EECS 3215

“Software Design”

James Andrew Smith, PhD, PEng
York University

The Architecture for the Digital World®

ARM

Material Source

ARM University
Worldwide Education Program

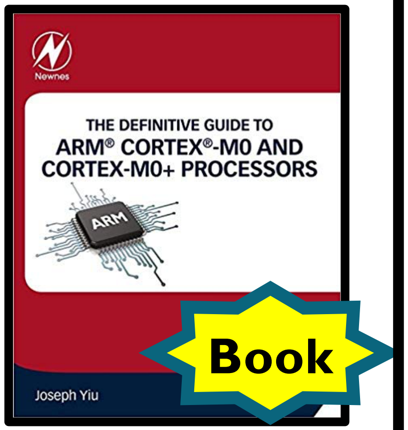
ARM-based Embedded Systems Design Lab-in-a-Box
Freescale Freedom Board Edition

Content is used with permission of Arm Limited.

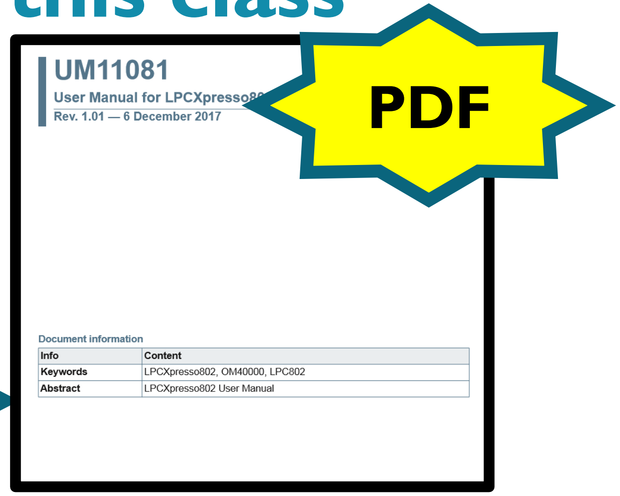
ARM-specific content comes courtesy of the ARM University Worldwide Education Program. Main author of this Arm University content is Dr. Alexander Dean, North Carolina State University, with updates and changes by J.A. Smith.

ARM

Material relevant to this class



https://amzn.to/2Qw4aye



https://bit.ly/2BjTe0z

https://bit.ly/2EkP4IX

https://bit.ly/2SIYSRR

https://bit.ly/2zXB31w

ARM

Overview

- Software Engineering for Embedded Systems
  - How do we develop working code quickly & properly?
- Concurrency (Operating Systems & Schedulers)
  - How do we make things happen at the right times?

SOFTWARE ENGINEERING FOR EMBEDDED SYSTEMS

Good Enough Software, Soon Enough

- How do we make software *correct enough* without going bankrupt?
  - Need to be able to develop (and test) software efficiently
- Follow a good plan
  - Start with customer requirements
  - Design architectures to define the building blocks of the systems (tasks, modules, etc.)
  - Add missing requirements
    - Fault detection, management and logging
    - Real-time issues
    - Compliance to a firmware standards manual
    - Fail-safes
  - Create detailed design
  - Implement the code, following a good development process
    - Perform frequent design and code reviews
    - Perform frequent testing (unit and system testing, preferably automated)
    - Use revision control to manage changes
  - Perform post-mortems to improve development process

ARM

What happens when the plan meets reality?

- We want a robust plan which considers likely risks
  - What if the code turns out to be a lot more complex than we expected?
  - What if there is a bug in our code (or a library)?
  - What if the system doesn't have enough memory or throughput?
  - What if the system is too expensive?
  - What if the lead developer quits?
  - What if the lead developer is incompetent, lazy, or both (and *won't* quit!)?
  - What if the rest of the team gets sick?
  - What if the customer adds new requirements?
  - What if the customer wants the product two months early?

- Successful software engineering depends on balancing many factors, many of which are non-technical!

ARM

Risk Reduction

- Plan to the work to accommodate risks
- Identify likely risks up front
  - Historical problem areas
  - New implementation technologies
  - New product features
  - New product line
- Severity of risk is a combination of likelihood and impact of failure

Software Lifecycle Concepts

- Coding is the most visible part of a software development process but is not the only one

- Before we can code, we must know
  - What must the code do? *Requirements specification*
  - How will the code be structured? *Design specification*
    - *(only at this point can we start writing code)*

- How will we know if the code works? *Test plan*
  - Best performed when defining requirements

- The software will likely be enhanced over time - *Extensive downstream modification and maintenance!*
  - Corrections, adaptations, enhancements & preventive maintenance

ARM

Product Development Lifecycle

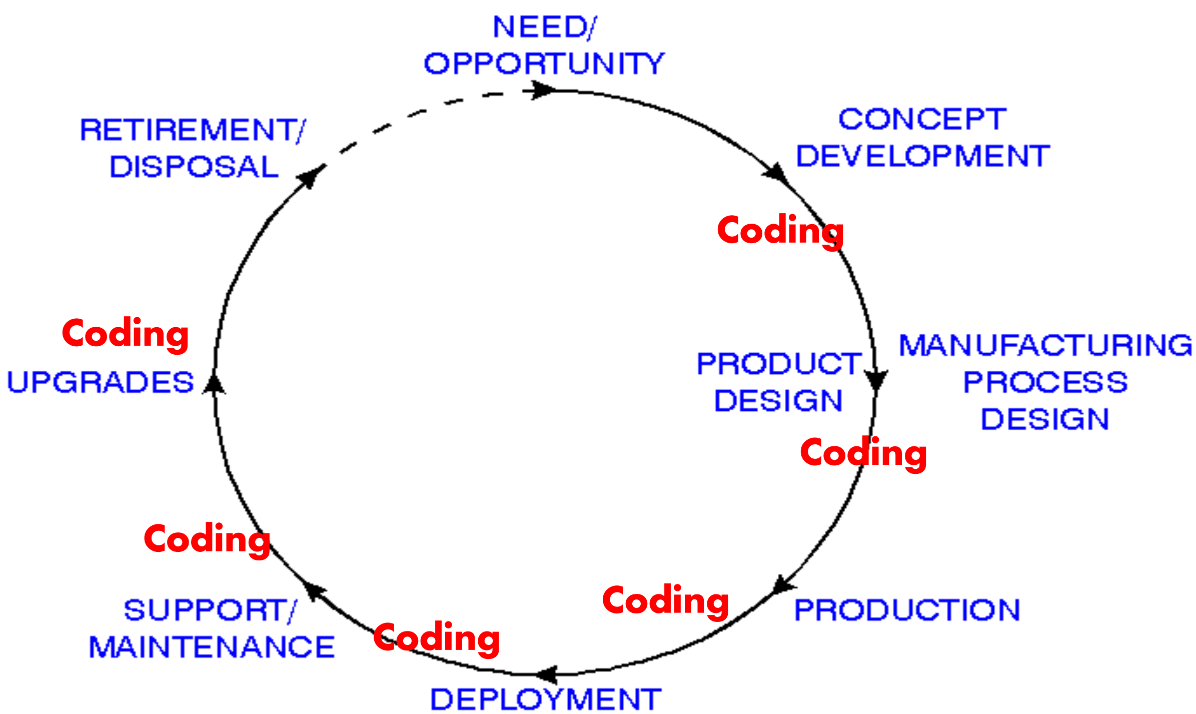


Diagram: Phil Koopman, Carnegie Mellon University

- With all this code development and modification, it is worth putting extra effort into simplifying code development activities: understanding, maintaining, enhancing, testing

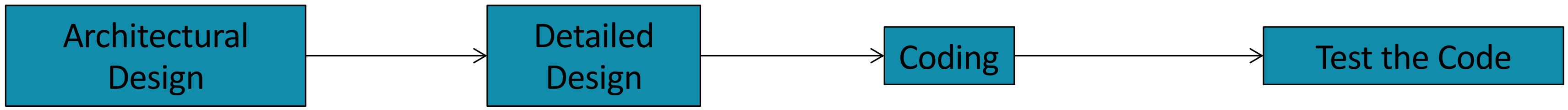
ARM

Requirements

- Ganssle's Reason #5 for why embedded projects fail: ***Vague Requirements***
- Types of requirements
  - Functional - what the system needs to do
  - Nonfunctional - emergent system behaviors such as response time, reliability, energy efficiency, safety, etc.
  - Constraints - limit design choices
- Representations
  - Text – Liable to be incomplete, bloated, ambiguous, even contradictory
  - Diagrams (state charts, flow charts, message sequence charts)
    - Concise
    - Can often be used as design documents
- Traceability
  - Each requirement should be verifiable with a test
- Stability
  - Requirements churn leads to inefficiency and often "recency" problems (most recent requirement change is assumed to be most important)

Jack Ganssle's "top 10 reasons for failure": http://www.ganssle.com/articles/jackstoptenlist.htm

