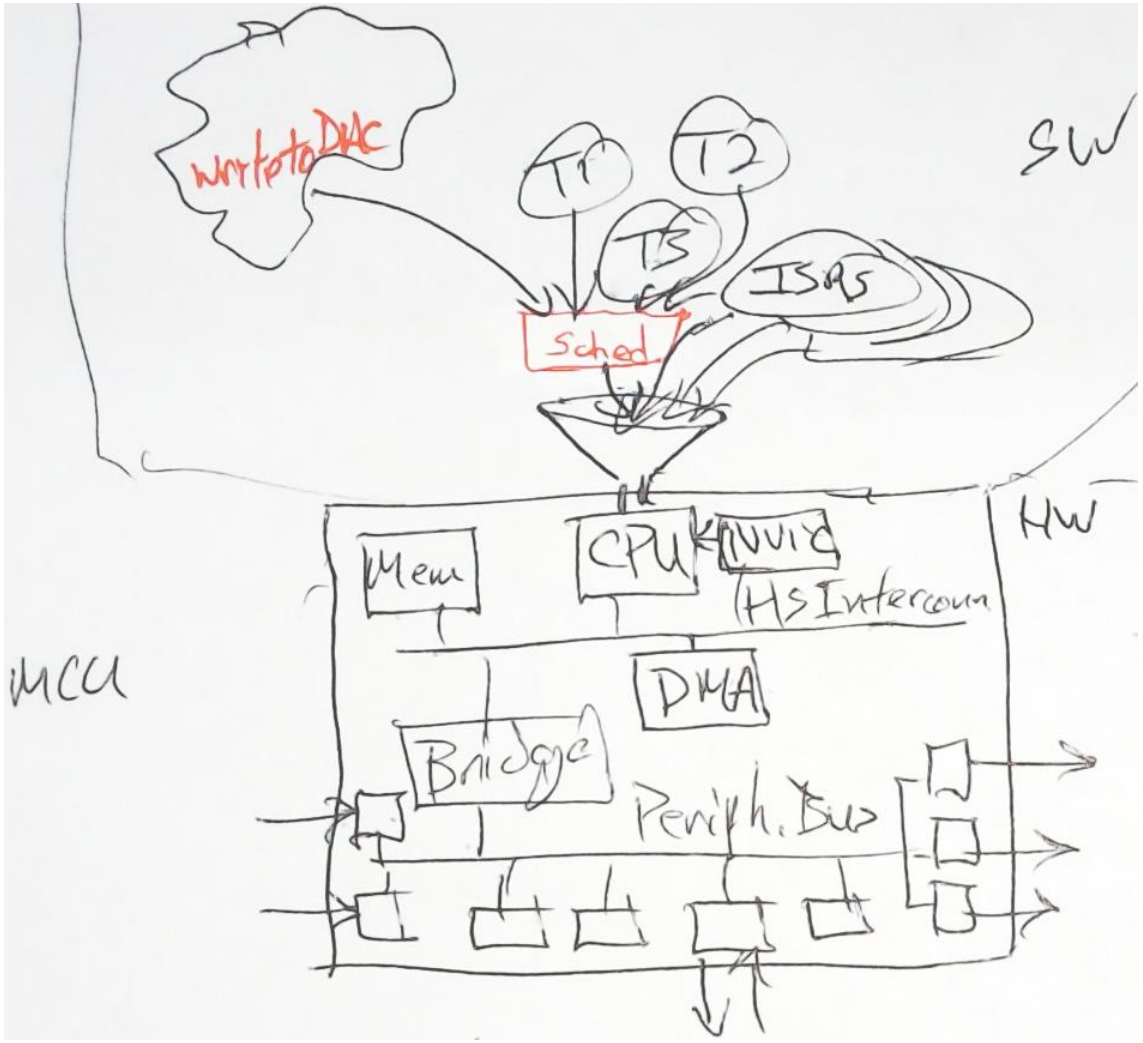



```
Process A
…
// Detector/Synchronizer
while (ADC->Result < V_Threshold)
  ;
// No Scheduler
// No Dispatcher
// Handler process
x = 0;
for (n=0; n<NS; n++) {
  r = ADC->Result;
  y = scale(r);
  LCD_Plot(x++,y);
}
```

- Synchronization built into SW process A
- Simple, but doesn't scale up well with multiple software processes

System Timing Performance: Software and Hardware

Use Software or Hardware? Flexibility vs. Timing Stability



- Software
  - Program gives very flexible functionality
  - Interrupt system (e.g. NVIC) and scheduler (if any) determines **what** software runs on CPU and **when**
  - Software very vulnerable to timing interference. **Need synchronization.** Use interrupts, scheduler to improve timing stability
- Hardware
  - Very stable timing (when independent of software)
  - Functionality limited to what is built into hardware (and your creativity)

