

Disclaimer

- *Arm is committed to making the language we use inclusive, meaningful, and respectful. Our goal is to remove and replace non-inclusive language from our vocabulary to reflect our values and represent our global ecosystem.*
-
- *Arm is working actively with our partners, standards bodies, and the wider ecosystem to adopt a consistent approach to the use of inclusive language and to eradicate and replace offensive terms. We recognise that this will take time. This course contains references to non-inclusive language; it will be updated with newer terms as those terms are agreed and ratified with the wider community.*
-
- *Contact us at education@arm.com with questions or comments about this course. You can also report non-inclusive and offensive terminology usage in Arm content at terms@arm.com.*



Introduction to Embedded Systems Design

Learning Objectives

At the end of this lecture, you should be able to:

- Outline what is meant by Embedded Systems, give examples of its application and state its benefits.
- Describe the attributes of embedded systems.
- Outline the constraints on embedded systems including their impact.

Outline

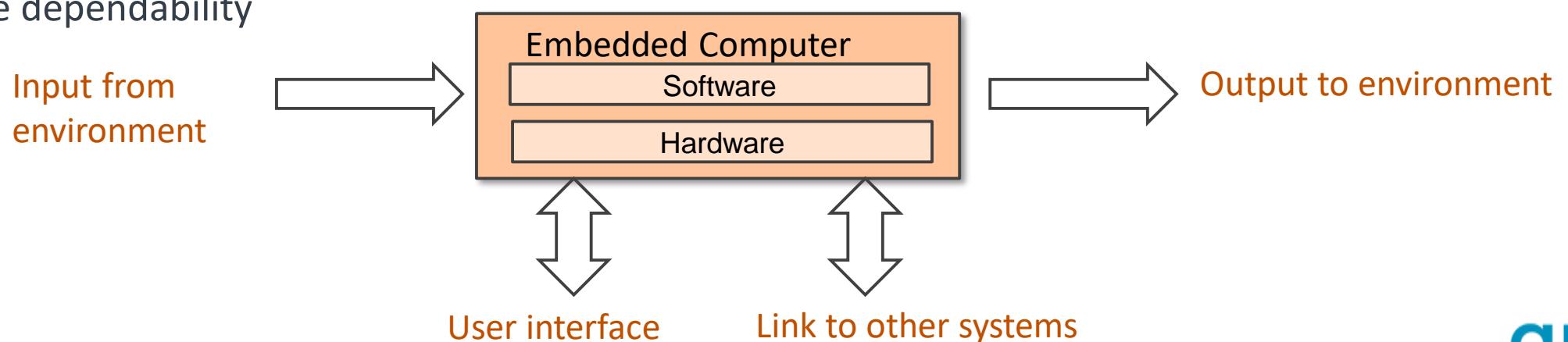
- Introduction to Embedded Systems
- Example Embedded Systems:
 - Bike Computer
 - Motor Control Unit
 - Gasoline automobile engine control unit
- Options for Building Embedded Systems
- Benefits of embedded systems
- Embedded System Functions
- Attributes of embedded systems
- MCU Hardware & Software for Concurrency
- Embedded System Constraints and their impacts

Introduction

- What is an Embedded System?
 - Application-specific computer system
 - Built into a larger system
- Why add a computer to the larger system?
 - Better performance
 - More functions and features
 - Lower cost
 - More dependability
- Economics
 - Microcontrollers (used for embedded computers) are high-volume, so recurring cost is low
 - Nonrecurring cost dominated by software development
- Networks
 - Often embedded system will use multiple processors communicating across a network to lower parts and assembly costs and improve reliability

Introduction to embedded systems

- What is an embedded system?
 - Application-specific computer system
 - Built into a larger system
 - Often with real-time computing constraints
- Why add an embedded computer to a larger system?
 - Better performance
 - More functions and features
 - Lower cost, e.g., through automation
 - More dependability

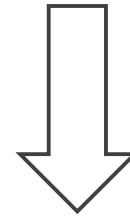


Example Embedded System: Bike Computer

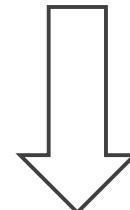
- Functions
 - Speed and distance measurement
- Constraints
 - Size
 - Cost
 - Power and Energy
 - Weight
- Inputs
 - Wheel rotation indicator
 - Mode key
- Output
 - Liquid Crystal Display
- Low performance MCU
 - 8-bit, 10 MIPS

Example embedded system: bike computer

- Functions
 - Speed and distance measurement
- Constraints
 - Size
 - Cost
 - Power and energy
 - Weight
- Inputs
 - Wheel rotation indicator
 - Mode key
- Output
 - Liquid Crystal Display
- Use Low Performance Microcontroller
 - 8-bit, 10 MIPS



Input:
Wheel rotation
Mode key



Output:
Display speed and
distance

Motor Control Unit

- Functions
 - Motor control
 - System communications
 - Current monitoring
 - Rotation speed detection
- Constraints
 - Reliability in harsh environment
 - Cost
 - Weight
- Many Inputs and Outputs
 - Discrete sensors & actuators
 - Network interface to rest of car
- High-Performance MCU
 - 32-bit, 256KB flash memory, 80MHz

Gasoline automobile engine control unit

- Functions
 - Fuel injection
 - Air intake setting
 - Spark timing
 - Exhaust gas circulation
 - Electronic throttle control
 - Knock control
- Constraints
 - Reliability in harsh environment
 - Cost
 - Weight
- Many inputs and outputs
 - Discrete sensors & actuators
 - Network interface to rest of car
- Use high-performance microcontroller
 - E.g. 32-bit, 3MB flash memory, 150-300 MHz



Options for Building Embedded Systems

Dedicated Hardware

Implementation	Design Cost	Unit Cost	Upgrades & Bug Fixes	Size	Weight	Power	System Speed
Discrete Logic	low	mid	hard	large	high	?	very fast
ASIC	high (\$500K/mask set)	very low	hard	tiny - 1 die	very low	low	extremely fast
Programmable logic – FPGA, PLD	low	mid	easy	small	low	medium to high	very fast
Microprocessor + memory + peripherals	low to mid	mid	easy	small to med.	low to moderate	medium	moderate
Microcontroller (int. memory & peripherals)	low	mid to low	easy	small	low	medium	slow to moderate
Embedded PC	low	high	easy	medium	moderate to high	medium to high	fast

Software Running on Generic Hardware

Benefits of embedded systems

- Greater performance and efficiency
 - Software makes it possible to provide sophisticated controls
- Lower costs
 - Less expensive components can be used
 - Manufacturing costs reduced
 - Operating costs reduced
 - Maintenance costs reduced
- More features
 - Many not possible or practical with other approaches
- Better dependability
 - Adaptive system which can compensate for failures
 - Better diagnostics to improve repair time

Embedded System Functions

- Closed-loop control system
 - Monitor a process, adjust an output to maintain desired set point (temperature, speed, direction, etc.)
- Sequencing
 - Step through different stages based on environment and system
- Signal processing
 - Remove noise, select desired signal features
- Communications and networking
 - Exchange information reliably and quickly

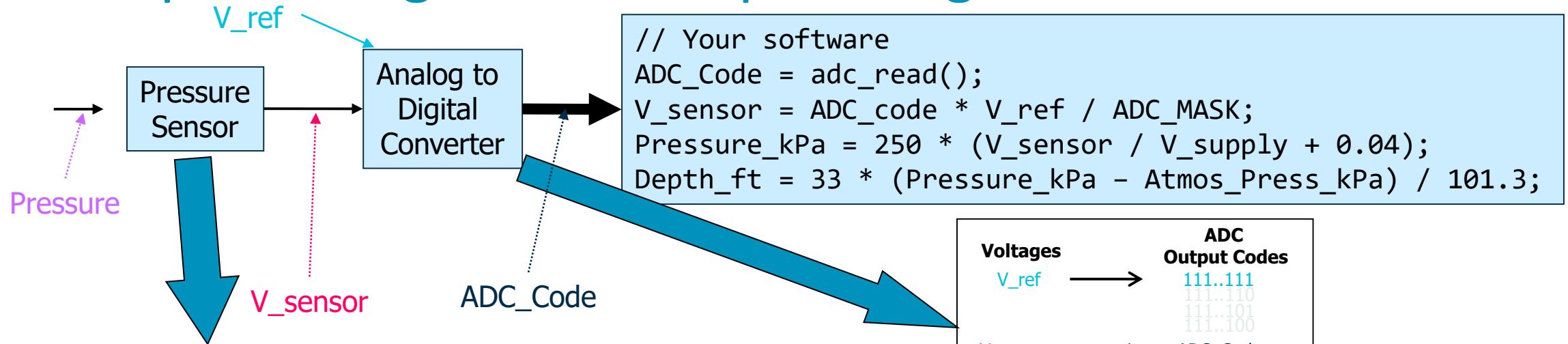
Attributes of embedded systems

- Interfacing with larger system and environment
 - Analog signals for reading sensors
 - Typically use a voltage to represent a physical value
 - Power electronics for driving motors, solenoids
 - Digital interfaces for communicating with other digital devices
 - Simple – switches
 - Complex – displays
- Concurrent, reactive behaviors
 - Must respond to sequences and combinations of events
 - Real-time systems have deadlines on responses
 - Typically must perform multiple separate activities concurrently

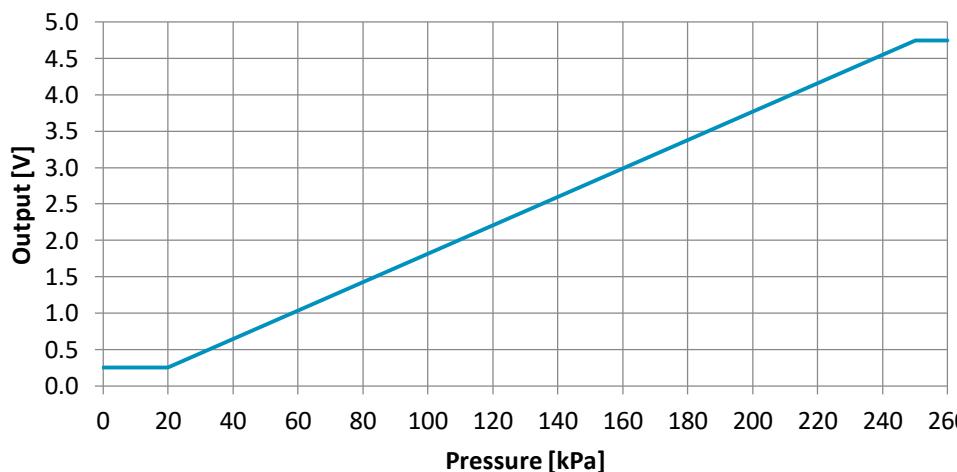
Attributes of embedded systems

- Fault handling
 - Many systems must operate independently for long periods of time, requiring them to handle likely faults without crashing
 - Often fault-handling code is larger and more complex than the normal-case code
- Diagnostics
 - Help service personnel determine problems quickly

Example Analog Sensor - Depth Gauge



Typical Absolute Pressure vs. Output



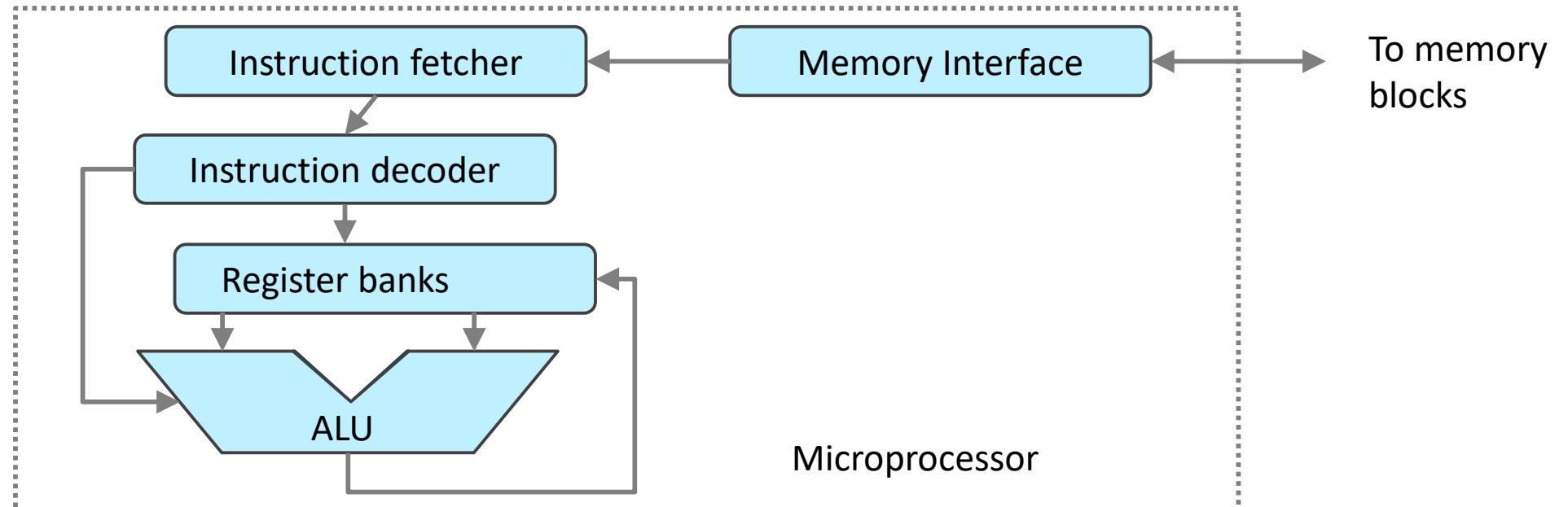
- Sensor detects pressure and generates a proportional output voltage V_{sensor}
- ADC generates a proportional digital integer (code) based on V_{sensor} and V_{ref}
- Code can convert that integer to something more useful
 - first a float representing the voltage,
 - then another float representing pressure,
 - finally another float representing depth

Microcontroller vs. Microprocessor

- Both have a CPU core to execute instructions
- Microcontroller has peripherals for concurrent embedded interfacing and control
 - Analog
 - Non-logic level signals
 - Timing
 - Clock generators
 - Communications
 - Reliability and safety

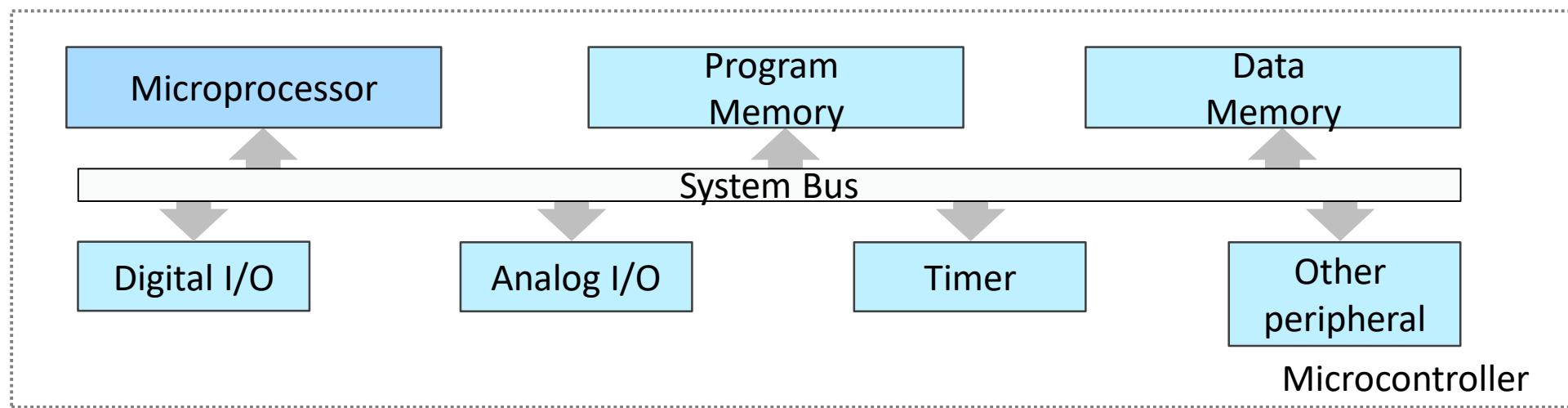
CPUs → MCUs → embedded systems

- Microprocessor (CPU)
 - Defined typically as a single processor core that supports at least instruction fetching, decoding, and executing
 - Normally can be used for general-purpose computing, but needs to be supported with memories and Input/Outputs(IOs)



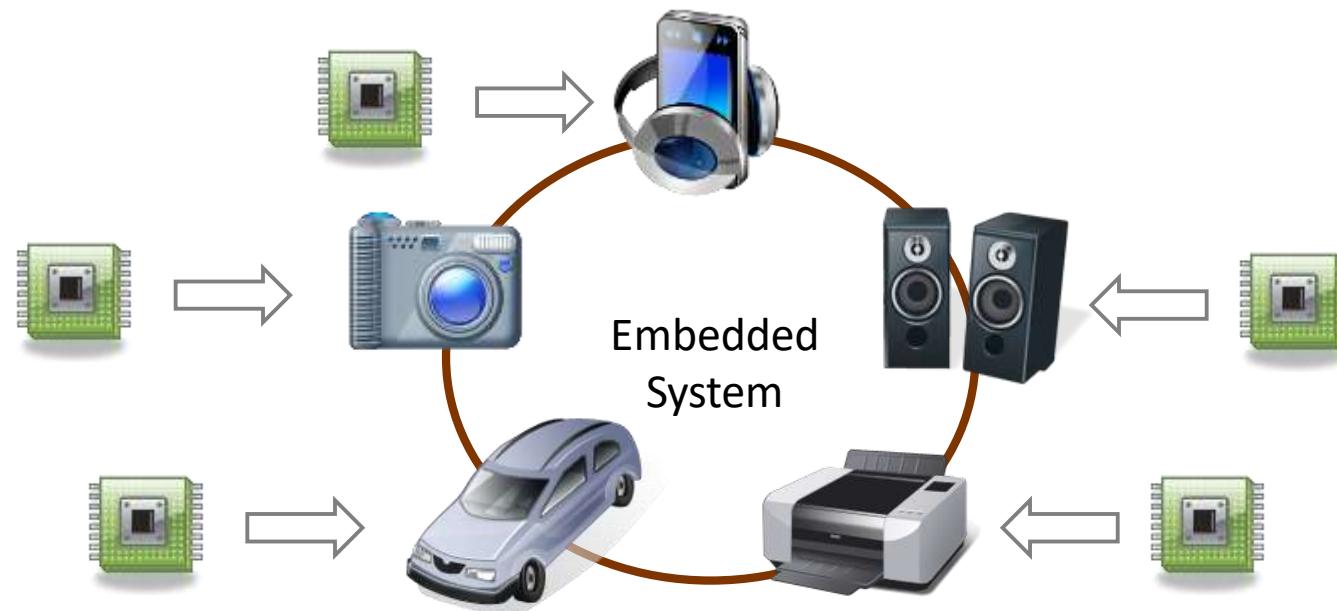
CPUs → MCUs → embedded systems

- Microcontroller (MCU)
 - Typically has a single processor core
 - Has memory blocks, Digital IOs, Analog IOs, and other basic peripherals
 - Typically used for basic control purpose, such as embedded applications



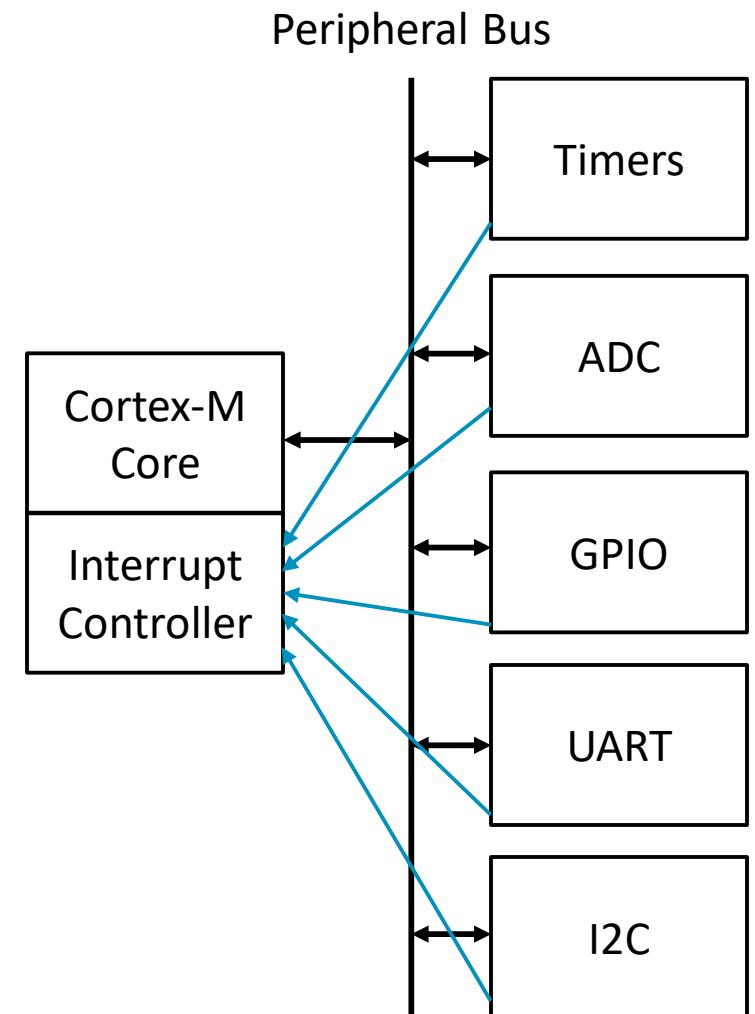
CPUs → MCUs → embedded systems

- Embedded System
 - Typically implemented using MCUs
 - Often integrated into a larger mechanical or electrical system
 - Usually has real-time constraints

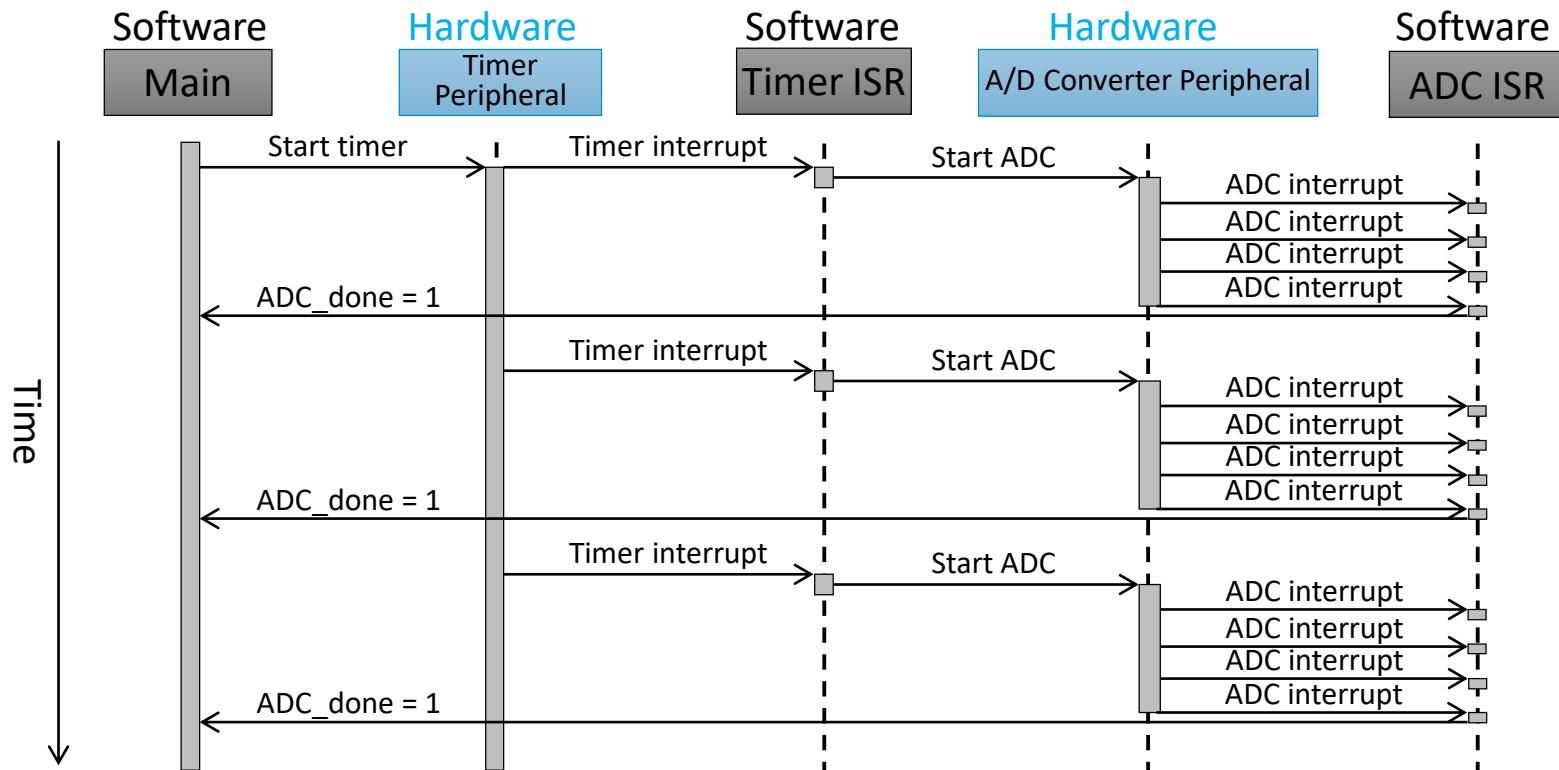


MCU Hardware & Software for Concurrency

- CPU executes instructions from one or more threads of execution
- Specialized hardware peripherals add dedicated concurrent processing
 - Watchdog timer
 - Analog interfacing
 - Timers
 - Communications with other devices
 - Detecting external signal events
 - LCD driver
- 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

Constraints

- Cost
 - Competitive markets penalize products which don't deliver adequate value for the cost
- Size and weight limits
 - Mobile (aviation, automotive) and portable (e.g. handheld) systems
- Power and energy limits
 - Battery capacity
 - Cooling limits
- Environment
 - Temperatures may range from -40° C to 125° C, or even more

Impact of Constraints

- Microcontrollers used (rather than microprocessors)
 - Include peripherals to interface with other devices, respond efficiently
 - On-chip RAM, ROM reduce circuit board complexity and cost
- Programming language
 - Programmed in C rather than Java (smaller and faster code, so less expensive MCU)
 - Some performance-critical code may be in assembly language
- Operating system
 - Typically, no OS, but instead simple scheduler (or even just interrupts + main code (foreground/background system))
 - If OS is used, likely to be a lean RTOS

Building embedded systems using MCUs

- In most embedded systems, MCUs are chosen to be the best solution, since they offer:
 - Low development and manufacturing cost
 - Easy porting and updating
 - Light footprint
 - Relatively low power consumption
 - Satisfactory performance for low-end products

Curriculum Overview

- Introductory Course: Building an Embedded System with an MCU
 - Microcontroller concepts
 - Software design basics
 - Processor core architecture and interrupt system
 - C as implemented in assembly language
 - Peripherals and interfacing

Why Are We...?

- Using C instead of Java (or Python, or your other favorite language)?
 - C is the de facto standard for embedded systems because of:
 - Precise control over what the processor is doing.
 - Modest requirements for ROM, RAM, and MIPS, so much cheaper system
 - Predictable behavior, no OS (e.g. Garbage Collection) preemption
- Learning assembly language?
 - The compiler translates C into assembly language. To understand whether the compiler is doing a reasonable job, you need to understand what it has produced.
 - Sometimes we may need to improve performance by writing assembly versions of functions.
- Required to have a microcontroller board?
 - The best way to learn is hands-on.



[†]The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks



Software Design Basics

Learning Objectives

At the end of this lecture, you should be able to:

- Outline the meaning of concurrency and give examples .
- Define the following terms: TRelease(i), TLatency(i), TResponse(i), TTask(i) and TISR(i).
- Describe the following two scheduling approaches: hardware interrupt and software scheduler.
- Explain static, dynamic run-to-completion and dynamic pre-emptive scheduling.
- Describe the following: Cyclic Executive with Interrupts, Run-To-Completion Scheduler and Preemptive Scheduler.
- Identify the components of RTOS .
- Outline the steps in Waterfall V software development models.

