Microprocessor Systems

Fall 2024

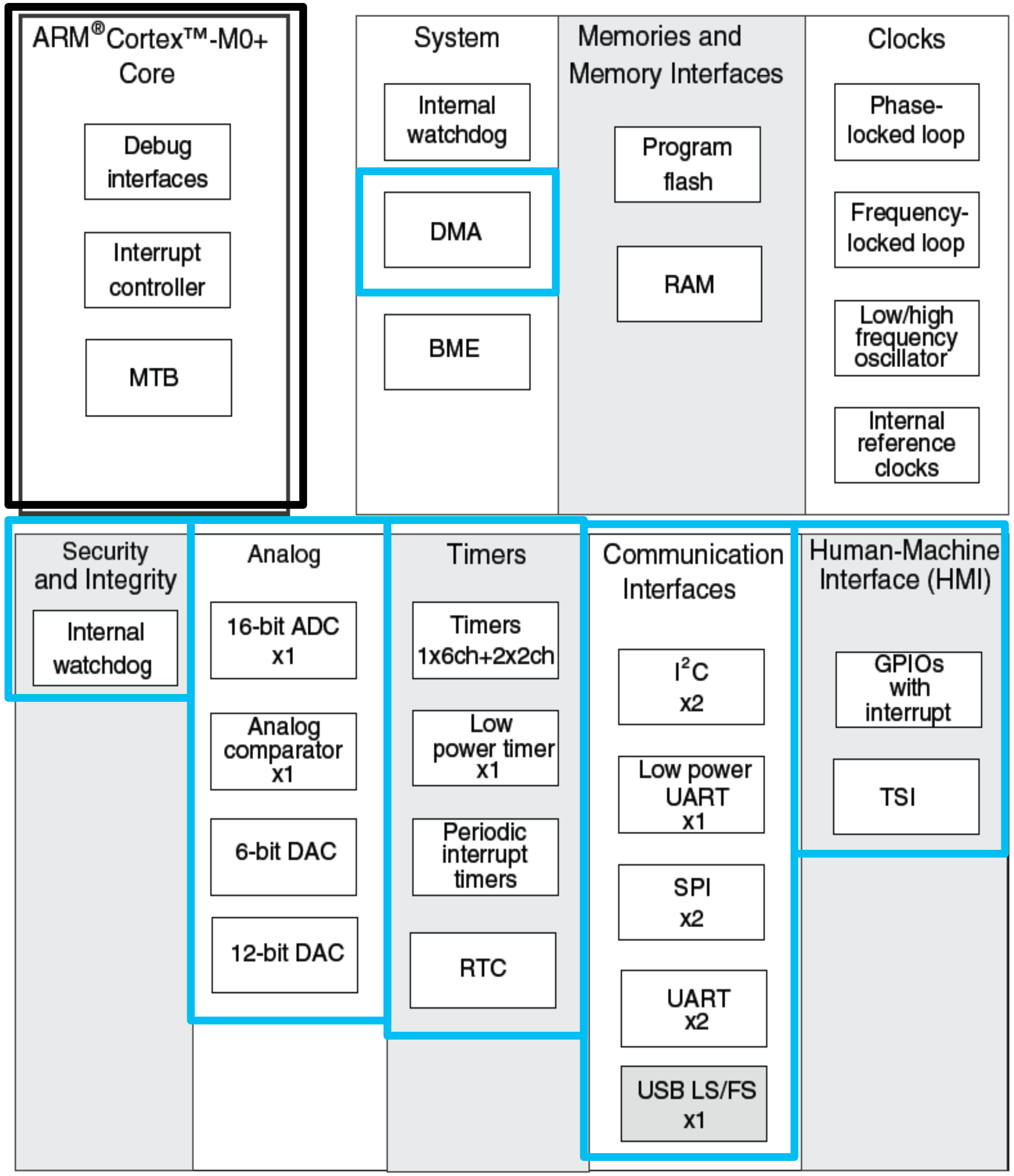
CONCURRENCY

Overview

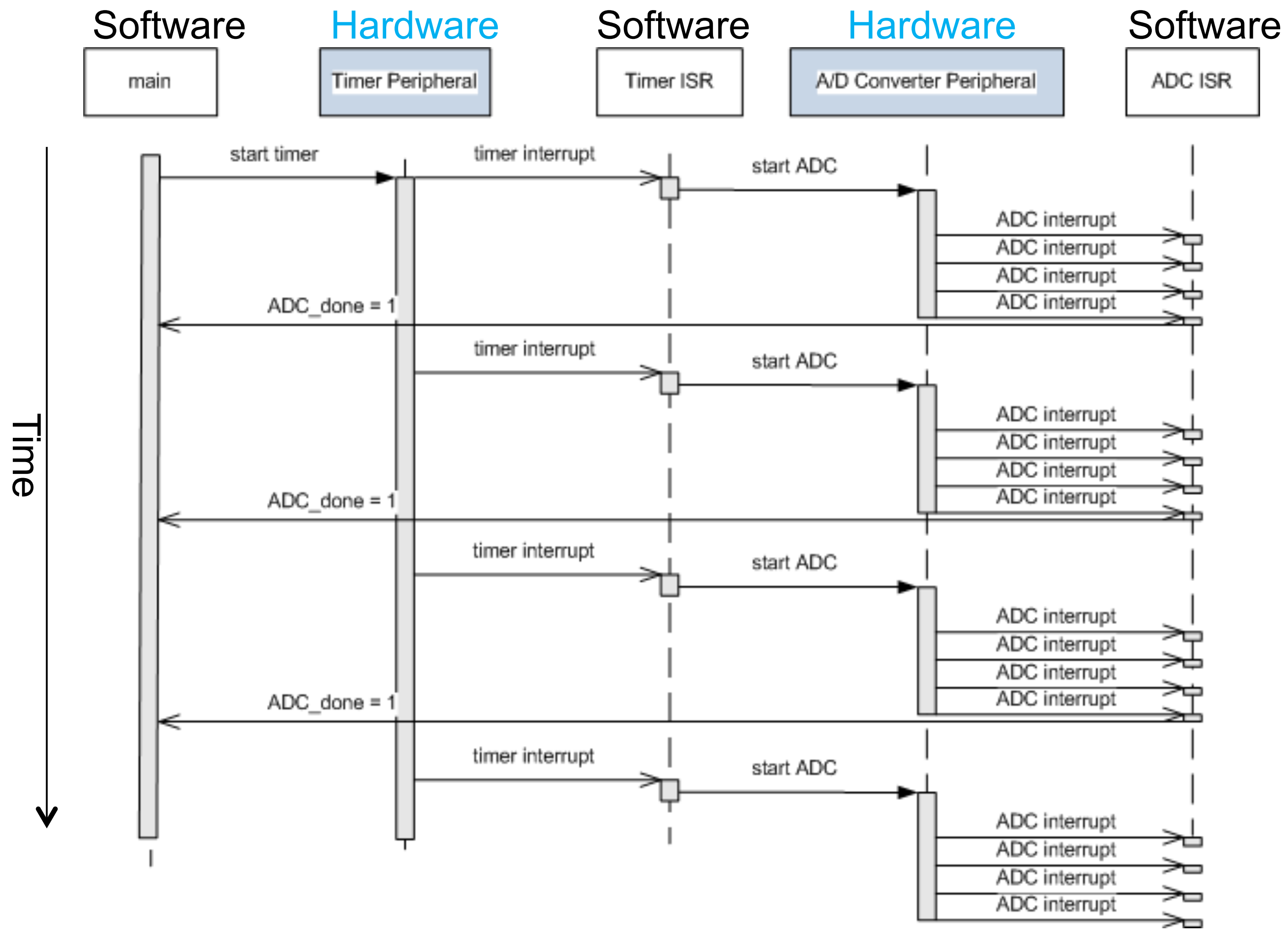
- Concurrency
  - How do we make things happen at the right times?