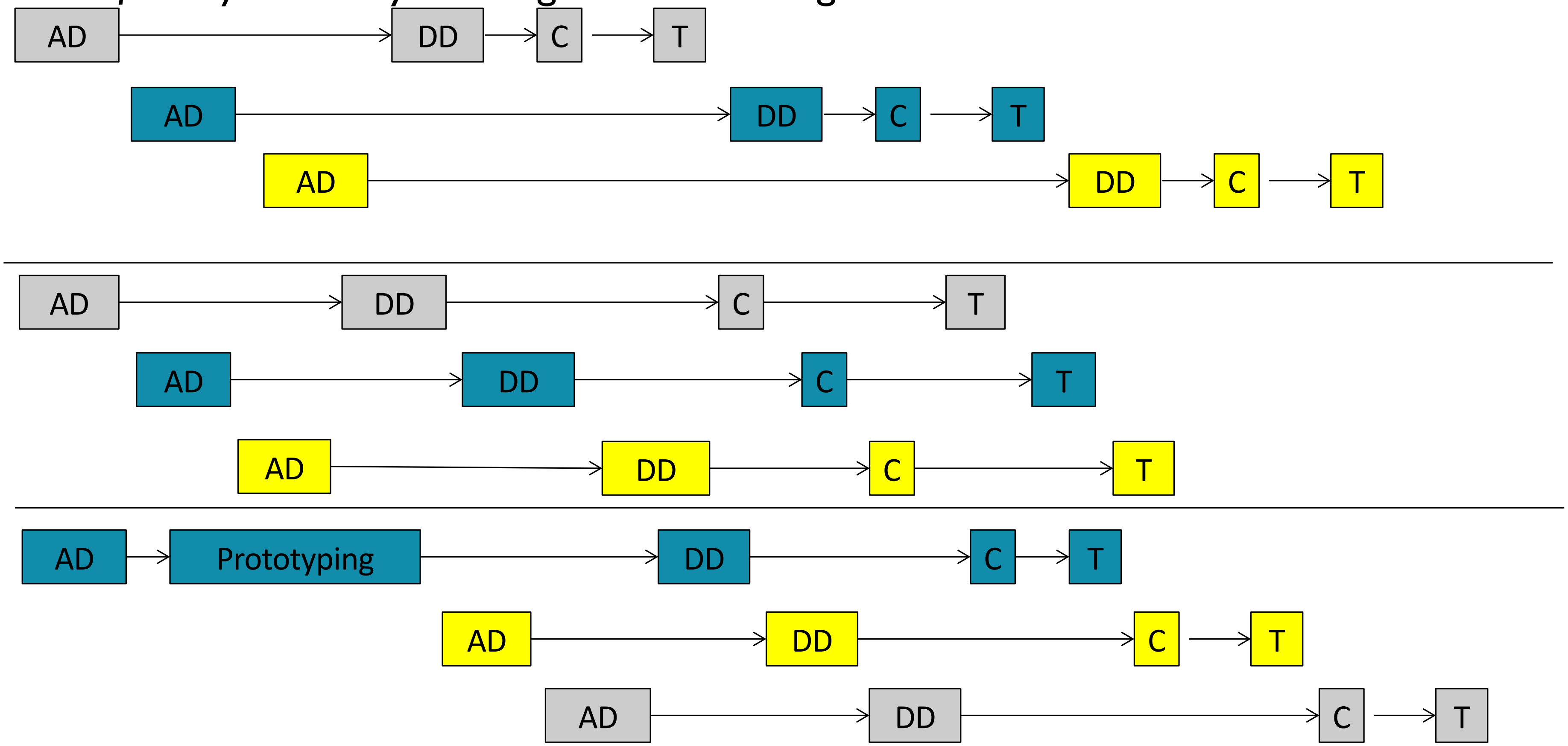
Design Before Coding



- Ganssle's reason #9: ***Starting coding too soon***
- Underestimating the complexity of the needed software is a very common risk
- Writing code locks you in to specific implementations
  - Starting too early may paint you into a corner
- Benefits of **designing** system before **coding**
  - Get early insight into system's complexity, allowing more accurate effort estimation and scheduling
  - Can use design diagrams rather than code to discuss what system should do and how. Ganssle's reason #7: ***Bad Science***
  - Can use design diagrams in documentation to simplify code maintenance and reduce risks of staff turnover

Design Before Coding

- *How much of the system* do you design before coding?



Development Models

- How do we schedule these pieces?

- **Consider** amount of **development risk**
  - New MCU?
  - Exceptional requirements (throughput, power, safety certification, etc.)
  - New product?
  - New customer?
  - Changing requirements?

- **Choose** model **based on risk**
  - Low risk: Can create detailed plan.
    - Big "up front" design
    - e.g. Waterfall
  - High risk: Use iterative / spiral method
    - Prototype high-risk parts first
    - e.g. Agile development

Waterfall (Idealized)

- Plan the work, and then work the plan
- BUFD: Big Up-Front Design
- Model implies that we and the customers know
  - All of the requirements up front
  - All of the interactions between components, etc.
  - How long it will take to write the software and debug it

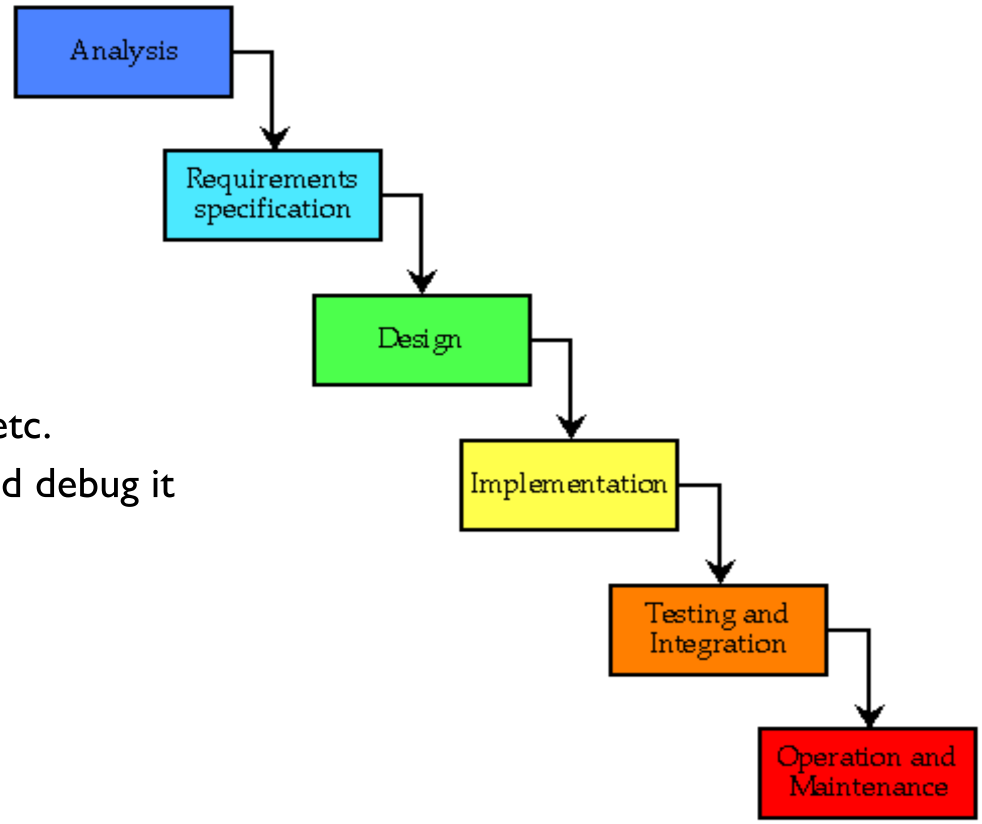


Diagram: Jon McCormack, Monash University

Waterfall (As Implemented)