"Sloppy" Software Timing Behavior

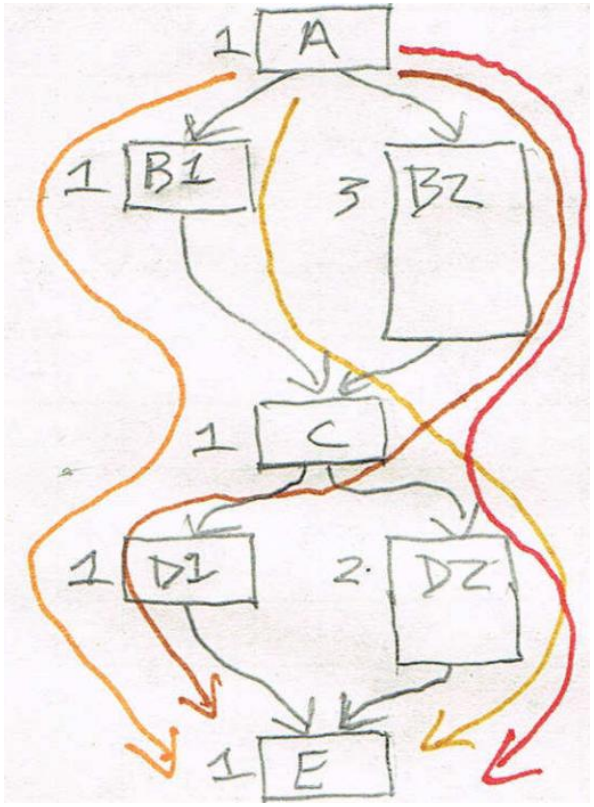
- Time to execute code is…
  - Hard to predict accurately: Timing behavior depends on machine language instructions generated from source code by compiler, CPU used, data-dependent instruction timing, system speed….

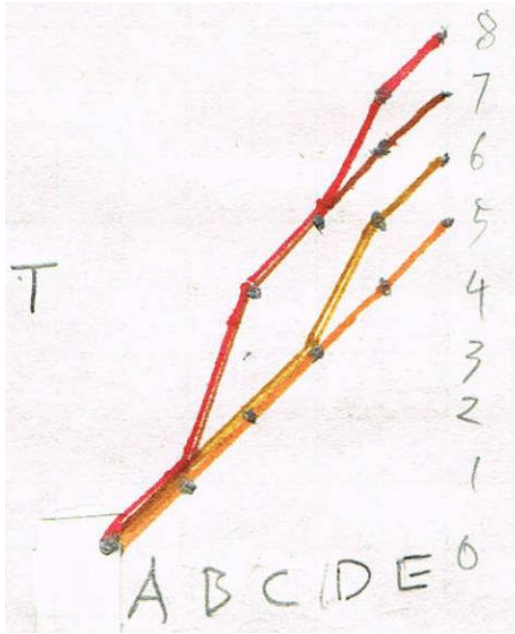
```
if (x>0)
  j += r;
else
  x++;
x = x/8;
if (j>3)
  x -= 17;
else
  r *= 7;
```

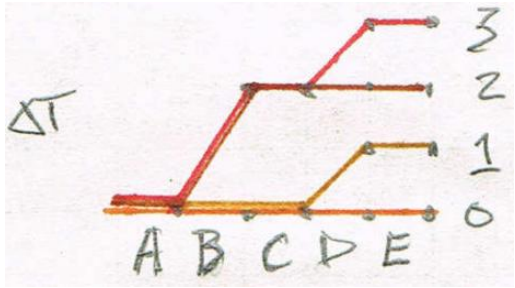
Compile, assemble and link

????
(machine code instructions)

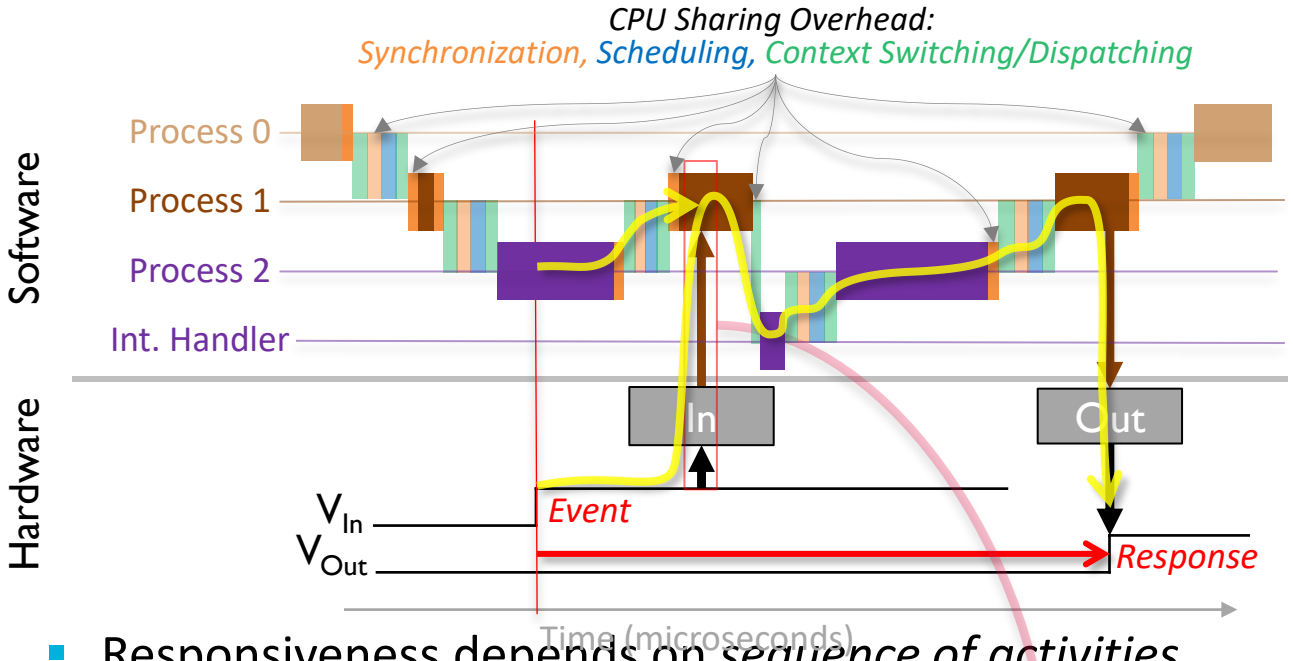
- Unstable ("fragile"): Depends on paths taken through conditionals, loop repeat counts, etc. Paths may depend on input data, execution history, etc.



