NC STATE UNIVERSITY

ECE 460/560
Embedded Systems Architectures: Introduction

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

8/19/2025

NC STATE UNIVERSITY

Embedded Systems Topics (and Dependencies)

NC STATE UNIVERSITY

Embedded Systems High-Level View (1)

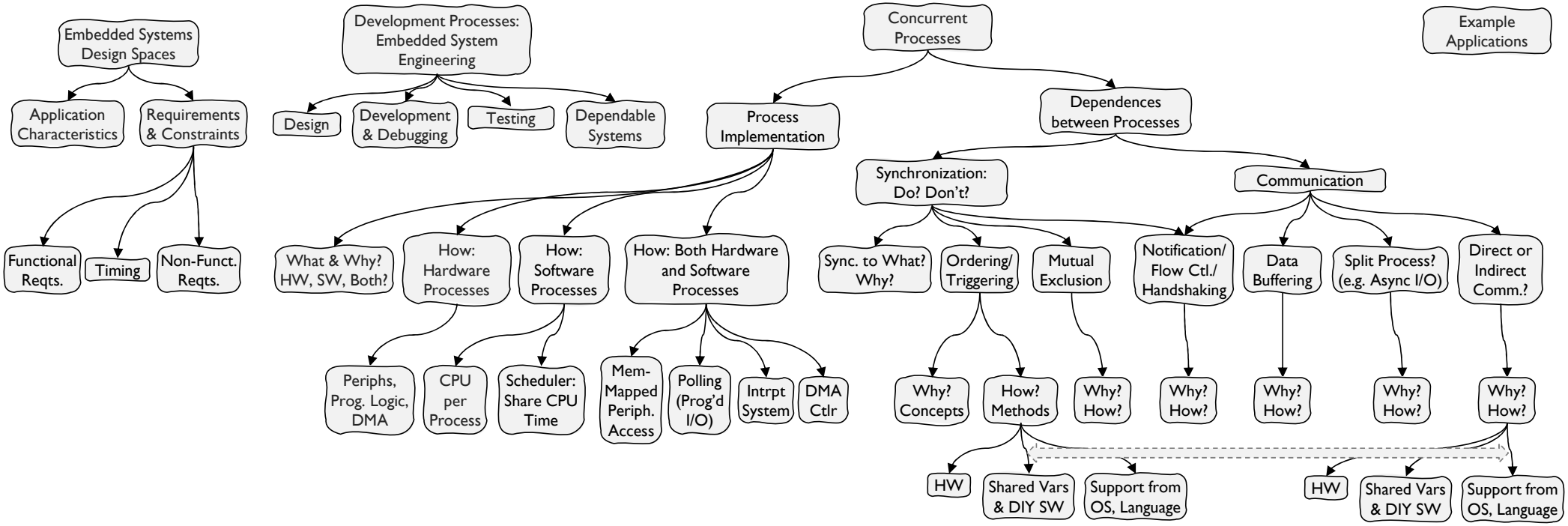
- Embedded Computer Systems frequently target **control applications**
  - Get input (read signal, detect event), Compute new output value, Update output
  - Microcontroller = Microprocessor + memory + hardware peripherals to support control
- Embedded Systems have **processes**, different implementation options
  - **Software** can do almost anything (eventually). Timing is slow, very sloppy.
  - **Hardware** is very fast and energy-efficient, uses dedicated circuits. Stable timing. Limited functionality available.
- Typically have **multiple concurrent processes** due to application requirements
- These processes often have **diverse I/O operations**
  - Digital signals, analog signals (must be converted to digital)
  - Bursts of events (e.g. PWM, serialized data, etc.),
  - Sample input periodically vs. receive event notification,
  - Range of I/O operation frequencies

NC STATE UNIVERSITY

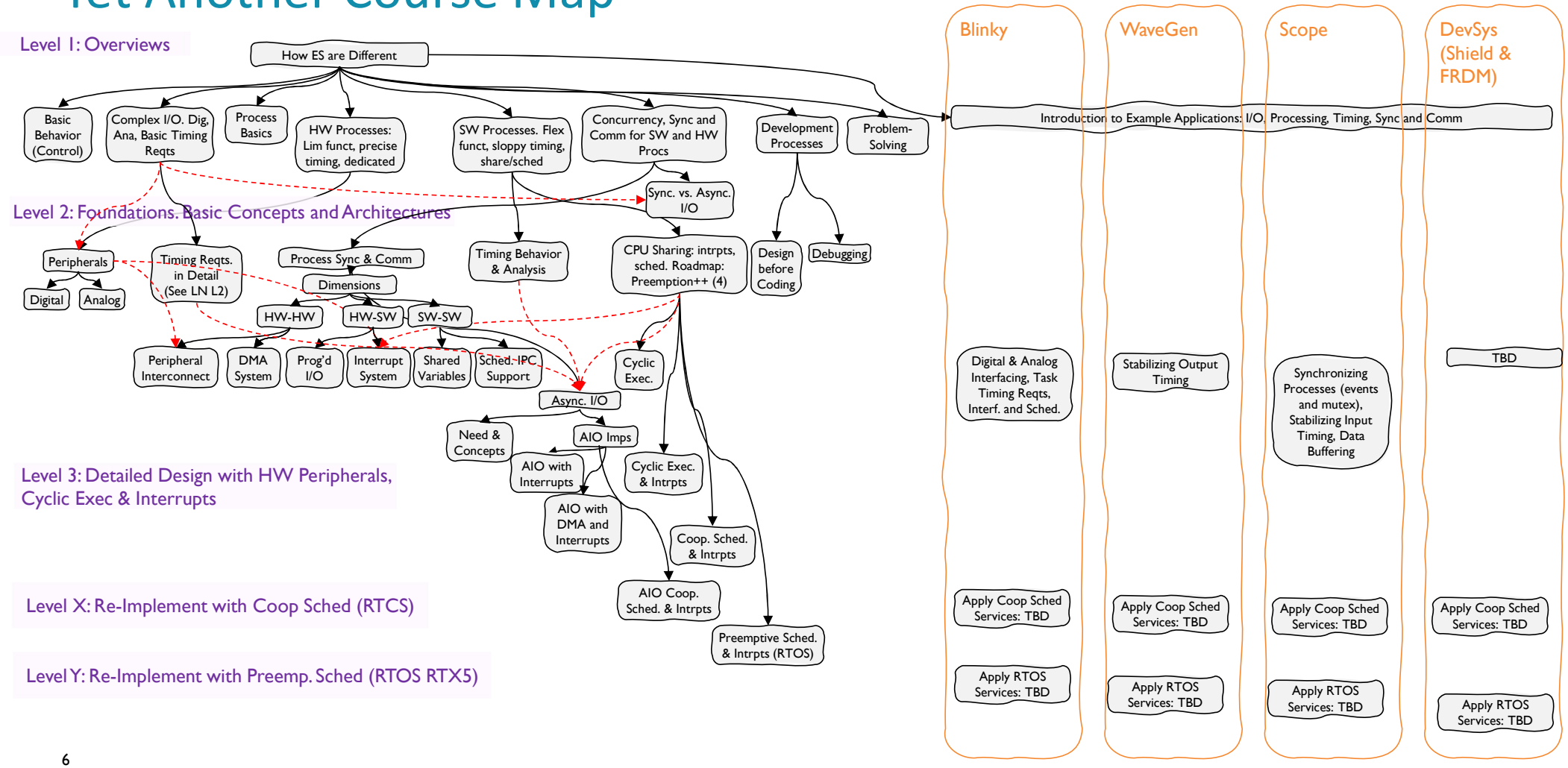
Embedded Systems High-Level View (2)

- The **I/O** for a process often has **challenging timing requirements**
  - **Periodic** events**,** events **synched to other/previous events** on this/other signals
- **Decouple the I/O** 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**.**
  - We may move some processing to **hardware peripheral circuits**.
- These processes need to **synchronize** and **communicate** (data buffering).
- We use **interrupts**, **HW peripherals** and **DMA** to make a **low-cost** and **feasible** solution with a low-frequency CPU.

High-Level Topic Map

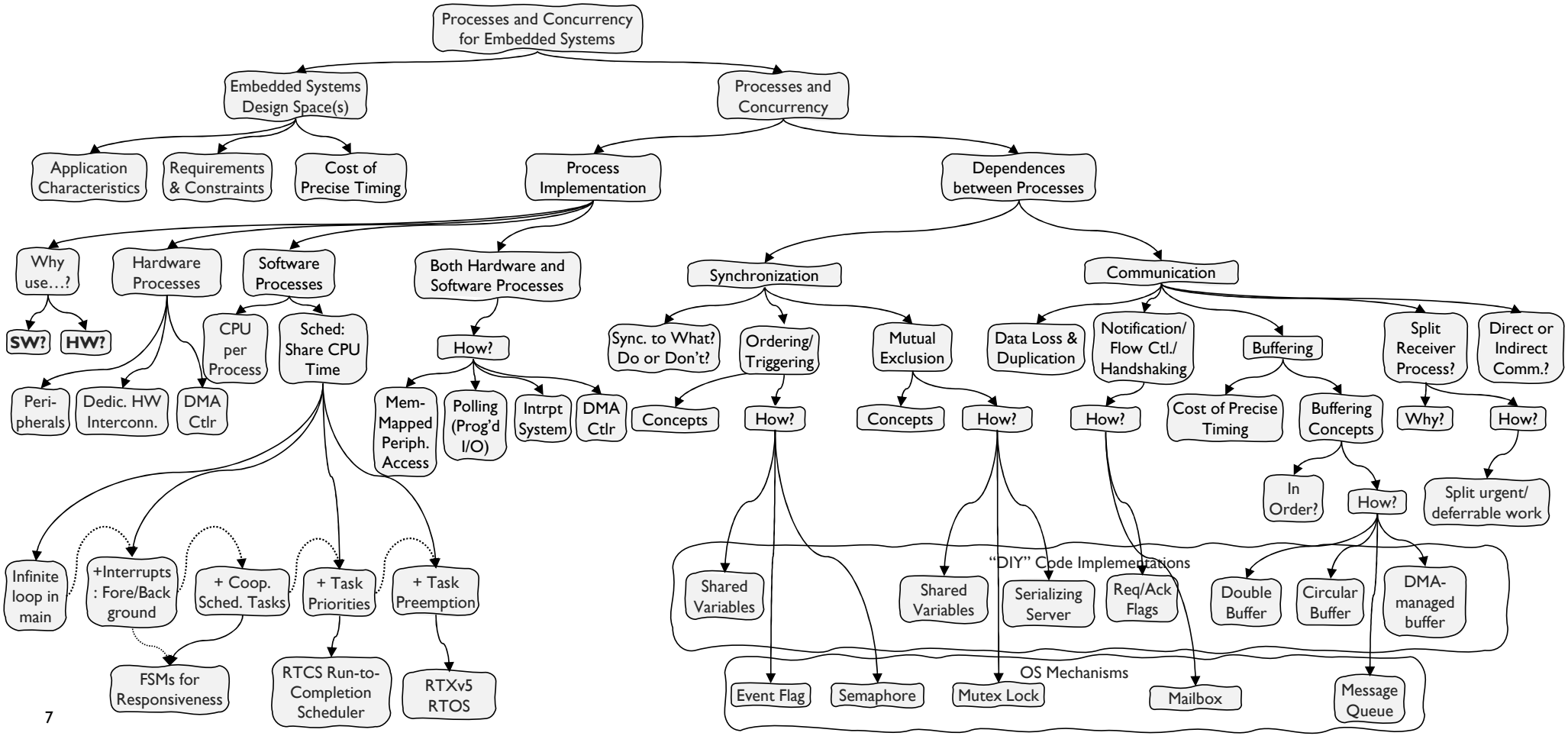


Yet Another Course Map



NC STATE UNIVERSITY

Extending the Topic Map



NC STATE UNIVERSITY

Apply to Examples

Concepts and Methods											Problem-Solving					Examples								
Application Requirements		Development Processes	Process Implementation		Process Scheduling		Process Synchronization and Communication				Correct Functionality	Timing Stability	Responsiveness	Compute Efficiency	Throughput	Blinky Lights	Waveform Generator	Oscilloscope	FRDM		Shield			
Inputs, Outputs, Functionality	Timing, other Non-Functional		HW	SW	HW	SW	HW	SW	SW->HW	HW->SW									Serial Comm.	I^2C Comm.	LCD Controller	Touchscreen	SMPS Controller	µSD via SPI

NC STATE UNIVERSITY

Many Interconnected Methods

Concepts and Methods										Problem-Solving				
Application Requirements		Development Processes	Process Implementation		Process Scheduling		Process Synchronization and Communication			Correct Functionality	Timing Stability	Responsiveness	Compute Efficiency	Throughput
Inputs, Outputs, Functionality	Timing, other Non-Functional		HW	SW	HW	SW	HW	SW	SW->HW	HW->SW					
User interface, Control Systems, Media DSP, Data logging, Sensor data processing & fusion, etc. …	I/O event timing, internal timing, power and energy consumption, code size	Defining requirements, Design before coding, Estimation, Design for X, Testing, Dev. Processes for dependable and safety-critical systems	Peripherals, DMA controller Prog. logic,	Source code, build toolchain, object code	Peripheral interconn., DMA	Events vs. polling, While 1 loop, Interrupt system, Cooperative tasks, Preemptive Tasks. Priorities, preemption	Peripheral interconn., DMA	Shared variables with algorithms, OS/Language support	Sync. Output, Async. Output, Data buffering	Sync Input, Interrupts, Async Input, Data buffering	Concurrency bugs, Testing, Debugging, Dependable system architecture	Timing analysis, Time synchronization, Timer peripheral, sched/OS timer, preemption & blocking	SW process Timing Analysis , System Response time analysis, Prioritization, blocking, preemption, Real-Time	Overhead, batch processing, SW->HW	Overhead, batch processing, SW ->HW

NC STATE UNIVERSITY

Class 02 Overview

- Review of **how** ES computers are different from GP, and **why**
  - Diagram with factors and decisions
  - Scope example
    - Uses common hardware peripherals to offload work from software, improve performance
    - Scope triggering is one kind of synchronization
    - Low cost hardware
  - Timing
    - Timing variability of software
    - System response time
- Example application overview and types of timing requirements
  - ECE 306 line-following car
  - Motor speed and position control
    - Input timing requirements for shaft position encoder. In -> Compute. Response activities, response time limits shaft speed, missing deadline may give wrong direction.
    - Output timing requirements for variable speed (pulse-width modulated) motor drive. Out -> Out. Missing deadline (early or late) affects motor speed, but less critical (inertia limits impact of error).
  - Waveform Generator
    - Stabilize output update time
- Comparing timing requirements with system timing capabilities and behaviors of hardware and software

Computers for Embedded Systems vs. General-Purpose Systems

NC STATE UNIVERSITY

*"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 …)
- 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

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.

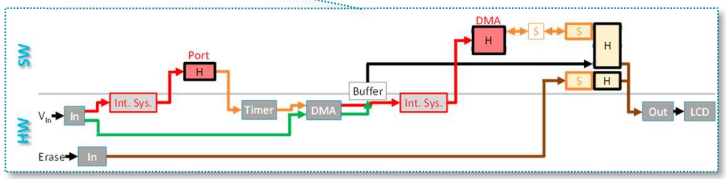
When you have only a hammer, everything looks like a nail. CS education typically omits digital design (other than CPU, maybe memory system, AI accelerators, …).

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)

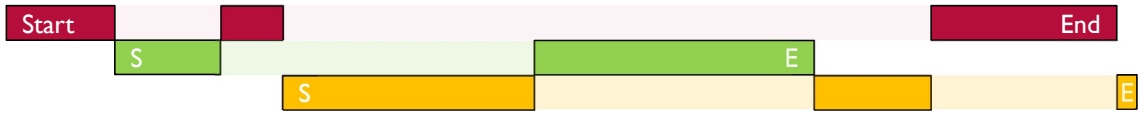
Implementations & Mechanisms (outside of CPU ISA)

- **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)**

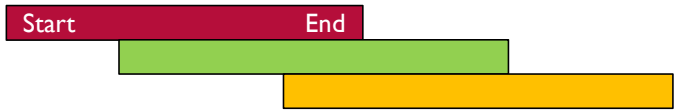
Process Relationships

- 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

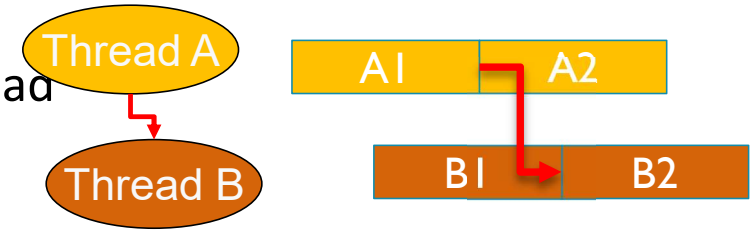
1 Core

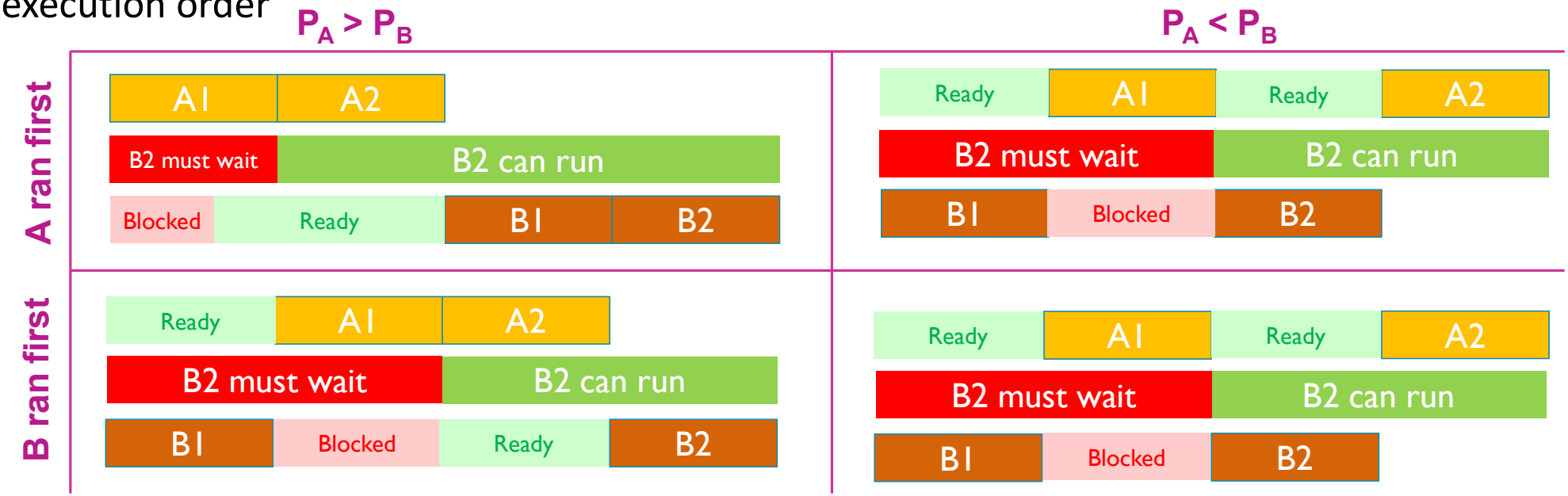
2 Cores

3 Cores

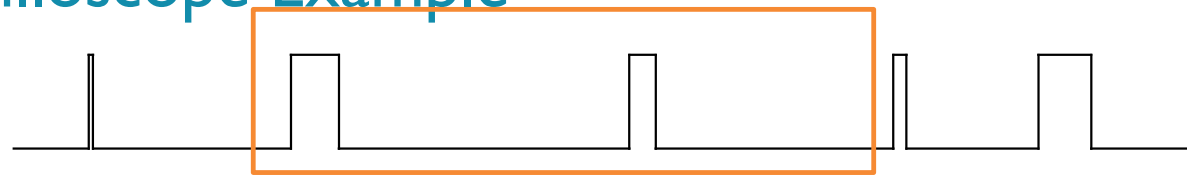
May Need to Synchronize Process Execution

- (A thread is a type of a software process)
- Don't let Thread B start to execute section B2 until Thread A has completed section A1
  - Includes case where each thread has only one section
- Four possible cases based on thread priority, initial thread execution order

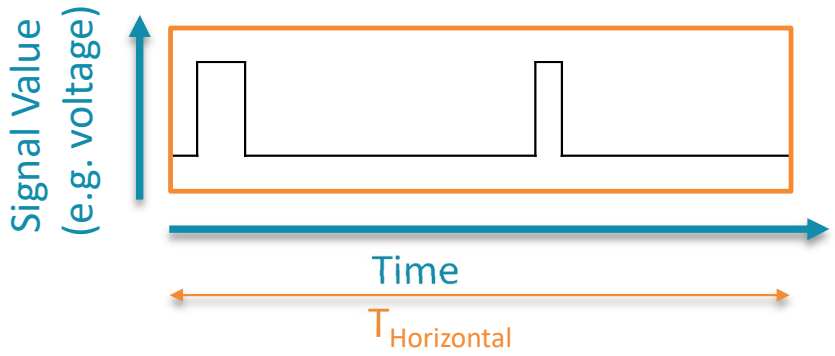




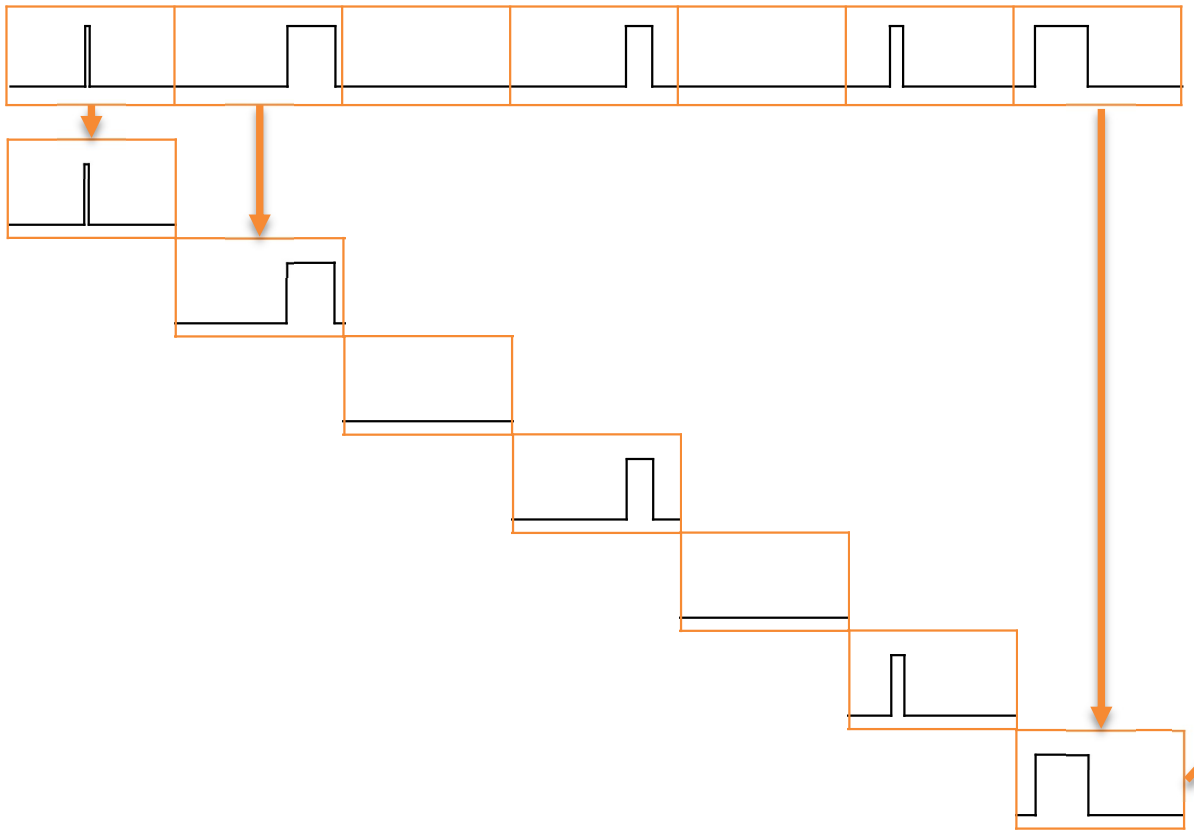
Synchronization: Simple Oscilloscope Example