- Reality: We are not omniscient, so there is plenty of backtracking

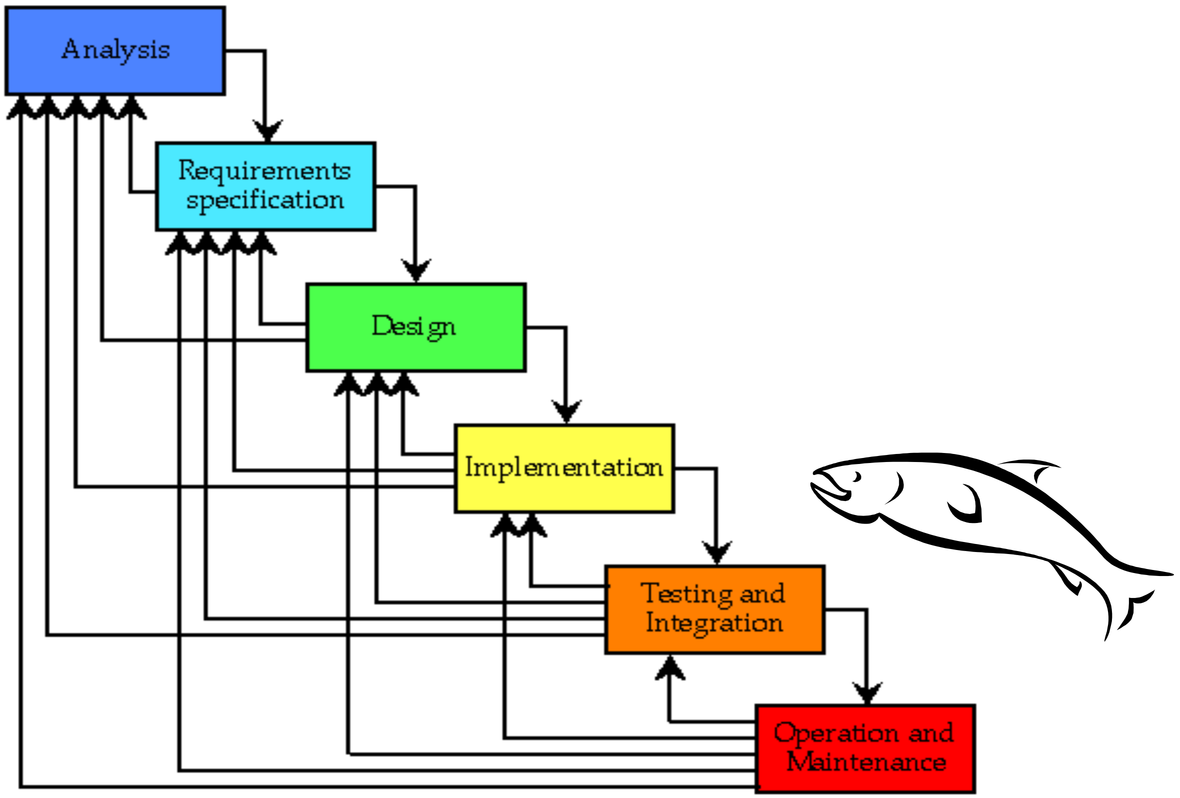


Diagram: Jon McCormack, Monash University

Agile Development: SCRUM

- http://www.mountaingoatsoftware.com/topics/scrum
- http://www.codeproject.com/KB/architecture/scrum.aspx

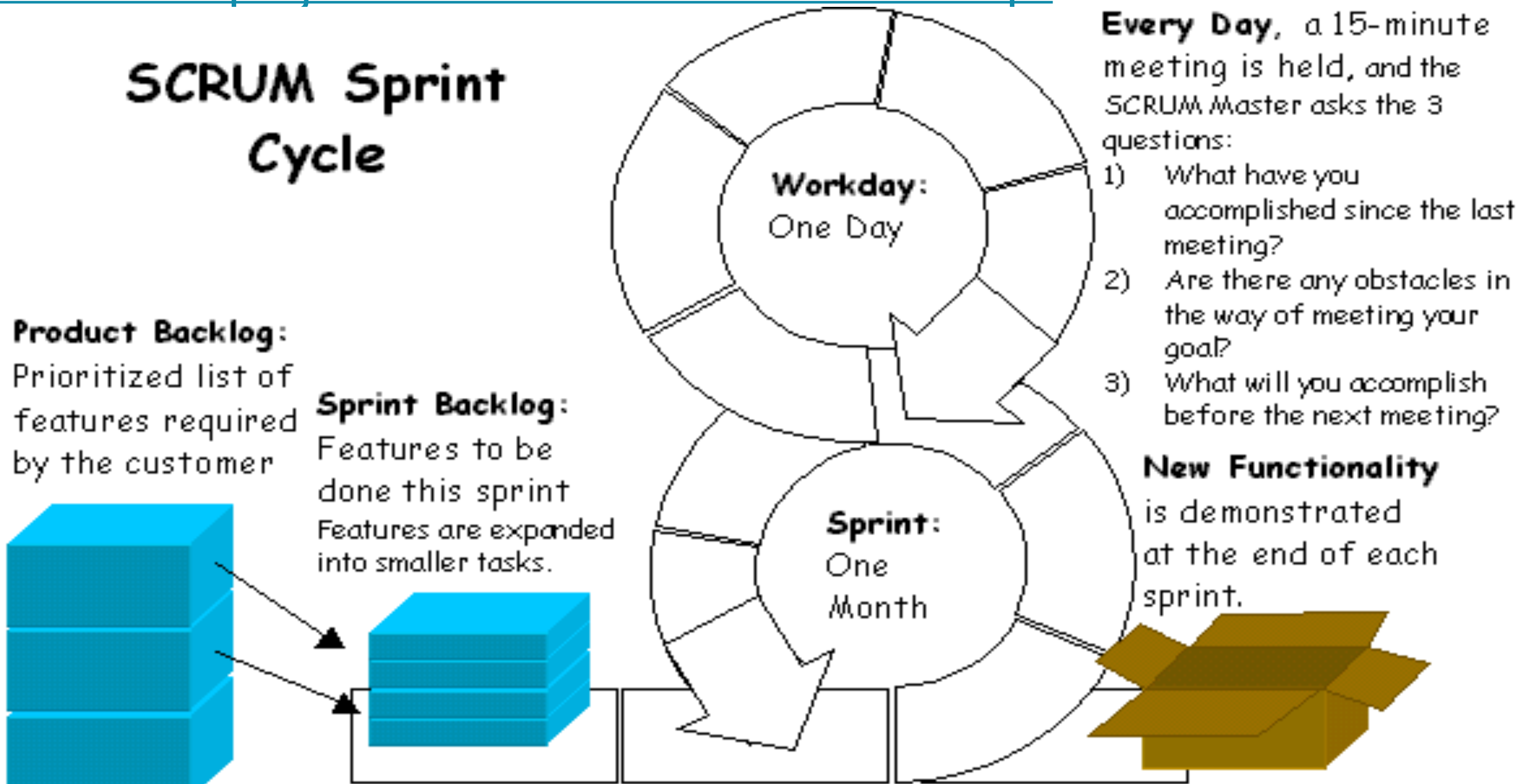
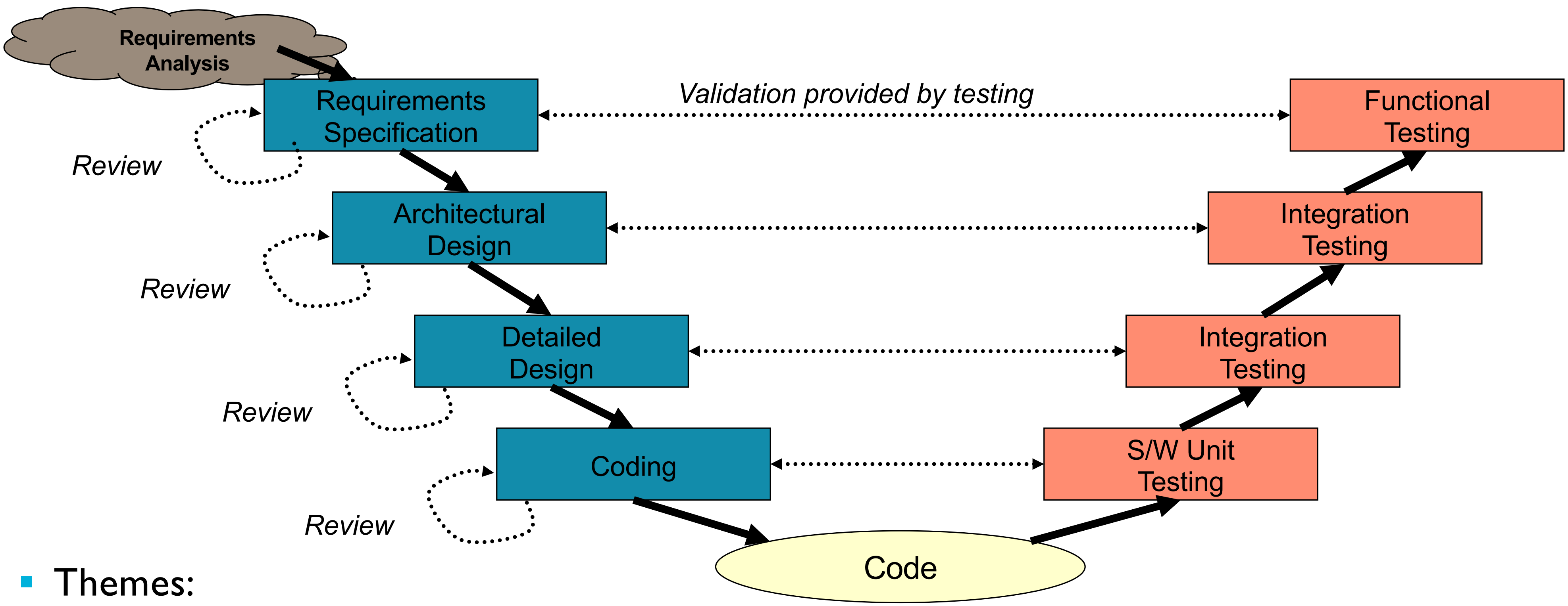


Diagram: Wikipedia

ARM

V Model Overview



- Themes:
  - Link front and back ends of life-cycle for efficiency
  - Provide "traceability" to ensure nothing falls through the cracks

1. Requirements Specification and Validation Plan

- Result of Requirements Analysis
- Should contain:
  - *Introduction* with goals and objectives of system
  - *Description of problem* to solve
  - *Functional description*
    - provides a "processing narrative" per function
    - lists and justifies design constraints
    - explains performance requirements
  - *Behavioral description* shows how system reacts to internal or external events and situations
    - State-based behavior
    - General control flow
    - General data flow
  - *Validation criteria*
    - tell us how we can decide that a system is acceptable. (*Are we done yet?*)
    - is the foundation for a validation test plan
  - *Bibliography and Appendix* refer to all documents related to project and provide supplementary information

