NC STATE UNIVERSITY

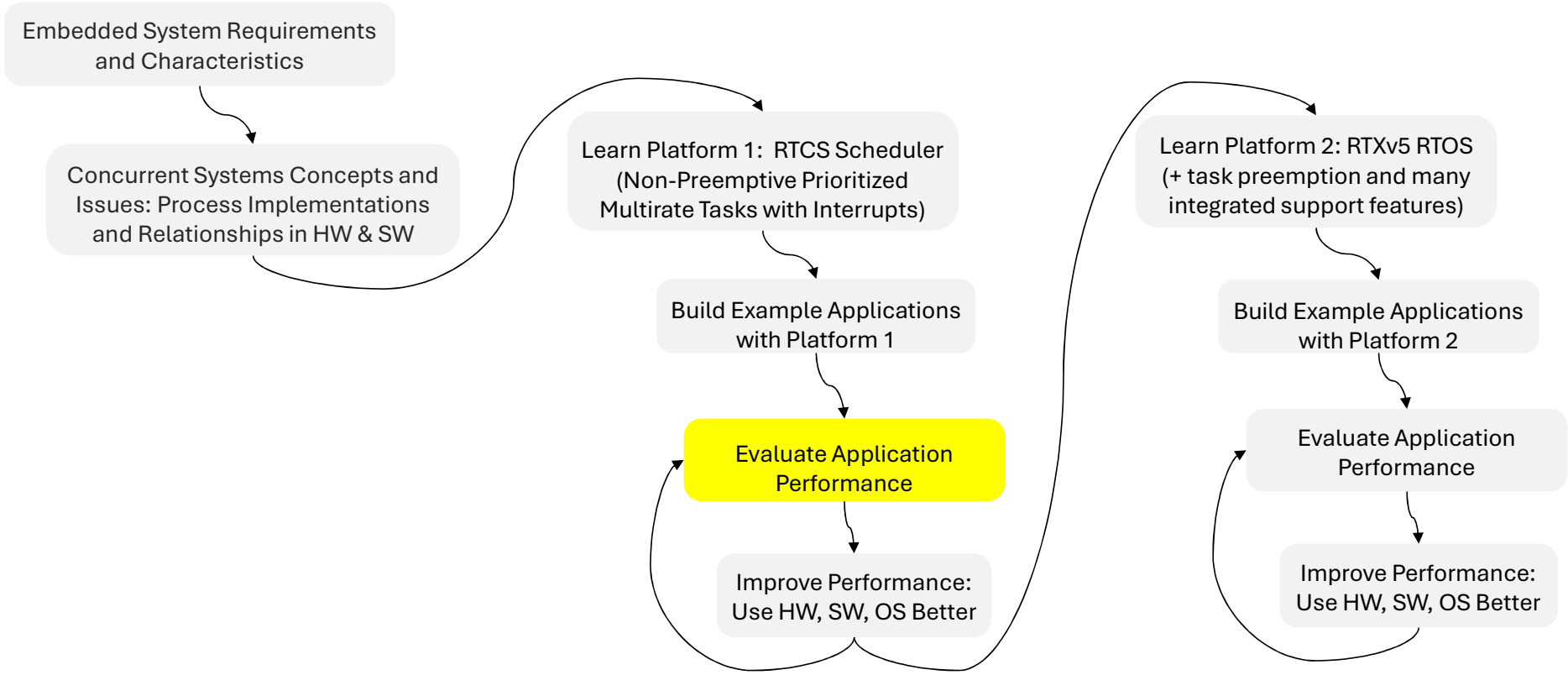
PERFORMANCE ANALYSIS OF EXAMPLE APPLICATIONS ON PLATFORM 1 (RTC SCHEDULER WITH INTERRUPTS, BASIC PERIPHERAL USE)

V2

9/25/2025

NC STATE UNIVERSITY

Where are we in the class?