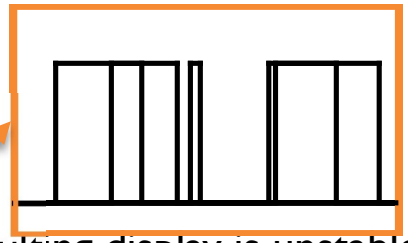
- Input signal
  - Start with simple one-bit digital signal (do analog later)
  - Pulses have irregular start times, changing pulse widths
- Displaying the signal
  - Oscilloscope ("scope") plots signal value (e.g. voltage) vertically vs. time horizontally
  - Horizontal time base determines amount of 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

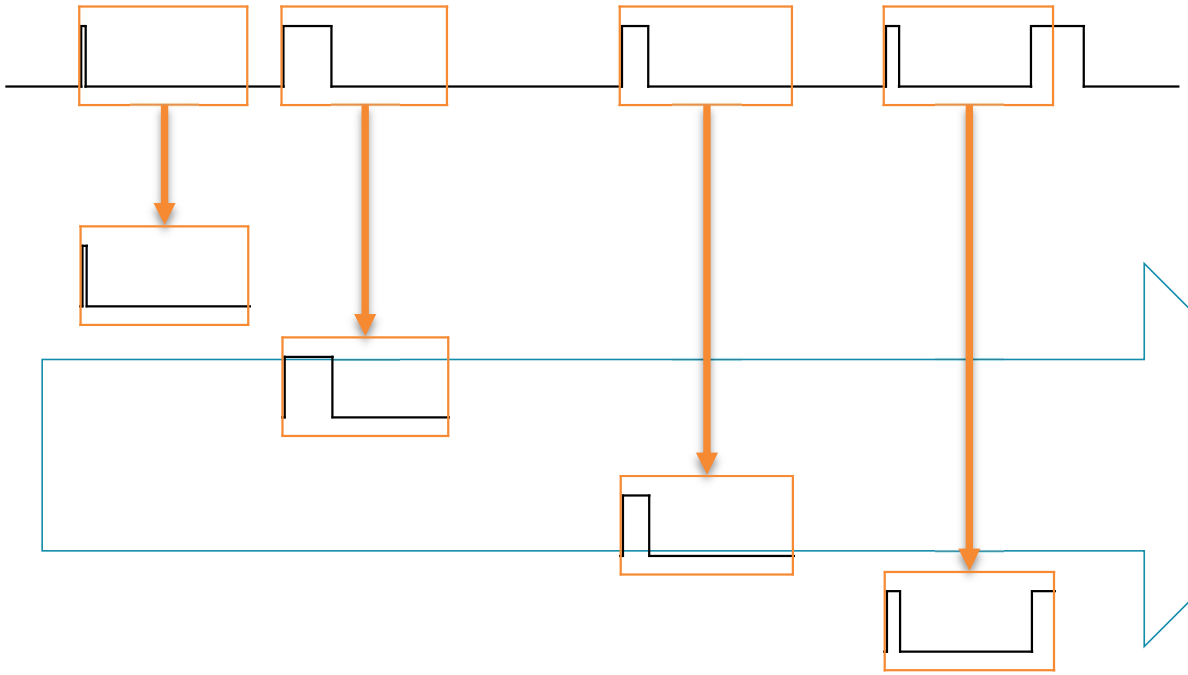


- Sequence
  - Display signal from 0 to T_{Horiz}
  - Display signal from T_{Horiz} to $2*T_{Horiz}$
  - Display signal from $2*T_{Horiz}$ to $3*T_{Horiz}$
  - Display signal from $3*T_{Horiz}$ to $4*T_{Horiz}$
  - Display signal from $4*T_{Horiz}$ to $5*T_{Horiz}$
  - Display signal from $5*T_{Horiz}$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

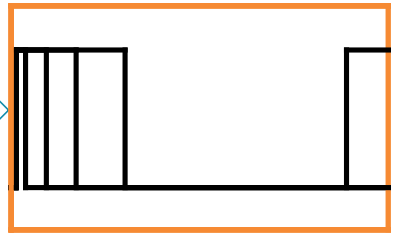


- Resulting display is unstable, jumps around over time.

Stabilize Display with Triggering

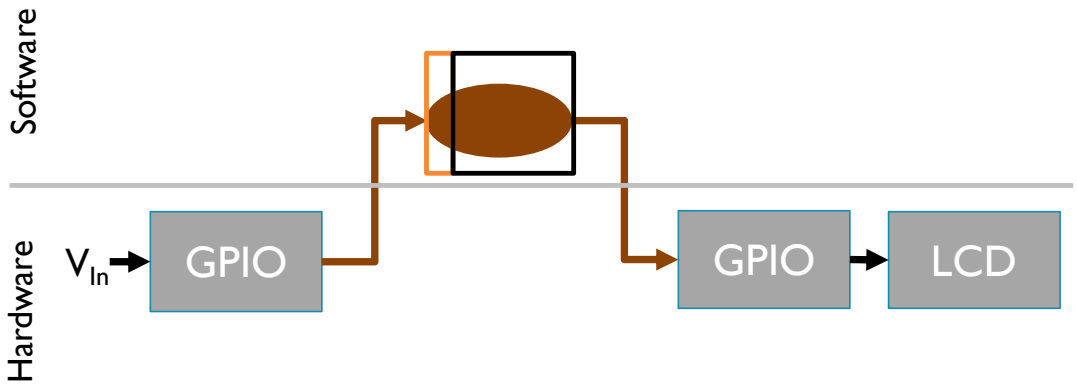


- Scope does nothing until triggered
- Event from input signal (e.g. 0 to 1 edge) triggers scope to start displaying signal
  - Triggering **synchronizes** the scope's start of data display to input signal event



- Resulting display is much more stable
  - Rising edge of signal is stable
    - Except for last acquisition, where time between rising edges < T_{horiz}
  - Falling edge is not stable, because pulse width varies

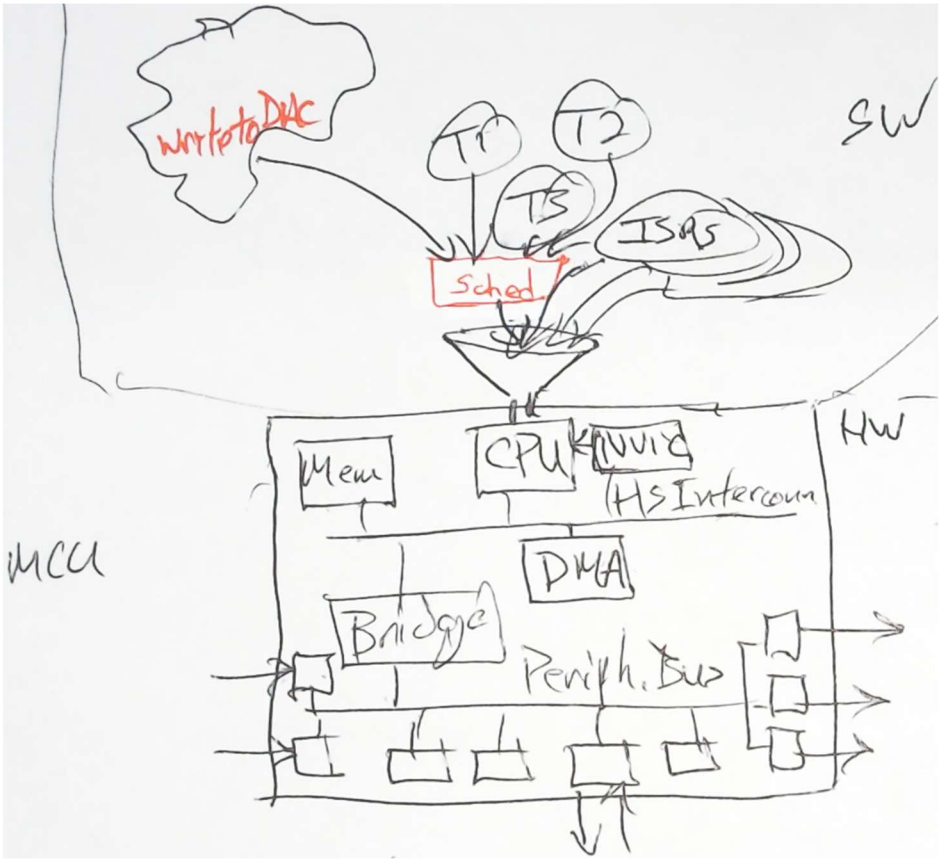
Simple Busy-Wait Loop



- Synchronize: In Process A
- Schedule: Implicit
- Dispatch: Implicit

```
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);
}
```

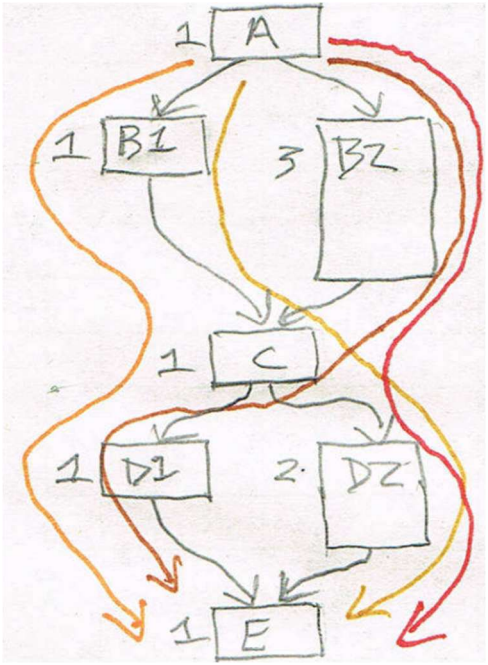
Use Software or Hardware? Flexibility vs. Timing Stability

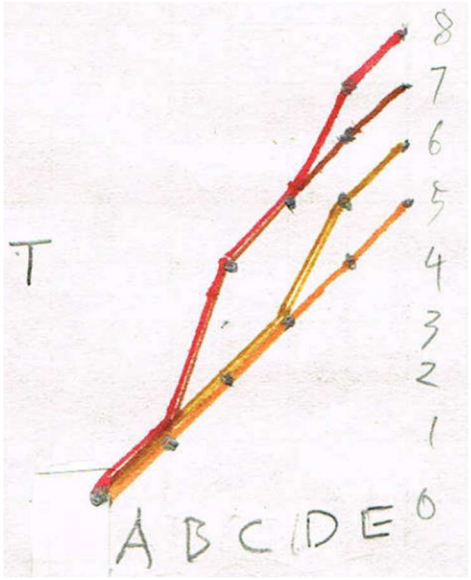


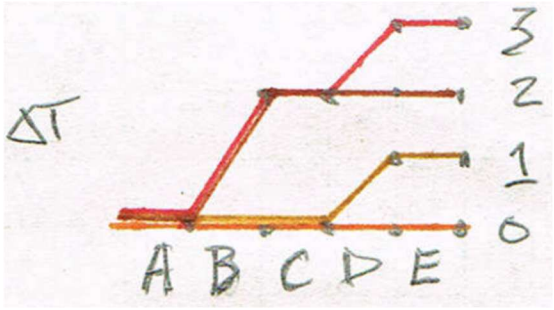
- Software
  - Program gives very flexible functionality
  - Interrupt system (e.g. NVIC) and scheduler (if any) determines **what** software runs on CPU and **when**
  - Software very vulnerable to timing interference. **Need synchronization.** Use interrupts, scheduler to improve timing stability
-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

NC STATE UNIVERSITY

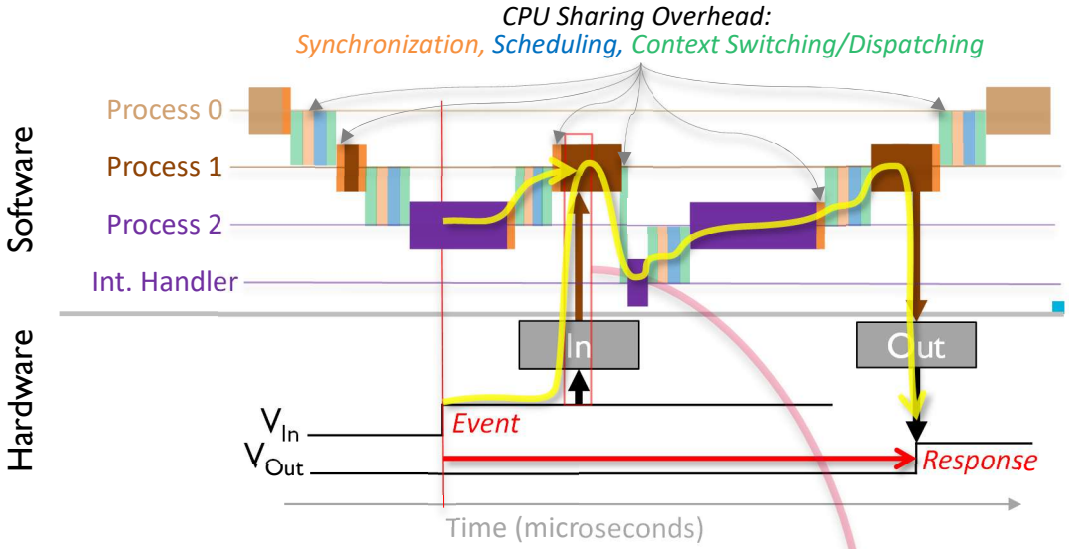
Software Timing Analysis



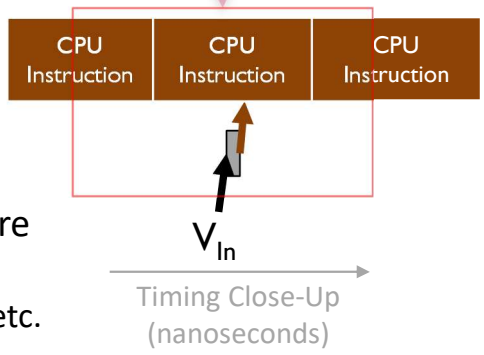




System Responsiveness Depends on Processes



- Responsiveness depends on *sequence of activities* between input event and system's response
- 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.

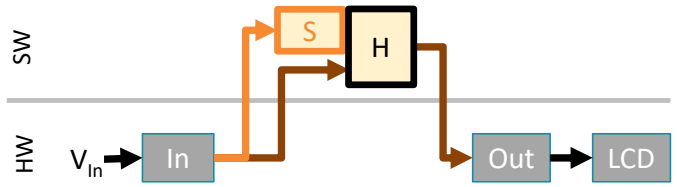


Software process timing: much slower, unstable, hard to predict precisely

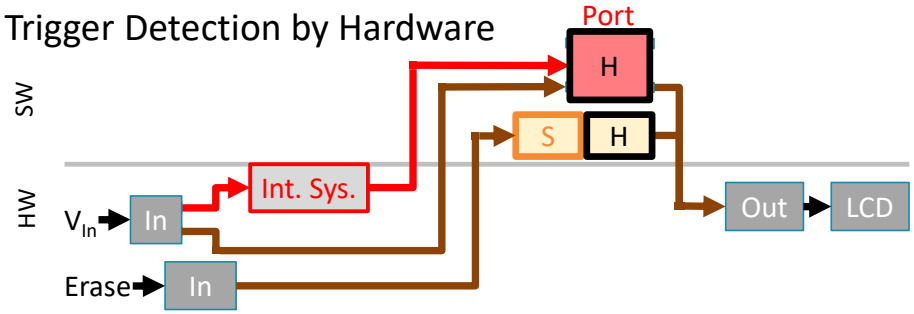
- Time to execute a software process is hard to tell from source code. Often varies when input data triggers different behavior (conditionals, loops, etc.)
- Sharing CPU among multiple software processes delays a process
  - Inherent delays and processing overhead (may be in program, interrupt system, 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)

Improve Timing by Moving from Software to Hardware

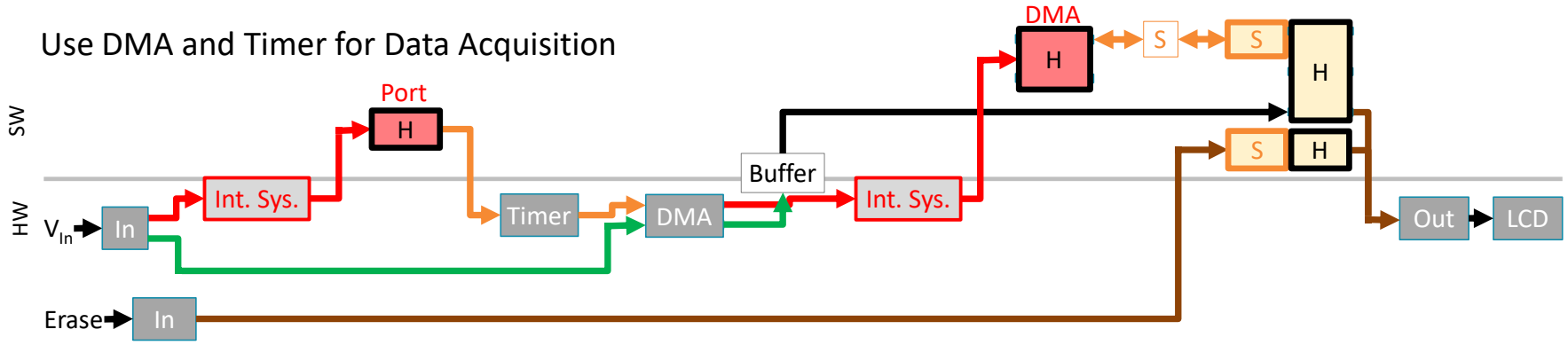
Trigger Detection by Software



Trigger Detection by Hardware



Use DMA and Timer for Data Acquisition



NC STATE UNIVERSITY

Goals – Low Costs

- Development costs
- Hardware costs
  - Slower MCU
- Maintenance costs

NC STATE UNIVERSITY

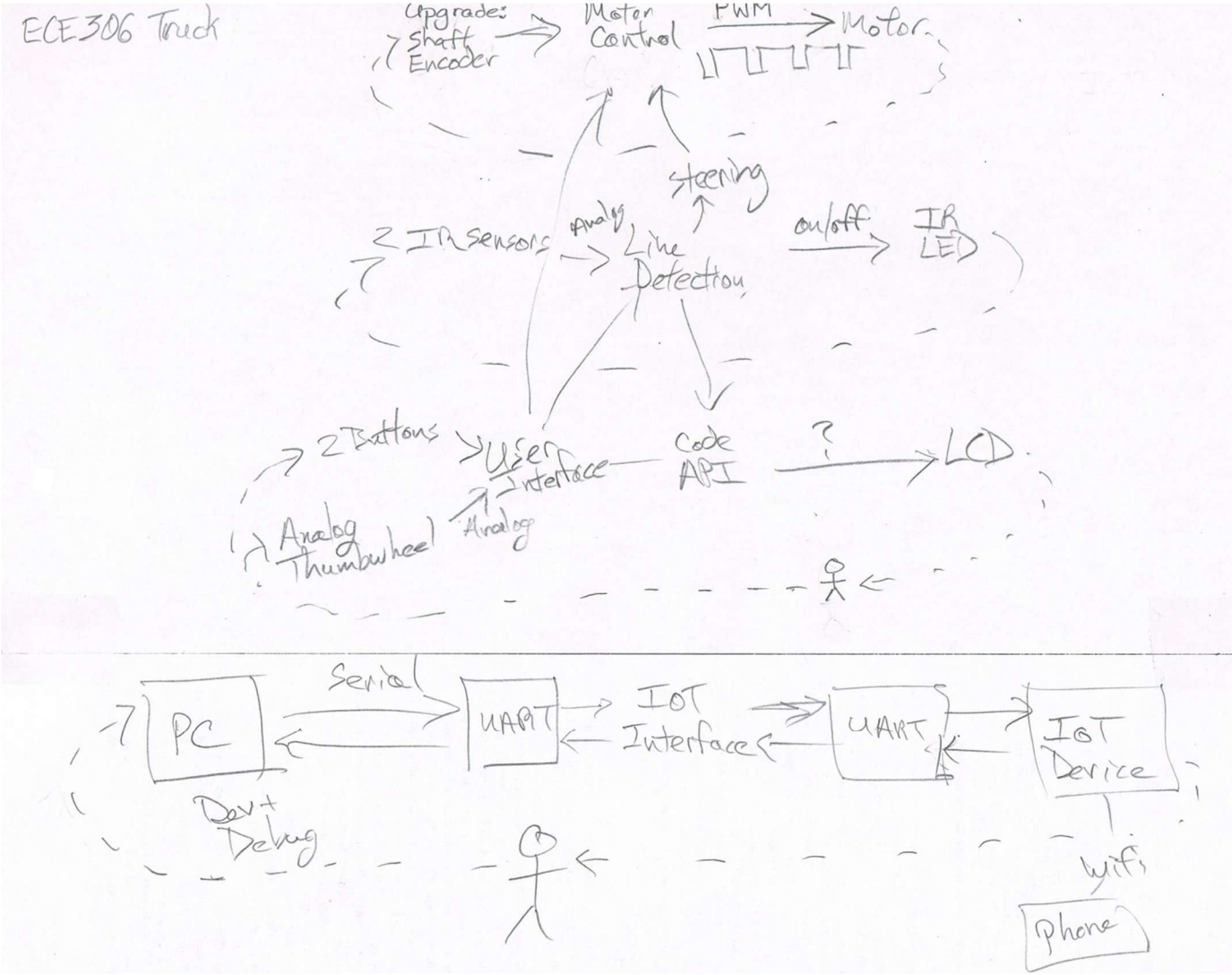
DESIGN EXAMPLES: LEVEL I

NC STATE UNIVERSITY

Scope (Oscilloscope): One Process

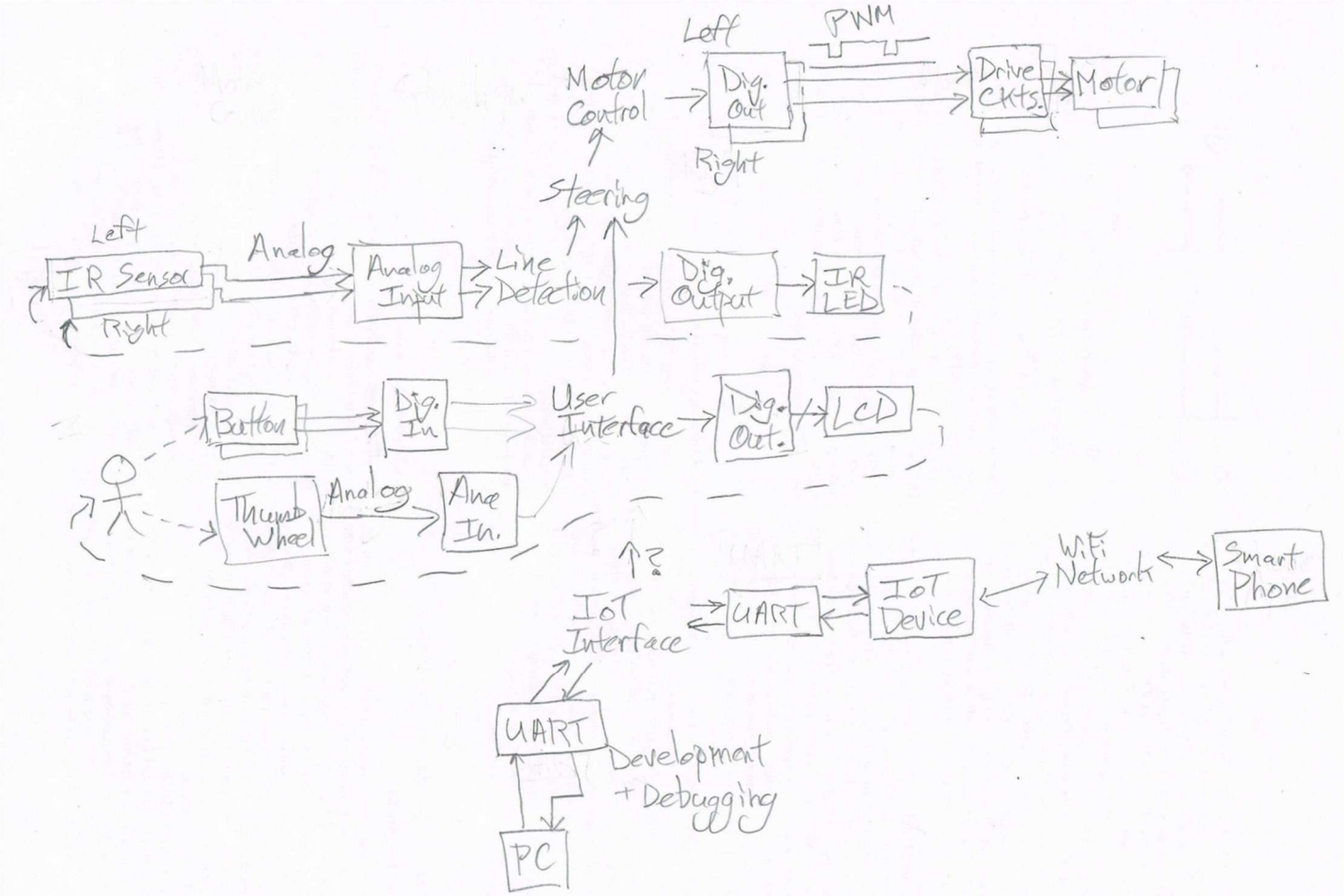
Process	Input Device	Input Peripheral	Processing	Output Peripherals	Output Devices	Timing Requirements