ARM

2. Architectural (High-Level) Design

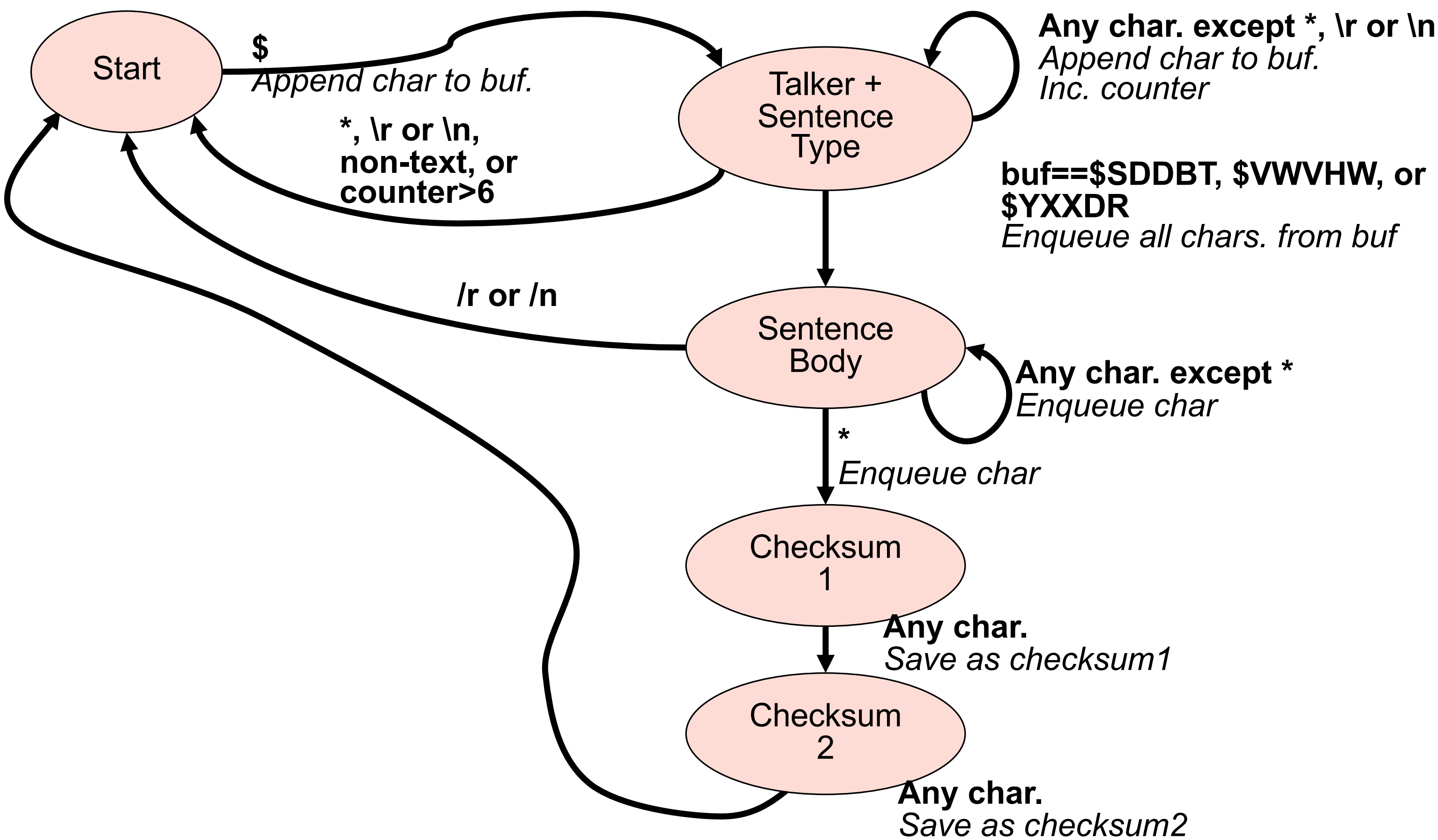
- Architecture defines the structure of the system
  - Components
  - Externally visible properties of components
  - Relationships among components
- Architecture is a representation which lets the designer…
  - Analyze the design's effectiveness in meeting requirements
  - Consider alternative architectures early
  - Reduce down-stream implementation risks
- Architecture matters because…
  - It's small and simple enough to fit into a single person's brain (as opposed to comprehending the entire program's source code)
  - It gives stakeholders a way to describe and therefore discuss the system

ARM

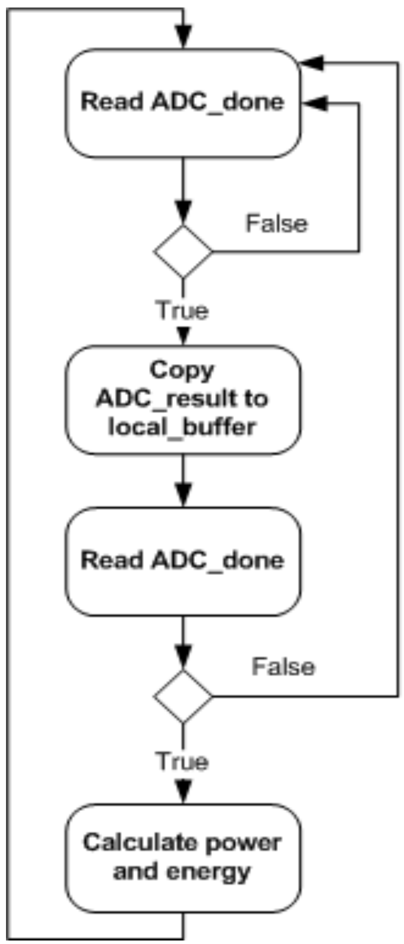
3. Detailed Design

- Describe aspects of how system behaves
  - Flow charts for control or data
  - State machine diagram
  - Event sequences
- Graphical representations very helpful
  - Can provide clear, single-page visualization of what system component should do

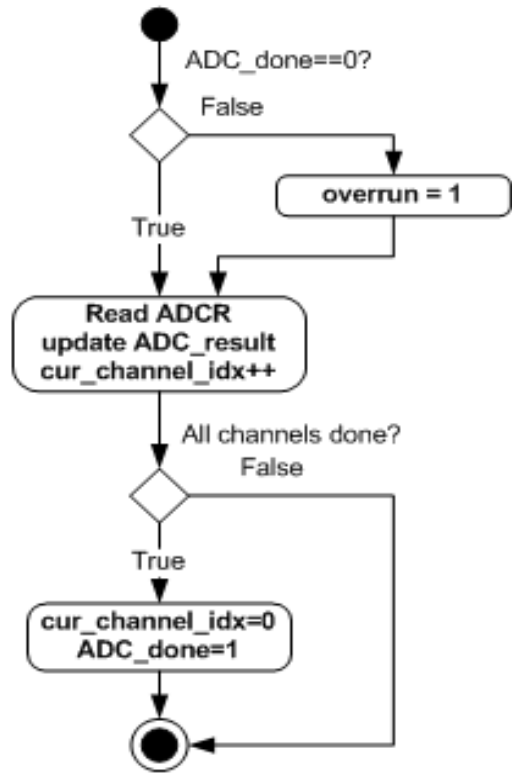
State Machine for Parsing NMEA-0183 (Serial Marine Comms)