NC STATE UNIVERSITY

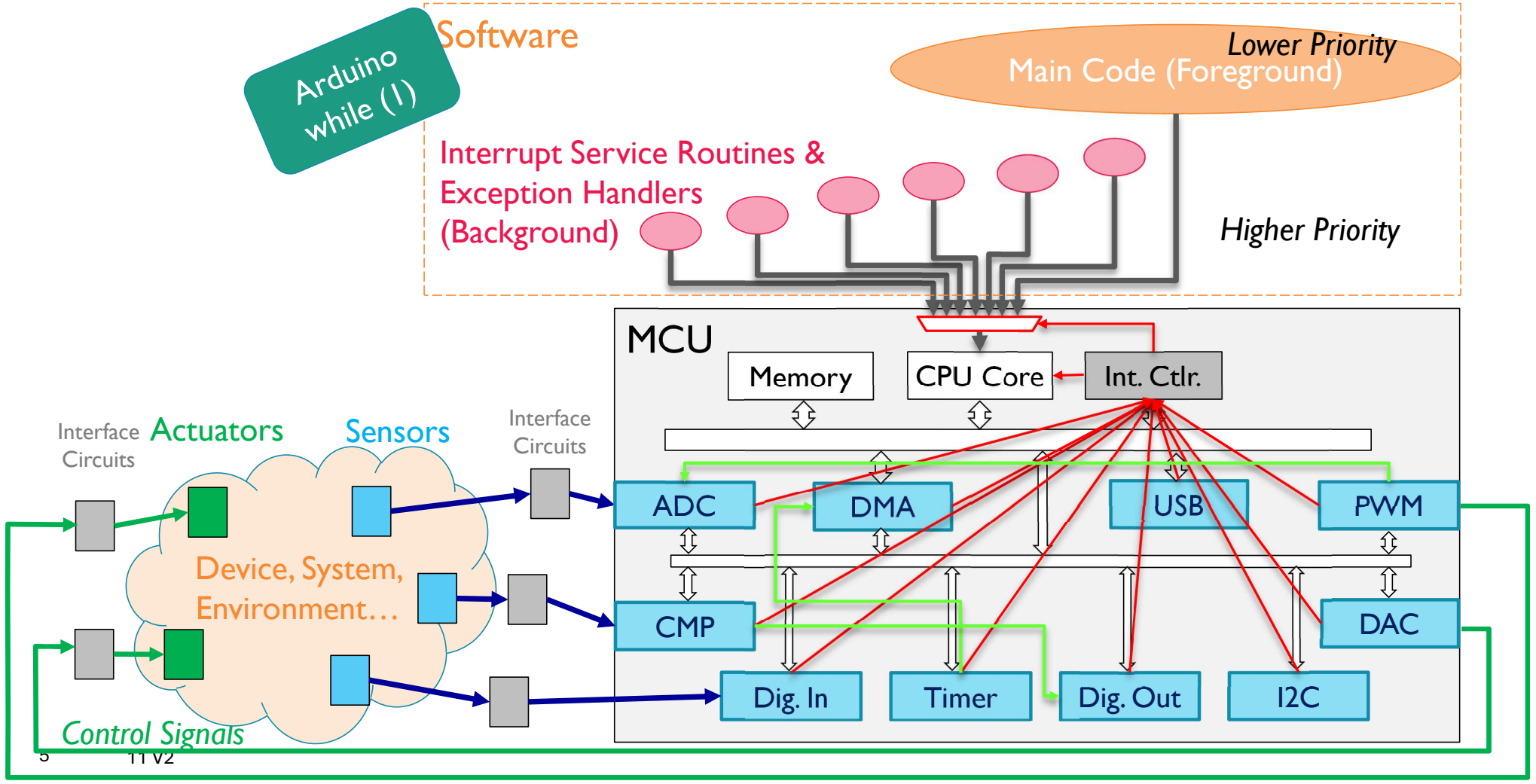
LN11 – Performance Analysis

- Summarize key characteristics of first implementations
  - Processes, triggers (external sync), detectors, schedulers, workers, internal sync in workers
- Response time analysis to bound system interference
  - ISR blocking by ISRs
  - ISR preemption by ISRs
  - Task blocking by tasks
  - Task preemption by ISRs
    - Tick ISR
    - App ISRs?
- Characterize basic performance limits of Platform 1 (48 MHz CM0+)
  - Interrupt System
    - Latency: 15 clock cycles
  - RTCS
    - Time-triggered scheduling resolution
    - Scheduler latencies: time to examine table, find next ready task
- Evaluate first implementations vs. performance limits
  - Identify performance gaps and risks, mark in table
- Evaluate response times for first implementations vs. requirements
  - Identify performance gaps and risks, mark in table

NC STATE UNIVERSITY

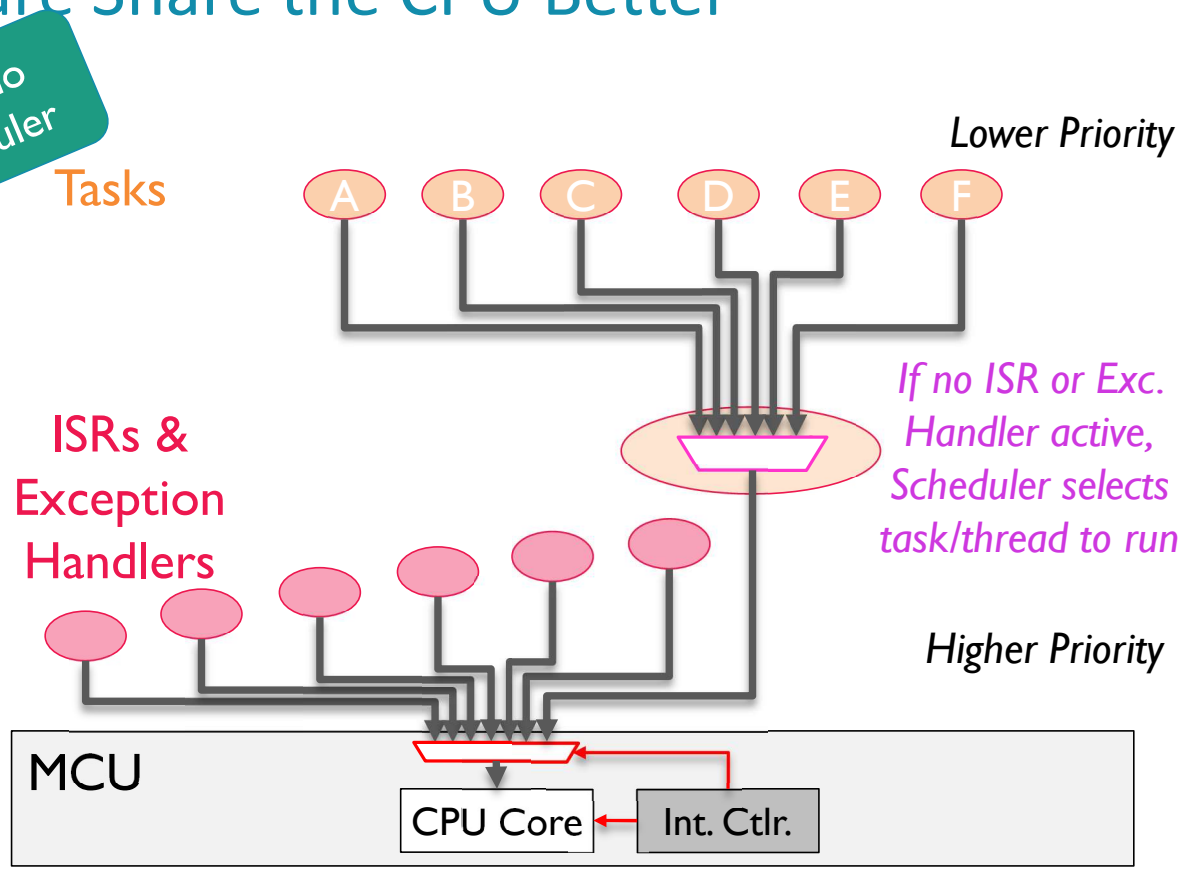
PERFORMANCE ANALYSIS 1: RESPONSE TIME ANALYSIS