ECE 306 Truck/Car



NC STATE UNIVERSITY

Alt 306 Truck Diagram

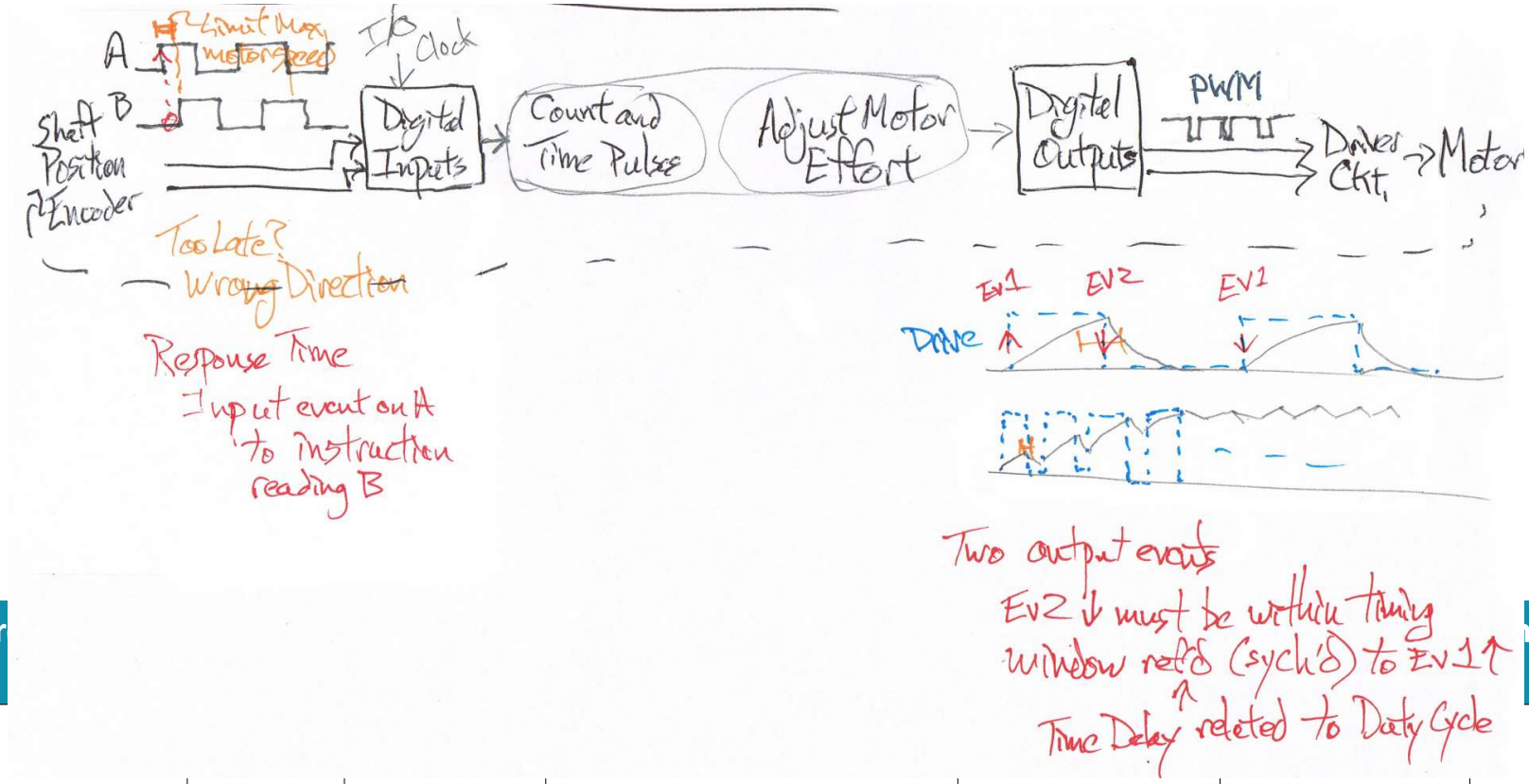


NC STATE UNIVERSITY

Processes in ECE 306 Truck

Process	Input Device	Input Peripheral	Processing	Output Peripherals	Output Devices	Timing Requirements

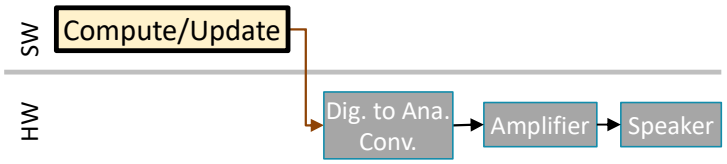
Motor Speed and Position Control

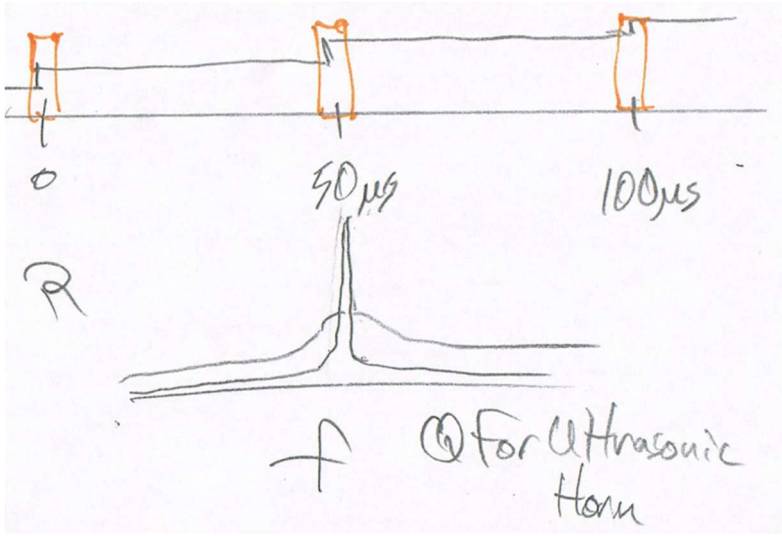


Pr						ming Requirements

Waveform Generator Subsystem: One Process

W1. WaveGen, base design





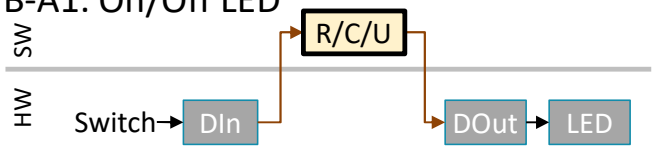
- Part of a larger system with other processes (e.g. user interface)
- Want to update DAC output every 50 us for a 20 kHz update rate
  - DAC signal amplified to drive speaker

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 (?)

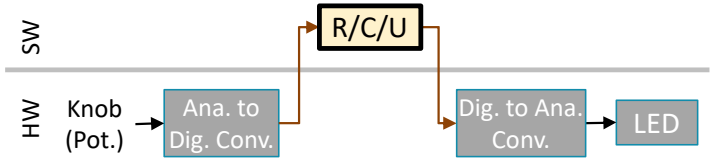
NC STATE UNIVERSITY

Blinky Control Panel: Four Concurrent Processes

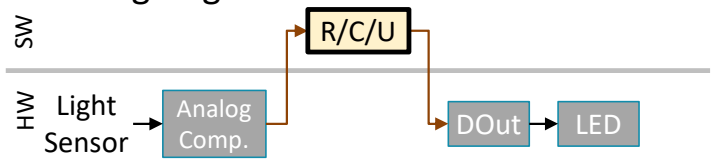
B-A1. On/Off LED



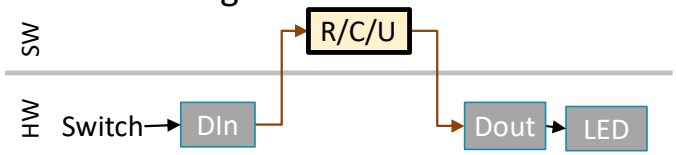
B-C1. Analog-Dimmable LED



B-B1. Nightlight LED



B-D1. Flashing LED



Process	Input Device	Input Peripheral	Processing	Output Peripheral	Output Device	Timing Req'd.
A: Switched LED	Switch	Digital input port	Read port, mask off switch input bit, shift it to LED's bit position in output port and write it.	Digital output port	LED	Within 100 ms
B: Night-Light LED	Photosensor	Analog comparator	Read port, mask off comparator's output bit, shift it to LED's bit position in output port.	Digital output port	LED	Within 500 ms
C: Dimmable LED	Potentiometer voltage divider	Analog-to-digital converter (ADC)	Convert analog voltage to digital value, process reading (negate and scale), convert digital value to analog voltage	Digital-to-analog converter (DAC)	LED	Within 100 ms
D: Switched Flashing LED	Switch	Digital input port	Read port, mask off switch input bit, shift it to LED's bit position in output port and write it.	Digital output port	LED	Within 100 ms

NC STATE UNIVERSITY

FRDM: Serial Communications Subsystem

Process	Input Device	Input Peripheral	Processing	Output Peripherals	Output Devices	Timing Requirements

NC STATE UNIVERSITY

FRDM: Accelerometer (& I^2C) Subsystem

Process	Input Device	Input Peripheral	Processing	Output Peripherals	Output Devices	Timing Requirements

NC STATE UNIVERSITY

Shield: SMPS Controller Subsystem

Process	Input Device	Input Peripheral	Processing	Output Peripherals	Output Devices	Timing Requirements

NC STATE UNIVERSITY

Shield LCD Interface:

Process	Input Device	Input Peripheral	Processing	Output Peripherals	Output Devices	Timing Requirements

NC STATE UNIVERSITY

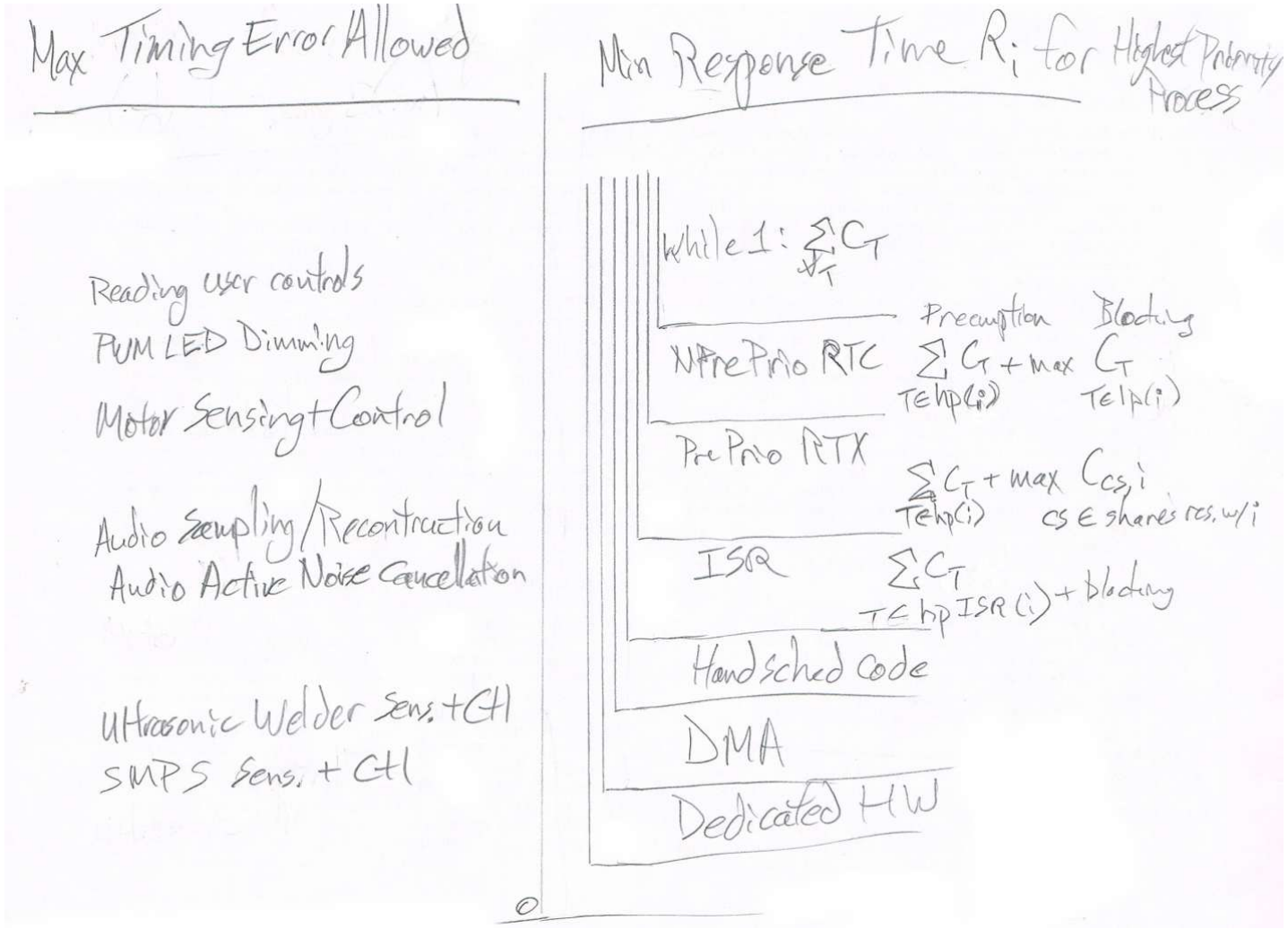
Shield: Touchscreen Interface

Process	Input Device	Input Peripheral	Processing	Output Peripherals	Output Devices	Timing Requirements

NC STATE UNIVERSITY

Timing Requirements

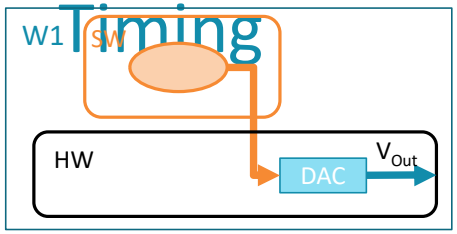
Timing Requirements vs. Capabilities

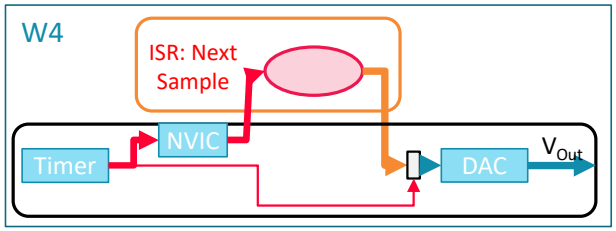


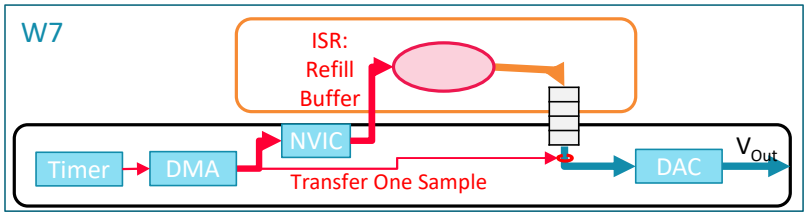
NC STATE UNIVERSITY

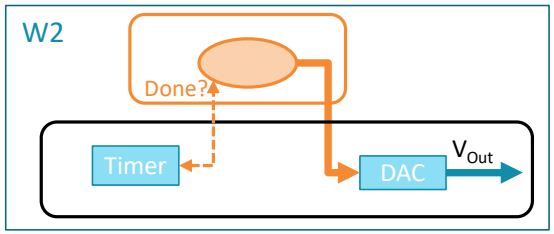
Timing Characteristics of Software

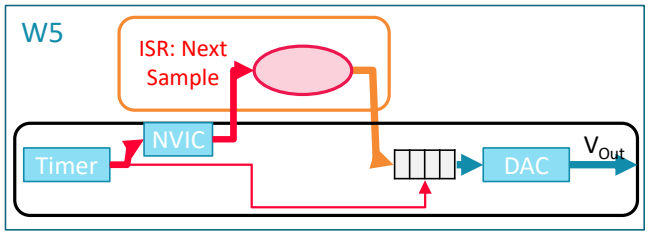
Waveform Generator Design Evolution with Software and Hardware Components: Want Output Updates with Stable Timing

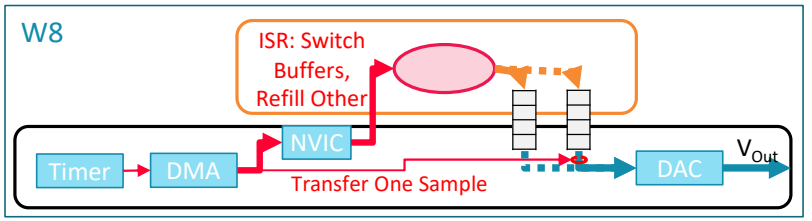


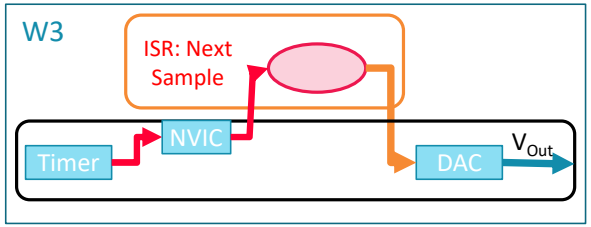


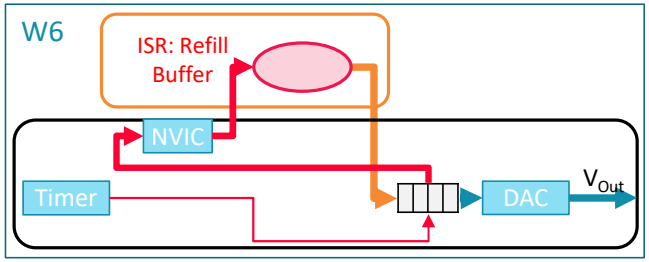


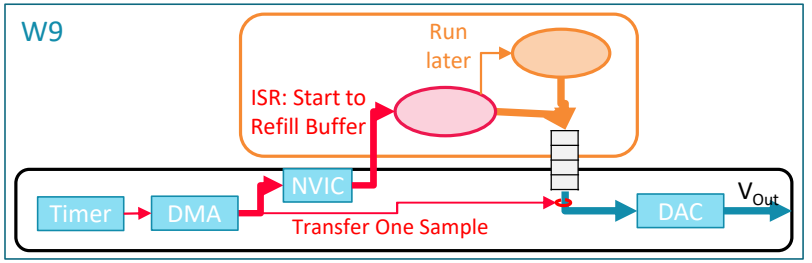




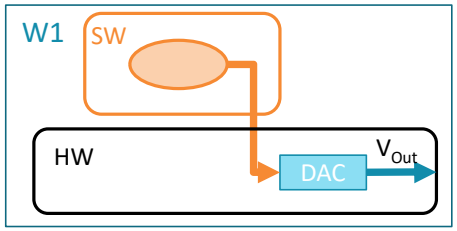


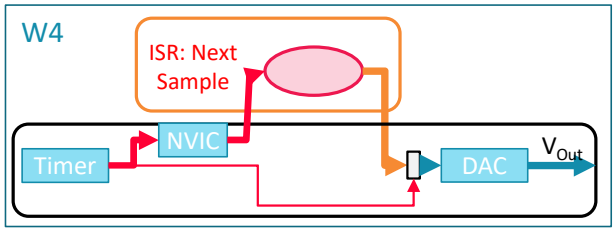


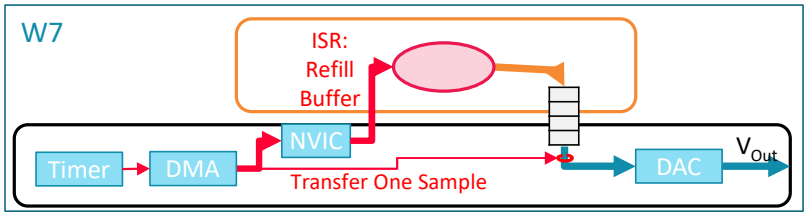


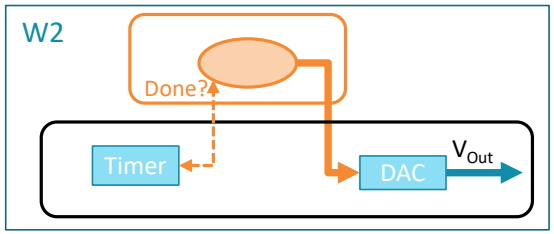


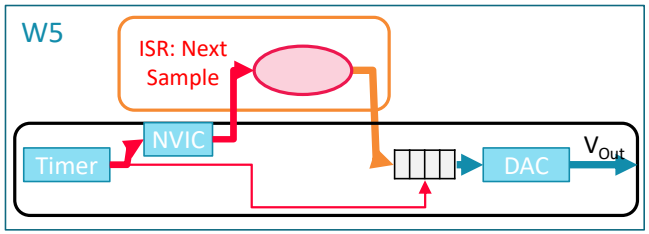
Waveform Generator Design Evolution with Software and Hardware Components: Want Output Updates with Stable Timing