Flowcharts

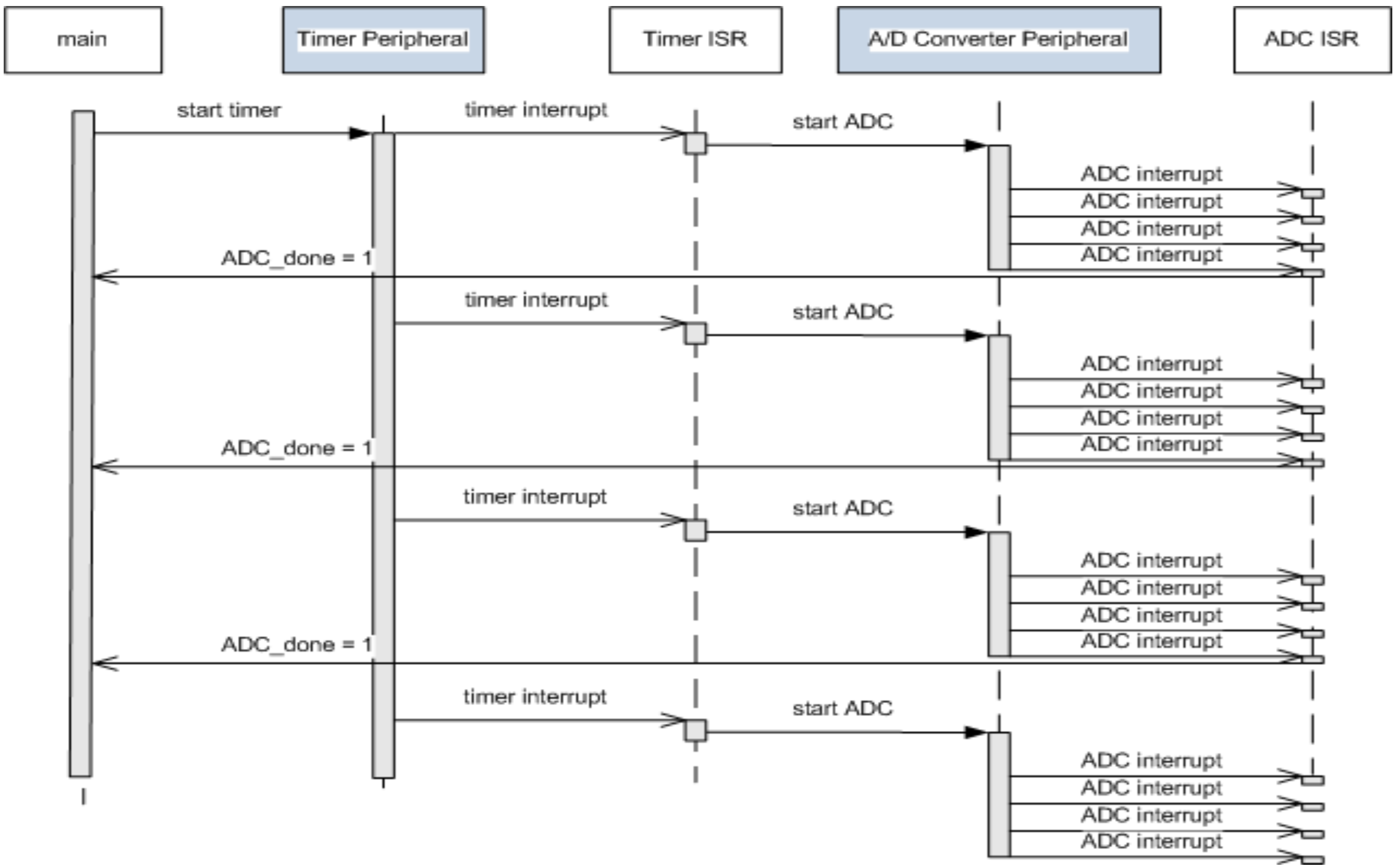


Read ADC_done
False
True
Copy
ADC_result to
local_buffer
Read ADC_done
False
True
Calculate power
and energy



ADC_done==0?
False
overrun = 1
True
Read ADCR
update ADC_result
cur_channel_idx++
All channels done?
False
True
cur_channel_idx=0
ADC_done=1

Sequence of Interactions between Components



4. Coding and Code Inspections

- Coding driven directly by Detailed Design Specification
- Use a version control system while developing the code
- Follow a coding standard
  - Eliminate stylistic variations which make understanding code more difficult
  - Avoid known questionable practices
  - Spell out best practices to make them easier to follow
- Perform code reviews
- Perform unit testing on modules as appropriate

Peer Code Review

- Inspect the code before testing it
- Extensive positive industry results from code inspections
  - IBM removed 82% of bugs
  - 9 hours saved by finding each defect
  - For AT&T quality rose by 1000% and productivity by 14%
- Finds bugs which testing often misses
  - 80% of the errors detected by HP's inspections were unlikely to be caught by testing
  - HP, Shell Research, Bell Northern, AT&T: inspections 20-30x more efficient than testing

5. Software Testing

- Testing IS NOT "the process of verifying the program works correctly"
  - The program probably won't work correctly in all possible cases
    - Professional programmers have 1-3 bugs per 100 lines of code after it is "done"
  - Testers shouldn't try to prove the program works correctly (impossible)
    - If you want and expect your program to work, you'll unconsciously miss failure because human beings are inherently biased
- The purpose of testing is to find problems quickly
  - Does the software violate the (performance) specifications?
  - Does the software violate unstated requirements?
- The purpose of finding problems is to fix the ones which matter
  - Fix the most important problems, as there isn't enough time to fix all of them
  - The *Pareto Principle* defines "the vital few, the trivial many"
    - Bugs are uneven in frequency – a vital few contribute the majority of the program failures. Fix these first.

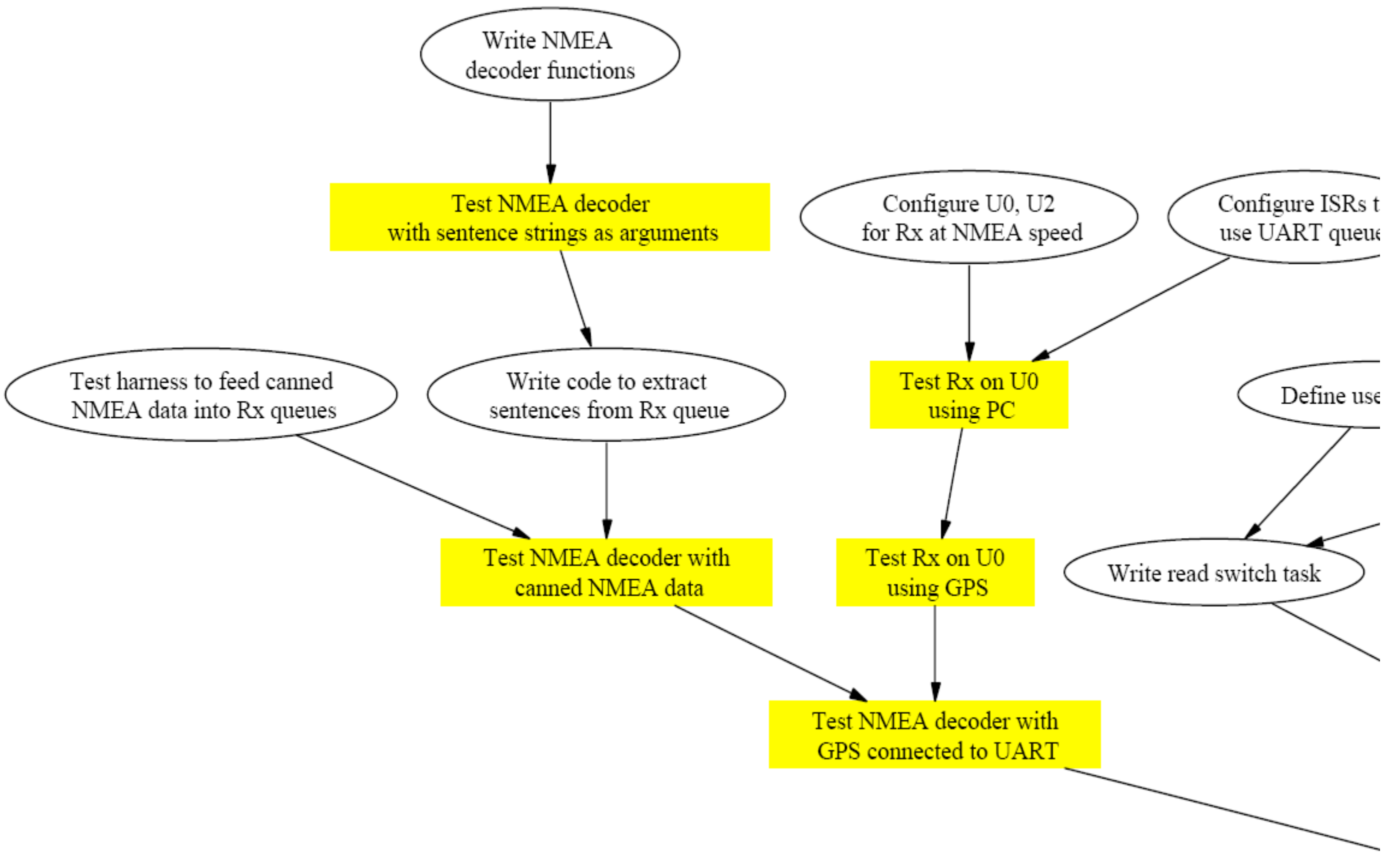
Approaches to Testing

- Incremental Testing
  - Code a function and then test it (*module/unit/element testing*)
  - Then test a few working functions together (*integration testing)*
    - Continue enlarging the scope of tests as you write new functions
  - Incremental testing requires extra code for the *test harness*
    - A *driver* function calls the function to be tested
    - A *stub* function might be needed to simulate a function called by the function under test, and which returns or modifies data.
    - The test harness can *automate* the testing of individual functions to detect later bugs
- Big Bang Testing
  - Code up all of the functions to create the system
  - Test the complete system
    - Plug and pray

Why Test Incrementally?