Parallel Hardware … but Serialized Software



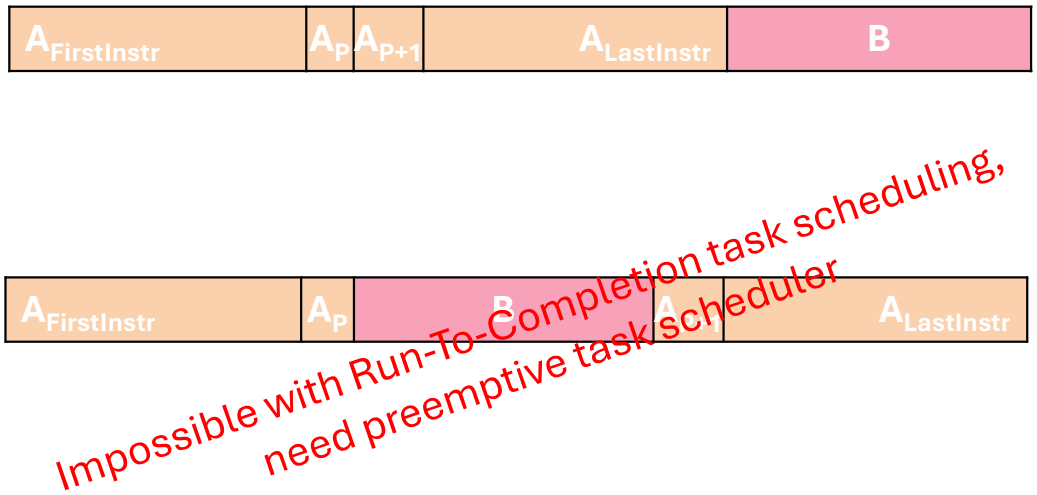
Schedulers: Helping Software Share the CPU Better

- Build modular program
  - Separate tasks/threads and ISRs, each running (mostly) independently
  - Easier to develop, maintain, debug
- What code does CPU run?
  - Normally CPU executes next instruction in program,
  - But **interrupt controller** can force CPU to execute handler code for interrupt or exception request
  - **Task scheduler can** decide which task/thread to run next

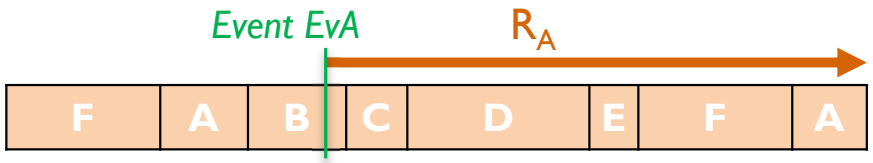


Run-To-Completion Tasks and Task Preemption

- With run-to-completion task scheduling…
  - Scheduler must wait for current task to **complete** before running another task
  - Tasks cannot pause (or be preempted by other tasks) partway through, and later resume at that point within the task.
  - If scheduler is running task A, it cannot
    - Pause A partway through (after instruction A_P),
    - Switch in task B and run it,
    - Resume task A partway through (at instruction A_{P+1})
- Preemptive task scheduler will support such task switching …
  - Letting task B preempt task A
  - Letting task A yield the CPU and later pick up where it left off



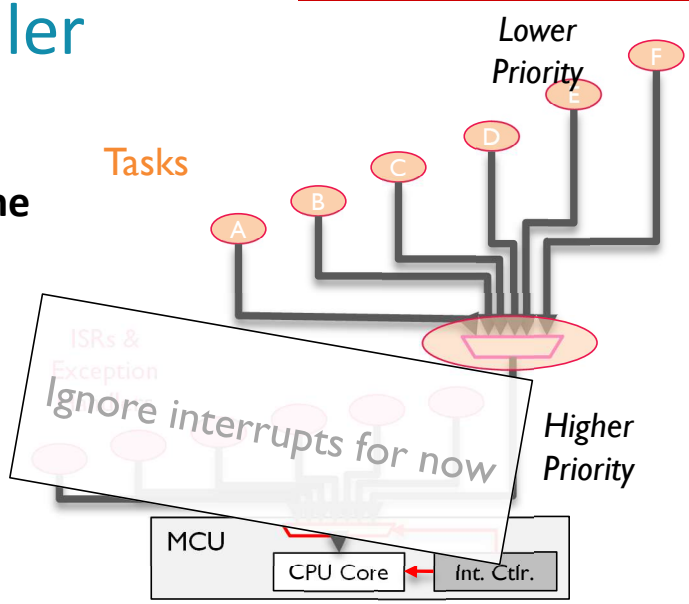
Responsiveness of While (1) Loop Scheduler