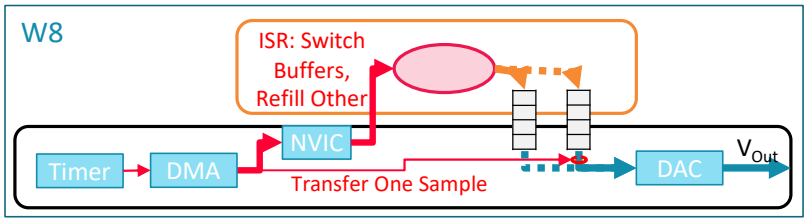


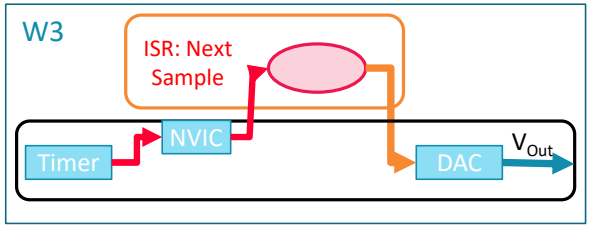


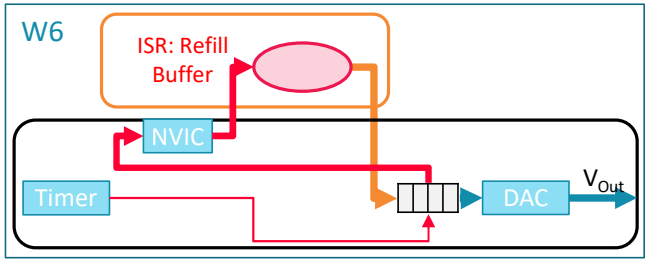


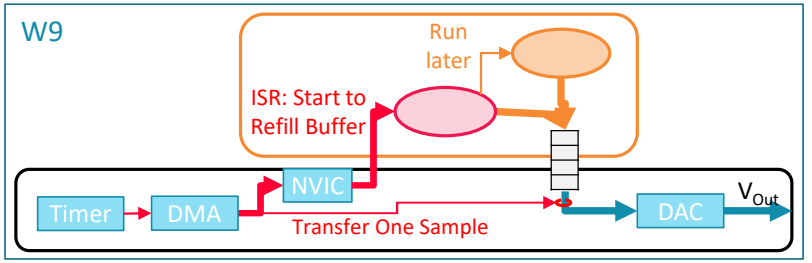






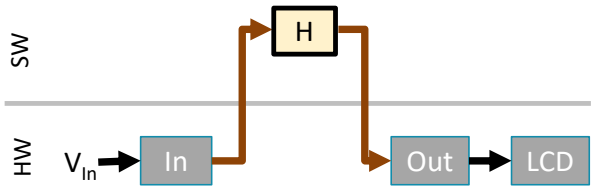




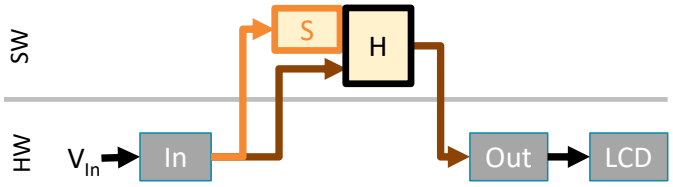


Scope Design Evolution with Software and Hardware Components

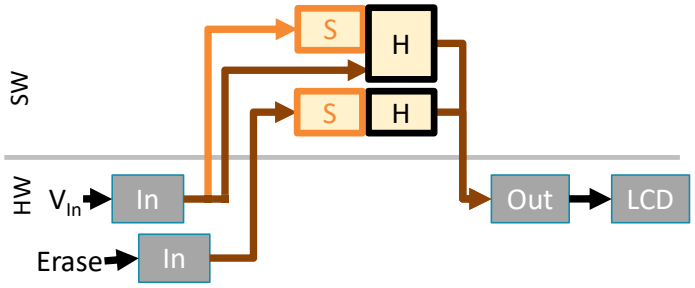
1. Basic



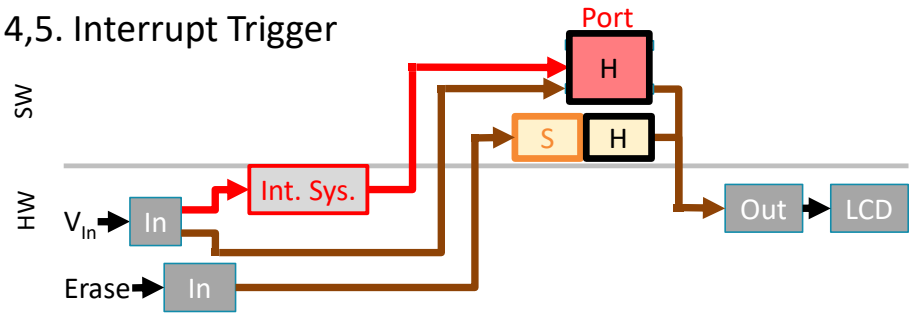
2. Polling Trigger



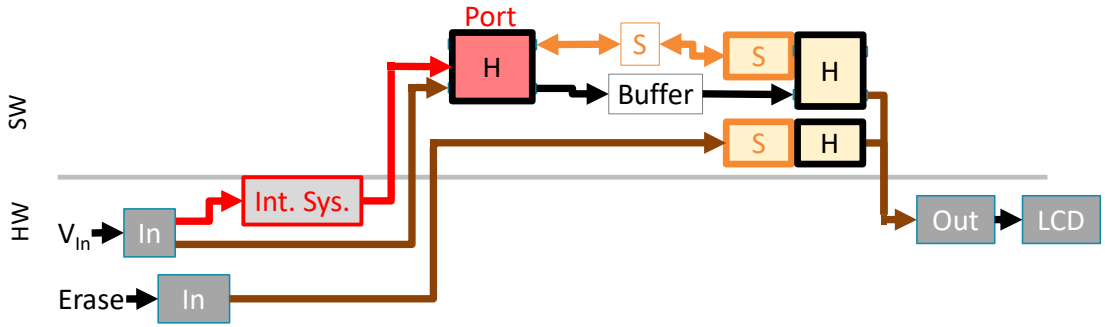
3. Erase Button



4,5. Interrupt Trigger

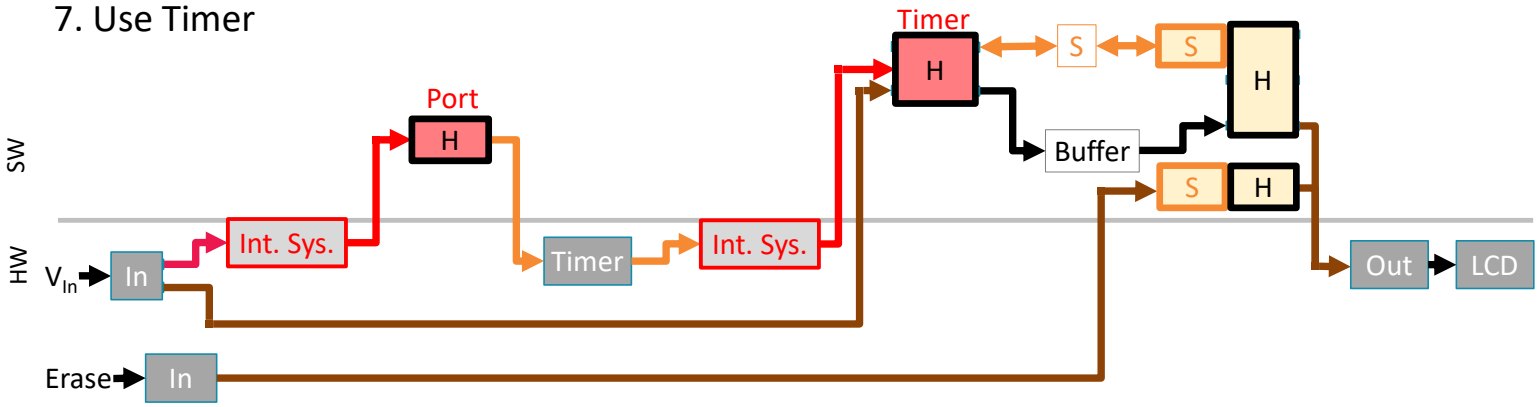


6. Defer LCD Updates

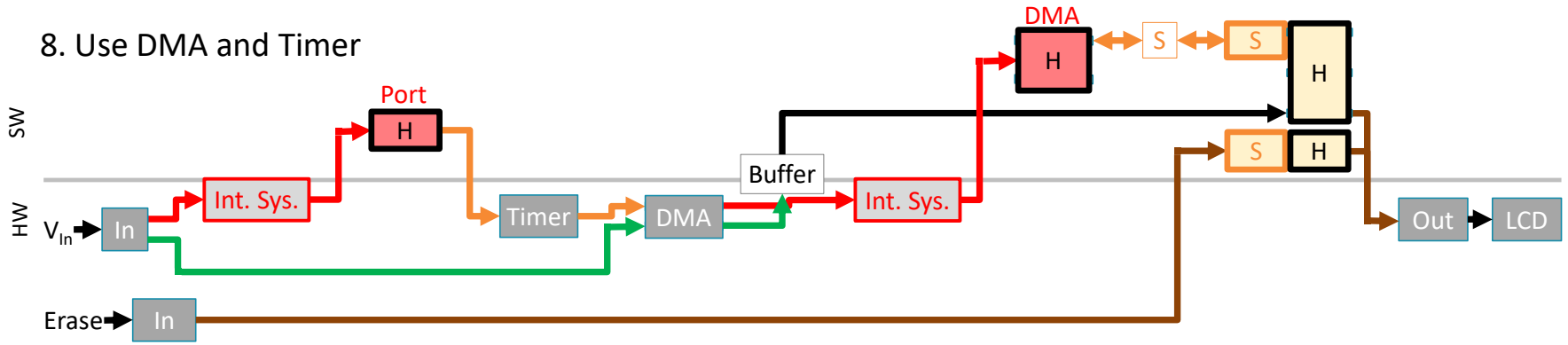


Scope Design Evolution with Software and Hardware Components

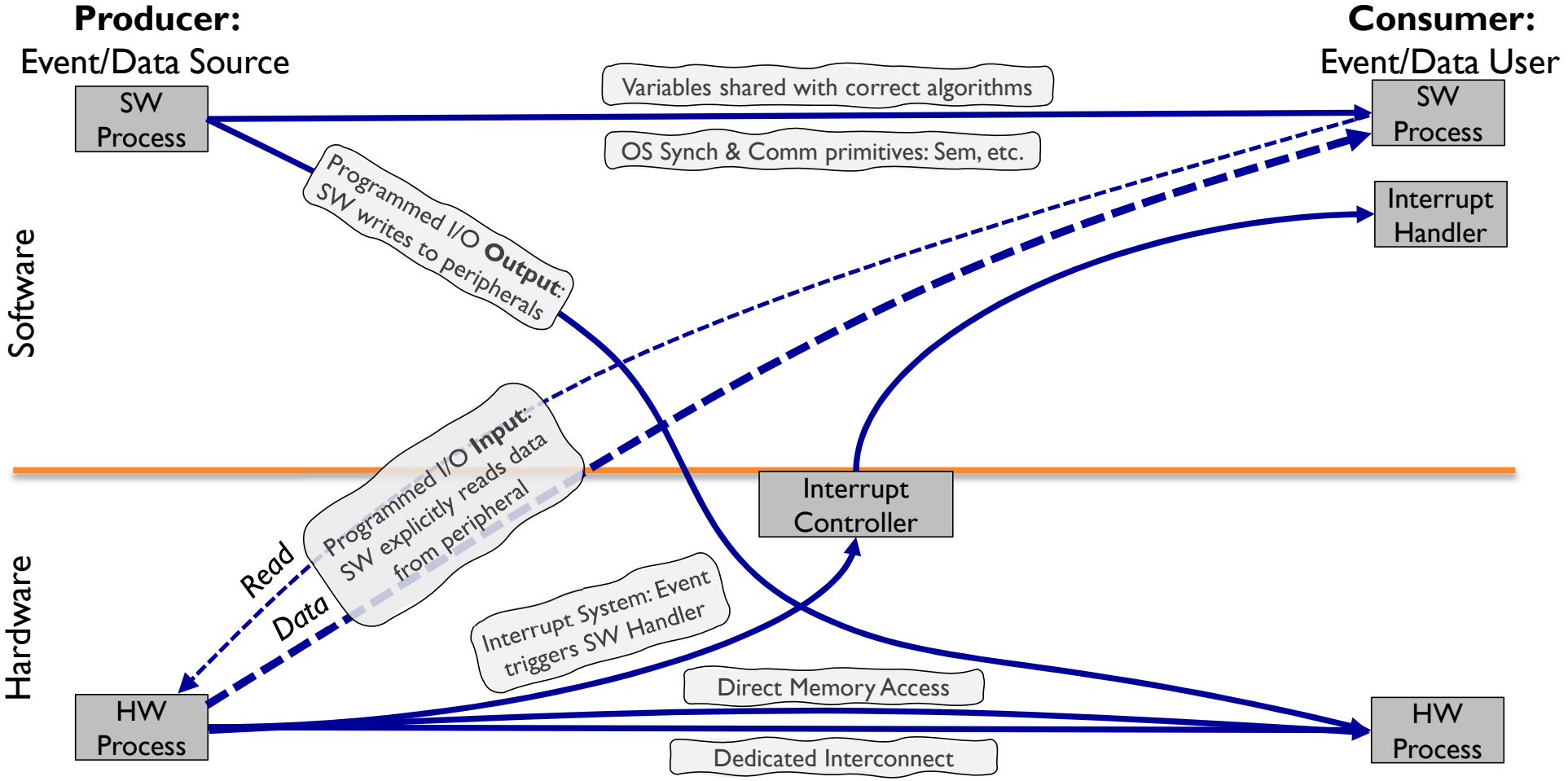
7. Use Timer



8. Use DMA and Timer

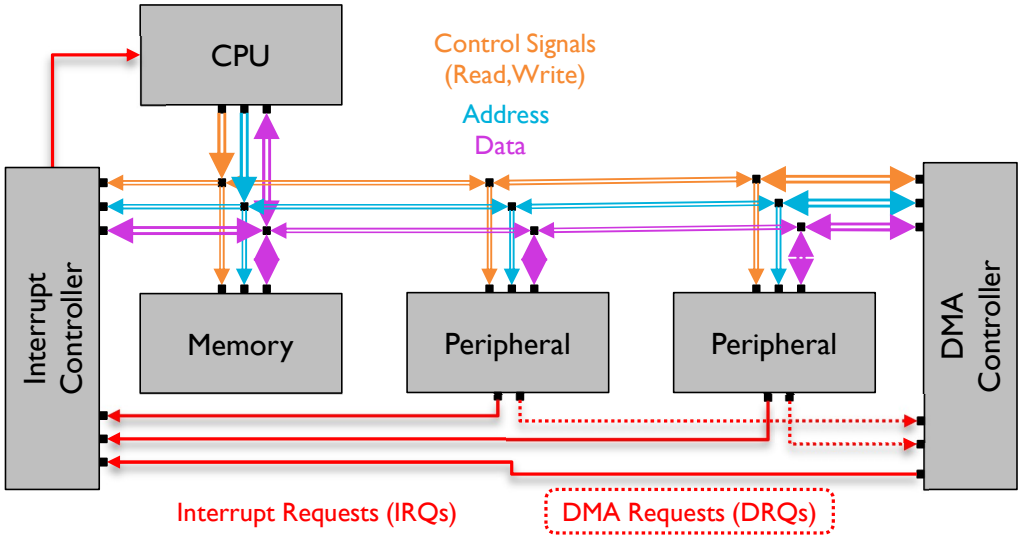


Sync. and Comm. Paths for HW and SW Processes



Direct Memory Access Controller

Allows Hardware->Hardware communication without using CPU

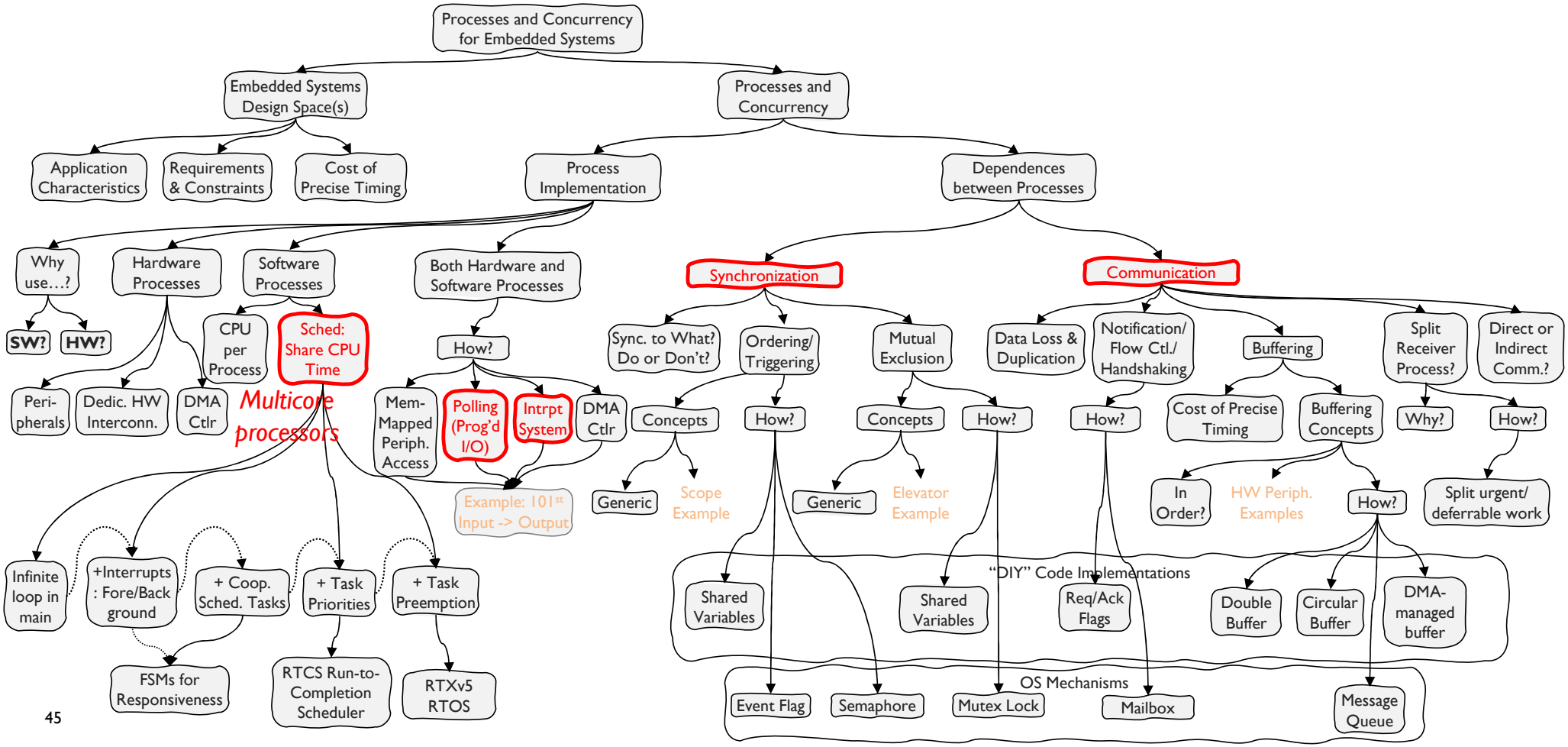


- How to access memory and peripherals?
  - CPU uses **memory bus** (address, data, control) to access memory and peripheral devices
  - Memory bus can also be controlled by DMA Controller (DMAC) peripheral

- 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

Big Picture 2: Synchronization, Communication and Scheduling

All three are interconnected. Different goals -> diff. design points -> diff. implementations



General Approach

- Teach through problem solving for design challenges. How to …
  - Read inputs, write outputs. Simple digital, analog
  - Stabilize timing for reading inputs, writing outputs
  - Support multiple processes
  - Synchronize processes
  - Let processes within system communicate
  - Let different systems communicate (protocols: SPI, serial/UART, I^2C, etc.)
  - Analyze and improve system responsiveness
  - Tolerate timing mismatches/variability
  - Improve dependability and robustness
  - Improve efficiency

- Iterative, demand-driven design process. Pull methods and approaches from "toolboxes"
  - HW peripherals: port, timers/counter, ADC, DAC, comparator, SPI, UART
  - CPU Sharing:
    - Scheduling concurrent software processes (interrupts, non-preemptive & preemptive threads)
    - HW and SW process sync. and comm.
      - SW->SW, SW->HW, HW->SW, HW->HW
    - In design examples, iterative refinement may move sched/sync/comm components between hardware, thread SW, OS SW
  - Response time analysis for SW & HW. Concepts, modeling, experimental measurement
  - Data buffering
- Homework assignments
  - Theory: Concepts, what-ifs
  - Practical: Hands-on development of systems and code, debugging, analysis with test equipment

Concurrent Process View of an Embedded System

A Time-Sensitive System of Concurrent HW and SW Processes Interacting with the Environment and Each Other

- Processes
  - Single vs. multiple processes. Sequential vs. concurrent processes
  - Implementing processes in software (universal functionality, poor timing) or hardware (limited functionality, fast stable timing)
  - Running multiple software processes to share CPU core(s) requires scheduling those processes (big topic)
- Process Interactions
  - Which processes interact?
    - From an input signal/event to a process (e.g. async. input)
    - From a process to another process
    - From a process to an output signal/event
  - What interactions are possible?
    - Synchronization, communication, both
  - Processes aren't just SW, but HW too. Domains and sync/comm methods:
    - SW->SW: user SW, OS mechanisms
    - SW->HW: programmed I/O
    - HW->SW: interrupts, programmed I/O
    - HW->HW: peripheral features, periph. interconnect, DMA transfers
  - Implementation of sched., sync. and comm.
    - Components may be in SW and/or HW.
      - Some are tightly coupled, affecting design choice viability
      - CPU's Interrupt System is built-in scheduler which syncs ISRs to requests (from HW peripherals, system exceptions…)
    - SW components may be in application thread(s), OS or both.
    - Many possible solutions. Must balance efficiency, responsiveness, complexity, maintainability, etc. based on requirements and constraints.

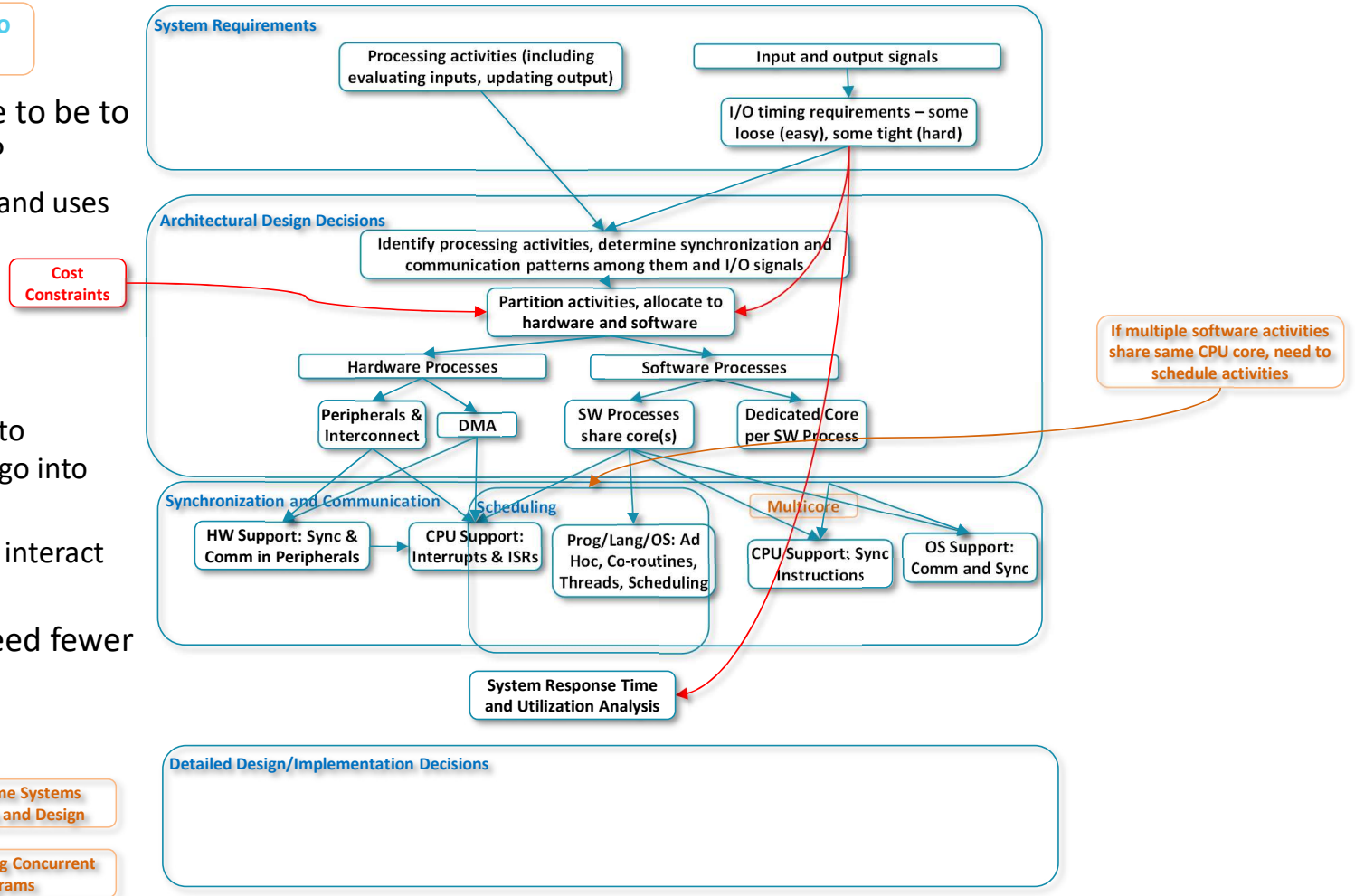
- Understanding System Timing Requirements
  - What to do?
    - Sample inputs, update outputs at given times
    - Respond to input events/changes within a given relative time
  - What is reference for timing requirements?
    - Elapsed time, absolute time (wall-clock). Periodic input every 10 us, etc.
    - Input event, phase of input signal, subsystem state. 0->1 transition, AC power zero-crossing.
  - Strictness of timing requirements
    - Are early/late responses useful? -> Value vs. timing error -> Timing window width, hard vs. soft deadline

- Designing to Meet Timing Requirements
  - What is system's actual timing behavior (distribution, bounds, statistics)?
    - Derive periodic task model (execution times, periodicity, deadlines)
    - Define process interactions
    - Select suitable Sched/Sync/Comm approaches
    - Model system timing behavior based on HW, task model and sched/sync/comm approaches used
    - Add data buffers between processes
      - Tolerate timing mismatches. Set buffer size based on timing of producer activity bursts, and delayed consumer service.
      - Improve efficiency with batch processing, reducing overhead
  - Does it meet requirements? Iterate as needed

Drivers and Constraints

How fast of a CPU do we need to get the work done on time?

