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

Embedded Systems Topics (and Dependencies)

Embedded Systems High-Level View (1)

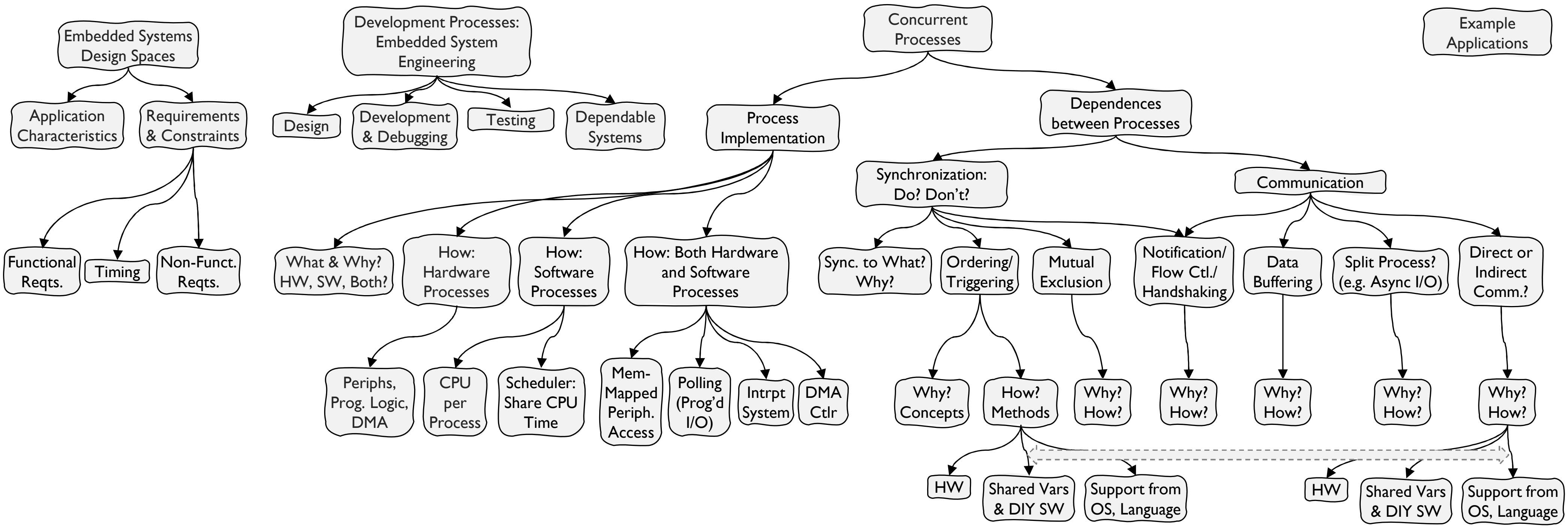
- Embedded Computer Systems frequently target **control applications**
  - Get input (read signal, detect event), Compute new output value, Update output
  - Microcontroller = Microprocessor + memory + hardware peripherals to support control
- Embedded Systems have **processes**, different implementation 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)