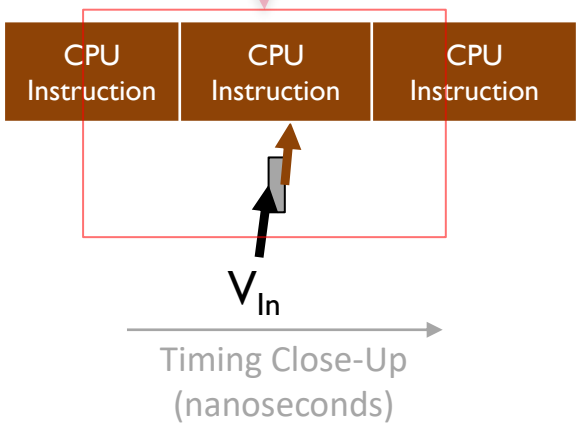




System Responsiveness Depends on Processes

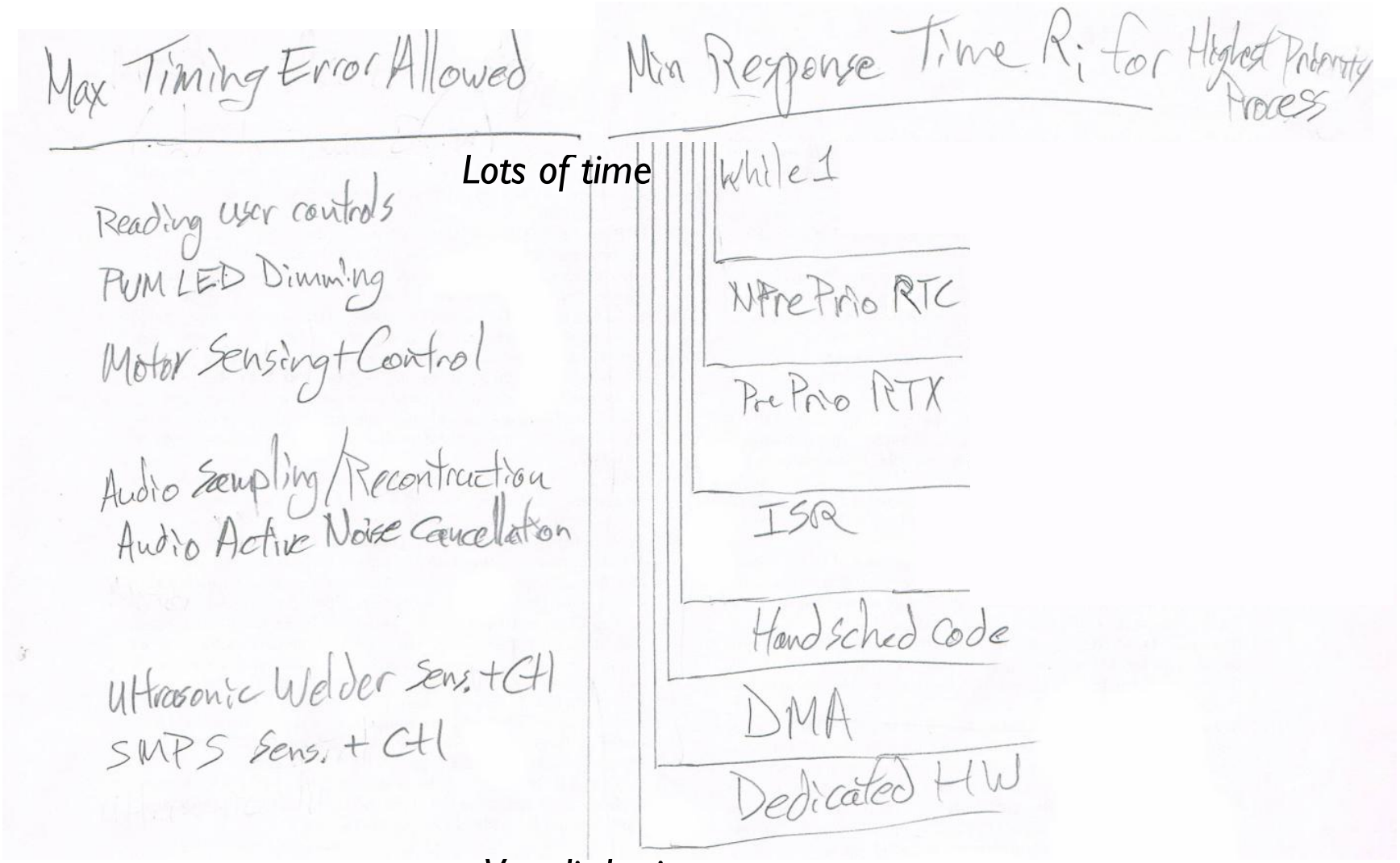


- Responsiveness depends on *sequence of activities* between input event and system's response
- Diagram
  - Process 1 samples V_{in}, looks for event (0 to 1 transition)
- Hardware process timing: fast, very stable, predictable
  - Typically faster than time for CPU to execute an instruction



- Uses hardware circuits which are dedicated (not shared)
  - Exceptions later: shared buses, etc.
- Software process timing: much slower, unstable, hard to predict precisely
  - Time to execute a software process is hard to predict, varies based on input data, history …
  - Sharing CPU among multiple software processes delays a process
    - Inherent delays and processing overhead (may be in program, interrupt system, OS/executive) for:
      - Synchronization: deciding if process may run (is ready) or must wait for event/condition
      - Scheduling: deciding which ready software process to run next
      - Context Switching and/or Dispatching: saving and restoring process contexts, starting next process running
    - Timing interference (preemption, blocking) from other software processes (threads, interrupt handlers)