- How fast does the CPU have to be to get the work done **on time**?
  - Slower is usually cheaper (and uses less power or energy)
- Depends on
  - Timing requirements
  - Design decisions
    - What parts should go into software? What should go into hardware?
    - How should those parts interact with each other?
- Some design approaches need fewer CPU cycles than others



Concurrency Goals and Background

- Introduce students to features needed to support concurrent HW and SW processes
  - Synchronization: is the process ready to run? or must not run yet?
  - Scheduling (essential if more software processes than CPU cores): pick which ready process to run
  - Dispatching/Context Switching: save previous process's state if needed, then start/resume running the scheduled code
  - Communication: sharing information between processes, often includes synchronization for correctness/hand-shaking/flow control
- Use multiple examples, iteratively refine them
  - Waveform Generator with stable output timing (and more!)
  - Scope (Oscilloscope) with responsive input edge detection (triggering), stable input sample timing (and more!)
- Range of approaches used across embedded systems
  - main while (1) -> ? -> (Arduino) -> ? -> ? -> RTOS -> ? -> ROS -> ? -> Linux
  - Approaches may use both software and hardware to provide features for sync/sched/dispatching/CS/communication
  - Software features may be implemented in one or more places: user thread code, ISR, OS/RTOS.

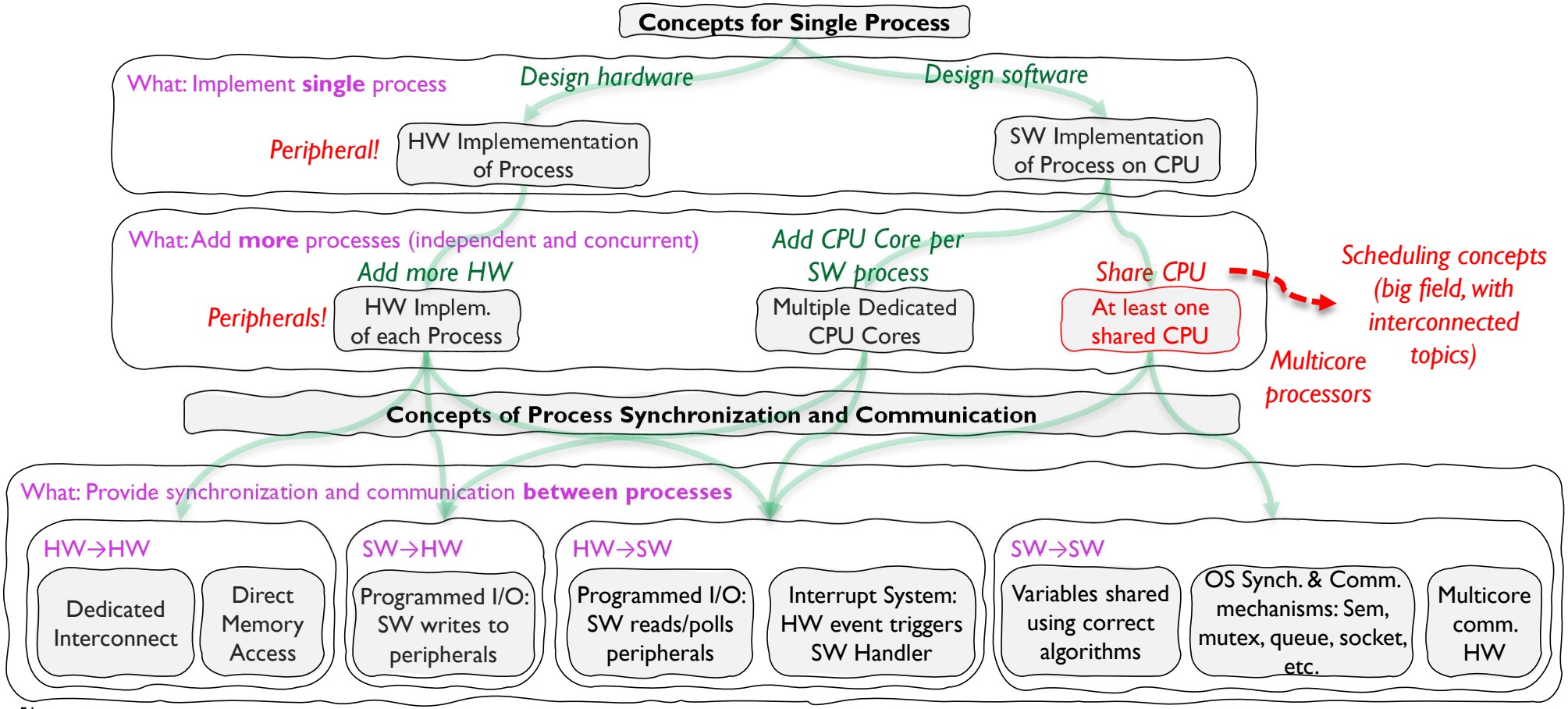
- Key points in scheduler (etc.) design space
  - Start with cyclic executive loop in main thread
  - Add interrupt system and peripherals
    - Peripherals synchronize SW to HW: detect events, request interrupt service
    - Interrupt controller (& CPU)
      - Save partial context (HW register stacking)
      - Schedule highest-priority interrupt request
      - Dispatch handler
      - CPU executes handler, returns from interrupt
      - Restore partial context (HW register unstacking)
      - CPU resumes previous execution
  - Add cooperative scheduling
    - Non-preemptive scheduling of prioritized tasks
    - Dispatch via subroutine calls, so tasks must be run-to-completion
    - Basic synchronization (timer tick, periodic task releases, etc.)
    - Improve responsiveness by converting long tasks into SW FSMs
  - Replace coop. sched. of RTC tasks with preemptive scheduling of run-forever threads. Build on interrupt/exception system:
    - Access OS through exception instructions (software interrupts, supervisor/service calls, etc.)
    - Build context switches on existing partial context save/restore performed in hardware for interrupt/exception processing
    - Leverage PendSVC exception for user ISR to trigger RTOS activities (Arm Cortex-M)

The Basic **Concurrency** Story

1. Concurrency in an embedded system starts long before reaching multithreaded software and/or multicore hardware. It starts with the hardware peripherals in MCUs, which provide concurrent execution of various common activities, simplifying the software and lowering the bar for CPU performance.
2. For concurrent processes to interact, they need synchronization and communication. They need scheduling to get time on a CPU according to the sync/comm relationships. Even the most basic MCU provides a foundation for these needs with its peripheral interconnect, interrupt system, and DMA system. Designers must solve sched/sync/comm problems here to make the system work correctly, and they may cross between software and hardware.
3. The design space changes when you scale up to multithreaded programming and/or multicore systems, but the same problems occur. The available solutions may change (mechanisms in OS/RTOS, hardware peripherals, instructions, memory system support for coherence and consistency) as will their costs. So it is good to learn these concepts now.

Big Picture 1: Building Up a System with Concurrent HW and SW Processes

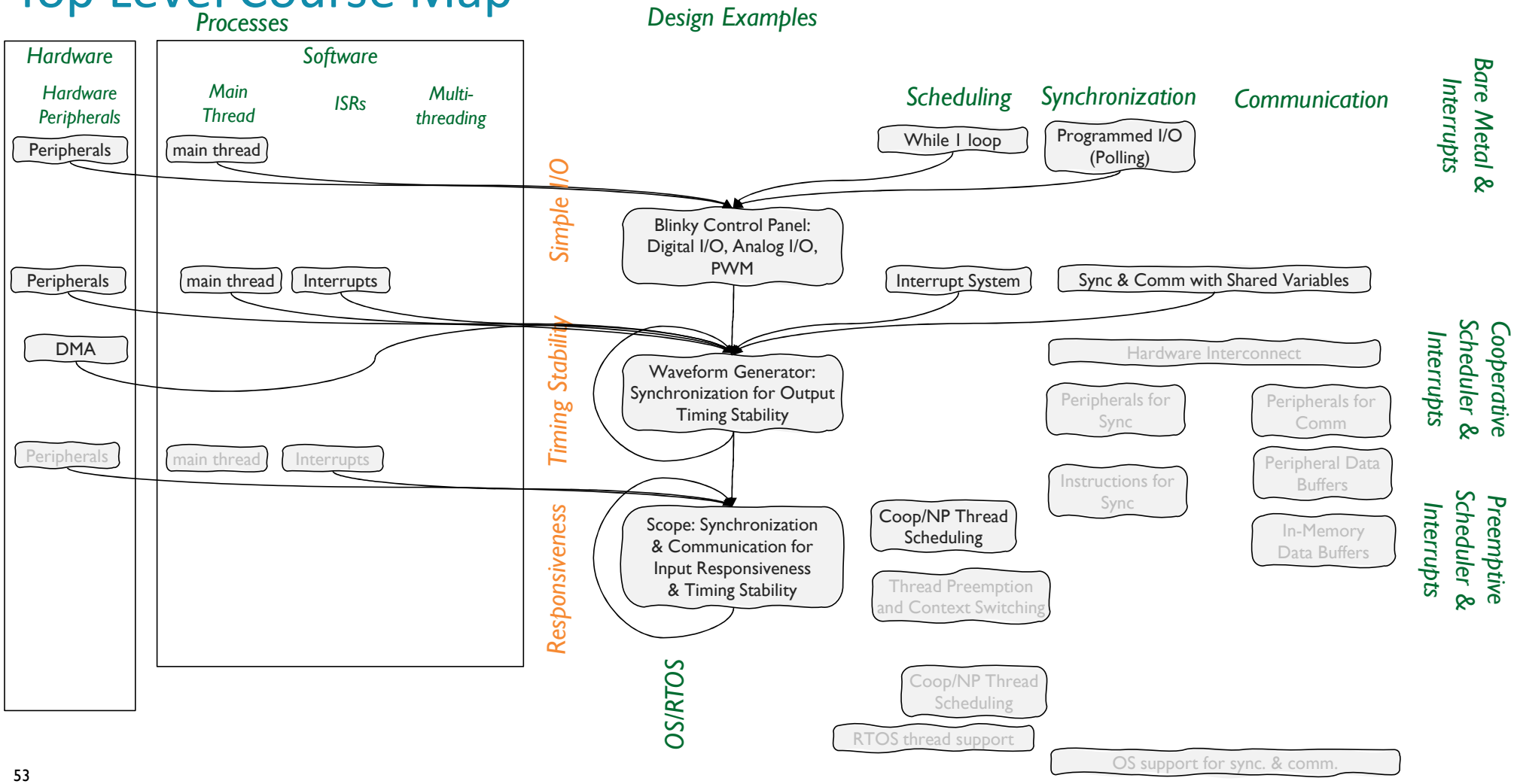
Start simple, then examine options as you build up the system



NC STATE UNIVERSITY

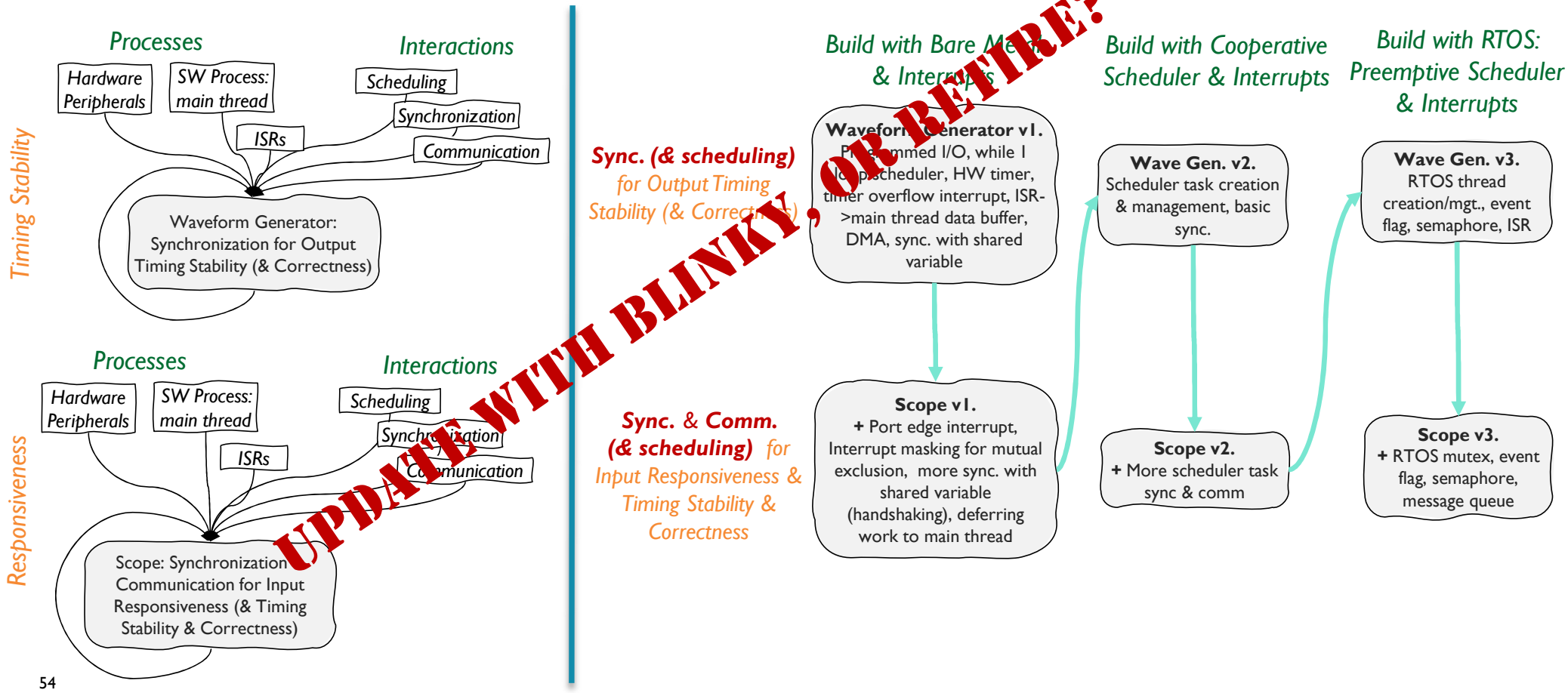
ORPHANS

Top Level Course Map



Main Design Examples Used: Waveform Generator & Scope

Concurrent HW & SW process scheduling, synchronization and communication: Why and How



Getting HW Signals To and From SW: Dimensions

- Software Refresher
  - Runs on digital hardware (CPU, memory, etc.)
  - Uses instructions (read, write, in, out) to read digital value from input hardware, write digital value to output hardware
- Digital vs. Analog
  - Digital: signal has 2 possible levels
  - Analog: signal has >2 (many, infinite) possible levels
- Signal Direction: Input vs. Output
  - Input: HW -> SW
  - Output: SW -> HW

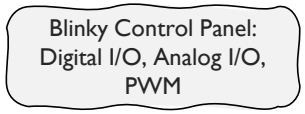
- Timing relationship between SW and HW
  - **Synchronous** timing **couples** SW and HW activities
    - SW instruction execution causes HW signal to be read or written **immediately** (or with tiny fixed delay)
  - **Asynchronous** timing **decouples** SW and HW activities
    - Output: SW write instruction executes, **eventually** HW output event happens
    - Input: HW input event happens, **eventually** SW read executes, getting that saved data
    - **Eventually**? Depends on processing and other events.
    - Very useful for a system with concurrent processes where timing matters

SW/HW timing relationship: Synchronous vs. Asynchronous

	Input	Output
Synchronous	SW read instruction gets current value of input signal.	SW write instruction updates output signal immediately.
Asynchronous	HW signal event happens before software executes, is saved/buffered until used by SW. **Port Input Interrupt**: input event triggers later SW (ISR) execution. Buffered data: Which interrupt happened? **Input Time Capture**: input event triggers capture of time stamp (from timer's counter) to be read later by SW. Buffered data: time stamp, capture has happened.	SW write instruction updates buffer. HW is later triggered by an event to update output signal from buffer. **Output Update**: SW writes new output data to buffer, which HW uses when triggered by a timing reference

- What's decoupled?
  - Control (SW execution): HW input event causes SW instruction(s) to eventually execute. Example: input interrupts triggers interrupt service routine
  - Data decoupling: SW gets input HW value saved previously at event
  - Both: Input event triggers timer capture and requests interrupt. ISR reads buffered timer value.
- Synchronous: SW instruction (read in, write out) determines activity timing
  - Input: SW reads input's present value. SW determines the timing.
  - Output: SW write to output immediately changes output. SW determines the timing.
- Asynchronous: HW event determines activity timing
  - Input: HW signal event (e.g. rising edge) triggers software activity (ISR), captures timer counter value, etc.
  - Output: HW signal event (e.g. periodic timer event) triggers output update from buffered value

More Async Digital Signals



- PWM signal
  - What PWM is, why it is useful
  - Is async from software – want output to change at fixed time, loosening SW timing requirements
  - Absolute timing is critical, hard to do in SW, so offload time tracking, output generation to HW
  - Approach: sync to hardware timer, which tracks absolute time. Buffered count value, so output is updated in next HW timer period
  - Some overlap with WaveGen example
- Communication protocols, e.g. UART, SPI, I^2C, USB, modulated analog signals, …
  - UART vs. SPI

NC STATE UNIVERSITY

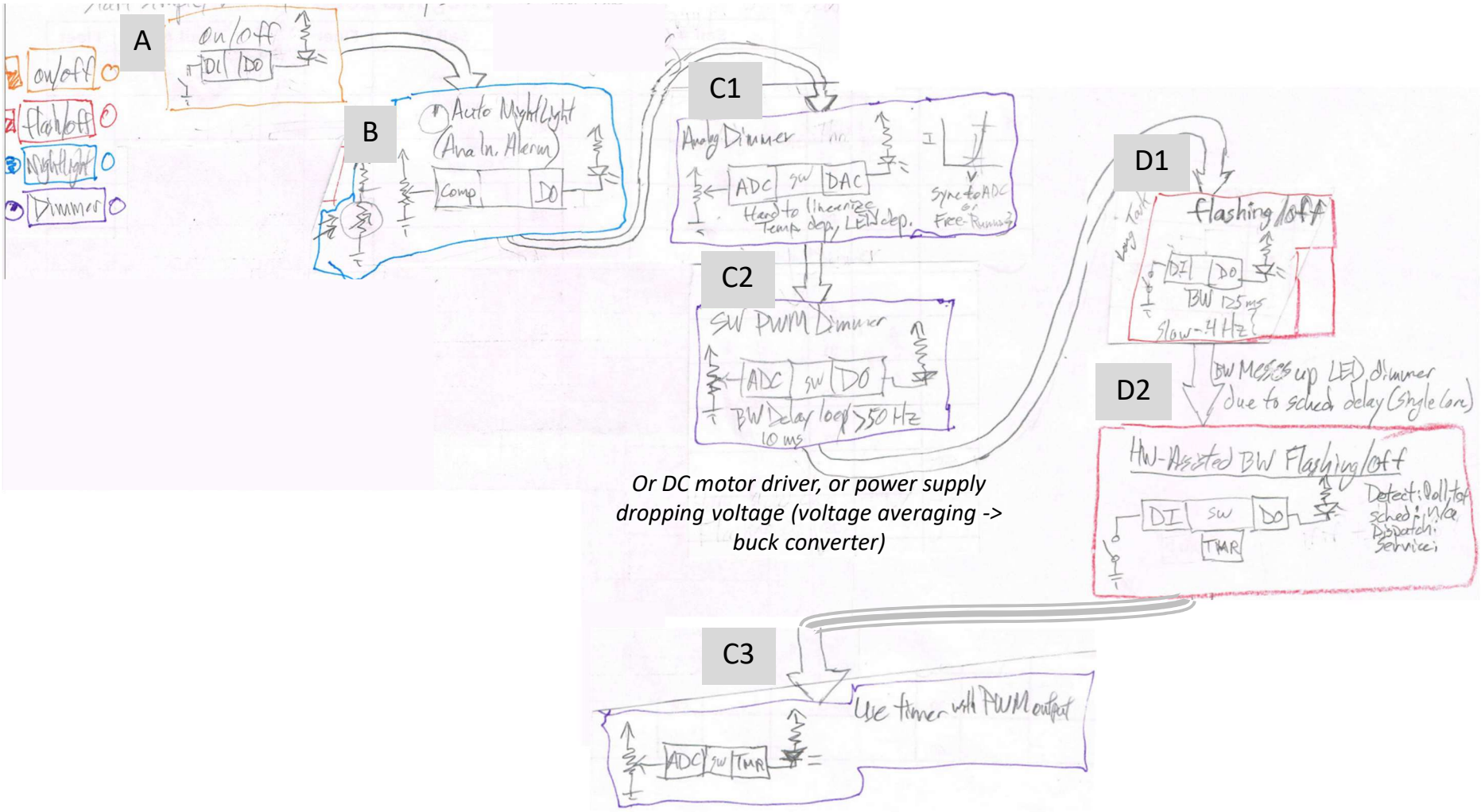
DESIGN EXAMPLE 1: BLINKY CONTROL PANEL

Blinky Control Panel Design Example Learning Objectives

- Basic concurrent system concepts
  - Processes: Software thread(s) on CPU core + memory, Hardware (state machines, digital and analog, etc.)
  - Concurrency: Possible to overlap/interleave start/execution/stop of different processes
  - Scheduling: Sharing resources (CPU core) to make process execute
  - Synchronization: At correct times/under right conditions, allow process (or part) to run, or prevent it from running
  - Communication: Sharing information between processes
- Simple control concepts
  - Motivating Examples with LEDs: on/off, nightlight, dimmer (I/V curve dependence on PVT), flasher
  - Use Feedback? Open vs. closed loop
  - When to Control: Event-driven, periodic, or both?
  - Control activities: Read/detect input, compute new output value, update output
  - Stability: concept, dependence on timing of input, output
- Basic interfacing with external devices
  - Simple signal types: Digital, analog, PWM
  - Use synchronous I/O. Specific SW instructions:
    - Trigger input read/sampling
    - Trigger output change (maybe with small, fixed delay)
  - Simple digital peripherals:
    - Inputs and outputs: Port/GPIO
  - Simple analog peripherals
    - Sampling and quantization concepts
    - Inputs: Comparator, ADC
    - Outpus: DAC
  - Timer Peripheral for PWM signal generation
- Sharing CPU among independent SW processes (scheduling)
  - Simple software scheduling
    - Merge conceptual processes into single SW process
    - Implicit sequential code vs. cyclic executive loop
  - Simple timing analysis
    - Source vs. object code, instruction set
    - System clock speed, instruction execution timing
  - Sources of timing variability
    - Dependence on data, control flow
    - Timing interference from SW processes sharing CPU
  - Could allocate CPU time better with better scheduler:
    - Multirate tasks
    - Better responsiveness from task prioritization and preemption
- Off-loading work from software to hardware
  - Synchronizing software to timer peripheral overflow
    - Polling, event detection, (scheduling/dispatching)
    - Stabilizes timing somewhat
  - Asynchronous Output
    - Timer peripheral handles entire cycle, eliminates SW synchronization
    - Stabilizing timing further (to design's PWM period)

NC STATE UNIVERSITY

Blinky Control Panel Design Evolution

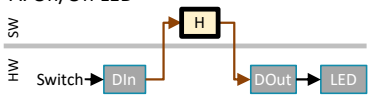


Or DC motor driver, or power supply dropping voltage (voltage averaging -> buck converter)