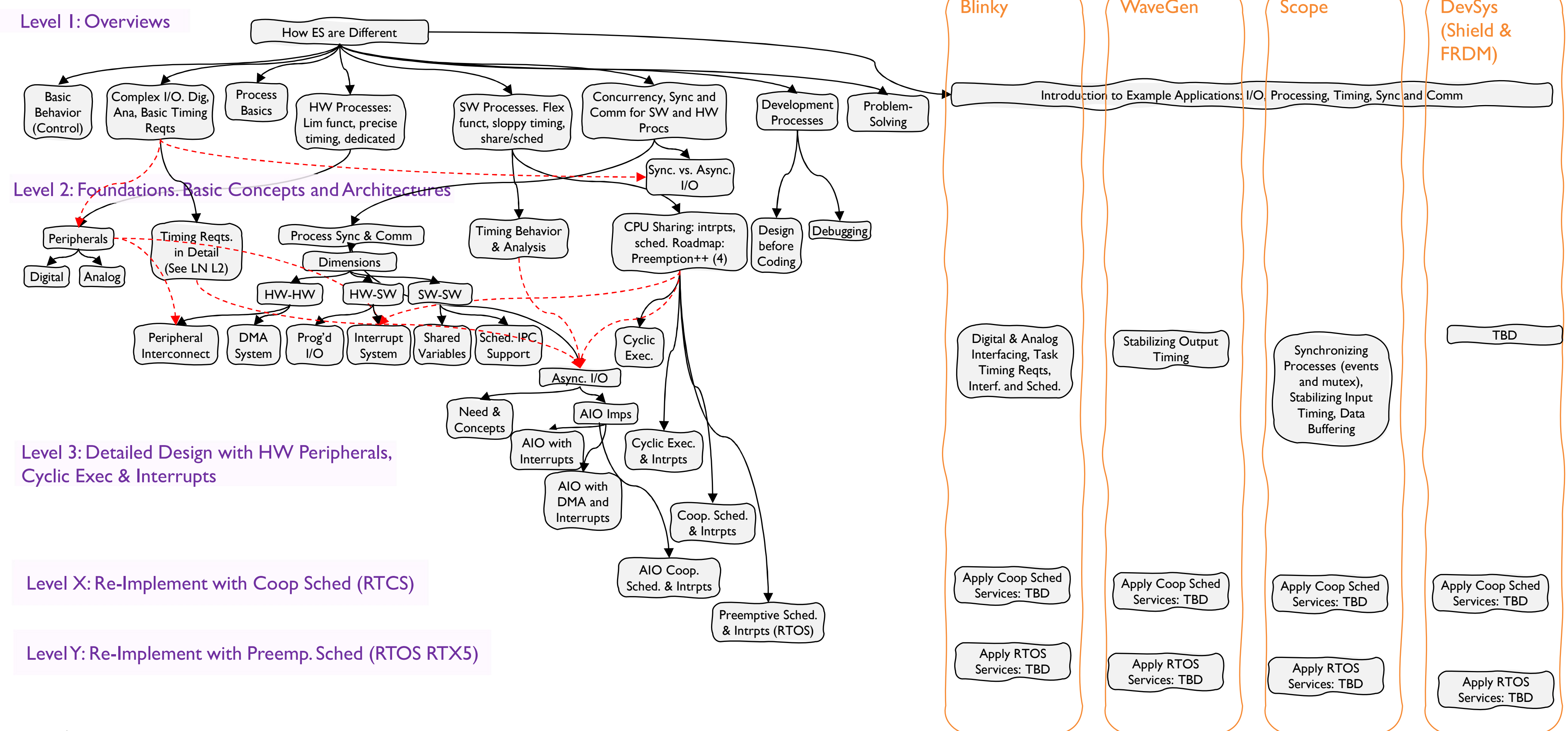
- The **I/O** for a process often has **challenging timing requirements**
  - **Periodic** events**,** events **synched to other/previous events** on this/other signals
- **Decouple the I/O** from compute software (bad timing characteristics) by splitting it into two or more processes to make **input 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