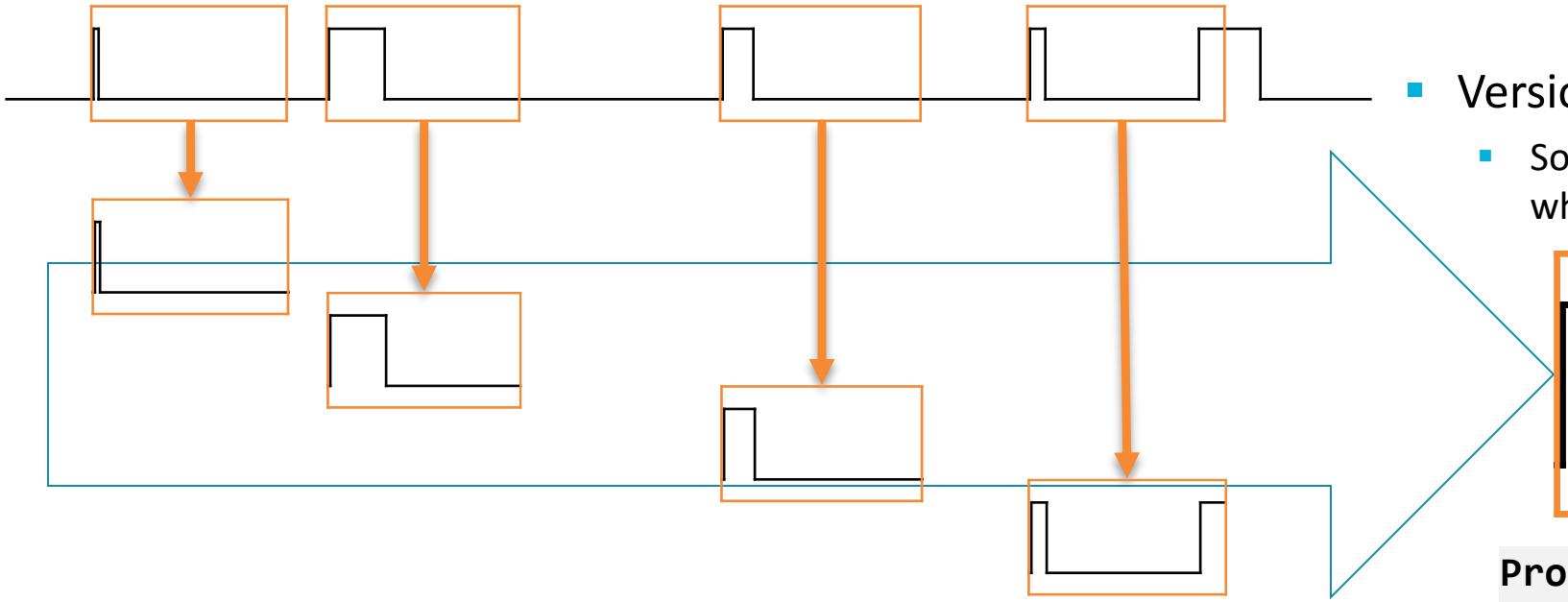
Timing Requirements vs. Response Time Capabilities for Different Design Approaches



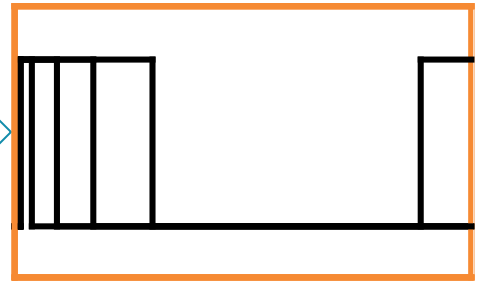
Design Examples

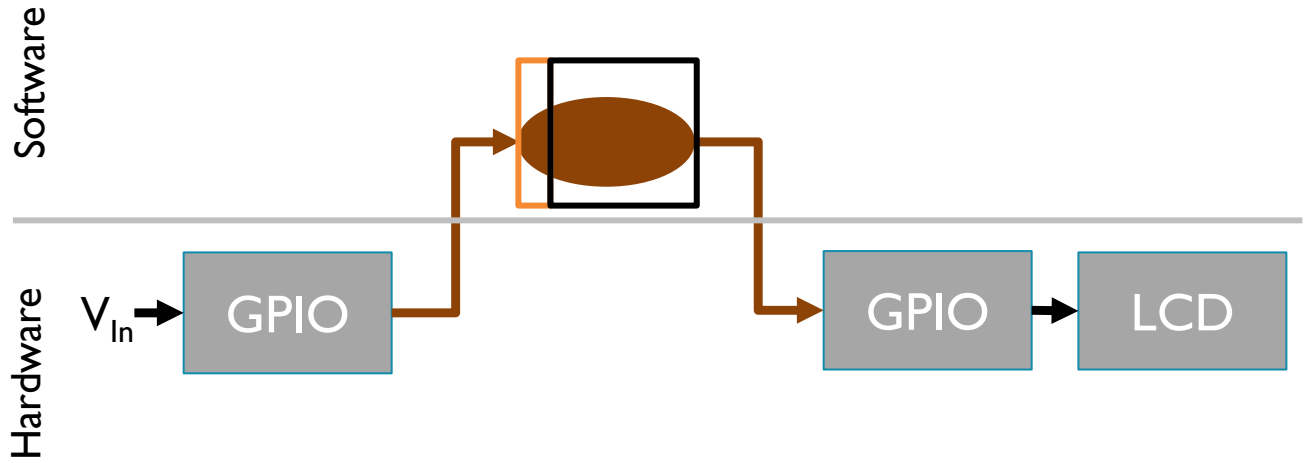
- Oscilloscope
  - Synchronize to input signal rising across trigger voltage level, then capture data samples at precise, frequent times
- ECE 306 line-following car
  - Multiple processes
- Motor position sensing and control
  - Monitor motor position using quadrature shaft encoder
- Waveform generator
  - Generate analog waveform with consistent, precise timing for output updates