- Finding out what failed is much easier
  - With Big Bang, since no function has been thoroughly tested, most probably have bugs
  - Now the question is "Which bug in which module causes the failure I see?"
  - Errors in one module can make it difficult to test another module
    - Errors in fundamental modules (e.g. kernel) can appear as bugs in other many other dependent modules
- Less finger pointing = happier SW development team
  - It's clear who made the mistake, and it's clear who needs to fix it
- Better automation
  - Drivers and stubs initially require time to develop, but save time for future testing

Example Test Plan



6. Perform Project Retrospectives

- Goals – improve your engineering processes
  - Extract all useful information learned from the just-completed project – provide "virtual experience" to others
  - Provide positive non-confrontational feedback
  - Document problems and solutions clearly and concisely for future use

- Basic rule: problems need solutions

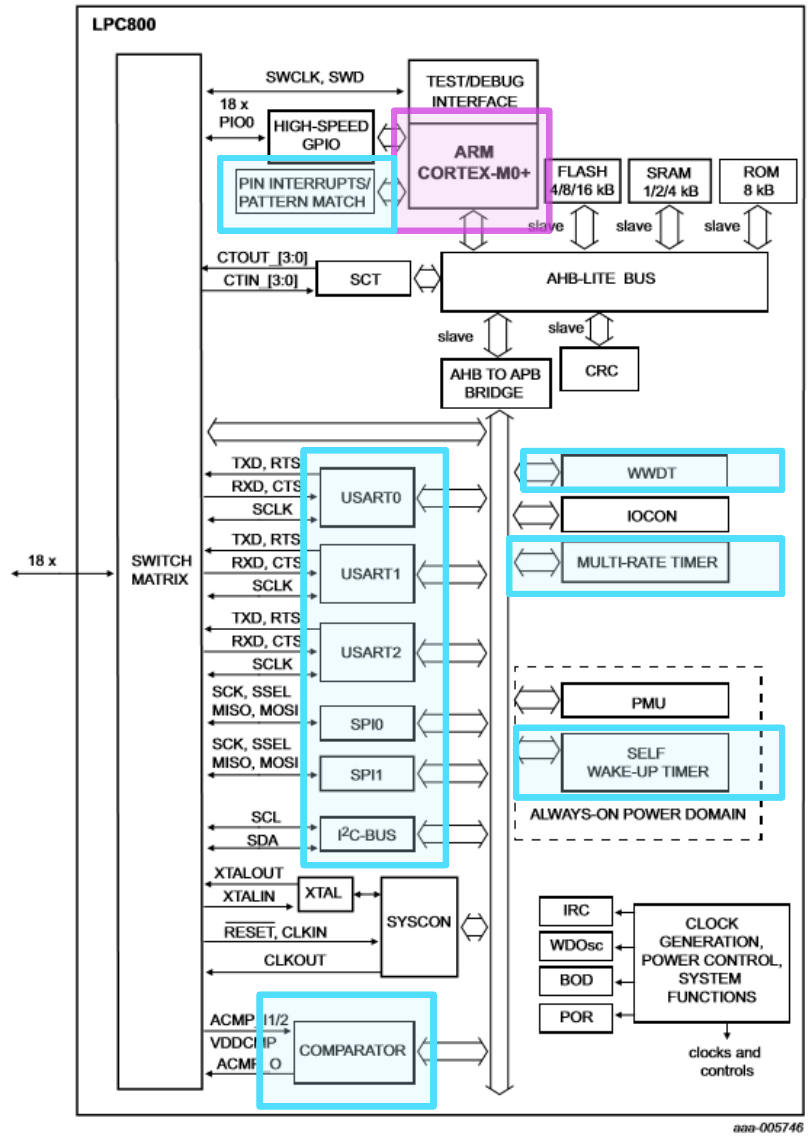
- Often small changes improve performance, but are easy to forget

ARM

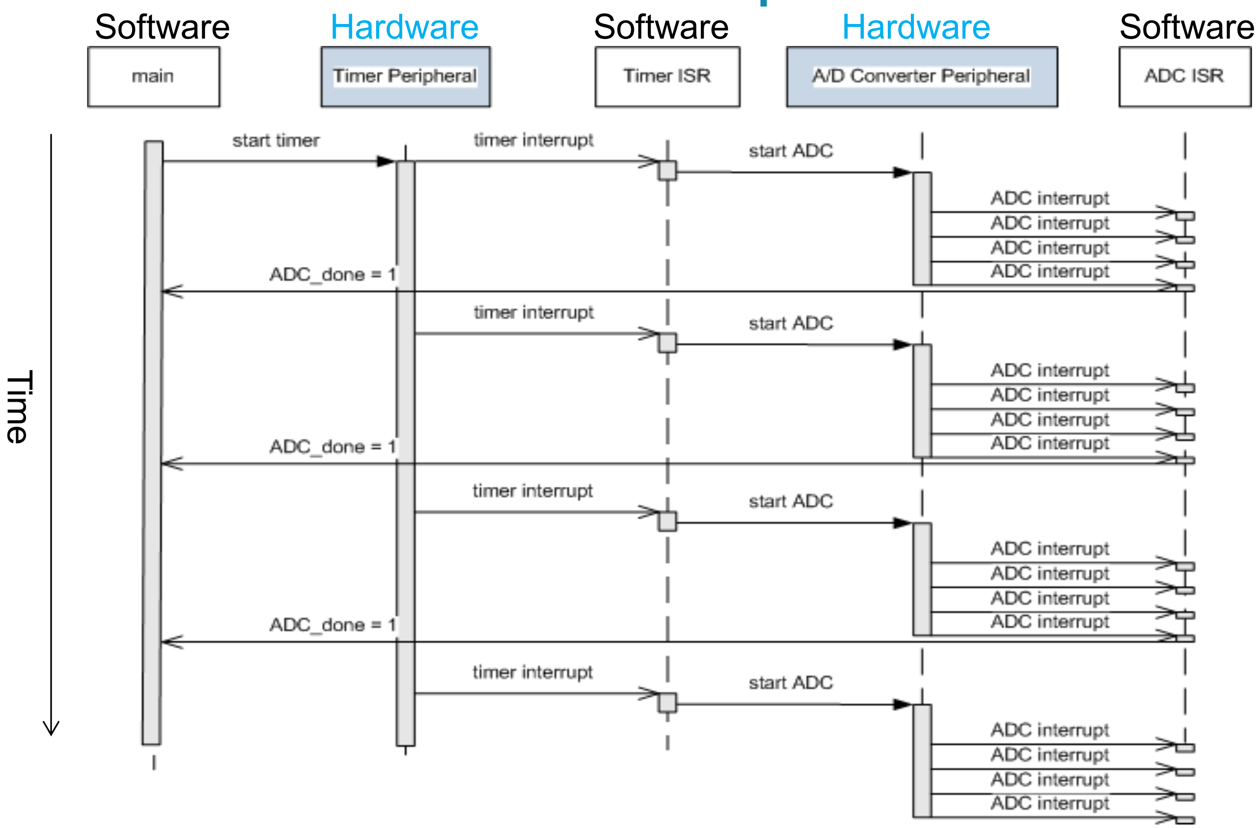
CONCURRENCY

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 (comparator)
  - Timers
  - Communications with other devices
  - Detecting external signal events
- Peripherals use ***interrupts*** to notify CPU of events



Concurrent Hardware & Software Operation



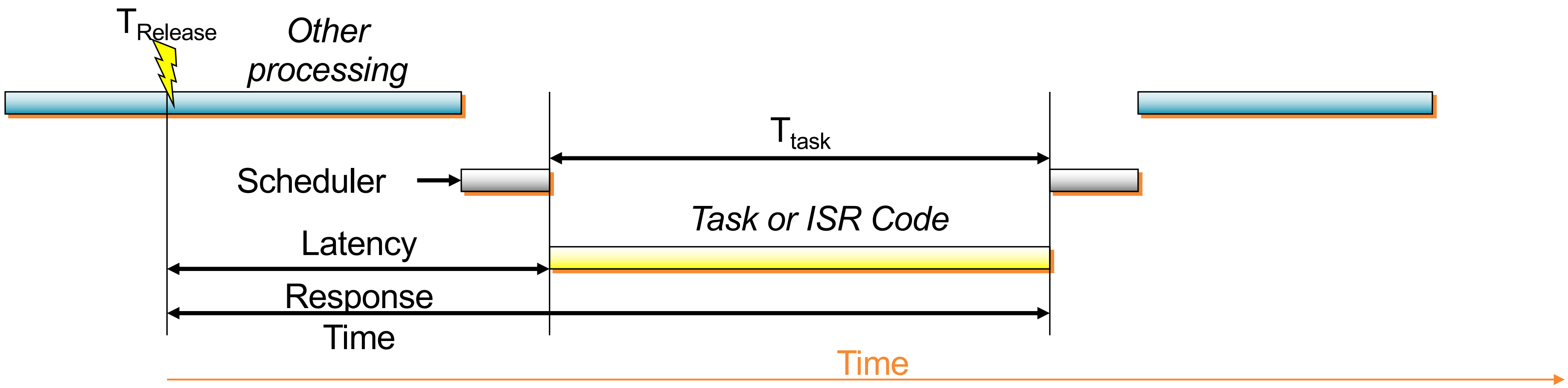
- Embedded systems rely on both MCU ***hardware peripherals*** and ***software*** to get everything done on time

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.