MCU Hardware & Software for Concurrency

- CPU executes instructions from one or more threads of execution
- Specialized hardware peripherals add dedicated concurrent processing
  - DMA - transferring data between memory and peripherals
  - Watchdog timer
  - Analog interfacing
  - Timers
  - Communications with other devices
  - Detecting external signal events
- Peripherals use ***interrupts*** to notify CPU of events



Concurrent Hardware & Software Operation



- Embedded systems rely on both MCU ***hardware peripherals*** and ***software*** to get everything done on time

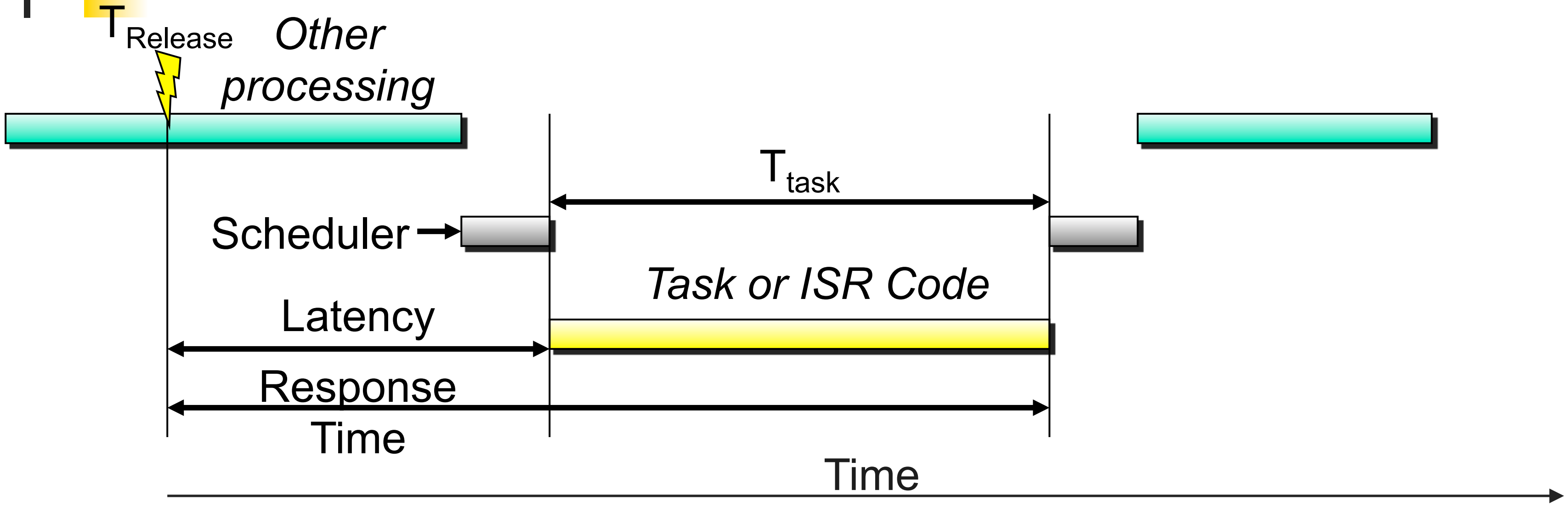
CPU Scheduling

- MCU's Interrupt system provides a basic scheduling approach for CPU
  - "Run this subroutine every time this hardware event occurs"
  - Is adequate for simple systems

- More complex systems need to support multiple concurrent independent threads of execution
  - Use task scheduler to share CPU
  - Different approaches to task scheduling

- How do we make the processor responsive? (How do we make it do the right things at the right times?)
  - If we have more software threads than hardware threads, we need to share the processor.