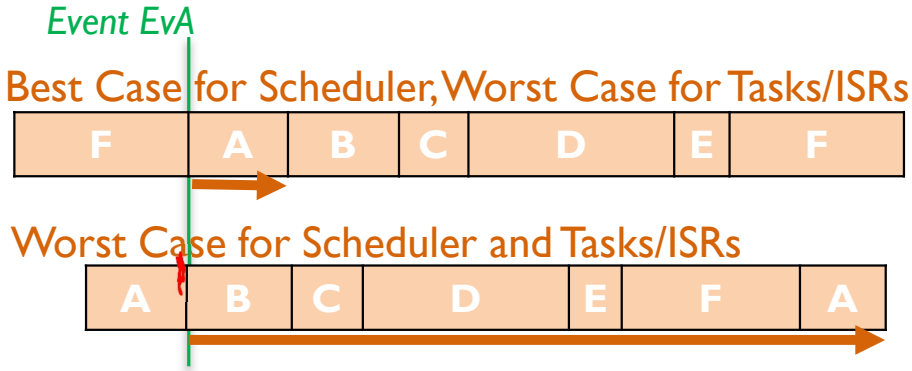
- Task **A** must run to service (handle) its event **EvA**
  - EvA makes scheduler ***release*** task A
- Task A's **response time** (R_A): How long from event EvA until task A **finishes** servicing it?
- Scheduler is While (1) loop
  - Tasks run to completion.
  - Fixed schedule: same task order every time
  - Round-robin: each task gets same number of chances to run
- Scheduler behavior:
  - EvA happened? Release A, run A until done.
  - EvB happened? Release B, run B until done.
  - EvC happened? Release C, run C until done.
  - Continue for all events/tasks, then repeat with EvA
- Note
  - We assumed each task takes a constant amount of time to execute
  - Task i probably has range of possible execution times, between $C_{i,Min}$ and $C_{i,Max}$

- Simplify timing model by making some worst-case performance assumptions
  - Design for worst case, so assume task i always takes $C_i = C_{i,Max}$
- Model will likely overestimate response time, but will never underestimate it – so it will be **safe**.

Responsiveness of While (1) Loop Scheduler

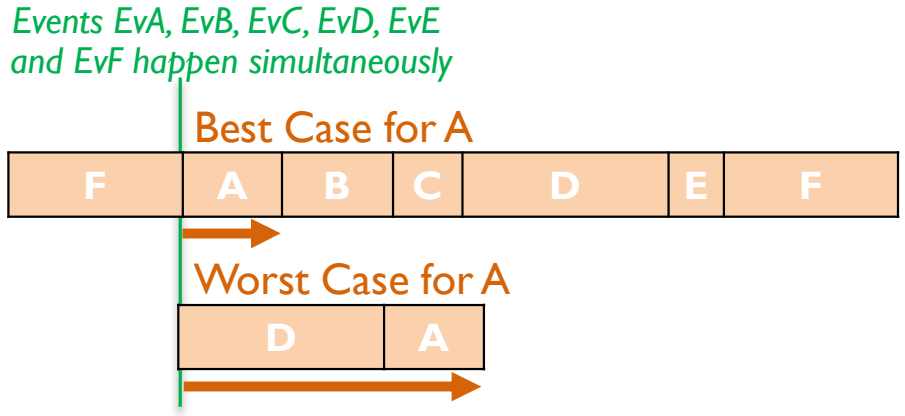


- Task A's **worst-case** response time: What is the **longest possible time** from event EvA until task A finishes servicing it?
  - Depends on what code runs: ISRs scheduled by Interrupt Controller, tasks scheduled by task scheduler



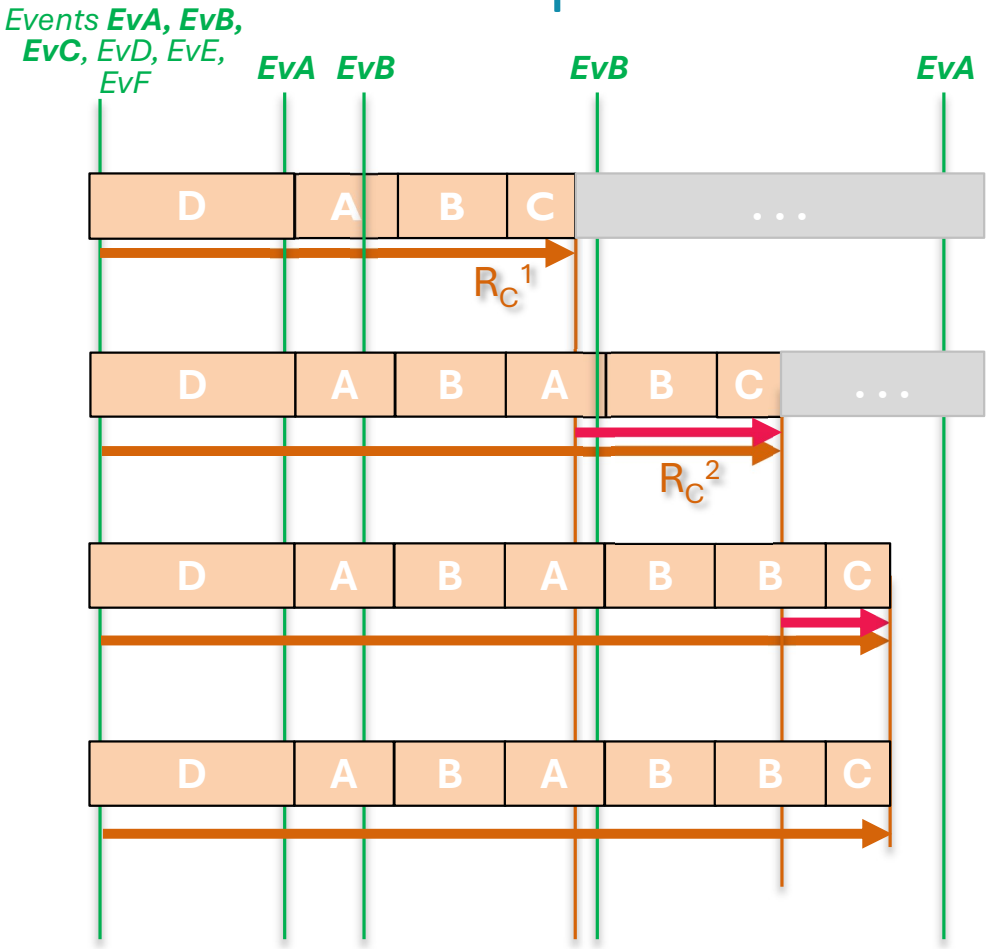
- Simplify: Initially ignore time taken by scheduler, interrupt system and interrupt handlers.
  - Best case: EvA happens just before scheduler checks it. $R_A = C_A$
  - Worst case: Every other event (EvB – EvF) happens before scheduler checks EvA, and EvA happens just after that check: $R_A = C_B + C_C + C_D + C_E + C_F + C_A$