Scope: Stabilize Display with Triggering



- Version 1: Simple Busy-Wait Loop
  - Software detects trigger event using small loop which blocks progress through process/thread

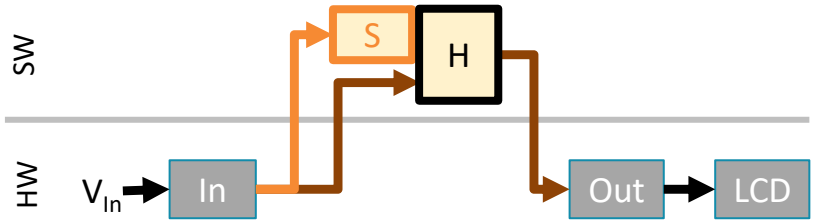





```
Process A
// Detector/Synchronizer
while (ADC->Result < V_Threshold)
  ;
// No Scheduler
// No Dispatcher
// Handler process
Loop for all columns in screen
  Get input data sample,
  Scale it,
  Plot it on LCD
```

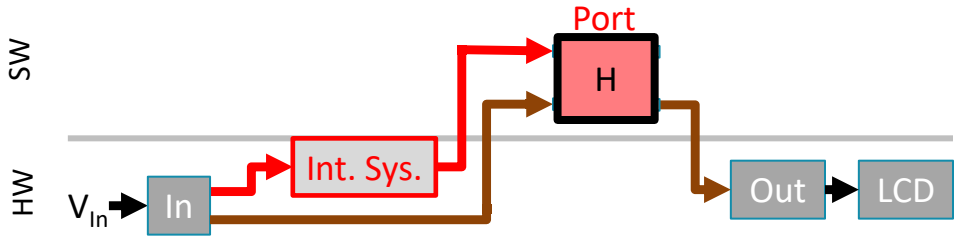
Improve Timing by Moving Activities from Software to Hardware

Trigger Detection by Software



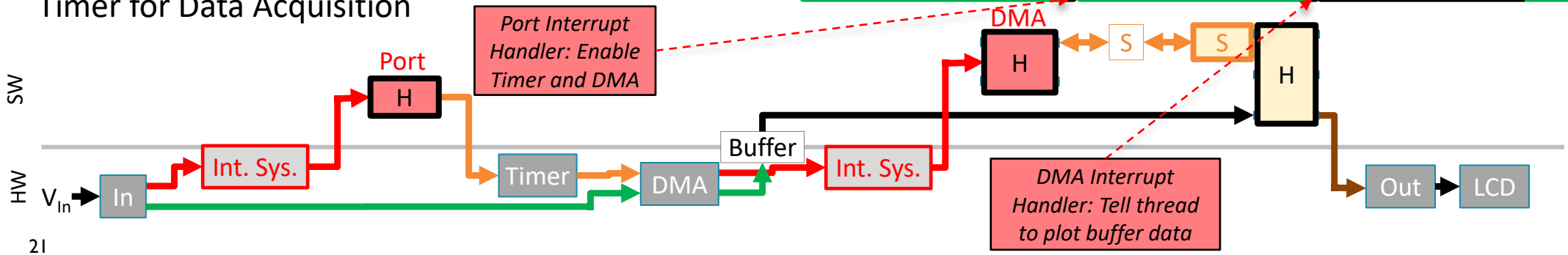
Sync: Loop until rising edge event detected | Thread: Sample input, plot on LCD | Sync: Loop until rising edge event detected

Trigger Detection by Hardware



CPU available for other SW processes | Port Interrupt Handler: Sample input, plot on LCD | CPU available for other SW processes

Hardware Trigger Detection and DMA (Direct Memory Access) + Timer for Data Acquisition