ARM

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

ARM

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

ARM

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
```

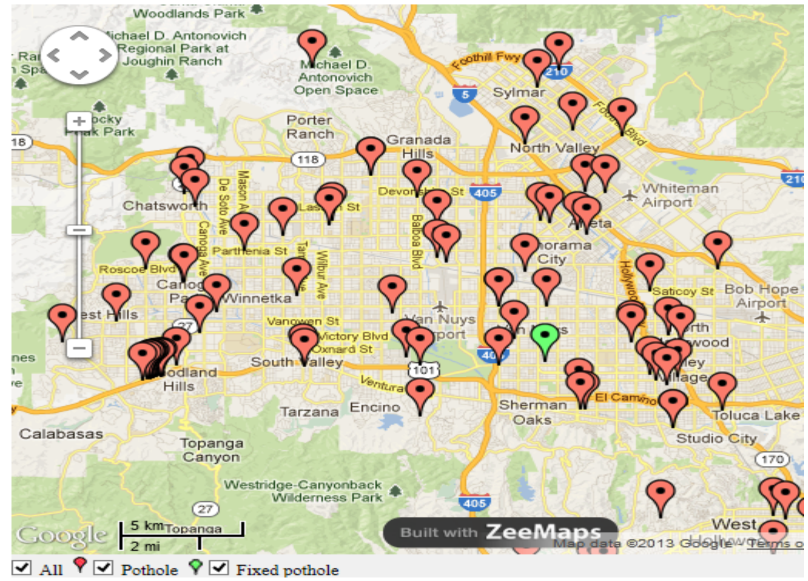
ISR 2: Mode Key

```
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

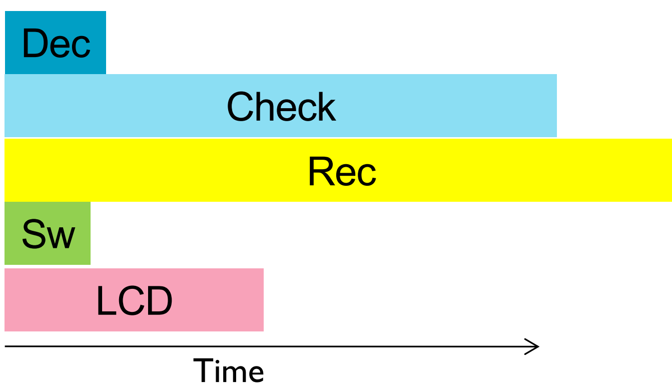


- GPS-based Pothole Alarm and Moving Map
  - Sounds alarm when approaching a pothole
  - Display’s vehicle position on LCD
  - Also logs driver’s position information
  - Hardware: GPS, user switches, speaker, LCD, flash memory

ARM

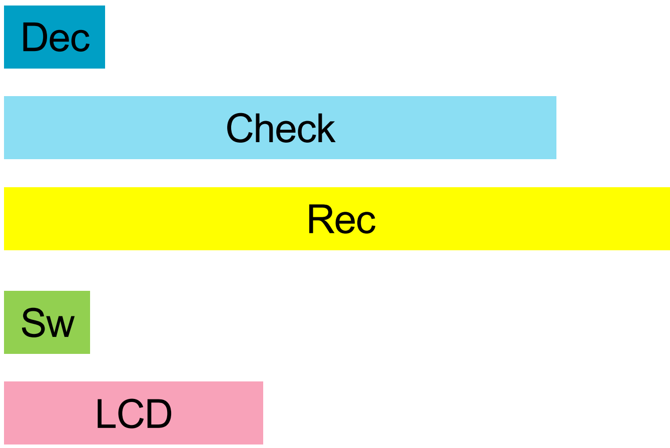
Application Software Tasks

- 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



ARM

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)

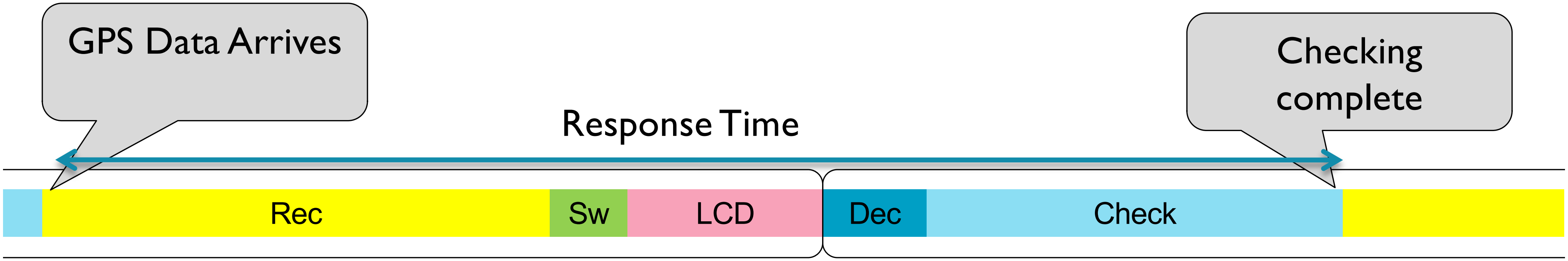
Static Schedule (Cyclic Executive)

- 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();
}
```


ARM

Static Schedule Example



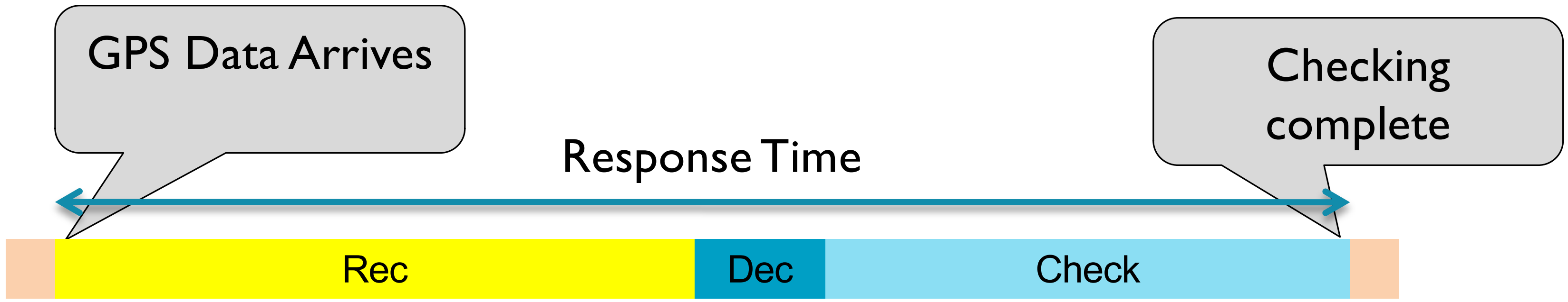
- What if we receive GPS position right after Rec starts running?
- Delays
  - Have to wait for Rec, Sw, LCD before we start decoding position with Dec.
  - Have to wait for Rec, Sw, LCD, Dec, Check before we know if we are approaching a pothole!

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.

ARM

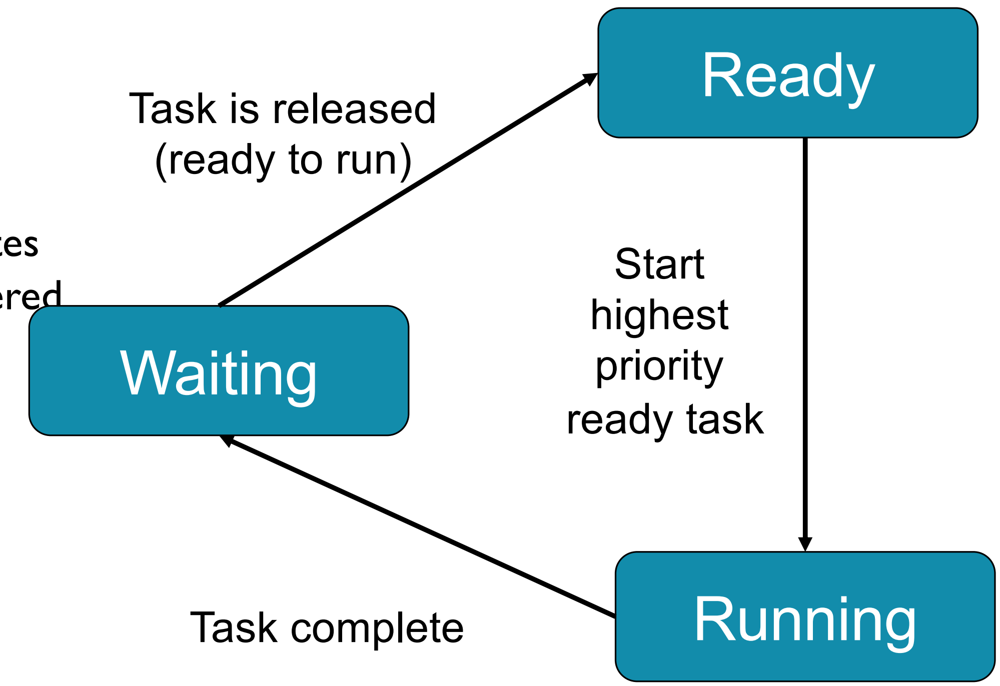
Dynamic Run to Completion (RTC) Schedule



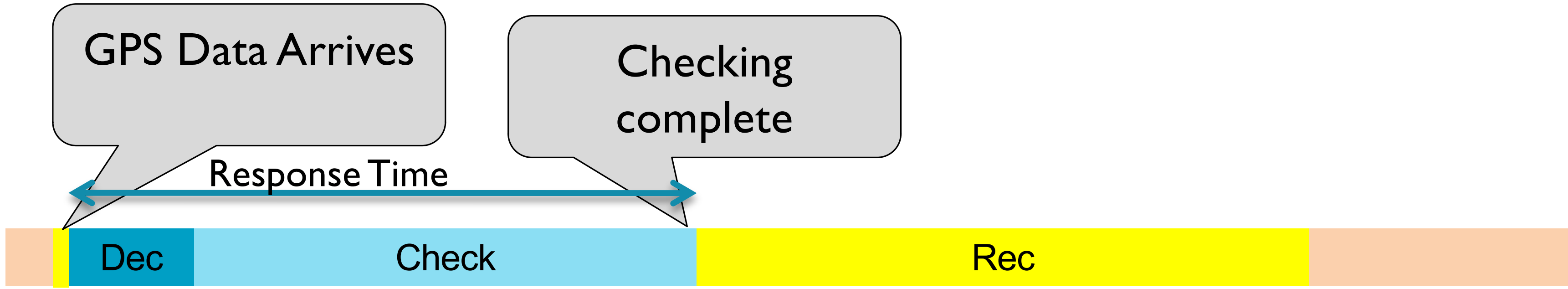
- RTC is "cooperative" / "non-preemptive"
- What if we receive GPS position right after Rec starts running?
- Delays
  - Have to wait for Rec to finish before we start decoding position with Dec.
  - Have to wait for Rec, Dec, Check before we know if we are approaching a pothole

RTC Task State and Scheduling Rules

- Run to Completion is *cooperative*.
- 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

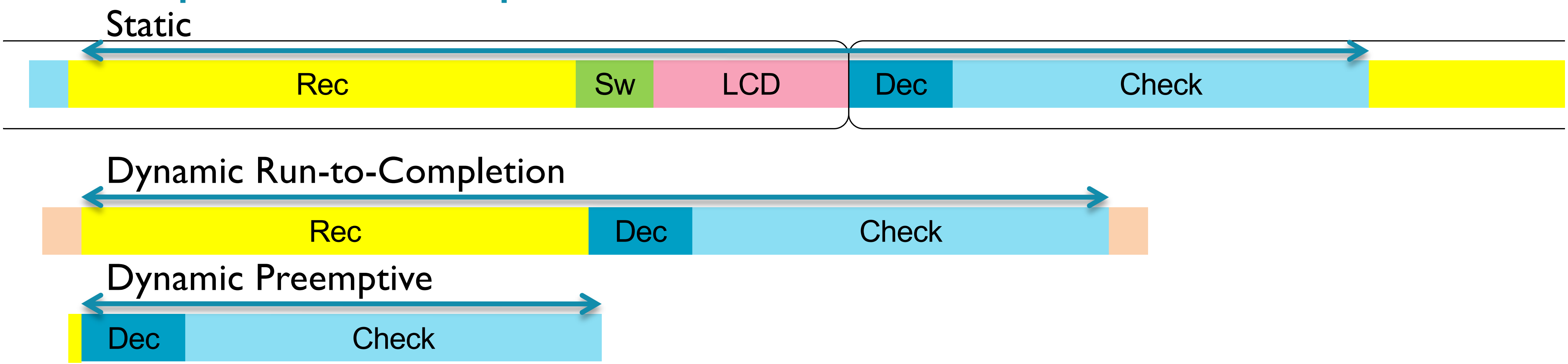


Dynamic Preemptive Schedule



- Preemption is "selfish" / *not cooperative*.
- 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

ARM

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();
    }
  }
}
```