Overview

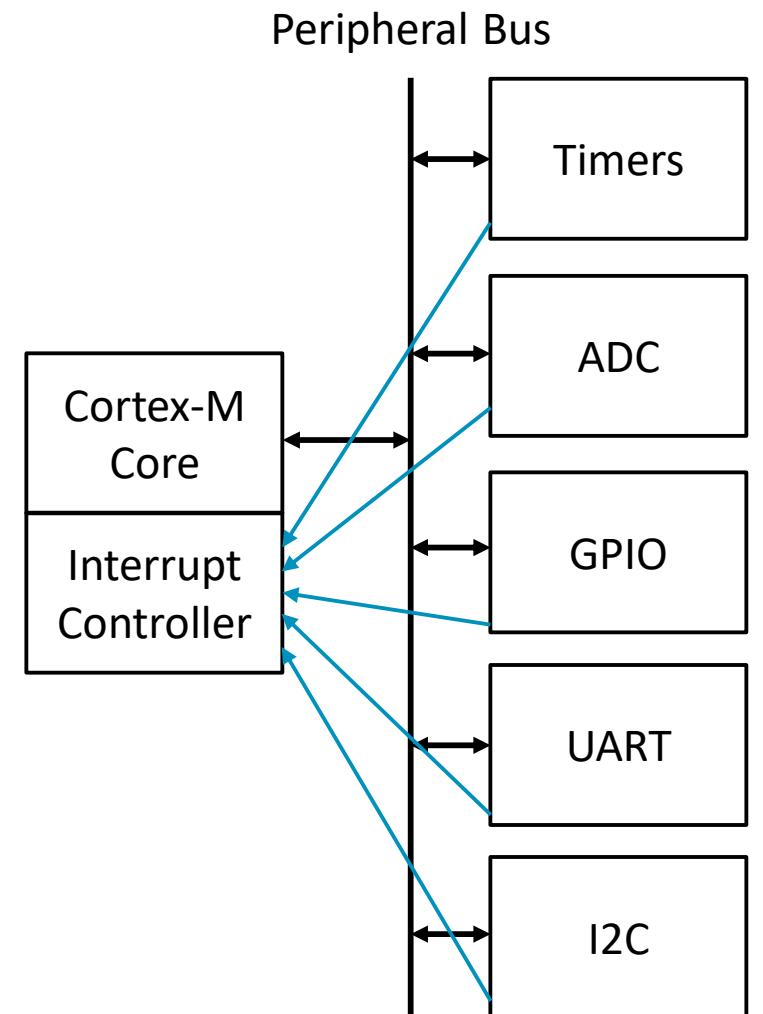
- Concurrency
 - How do we make things happen at the right times?
- Software Engineering for Embedded Systems
 - How do we develop working code quickly?

arm

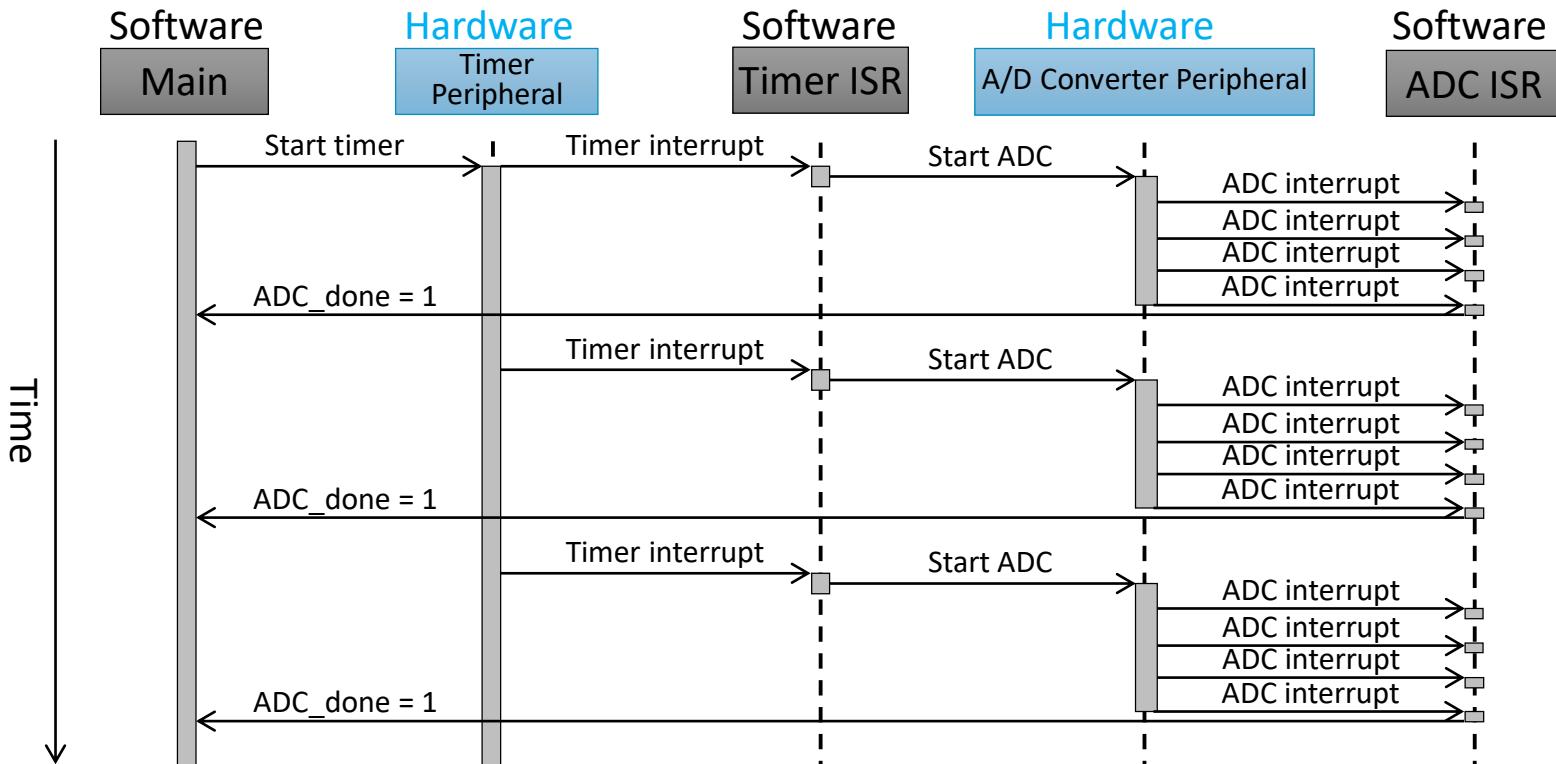
CONCURRENCY

MCU Hardware & Software for Concurrency

- CPU executes instructions from one or more thread of execution
- Specialized hardware peripherals add dedicated concurrent processing
 - Watchdog timer
 - Analog interfacing
 - Timers
 - Communications with other devices
 - Detecting external signal events
 - LCD driver
- Peripherals use interrupts to notify CPU of events



Concurrent Hardware & Software Operation

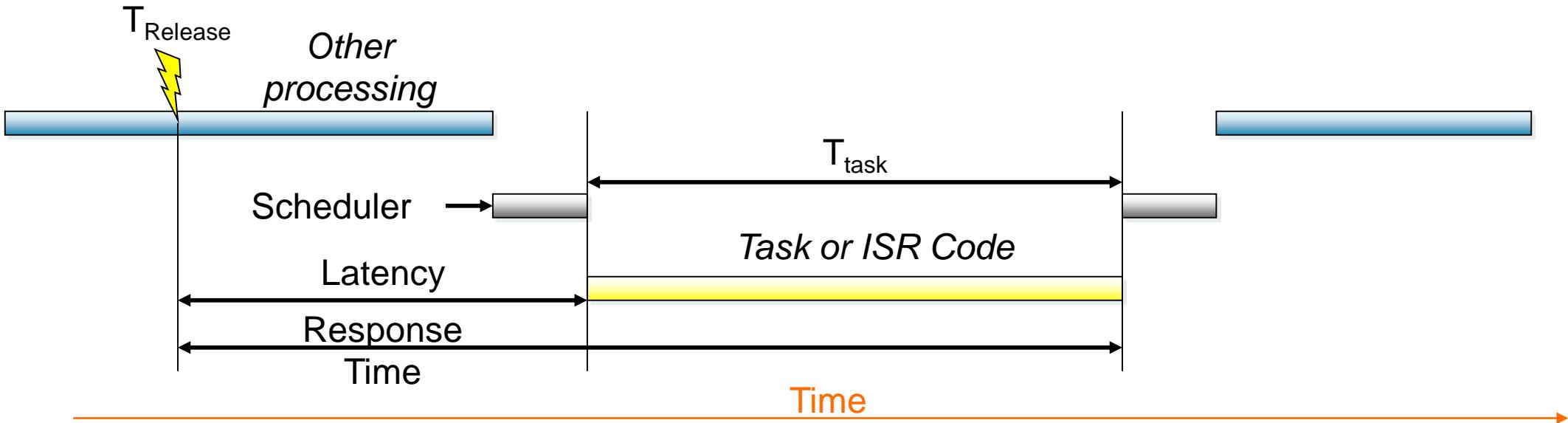


- Embedded systems rely on both MCU hardware peripherals and software to get everything done on time

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



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

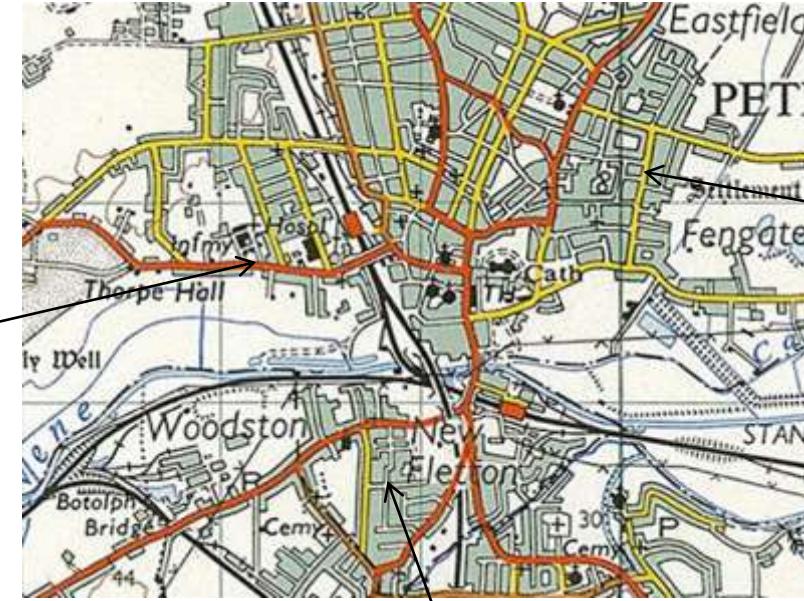
ISR 2:
Mode Key

```
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  
}
```

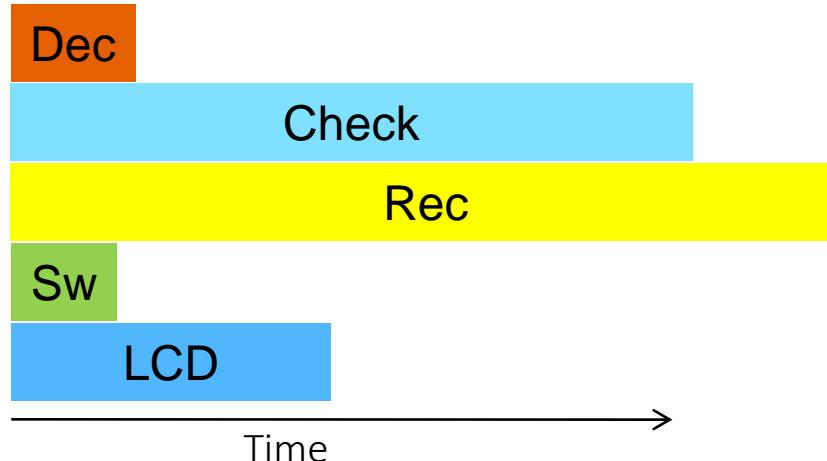
A More Complex Application



- 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?

Dec

Check

Rec

Sw

LCD

- Task scheduling: Deciding which task should be running 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)

Dec

Check

Rec

Sw

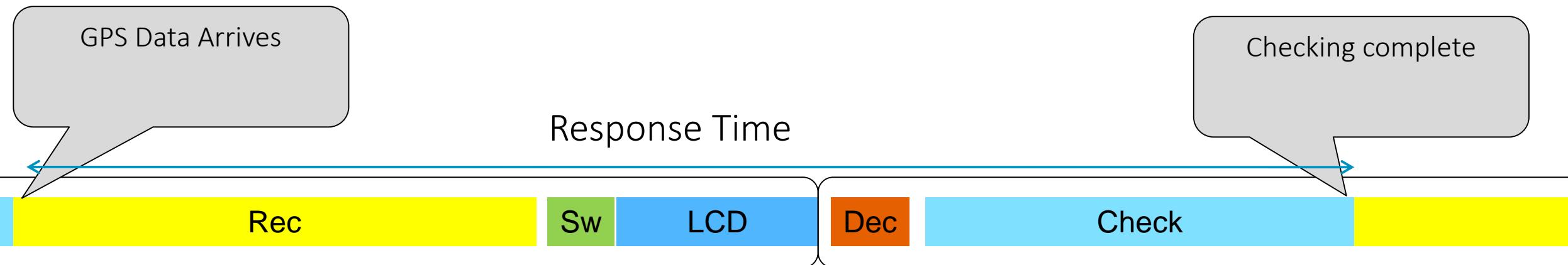
LCD

Dec

- 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.

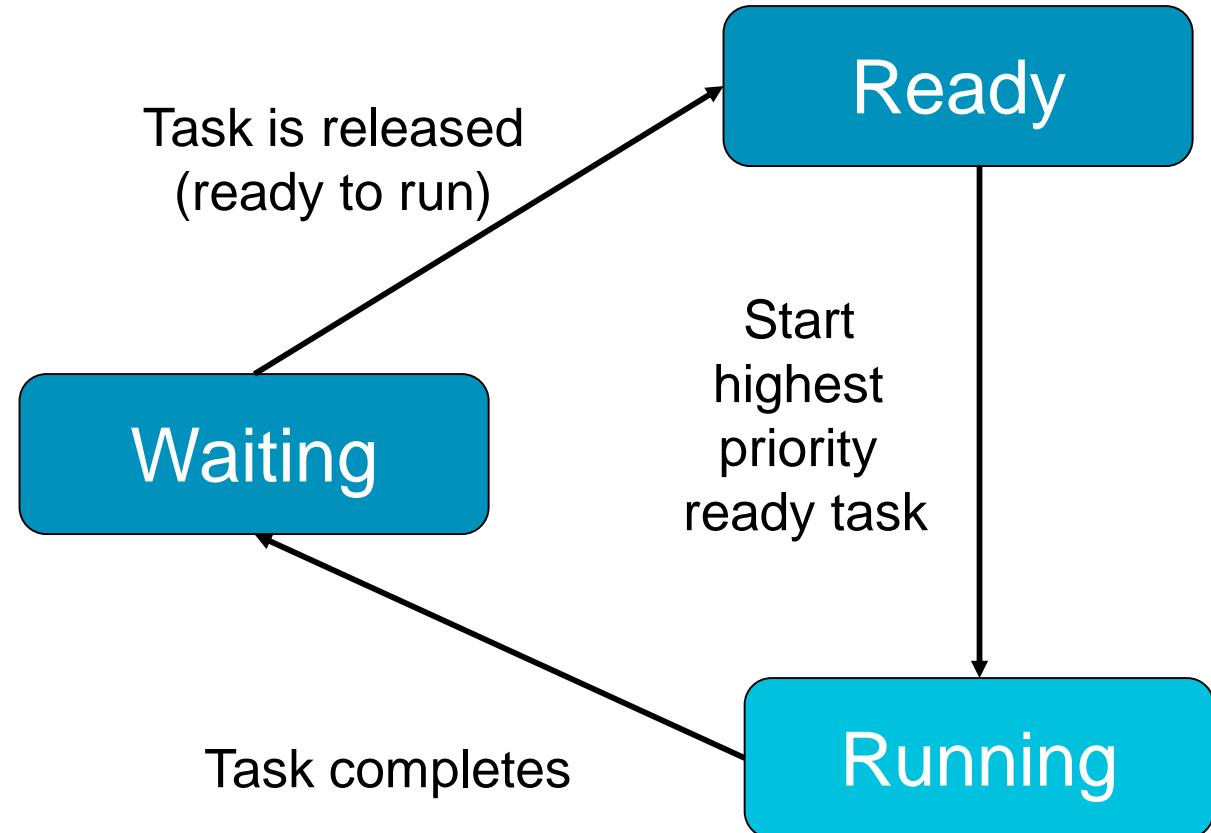
Dynamic RTC Schedule



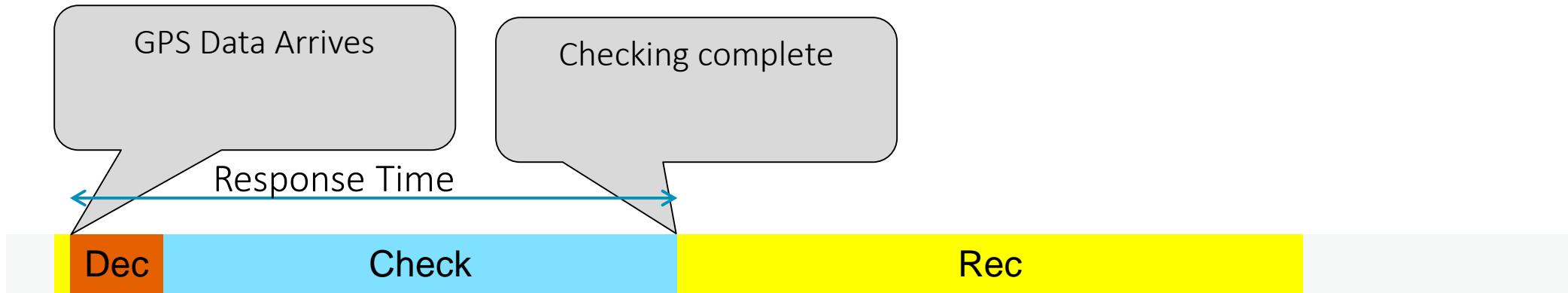
- What if we receive GPS position right after Rec starts running?
- Delays
 - Have to wait for Rec to finish before we start decoding position with Dec.
 - Have to wait for Rec, Dec, Check before we know if we are approaching a pothole