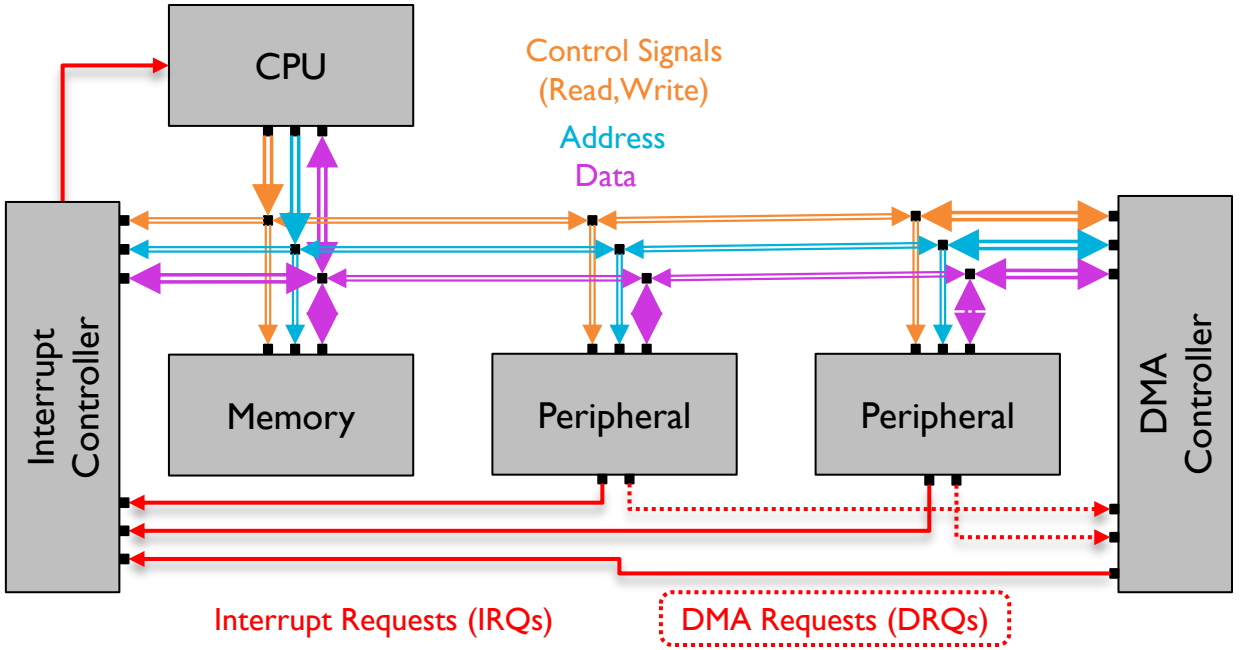


Sync: Data Ready to Plot?

CPU available for other SW processes | CPU available for other SW processes | Thread: Plot buffered samples on LCD | CPU available for other SW processes

Direct Memory Access Controller

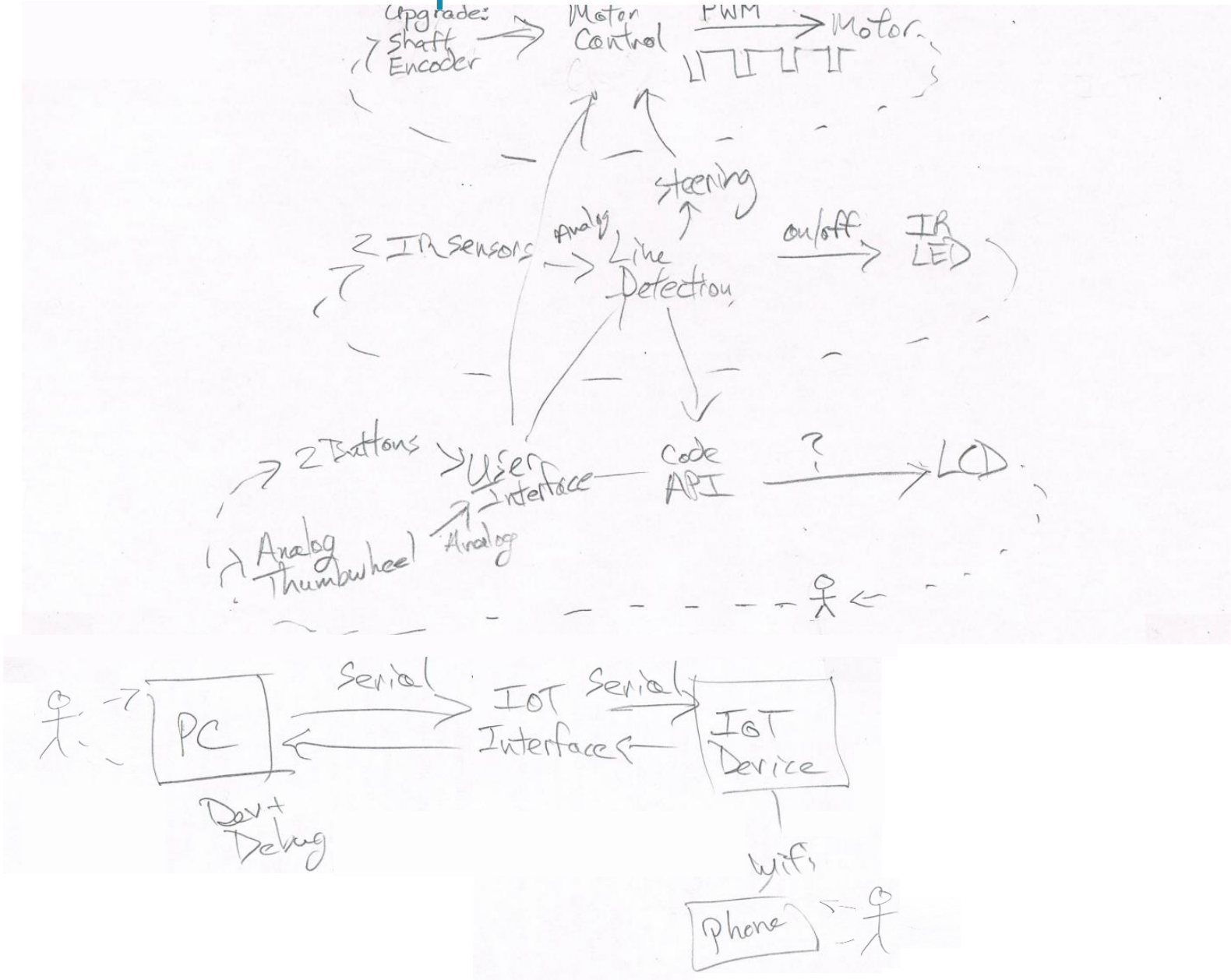
Allows Hardware->Hardware communication without using CPU