Definitions



- $T_{Release}(i)$ = Time at which task (or interrupt) i requests service/is released/is ready to run
- $T_{Latency}(i)$ = Delay between release and *start of service* for task i
- $T_{Response}(i)$ = Delay between request for service and *completion of service* for task i
- $T_{Task}(i)$ = Time needed to perform computations for task i
- $T_{ISR}(i)$ = Time needed to perform interrupt service routine i

Scheduling Approaches

- Rely on MCU’s hardware interrupt system to run right code
  - Event-triggered scheduling with interrupts
  - Works well for many simple systems

- Use software to schedule CPU’s time
  - Static cyclic executive
  - Dynamic priority
    - Without task-level preemption
    - With task-level preemption

Event-Triggered Scheduling using Interrupts

- Basic architecture, useful for simple low-power devices
  - Very little code or time overhead
- Leverages built-in task dispatching of interrupt system
  - Can trigger ISRs with input changes, timer expiration, UART data reception, analog input level crossing comparator threshold
- Function types
  - Main function configures system and then goes to sleep
    - If interrupted, it goes right back to sleep
  - Only interrupts are used for normal program operation
- Example: bike computer
  - Int1: wheel rotation
  - Int2: mode key
  - Int3: clock
  - Output: Liquid Crystal Display

Bike Computer Functions

Reset

```
Configure timer
inputs and
outputs

cur_time = 0;
rotations = 0;
tenth_miles = 0;

while (1) {
  sleep;
}
```

ISR 1: Wheel rotation

```
rotations++;
if (rotations>
  R_PER_MILE/10) {
  tenth_miles++;
  rotations = 0;
}
speed =
 circumference/
 (cur_time – prev_time);
compute avg_speed;
prev_time = cur_time;
return from interrupt
```

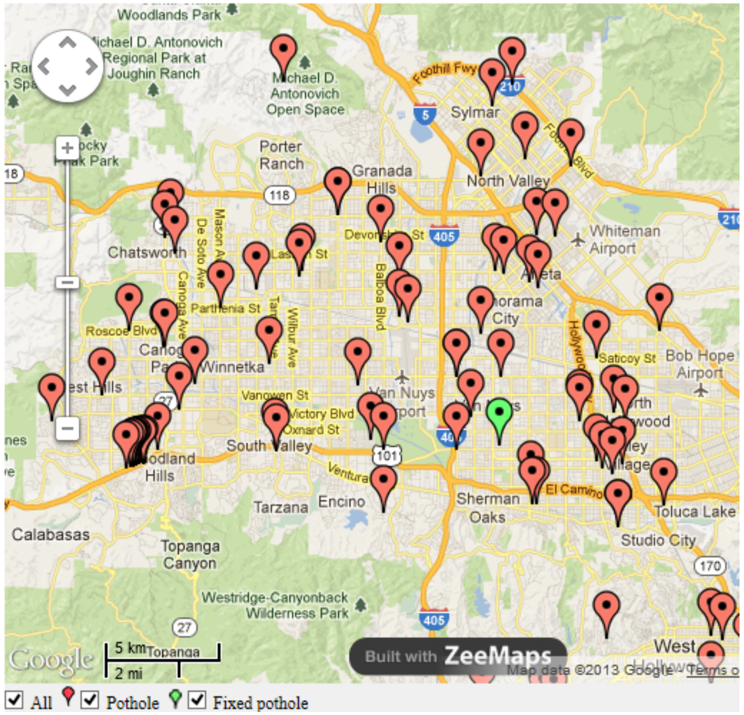
ISR 2: Mode Key

```
mode++;
mode = mode %
   NUM_MODES;
return from interrupt;
```

ISR 3: Time of Day Timer

```
cur_time ++;
lcd_refresh--;
if (lcd_refresh==0) {
 convert tenth_miles
    and display
 convert speed
    and display
  if (mode == 0)
    convert cur_time
      and display
  else
    convert avg_speed
      and display
  lcd_refresh =
     LCD_REF_PERIOD
}
```

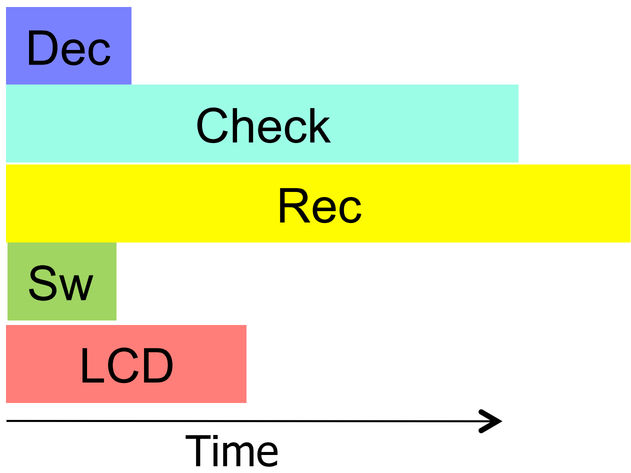
A More Complex Application



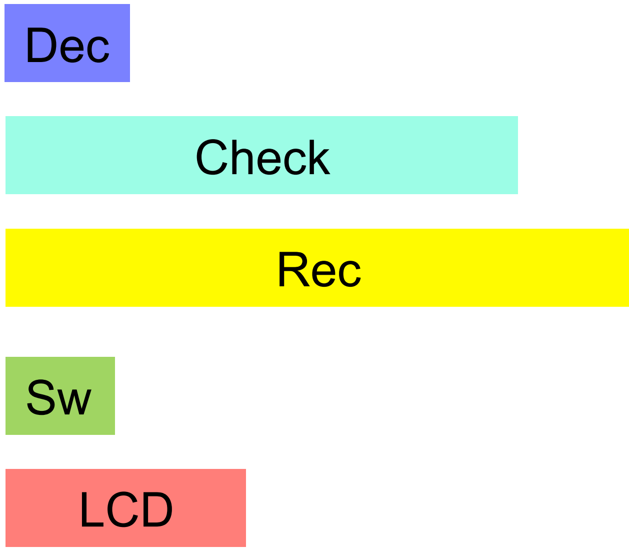
- GPS-based Pothole Alarm and Moving Map
  - Sounds alarm when approaching a pothole
  - Display's vehicle position on LCD
  - Also logs driver's position information
  - Hardware: GPS, user switches, speaker, LCD, flash memory