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

Static

Rec

Sw

LCD

Dec

Check

Dynamic Run-to-Completion

Rec

Dec

Check

Dynamic Preemptive

Dec

Check

- 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 – foreground
 - Interrupts – background
- Example of a foreground / background system
- Main user code runs in foreground
- Interrupt routines run in background (high priority)
 - Run when triggered
 - Handle most urgent work
 - Set flags to request processing by main loop

```
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

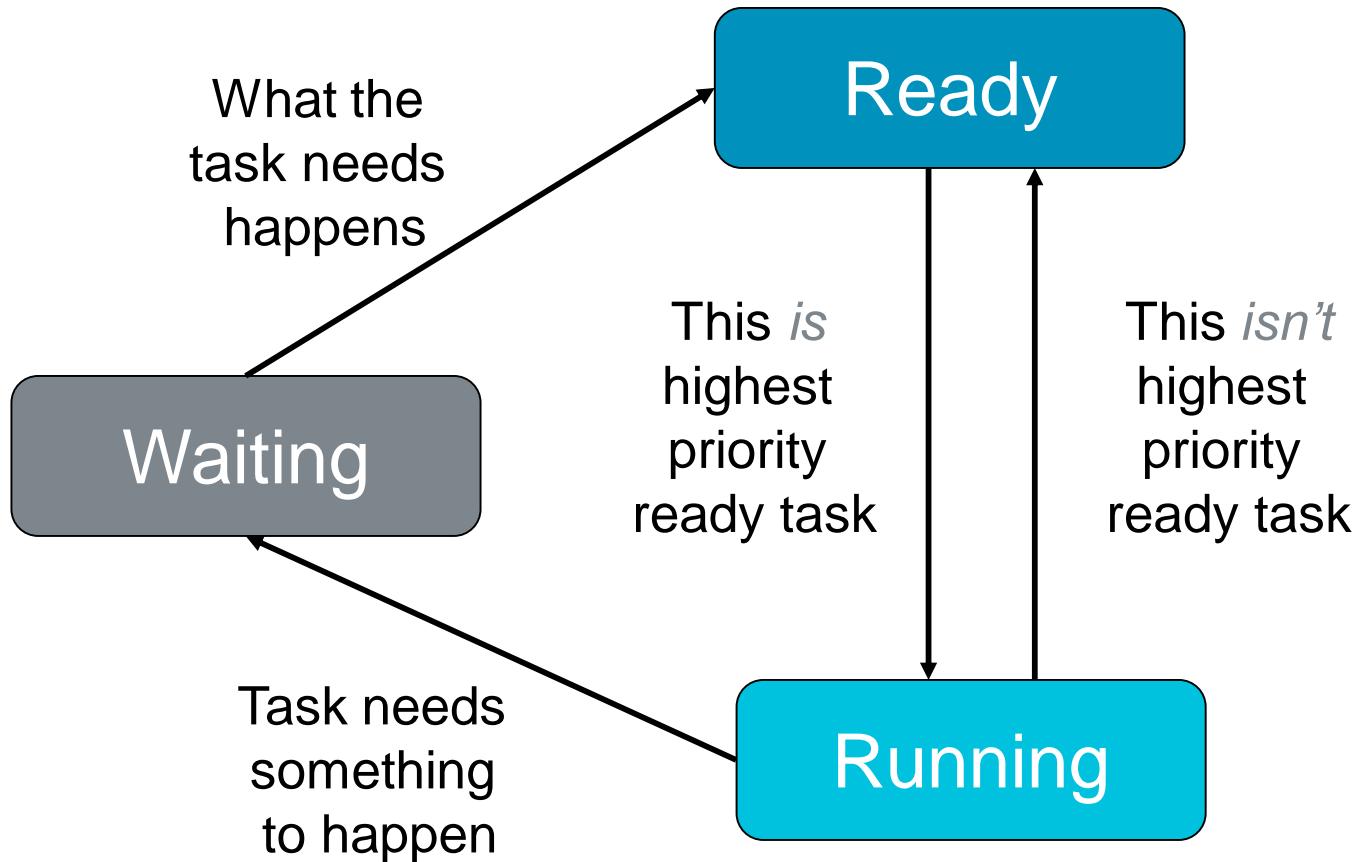
- Use a scheduler function to run task functions at the right rates
 - Table stores information per task
 - Period: How many ticks between each task release
 - Release Time: how long until task is ready to run
 - ReadyToRun: task is ready to run immediately
 - Scheduler runs forever, examining schedule table which indicates tasks which are ready to run (have been “released”)
 - A periodic timer interrupt triggers an ISR, which updates the schedule table
 - Decrement “time until next release”
 - If this time reaches 0, set that 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*

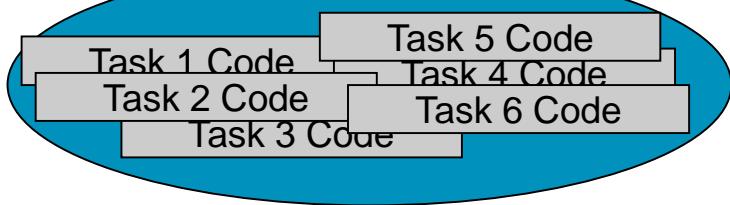
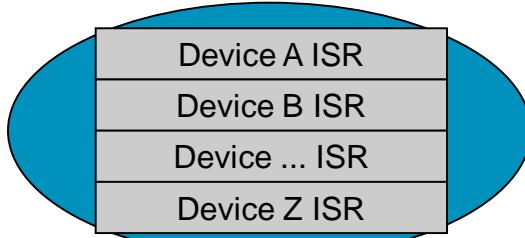


What's an RTOS?

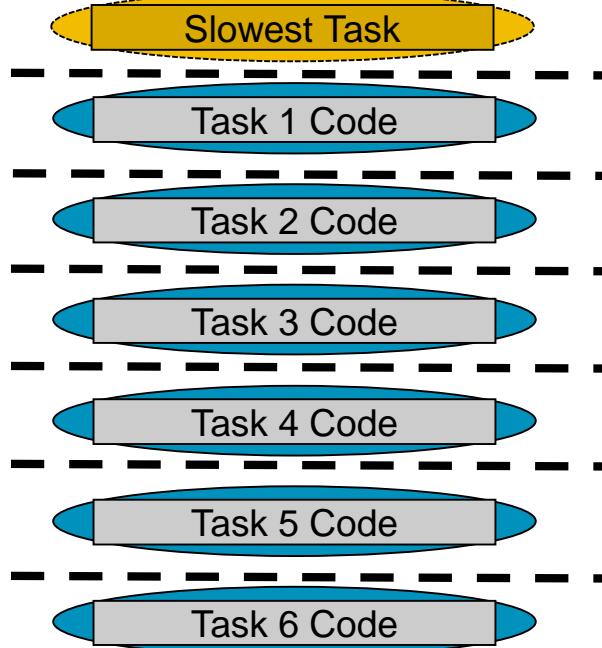
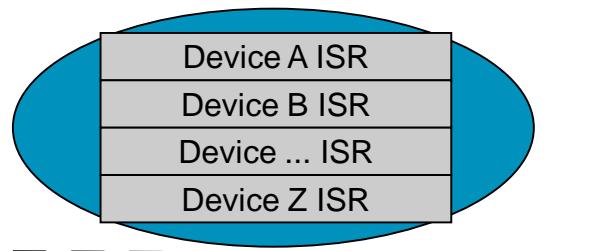
- What does Real-Time mean?
 - Can calculate and guarantee the maximum response time for each task and interrupt service routine
 - This “bounding” of response times allows use in hard-real-time systems (which have deadlines which must be met)
- What's in the RTOS
 - Task Scheduler
 - Preemptive, prioritized 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

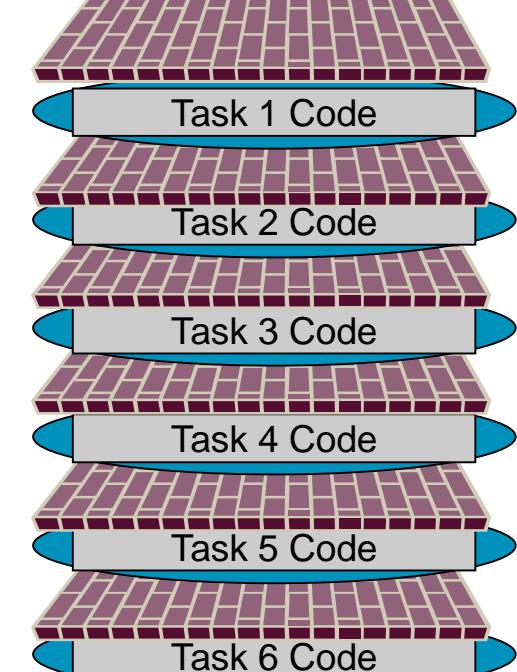
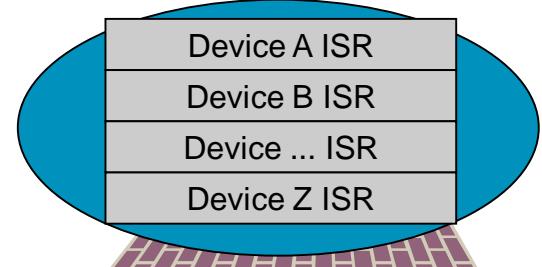
*Non-preemptive
Static*



Non-preemptive Dynamic

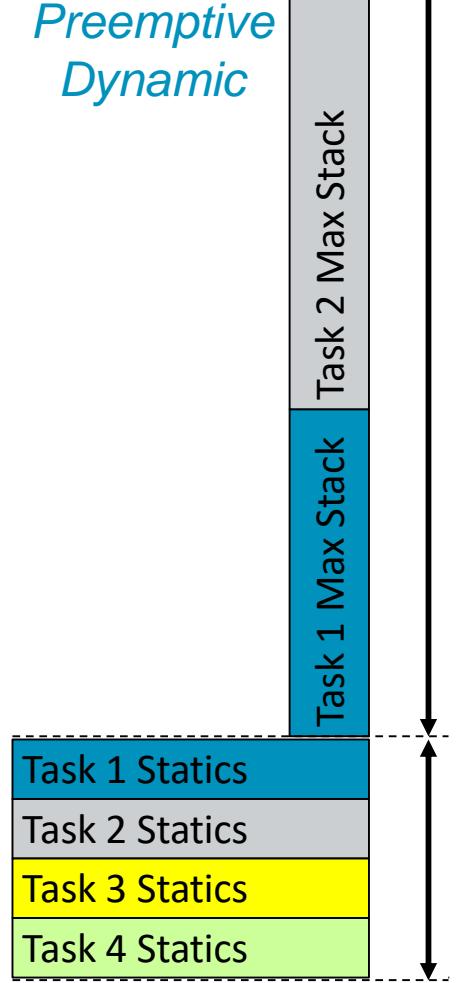
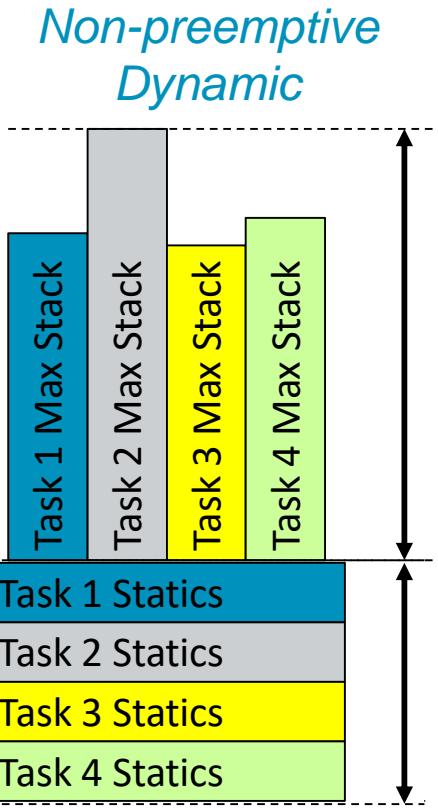
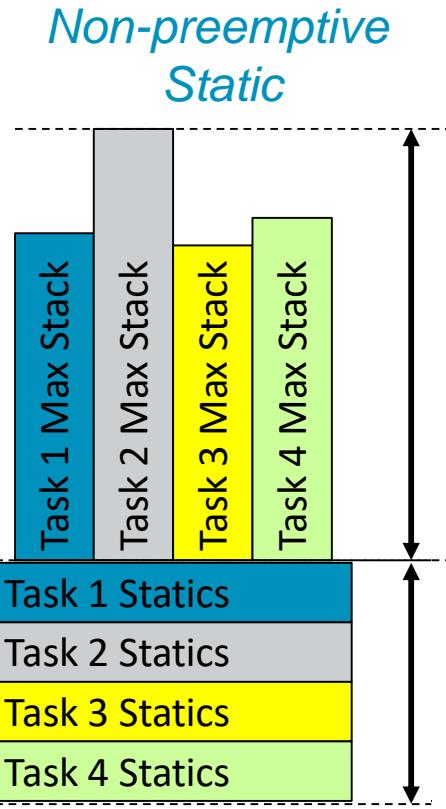


Preemptive Dynamic



- 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*



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 postmortems to improve development process

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!

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 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

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” problem (most recent requirement change is assumed to be most important)

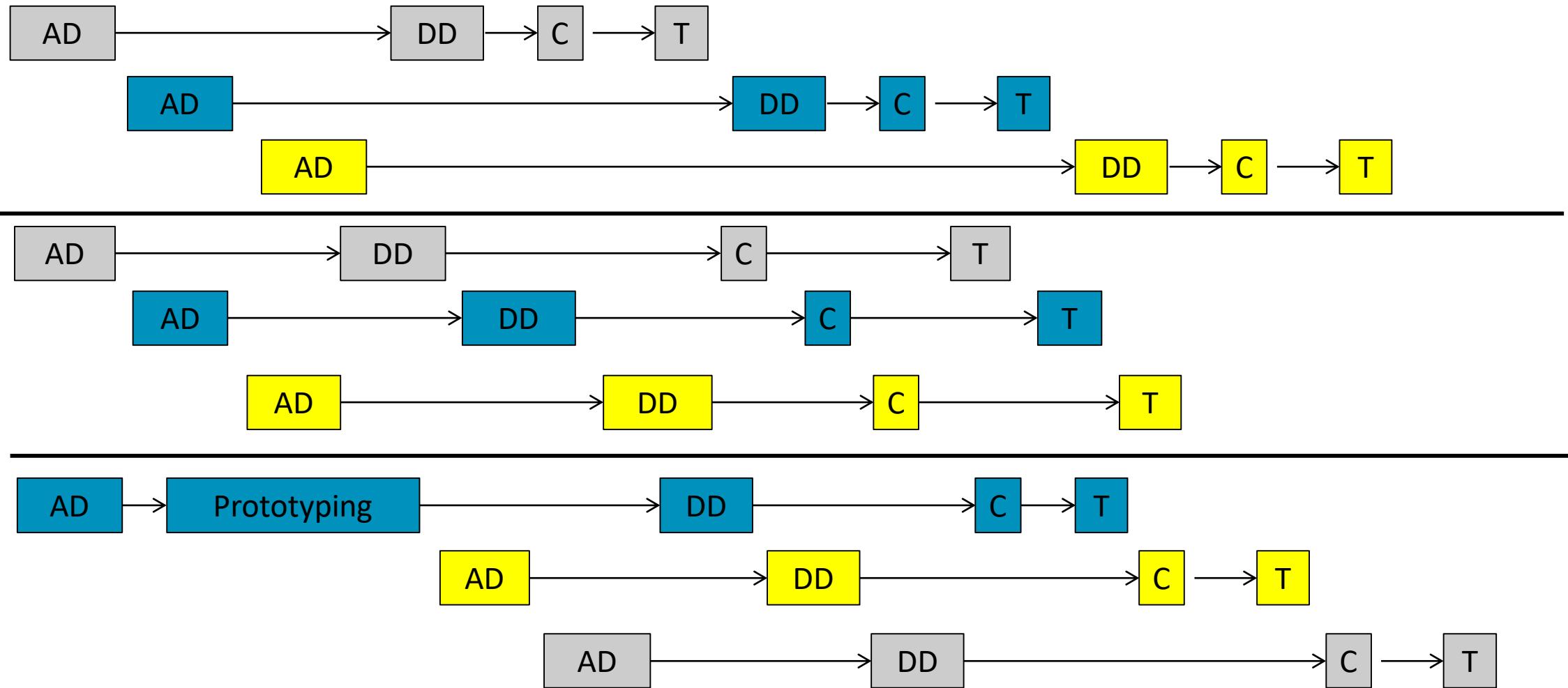
Design Before Coding



- Ganssle's reason #9: *Starting coding too soon*
- Underestimating the complexity of the needed software is a very common risk
- Writing code locks you into 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?

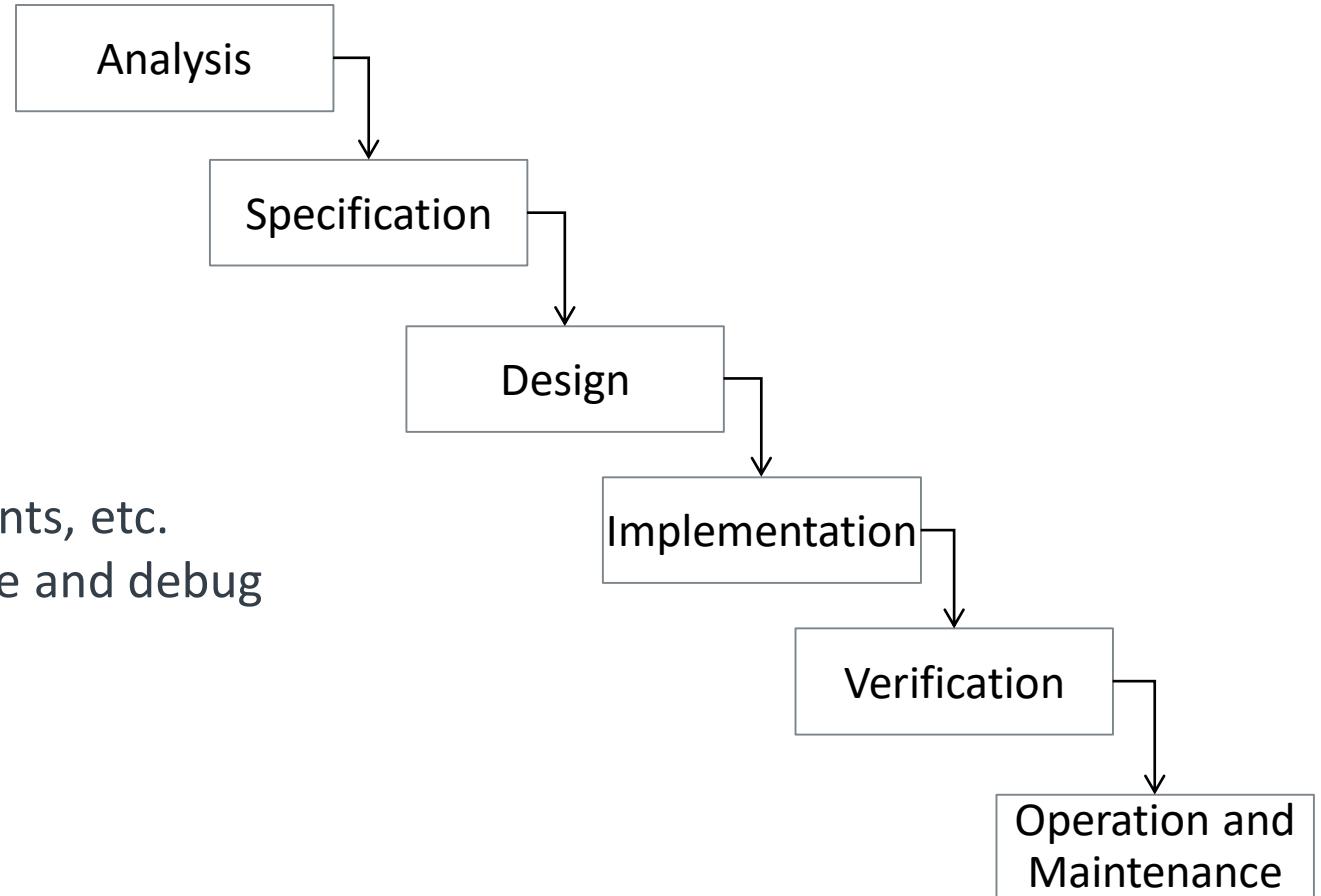


Development Models

- 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: Can create detailed plan. Big-up-front design, waterfall
 - High: Use iterative or agile development method, spiral. Prototype high-risk parts first

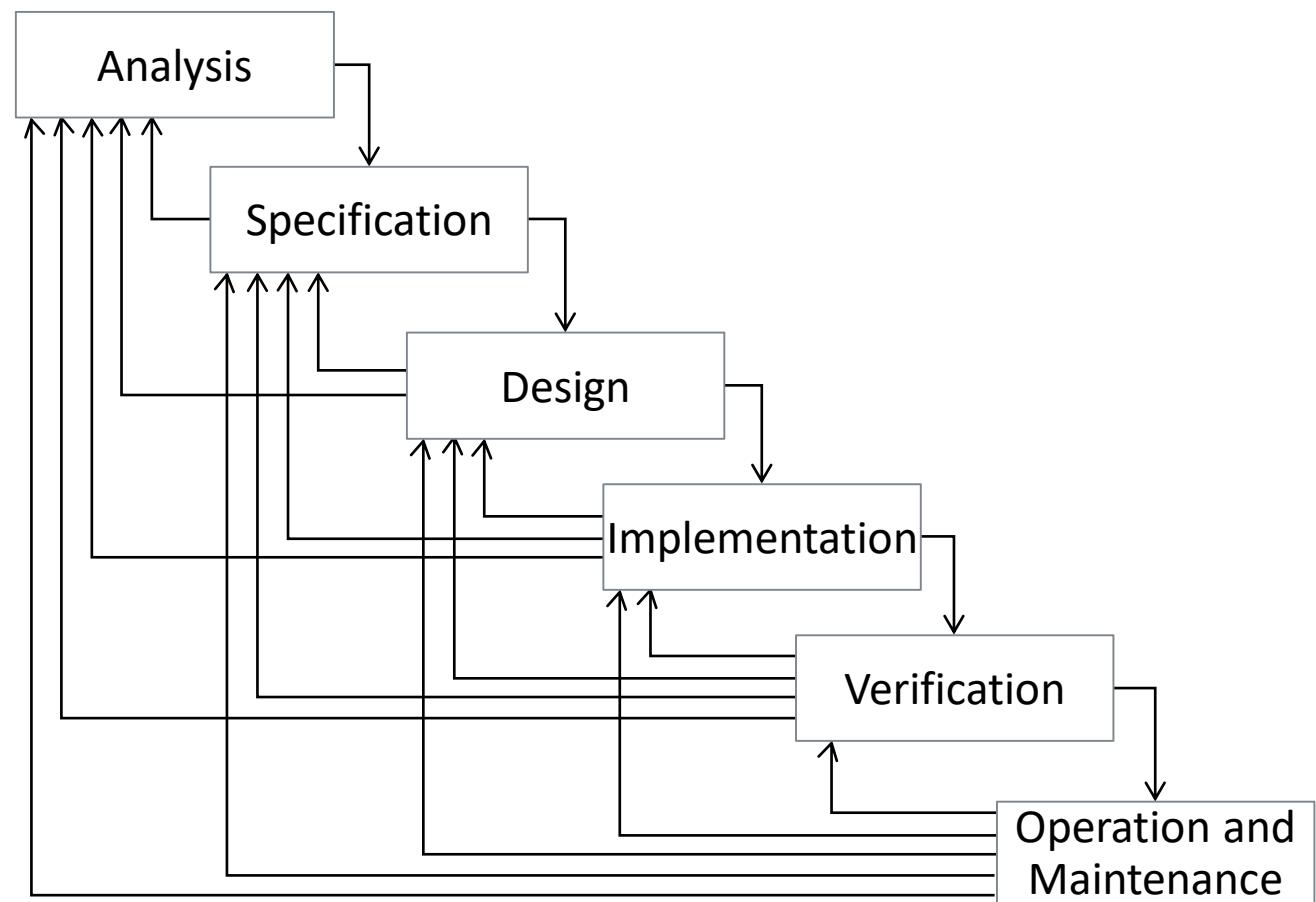
Waterfall (Idealized)

- 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

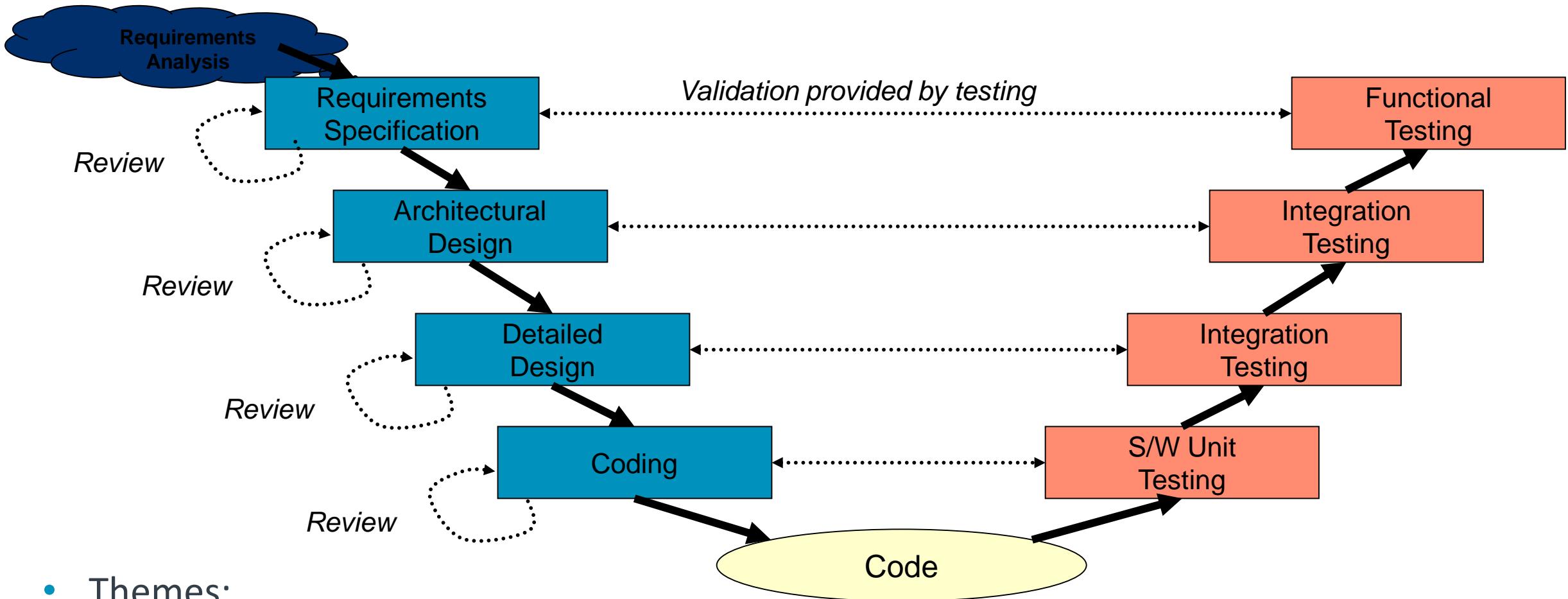


Waterfall (As Implemented)

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



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

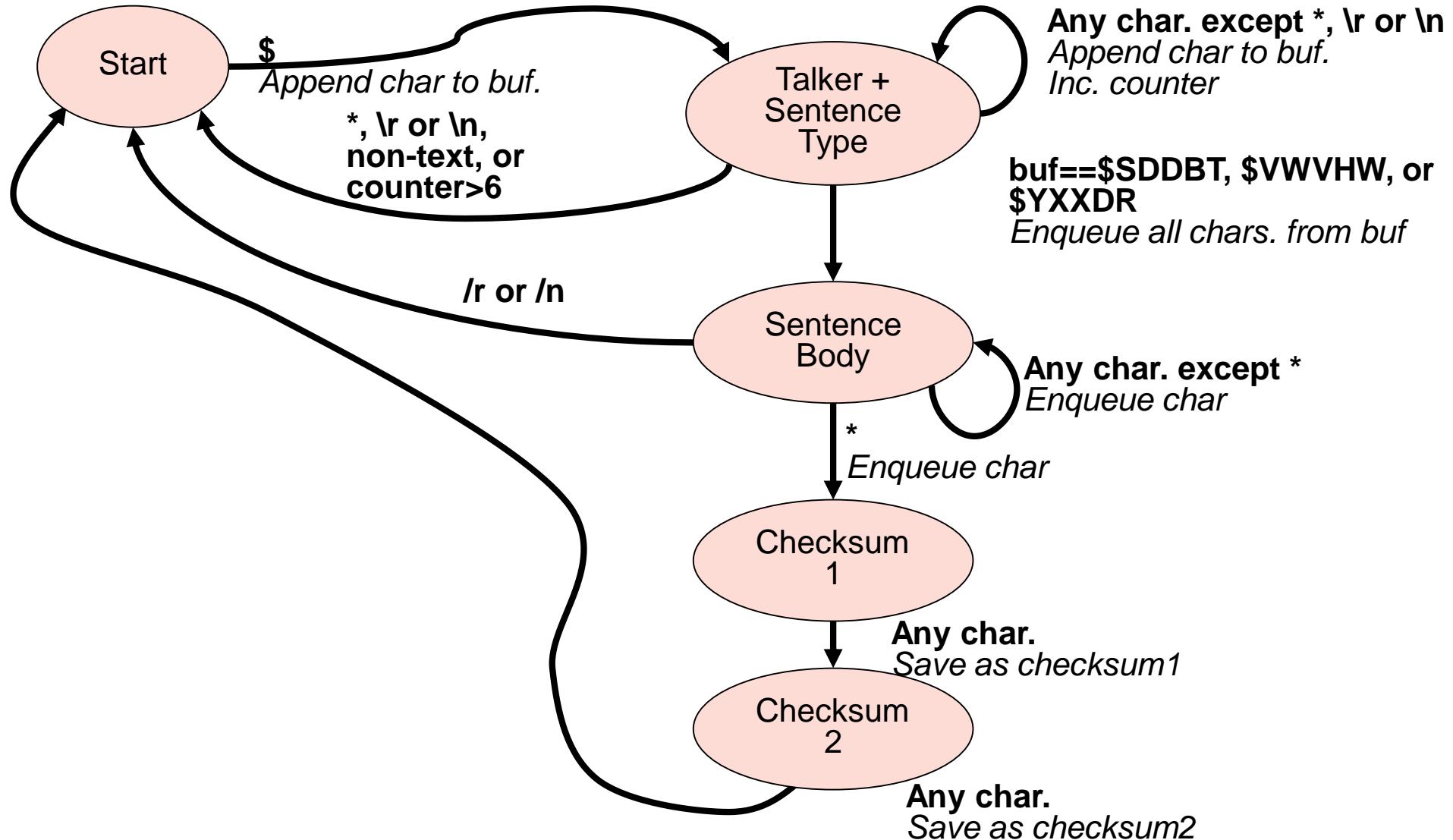
2. Architectural (High-Level) Design

- 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

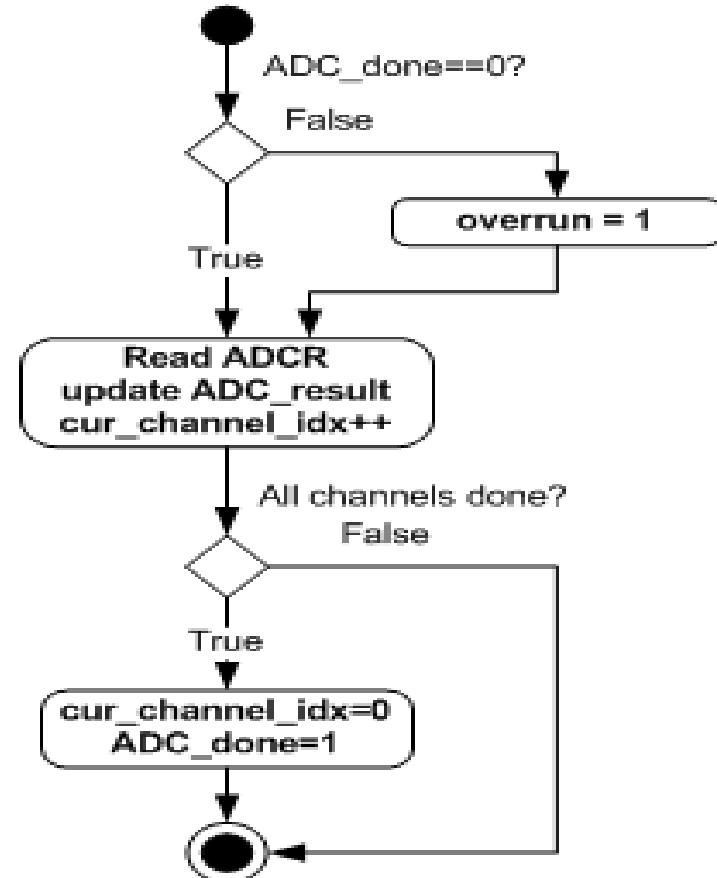
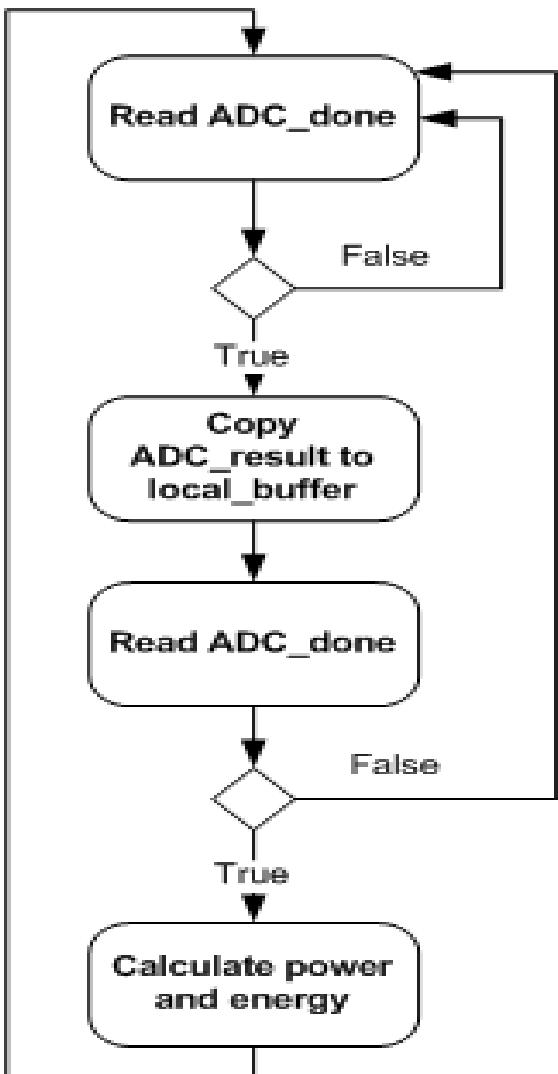
3. Detailed Design

- 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
- Unified Modeling Language (UML)
 - Provides many types of diagrams
 - Some are useful for embedded system design to describe structure or behavior

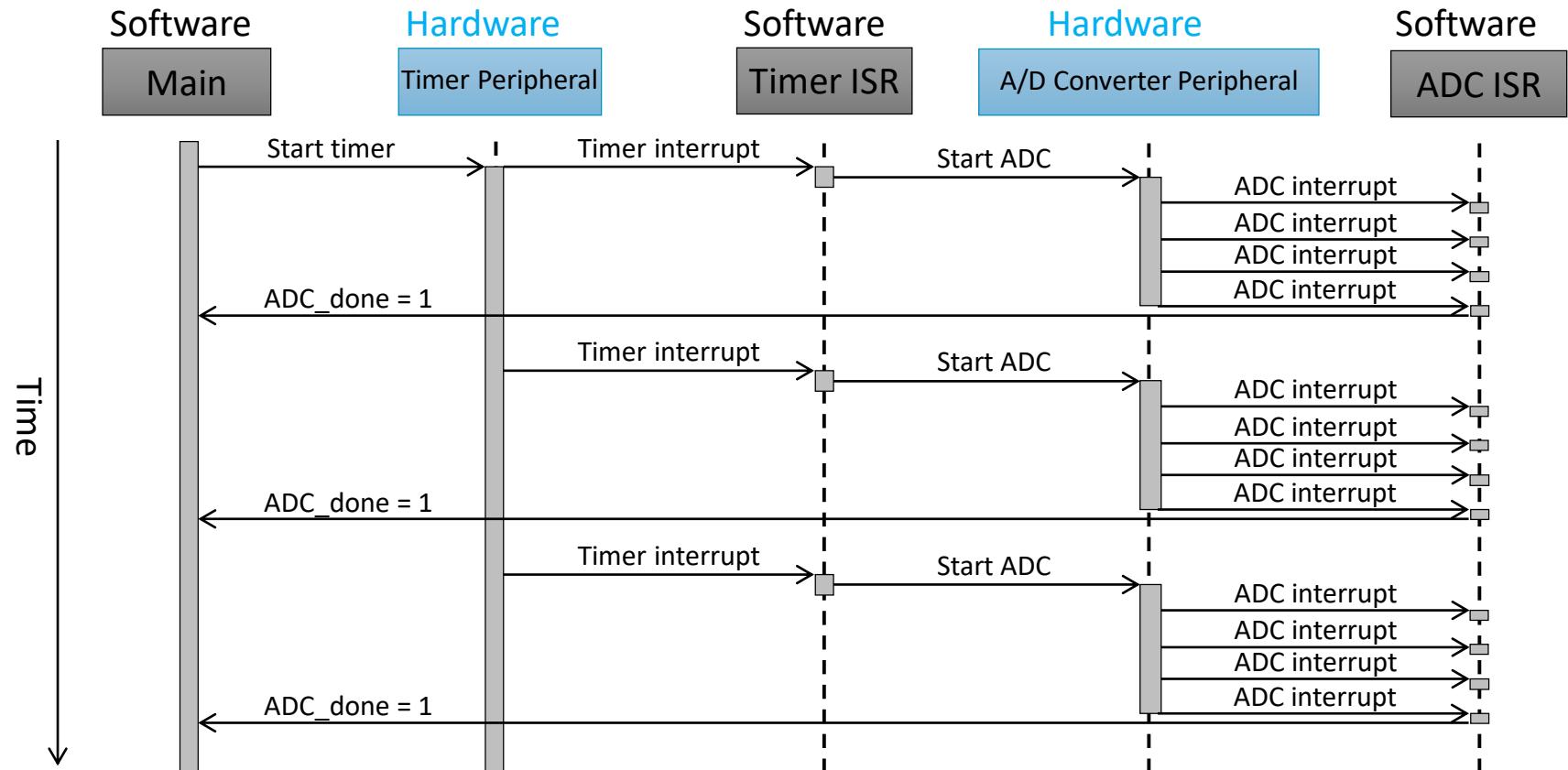
State Machine for Parsing NMEA-0183



Flowcharts



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
- Test effectively
 - Automation
 - Regression testing

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 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

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

Example Postmortem Structure

- Product
 - Bugs
 - Software design
 - Hardware design
- Process
 - Code standards
 - Code interfacing
 - Change control
 - How we did it
 - Team coordination
- Support
 - Tools
 - Team burnout
 - Change orders
 - Personnel availability



[†]The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks



The Arm Cortex-M4 Processor Architecture

Learning Objectives

At the end of this lecture, you should be able to:

- Outline the features and benefits of the Arm Cortex-M4 processor.
- Outline the functions of the Cortex-M4 processor components including Nested Vectored Interrupt Controller (NVIC), Wakeup Interrupt Controller (WIC), Memory Protection Unit (MPU), Bus Interconnect and Debug System.
- Describe the Cortex-M4 processor core registers including their functions.

Module syllabus

- Arm Architectures and Processors
 - What is Arm Architecture
 - Arm Processor Families
 - Arm Cortex-M Series
 - Cortex-M4 Processor
 - Arm Processor vs. Arm Architectures
- Arm Cortex-M4 Processor
 - Cortex-M4 Processor Overview
 - Cortex-M4 Block Diagram
 - Cortex-M4 Registers

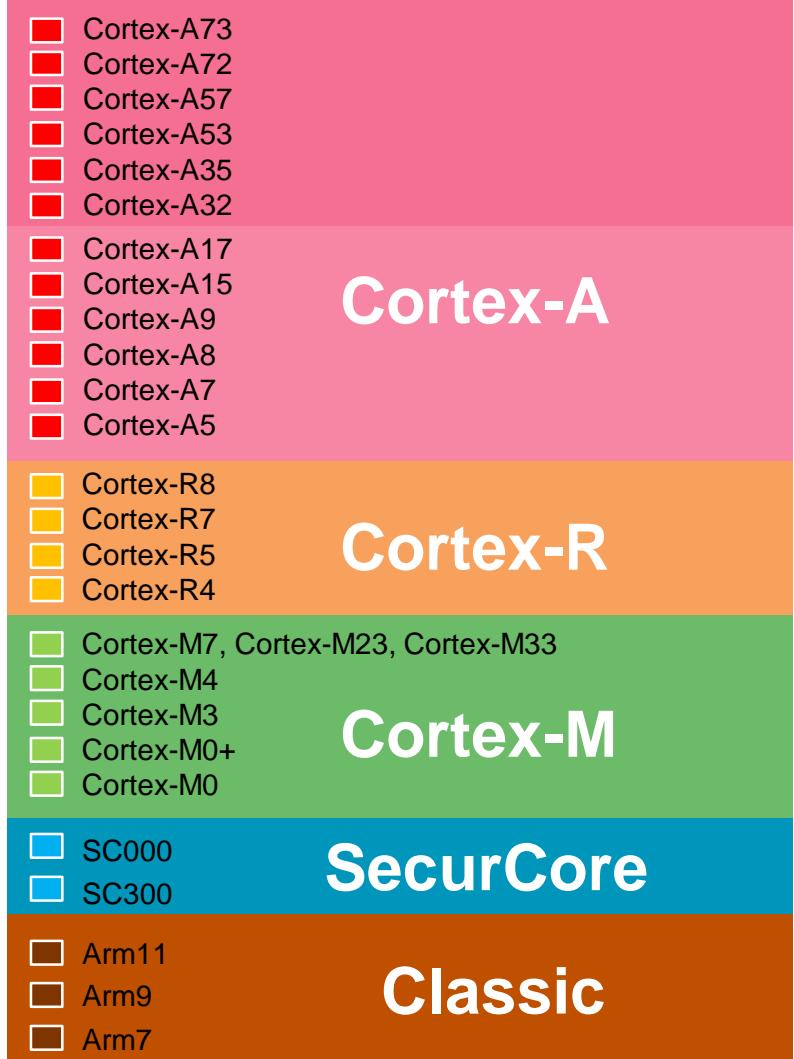
Arm architectures and processors

- Arm architecture is a family of RISC-based processor architectures
 - Well-known for its power efficiency
 - Hence widely used in mobile devices, e.g., smartphones and tablets
 - Designed and licensed by Arm to a wide eco-system of partners
- Arm Holdings
 - The company that designs Arm-based processors
 - Arm does not manufacture, but it licenses designs to semiconductor partners who add their own Intellectual Property (IP) on top of Arm's IP, which they then fabricate and sell to customers.
 - Arm also offers IP other than processors, such as physical IPs, interconnect IPs, graphics cores and development tools.



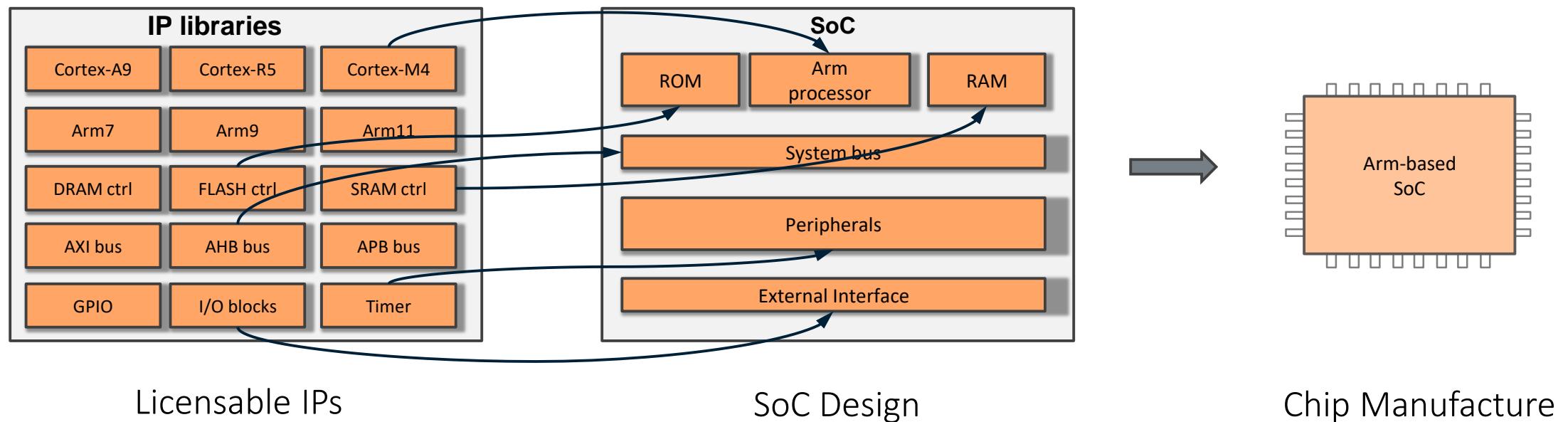
Arm processor families

- Cortex-A series (Application)
 - High performance processors capable of full Operating System (OS) support
 - Applications include smartphones, digital TV, smart books
- Cortex-R series (Real-time)
 - High performance and reliability for real-time applications;
 - Applications include automotive braking system, powertrains
- Cortex-M series (Microcontroller)
 - Cost-sensitive solutions for deterministic microcontroller applications
 - Applications include microcontrollers, smart sensors
- SecurCore series for high security applications
- Earlier classic processors including Arm7, Arm9, Arm11 families



How to design an Arm-based SoC

- Select a set of IP cores from Arm and/or other third-party IP vendors
- Integrate IP cores into a single chip design
- Give design to semiconductor foundries for chip fabrication



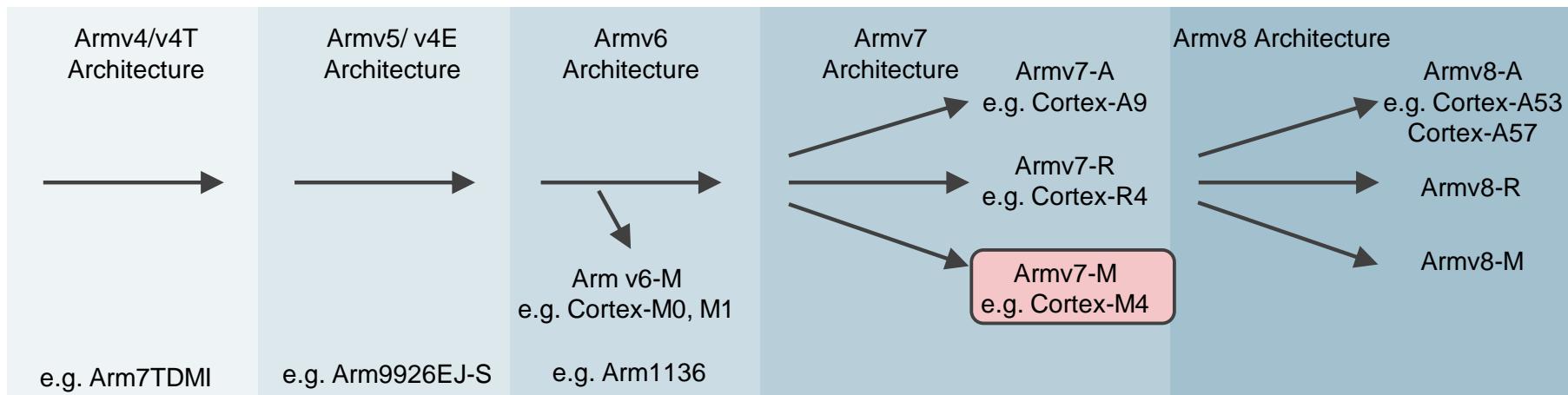
Arm Cortex-M series

- Cortex-M series: Cortex-M0, M0+, M3, M4, M7, M23, M33, M55
- Energy-efficiency
 - Lower energy cost, longer battery life
- Smaller code
 - Lower silicon costs
- Ease of use
 - Faster software development and reuse
- Embedded applications
 - Smart metering, human interface devices, automotive and industrial control systems, white goods, consumer products and medical instrumentation



Arm processors vs. Arm architectures

- Arm architecture
 - Describes the details of instruction set, programmer's model, exception model, and memory map
 - Documented in the Architecture Reference Manual
- Arm processor
 - Developed using one of the Arm architectures
 - More implementation details, such as timing information
 - Documented in processor's Technical Reference Manual



Arm Cortex-M series family

Processor	Arm Architecture	Core Architecture	Thumb®	Thumb®-2	Hardware Multiply	Hardware Divide	Saturated Math	DSP Extensions	Floating Point
Cortex-M0	Armv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M0+	Armv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M3	Armv7-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	No	No
Cortex-M4	Armv7E-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional
Cortex-M7	Armv7E-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional

Cortex-M4 processor overview

- Cortex-M4 Processor
 - Introduced in 2010
 - Designed with a large variety of highly efficient signal processing features
 - Features extended single-cycle multiply accumulate instructions, optimized SIMD arithmetic, saturating arithmetic and an optional Floating Point Unit.
- High performance efficiency
 - 1.25 DMIPS/MHz (Dhrystone Million Instructions Per Second / MHz) at the order of μ Watts / MHz
- Low power consumption
 - Longer battery life – especially critical in mobile products
- Enhanced determinism
 - The critical tasks and interrupt routines can be served quickly in a known number of cycles

Cortex-M4 processor features

- 32-bit Reduced Instruction Set Computing (RISC) processor
- Harvard architecture
 - Separated data bus and instruction bus
- Instruction set
 - Includes the entire Thumb®-1 (16-bit) and Thumb®-2 (16/ 32-bit) instruction sets
- 3-stage + branch speculation pipeline
- Performance efficiency
 - 1.25 – 1.95 DMIPS/MHz (Dhrystone Million Instructions Per Second / MHz)
- Supported interrupts
 - Non-Maskable Interrupt (NMI) + 1 to 240 physical interrupts
 - 8 to 256 interrupt priority levels

Cortex-M4 processor features

- Supports sleep modes
 - Up to 240 wake-up interrupts
 - Integrated WFI (Wait For Interrupt) and WFE (Wait For Event) instructions and sleep on exit capability (to be covered in more detail later)
 - Sleep & deep sleep signals
 - Optional retention mode with Arm Power Management Kit
- Enhanced instructions
 - Hardware divide (2-12 Cycles)
 - Single-cycle 16, 32-bit MAC, single-cycle dual 16-bit MAC
 - 8, 16-bit SIMD arithmetic

Cortex-M4 processor features

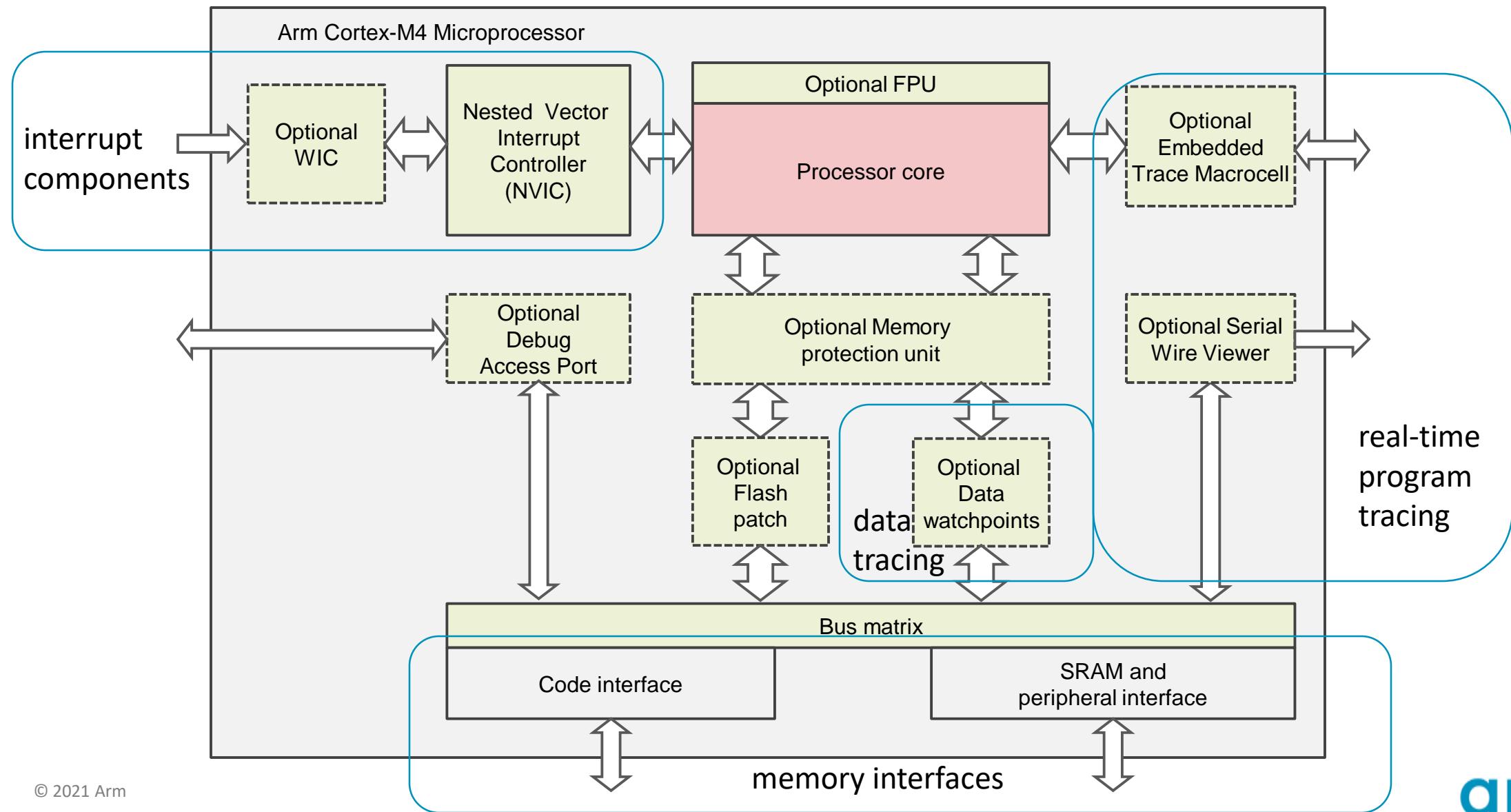
- Debug
 - Optional JTAG & Serial-Wire Debug (SWD) Ports
 - Up to 8 breakpoints and 4 watchpoints
- Memory Protection Unit (MPU)
 - Optional 8 region MPU with sub regions and background region

Cortex-M4 processor features

- The Cortex-M4 processor is designed to meet the challenges of low dynamic power constraints while retaining light footprints
 - 180ULL ultra low power process – 151 µW/MHz
 - 90LP low power process – 32.82 µW/MHz
 - 40LP low power process – 12.26 µW/MHz

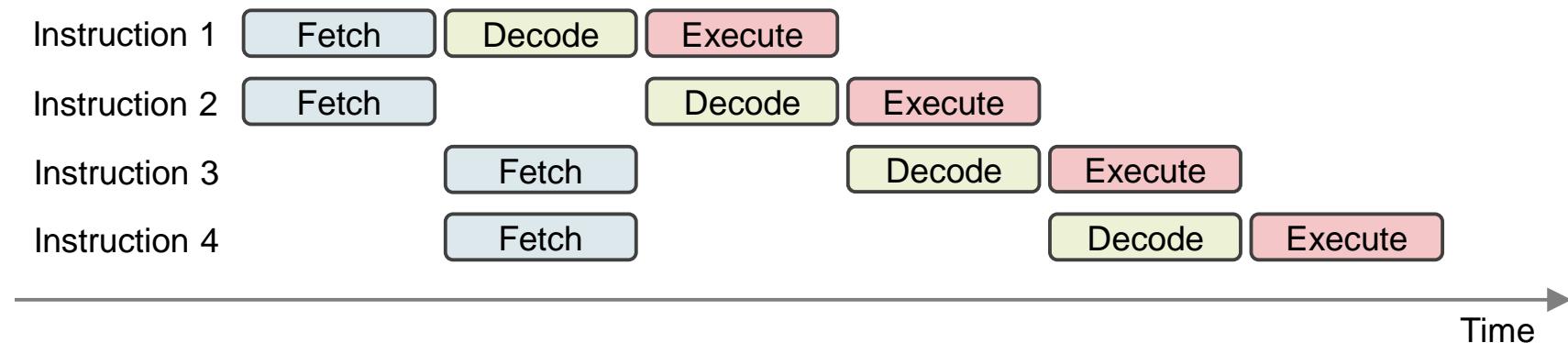
Arm Cortex-M4 Implementation Data			
Process	180ULL (7-track, typical 1.8v, 25C)	90LP (7-track, typical 1.2v, 25C)	40G (9-track, typical 0.9v, 25C)
Dynamic Power	151 µW/MHz	32.82 µW/MHz	12.26 µW/MHz
Floorplanned Area	0.44 mm ²	0.119 mm ²	0.028 mm ²

Cortex-M4 block diagram



Cortex-M4 block diagram

- Processor core
 - Contains internal registers, the ALU, data path, and some control logic
 - Registers include sixteen 32-bit registers for both general and special usage
- Processor pipeline stages
 - Three-stage pipeline: fetch, decode, and execution
 - Some instructions may take multiple cycles to execute, in which case the pipeline will be stalled
 - Speculatively prefetches instructions from branch target addresses
 - Up to two instructions can be fetched in one transfer (16-bit instructions)



Cortex-M4 block diagram

- Nested Vectored Interrupt Controller (NVIC)
 - Up to 240 interrupt request signals and a Non-Maskable Interrupt (NMI)
 - Automatically handles nested interrupts, such as comparing priorities between interrupt requests and the current priority level
- Wakeup Interrupt Controller (WIC)
 - For low-power applications, the microcontroller can enter sleep mode by shutting down most of the components.
 - When an interrupt request is detected, the WIC can inform the power management unit to power up the system.
- Memory protection unit (optional)
 - Used to protect memory content, e.g., make some memory regions read-only or preventing user applications from accessing privileged application data

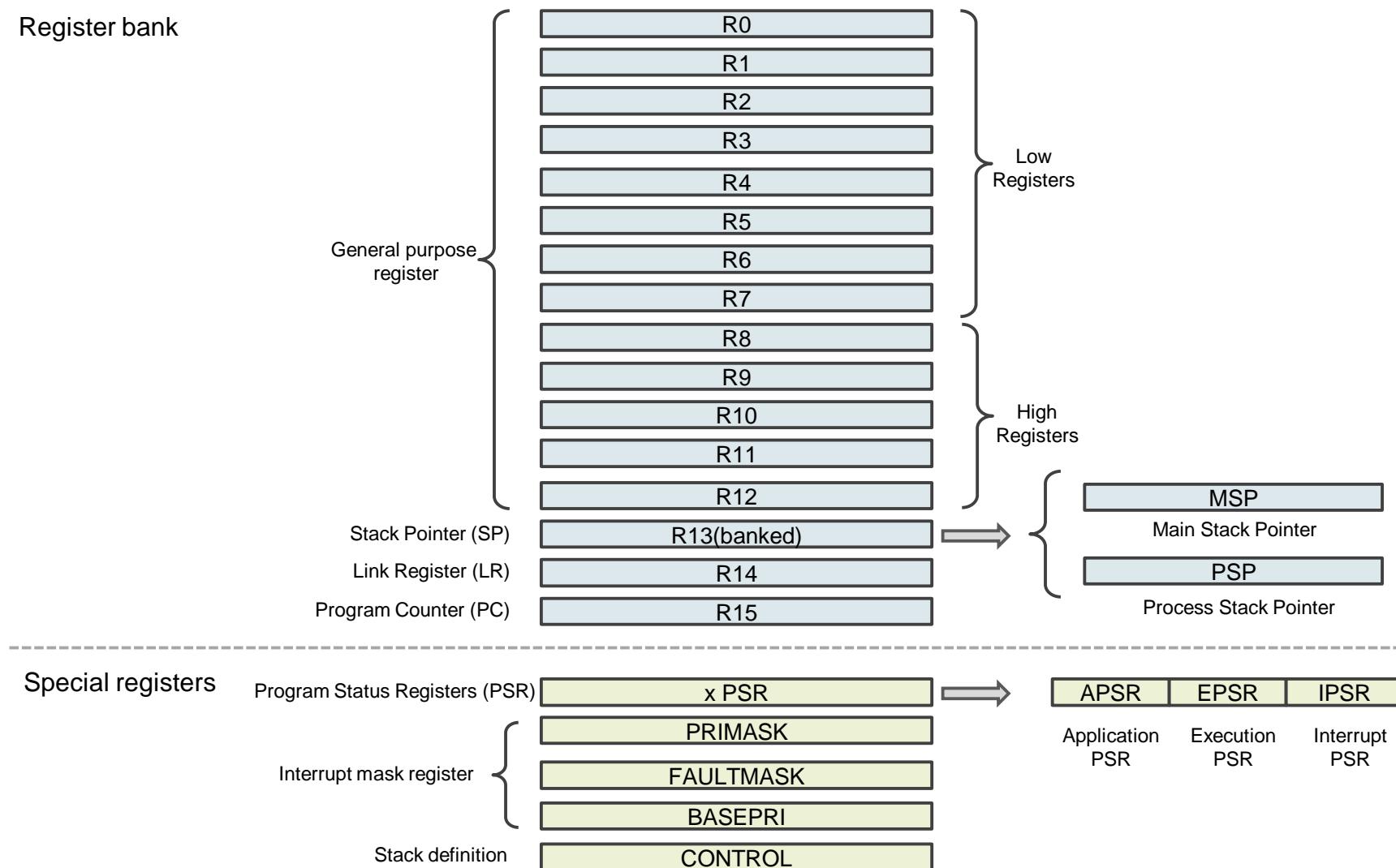
Cortex-M4 block diagram

- Bus interconnect
 - Allows data transfer to take place on different buses simultaneously
 - Provides data transfer management, e.g. write buffer, bit-oriented operations (bit-band)
 - May include bus bridges (e.g. AHB-to-APB bus bridge) to connect different buses into a network using a single global memory space
 - Includes the internal bus system, the data path in the processor core, and the AHB LITE interface unit
- Debug subsystem
 - Handles debug control, program breakpoints, and data watchpoints
 - When a debug event occurs, it can put the processor core in a halted state, so developers can analyse the status of the processor, such as register values and flags, at that point.

Arm Cortex-M4 processor registers

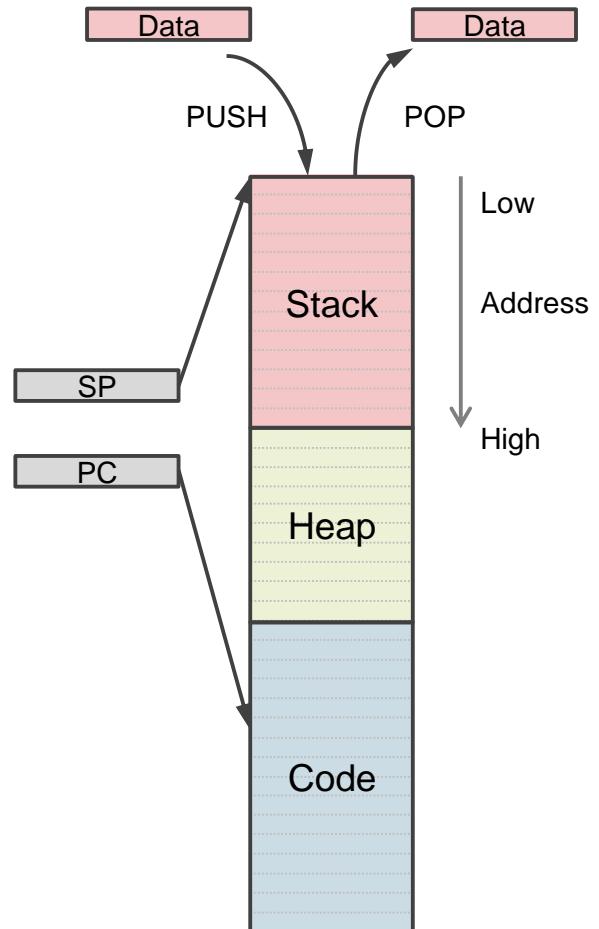
- Processor registers
 - The internal registers are used to store and process temporary data within the processor core
 - All registers are inside the processor core, hence they can be accessed quickly
 - Load-store architecture
 - To process memory data, they have to be first loaded from memory to registers, processed inside the processor core using register data only, and then written back to memory if needed
- Cortex-M4 registers
 - Register bank
 - Sixteen 32-bit registers (thirteen are used for general-purpose)
 - Special registers

Cortex-M4 registers



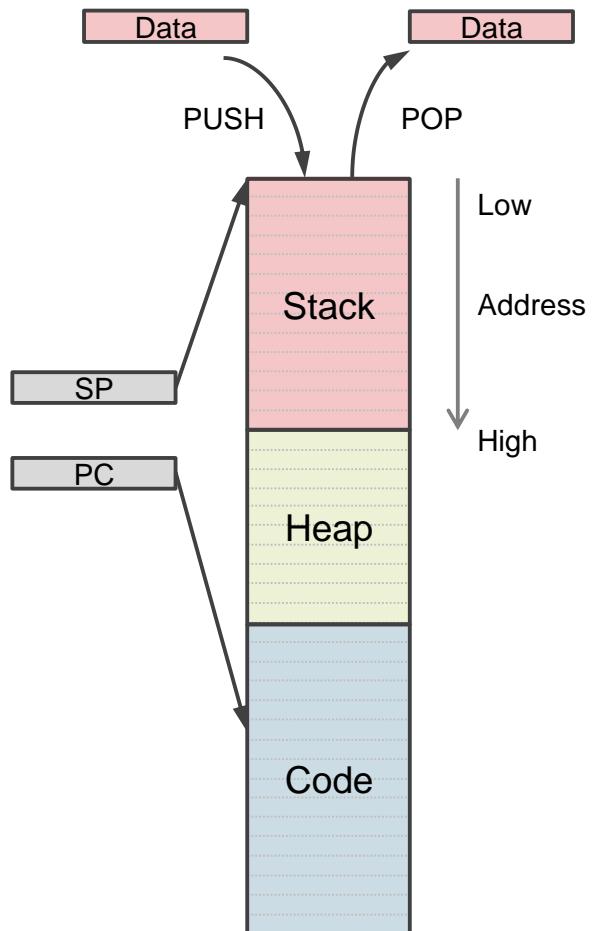
Cortex-M4 registers

- R0 – R12: general purpose registers
 - Low registers (R0 – R7) can be accessed by any instruction
 - High registers (R8 – R12) sometimes cannot be accessed e.g. by some Thumb (16-bit) instructions
- R13: Stack Pointer (SP)
 - Records the current address of the stack
 - Used for saving the context of a program while switching between tasks
 - Cortex-M4 has two SPs: Main SP, used in applications that require privileged access e.g. OS kernel, and Process SP, used in base-level application code (when not running an exception handler)



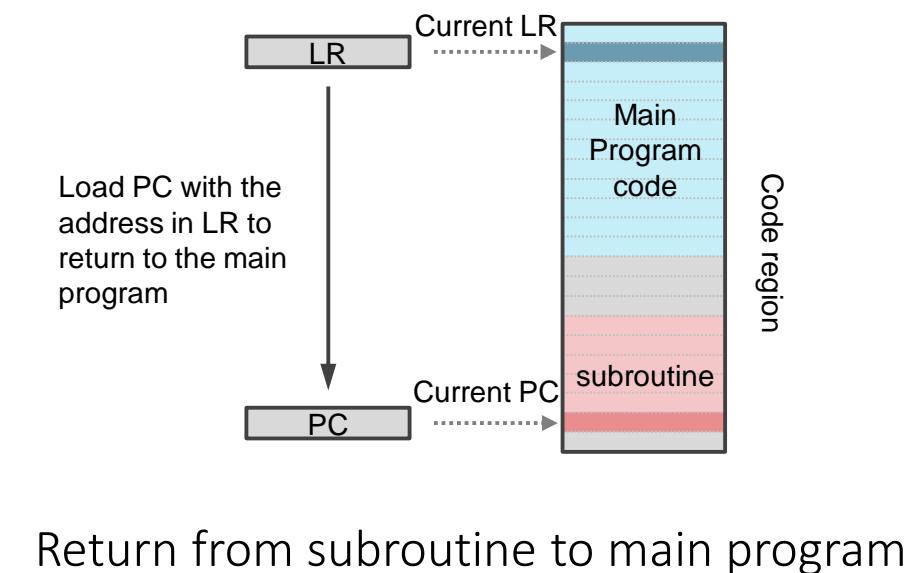
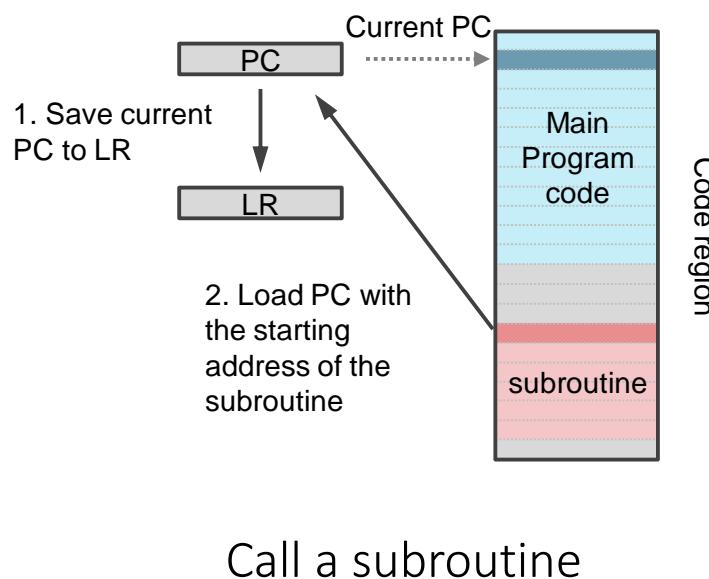
Cortex-M4 registers

- Program Counter (PC)
 - Records the address of the current instruction code
 - Automatically incremented by four at each operation (for 32-bit instruction code), except branching operations
 - A branching operation, such as function calls, will change the PC to a specific address, while saving the current PC to the Link Register (LR)



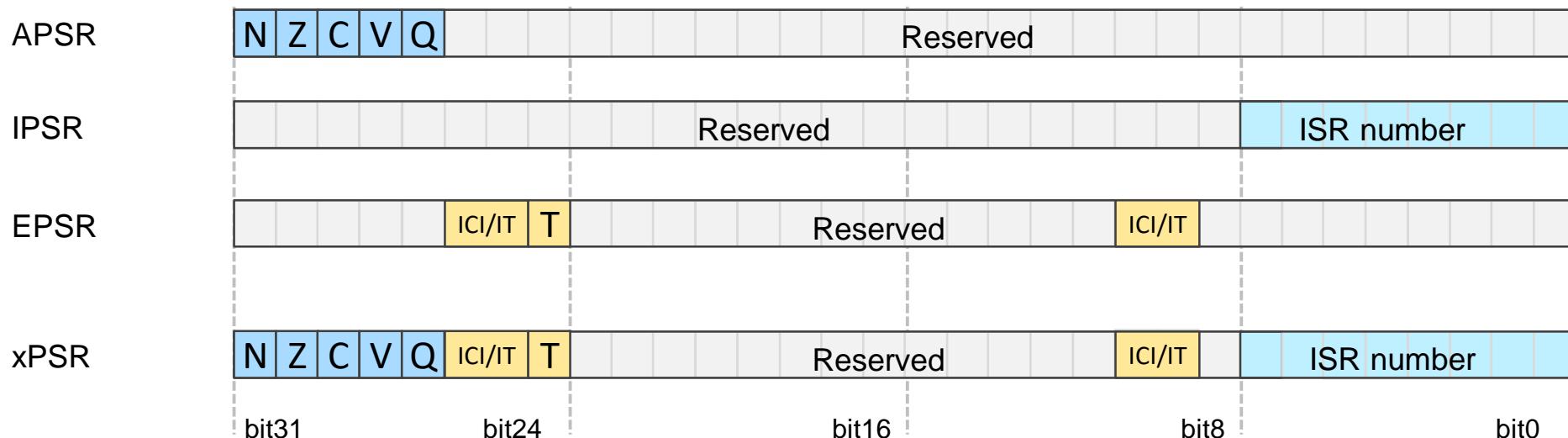
Cortex-M4 registers

- R14: Link Register (LR)
 - The LR is used to store the return address of a subroutine or a function call
 - The program counter (PC) will load the value from LR after a function is finished



Cortex-M4 registers

- xPSR, combined Program Status Register
 - Provides information about program execution and ALU flags
 - Application PSR (APSR)
 - Interrupt PSR (IPSR)
 - Execution PSR (EPSR)



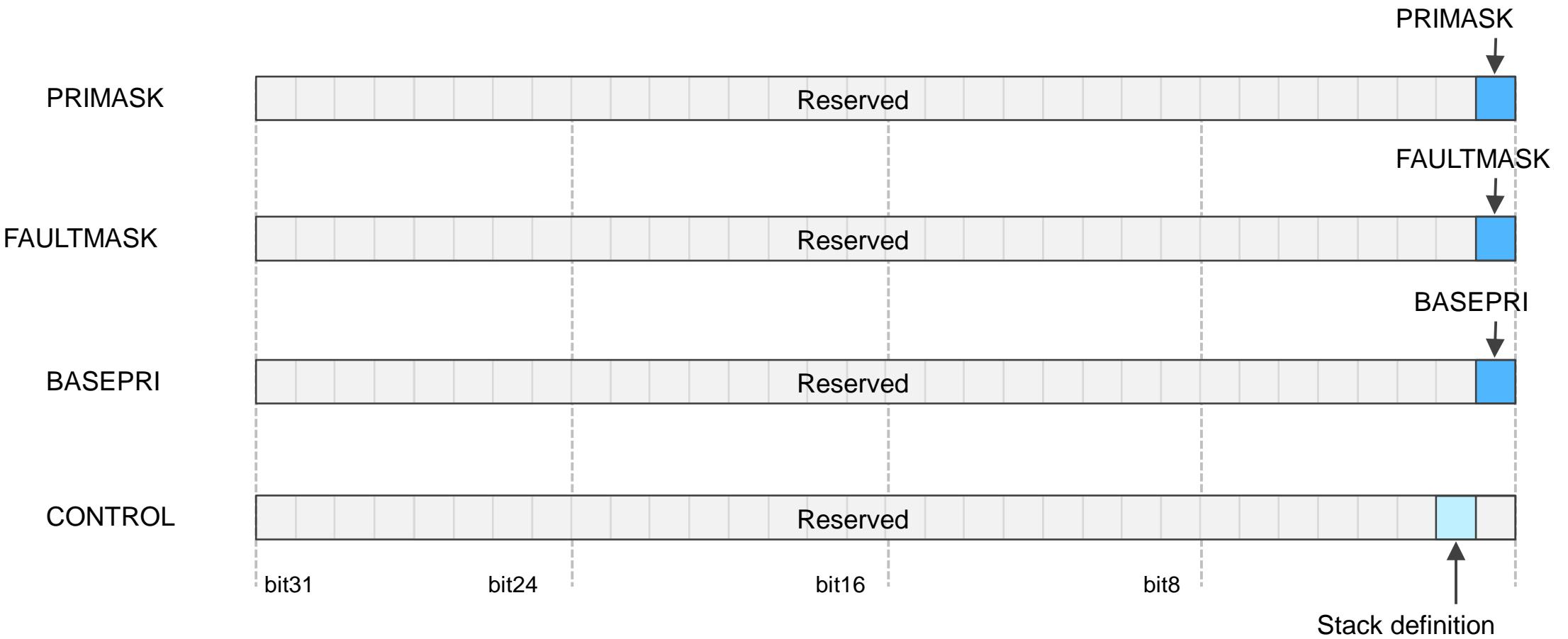
Cortex-M4 registers

- APSR
 - N: negative flag – set to one if the result from ALU is negative
 - Z: zero flag – set to one if the result from ALU is zero
 - C: carry flag – set to one if an unsigned overflow occurs
 - V: overflow flag – set to one if a signed overflow occurs
 - Q: sticky saturation flag – set to one if saturation has occurred in saturating arithmetic instructions, or overflow has occurred in certain multiply instructions
- IPSR
 - ISR number – current executing interrupt service routine number
- EPSR
 - T: Thumb state – always one since Cortex-M4 only supports the Thumb state (more on processor states in the next module)
 - IC/IT: Interrupt-Continuable Instruction (ICI) bit, IF-THEN instruction status bit

Cortex-M4 registers

- Exception mask registers
 - 1-bit PRIMASK
 - If set to one, will block all the interrupts apart from non-maskable interrupt (NMI) and the hard fault exception
 - 1-bit FAULTMASK
 - If set to one, will block all the interrupts apart from NMI
 - 1-bit BASEPRI
 - If set to one, will block all interrupts of the same or lower level (only allowing for interrupts with higher priorities)
- CONTROL: special register
 - 1-bit stack definition
 - Set to one to use the process stack pointer (PSP)
 - Clear to zero to use the main stack pointer (MSP)

Cortex-M4 registers



Useful resources

- Architecture Reference Manual:
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403c/index.html>
- Cortex-M4 Technical Reference Manual:
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439d/DDI0439D_cortex_m4_processor_r0p1_trm.pdf
- Cortex-M4 Devices Generic User Guide:
http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf



The Arm Cortex-M4 Processor Architecture

Learning Objectives

At the end of this lecture, you should be able to:

- Outline the Cortex-M4 processor memory map and its memory regions including their functions.
- Describe bit-band operation and describe its benefits.
- Define Endianness and the concepts of Little-endian and big-endian.
- Explain key features of the Thumb instruction sets.

Module syllabus

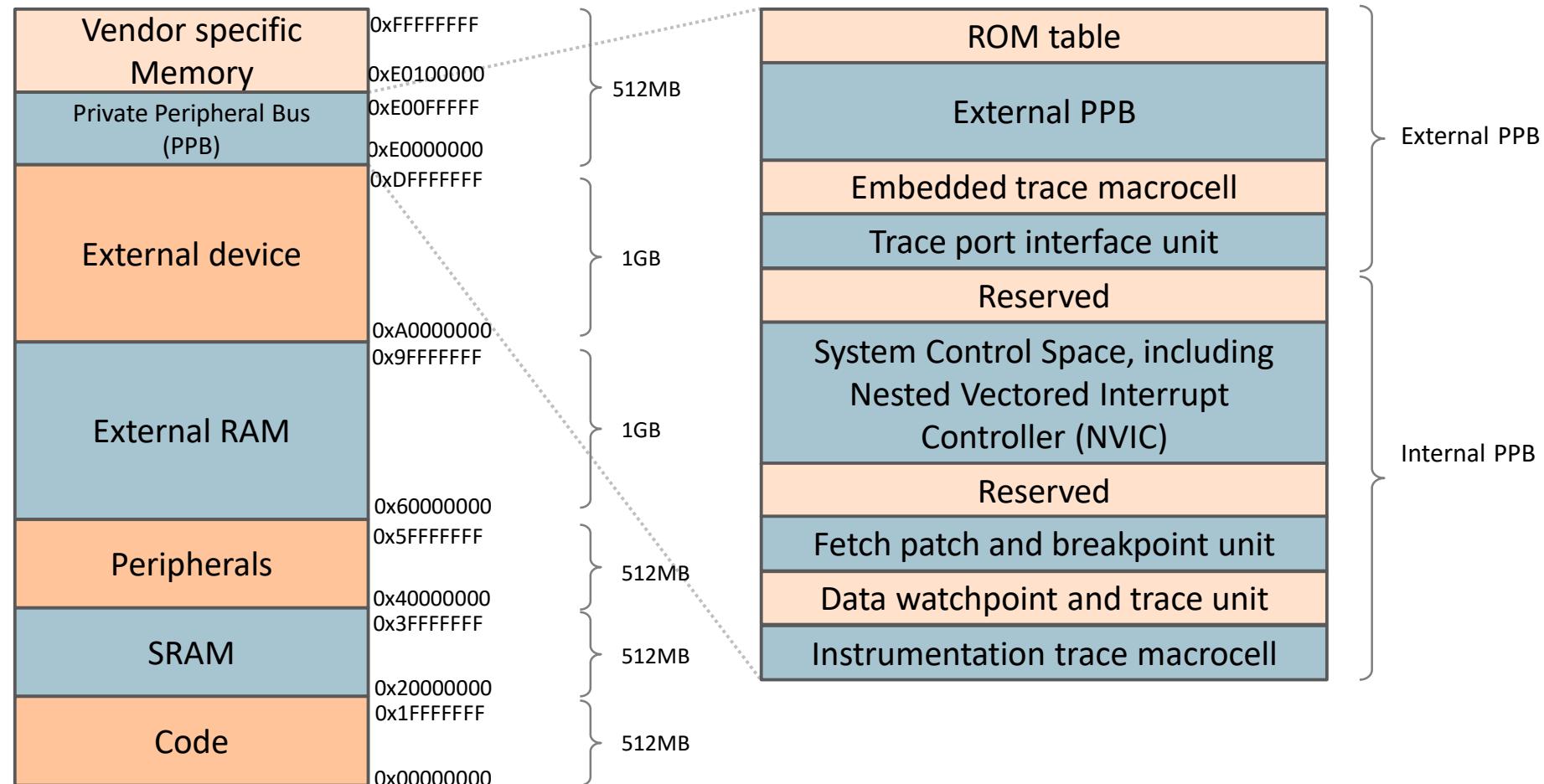
- Cortex-M4 Memory Map
 - Cortex-M4 Memory Map
 - Bit-band Operations
 - Cortex-M4 Program Image and Endianness
- Arm Cortex-M4 Processor Instruction Set
 - Arm and Thumb Instruction Set
 - Cortex-M4 Instruction Set

Arm Cortex-M4 memory map

- The Cortex-M4 processor has 4 GB of memory address space
 - Support for bit-band operation (detailed later)
- The 4GB memory space is architecturally defined with a number of regions
 - Each region is designed for particular recommended uses
 - Easy for software programmer to port between different devices
- Note that, despite the default definitions, the actual usage of the memory map can also be flexibly defined by the user, apart from some fixed memory addresses such as the internal private peripheral bus.

Arm Cortex-M4 memory map

- Reserved for other purposes
- Private peripherals
e.g. NVIC, SCS
- Mainly used for external peripherals
e.g. SD card
- Mainly used for external memories
e.g. external DDR, FLASH, LCD
- Mainly used for on-chip peripherals
e.g. AHB, APB peripherals
- Mainly used for data memory
e.g. on-chip SRAM, SDRAM
- Mainly used for program code
e.g. on-chip FLASH



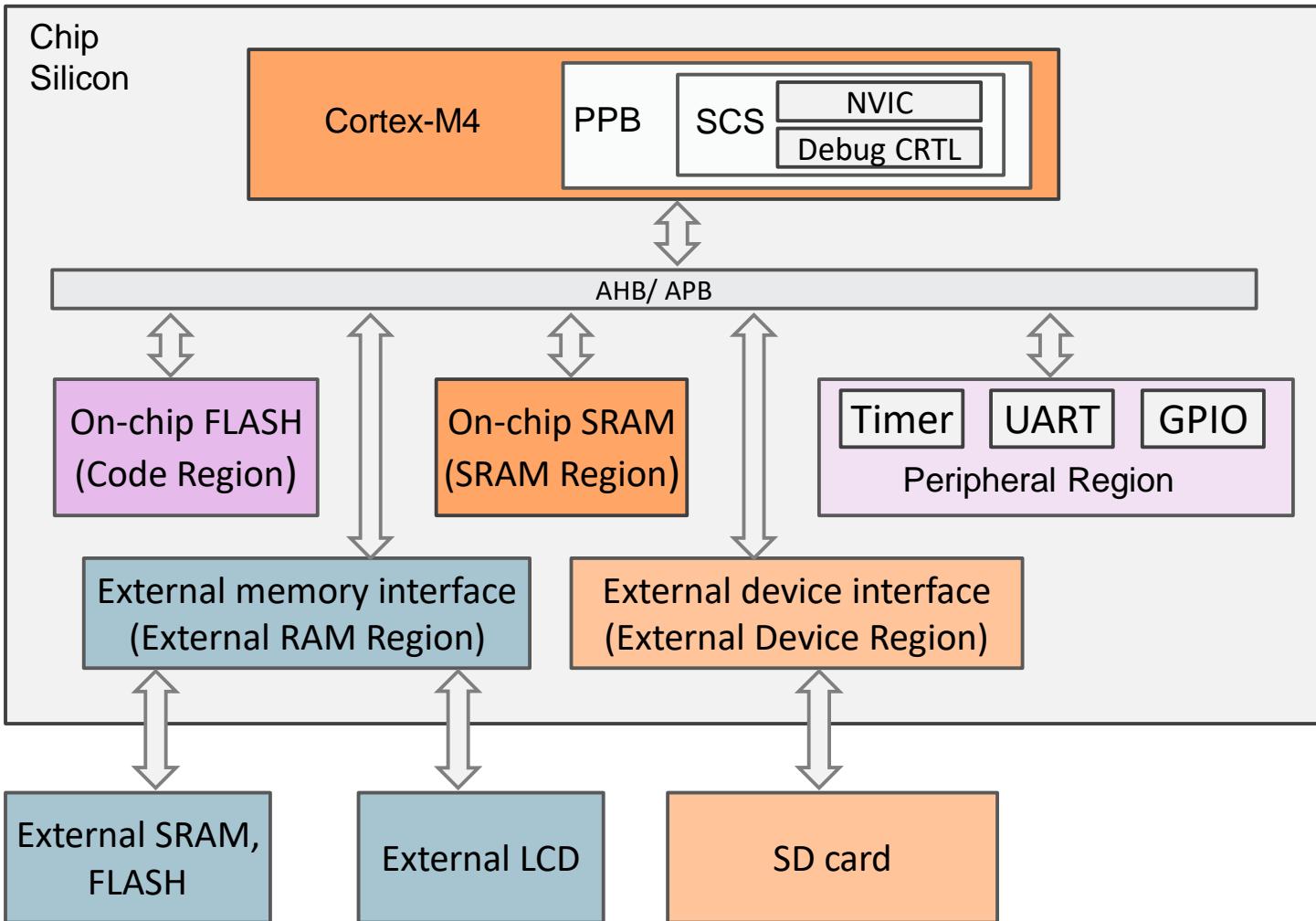
Arm Cortex-M4 memory map

- Code region
 - Primarily used to store program code
 - Can also be used for data memory
 - On-chip memory, such as on-chip FLASH
- SRAM region
 - Primarily used to store data, such as heaps and stacks
 - Can also be used for program code
 - On-chip memory; despite its name “SRAM”, the actual device could be SRAM, SDRAM, etc
- Peripheral region
 - Primarily used for peripherals, such as Advanced High-performance Bus (AHB) or Advanced Peripheral Bus (APB) peripherals
 - On-chip peripherals

Arm Cortex-M4 memory map

- External RAM region
 - Primarily used to store large data blocks or memory caches
 - Off-chip memory, slower than on-chip SRAM region
- External device region
 - Primarily used to map to external devices
 - Off-chip devices, such as SD card
- Private Peripheral Bus (PPB)
 - Provides access to internal and external processor resources

Cortex-M4 memory map example



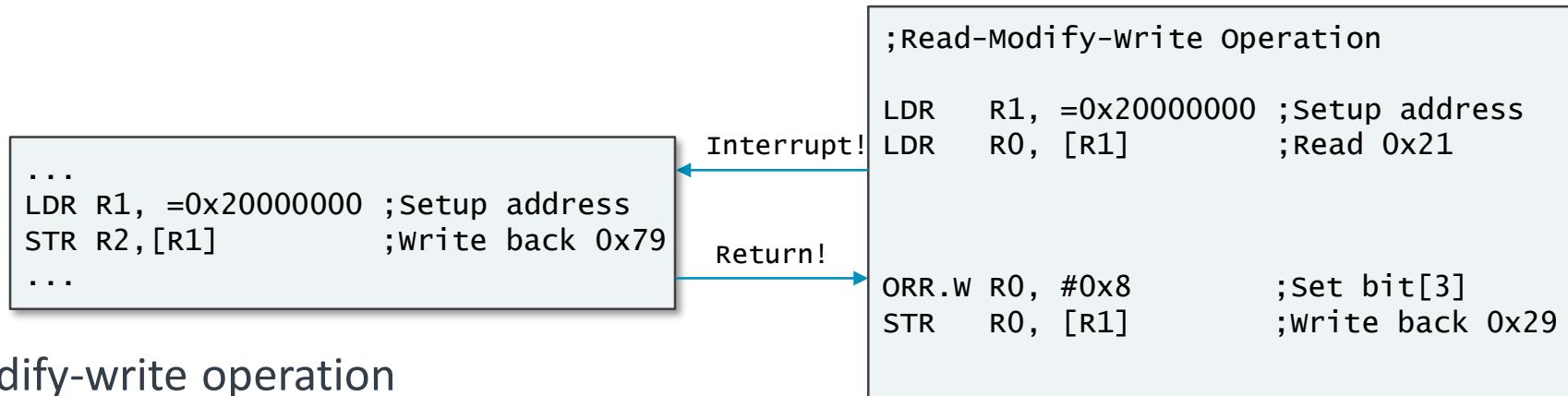
Vendor specific Memory	0xFFFFFFFF
Private Peripheral Bus (PPB)	0xE0100000 0xE00FFFFF
External device	0xE0000000 0xDFFFFFFF
External RAM	0xA0000000 0x9FFFFFFF
Peripherals	0x60000000 0x5FFFFFFF
SRAM	0x40000000 0x3FFFFFFF
Code	0x20000000 0x1FFFFFFF
	0x00000000

Bit-band operations

- Bit-band operations allow a single load/store operation to access a single bit in the memory, for example, to change a single bit of one 32-bit data:
 - Normal operation without bit-band (read-modify-write)
 - Read the value of 32-bit data
 - Modify a single bit of the 32-bit value (keep other bits unchanged)
 - Write the value back to the address
 - Bit-band operation
 - Directly write a single bit (0 or 1) to the “bit-band alias address” of the data
- Bit-band alias address
 - Each bit-band alias address is mapped to a real data address
 - When writing to the bit-band alias address, only a single bit of the data will be changed

Bit-band operation example

- For example, in order to set bit[3] in word data in address 0x20000000:



- Read-modify-write operation
 - Reads the data (0x21) from the address 0x20000000
 - The interrupt changes the data of 0x20000000 address to 0x79 and then returns
 - Writes back the modified old data back
 - 0x79 has been lost!

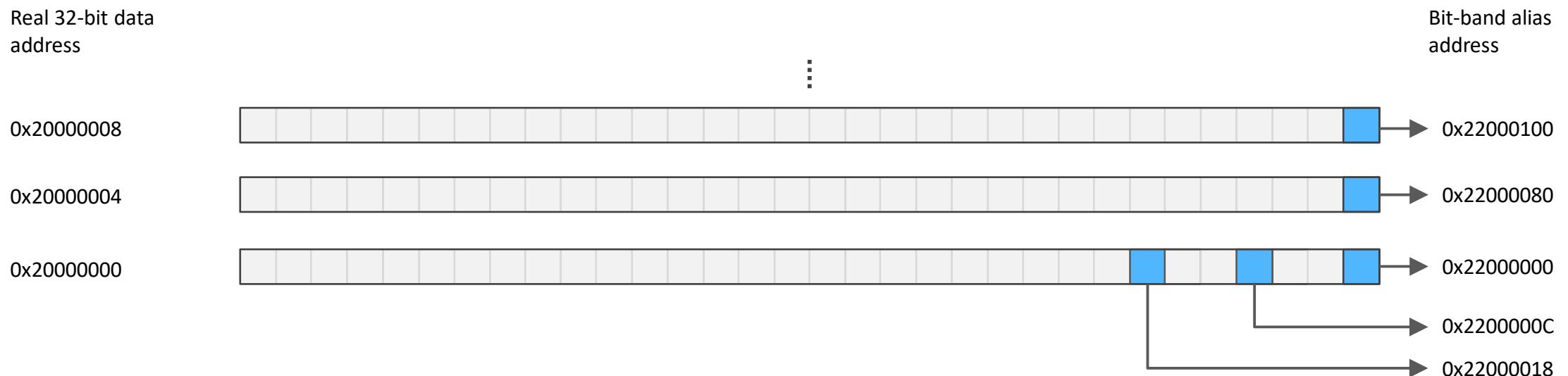
Bit-band operation example