Improvement: Prioritized Tasks



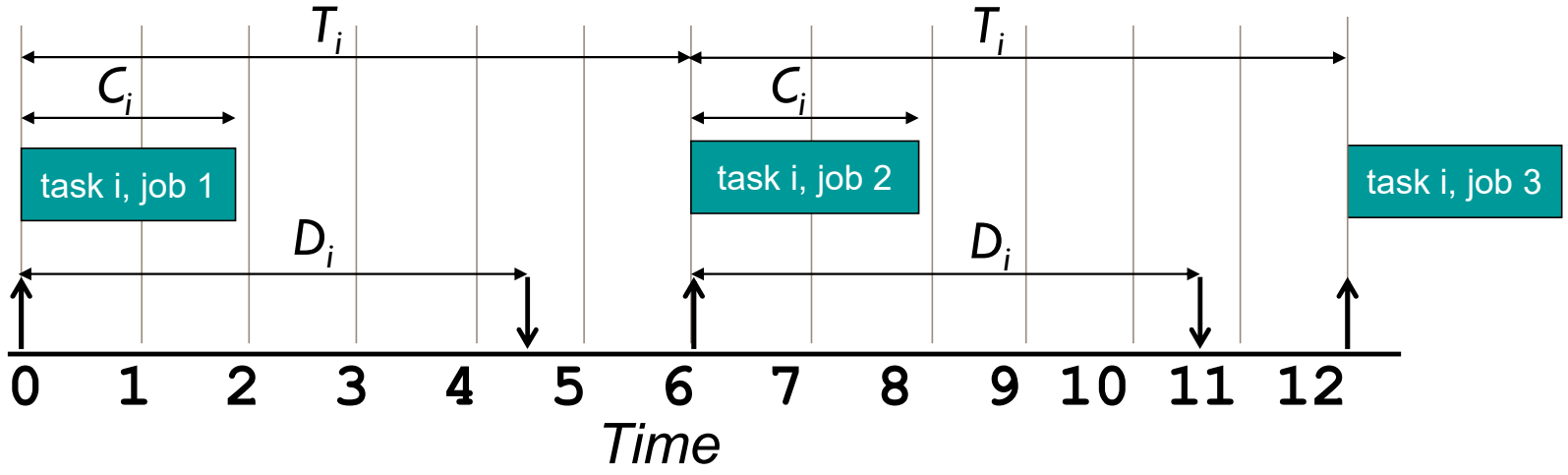
- Change scheduler to prioritize A > B > C etc.
- New behavior:
  - If EvA happened, run A, then check for EvA again.
    - Else if EvB happened, run B, then check for EvA again.
      - Else if EvC happened, run C, then check for EvA again.
        - Et cetera
- Implications
  - Not round-robin. Now have **dynamic** (not static) schedule of task orders, since higher priority tasks get chance to run before lower priority tasks.
  - Higher priority task may run multiple times before lower priority task gets to run once.
  - There may be more events (and task releases) further delaying the start of a task.
- Best case for Task A: Same as before. $R_A = C_A$
- Worst case for Task A (highest priority)?
  - Delayed by longest task (D). $R_A = C_A + \text{Max}(C_A, C_B, C_C, C_D, C_E, C_F)$
- Worst case for lower-priority tasks (B, C, D, E, F)?
  - Also may be delayed by higher-priority tasks. Details on next slide.

What about Response Time for Lower Priority Tasks?