Application Software Tasks

- Dec: Decode GPS sentence to find current vehicle position.
- Check: Check to see if approaching any pothole locations. Takes longer as the number of potholes in database increases.
- Rec: Record position to flash memory. Takes a long time if erasing a block.
- Sw: Read user input switches. Run 10 times per second
- LCD: Update LCD with map. Run 4 times per second

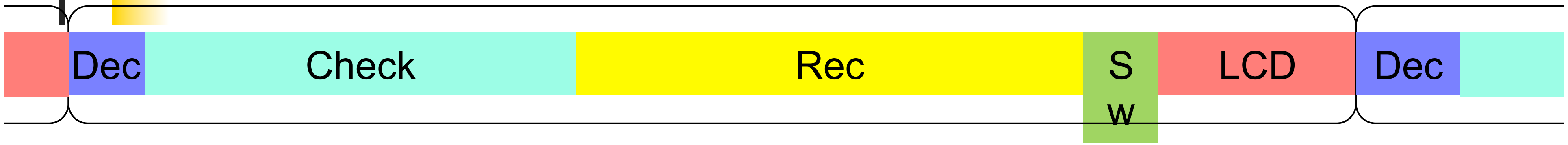


How do we schedule these tasks?



- Task scheduling: Deciding which task should be run now
- Two fundamental questions
  - **Do we run tasks in the same order every time?**
    - Yes: Static schedule (cyclic executive, round-robin)
    - No: Dynamic, prioritized schedule
  - **Can one task preempt another, or must it wait for completion?**
    - Yes: Preemptive
    - No: Non-preemptive (cooperative, run-to-completion)

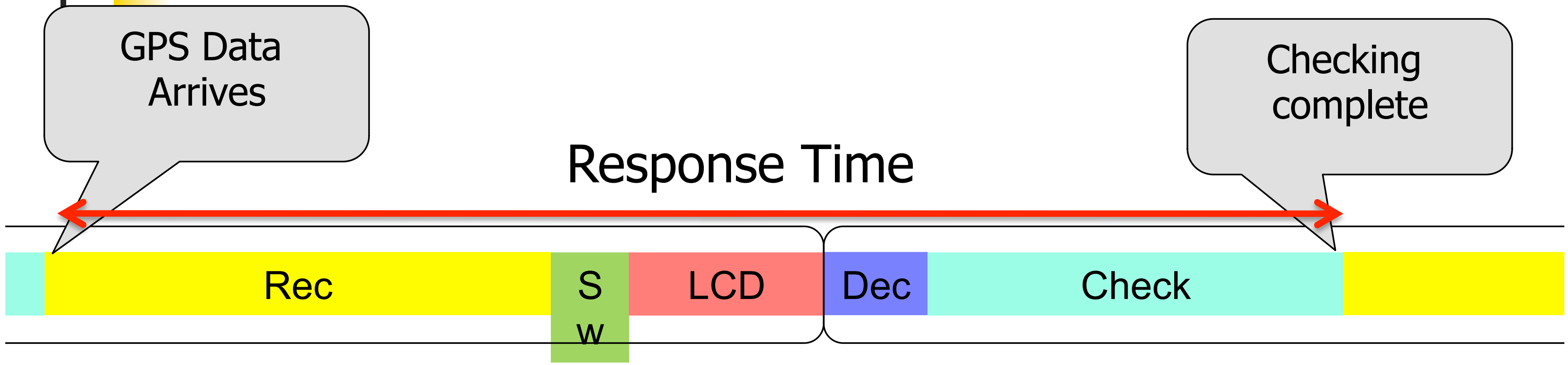
Static Schedule (Cyclic Executive)



- Pros
  - Very simple
- Cons
  - Always run the same schedule, regardless of changing conditions and relative importance of tasks.
  - All tasks run at same rate. Changing rates requires adding extra calls to the function.
  - Maximum delay is sum of all task run times. Polling/execution rate is 1/ maximum delay.

```
while (1){
        Dec();
        Check();
        Rec();
        Sw();
        LCD();
}
```

Static Schedule Example



- What if we receive GPS position right after Rec starts running?
- Delays
  - Have to wait for Rec, Sw, LCD before we start decoding position with Dec.
  - Have to wait for Rec, Sw, LCD, Dec, Check before we know if we are approaching a pothole!

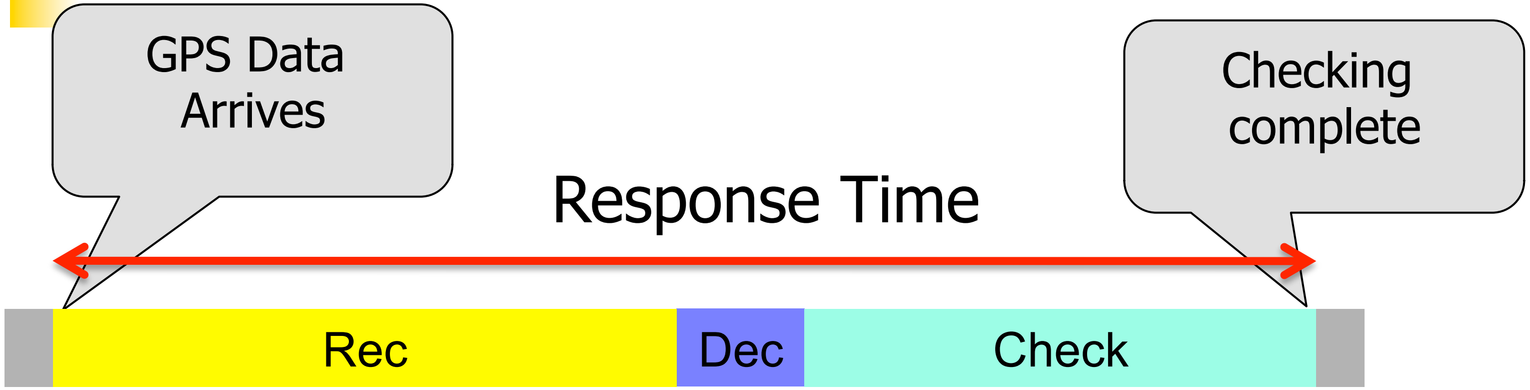
Dynamic Scheduling

- Allow schedule to be computed on-the-fly
  - Based on importance or something else
  - Simplifies creating multi-rate systems

- Schedule based on importance
  - Prioritization means that less important tasks don't delay more important ones

- How often do we decide what to run?
  - Coarse grain – After a task finishes. Called *Run-to-Completion* (RTC) or non-preemptive
  - Fine grain – Any time. Called *Preemptive*, since one task can preempt another.