```
;Bit-band Operation  
LDR    R1, =0x2200000C ;Setup address  
MOV    R0, #1           ;Load data  
STR    R0, [R1]          ;Write
```

- Bit-band operation
 - Directly set the bit by writing ‘1’ to address 0x2200000C, which is the alias address of the fourth bit of the 32-bit data at 0x20000000
 - In effect, this single instruction is mapped to 2 bus transfers: read data from 0x20000000 to the buffer, and then write to 0x20000000 from the buffer with bit [3] set

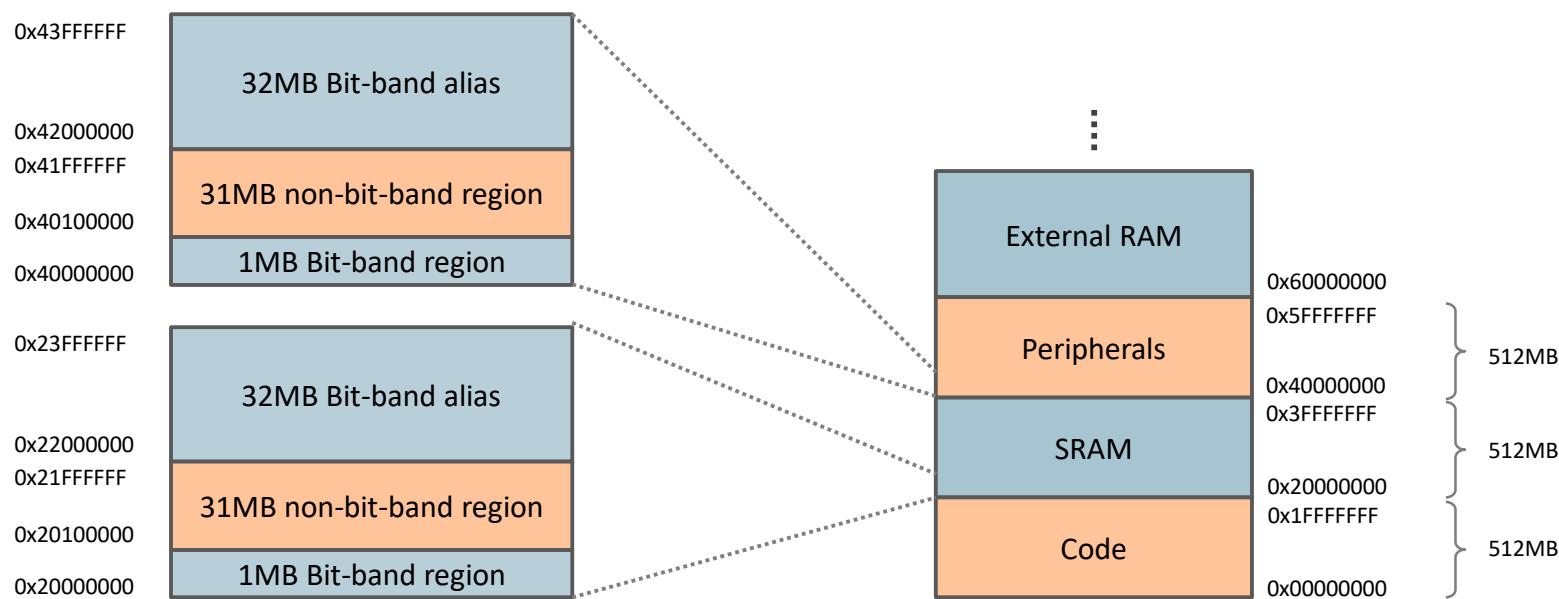
Bit-band alias address

- Each bit of the 32-bit data is one-to-one mapped to the bit-band alias address
 - For example, the fourth bit (bit [3]) of the data at 0x20000000 is mapped to the bit-band alias address at 0x2200000C
 - Hence, to set bit [3] of the data at 0x20000000, we only need to write '1' to address 0x2200000C
 - In Cortex-M4, there are two pre-defined bit-band alias regions: one for SRAM region, and one for peripherals region



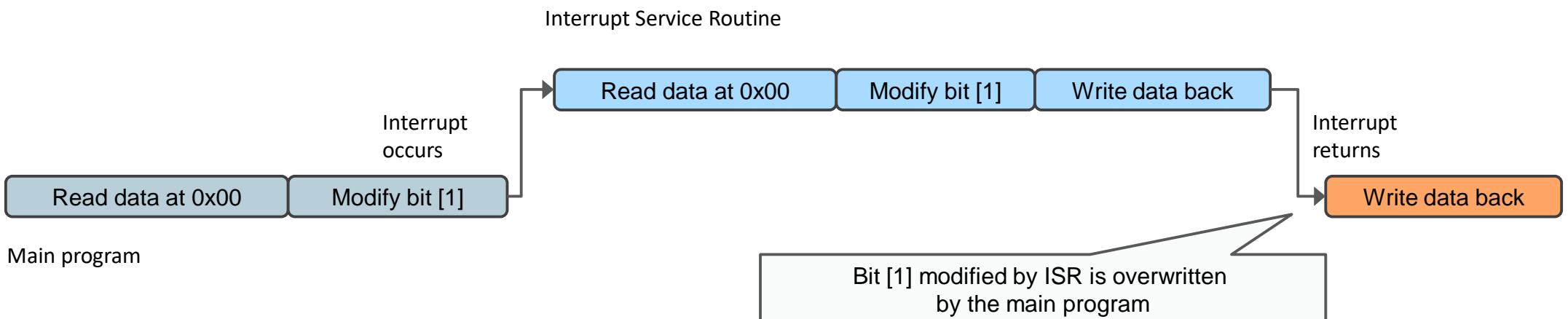
Bit-band alias address

- SRAM region
 - 32MB memory space (0x22000000 – 0x23FFFFFF) is used as the bit-band alias region for 1MB data (0x20000000 – 0x200FFFFF)
- Peripherals region
 - 32MB memory space (0x42000000 – 0x43FFFFFF) is used as the bit-band alias region for 1MB data (0x40000000 – 0x400FFFFF)



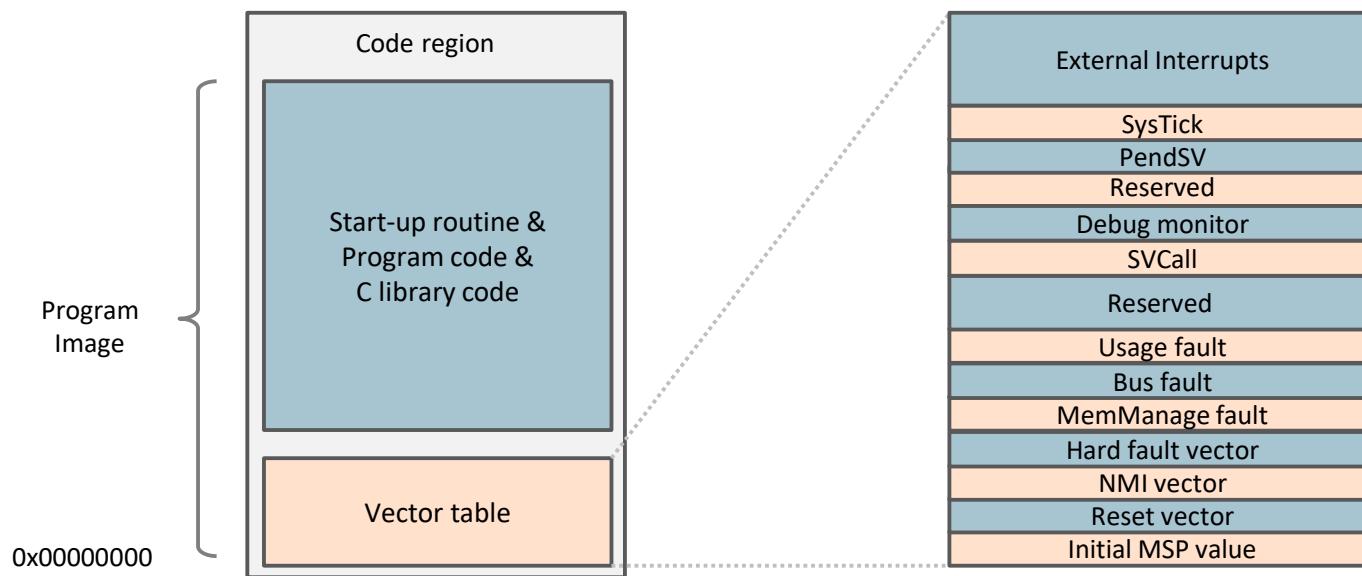
Benefits of bit-band operations

- Faster bit operations
- Fewer instructions
- Atomic operation, avoid hazards
 - For example, if an interrupt is triggered and served during the read-modify-write operations, and the interrupt service routine modifies the same data, a data conflict will occur



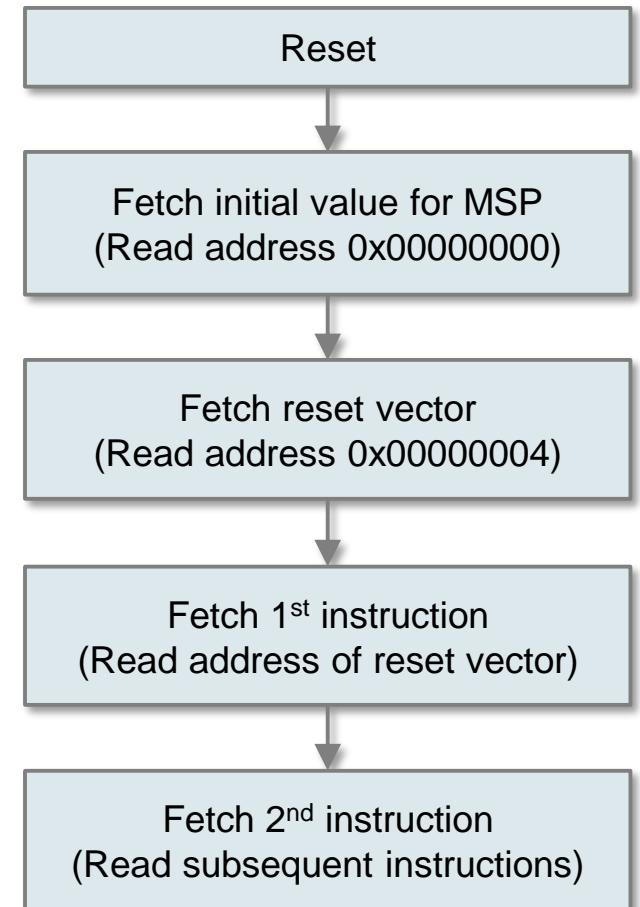
Cortex-M4 program image

- The program image in Cortex-M4 contains
 - Vector table – includes the starting addresses of exceptions (vectors) and the value of the main stack point (MSP)
 - C start-up routine
 - Program code – application code and data
 - C library code – program codes for C library functions



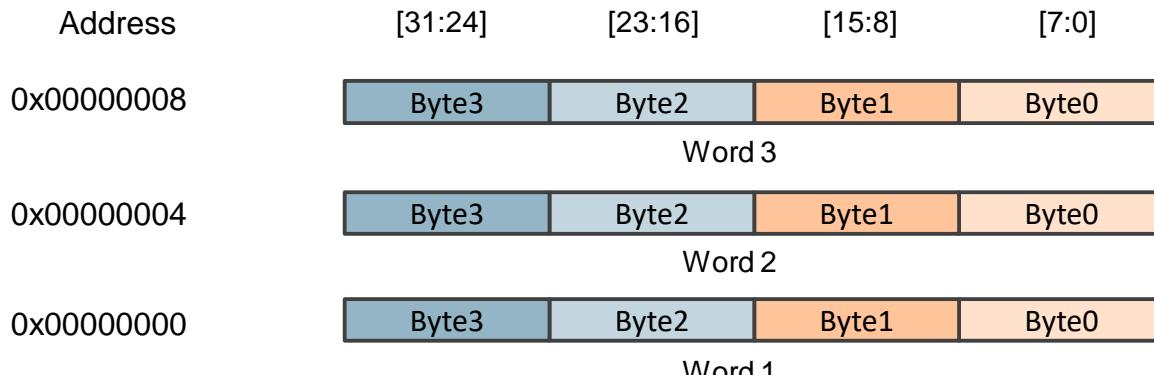
Cortex-M4 program image

- After reset, the processor:
 - First reads the initial MSP value
 - Then reads the reset vector
 - Then branches to the start of the program execution address (reset handler)
 - Subsequently executes program instructions

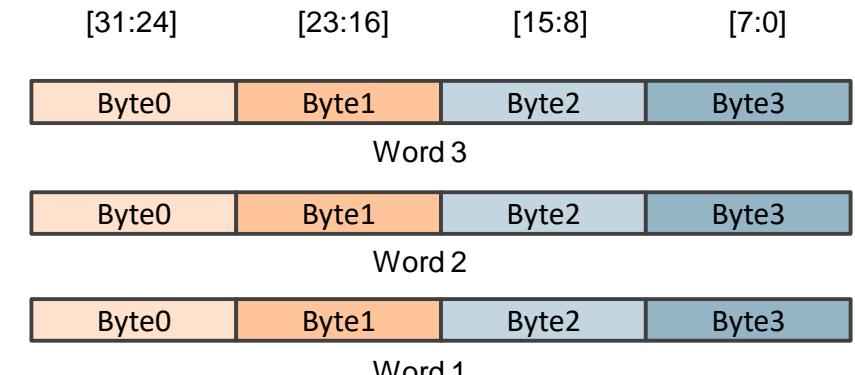


Cortex-M4 endianness

- Endian refers to the order of bytes stored in memory
 - Big endian: lowest byte of a word-size data is stored in bit 0 to bit 7
 - Big endian: lowest byte of a word-size data is stored in bit 24 to bit 31
- Cortex-M4 supports both little endian and big endian
- However, endianness only exists in the hardware level



Little endian 32-bit memory



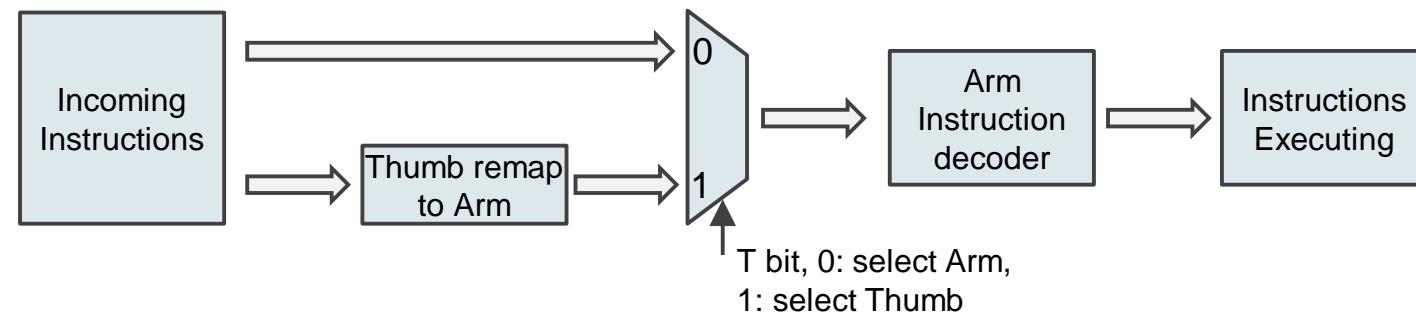
Big endian 32-bit memory

Arm Cortex-M4 processor instruction set and Thumb instruction set

- Early Arm instruction set
 - 32-bit instruction set, called the Arm instructions
 - Powerful and good performance
 - Larger program memory compared to 8-bit and 16-bit processors
 - Larger power consumption
- Thumb-1 instruction set
 - 16-bit instruction set, first used in Arm7TDMI processor in 1995
 - Provides subset of Arm instructions, with better code density compared to 32-bit RISC architecture
 - Code size is reduced by ~30%, but performance is also reduced by ~20%

Arm and Thumb instruction set

- Mix of Arm and Thumb-1 Instruction sets
 - Benefit from both 32-bit Arm (high performance) and 16-bit Thumb-1 (high code density)
 - A multiplexer is used to switch between two states: Arm state (32-bit) and Thumb state (16-bit), which requires a switching overhead



- Thumb-2 instruction set
 - Consists of both 32-bit Thumb instructions and original 16-bit Thumb-1 instruction sets
 - Compared to 32-bit Arm instructions set, code size is reduced by ~26%, with similar performance
 - Capable of handling all processing requirements in one operation state

Cortex-M4 instruction set

- Cortex-M4 processor
 - Armv7-M architecture
 - Supports 32-bit Thumb-2 instructions
 - Can handle all processing requirements in one operation state (Thumb state)
 - Compared with traditional Arm processors (which use state switching), advantages include:
 - No state switching overhead – both execution time and instruction space are saved
 - No need to separate Arm code and Thumb code source files, which makes the development and maintenance of software easier
 - Easier to get optimized efficiency and performance

Cortex-M4 instruction set

- Arm assembly syntax:

label

mnemonic *operand1*, *operand2, ...* ; *Comments*

- Label is used as a reference to an address location
- Mnemonic is the name of the instruction
- Operand1 is the destination of the operation
- Operand2 is normally the source of the operation
- Comments are written after “;”, which does not affect the program, e.g.:

MOVS R3, #0x11 ;*Set register R3 to 0x11*

- Assembly code can be assembled by either Arm assembler (`armasm`) or assembly tools from a variety of vendors (e.g. GNU tool chain). When using the GNU tool chain, the syntax for labels and comments is slightly different.

Arm Cortex M4 Instruction Set - Overview

Instructions supported by the Cortex-M4 processor can be grouped as follows:

- Memory access instructions
- General data processing instructions
- Multiply and divide instructions
- Saturating instructions
- Packing and unpacking instructions
- Bitfield instructions
- Branch and control instructions
- Miscellaneous instructions
- Floating-point instructions

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	N,Z,C,V
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V
ADR	Rd, label	Load PC-relative Address	
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C
ASR, ASRS	Rd, Rm, <Rs #n>	Arithmetic Shift Right	N,Z,C
B	label	Branch	
BFC	Rd, #lsb, #width	Bit Field Clear	
BFI	Rd, Rn, #lsb, #width	Bit Field Insert	
BIC, BICS	{Rd,} Rn, Op2	Bit Clear	N,Z,C
BKPT	#imm	Breakpoint	
BL	label	Branch with Link	
BLX	Rm	Branch indirect with Link	
BX	Rm	Branch indirect	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
CBNZ	Rn, label	Compare and Branch if Non Zero	
CBZ	Rn, label	Compare and Branch if Zero	
CLREX		Clear Exclusive	
CLZ	Rd, Rm	Count Leading Zeros	
CMN	Rn, Op2	Compare Negative	N,Z,C,V
CMP	Rn, Op2	Compare	N,Z,C,V
CPSID	i	Change Processor State, Disable Interrupts	
CPSIE	i	Change Processor State, Enable Interrupts	
DMB		Data Memory Barrier	
DSB		Data Synchronization Barrier	
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C
ISB	-	Instruction Synchronization Barrier	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
IT		If-Then condition block	
LDM	Rn{!}, reglist	Load Multiple registers, increment after	
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple registers, decrement before	
LDMFD, LDMIA	Rn{!}, reglist	Load Multiple registers, increment after	
LDR	Rt, [Rn, #offset]	Load Register with word	
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register with byte	
LDRD	Rt, Rt2, [Rn, #offset]	Load Register with two bytes	
LDREX	Rt, [Rn, #offset]	Load Register Exclusive	
LDREXB	Rt, [Rn]	Load Register Exclusive with Byte	
LDREXH	Rt, [Rn]	Load Register Exclusive with Halfword	
LDRH, LDRHT	Rt, [Rn, #offset]	Load Register with Halfword	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load Register with Signed Byte	
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load Register with Signed Halfword	
LDRT	Rt, [Rn, #offset]	Load Register with word	
LSL, LSLS	Rd, Rm, <Rs #n>	Logical Shift Left	N,Z,C
LSR, LSRS	Rd, Rm, <Rs #n>	Logical Shift Right	N,Z,C
MLA	Rd, Rn, Rm, Ra	Multiply with Accumulate, 32-bit result	
MLS	Rd, Rn, Rm, Ra	Multiply and Subtract, 32-bit result	
MOV, MOVS	Rd, Op2	Move	N,Z,C
MOVT	Rd, #imm16	Move Top	
MOVW, MOV	Rd, #imm16	Move 16-bit constant	N,Z,C
MRS	Rd, spec_reg	Move from Special Register to general register	
MSR	spec_reg, Rm	Move from general register to Special Register	N,Z,C,V

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C
NOP		No Operation	
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C
PKHTB, PKHBT	{Rd, } Rn, Rm, Op2	Pack Halfword	
POP	reglist	Pop registers from stack	
PUSH	reglist	Push registers onto stack	
QADD	{Rd, } Rn, Rm	Saturating double and Add	Q
QADD16	{Rd, } Rn, Rm	Saturating Add 16	
QADD8	{Rd, } Rn, Rm	Saturating Add 8	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
QASX	{Rd, } Rn, Rm	Saturating Add and Subtract with Exchange	
QDADD	{Rd, } Rn, Rm	Saturating Add	Q
QDSUB	{Rd, } Rn, Rm	Saturating double and Subtract	Q
QSAX	{Rd, } Rn, Rm	Saturating Subtract and Add with Exchange	
QSUB	{Rd, } Rn, Rm	Saturating Subtract	Q
QSUB16	{Rd, } Rn, Rm	Saturating Subtract 16	
QSUB8	{Rd, } Rn, Rm	Saturating Subtract 8	
RBIT	Rd, Rn	Reverse Bits	
REV	Rd, Rn	Reverse byte order in a word	
REV16	Rd, Rn	Reverse byte order in each halfword	
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	
ROR, RORS	Rd, Rm, <Rs #n>	Rotate Right	N,Z,C

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
RRX, RRXS	Rd, Rm	Rotate Right with Extend	N,Z,C
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract	N,Z,C,V
SADD16	{Rd, } Rn, Rm	Signed Add 16	GE
SADD8	{Rd, } Rn, Rm	Signed Add 8	GE
SASX	{Rd, } Rn, Rm	Signed Add and Subtract with Exchange	GE
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry	N,Z,C,V
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract	
SDIV	{Rd,} Rn, Rm	Signed Divide	
SEV		Send Event	
SHADD16	{Rd,} Rn, Rm	Signed Halving Add 16	
SHADD8	{Rd,} Rn, Rm	Signed Halving Add 8	
SHASX	{Rd,} Rn, Rm	Signed Halving Add and Subtract with Exchange	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
SHSAX	{Rd,} Rn, Rm	Signed Halving Subtract and Add with Exchange	
SHSUB16	{Rd,} Rn, Rm	Signed Halving Subtract 16	
SHSUB8	{Rd,} Rn, Rm	Signed Halving Subtract 8	
SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate Long (halfwords)	Q
SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed Multiply Accumulate Dual	Q
SMLAL	RdLo, RdHi, Rn, Rm	Signed Multiply with Accumulate (32 x 32 + 64), 64-bit result	
SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long, halfwords	
SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long Dual	
SMLAWB, SMLAWT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate, word by halfword	Q
SMLSD	Rd, Rn, Rm, Ra	Signed Multiply Subtract Dual	Q
SMLSDF	RdLo, RdHi, Rn, Rm	Signed Multiply Subtract Long Dual	
SMMLA	Rd, Rn, Rm, Ra	Signed Most significant word Multiply Accumulate	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
SMMLS, SMMLR	Rd, Rn, Rm, Ra	Signed Most significant word Multiply Subtract	
SMMUL, SMMULR	{Rd,} Rn, Rm	Signed Most significant word Multiply	
SMUAD	{Rd,} Rn, Rm	Signed dual Multiply Add	Q
SMULBB, SMULBT SMULTB, SMULTT	{Rd,} Rn, Rm	Signed Multiply (halfwords)	
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply (32 x 32), 64-bit result	
SMULWB, SMULWT	{Rd,} Rn, Rm	Signed Multiply word by halfword	
SMUSD, SMUSDX	{Rd,} Rn, Rm	Signed dual Multiply Subtract	
SSAT	Rd, #n, Rm {,shift #s}	Signed Saturate	Q
SSAT16	Rd, #n, Rm	Signed Saturate 16	Q
SSAX	{Rd,} Rn, Rm	Signed Subtract and Add with Exchange	GE
SSUB16	{Rd,} Rn, Rm	Signed Subtract 16	
SSUB8	{Rd,} Rn, Rm	Signed Subtract 8	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
STM	Rn{!}, reglist	Store Multiple registers, increment after	
STMDB, STMEA	Rn{!}, reglist	Store Multiple registers, decrement before	
STMFD, STMIA	Rn{!}, reglist	Store Multiple registers, increment after	
STR	Rt, [Rn, #offset]	Store Register word	
STRB, STRBT	Rt, [Rn, #offset]	Store Register byte	
STRD	Rt, Rt2, [Rn, #offset]	Store Register two words	
STREX	Rd, Rt, [Rn, #offset]	Store Register Exclusive	
STREXB	Rd, Rt, [Rn]	Store Register Exclusive Byte	
STREXH	Rd, Rt, [Rn]	Store Register Exclusive Halfword	
STRH, STRHT	Rt, [Rn, #offset]	Store Register Halfword	
STRT	Rt, [Rn, #offset]	Store Register word	
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	N,Z,C,V
SVC	#imm	Supervisor Call	
SXTAB	{Rd,} Rn, Rm,{,ROR #}	Extend 8 bits to 32 and add	
SXTAB16	{Rd,} Rn, Rm,{,ROR #}	Dual extend 8 bits to 16 and add	
SXTAH	{Rd,} Rn, Rm,{,ROR #}	Extend 16 bits to 32 and add	
SXTB16	{Rd,} Rm {,ROR #n}	Signed Extend Byte 16	
SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a halfword	
TBB	[Rn, Rm]	Table Branch Byte	
TBH	[Rn, Rm, LSL #1]	Table Branch Halfword	
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
UADD16	{Rd,} Rn, Rm	Unsigned Add 16	GE
UADD8	{Rd,} Rn, Rm	Unsigned Add 8	GE
USAX	{Rd,} Rn, Rm	Unsigned Subtract and Add with Exchange	GE
UHADD16	{Rd,} Rn, Rm	Unsigned Halving Add 16	
UHADD8	{Rd,} Rn, Rm	Unsigned Halving Add 8	
UHASX	{Rd,} Rn, Rm	Unsigned Halving Add and Subtract with Exchange	
UHSAX	{Rd,} Rn, Rm	Unsigned Halving Subtract and Add with Exchange	
UHSUB16	{Rd,} Rn, Rm	Unsigned Halving Subtract 16	
UHSUB8	{Rd,} Rn, Rm	Unsigned Halving Subtract 8	
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract	
UDIV	{Rd,} Rn, Rm	Unsigned Divide	
UMAAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Accumulate Accumulate Long (32 x 32 + 32 +32), 64-bit result	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply with Accumulate ($32 \times 32 + 64$), 64-bit result	
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply (32×32), 64-bit result	
UQADD16	{Rd,} Rn, Rm	Unsigned Saturating Add 16	
UQADD8	{Rd,} Rn, Rm	Unsigned Saturating Add 8	
UQASX	{Rd,} Rn, Rm	Unsigned Saturating Add and Subtract with Exchange	
UQSAX	{Rd,} Rn, Rm	Unsigned Saturating Subtract and Add with Exchange	
UQSUB16	{Rd,} Rn, Rm	Unsigned Saturating Subtract 16	
UQSUB8	{Rd,} Rn, Rm	Unsigned Saturating Subtract 8	
USAD8	{Rd,} Rn, Rm	Unsigned Sum of Absolute Differences	
USADA8	{Rd,} Rn, Rm, Ra	Unsigned Sum of Absolute Differences and Accumulate	
USAT	Rd, #n, Rm {,shift #s}	Unsigned Saturate	Q
USAT16	Rd, #n, Rm	Unsigned Saturate 16	Q

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
UASX	{Rd,} Rn, Rm	Unsigned Add and Subtract with Exchange	GE
USUB16	{Rd,} Rn, Rm	Unsigned Subtract 16	GE
USUB8	{Rd,} Rn, Rm	Unsigned Subtract 8	GE
UXTAB	{Rd,} Rn, Rm,{,ROR #}	Rotate, extend 8 bits to 32 and Add	
UXTAB16	{Rd,} Rn, Rm,{,ROR #}	Rotate, dual extend 8 bits to 16 and Add	
UXTAH	{Rd,} Rn, Rm,{,ROR #}	Rotate, unsigned extend and Add Halfword	
UXTB	{Rd,} Rm {,ROR #n}	Zero extend a Byte	
UXTB16	{Rd,} Rm {,ROR #n}	Unsigned Extend Byte 16	
UXTH	{Rd,} Rm {,ROR #n}	Zero extend a Halfword	
VABS.F32	Sd, Sm	Floating-point Absolute	
VADD.F32	{Sd,} Sn, Sm	Floating-point Add	
VCMP.F32	Sd, <Sm #0.0>	Compare two floating-point registers, or one floating-point register and zero	FPSCR

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
VCMPE.F32	Sd, <Sm #0.0>	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	FPSCR
VCVT.S32.F32	Sd, Sm	Convert between floating-point and integer	
VCVT.S16.F32	Sd, Sd, #fbits	Convert between floating-point and fixed point	
VCVTR.S32.F32	Sd, Sm	Convert between floating-point and integer with rounding	
VCVT<B H>.F32.F16	Sd, Sm	Converts half-precision value to single-precision	
VCVTT<B T>.F32.F16	Sd, Sm	Converts single-precision register to half-precision	
VDIV.F32	{Sd,} Sn, Sm	Floating-point Divide	
VFMA.F32	{Sd,} Sn, Sm	Floating-point Fused Multiply Accumulate	
VFNMA.F32	{Sd,} Sn, Sm	Floating-point Fused Negate Multiply Accumulate	
VFMS.F32	{Sd,} Sn, Sm	Floating-point Fused Multiply Subtract	
VFNMS.F32	{Sd,} Sn, Sm	Floating-point Fused Negate Multiply Subtract	
VLDM.F<32 64>	Rn{!}, list	Load Multiple extension registers	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
VLDR.F<32 64>	<Dd Sd>, [Rn]	Load an extension register from memory	
VLMA.F32	{Sd,} Sn, Sm	Floating-point Multiply Accumulate	
VLMS.F32	{Sd,} Sn, Sm	Floating-point Multiply Subtract	
VMOV.F32	Sd, #imm	Floating-point Move immediate	
VMOV	Sd, Sm	Floating-point Move register	
VMOV	Sn, Rt	Copy Arm core register to single precision	
VMOV	Sm, Sm1, Rt, Rt2	Copy 2 Arm core registers to 2 single precision	
VMOV	Dd[x], Rt	Copy Arm core register to scalar	
VMOV	Rt, Dn[x]	Copy scalar to Arm core register	
VMRS	Rt, FPSCR	Move FPSCR to Arm core register or APSR	N,Z,C,V
VMSR	FPSCR, Rt	Move to FPSCR from Arm Core register	FPSCR
VMUL.F32	{Sd,} Sn, Sm	Floating-point Multiply	

Cortex-M4 instruction set

Mnemonic	Operands	Brief description	Flags
VNEG.F32	Sd, Sm	Floating-point Negate	
VNMLA.F32	Sd, Sn, Sm	Floating-point Multiply and Add	
VNMLS.F32	Sd, Sn, Sm	Floating-point Multiply and Subtract	
VNMUL	{Sd,} Sn, Sm	Floating-point Multiply	
VPOP	list	Pop extension registers	
VPUSH	list	Push extension registers	
VSQRT.F32	Sd, Sm	Calculates floating-point Square Root	
VSTM	Rn{!}, list	Floating-point register Store Multiple	
VSTR.F<32 64>	Sd, [Rn]	Stores an extension register to memory	
VSUB.F<32 64>	{Sd,} Sn, Sm	Floating-point Subtract	
WFE		Wait For Event	
WFI		Wait For Interrupt	

Note: full explanation of each instruction can be found in Cortex-M4 Devices' Generic User Guide (Ref-4)

Cortex-M4 instruction set

- Cortex-M4 suffix
 - Some instructions can be followed by suffixes to update processor flags or execute the instruction on a certain condition

Suffix	Description	Example	Example explanation
S	Update APSR (flags)	ADDS R1, #0x21	Add 0x21 to R1 and update APSR
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Condition execution e.g. EQ= equal, NE= not equal, LT= less than	BNE label	Branch to the label if not equal

Data insertion and alignment

- Insert data inside programs
 - DCD: insert a word-size data
 - DCB: insert a byte-size data
 - ALIGN:
 - used before inserting a word-size data
 - Uses a number to determine the alignment size
- For example:

```
...
ALIGN        4          ; Align to a word boundary
MY_DATA      DCD      0x12345678 ; Insert a word-size data
MY_STRINGDCB "Hello",     0       ; Null terminated string
...
...
```

Useful resources

- Architecture Reference Manual:
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403c/index.html>
- Cortex-M4 Technical Reference Manual:
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439d/DDI0439D_cortex_m4_processor_rOp1_trm.pdf
- Cortex-M4 Devices Generic User Guide:
http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf



[†]The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks



C as Implemented in Assembly Language

Learning Objectives

At the end of this lecture, you should be able to:

- Outline the compiler build stages.
- Identify the role of each register according to the AAPCS Core Register Use standard.
- Identify the type of memory (read-only or read/write) suitable for a given type of information.
- Outline the variable class qualifiers.
- Outline the function of the linker.
- Identify how pointers are used in a C program.
- Identify and define simple 1-dimension and 2-dimension arrays.
- Explain the terms and functions of prolog and epilog.

Module Syllabus

- We program in C for convenience
- There are no MCUs which execute C, only machine code
- So we compile the C to assembly code, a human-readable representation of machine code
- We need to know what the assembly code implementing the C looks like
 - To use the processor efficiently
 - To analyze the code with precision
 - To find performance and other problems
- An overview of what C gets compiled to
 - C start-up module, subroutines calls, stacks, data classes and layout, pointers, control flow, etc.

C Programmer's World: a comprehensive set of features

- As many functions and variables as you want!
- All the memory you could ask for!
- So many data types! Integers, floating point, etc.
- So many data structures! Arrays, lists, trees, sets, dictionaries
- So many control structures! Subroutines, if/then/else, loops, etc.
- Iterators! Polymorphism!

Processor's World

- Data types
 - Integers
 - More if you're lucky!
- Instructions
 - Math: +, -, *, /
 - Logic: AND, OR
 - Shift, rotate
 - Move, swap
 - Compare
 - Jump, branch

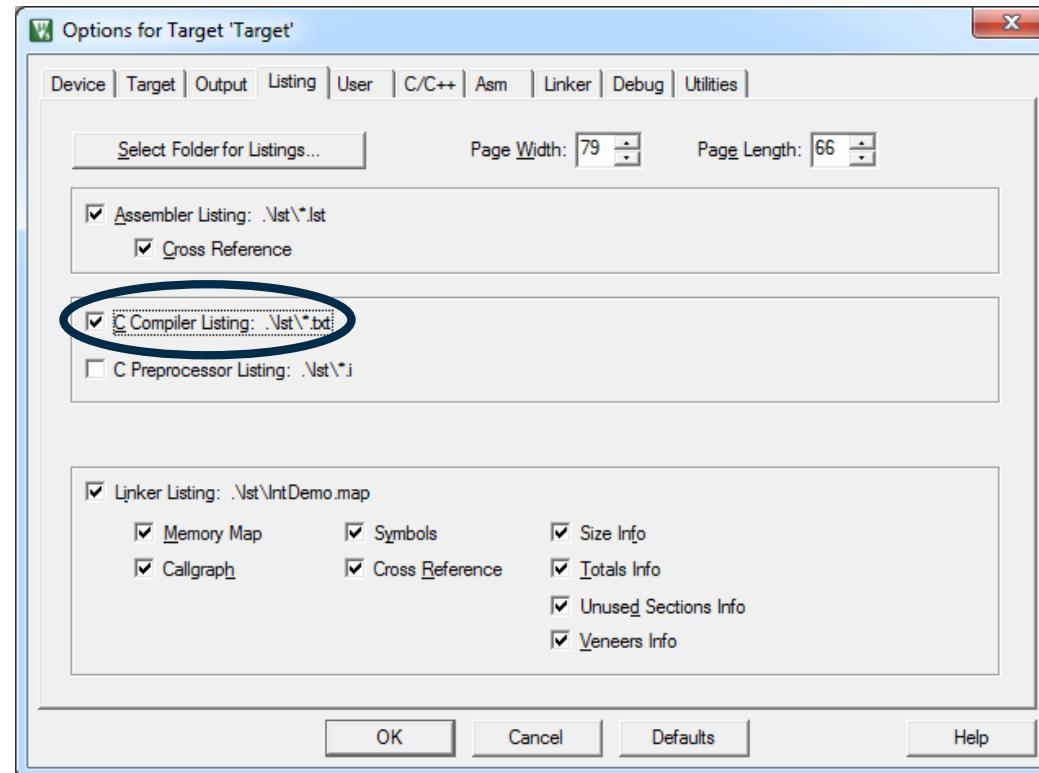
23	251	151	11	3	1	1	1
213	6	234	2	u	1	1	1
2	33	72	1	a	1	1	a
a	4	h	e	l	l	o	1
67	96	a	0	9	9	9	1
6	11	d	72	7	0	0	0
28	289	37	54	42	0	0	0
213	6	234	2	31	1	1	1

Compiler Stages

- Parser
 - reads in C code
 - checks for syntax errors
 - forms intermediate code (tree representation)
- High-Level Optimizer
 - Modifies intermediate code (processor-independent)
- Code Generator
 - Creates assembly code step by step from each node of the intermediate code
 - Allocates variable uses to registers
- Low-Level Optimizer
 - Modifies assembly code (parts are processor-specific)
- Assembler
 - Creates object code (machine code)
- Linker/Loader
 - Creates executable image from object file

Examining Assembly Code before Debugger

- Compiler can generate assembly code listing for reference
- Select in project options



Examining Disassembled Program in Debugger

The screenshot shows a debugger interface with two panes. The left pane displays the C source code for a main function, and the right pane shows the corresponding ARM assembly code.

C Source Code (Left):