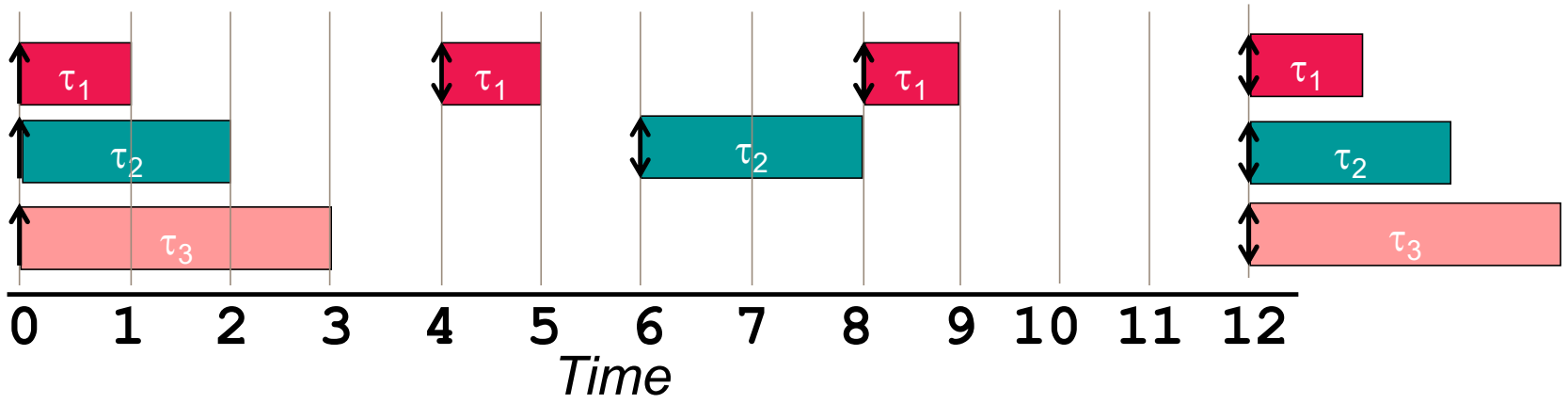
- First estimate (R_C^1) of response time R_C
  - Task C's finish may be …
    - delayed by **blocking** once by longest task if already running: Max(C_A, C_B, C_C, C_D, C_E, C_F)
    - delayed at least once by each higher priority task (C_A, C_B)
  - Equations
    - R_C^1 = Max(C_A, C_B, C_C, C_D, C_E, C_F) + C_A + C_B + C_C
    - Here: R_C^1 = C_D + C_A + C_B + C_C
- Second estimate (R_C^2)
  - **More** events for higher-priority tasks (A, B) may happen before C starts (during vulnerable time), leading to more releases before C can start
    - R_C^2 = Max(C_A, C_B, C_C, C_D, C_E, C_F) + 2*C_A + 2*C_B + C_C
    - Here: R_C^2 = C_D + 2*C_A + 2*C_B + C_C
- What if C ***still hasn't started*** when A or B is released again? Must ***repeat*** to see if A or B are released again during this additional vulnerable time
  - Depends on minimum time between releases (EvA to EvA, EvB to EvB) in the worst case (burst)

Periodic Task Model of Computational Requirements



- Periodic Task Model describes characteristics for each task τ_i
  - Job = a specific instance of that task running
  - Task releases job so scheduler can run it
- A periodic task *i* releases a job every T_i time units
- Job may have an absolute deadline D_i after its release
- Job takes a constant time C_i to execute
- Simplifying assumptions include
  - no time needed for scheduler, task switching, ISR response/return

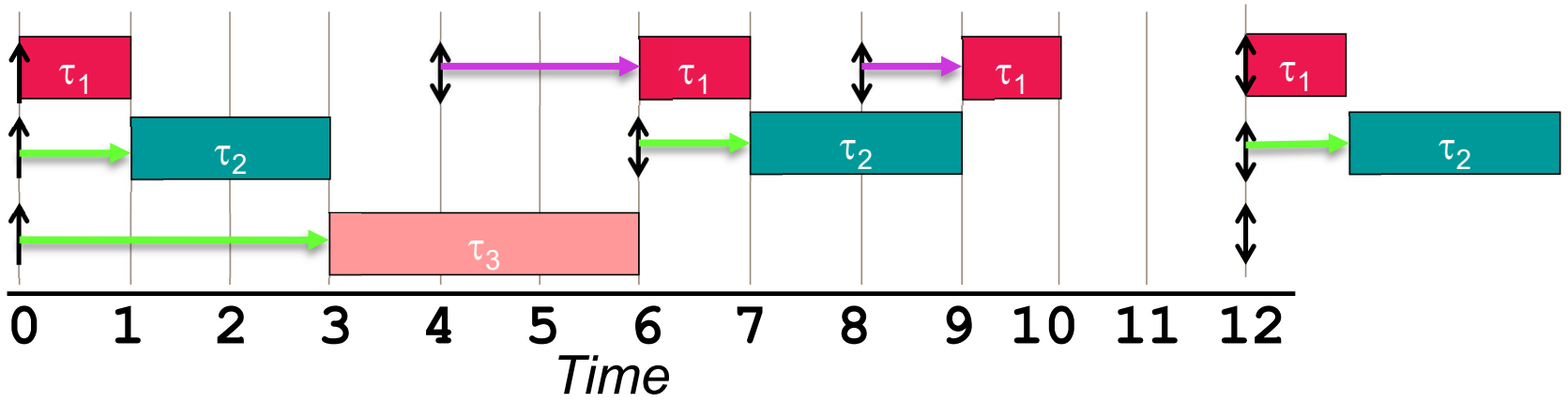
Example Workload: What We Ask For



- Set of tasks with real-time requirements
- What gets executed when?
  - Depends on scheduler and task priorities

Task	Exec. Time C_i	Period T_i	Deadline D_i
τ_1	1	4	4
τ_2	2	6	6
τ_3	3	12	12

Scheduled Workload: What We Get



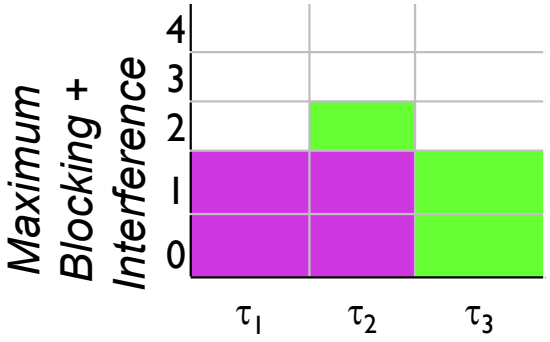
- Example: Scheduler and task fixed priorities
  - Assign priorities as shown
  - Use a non-preemptive scheduler
- What can delay a task?
  - I: Interference caused by higher priority tasks
  - B: Blocking caused by lower priority tasks
- Response time = Computation + Blocking + Interference