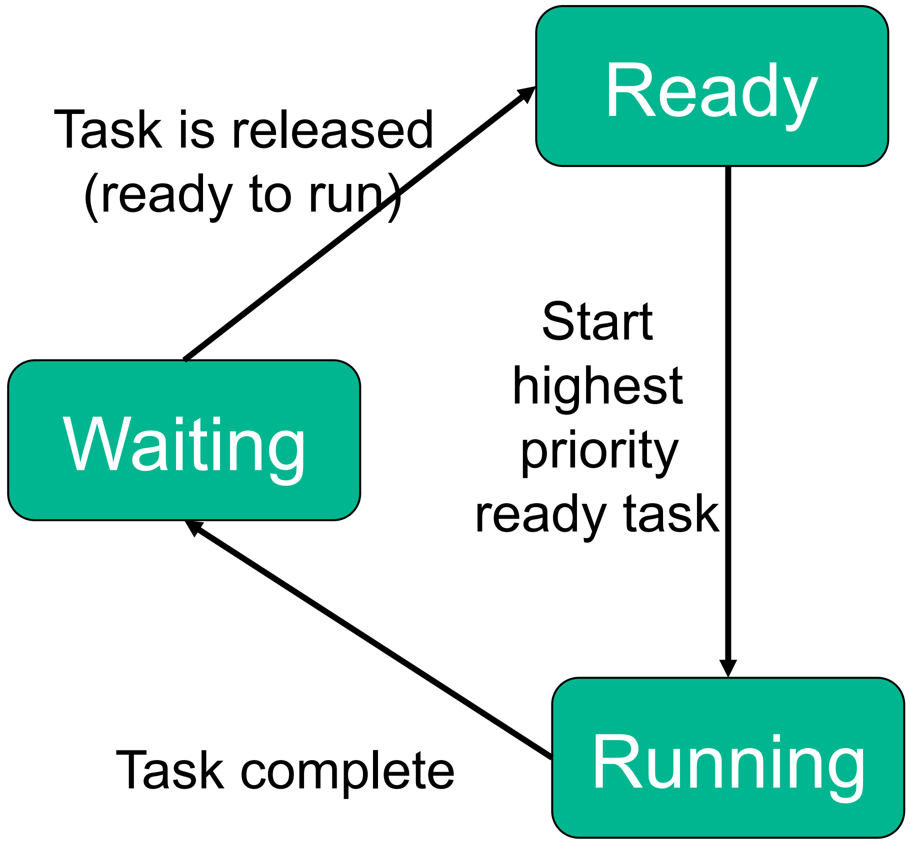
Dynamic RTC Schedule



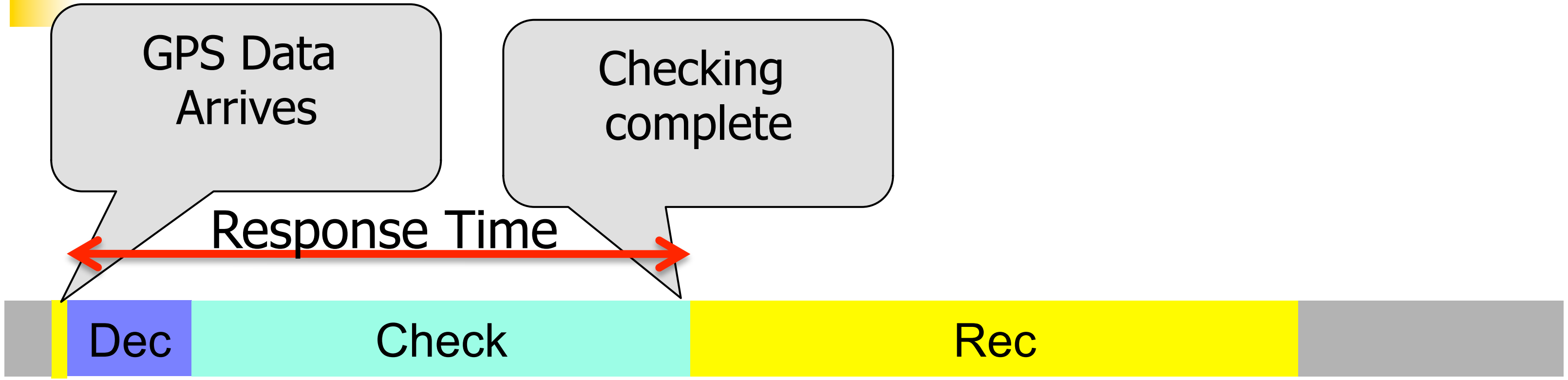
- What if we receive GPS position right after Rec starts running?
- Delays
  - Have to wait for Rec to finish before we start decoding position with Dec.
  - Have to wait for Rec, Dec, Check before we know if we are approaching a pothole

Task State and Scheduling Rules

- Scheduler chooses among *Ready* tasks for execution based on priority
- Scheduling Rules
  - If no task is running, scheduler starts the highest priority ready task
  - Once started, a task runs until it completes
  - Tasks then enter waiting state until triggered or released again