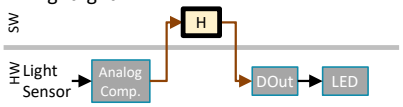
NC STATE UNIVERSITY

Design Evolution with Software and Hardware Components

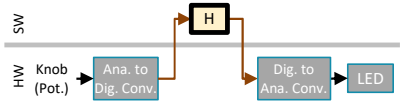
A. On/Off LED



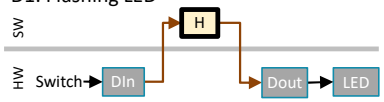
B. Nightlight LED



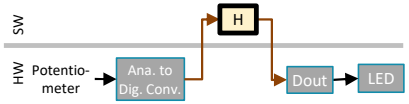
C1. Analog-Dimmable LED



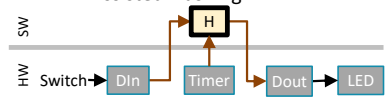
D1. Flashing LED



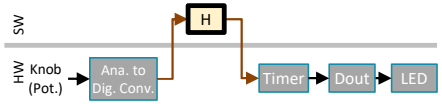
C2. SW PWM-Dimmable LED



D2. HW-Assisted Flashing LED



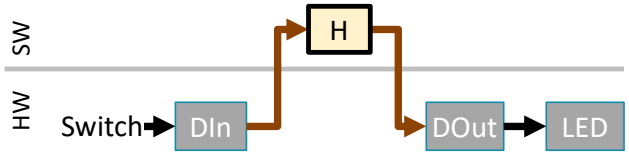
C3. HW PWM-Dimmable LED



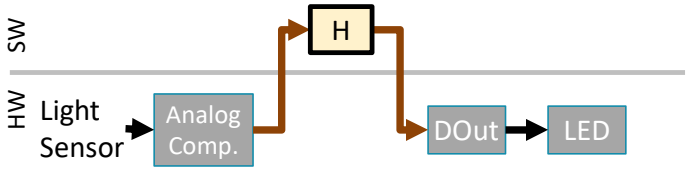
NC STATE UNIVERSITY

Design Evolution with Software and Hardware Components

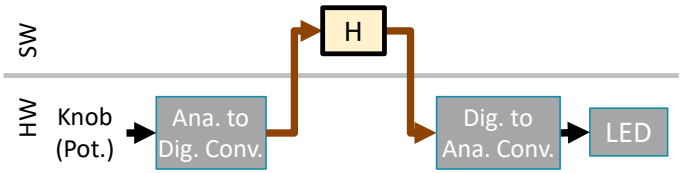
A. On/Off LED



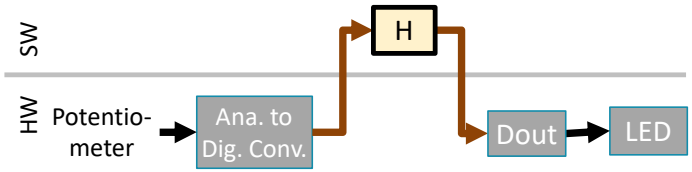
B. Nightlight LED



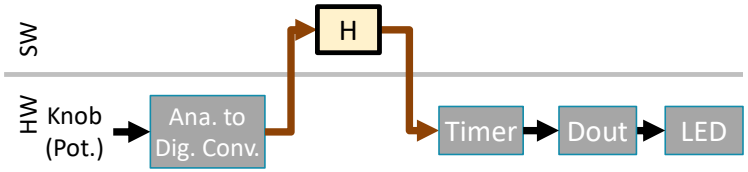
C1. Analog-Dimmable LED



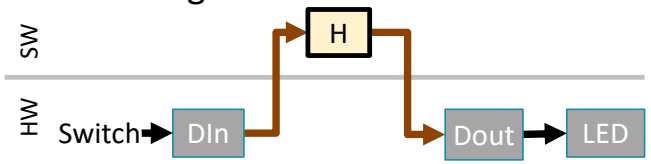
C2. SW PWM-Dimmable LED



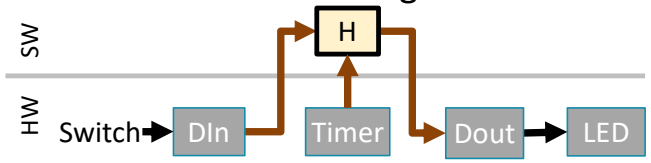
C3. HW PWM-Dimmable LED



D1. Flashing LED



D2. HW-Assisted Flashing LED



Switch	Off	On	Off	On
Code	Polling	Polling	Polling	Polling
LED	Off	On	Off	On

NC STATE UNIVERSITY

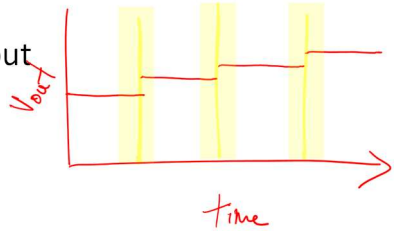
DESIGN EXAMPLE 2: WAVEFORM GENERATOR

Waveform Generation Design Example Learning Objectives

- Motivating Example:
  - Change DAC output signal periodically at specific times to generate accurate signal despite timing interference of other system software
- Stabilize output timing
  - Compensate or avoid timing interference from other processes
  - Poll HW timer to synchronize output
  - Convert to asynchronous output using hardware support
  - Improving timing stability
    - Handle events in better software or else hardware
    - Reduce number of events to handle in software
    - Progression: thread code per sample, interrupt code per sample, hardware event per sample & interrupt code per buffer refill
- Other benefits
  - Ease timing requirements for software to refill buffer
  - Reduce CPU overhead per sample
- Concurrent system concepts
  - Scheduling: Using interrupts to schedule SW (ISRs) on CPU
  - Synchronization: Move from Sync. output to Async. output, leveraging HW sync. signals (IRQ, DRQ, event)
    - Revisit SW sync to timer overflow with polling, then convert to interrupt
    - SW to HW: Must trigger code to
      - update DAC output,
      - refill periph. HW buffer (single, FIFO)
      - refill correct double buffer in memory
      - refill correct buffer in memory: ISR for urgent buffer, thread code for non-urgent buffer
    - HW to HW:
      - timer triggers DAC & buffer updates
      - timer triggers DMA transfer
  - Communication:
    - SW storing output data in DAC or buffer (peripheral or memory)
    - Select which double buffer to refill, which to reload from
- Off-loading work from software to hardware
  - Asynchronous Output
    - HW+SW: Timer peripheral generates interrupt request, interrupt handler/ISR updates output
    - HW+HW: Timer peripheral triggers data transfer from HW buffer/FIFO to DAC. Buffer may generate interrupt requesting refill.
    - HW+HW: Timer peripheral triggers DMA to transfer data from memory buffer to DAC. DMA triggers interrupt when done with set of transfers.

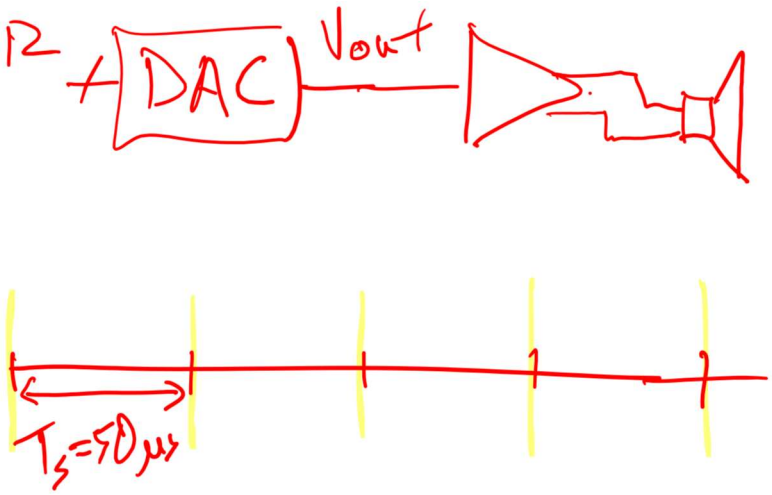
Waveform Generator: Update Output Signal at Correct Times

- Some output signals may need to be updated at specific times
  - Audio signal reconstruction needs **periodic updates**: change output every T_{Sample}
  - Controlling a switch-mode power converter or motor driver needs updates synchronized to system phase
- How does an early or late update affect the system performance?
  - Absolute, hard deadlines: updating output without meeting timing requirements is useless
    - Deadline: do it before $T_{Deadline}$
    - Window: do it before T_{WOpen} and before T_{WClose}
  - Soft deadlines: ok to be earlier or late, or miss a deadline occasionally. Impact depends on timing error.
  - Do all valid update times give the same performance, or are some better than others? Value function indicates impact of timing error. Flat top vs sloping.
  - As window gets narrower, becomes harder create system which meets timing requirements
  - Where are the sweet spots for timing windows?
    - Depends on: instruction execution time, code to do the work, interrupts (response latency, higher-priority, masking, blocking), scheduler (cooperative/preemptive, context switching latency, higher-priority, scheduler locking, blocking),
- Timing controllability and determinism depend on system implementation and interference by other parts of system
  - Hardware:
  - Software: how concurrency is handled. Synchronization, event detection, scheduling approach (if sharing CPU), event handling (may include more synchronization and communication)
- Hard to stabilize timing for software processes
  - Translation (compilation) obscures timing of source code.
    - Source code is translated to executable machine code
    - Can measure and observe machine code, not source code
  - Sharing CPU with other processes and handlers (scheduling) can delay or preempt code generating output signals
- Use hardware (and software) to help stabilize output timing of DAC (Digital to Analog Converter)



Waveform Generator Design Details

- Want to update DAC output every 50 us for a 20 kHz update rate
  - DAC signal amplified to drive speaker
- Timing analysis approach - Vulnerabilities?
  - What kinds of events and over what time periods can affect the output update time?
  - Events
    - Unbuffered solutions: each sample
    - Buffered solutions: each buffer refill
- Solutions
  - Use hardware to help (or even replace) software doing synchronization, scheduling, or work.
    - Synchronization: determining when to update output
    - Scheduling: selecting code to run
    - Work: updating output
  - Buffer data to loosen (simplify) software timing requirements



WG Timing Stability Overview 1: What and Wh

Refer to diagrams on next page

IC STATE UNIVERSITY

Version. Task/Thread, ISR, HW

Process Scheduling, Synchronization & Communication Highlights

Performance, behavior.

Problem(s).

Changes to solve problem(s).

Output timing bad: very unstable and hard to predict (non-deterministic).
*Timing is **always vulnerable** to **variations** in time to calculate next sample, **interference** from other software (in main loop, other processes, ISRs).*
*Timing errors **accumulate**.*
***Greedy**, doesn't share CPU.*

W1. Thread loop calculates new output sample, busy-waits for fixed number of instructions, writes sample to DAC,
DAC updates output immediately

Main thread loop has no synchronization, just schedules output updates based on fixed number of instructions.

Add HW timer (tracks time much better than SW)

B. Thread polls Hardware Timer
Thread loop calculates new output sample, polls timer and blocks until reaching target time, then writes to DAC, calculates next target time,
DAC updates output immediately

Main loop synchronizes to hardware target time (counter value) before updating output.

Output timing better: Tolerates more calculation variability and interference (up to slack time between samples), and errors don't accumulate.
Greedy, doesn't share CPU. Timing is still somewhat vulnerable to other software once per sample (between exiting sync loop and updating DAC).

Add HW timer ISR

C. Hardware Timer triggers ISR
Timer ISR writes sample data to DAC, calculates next sample,
DAC updates output immediately

Timer & Interrupt system sync output update to hardware target time. All of Main loop time available to do other application work.

Output timing: Even better.
Still vulnerable to other ISRs and interrupt locking once per sample (between timer IRQ and updating DAC).

Add 1-deep DAC input buffer

D. Use Single-Entry DAC Input Buffer
Timer advances buffer data to DAC,
Timer ISR calculates next sample, writes it to buffer

Timer & DAC sync output update.
Timer & Int. Sys. sync calc./save next sample to when buffer is free. All of Main loop time available to do other application work.

Interrupt overhead for each sample wastes CPU time

Add N-deep DAC input buffer with low warning ISR (W samples left)

E2. Timer advances buffer data to DAC.
Buffer low ISR writes next batch of data to buffer

Timer & DAC sync output update.
Timer & Int. Sys. sync start of refilling buffer to when buffer is nearly empty.
All of Main loop time available to do other application work.
Synchronization issue with buffer: Could ISR possibly overwrite unused samples in buffer? Depends on timing relationships between sample rate, refill rate, delays from other software processes. If so, how to handle it? (see deferring urgent work design point.)

Completely stable output update timing (no interference from main (just int. masking). Batch processing to refill buffer cuts overhead.
Deadline to refill first buffer entry extended to $T_{Sample}*(W+1)$.
Must manage buffer access: what if refill catches up to timer-driven reads?

Add HW timer, DMA with ISR, sample buffer in memory

W6. Timer triggers each buffer->DAC DMA transfer,
DMA ISR runs after last transfer to reload buffer with new data

Timer & DMA sync output update.
DMA & Int. Sys. sync start of buffer refill after last transfer (buffer empty).

Stable output update timing. No interference from main code (just int. sys. & masking).
Refilling buffer with batch processing cuts overhead.
1.Tight Deadline: ISR must write first new sample to buffer within T_{Sample} or else old data will be reused.
2. DMA ISR updates buffer with N samples, delays other processing

Use double buffering, split into two buffers (each N/2 entries) to ease first sample's deadline and cut ISR duration in half.

W7. Timer triggers each buffer->DAC DMA transfer, DMA ISR runs after last transfer to switch buffers and reload old buffer with new data

Timer & DMA sync output update.
DMA & Int. Sys. sync buffer switch and start of old buffer refill after last transfer.

Stable output update timing (no interference from main code (just int. masking). Batch processing to refill buffer cuts overhead.
Deadline to refill first buffer entry extended to $T_{Sample}*(N/2+1)$.
DMA ISR updates buffer with N/2 samples, delays other processing.

Split buffer refill work. Do urgent work (U samples) in ISR, move non-urgent work (last N-U samples) to task

W8. Timer triggers each buffer->DAC DMA transfer, DMA ISR writes urgent data (U samples) to buffer and releases task to write rest of data to buffer

Timer & DMA sync output update.
DMA & Int. Sys. sync start of urgent buffer refill after last transfer.
Main loop syncs refill of rest of buffer to signal from ISR.

Stable output update timing (no interference from main code (just int. masking). Batch processing to refill buffer cuts overhead.
Deadline to refill first buffer entry extended to $T_{Sample}*(N/2+1)$.
DMA ISR shorter, only updates U samples.
Task must start to update buffer (write sample U+1) within $T_{Sample}*(U+1)$.

WG Timing Stability Overview: What and Why

Refer to diagrams on next page

Version. Task/Thread, ISR, HW

Process Scheduling, Synchronization & Communication Highlights

Performance, behavior.
Problem(s).
Changes to solve problem(s).

W1. Thread loop calculates new output sample, busy-waits for fixed number of instructions, writes sample to DAC,
DAC updates output immediately

Main thread loop has no synchronization, just schedules output updates based on fixed number of instructions.

*Output timing bad: very unstable and hard to predict (non-deterministic). Timing is **always vulnerable** to **variations** in time to calculate next sample, **interference** from other software (in main loop, other processes, ISRs). Timing errors **accumulate**.*
***Greedy**, doesn't share CPU.*

Add HW timer (tracks time much better than SW)

W2. Thread polls Hardware Timer
Thread loop calculates new output sample,
polls timer and blocks until reaching target time,
then writes to DAC, calculates next target time,
DAC updates output immediately

Main loop synchronizes to hardware target time (counter value) before updating output.

Output timing better: Tolerates more calculation variability and interference (up to slack time between samples), and errors don't accumulate.
Greedy, doesn't share CPU. Timing is still somewhat vulnerable to other software once per sample (between exiting sync loop and updating DAC).

Add HW timer ISR

W3. Hardware Timer triggers ISR
Timer ISR writes sample data to DAC, calculates next sample,
DAC updates output immediately

Timer & Interrupt system sync output update to hardware target time.
All of Main loop time available to do other application work.

Output timing: Even better.
Still vulnerable to other ISRs and interrupt locking once per sample (between timer IRQ and updating DAC).

Add 1-deep DAC input buffer

W4. Use Single-Entry DAC Input Buffer
Timer advances buffer data to DAC,
Timer ISR calculates next sample, writes it to buffer

Timer & DAC sync output update.
Timer & Int. Sys.sync calc./save next sample to when buffer is free.
All of Main loop time available to do other application work.

Interrupt overhead for each sample wastes CPU time

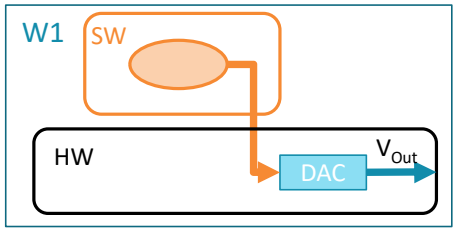
Add N-deep DAC input buffer with low warning ISR (W samples left)

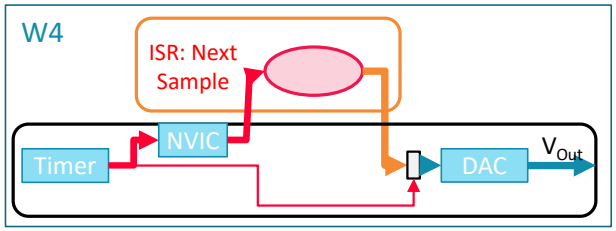
W5. Timer advances buffer data to DAC.
Buffer low ISR writes next batch of data to buffer

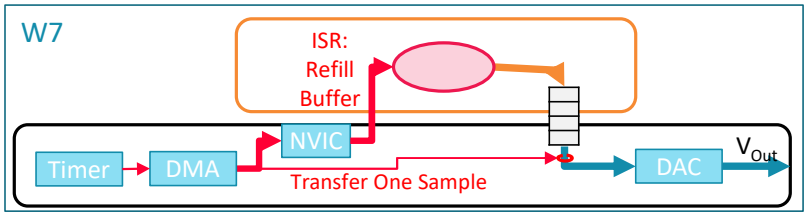
Timer & DAC sync output update.
Timer & Int. Sys. sync start of refilling buffer to when buffer is nearly empty.
All of Main loop time available to do other application work.
Synchronization issue with buffer: Could ISR possibly overwrite unused samples in buffer? Depends on timing relationships between sample rate, refill rate, delays from other software processes. If so, how to handle it? (see deferring urgent work design point.)

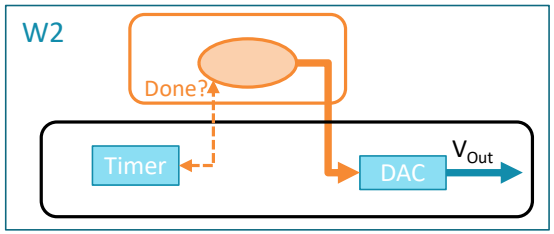
Completely stable output update timing (no interference from main (just int. masking). Batch processing to refill buffer cuts overhead.
Deadline to refill first buffer entry extended to T_{Sample}(W+1).*
Must manage buffer access: what if refill catches up to timer-driven reads?

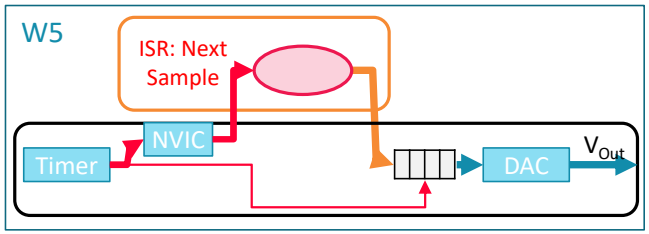
Waveform Generator Design Evolution with Software and Hardware Components: Want Output Updates with Stable Timing

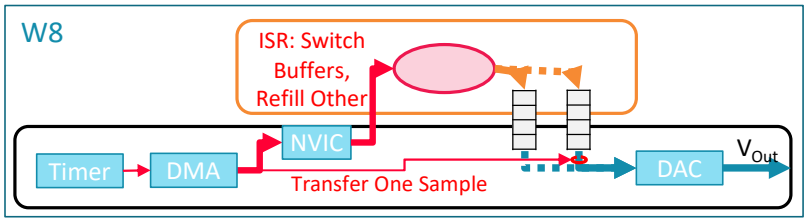


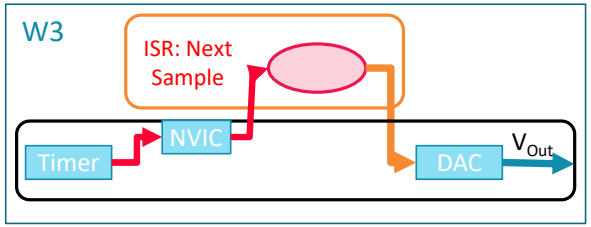


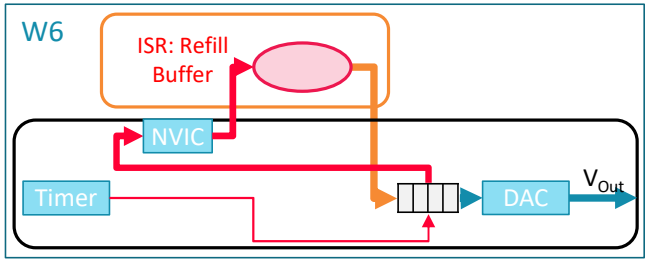


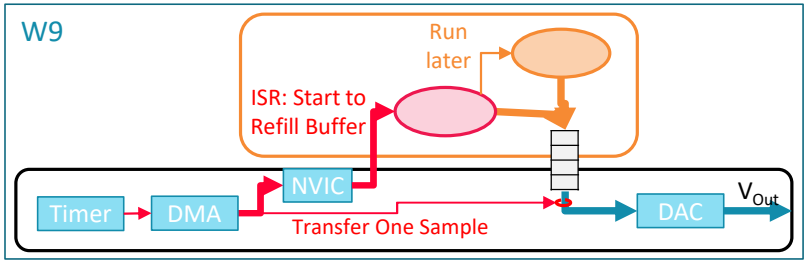












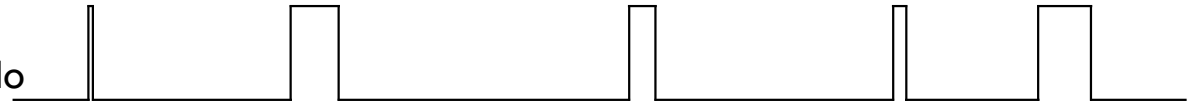
DESIGN EXAMPLE 3: SCOPE (OSCILLOSCOPE)

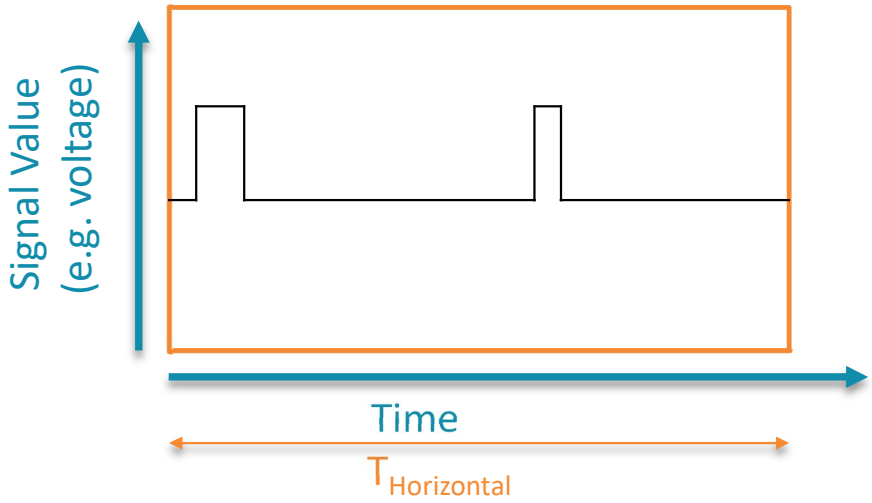
Scope Design Example Learning Objectives

- *Basic concurrent system concepts*
  - *Processes: Software thread(s) on CPU core + memory, Hardware (state machines, digital and analog, etc.)*
  - *Concurrency: Possible to overlap/interleave start/execution/stop of different processes*
  - *Scheduling: Sharing resources (CPU core) to make process execute*
  - *Synchronization: At correct times/under right conditions, allow process (or part) to run, or prevent it from running*
  - *Communication: Sharing information between processes*
- *Simple control concepts*
  - *Motivating Examples with LEDs: on/off, nightlight, dimmer (I/V curve dependence on PVT), flasher*
  - *Use Feedback? Open vs. closed loop*
  - *When to Control: Event-driven, periodic, or both?*
  - *Control activities: Read/detect input, compute new output value, update output*
  - *Stability: concept, dependence on timing of input, output*
- *Basic interfacing with external devices*
  - *Simple signal types: Digital, analog, PWM*
  - *Use synchronous I/O. Specific SW instructions:*
    - *Trigger input read/sampling*
    - *Trigger output change (maybe with small, fixed delay)*
  - *Simple digital peripherals:*
    - *Inputs and outputs: Port/GPIO*
  - *Simple analog peripherals*
    - *Sampling and quantization concepts*
    - *Inputs: Comparator, ADC*
    - *Outpus: DAC*
  - *Timer Peripheral for PWM signal generation*
- *Sharing CPU among independent SW processes (scheduling)*
  - *Simple software scheduling*
    - *Merge conceptual processes into single SW process*
    - *Implicit sequential code vs. cyclic executive loop*
  - *Simple timing analysis*
    - *Source vs. object code, instruction set*
    - *System clock speed, instruction execution timing*
  - *Sources of timing variability*
    - *Dependence on data, control flow*
    - *Timing interference from SW processes sharing CPU*
  - *Could allocate CPU time better with better scheduler:*
    - *Multirate tasks*
    - *Better responsiveness from task prioritization and preemption*
- *Off-loading work from software to hardware*
  - *Synchronizing software to timer peripheral overflow*
    - *Polling, event detection, (scheduling/dispatching)*
    - *Stabilizes timing somewhat*
  - *Asynchronous Output*
    - *Timer peripheral handles entire cycle, eliminates SW synchronization*
    - *Stabilizing timing further (to design's PWM period)*