Task	Exec. Time C_i	Period T_i	Deadline D_i	Priority
τ_1	1	4	4	*High*
τ_2	2	6	6	*Medium*
τ_3	3	12	12	*Low*

$$R_i = C_i + B_i + I_i$$

Non-Preemptive Scheduling

Task	Exec. Time C_i	Period T_i	Priority
τ_1	1	4	*High*
τ_2	1	5	*Medium*
τ_3	3	7	*Low*



τ_1	Job 1					Job 2					Job 3		Job 4					Job 5			Job 6
τ_2		Job 1					Job 2					Job 3							Job 4		
τ_3			Job 1	Job 1	Job 1			Job 2	Job 2	Job 2					Job 3	Job 3	Job 3				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

NUMERICAL RESPONSE TIME ANALYSIS

Numerical Response Time Analysis, Step 1

- How long could it take for task *i* to complete? What is its *response time* R_i?
- Initial estimate based on worst case:

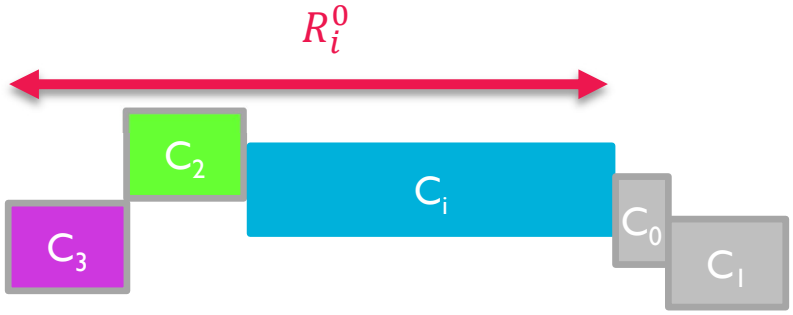
  R_i^0 = computation time for task *i* + computation time for other tasks.

  - Non-prioritized scheduling: Every other task can run once

```
while (1) {
  for (j=0; j<NUM_TASKS; j++) {
    if (Tasks[j].RP > 0) {
      Tasks[j].RP--;
      Tasks[j].Task();
} } }
```

$$R_i^0 = C_i + \sum_{j \neq i} C_j$$

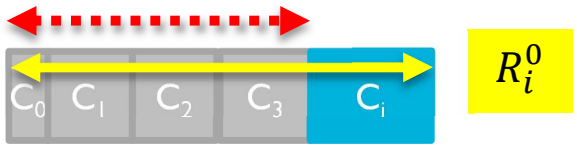
  - Prioritized scheduling: All higher-priority tasks (+ longest task if non-preemptive) can run once



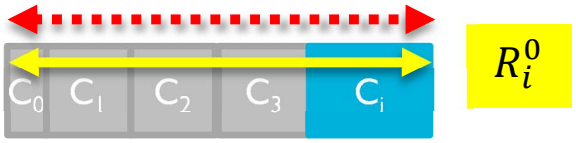
$$R_i^0 = C_i + \max_j(C_j) + \sum_{j \in hp(i)} C_j$$

Additional Timing Interference, Steps 2, 3, 4 …

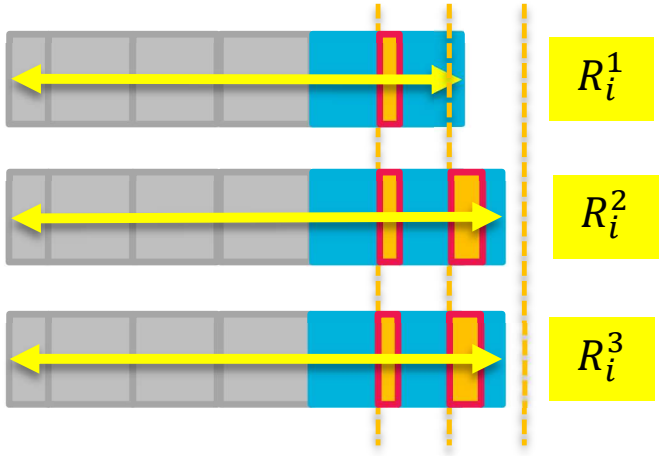
- Task *i* is vulnerable to delays from *new job releases* during vulnerable time
  - Non-preemptive: 0 to $R_i^n - C_i$ since task *i* can't be preempted after it starts



  - Preemptive: 0 to R_i^n since higher-priority task can preempt task *i*



- Consider new releases to update completion time estimate R_i^{n+1}
- Repeat until no new releases, or any deadline (if present) is missed



How Many T_i Releases Possible During Vulnerable Time?