ARM

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.

ARM

Preemptive Scheduler

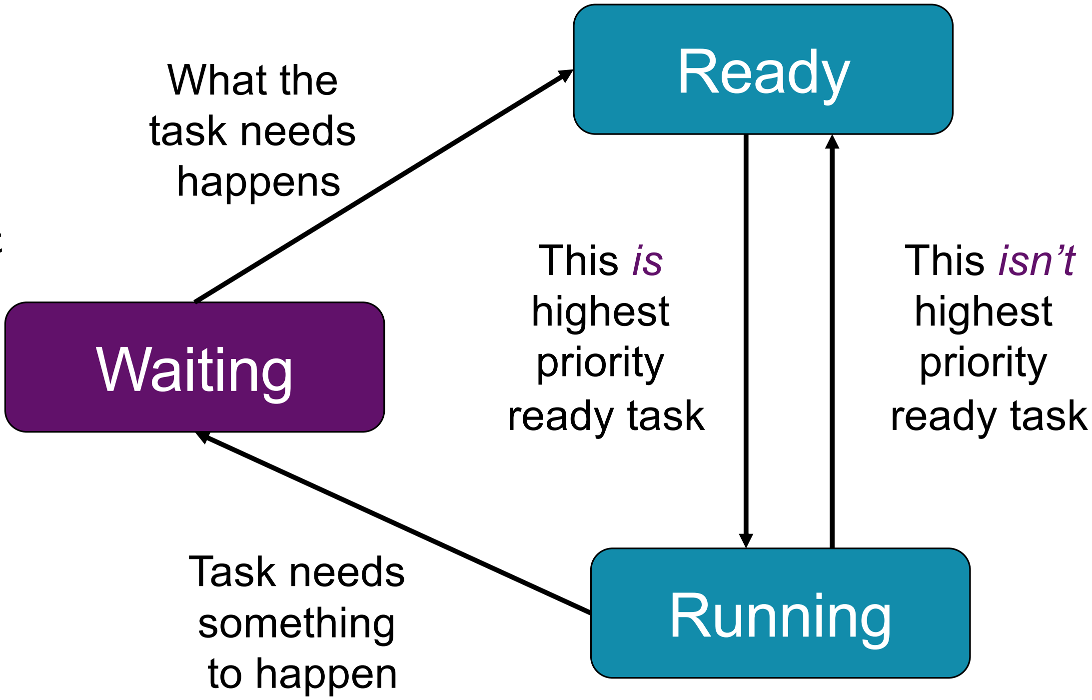
- Task functions need not run to completion, but can be interleaved with each other
  - Simplifies writing software
  - Improves response time
  - Introduces new potential problems

- 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?

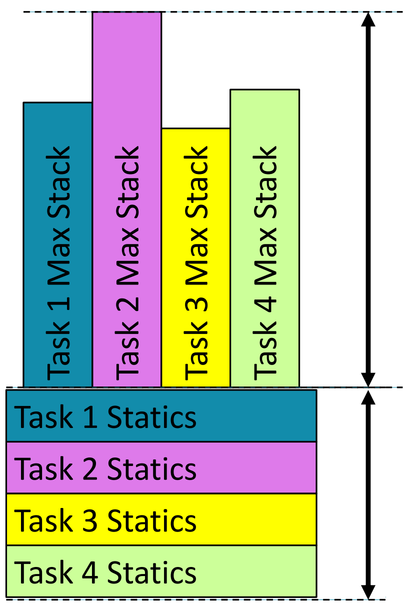
- What does Real-Time mean?
  - Can calculate and guarantee the *maximum response time* for each task and interrupt service routine
  - This "bounding" of response times allows use in hard-real-time systems (which have deadlines which must be met)
  - *Embedded systems RT is different than in other contexts.*

- 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??*

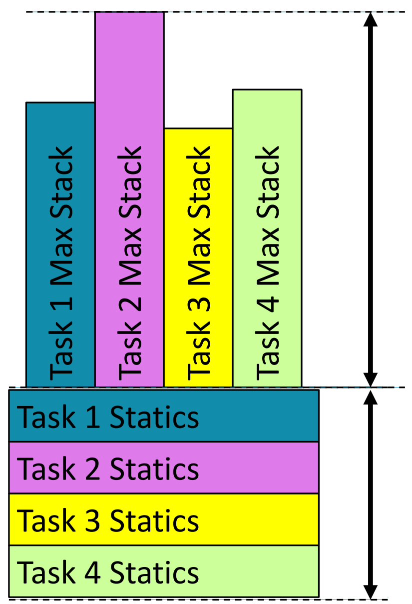
ARM

Comparison of RAM Requirements

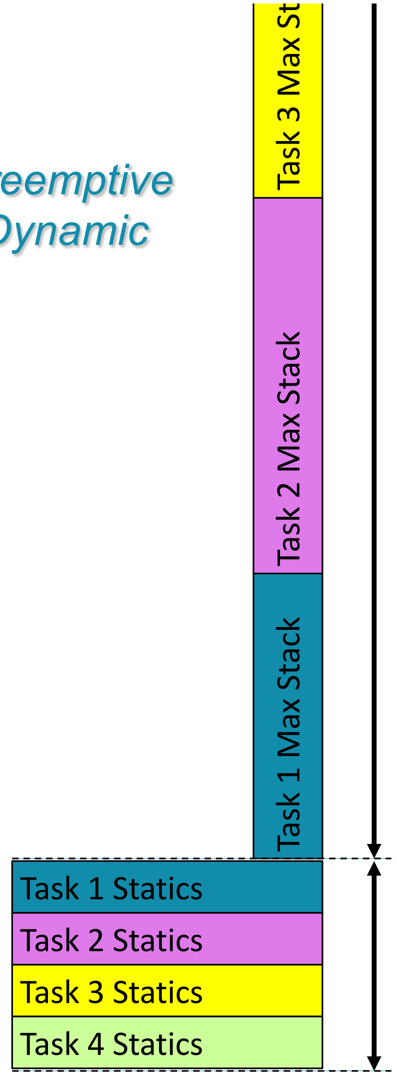
Non-preemptive Static



Non-preemptive Dynamic



Preemptive Dynamic



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

**except for certain special cases*

End.

Material Source

ARM University
Worldwide Education Program

ARM-based Embedded Systems Design Lab-in-a-Box
Freescale Freedom Board Edition

Content is used with permission of Arm Limited.

ARM-specific content comes courtesy of the ARM University Worldwide Education Program. Main author of this Arm University content is Dr. Alexander Dean, North Carolina State University, with updates and changes by J.A. Smith.

ARM