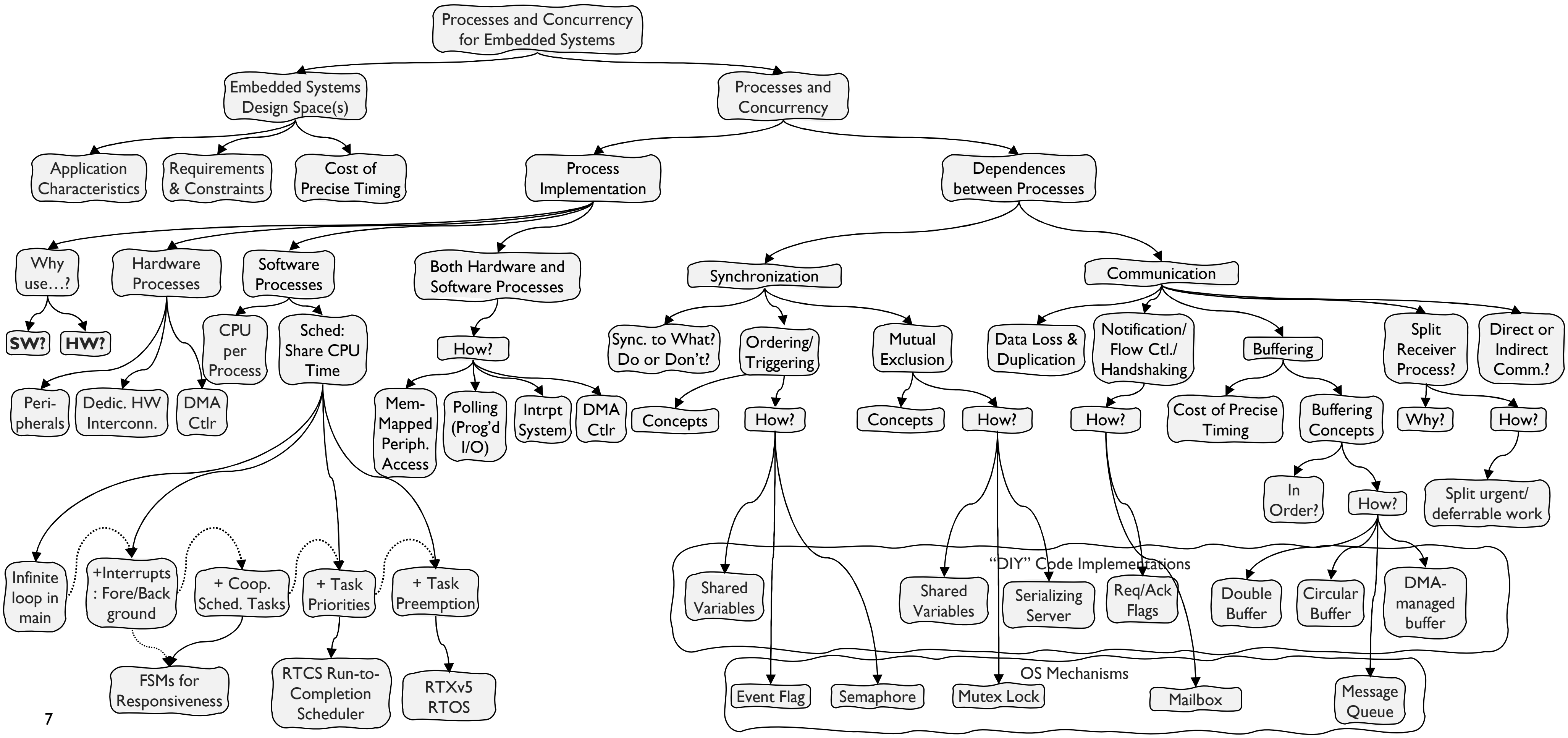
NC STATE UNIVERSITY

Yet Another Course Map



NC STATE UNIVERSITY

Extending the Topic Map



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																		Serial Comm.	I^2C Comm.	LCD Controller	Touchscreen	SMPS Controller	µSD via SPI
			HW	SW	HW	SW	HW	SW	SW->HW	HW->SW														

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 I 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

Class 02 Overview

- Review of **how** ES computers are different from GP, and **why**
- Topics
  - Example applications: functionality
  - Timing requirements
  - Timing capabilities and behaviors of hardware and software
  - Synchronization

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**.

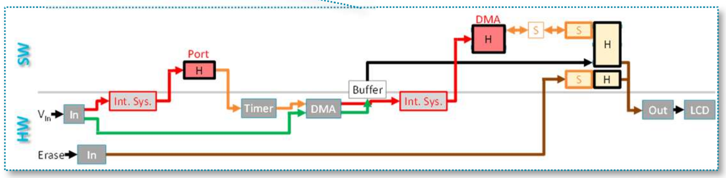
Sync for initial triggering (event generators/detectors)

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

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 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).