```
1 #include "project.h"
2 #include "data.h"
3
4 int main(void)
5 {
6     arrays(2, 4);
7     fun4(1,2000,3);
8     static_auto_local();
9
10    while (1)
11    ;
12 }
13
14 // *****
15 // ***** ARM Univ
```

Assembly Code (Right):

Address	OpCode	Instruction	Description
0x00000208	D2F9	BCS 0x000001FE	
0x0000020A	BD70	POP {r4-r6,pc}	
6:		arrays(2, 4);	
0x0000020C	2104	MOVS r1,#0x04	
0x0000020E	2002	MOVS r0,#0x02	
0x00000210	F000F858	BL.W arrays (0x000002C4)	
7:		fun4(1,2000,3);	
0x00000214	2203	MOVS r2,#0x03	
0x00000216	217D	MOVS r1,#0x7D	
0x00000218	0109	LSLS r1,r1,#4	
0x0000021A	2001	MOVS r0,#0x01	
0x0000021C	F000F88D	BL.W fun4 (0x0000033A)	
8:		static_auto_local();	
9:			
0x00000220	F000F802	BL.W static_auto_local (0x00000228)	
10:		while (1)	
0x00000224	BF00	NOP	
0x00000226	E7FE	B 0x00000226	

A Word on Code Optimizations

- Compiler and rest of toolchain try to optimize code:
 - Simplifying operations
 - Removing “dead” code
 - Using registers
- These optimizations often get in way of understanding what the code does
 - Fundamental trade-off: Fast or comprehensible code?
 - Compiler optimization levels: Level 0 to Level 3
- Code examples here may use “volatile” data type modifier to reduce compiler optimizations and improve readability

Application Binary Interface

- Defines rules which allow separately developed functions to work together
- Arm Architecture Procedure Call Standard (AAPCS)
 - Which registers must be saved and restored
 - How to call procedures
 - How to return from procedures
- C Library ABI (CLIBABI)
 - C Library functions
- Runtime ABI (RTABI)
 - Runtime helper functions: 32/32 integer division, memory copying, floating-point operations, data type conversions, etc.



Using Registers

AAPCS Register Use Conventions

- Make it easier to create modular, isolated and integrated code
- Scratch registers are not expected to be preserved upon returning from a called subroutine
 - r0-r3
- Preserved (“variable”) registers are expected to have their original values upon returning from a called subroutine
 - r4-r8, r10-r11

AAPCS Core Register Use

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6,SB,TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Must be saved, restored by callee-procedure, it may modify them.

Must be saved, restored by callee-procedure, it may modify them.
Calling subroutine expects these to retain their value.

Don't need to be saved. May be used for arguments, results, or temporary values.



Memory requirements

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
    int e;
    char f[32];
    e = d + 7;
    a = e + 29999;
    strcpy(f,"Hello!");
}
```

- Eight possible types of ‘information’
 - Code
 - Read-only static data
 - Writable static data
 - Initialized
 - Zero-initialized
 - Uninitialized
 - Heap
 - Stack
- What goes where?
 - Code is obvious
 - And the others?

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
    int e;
    char f[32];
    e = d + 7;
    a = e + 29999;
    strcpy(f,"Hello!");
}
```

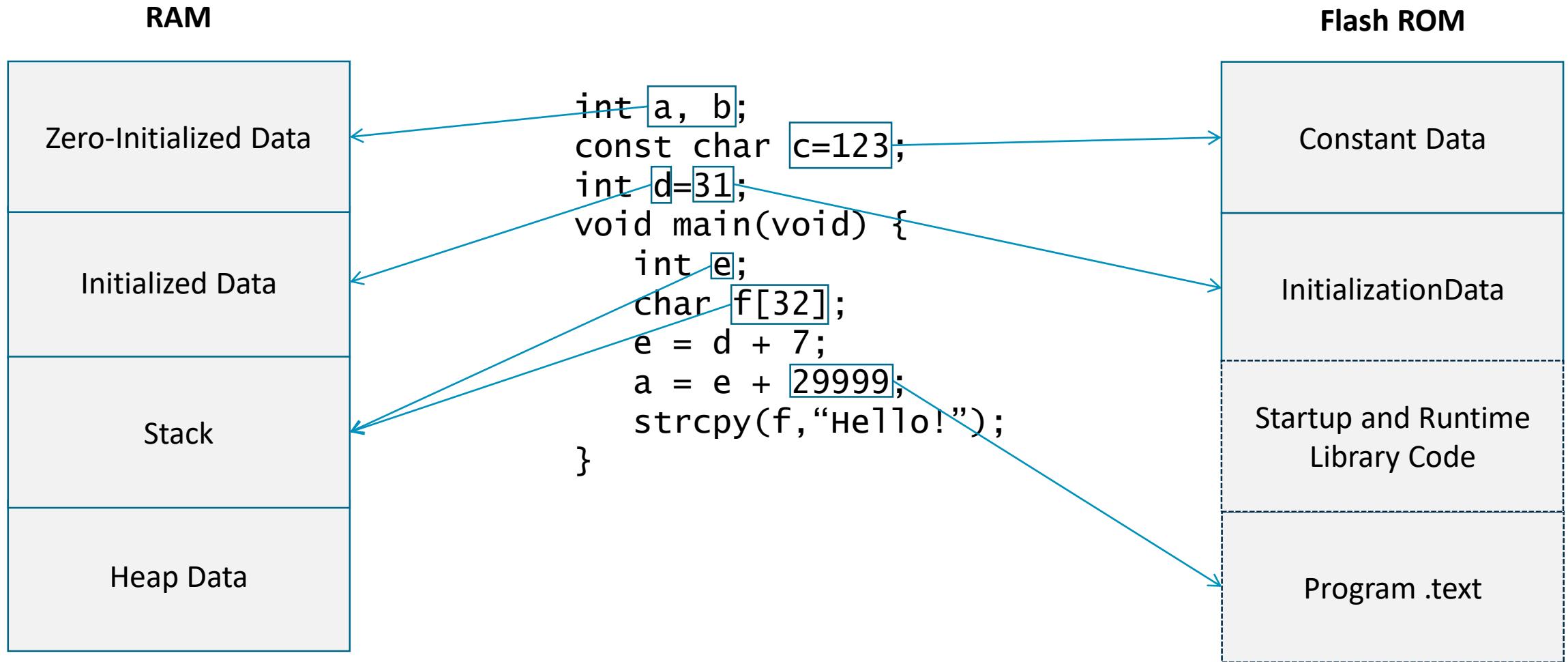
- Can the information change?
 - No? Put it in read-only, nonvolatile memory
 - Instructions
 - Constant strings
 - Constant operands
 - Initialization values
 - Yes? Put it in read/write memory
 - Variables
 - Intermediate computations
 - Return address
 - Other housekeeping data

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
    int e;
    char f[32];
    e = d + 7;
    a = e + 29999;
    strcpy(f,"Hello!");
}
```

- How long does the data need to exist? Reuse memory if possible.
 - Statically allocated
 - Exists from program start to end
 - Each variable has its own fixed location
 - Space is not reused
 - Automatically allocated
 - Exists from function start to end
 - Space can be reused
 - Dynamically allocated
 - Exists from explicit allocation to explicit de-allocation
 - Space can be reused

Program Memory Use



Activation Record

- Activation records are located on the stack
 - Calling a function creates an activation record
 - Returning from a function deletes the activation record
- Automatic variables and housekeeping information are stored in a function's activation record
- Not all fields (LS, RA, Arg) may be present for each activation record

Lower address

	(Free stack space)
Activation record for current function	Local storage Return address Arguments
Activation record for caller function	Local storage Return address Arguments
Activation record for caller's caller function	Local storage Return address Arguments
Activation record for caller's caller's caller function	Local storage Return address Arguments

<- Stack ptr

Higher address

Type and Class Qualifiers

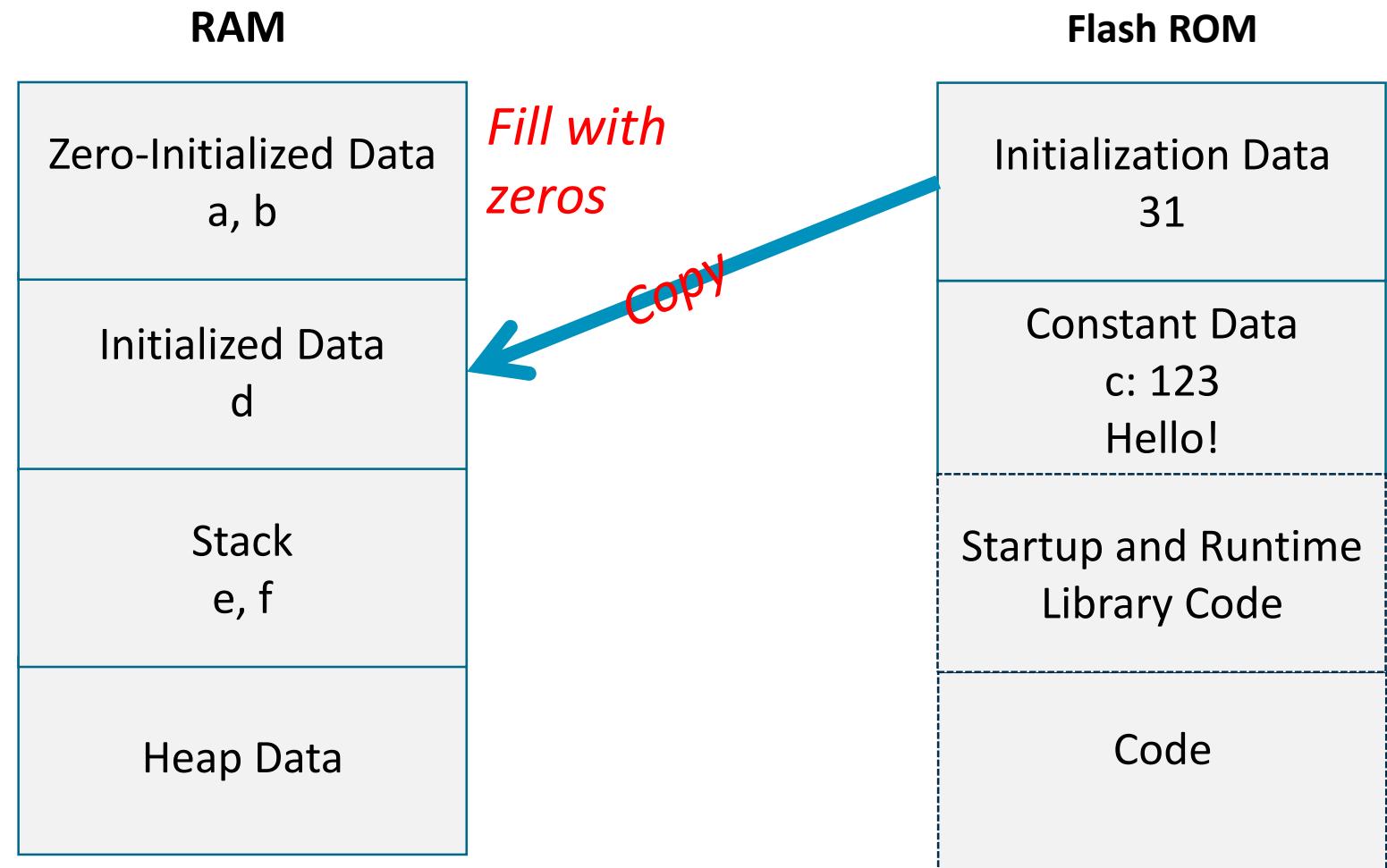
- Const
 - Never written by program, can be put in ROM to save RAM
- Volatile
 - Can be changed outside of normal program flow: ISR, hardware register
 - Compiler must be careful with optimizations
- Static
 - Declared within function, retains value between function invocations
 - Scope is limited to function

Linker Map File

- Contains extensive information on functions and variables
 - Value, type, size, object
- Cross references between sections
- Memory map of image
- Sizes of image components
- Summary of memory requirements

C Run-Time Start-Up Module

- After reset, MCU must:
- Initialize hardware
 - Peripherals, etc.
 - Set up stack pointer
- Initialize C or C++ runtime environment
 - Set up heap memory
 - Initialize variables





Accessing data in Memory

Accessing Data

- What does it take to get at a variable in memory?
 - Depends on location, which depends on storage type (static, automatic, dynamic)

```
int siA;
void static_auto_local() {
    int aiB;
    static int siC=3;
    int * apD;
    int aiE=4, aiF=5, aiG=6;

    siA = 2;
    aiB = siC + siA;
    apD = & aiB;
    (*apD)++;
    apD = &siC;
    (*apD) += 9;
    apD = &siA;
    apD = &aiE;
    apD = &aiF;
    apD = &aiG;
    (*apD)++;
    aiE+=7;
    *apD = aiE + aiF;
}
```

Static Variables

- Static var can be located anywhere in 32-bit memory space, so need a 32-bit pointer
- Can't fit a 32-bit pointer into a 16-bit instruction (or a 32-bit instruction), so save the pointer separate from instruction, but nearby so we can access it with a short PC-relative offset
- Load the pointer into a register (r0)
- Can now load variable's value into a register (r1) from memory using that pointer in r0
- Similarly can store a new value to the variable in memory

Load r0 with pointer to variable
Load r1 from [r0]
Use value of variable

Label:
32-bit pointer to variable

variable

Static Variables

- Key
 - variable's value
 - **variable's address**
 - address of copy of variable's address
- Addresses of siA and siC are stored as literals to be loaded into pointers
- Variables siC and siA are located in .data section with initial values

```
AREA ||.text||, CODE, READONLY, ALIGN=2
;;;20          siA = 2;
00000e 2102 MOVS    r1,#2      ; r1 = 2
000010 4a37 LDR     r2,|L1.240| ; r2 = &siA
000012 6011 STR     r1,[r2,#0] ; *r2 = r1
;;;21          aiB = siC + siA;
000014 4937 LDR     r1,|L1.244| ; r1 = &siC
000016 6809 LDR     r1,[r1,#0] ; r1 = *r1
000018 6812 LDR     r2,[r2,#0] ; r2 = *r2
00001a 1889 ADDS   r1,r1,r2  ; r1 = r1 + r2
...
|L1.240|
DCD    ||siA||
|L1.244|
DCD    ||sic||
AREA ||.data||, DATA, ALIGN=2
||sic||
DCD    0x00000003
||siA||
DCD    0x00000000
```

Automatic Variables Stored on Stack

- Automatic variables are stored in a function's activation record (unless optimized and promoted to register)
- Activation records are located on the stack
- Calling a function creates an activation record, allocating space on stack
- Returning from a function deletes the activation record, freeing up space on stack
- Variables in C are implicitly automatic; there is no need to specify the keyword

```
int main(void) {  
    auto vars;  
    a();  
}  
  
void a(void) {  
    auto vars;  
    b();  
}  
  
void b(void) {  
    auto vars;  
    c();  
}  
  
void c(void) {  
    auto vars;  
    ...  
}
```

Automatic Variables

```
int main(void) {  
    auto vars;  
    a();  
}  
  
void a(void) {  
    auto vars;  
    b();  
}  
  
void b(void) {  
    auto vars;  
    c();  
}  
  
void c(void) {  
    auto vars;  
    ...  
}
```

Lower address

Higher address

	(Free stack space)
Activation record for current function C	Local storage Saved regs Arguments (optional)
Activation record for caller function B	Local storage Saved regs Arguments (optional)
Activation record for caller's caller function A	Local storage Saved regs Arguments (optional)
Activation record for caller's caller's caller function main	Local storage Saved regs Arguments (optional)

<- Stack pointer while executing C

<- Stack pointer while executing B

<- Stack pointer while executing A

<- Stack pointer while executing main

Addressing Automatic Variables

- Program must allocate space on stack for variables
- Stack addressing uses an offset from the stack pointer: [sp, #offset]
- Items on the stack are word aligned
 - In instructions, one byte used for offset, which is multiplied by four
 - Possible offsets: 0, 4, 8, ..., 1020
 - Maximum range addressable this way is 1024 bytes

Address	Contents
SP	
SP+0x4	
SP+0x8	
SP+0xC	
SP+0x10	
SP+0x14	
SP+0x18	
SP+0x1C	
SP+0x20	

Automatic Variables

Address	Contents
SP	aiG
SP+4	aiF
SP+8	aiE
SP+0xC	aiB
SP+0x10	r0
SP+0x14	r1
SP+0x18	r2
SP+0x1C	r3
SP+0x20	lr

- Initialize aiE
 - Initialize aiF
 - Initialize aiG
 - Store value for aiB
-
- ```
;;;14 void static_auto_local(void) {
000000 b50f PUSH {r0-r3,lr}
;;;15 int aiB;
;;;16 static int siC=3;
;;;17 int * apD;
;;;18 int aiE=4, aiF=5, aiG=6;
000002 2104 MOVS r1,#4
000004 9102 STR r1,[sp,#8]
000006 2105 MOVS r1,#5
000008 9101 STR r1,[sp,#4]
00000a 2106 MOVS r1,#6
00000c 9100 STR r1,[sp,#0]
...
;;;21 aiB = siC + siA;
...
00001c 9103 STR r1,[sp,#0xc]
```



# Using Pointers

# Using Pointers to Automatics

- C Pointer: a variable which holds the data's address
- aiB is on stack at SP+0xc
- Compute r0 with **variable's address** from **stack pointer and offset (0xc)**
- Load r1 with variable's value from memory
- Operate on r1, save back to **variable's address**

```
; ; ;22 apD = & aiB;
00001e a803 ADD r0, sp, #0xc
; ; ;23 (*apD)++;
000020 6801 LDR r1, [r0, #0]
000022 1c49 ADDS r1, r1, #1
000024 6001 STR r1, [r0, #0]
```