- Initial estimate was one release, so task's time is one job: 1*C_j

$$R_i^0 = C_i + \sum_{j \neq i} C_j$$

$$R_i^0 = C_i + \max_j(C_j) + \sum_{j \in hp(i)} C_j$$

- Remaining estimates must consider all job releases possible during vulnerable time: Ceiling(vulnerable time / T_j)*C_j

	Non-prioritized	Prioritized
Non-preemptive	$\sum_{j \neq i} \left\lceil \frac{R_i^n - C_i}{T_j} \right\rceil C_j$	$\sum_{j \in hp(i)} \left\lceil \frac{R_i^n - C_i}{T_j} \right\rceil C_j$
Preemptive	$\sum_{j \neq i} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$	$\sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$

Response Time: Indep. Tasks with Task Preemption + Prioritization

- Preemption …
  - Eliminates blocking of task *i* by lower-priority **independent** tasks.
  - Allows higher-priority tasks to preempt task *i*

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$$

NC STATE UNIVERSITY

SUMMARY OF APPLICATION DESIGN APPROACHES

NC STATE UNIVERSITY

Example Application and Subsystem Processes

1. Quadrature Decoder w/Limit Switch
2. Waveform Generator
3. Blinky Control Panel
4. Touchscreen
5. Serial UART Communications
6. LCD Controller
7. Scope
8. SMPS Controller
9. More Comms
    1. SPI
    2. Higher protocol layers: I^2C, Secure Digital via SPI

NC STATE UNIVERSITY

Applications: Functionality First, then Performance

			Quad. Dec. w/Z Limit Switch	Waveform Generator	Blinky Control Panel	Touch screen	Serial Comms.	LCD Controller	Scope	SMPS Controller	I²C Comms.	µSD via SPI Comms.
Providing Functionality	Interfacing	Simple Digital	In		In, Out	Out				PWM Out		
		Complex Digital			PWM		In, Out	Bus Out			In, Out	In, Out
		Analog		Out	ADC In, CMP In, DAC Out	ADC In			ADC In, Cmp In	ADC In with Sync. Sampling		
	Async	# Processes for async. exec.	1,2	1,2	4	0, 1	2	0	1,2	1	1	2
	Sync and …	Sync and Do: Coarse Triggering	Digital Edge Detection	Periodic Output Updates			Tx Rdy, Rx Done events. Producer & Consumer		Ana. Edge Det., Periodic In. Smplg., Buffer mgt.	ADC In with Sync. Sampling	Data Producers & Consumers. I2C Device read response.	Tx Rdy, Rx Done events. Prod. & Cons.
		Internal, Fine Grain Block/Sched/Trig			ADC conv. time	ADC conv. time	When to notify receiver?				I2C message internal events & timing reqts. for conditions, data	
		Sync and Don't: Sharing & Races					Tx, Rx byte queues		LCD Ctlr Sharing, Data buffer mgt.		Tx, Rx Msgs	Tx, Rx byte queues
	IPC	Inter-Process Comm.	Shared Position Variable						Data buffer			
Meeting Perf. Reqts.		Timing Stability		1	1				2	1		
		Responsiveness			1				1	1		1
		Reducing SW Overhead		2				1- Perf. Optimiz.	2	2	1 – series of timed I/O events per message. FSM vs. RTOS	
		Tolerating Timing Mismatches		2			2		2			2

NC STATE UNIVERSITY

Example Application Data

Application	Independent Process?	Triggers	Trigger Detection	Scheduler	Work done in ...	Internal Sync. #1 in Work Code	Internal Sync. #2 in Work Code
Quadrature Decoder	Y	Events: ↑A, ↑Z	Peripheral (PORT)	Interrupt System	ISR(s) for A, Z	-	
Waveform Generator	Y	Time - Periodic: T_{sample}	RTCS Tick Handler	RTCS	Task_Update_DAC	-	
Blinky Control Panel	Y	Time - Periodic: T_{Poll}	RTCS Tick Handler	RTCS	Task_On_Off	-	
	Y	Time - Periodic: T_{Poll}	RTCS Tick Handler	RTCS	Task_Level_Alarm	-	
	Y	Time - Periodic: T_{Poll}	RTCS Tick Handler	RTCS	Task_Dimmer	Wait for A/D conversion to complete	
	Y	Time - Periodic: T_{Flash}	RTCS Tick Handler	RTCS	Task_Flash	-	
Touchscreen	Y	Time - Periodic: T_{Poll}	RTCS Tick Handler	RTCS		Wait for 1st A/D conversion to complete	Wait for 2nd A/D conversion to complete
Serial UART Comms.	Y	Events: UART Tx events	Peripheral (UART)	Interrupt System	ISR for Tx	-	
	Y	Events: UART Rx events	Peripheral (UART)	Interrupt System	ISR for Rx	-	
LCD Controller	N	When called by process	n/a	n/a		Not needed.	
Oscilloscope	Y	Event: V_{In} ↑ $V_{Trigger}$	Software polling of ADC	?	Detect Trigger	Wait for A/D conversion complete	
		Trigger detected and Time - Periodic: T_{Sample}	?	?	Sample Data	Wait for A/D conversion complete	
		Sample Data completes			Plot Data	-	
SMPS Controller	Y	Event: PWM Timer Phase Reference	Peripheral (ADC)	(direct hardware connection)	ADC Peripheral		
	Y	Event: A/D Conversion Completed	Peripheral (ADC)	Interrupt System	ISR for ADC	Not needed.	
I^2C Comms.	Y	Events: I^2C message components					
µSD via SPI Comms.	Y	Events: Data Exchange events				Many: SD Ctlr delays	

NC STATE UNIVERSITY

PERFORMANCE ANALYSIS 2: PLATFORM LIMITS

NC STATE UNIVERSITY

Platform 1 Performance Limits

- Characterize basic performance limits of Platform 1 (48 MHz Cortex-M0+)
  - Interrupt System
    - Latency: 15 clock cycles
  - RTCS
    - Time-triggered scheduling resolution: 1 tick
    - Scheduler latencies: time to examine table, find next ready task, call task function

NC STATE UNIVERSITY

Evaluation of First Implementations

- Identify performance gaps, limits, and risks, mark in table

Application
Quadrature Decoder
Waveform Generator
Blinky Control Panel
Touchscreen
Serial UART Comms.
LCD Controller
Oscilloscope
SMPS Controller
I^2C Comms.
µSD via SPI Comms.