Scope: Detect Input Trigger, Sample and Display Data

- Input signal
  - Start with simple one-bit digital signal (do analog later)
  - Pulses have irregular start times, changing pulse widths
- Viewing 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





NC STATE UNIVERSITY

Scope Example Processes

Ver.	**Software**				**Hardware**		
	Threads	ISRs					
	Main	Port	Timer	DMA	Port	Timer	DMA
1	Sample, Display						
2	Detect Trigger Condition, Sample, Display						
3	Detect Trigger Condition, Sample, Update Display, Erase Display						
4, 5	Erase Display	Take Sample, Update Display			Detect Trigger Condition		
6	Erase Display, Update Display	Take Sample			Detect Trigger Condition		
7	Erase Display, Update Display		Take Sample		Detect Trigger Condition	Schedule Sample	
8	Erase Display, Update Display				Detect Trigger Condition	Schedule Sample	Take Sample

NC STATE UNIVERSITY

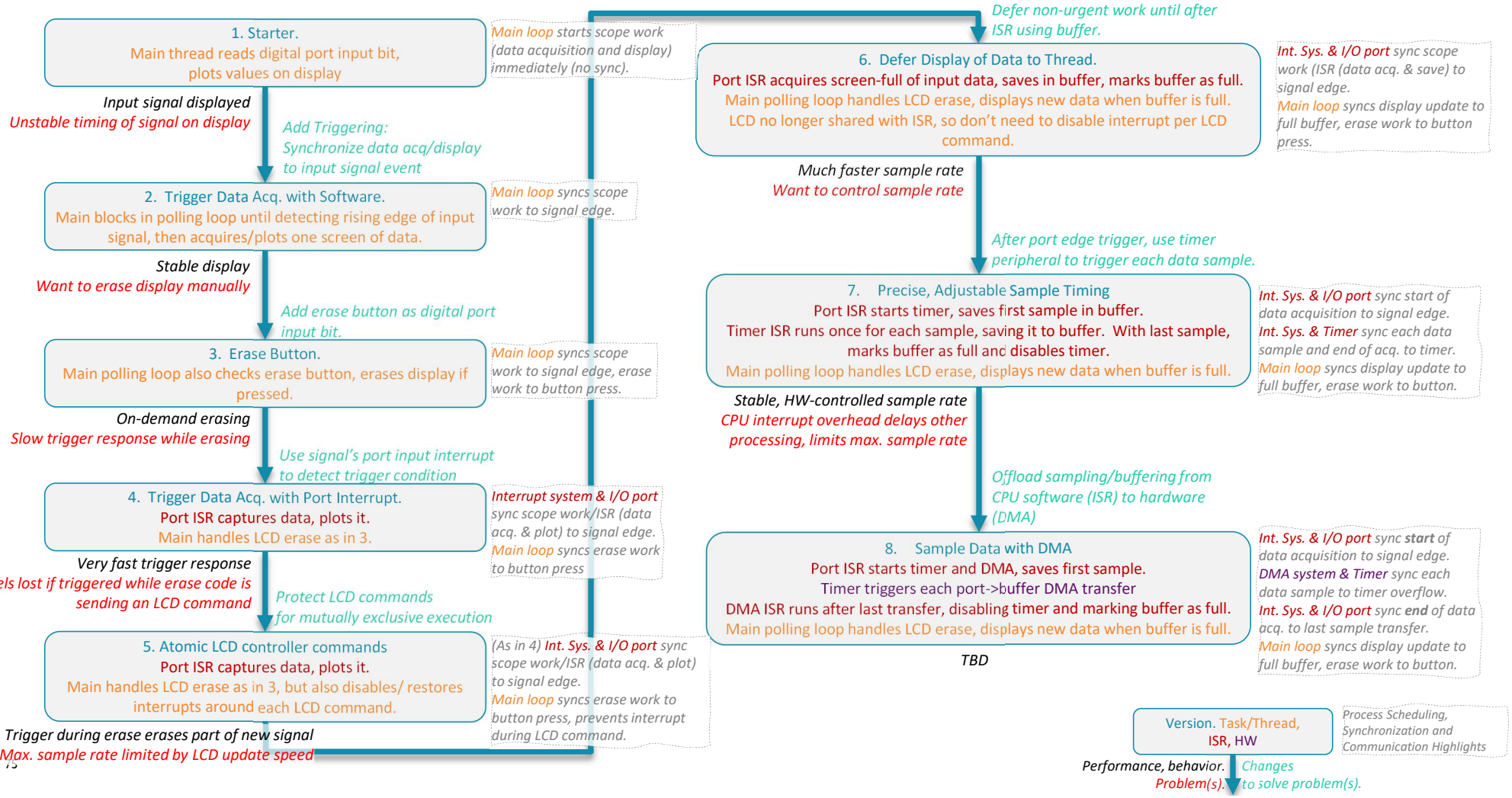
Scope Example Processes

		Detect Trigger Condition	Schedule Sample	Take Sample	Update Display	Erase Display
I	SW – Thread	Sample, Display				
	HW	Detect Trigger Condition, Sample, Display				

Responsiveness Overview: What and Why

Refer to diagrams on next 3 pages

NC STATE UNIVERSITY



A Different Diagram Syntax

Boxes: Processes, code, and data objects

Handler Process (thread or interrupt level)

Synchronization Code (includes scheduling, dispatching at this level)

Interrupt System (includes scheduling, dispatching at this level)

Synchronization data object

Data buffer in architecturally-visible memory,available to software and hardware

Arrows: Data and Synchronization (Control) Flows

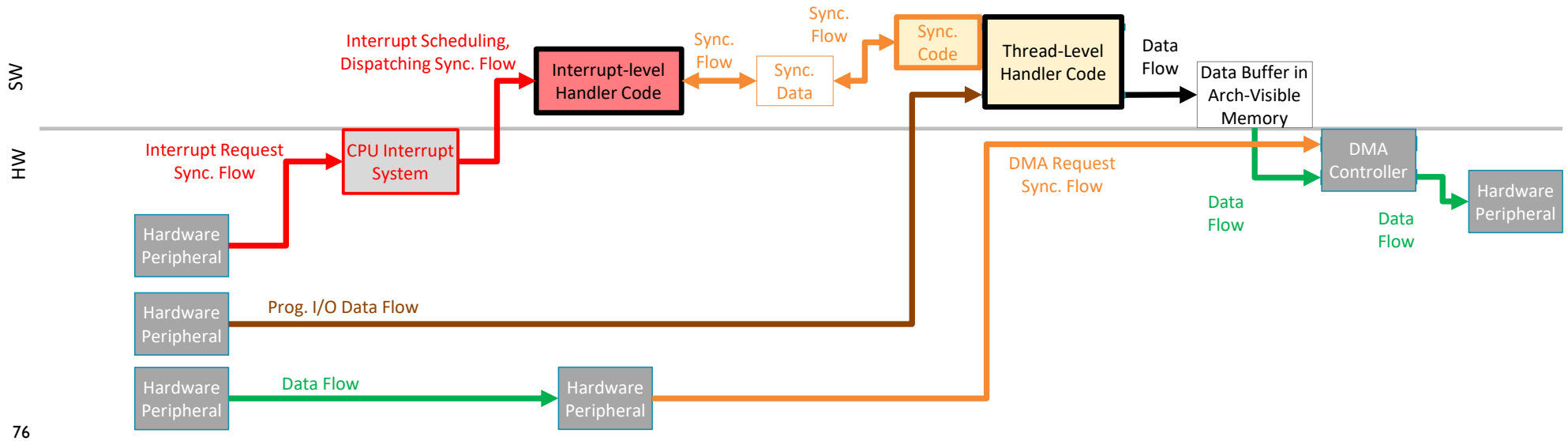
Software activity data flow

Programmed I/O data flow (software-driven)

Hardware activity data flow (hardware-driven)

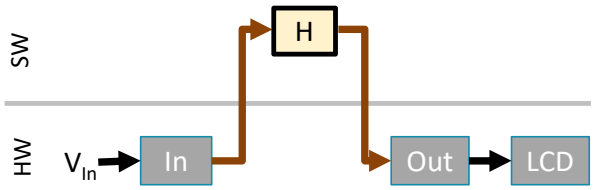
Synchronization flow

Interrupt flow

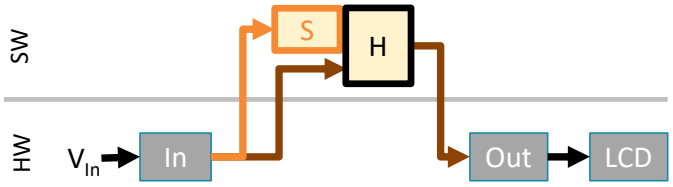


Design Evolution with Software and Hardware Components

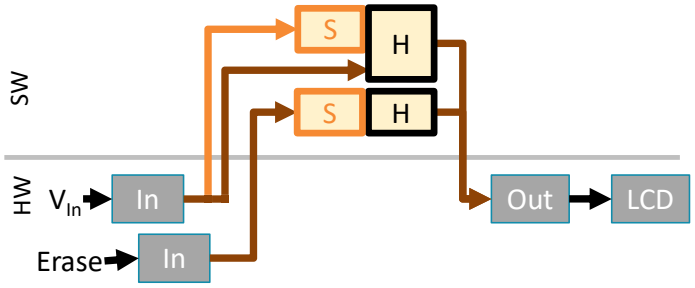
1. Basic



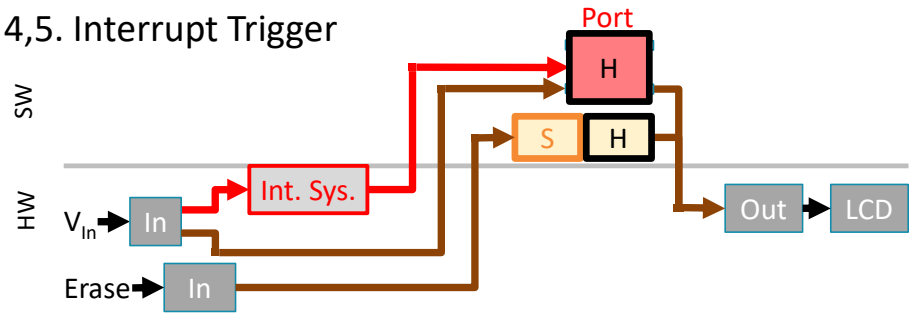
2. Polling Trigger



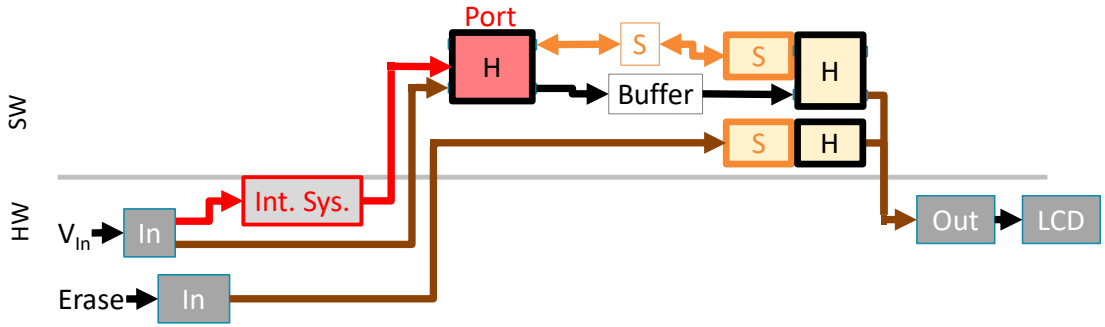
3. Erase Button



4,5. Interrupt Trigger

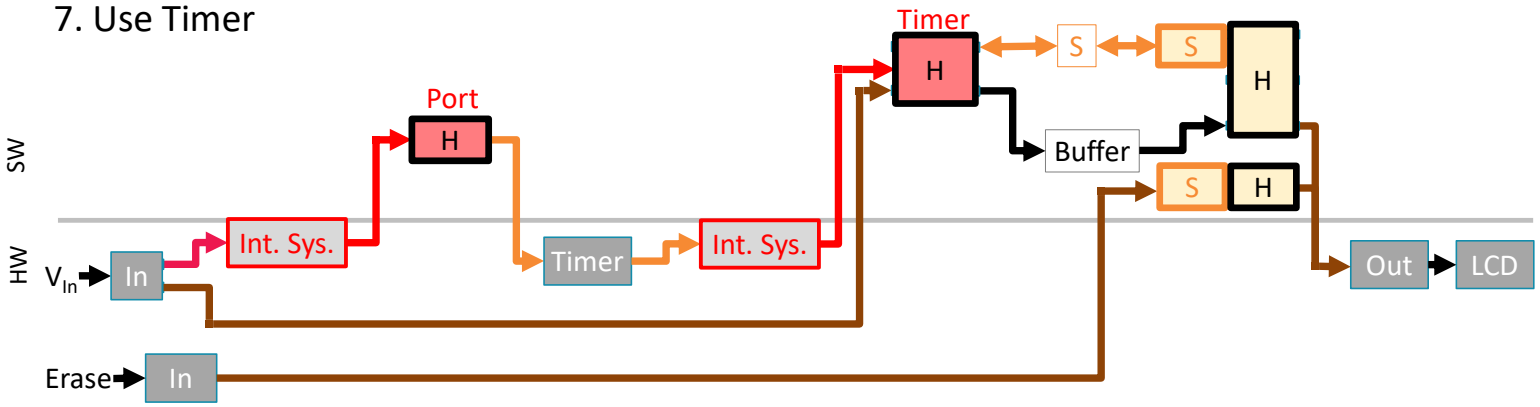


6. Defer LCD Updates

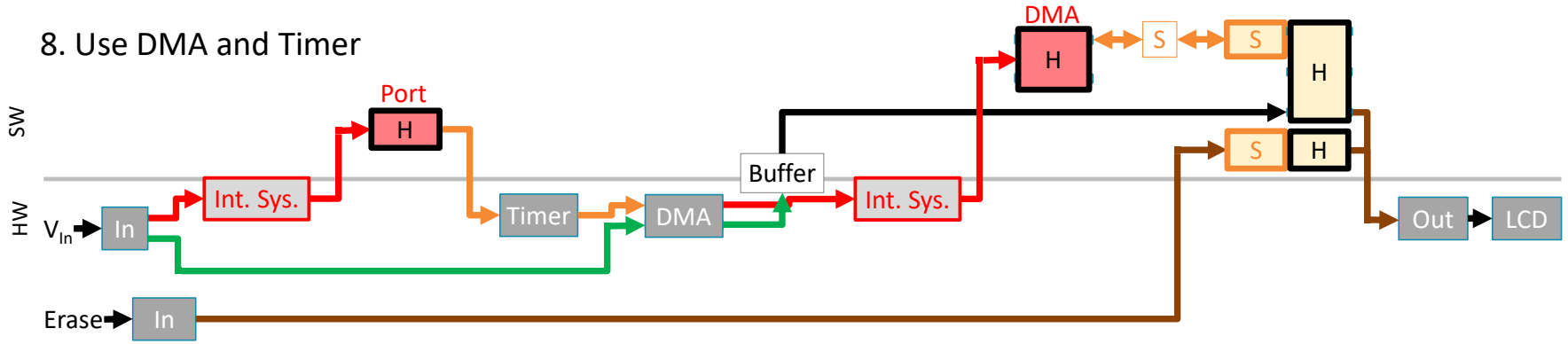


Design Evolution with Software and Hardware Components

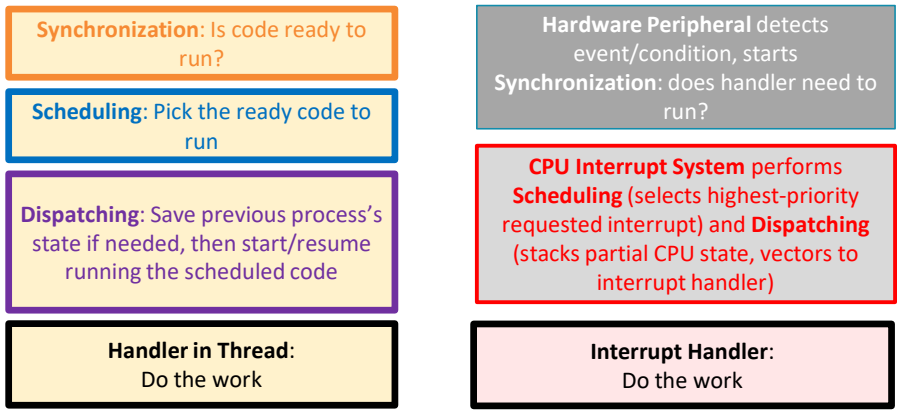
7. Use Timer



8. Use DMA and Timer



Can Zoom into Source Code Details of Sync., Sched., etc.



- Lightweight, responsive schedulers essential for many embedded systems.
  - Interrupt system (CPU's built-in scheduler) is foundation.
  - Synchronization, scheduling, dispatching/context switching, communication may be implemented in hardware, application software, OS software
- Syntax supports range of approaches: from bare-metal through preemptive OS. Examples on right.

Blocking Synchronization with Programmed I/O

```
Busy Wait Process
…
// 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);
}
```

Interrupts + RTOS

```
ISR_1
for (n=0; n<NS; n++) {
    r = ADC->Result;
}
osEventFlagSet(triggered);

Process B
while (1) {
      osEventFlagWait(triggered);
      x = 0;
      for (n=0; n<NS; n++) {
        y = scale(DataQ[n]);
        LCD_Plot(x++,y);
      }
}
```

Interrupts + Run-to-Completion Tasks in Simple Co-op. Scheduler

```
Scheduler Process
…
while (1) {
    for (i=0; i<NT; i++) {
        if (release_requested[i] > 0) {
            release_requested[i]--;
            task_pointer[i](); // dispatch
            break;
    }
}

ISR_timer_tick
    for (i=0; i<NT; i++) {
        if (is_waiting[i]) {
            if (--delay_to_release[i] == 0) {
                release_requested[i]++;
            }
        }
    }

ISR_2
    do urgent work;
    release_requested[Deferred_Handler_2]
```

NC STATE UNIVERSITY

RAW, UNUSED, OLD, LEFTOVER SLIDES

Hardware Support for Multicore Synchronization

- Example System: Cypress PSoC6 MCU (CY8C62x8/A)
- IPC Peripheral
  - 16 IPC channels: hardware support for atomic acquisition. SW write to indicate notify, release can generate event information for any/all IPC interrupt structures. Two data registers, status register.
  - 16 IPC interrupt structures: generate interrupt requests based on monitored notify, release events
- How used
  - Lock: Use IPC channel
  - Message passing: Use IPC channel and IPC interrupt.
    - Sender uses IPC channel to lock access to data registers, notify of send triggering interrupt for receiver, which releases channel after reception and can notify sender with interrupt

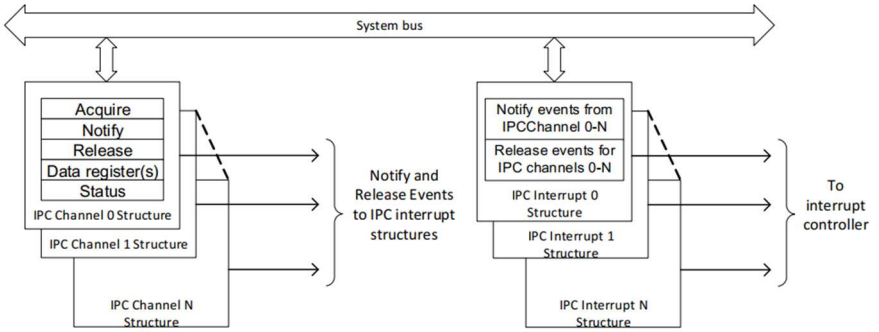


Figure 6-8. Half Duplex with Independent Event Handling

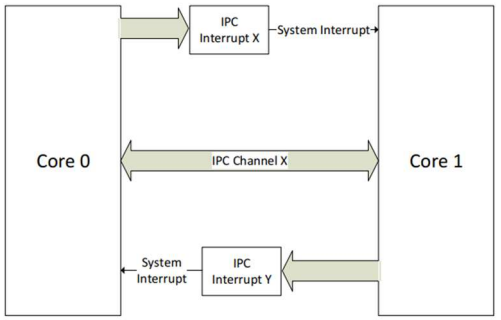
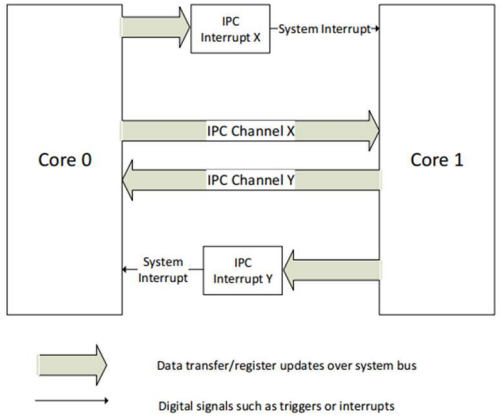
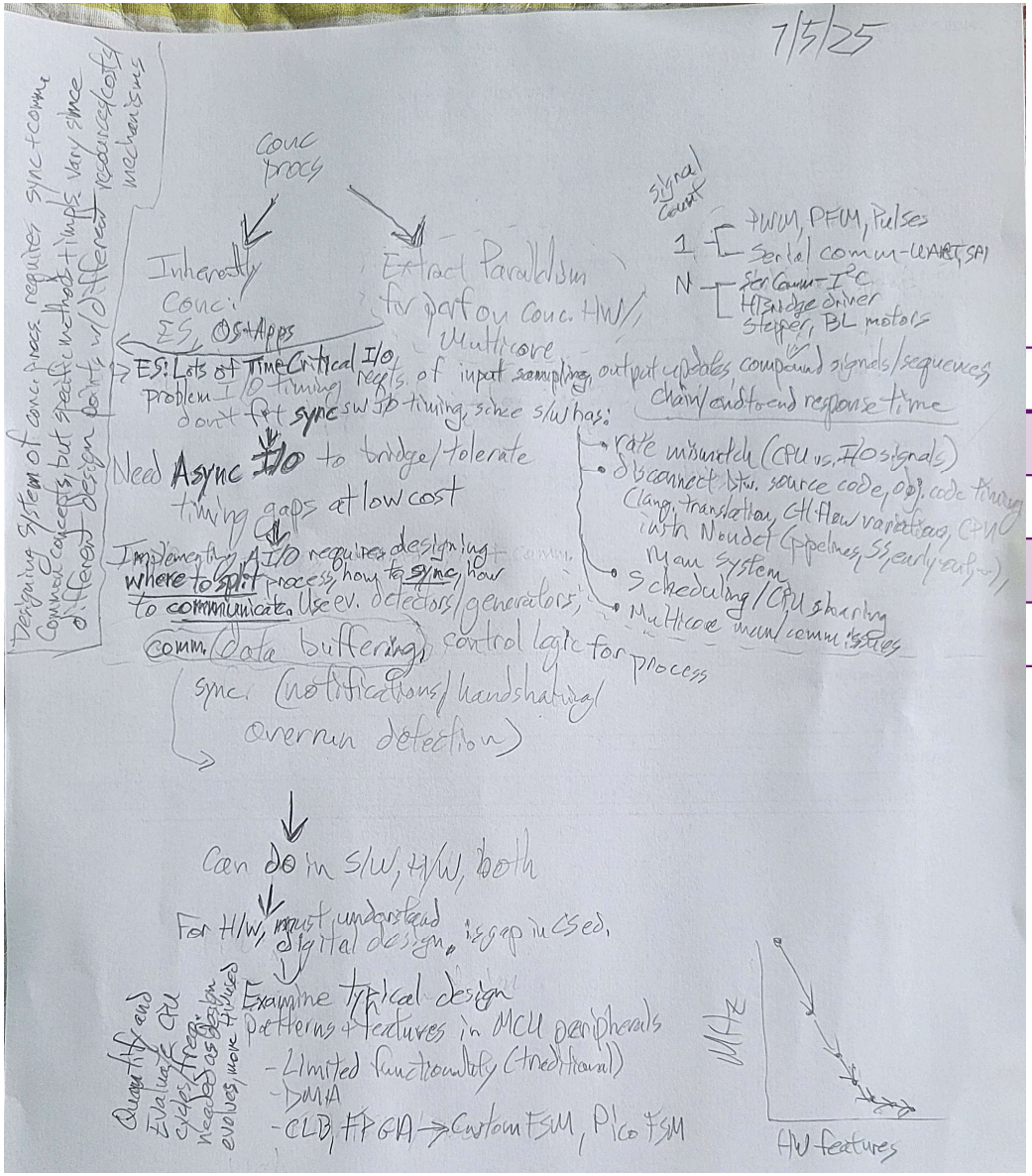