# Using Pointers to Statics

- Load r0 with variable's address from address of copy of variable's address
- Load r1 with variable's value from memory
- Operate on r1, save back to variable's address

```
; ; ;24 apD = &sic;
000026 4833 LDR r0, |L1.244|
; ; ;25 (*apD) += 9;
000028 6801 LDR r1, [r0,#0]
00002a 3109 ADDS r1, r1, #9
00002c 6001 STR r1, [r0,#0]
|L1.244| DCD ||sic||
||sic|| AREA || .data ||, DATA, ALIGN=2
 DCD 0x00000003
```



## Array Access

# Array Access

- What does it take to get at an array element in memory?
  - Depends on how many dimensions
  - Depends on element size and row width
  - Depends on location, which depends on storage type (static, automatic, dynamic)

```
uint8 buff2[3];
uint16 buff3[5][7];

uint32 arrays(uint8 n, uint8 j) {
 volatile uint32 i;
 i = buff2[0] + buff2[n];
 i += buff3[n][j];
 return i;
}
```

# Accessing 1-D Array Elements

- Need to calculate element address: sum of:
    - array start address
    - offset: index \* element size
  - buff2 is array of unsigned characters
- 
- Move n (argument) from r0 into r2
  - Load r3 with pointer to buff2
  - Load (byte) r3 with first element of buff2
  - Load r4 with pointer to buff2
  - Load (byte) r4 with element at address buff2+r2
    - r2 holds argument n
  - Add r3 and r4 to form sum

| Address   | Contents |
|-----------|----------|
| buff2     | buff2[0] |
| buff2 + 1 | buff2[1] |
| buff2 + 2 | buff2[2] |

```
00009e 4602 MOV r2,r0
;; ;76 i = buff2[0] + buff2[n];
0000a0 4b1b LDR r3,|L1.272|
0000a2 781b LDRB r3,[r3,#0] ; buff2
0000a4 4c1a LDR r4,|L1.272|
0000a6 5ca4 LDRB r4,[r4,r2]
0000a8 1918 ADDS r0,r3,r4
|L1.272|
DCD buff2
```

# Accessing 2-D Array Elements

`uint16 buff3[5][7]`

| Address  | Contents    |
|----------|-------------|
| buff3    | buff3[0][0] |
| buff3+1  |             |
| buff3+2  | buff3[0][1] |
| buff3+3  |             |
| (etc.)   |             |
| buff3+10 | buff3[0][5] |
| buff3+11 |             |
| buff3+12 | buff3[0][6] |
| buff3+13 |             |
| buff3+14 | buff3[1][0] |
| buff3+15 |             |
| buff3+16 | buff3[1][1] |
| buff3+17 |             |
| (etc.)   |             |
| buff3+68 | buff3[4][6] |
| buff3+69 |             |

- `var[rows][columns]`
- Sizes
  - Element: 2 bytes
  - Row:  $7 \times 2$  bytes = 14 bytes (0xe)
- Offset based on row index and column index
  - column offset = column index \* element size
  - row offset = row index \* row size

# Code to Access 2-D Array

| Instruction                       | r0            | r1 | r2 | r3                                                                              | r4     | Description                                                              |
|-----------------------------------|---------------|----|----|---------------------------------------------------------------------------------|--------|--------------------------------------------------------------------------|
| <code>;; i += buff3[n][j];</code> | i             | j  | n  | -                                                                               | -      |                                                                          |
| MOVS r3,#0xe                      | -             | -  | -  | 0xe                                                                             | -      | Load row size                                                            |
| MULS r3,r2,r3                     | -             | -  | n  | n*0xe                                                                           | -      | Multiply by row number                                                   |
| LDR<br>r4,  L1.276<br>            | -             | -  | -  | -                                                                               | &buff3 | Load address of buff3                                                    |
| ADDS r3,r3,r4                     | -             | -  | -  | &buff3+n*0xe                                                                    | -      | Add buff3 address to row offset                                          |
| LSLS r4,r1,#1                     | -             | j  | -  | -                                                                               | j<<1   | Multiply column number by 2 (buff3 is uint16 array)                      |
| LDRH r3,[r3,r4]                   | -             | -  | -  | $*(\text{uint16})(\&\text{buff3} + n * 0xe + j << 1)$<br>$= \text{buff3}[n][j]$ | j<<1   | Load halfword r3 with element at r3+r4 (buff3 + row offset + col offset) |
| ADDS r0,r3,r0                     | i+buff3[n][j] | -  | -  | buff3[n][j]                                                                     |        | Add r3 to r0 (i)                                                         |



# Function Prolog and Epilog

# Prolog and Epilog

- A function's Prolog and Epilog are responsible for creating and destroying its activation record
- Remember AAPCS
  - Scratch registers r0-r3 are not expected to be preserved upon returning from a called subroutine, can be overwritten
  - Preserved ("variable") registers r4-r8, r10-r11 must have their original values upon returning from a called subroutine
  - Prolog must save preserved registers on stack
  - Epilog must restore preserved registers from stack
- Prolog also may
  - Handle function arguments
  - Allocate temporary storage space on stack (subtract from SP)
- Epilog
  - May de-allocate stack space (add to SP)
  - Returns control to calling function

# Return Address

- Return address stored in LR by bl, blx instructions
- Consider case where a() calls b() which calls c()
  - On entry to b(), LR holds return address in a()
  - When b() calls c(), LR will be overwritten with return address in b()
  - After c() returns, b() will have lost its return address
- Does this function call a subroutine?
  - Yes: must save and restore LR on stack just like other preserved registers, but LR value is popped into PC rather than LR
  - No: don't need to save or restore LR, as it will not be modified

# Function Prolog and Epilog

- Save r4 (preserved register) and link register (return address)
- Allocate 32 (0x20) bytes on stack for array x by subtracting from SP
- Compute return value, placing in return register r0
- Deallocate 32 bytes from stack
- Pop r4 (preserved register) and PC (return address)

```
fun4 PROC
;; ;102 int fun4(char a, int b, char c)
{
;; ;103 volatile int x[8];
00010a b510 PUSH {r4,lr}
00010c b088 SUB sp,sp,#0x20
...
;; ;106 return a+b+c;
00011c 1858 ADDS r0,r3,r1
00011e 1880 ADDS r0,r0,r2
;; ;107
000120 }
b008 ADD sp,sp,#0x20
bd10 POP {r4,pc}
ENDP
```

# Activation Record Creation by Prolog

|                 |                |                      |                                                    |
|-----------------|----------------|----------------------|----------------------------------------------------|
| Smaller address | space for x[0] | Array x              | <- 3. SP after sub sp,sp,#0x20                     |
|                 | space for x[1] |                      |                                                    |
|                 | space for x[2] |                      |                                                    |
|                 | space for x[3] |                      |                                                    |
|                 | space for x[4] |                      |                                                    |
|                 | space for x[5] |                      |                                                    |
|                 | space for x[6] |                      |                                                    |
|                 | space for x[7] |                      |                                                    |
|                 | lr             | Return address       | <- 2. SP after push {r4,lr}                        |
| Larger address  | r4             | Preserved register   |                                                    |
|                 |                | Caller's stack frame | <- 1. SP on entry to function, before push {r4,lr} |

# Activation Record Destruction by Epilog

Smaller address

|                |                      |                                 |
|----------------|----------------------|---------------------------------|
| space for x[0] | Array x              | <- 1. SP before add sp,sp,#0x20 |
| space for x[1] |                      |                                 |
| space for x[2] |                      |                                 |
| space for x[3] |                      |                                 |
| space for x[4] |                      |                                 |
| space for x[5] |                      |                                 |
| space for x[6] |                      |                                 |
| space for x[7] |                      |                                 |
| lr             | Return address       | <- 2. SP after add sp,sp,#20    |
| r4             | Preserved register   |                                 |
|                | Caller's stack frame | <- 1. SP after pop {r4,pc}      |

Larger address



# Calling Functions

# AAPCS Core Register Use

| Register | Synonym | Special  | Role in the procedure call standard                                                  |
|----------|---------|----------|--------------------------------------------------------------------------------------|
| r15      |         | PC       | The Program Counter.                                                                 |
| r14      |         | LR       | The Link Register.                                                                   |
| r13      |         | SP       | The Stack Pointer.                                                                   |
| r12      |         | IP       | The Intra-Procedure-call scratch register.                                           |
| r11      | v8      |          | Variable-register 8.                                                                 |
| r10      | v7      |          | Variable-register 7.                                                                 |
| r9       |         | v6,SB,TR | Platform register. The meaning of this register is defined by the platform standard. |
| r8       | v5      |          | Variable-register 5.                                                                 |
| r7       | v4      |          | Variable register 4.                                                                 |
| r6       | v3      |          | Variable register 3.                                                                 |
| r5       | v2      |          | Variable register 2.                                                                 |
| r4       | v1      |          | Variable register 1.                                                                 |
| r3       | a4      |          | Argument / scratch register 4.                                                       |
| r2       | a3      |          | Argument / scratch register 3.                                                       |
| r1       | a2      |          | Argument / result / scratch register 2.                                              |
| r0       | a1      |          | Argument / result / scratch register 1.                                              |

# Function Arguments and Return Values

- First, pass the arguments
  - How to pass them?
    - Much faster to use registers than stack
    - But quantity of registers is limited
  - Basic rules
    - Process arguments in order they appear in source code
    - Round size up to be a multiple of 4 bytes
    - Copy arguments into core registers (r0-r3), aligning doubles to even registers
    - Copy remaining arguments onto stack, aligning doubles to even addresses
    - Specific rules in AAPCS, Section 5.5
- Second, call the function
  - Usually as subroutine with branch link (bl) or branch link and exchange instruction (blx)
  - Exceptions in AAPCS

# Return Values

- Callee passes Return Value in register(s) or stack
- Registers
- Stack
  - Caller function allocates space for return value, then passes pointer to space as an argument to callee
  - Callee stores result at location indicated by pointer

| Return value size  | Registers used for passing |                     |
|--------------------|----------------------------|---------------------|
|                    | Fundamental Data Type      | Composite Data Type |
| 1-4 bytes          | r0                         | r0                  |
| 8 bytes            | r0-r1                      | stack               |
| 16 bytes           | r0-r3                      | stack               |
| Indeterminate size | n/a                        | stack               |

## Call Example

```
int fun2(int arg2_1, int arg2_2) {
 int i;
 arg2_2 += fun3(arg2_1, 4, 5, 6);
 ...
}
```

- Argument 4 into r3
- Argument 3 into r2
- Argument 2 into r1
- Argument 0 into r0
- Call fun3 with BL instruction
- Result was returned in r0, so add to r4 (arg2\_2  
+= result)

```
fun2 PROC
;;:85 int fun2(int arg2_1, int
arg2_2) {
 ...
 0000e0 2306 MOVS r3,#6
 0000e2 2205 MOVS r2,#5
 0000e4 2104 MOVS r1,#4
 0000e6 4630 MOV r0,r6

 0000e8 f7fffffe BL fun3
 0000ec 1904 ADDS r4,r0,r4
```

## Call and Return Example

```
int fun3(int arg3_1, int arg3_2,
 int arg3_3, int arg3_4) {
 return arg3_1*arg3_2*
 arg3_3*arg3_4;
}
```

- Save r4 and Link Register on stack
- $r0 = arg3\_1 * arg3\_2$
- $r0 *= arg3\_3$
- $r0 *= arg3\_4$
- Restore r4 and return from subroutine
- Return value is in r0

```
fun3 PROC
;; ;81 int fun3(int arg3_1, int arg3_2,
int arg3_3, int arg3_4) {

0000ba b510 PUSH {r4,lr}

0000c0 4348 MULS r0,r1,r0
0000c2 4350 MULS r0,r2,r0
0000c4 4358 MULS r0,r3,r0

0000c6 bd10 POP {r4,pc}
```

arm

Control Flow

# Control Flow: Conditionals and Loops

- How does the compiler implement conditionals and loops?

```
if (x){
 y++;
} else {
 y--;
}
```

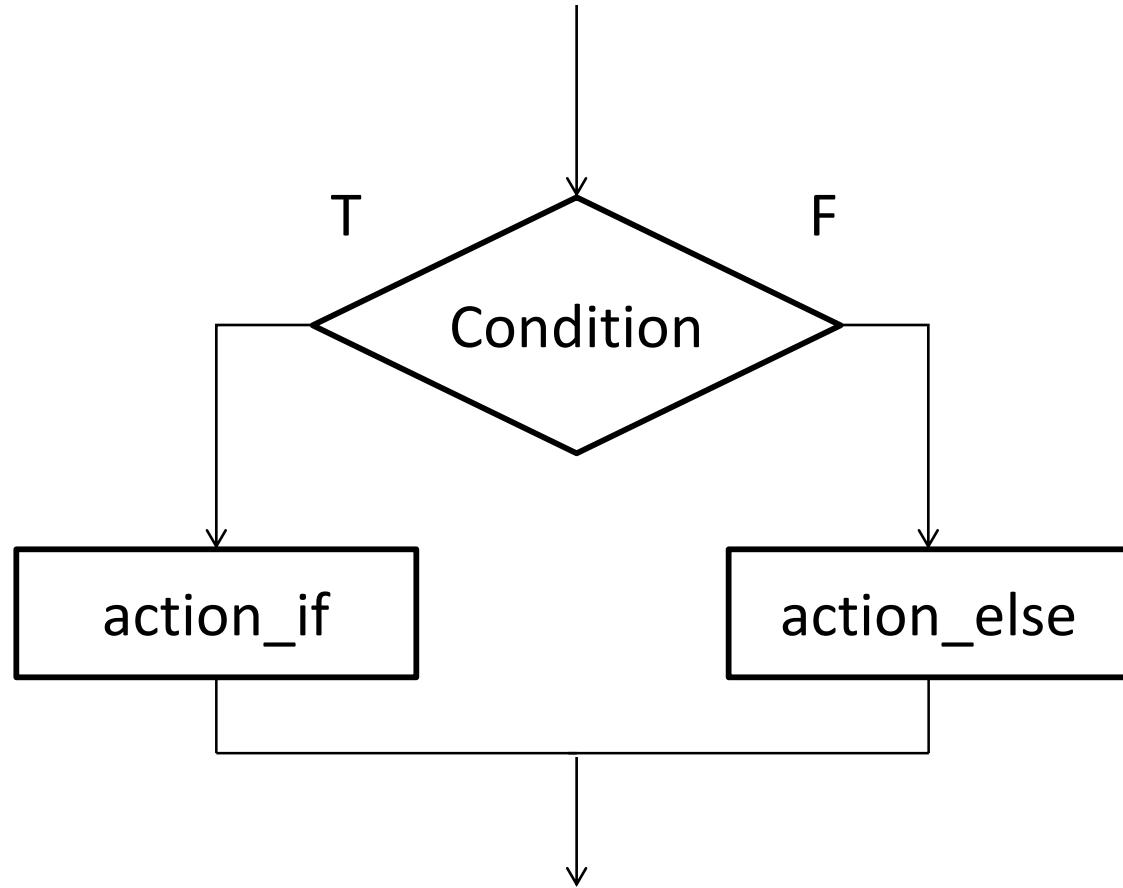
```
switch (x) {
 case 1:
 y += 3;
 break;
 case 31:
 y -= 5;
 break;
 default:
 y--;
 break;
}
```

```
for (i = 0; i < 10; i++){
 x += i;
}

while (x<10) {
 x = x + 1;
}

do {
 x += 2;
} while (x < 20);
```

# Control Flow: If/Else



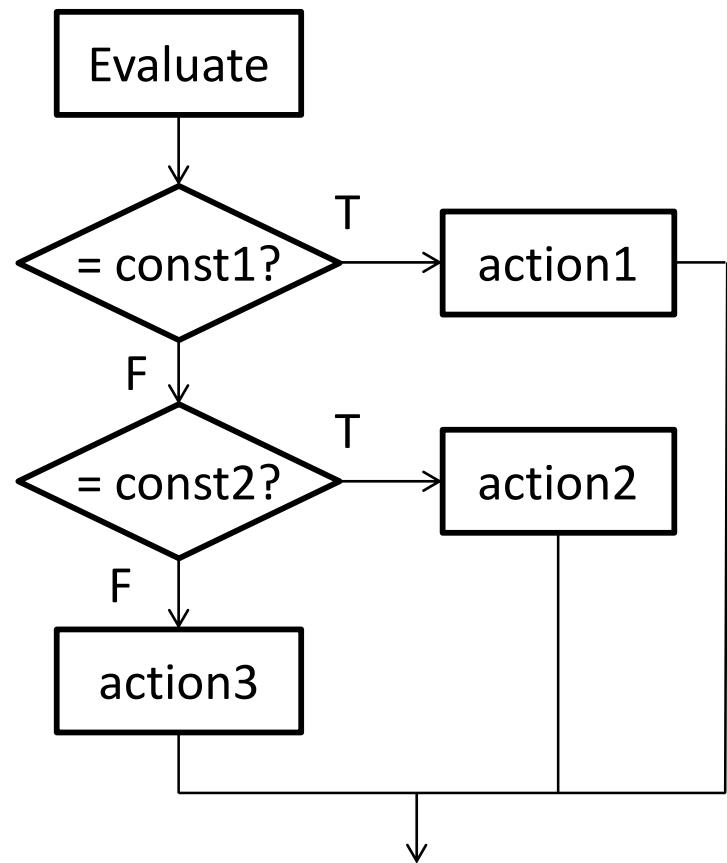
```
; ; ;39 if (x){
000056 2900 CMP r1,#0
000058 d001 BEQ |L1.94|

; ; ;40 y++;
00005a 1c52 ADDS r2,r2,#1
00005c e000 B |L1.96|

|L1.94|
; ; ;41 } else {
; ; ;42 y--;
00005e 1e52 SUBS r2,r2,#1

|L1.96|
; ; ;43 }
```

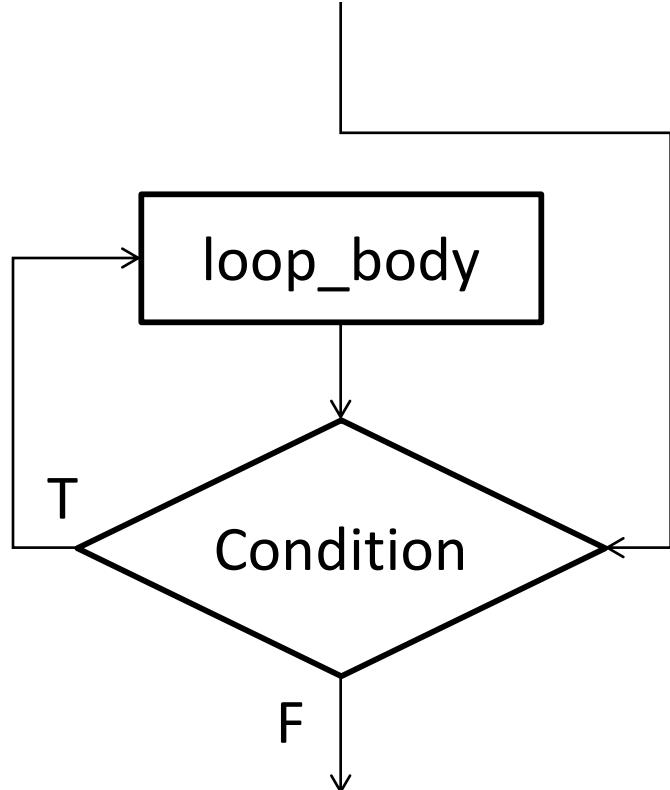
# Control Flow: Switch



```
; ; ; 45 switch (x) {
000060 2901 CMP r1,#1
000062 d002 BEQ |L1.106|
000064 291f CMP r1,#0x1f
```

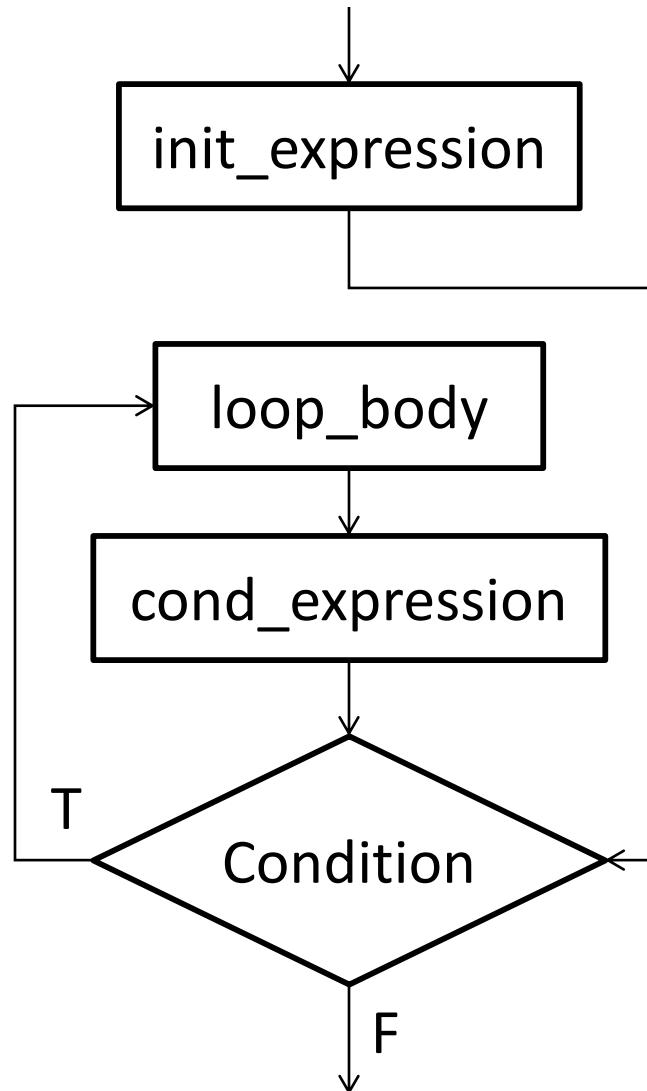
```
000066 d104 BNE |L1.114|
000068 e001 B |L1.110|
 |L1.106|
; ; ; 46 case 1:
; ; ; 47 y += 3;
00006a 1cd2 ADDS r2,r2,#3
; ; ; 48 break;
00006c e003 B |L1.118|
|L1.110|
; ; ; 49 case 31:
; ; ; 50 y -= 5;
00006e 1f52 SUBS r2,r2,#5
; ; ; 51 break;
000070 e001 B |L1.114|
|L1.114|
; ; ; 52 default:
; ; ; 53 y--;
000072 1e52 SUBS r2,r2,#1
; ; ; 54 break;
000074 bf00 NOP
|L1.118|
000076 bf00 NOP
; ; ; 55 }
```

# Iteration: While



```
; ; ; 57 while (x<10) {
000078 e000 B |L1.124|
 |L1.122|
; ; ; 58 x = x + 1;
00007a 1c49 ADDS r1,r1,#1
 |L1.124|
00007c 290a CMP r1,#0xa
; 57
00007e d3fc BCC |L1.122|
; ; ; 59 }
```

## Iteration: For

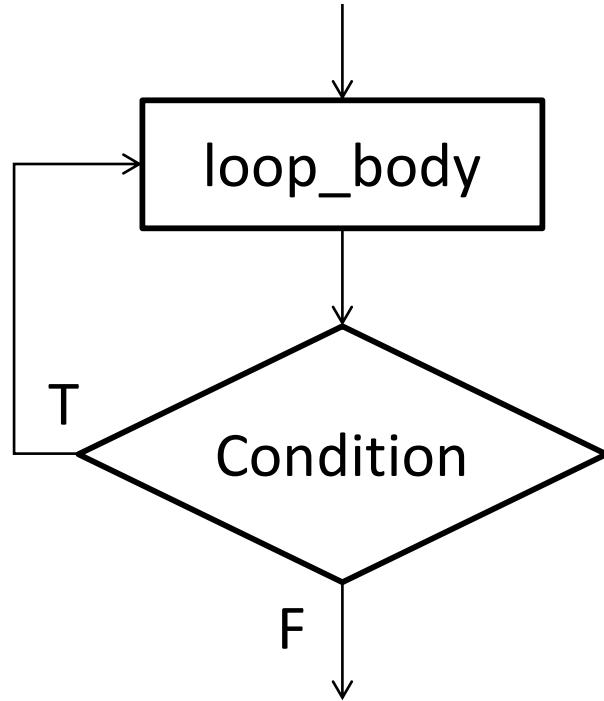


```
; ; ;61 for (i = 0; i < 10; i++){
000080 2300 MOVS r3,#0
000082 e001 B |L1.136|

; ; ;62 |L1.132|
000084 x += i;
000086 18c9 ADDS r1,r1,r3
;61 1c5b ADDS r3,r3,#1

000088 |L1.136|
;61 2b0a CMP r3,#0xa
00008a d3fb BCC |L1.132|
; ; ;63 }
```

# Iteration: Do/While



```
; ; ;65 do {
00008c bf00 NOP

| L1.142 |
; ; ;66 x += 2;
00008e 1c89 ADDS r1,r1,#2
; ; ;67 } while (x < 20);
000090 2914 CMP r1,#0x14
000092 d3fc BCC |L1.142|
```



<sup>†</sup>The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

[www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks)



# Interrupts and Low-Power Features

# Learning Objectives

At the end of this lecture, you should be able to:

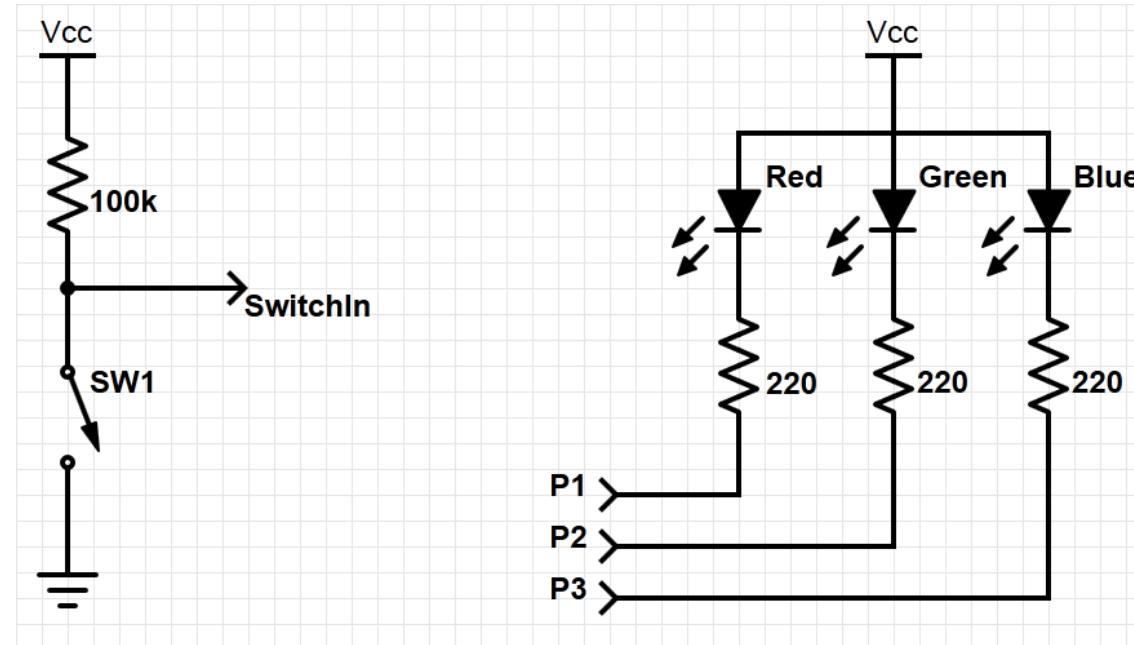
- Outline the different types of interrupts and their function.
- Describe the steps taken by the processor in handling interrupts and exceptions.
- Outline the differences between interrupts and exceptions.
- Describe the functions of the Nested Vectored Interrupt Controller (NVIC).

# Module syllabus

- Interrupts
  - What are interrupts?
  - Why use interrupts?
- Interrupts
  - Entering an exception handler
  - Exiting an exception handler
- Microcontroller interrupts
- Timing analysis
- Program design with interrupts
- Sharing data safely between ISRS and other threads

# Example system with interrupt

- Goal: to change the color of the RGB LED when the switch is pressed
- Need to add an external switch
  - (resistor is internal – a pull-up resistor)

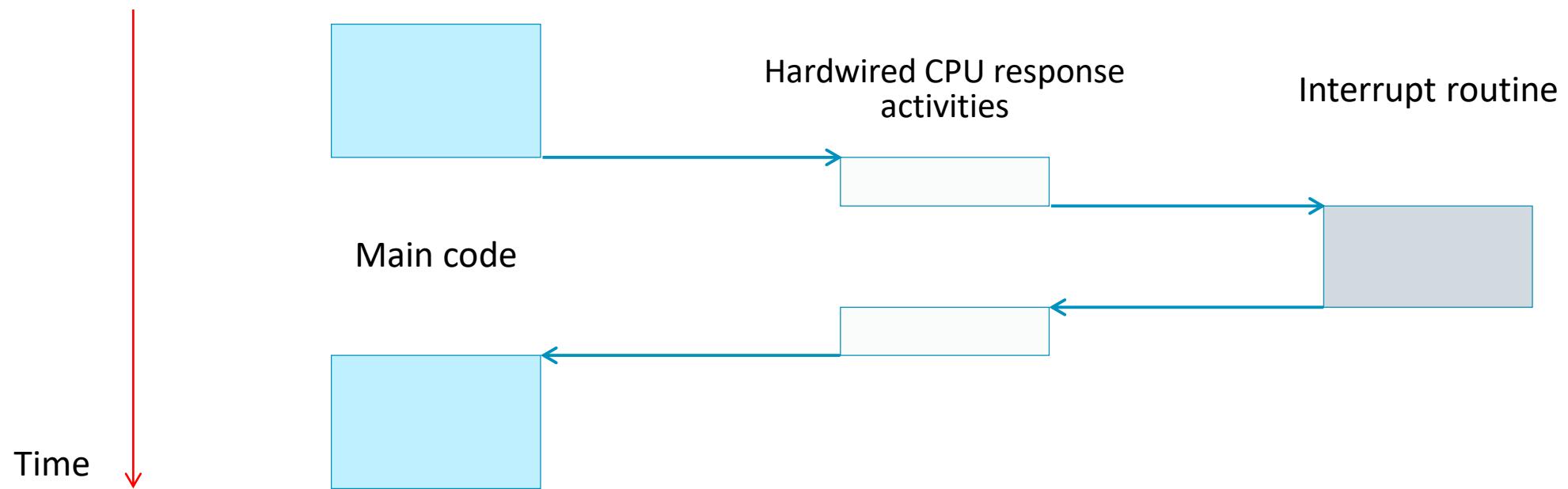


# How can we detect when a switch is pressed?

- One option is polling; e.g. using software to check it regularly. However, polling is:
  - Slow – user needs to check to see if switch is pressed
  - Wasteful of CPU time – the faster the response needed, the more often user needs to check
  - Not scalable – It's difficult to build a multi-activity system that can respond quickly. The system's response time depends on all other processing it has to do.
- A better option is an interrupt; e.g., using special hardware in the MCU to detect and run ISR in response.  
An interrupt is:
  - Efficient – the code runs only when necessary
  - Fast – it's a hardware mechanism
  - Scalable:
    - ISR response time doesn't depend on most other processing
    - Code modules can be developed independently

# Interrupt or Exception Processing Sequence

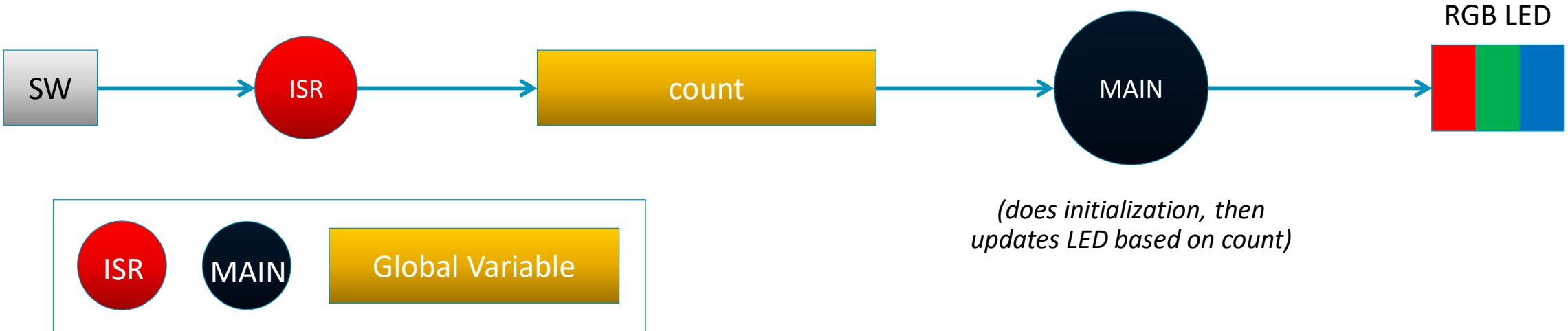
- Main code is running.
- When the interrupt trigger occurs:
  - The processor does some hardwired processing
  - The processor executes the ISR, including return-from-interrupt instruction at the end
- Then the processor resumes running main code



# Interrupts

- Hardware-triggered asynchronous software routine
  - Triggered by hardware signal from peripheral or external device
  - Asynchronous – can happen anywhere in the program (unless interrupt is disabled)
  - Software routine – Interrupt Service Routine runs in response to interrupt
- Fundamental mechanism of microcontrollers
  - Provides efficient event-based processing rather than polling
  - Provides quick response to events regardless of program state, complexity, location
  - Allows many multithreaded embedded systems to be responsive without an operating system (specifically task scheduler)

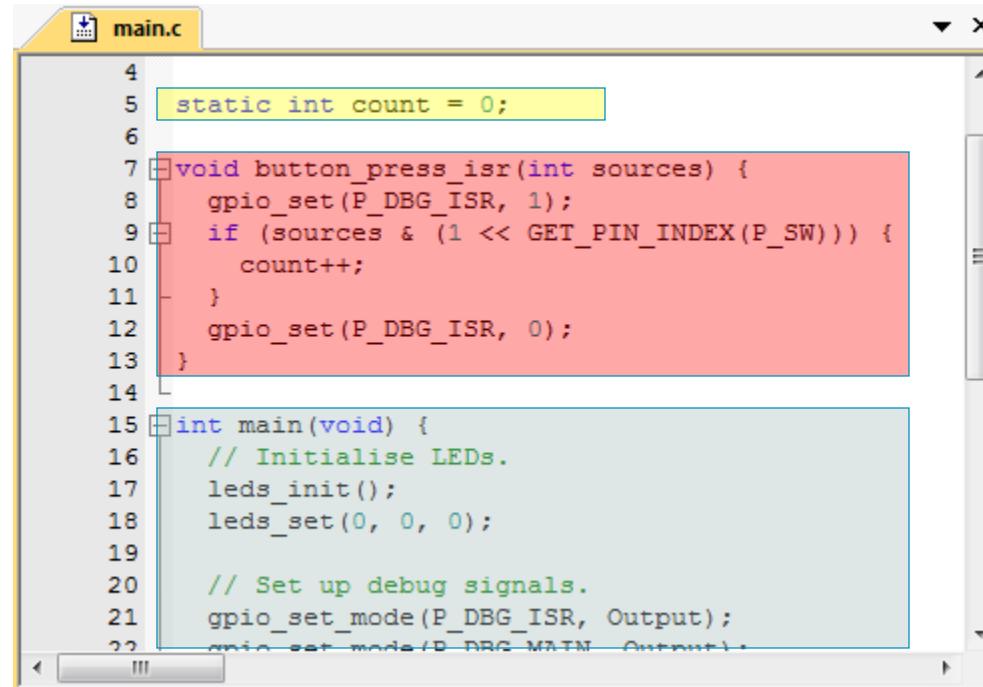
# Example program requirements and design



- When switch SW is pressed, ISR will increment count variable.
- Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1).
- Main code will toggle its debug line each time it executes.
- ISR will raise its debug line (and lower main's debug line) whenever it is executing.