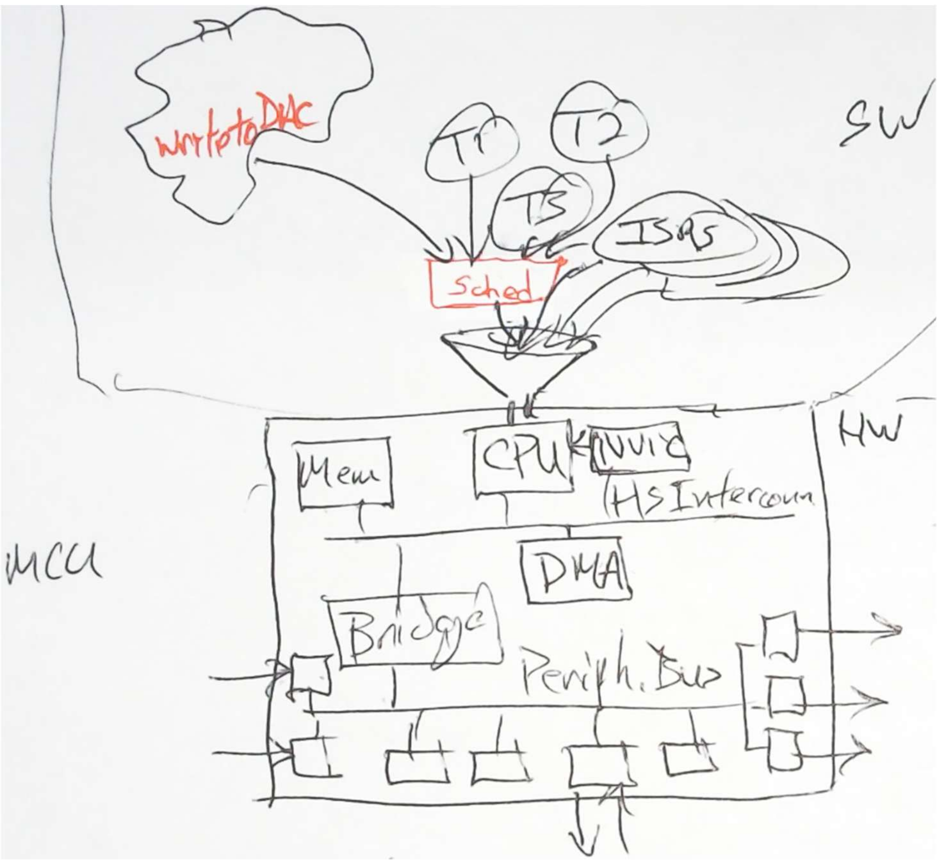
Must **understand some digital design** to effectively recognize and assess HW implementation options

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

Implementations & Mechanisms (outside of CPU ISA)

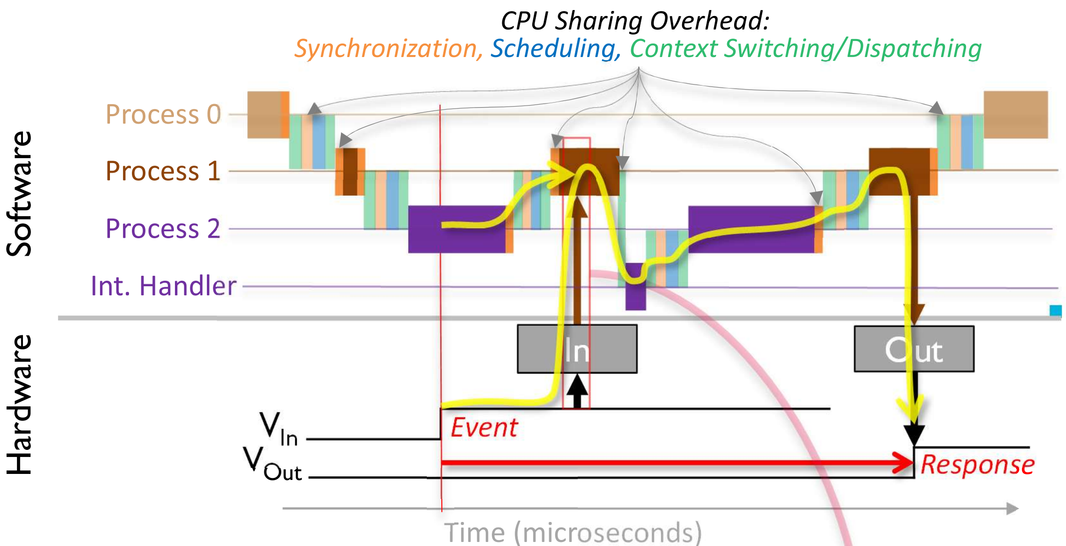
- **General HW Peripherals**
- **Event/Sync Interconnect for Peripherals**
- **DMA**
- **HW Peripherals for Sync/Comm Support**
- **Programmable logic with custom FSM. CLB, FPGA. Pico Prog. I/O blocks (FSMs)**