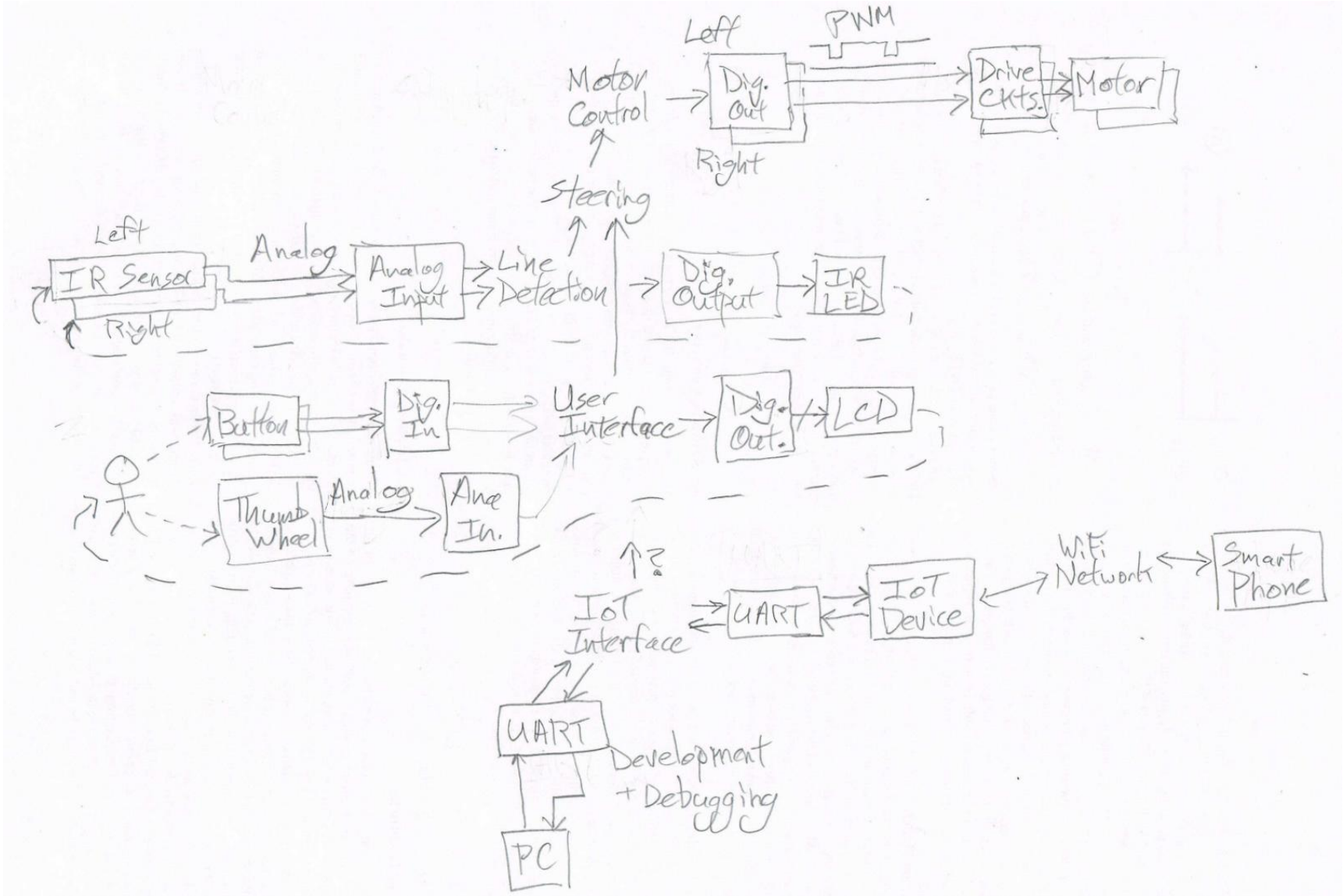
- How to access memory and peripherals?
  - CPU uses **memory bus** (address, data, control) to access memory and peripheral devices
  - Memory bus can also be controlled by DMA Controller (DMAC) peripheral

- DMA features
  - DMAC can transfer (copy) N data items within memory space from SrcAdx to DstAdx
    - SrcAdx, DstAdx: fixed or increment per item copied
    - Allows direct copy, but also accessing sequential items in memory array ("Save the next N ADC data values in memory starting at this address")
  - Transfer can be triggered by:
    - Hardware (DMA Request from peripheral device)
    - Software (CPU writing to DMA request control register)
  - Configurable bus sharing with CPU: can be greedy (burst of all transfers), round-robin, etc.
  - DMAC can generate interrupt when done
  - DMAC has multiple channels, each with individual trigger source, Adx pointers and behaviors, item count, interrupt behavior