Figure 6-7. Full Duplex IPC for Dual Core Communication

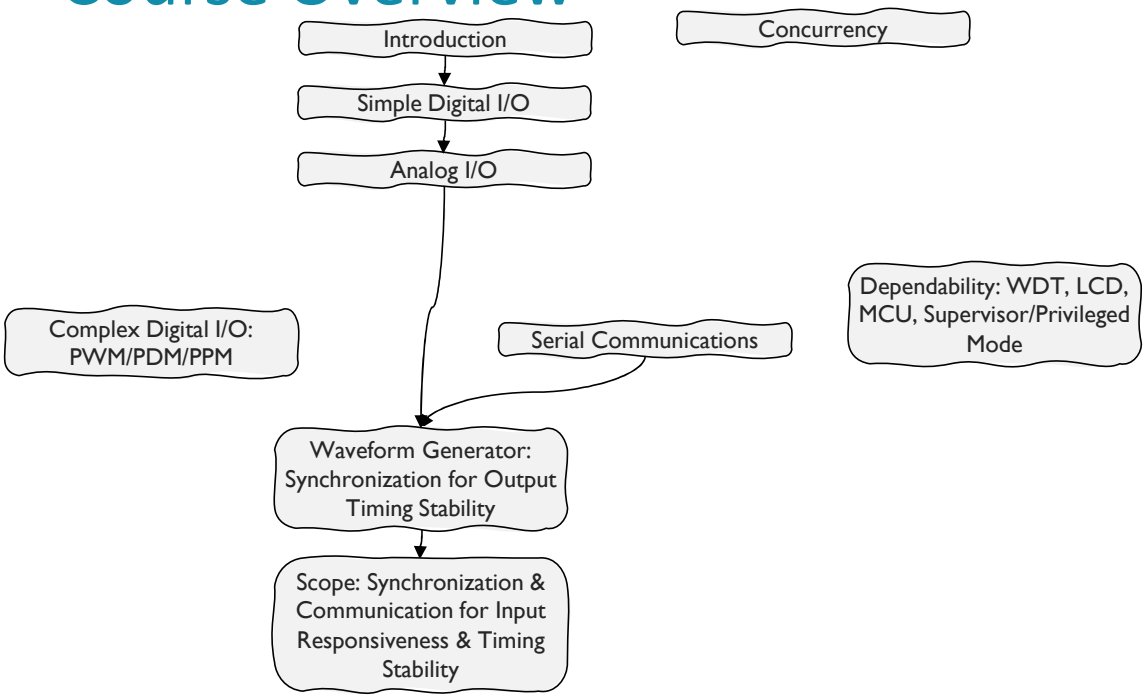


Designs

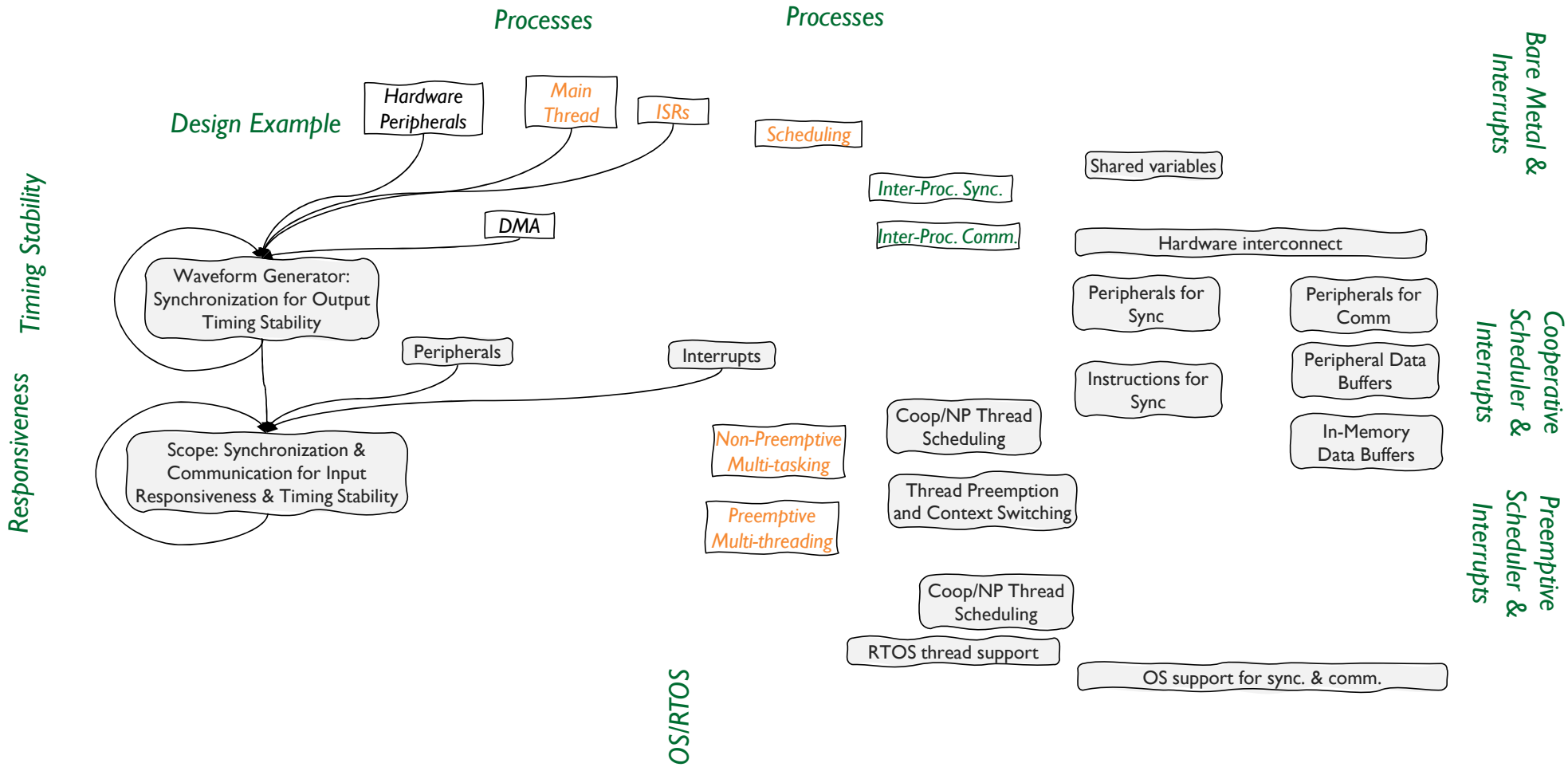
- TBD


7/5/25
Designing system of conc. procs requires sync + comm
Common concepts, but specific methods + APIs vary since different design tools w/ different resources/costs/mechanisms
Conc procs
Inherently Conc: ISR, OS+Apps
Extract Parallelism for perf on Conc. HW/ Multicore
signal count
1
N
PWM, PFM, Pulses
Serial comm - UART, SPI
Ser Comm - I2C
H-Bridge driver
Stepper, BL motors
ES: Lots of Time Critical I/O
Problem: I/O timing reqts of input sampling, output updates, command signals/sequences
don't fit sync SW I/O timing, since S/W has:
chain/end to end response time
Need Async I/O to bridge/tolerate timing gaps at low cost
rate mismatch (CPU vs. I/O signals)
Disconnect betw. source code, obj. code timing
Clang, translation, CH flow variations, CPU
with Nondet (pipelines, SS, early exit)
Main system
Scheduling / CPU sharing
Multicore mem/comm issues
Implementing A/IO requires designing where to split process, how to sync, how to communicate. Use ev. detectors/generators,
comm. (data buffering, control logic for process
sync. (notifications/handshaking
overrun detection)
Can do in S/W, H/W, both
For H/W, input understand digital design, i.e. gap in CS ed.
Examine typical design patterns + features in MCU peripherals
- Limited functionality (tried + true)
- DMA
- CLB, FPGA → Custom FSM, PIco FSM
Quantify and Evaluate CPU cycles, freq needed as design evolves, more H/W used
MHz
HW features

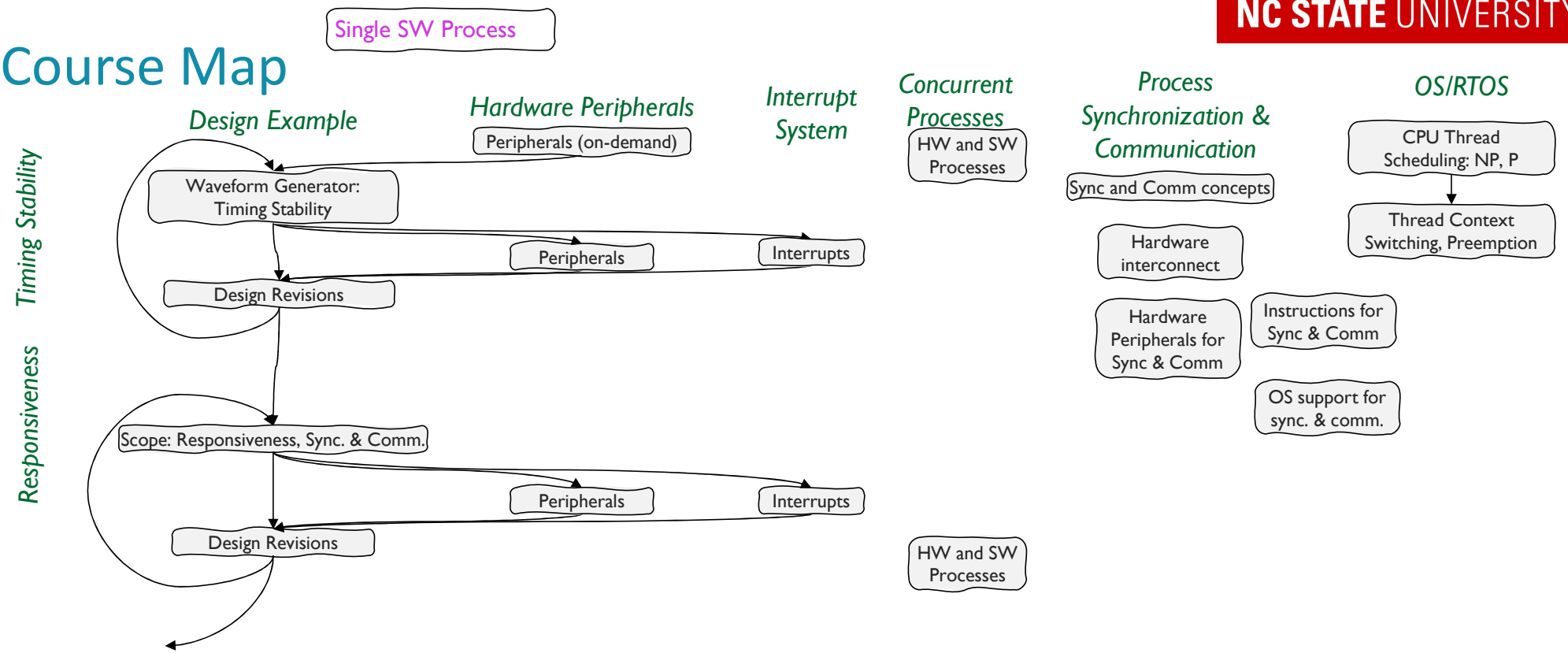
Course Overview



Top Level Course Map



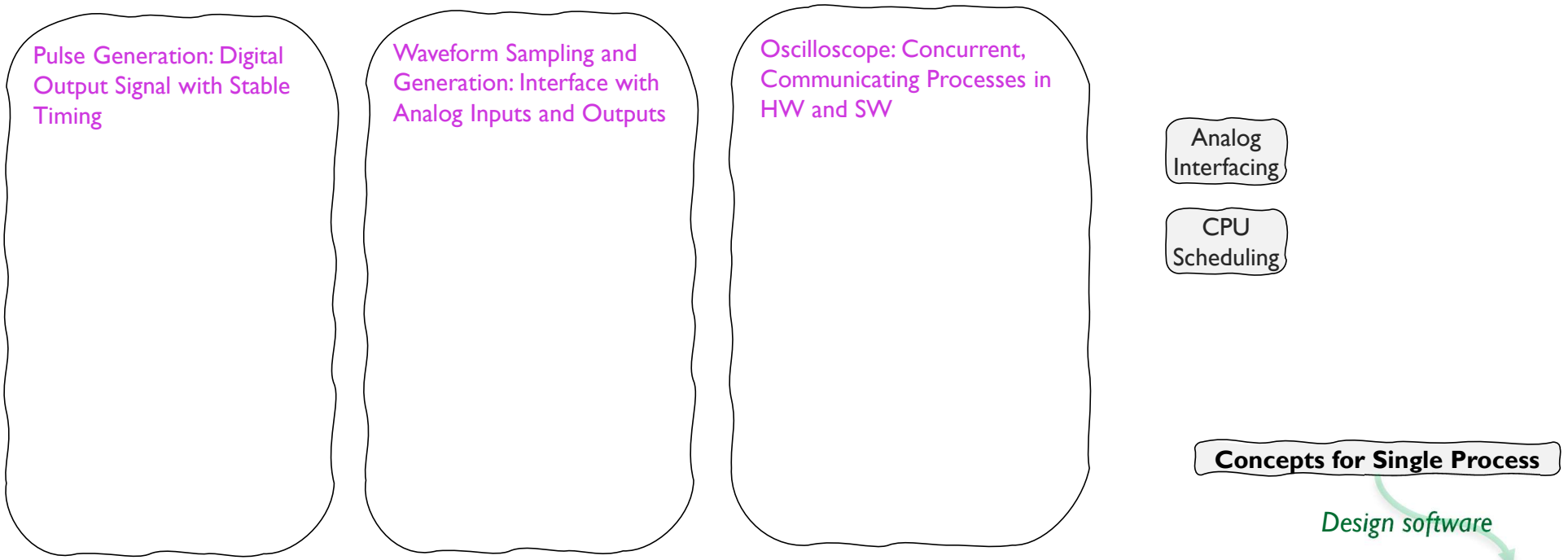
Course Map



Waveform Sampling and Generation: Interface with Analog Inputs and Outputs

Oscilloscope: Concurrent, Communicating Processes in HW and SW

Goal-Oriented Map



Course Map

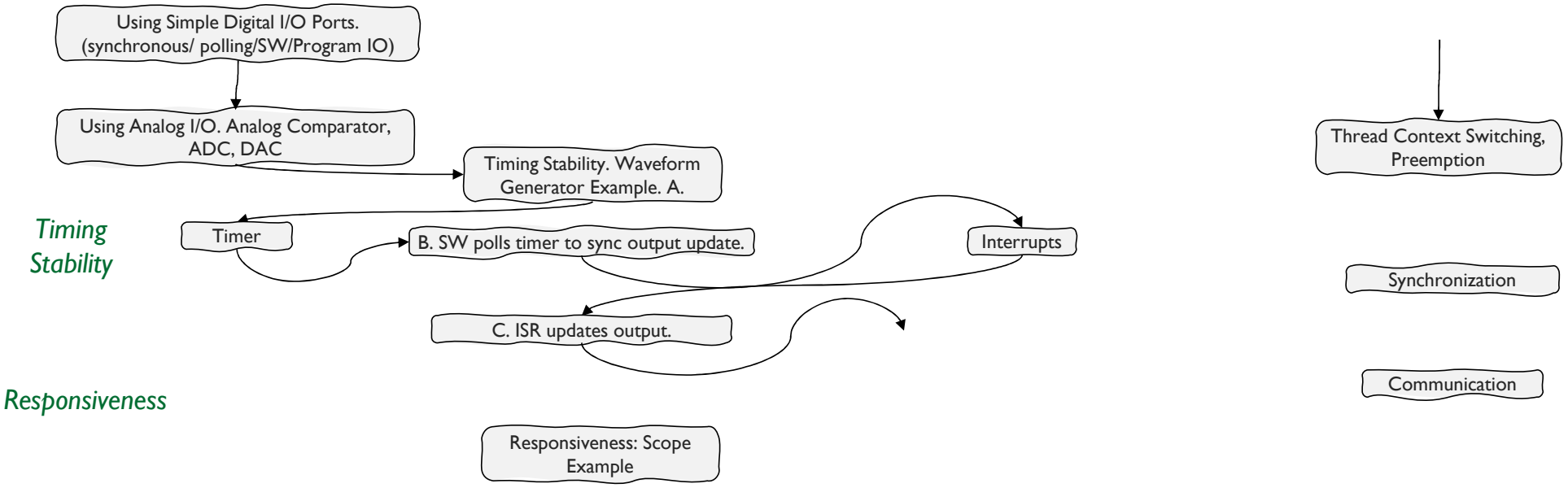
Single SW Process

Hardware Peripherals

Design Example

Concurrency Support: Interrupt System

Concurrency Support: RTOS

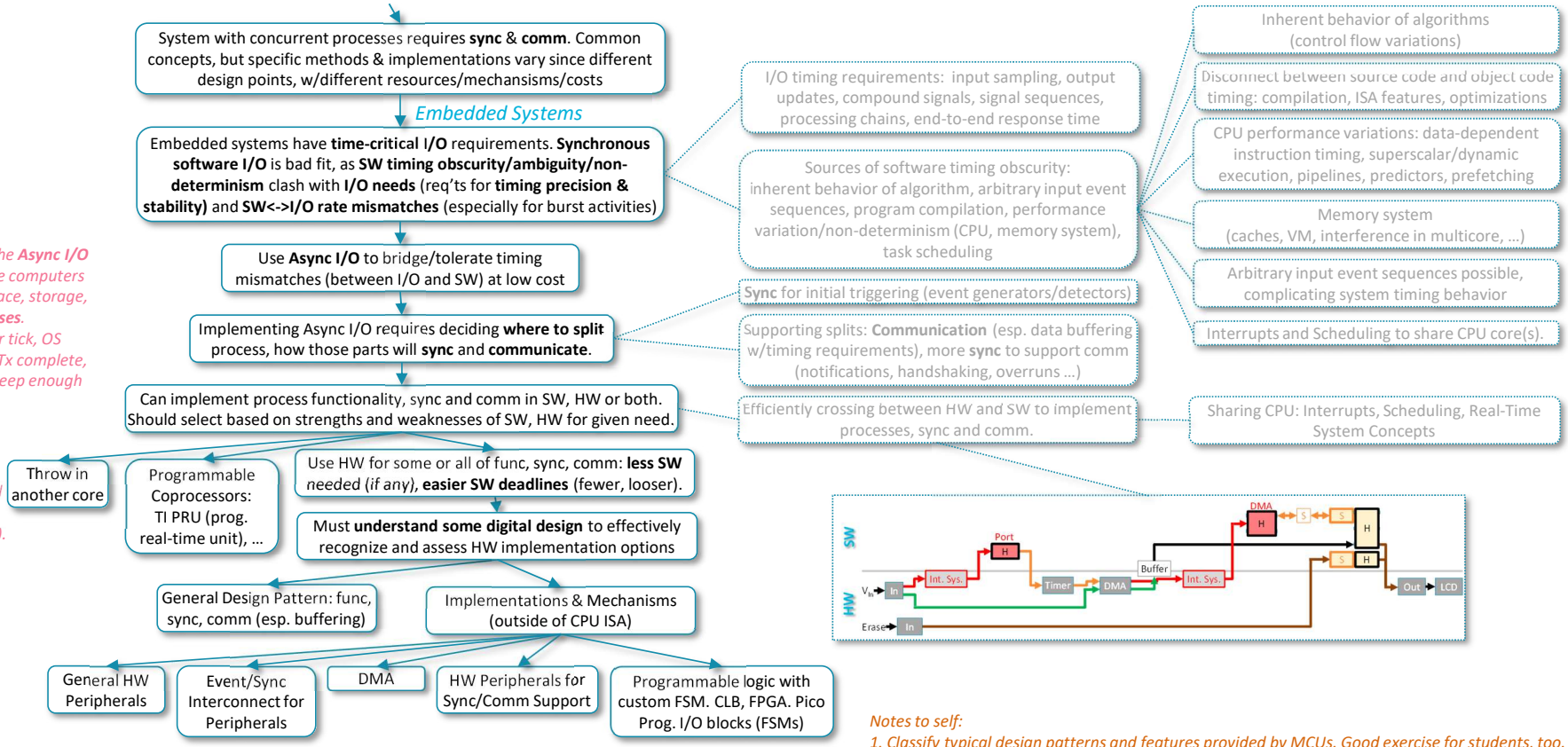


Waveform Sampling and Generation: Interface with Analog Inputs and Outputs

Oscilloscope: Concurrent, Communicating Processes in HW and SW

How are Embedded Computer Systems Different from General-Purpose Computers?

*Emb. Sys. have **concurrent processes** with **diverse, time-critical asynchronous I/O operations.** Use **HW peripherals** for **low-cost** solution.*



*Mainstream 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.*

When you have only a hammer, everything looks like a nail. CS education typically omits digital design (other than CPU, maybe memory system, AI accelerators, …).

Notes to self:
1. Classify typical design patterns and features provided by MCUs. Good exercise for students, too.
2. For design examples, quantify impact on CPU performance requirements (MHz).

How are Embedded Computer Systems Different from General-Purpose Computers?

*Emb. Sys. have **concurrent processes** with **diverse, time-critical asynchronous I/O operations.** Use **HW peripherals** for **low-cost** solution.*

Why Use Concurrent Processes?

Performance, i.e. Speed! Create processes to expose internal parallelism to use HW resources (multicore, multiprocessors)

Inherently Concurrent System: Embedded system, complex application, general purpose computer system

System with concurrent processes requires **sync** & **comm**. Common concepts, but specific methods & implementations vary since different design points, w/different resources/mechansisms/costs

Other → Mainstream concurrency education

Embedded Systems → Embedded systems have **time-critical I/O** requirements. **Synchronous software I/O** is bad fit, as ***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)

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**.

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.

- 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: func, 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)

*Mainstream 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.*

When you have only a hammer, everything looks like a nail. CS education typically omits digital design (other than CPU, maybe memory system, AI accelerators, …).

I/O timing requirements: input sampling, output updates, compound signals, signal sequences, processing chains, end-to-end response time

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

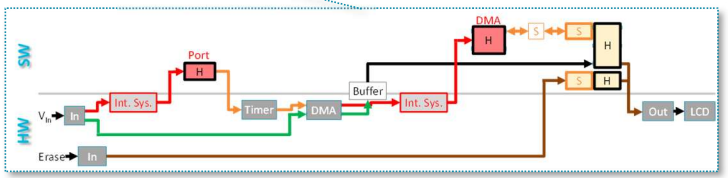
- Inherent behavior of algorithms (control flow variations)
- Disconnect between source code and object code timing: compilation, ISA features, optimizations
- CPU performance variations: data-dependent instruction timing, superscalar/dynamic execution, pipelines, predictors, prefetching
- Memory system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

Sync for initial triggering (event generators/detectors)

Supporting splits: **Communication** (esp. data buffering w/timing requirements), more **sync** to support comm (notifications, handshaking, overruns …)

Efficiently crossing between HW and SW to implement processes, sync and comm.

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



Notes to self:
1. Classify typical design patterns and features provided by MCUs. Good exercise for students, too.
2. For design examples, quantify impact on CPU performance requirements (MHz).

Concurrent Systems