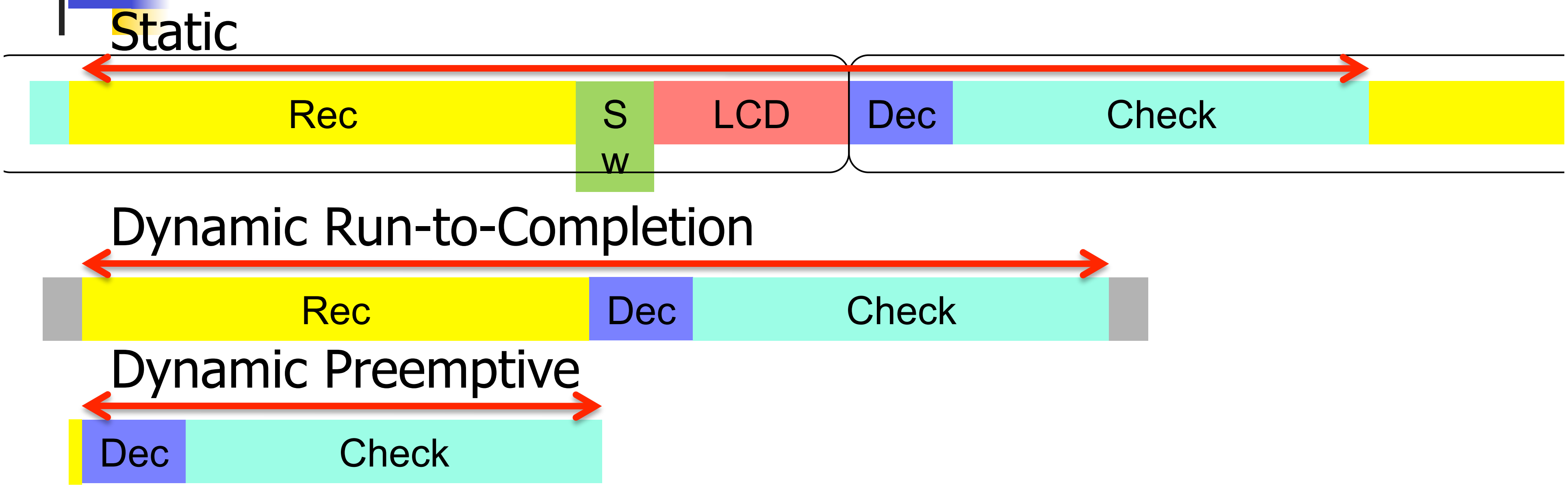


Dynamic Preemptive Schedule



- What if we receive GPS position right after Rec starts running?
- Delays
  - Scheduler switches out Rec so we can start decoding position with Dec immediately
  - Have to wait for Dec, Check to complete before we know if we are approaching a pothole

Comparison of Response Times



- Pros
  - Preemption offers best response time
    - Can do more processing (support more potholes, or higher vehicle speed)
    - Or can lower processor speed, saving money, power
- Cons
  - Requires more complicated programming, more memory
  - Introduces vulnerability to data race conditions

Common Schedulers

- Cyclic executive - non-preemptive and static
- Run-to-completion - non-preemptive and dynamic
- Preemptive and dynamic

Cyclic Executive with Interrupts

- Two priority levels
  - **main** code – background
  - Interrupts – foreground
- Example of a foreground / background system
- Interrupt routines run in foreground (high priority)
  - Run when triggered
  - Handle most urgent work
  - Set flags to request processing by main loop
- Main user code runs in background
  - Uses "round-robin" approach to pick tasks, takes turns
  - Tasks do not preempt each other

```
BOOL DeviceARequest,
DeviceBRequest, DeviceCRequest;
void interrupt HandleDeviceA(){
  /* do A's urgent work */
  ...
  DeviceARequest = TRUE;
}
void main(void) {
  while (TRUE) {
    if (DeviceARequest) {
      FinishDeviceA();
    }
    if (DeviceBRequest) {
      FinishDeviceB();
    }
    if (DeviceCRequest) {
      FinishDeviceC();
    }
  }
}
```

Run-To-Completion Scheduler

- Use a ***scheduler*** function to run task functions at the right rates
  - Table stores information per task
    - Period: How many ticks between each task release
    - Release Time: how long until task is ready to run
    - ReadyToRun: task is ready to run immediately
  - Scheduler runs forever, examining schedule table which indicates tasks which are ready to run (have been "released")
  - A periodic timer interrupt triggers an ISR, which updates the schedule table
    - Decrements "time until next release"
    - If this time reaches 0, set the task's Run flag and reload its time with the period
- Follows a "run-to-completion" model
  - A task's execution is ***not interleaved*** with any other task
  - Only ISRs can interrupt a task
  - After ISR completes, the previously-running task resumes
- Priority is typically static, so can use a table with highest priority tasks first for a fast, simple scheduler implementation.