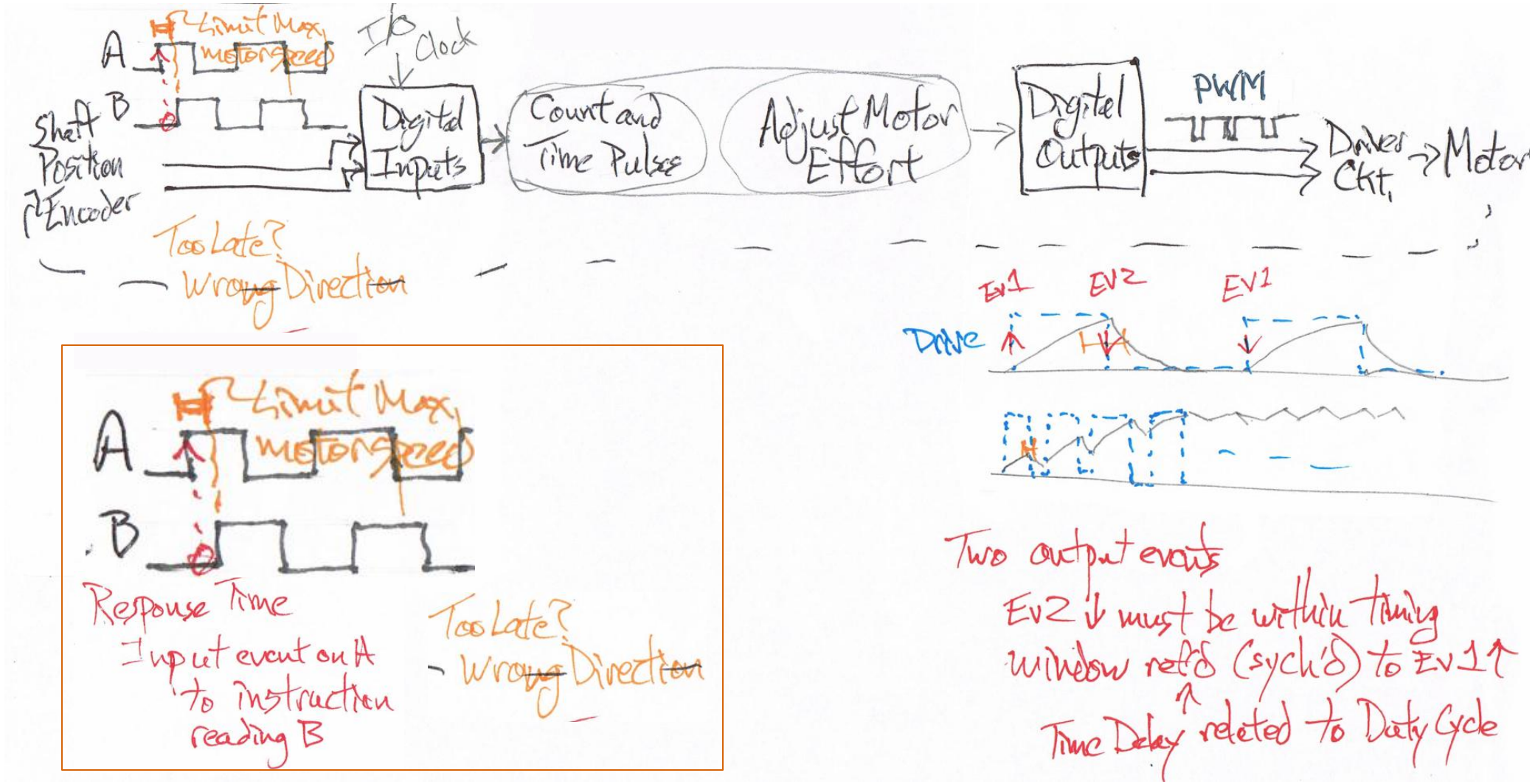
ECE 306 Car: Inputs, Processes and Outputs



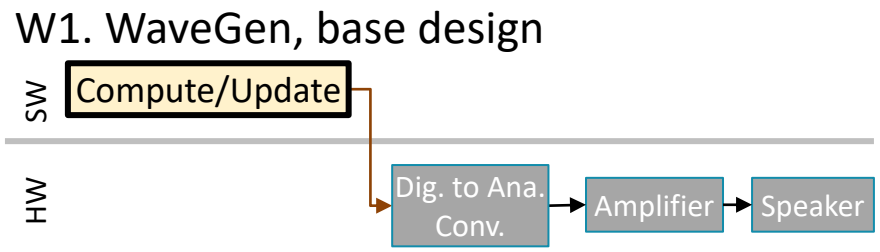
ECE 306 Car: Add Hardware Peripherals for Interfacing

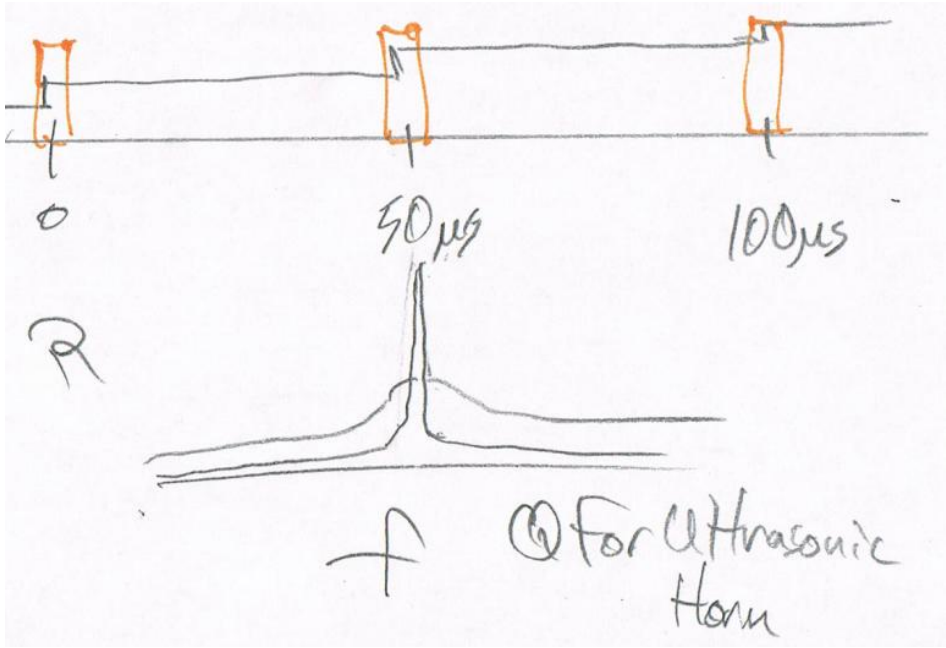


Motor Position Sensing and Control



Waveform Generator Subsystem: One Process





- Part of a larger system with other processes (e.g. user interface)
- Want to update DAC output every 50 us for a 20 kHz update rate
  - DAC signal amplified to drive speaker

Process	Input Device	Input Peripheral	Processing	Output Peripherals	Output Devices	Timing Requirements
W: Waveform Generator	n/a	n/a	Calculate new output value, wait fixed time, write output value to DAC	Digital-to-analog converter	Amplifier & Speaker	Every 50 us, +/- 5 us (?)