Use Software or Hardware? Flexibility vs. Timing Stability

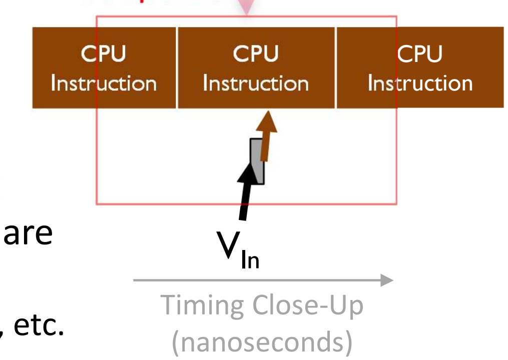


- Software
  - Program gives very flexible functionality
  - Interrupt system (e.g. NVIC) and scheduler (if any) determines **what** software runs on CPU and **when**
  - Software very vulnerable to timing interference. Use interrupts, scheduler to improve timing stability
- Hardware
  - Very stable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

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)

NC STATE UNIVERSITY

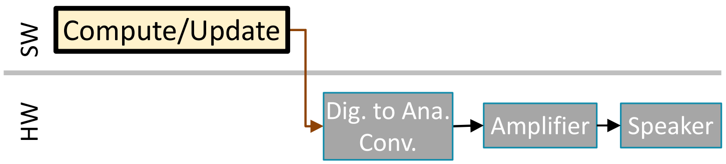
DESIGN EXAMPLES: LEVEL 1

Processes in ECE 306 Truck

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

Waveform Generator Subsystem: One Process

W1. WaveGen, base design



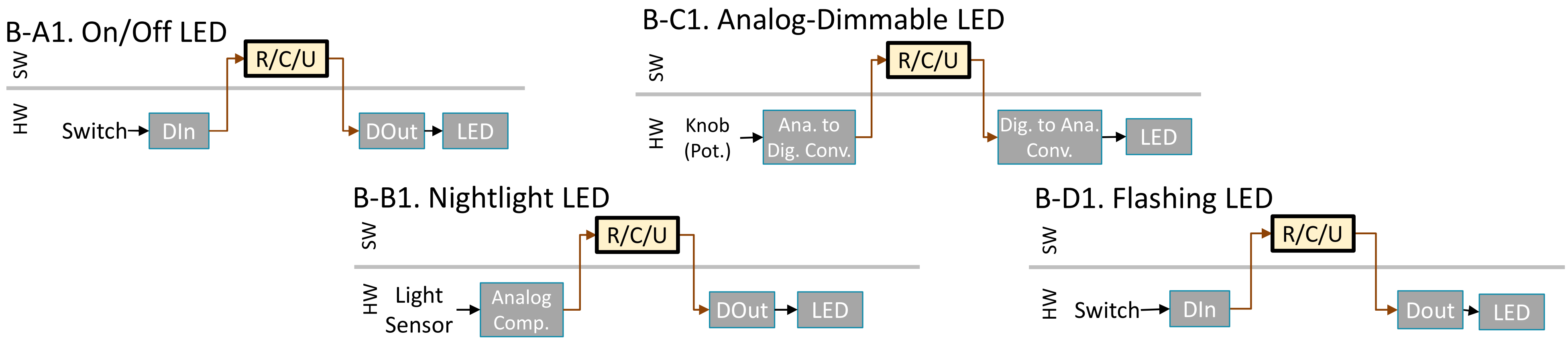
- Part of a larger system with other processes (e.g. user interface)
- Want to update DAC output every 50 us for a 20 kHz update rate
  - DAC signal amplified to drive speaker

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 (?)

Scope (Oscilloscope): One Process

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

Blinky Control Panel: Four Concurrent Processes



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

FRDM: Serial Communications Subsystem

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

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

Shield LCD Interface:

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

Shield: Touchscreen Interface

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

Timing Requirements

Timing Characteristics of Software