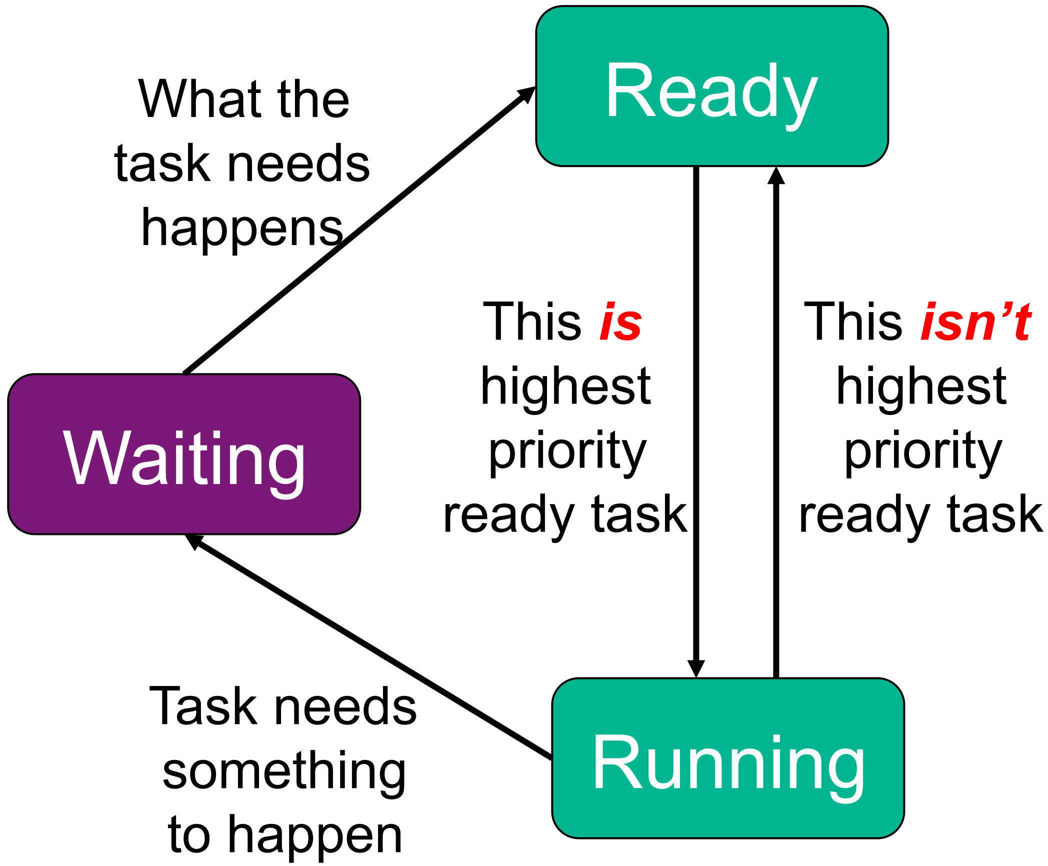
Preemptive Scheduler

- Task functions need not run to completion, but can be interleaved with each other
  - Simplifies writing software
  - Improves response time
  - Introduces new potential problems

- Worst case response time for highest priority task does not depend on other tasks, only ISRs and scheduler
  - Lower priority tasks depend only on higher priority tasks

Task State and Scheduling Rules

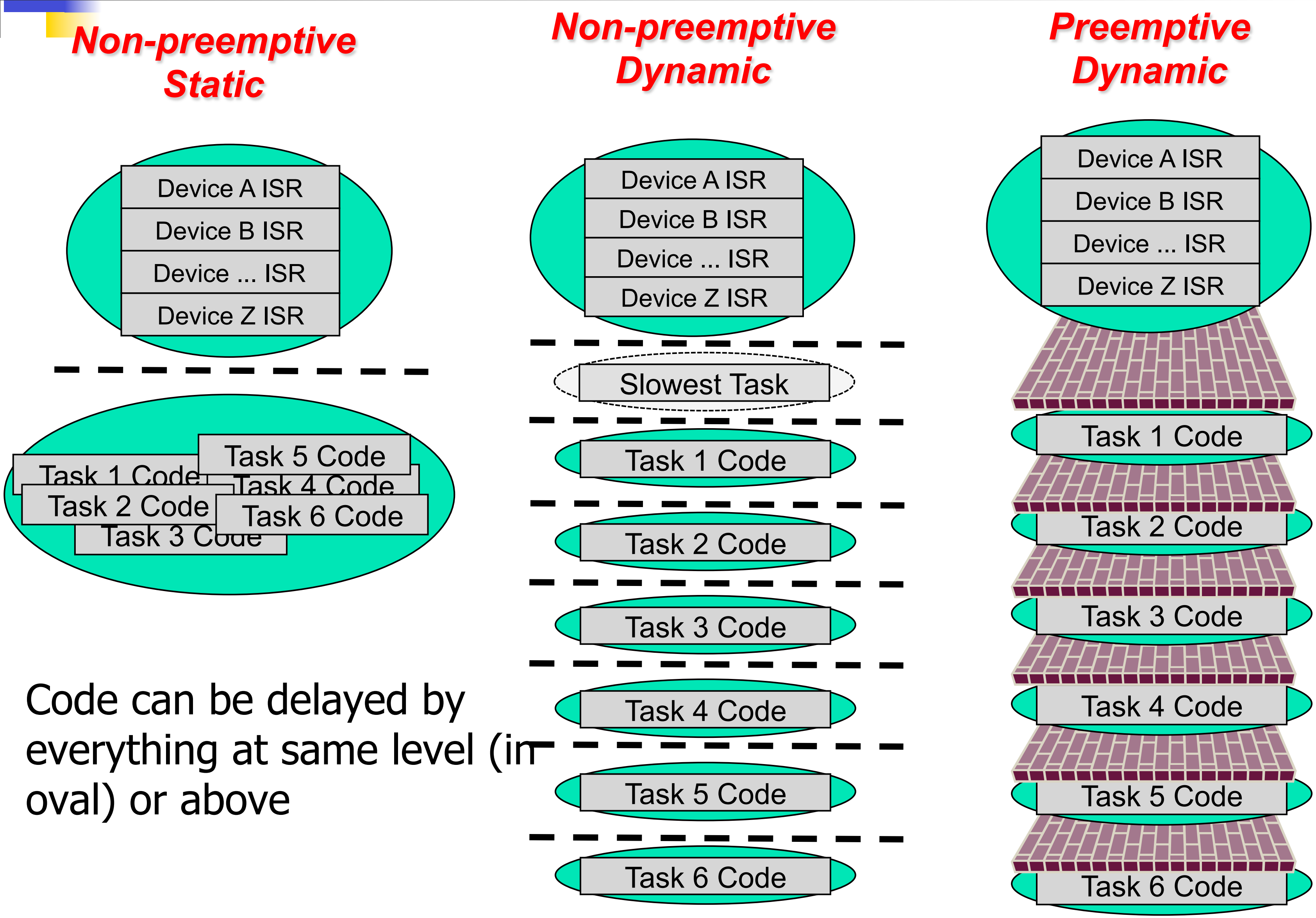
- Scheduler chooses among *Ready* tasks for execution based on priority
- Scheduling Rules
  - A task's activities may lead it to ***waiting (blocked)***
  - A ***waiting*** task never gets the CPU. It must be signaled by an ISR or another task.
  - Only the scheduler moves tasks between ***ready*** and ***running***



What's an RTOS?

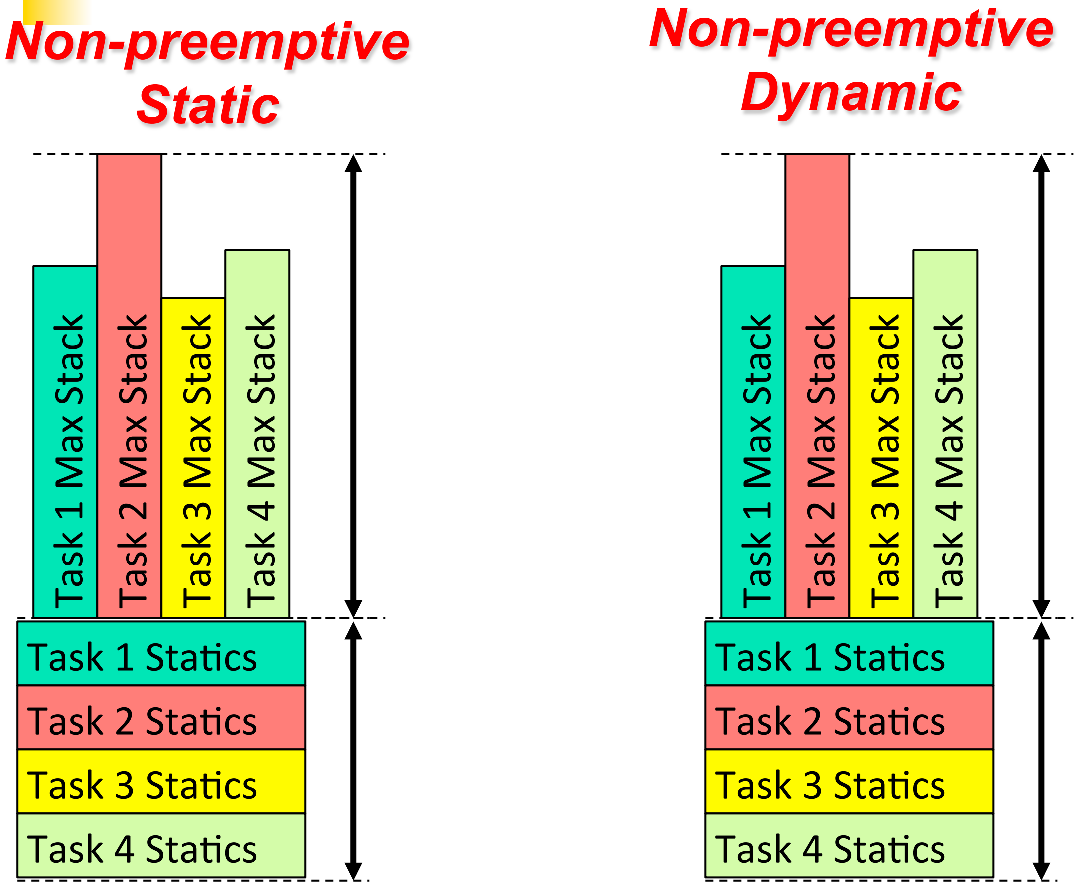
- What does Real-Time mean?
  - Can calculate and guarantee the *maximum response time* for each task and interrupt service routine
  - This "bounding" of response times allows use in hard-real-time systems (which have deadlines which must be met)
- What's in the RTOS
  - Task Scheduler
    - Preemptive, prioritized to minimize response times
    - Interrupt support
  - Core Integrated RTOS services
    - Inter-process communication and synchronization (safe data sharing)
    - Time management
  - Optional Integrated RTOS services
    - *I/O abstractions?*
    - *memory management?*
    - *file system?*
    - *networking support?*
    - *GUI??*

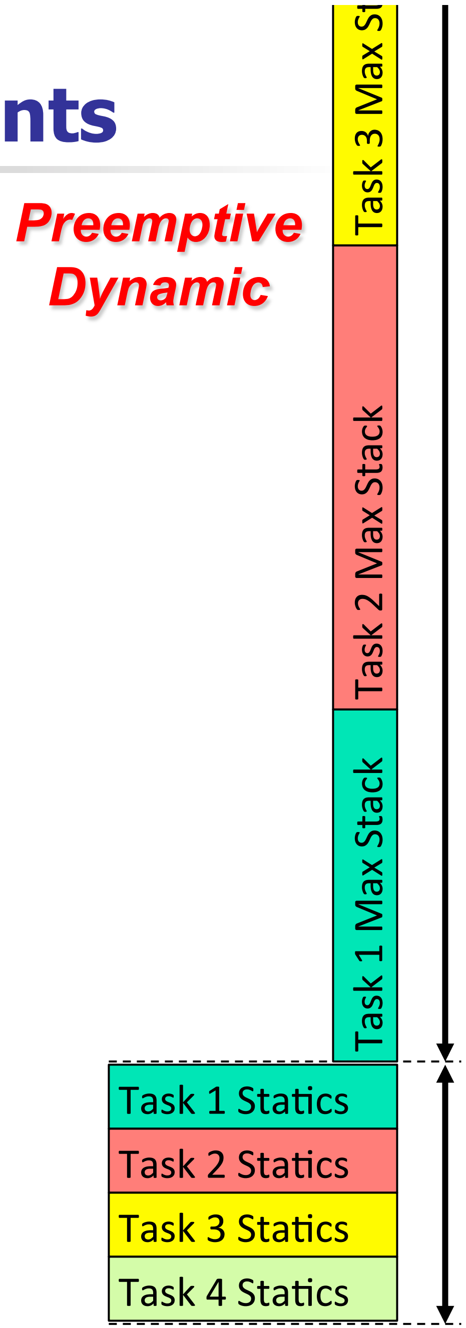
Comparison of Timing Dependence



- Code can be delayed by everything at same level (in oval) or above

Comparison of RAM Requirements





- Preemption requires space for each stack*
- Need space for all static variables (including globals)

**except for certain special cases*

References

- Slides adopted from Arm Teaching Kits