# Example exception handler

- Now we will examine the processor's response to exception in detail:



The screenshot shows a code editor window titled "main.c". The code is written in C and includes the following functions and variables:

```
4 static int count = 0;
5
6 void button_press_isr(int sources) {
7 gpio_set(P_DBG_ISR, 1);
8 if (sources & (1 << GET_PIN_INDEX(P_SW))) {
9 count++;
10 }
11 gpio_set(P_DBG_ISR, 0);
12}
13
14
15 int main(void) {
16 // Initialise LEDs.
17 leds_init();
18 leds_set(0, 0, 0);
19
20 // Set up debug signals.
21 gpio_set_mode(P_DBG_ISR, Output);
22 gpio_set_mode(P_DBG_MTM, Output);
```

The code editor uses color coding for syntax: yellow for comments, green for strings, blue for keywords, purple for identifiers, and red for errors or warnings. A pink rectangular box highlights the entire body of the `button_press_isr` function, while a light blue box highlights the entire body of the `main` function.

# Use debugger for detailed processor view

- Can see registers, stack, source code, disassembly (object code)
- Note: compiler may generate code for function entry
- Place breakpoint on the handler function declaration line in source code, not at the first line of code

The screenshot shows a debugger interface with three main panes:

- Registers** pane: Displays the ARM processor's general-purpose registers (R0-R12, R13 (SP), R14 (LR), R15 (PC), xPSR) and other processor states (Banked, System, Internal). The PC register is highlighted.
- Disassembly** pane: Shows the assembly code for the current function. A yellow highlight covers the assembly code from address 8 to 10. A red circle with a question mark is placed over the first instruction at address 8.
- Source code** pane: Displays the C source code for the function. A yellow highlight covers the same code block as in the disassembly pane. A red circle with a question mark is placed over the first line of code at line 7.

**Registers** pane content (approximate values):

| Register | Value      |
|----------|------------|
| R0       | 0x000C0080 |
| R1       | 0x200000DC |
| R2       | 0x40040400 |
| R3       | 0x00000001 |
| R4       | 0x000C0080 |
| R5       | 0x000C0080 |
| R6       | 0x40040000 |
| R7       | 0x0000020D |
| R8       | 0x9C3BB72F |
| R9       | 0x20000768 |
| R10      | 0x000011E0 |
| R11      | 0x000011E0 |
| R12      | 0xFFFFF0AA |
| R13 (SP) | 0x20000FA0 |
| R14 (LR) | 0x00000D8F |
| R15 (PC) | 0x00000210 |
| xPSR     | 0x21000010 |

**Disassembly** pane content (addresses 8-10):

```
8: gpio_set(P_DBG_ISR, 1);
0x00000210 2101 MOVS r1,#0x01
0x00000212 0488 LSLS r0,r1,#18
0x00000214 F000FCF4 BL.W gpio_set (0x00000C00)
9: if (sources & (1 << GET_PIN_INDEX(P_SW))) {
0x00000218 2080 MOVS r0,#0x80
0x0000021A 4020 ANDS r0,r0,r4
0x0000021C 2800 CMP r0,#0x00
0x0000021E D004 BEQ 0x0000022A
10: count++;
```

**Source code** pane content (main.c):

```
4
5 static int count = 0;
6
7 void button_press_isr(int sources) {
8 gpio_set(P_DBG_ISR, 1);
9 if (sources & (1 << GET_PIN_INDEX(P_SW))) {
10 count++;
11 }
12 gpio_set(P_DBG_ISR, 0);
13 }
14
15 int main(void) {
16 // Initialization LEDs
```

# Entering exception handler: CPU hardwired exception processing

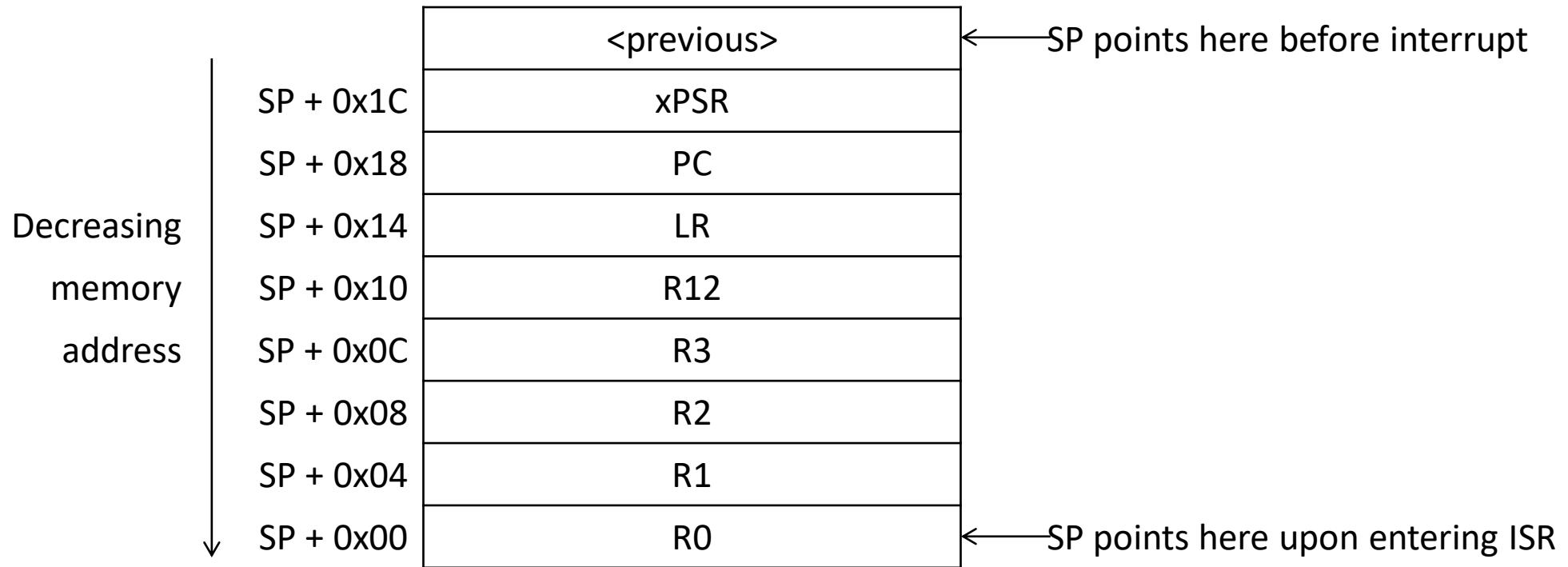
- Finish current instruction
  - Except for lengthy instructions
- Push context (registers) onto current stack (MSP or PSP)
  - xPSR, Return Address, LR(R14), R12, R3, R2, R1, R0
- Switch to handler/privileged mode, use MSP
- Load PC with address of interrupt handler
- Load LR with EXC\_RETURN code
- Load IPSR with exception number
- Start executing code of interrupt handler

Usually 16 cycles from exception request to execution of first instruction in handler

# 1. Finish current instruction

- Most instructions are short and finish quickly.
- Some instructions may take many cycles to execute.
  - Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS (32 cycles for some CPU core implementations)
- This will delay interrupt response significantly.
- If one of these is executing when the interrupt is requested, the processor:
  - *abandons* the instruction
  - responds to the interrupt
  - executes the ISR
  - returns from interrupt
  - *restarts* the abandoned instruction

## 2. Push context onto current stack

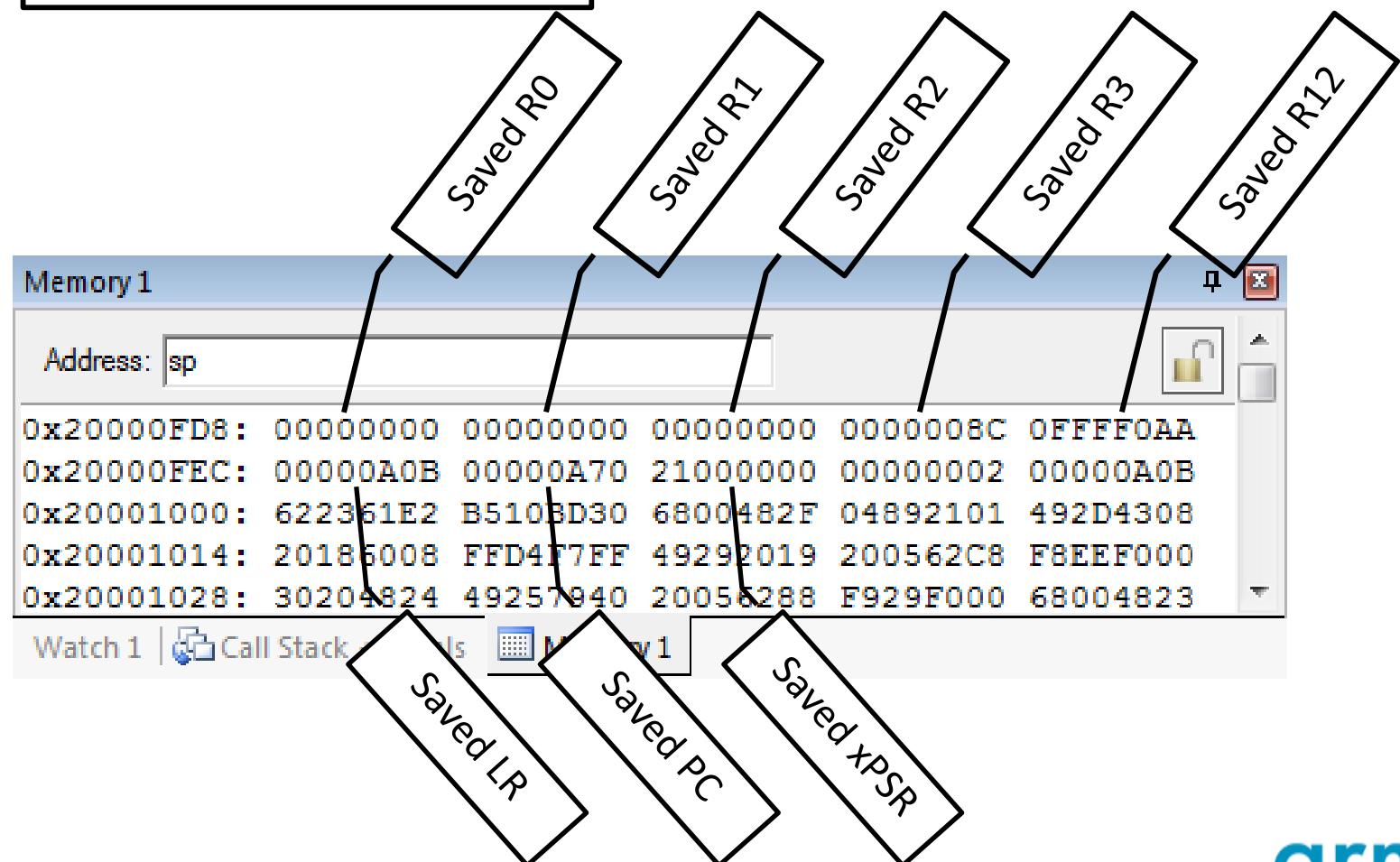


- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1
- Stack grows toward smaller addresses

# Context saved on stack

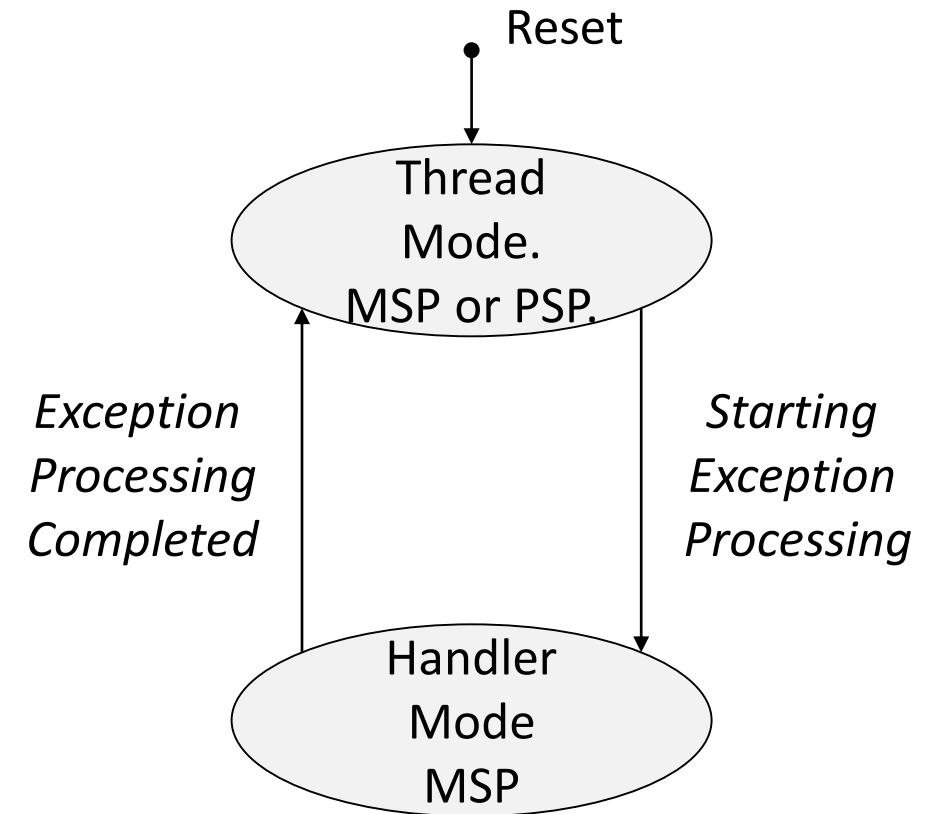
| Register        | Value      |
|-----------------|------------|
| <b>Core</b>     |            |
| R0              | 0x00000000 |
| R1              | 0x00000000 |
| R2              | 0x00000000 |
| R3              | 0x0000008C |
| R4              | 0x40040000 |
| R5              | 0x00000ECC |
| R6              | 0x2EE00000 |
| R7              | 0x00000EAB |
| R8              | 0x9C7FB76F |
| R9              | 0x20000768 |
| R10             | 0x00000ECC |
| R11             | 0x00000ECC |
| R12             | 0x0FFF0AA  |
| R13 (SP)        | 0x20000FD8 |
| R14 (LR)        |            |
| R15 (PC)        |            |
| xPSR            |            |
| <b>Banked</b>   |            |
| MSP             | 0x20000FD8 |
| PSP             | 0xBEF5DFF0 |
| <b>System</b>   |            |
| PRIMASK         | 0          |
| CONTROL         | 0x00       |
| <b>Internal</b> |            |
| Mode            |            |
| Stack           |            |

SP value is reduced since registers have been pushed onto stack

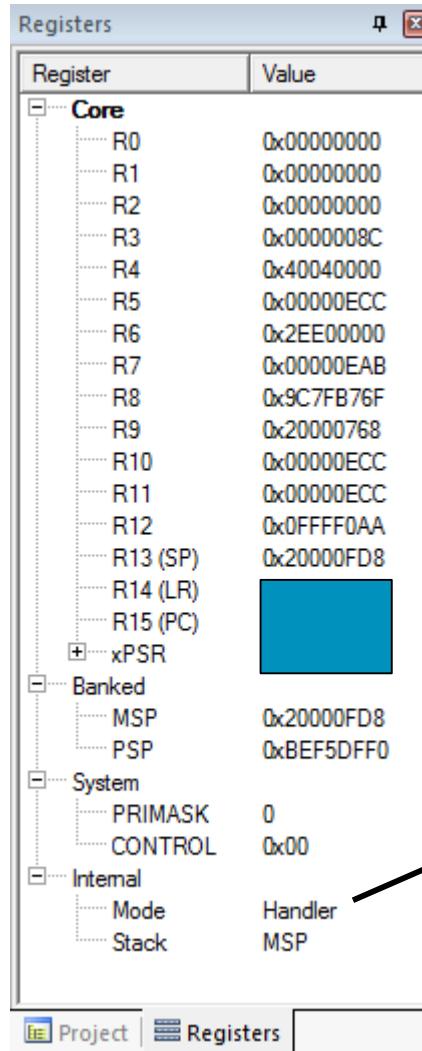


### 3. Switch to handler/privileged mode

- Handler mode always uses Main SP



# Handler and privileged mode



The screenshot shows the Registers window of a debugger. The window title is "Registers". The table has two columns: "Register" and "Value".

| Register | Value      |
|----------|------------|
| Core     |            |
| R0       | 0x00000000 |
| R1       | 0x00000000 |
| R2       | 0x00000000 |
| R3       | 0x0000008C |
| R4       | 0x40040000 |
| R5       | 0x00000ECC |
| R6       | 0x2EE00000 |
| R7       | 0x00000EAB |
| R8       | 0x9C7FB76F |
| R9       | 0x20000768 |
| R10      | 0x00000ECC |
| R11      | 0x00000ECC |
| R12      | 0x0FFFF0AA |
| R13 (SP) | 0x20000FD8 |
| R14 (LR) | [Redacted] |
| R15 (PC) | [Redacted] |
| xPSR     |            |
| Banked   |            |
| MSP      | 0x20000FD8 |
| PSP      | 0xBEF5DFF0 |
| System   |            |
| PRIMASK  | 0          |
| CONTROL  | 0x00       |
| Internal |            |
| Mode     | Handler    |
| Stack    | MSP        |

A callout box points from the "Mode" entry in the Internal section to a text box containing the following text:

Mode changed to Handler. Was already using MSP

# Update IPSR with exception number

| Register | Value      |
|----------|------------|
| Core     |            |
| R0       | 0x00000000 |
| R1       | 0x00000000 |
| R2       | 0x00000000 |
| R3       | 0x0000008C |
| R4       | 0x40040000 |
| R5       | 0x00000ECC |
| R6       | 0x2EE00000 |
| R7       | 0x00000EAB |
| R8       | 0x9C7FB76F |
| R9       | 0x20000768 |
| R10      | 0x00000ECC |
| R11      | 0x00000ECC |
| R12      | 0x0FFFF0AA |
| R13 (SP) | 0x20000FD8 |
| R14 (LR) |            |
| R15 (PC) | 0x21000010 |
| xPSR     | 0x21000010 |
| Banked   |            |
| MSP      | 0x20000FD8 |
| PSP      | 0xBEF5DFF0 |
| System   |            |
| PRIMASK  | 0          |
| CONTROL  | 0x00       |
| Internal |            |
| Mode     | Handler    |
| Stack    | MSP        |

Exception number 0x10  
(interrupt number + 0x10)

## 4. Load PC with address of exception handler

| Memory Address  | Value                 |
|-----------------|-----------------------|
| 0x0000_0000     | Initial Stack Pointer |
| 0x0000_0004     | Reset                 |
| 0x0000_0008     | NMI_IRQHandler        |
| ...             |                       |
|                 | IRQ0_Handler          |
|                 | IRQ1_Handler          |
| ...             |                       |
| Reset:          |                       |
| ...             |                       |
| NMI_IRQHandler: |                       |
| ...             |                       |
| IRQ0_Handler:   |                       |
| ...             |                       |
| IRQ1_Handler:   |                       |

- Which Program Counter is selected from the vector table depends on which exception is used

# Examine vector table with debugger

| Exception number | IRQ number | Vector     | Offset  |
|------------------|------------|------------|---------|
|                  |            | Initial SP | 0x00    |
| 1                |            | Reset      | 0x04    |
| 2                | -14        | NMI        | 0x08    |
| 3                | -13        | HardFault  | 0x0C    |
| 4                |            |            | 0x10    |
| 5                |            |            |         |
| 6                |            |            |         |
| 7                |            |            |         |
| 8                |            |            |         |
| 9                |            |            |         |
| 10               |            |            |         |
| 11               | -5         | SVCall     | 0x2C    |
| 12               |            |            |         |
| 13               |            |            |         |
| 14               | -2         | PendSV     | 0x38    |
| 15               | -1         | SysTick    | 0x3C    |
| 16               | 0          | IRQ0       | 0x40    |
| 17               | 1          | IRQ1       | 0x44    |
| 18               | 2          | IRQ2       | 0x48    |
| .                | .          | .          |         |
| 16+n             | n          | IRQn       | 0x40+4n |

- Why is the vector odd?
- LSB of address indicates that handler uses Thumb code

# Upon entry to handler

| Register    | Value      |
|-------------|------------|
| <b>Core</b> |            |
| R0          | 0x00000000 |
| R1          | 0x00000000 |
| R2          | 0x00000000 |
| R3          | 0x0000008C |
| R4          | 0x40040000 |
| R5          | 0x00000ECC |
| R6          | 0x2EE00000 |
| R7          | 0x00000EAB |
| R8          | 0x9C7FB76F |
| R9          | 0x20000768 |
| R10         | 0x00000ECC |
| R11         | 0x00000ECC |
| R12         | 0x0FFF0AA  |
| R13 (SP)    | 0x20000FD8 |
| R14 (LR)    | 0xFFFFFFF9 |
| R15 (PC)    | 0x00000ACC |
| xPSR        | 0x21000010 |
| Banked      |            |
| System      |            |
| Internal    |            |
| Mode        | Handler    |
| Stack       | MSP        |

```
42: void switch_isr(void) {
 0x00000ACC B510 PUSH {r4,lr}
```

PC has been loaded with  
start address of handler

## 5. Load LR with EXC\_RETURN code

- EXC\_RETURN value generated by CPU to provide information on how to return
  - Which SP to restore registers from? MSP (0) or PSP (1)
    - Previous value of SPSEL
  - Which mode to return to? Handler (0) or Thread (1)
    - Another exception handler may have been running when this exception was requested

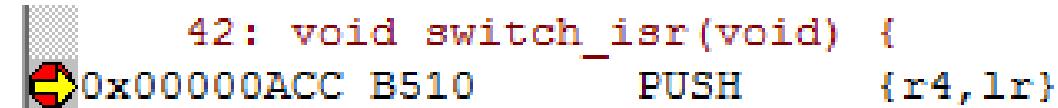
| EXC_RETURN  | Return Mode | Return Stack | Description                 |
|-------------|-------------|--------------|-----------------------------|
| 0xFFFF_FFF1 | 0 (Handler) | 0 (MSP)      | Return to exception handler |
| 0xFFFF_FFF9 | 1 (Thread)  | 0 (MSP)      | Return to thread with MSP   |
| 0xFFFF_FFFD | 1 (Thread)  | 1 (PSP)      | Return to thread with PSP   |

# Updated LR with EXC\_RETURN code

| Register    | Value      |
|-------------|------------|
| <b>Core</b> |            |
| R0          | 0x00000000 |
| R1          | 0x00000000 |
| R2          | 0x00000000 |
| R3          | 0x0000008C |
| R4          | 0x40040000 |
| R5          | 0x00000ECC |
| R6          | 0x2EE00000 |
| R7          | 0x00000EAB |
| R8          | 0x9C7FB76F |
| R9          | 0x20000768 |
| R10         | 0x00000ECC |
| R11         | 0x00000ECC |
| R12         | 0x0FFF0AA  |
| R13 (SP)    | 0x20000FD8 |
| R14 (LR)    | 0xFFFFFFF9 |
| R15 (PC)    | 0x00000ACC |
| + xPSR      | 0x21000010 |
| Banked      |            |
| System      |            |
| Internal    |            |
| Mode        | Handler    |
| Stack       | MSP        |

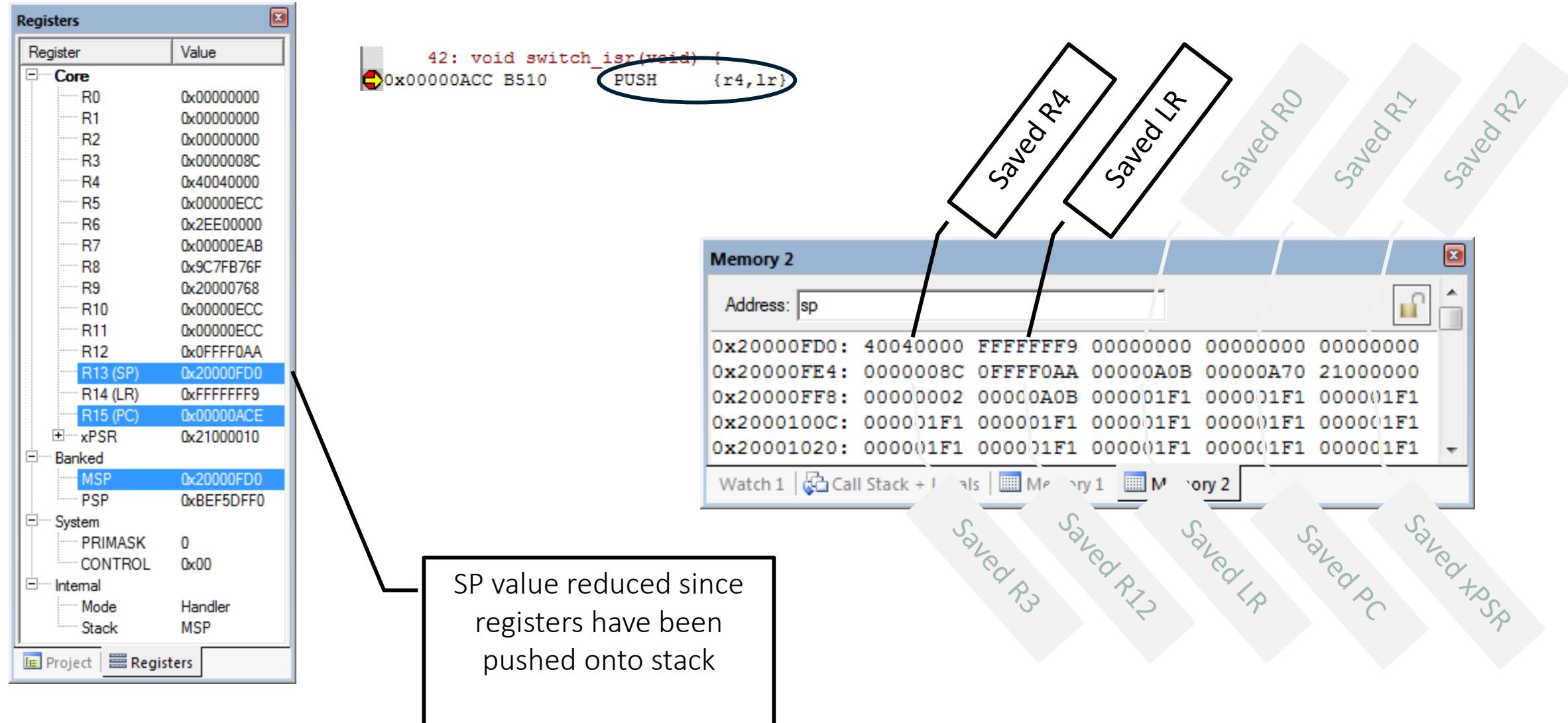
## 6. Start executing exception handler

- Exception handler starts running, unless preempted by a higher-priority exception
- Exception handler may save additional registers on stack
  - E.g. if handler may call a subroutine, LR and R4 must be saved



```
42: void switch_isr(void) {
 0x00000ACC B510 PUSH {r4,lr}
```

# After handler has saved more context



# Exiting an exception handler

1. Execute instruction triggering exception return processing
2. Select return stack, restore context from that stack
3. Resume execution of code at restored address

# 1. Execute instruction for exception return

- No “return from interrupt” instruction
- Use regular instruction instead
  - BX LR - Branch to address in LR by loading PC with LR contents
  - POP ..., PC - Pop address from stack into PC
- ... with a special value EXC\_RETURN loaded into the PC to trigger exception handling processing
  - BX LR used if EXC\_RETURN is still in LR
  - If EXC\_RETURN has been saved on stack, then use POP



```
51: }
⇒ 0x000000B08 BD10 POP {r4,pc}
```

# What will be popped from stack?

- R4: 0x4040\_0000
- PC: 0xFFFF\_FFF9

The screenshot shows the Registers and Memory windows of an ARM debugger.

**Registers Window:**

| Register | Value      |
|----------|------------|
| R0       | 0x0000008C |
| R1       | 0x40040000 |
| R2       | 0xE000E280 |
| R3       | 0x0000008C |
| R4       | 0x40040000 |
| R5       | 0x00000ECC |
| R6       | 0x2EE00000 |
| R7       | 0x00000EAB |
| R8       | 0x9C7FB76F |
| R9       | 0x20000768 |
| R10      | 0x00000ECC |
| R11      | 0x00000ECC |
| R12      | 0x0FFFF0AA |
| R13 (SP) | 0x20000FD0 |
| R14 (LR) | 0x00000AE1 |
| R15 (PC) | 0x00000B08 |
| xPSR     | 0x01000010 |

**Memory Window:**

| Address    | Value                                        |
|------------|----------------------------------------------|
| 0x20000FD0 | 40040000 FFFFFFF9 00000000 00000000 00000000 |
| 0x20000FE4 | 0000008C 0FFFF0AA 00000A0B 00000A70 21000000 |
| 0x20000FF8 | 00000002 00000A0B 000001F1 000001F1 000001F1 |
| 0x2000100C | 000001F1 000001F1 000001F1 000001F1 000001F1 |
| 0x20001020 | 000001F1 000001F1 000001F1 000001F1 000001F1 |

Annotations show the stack layout:

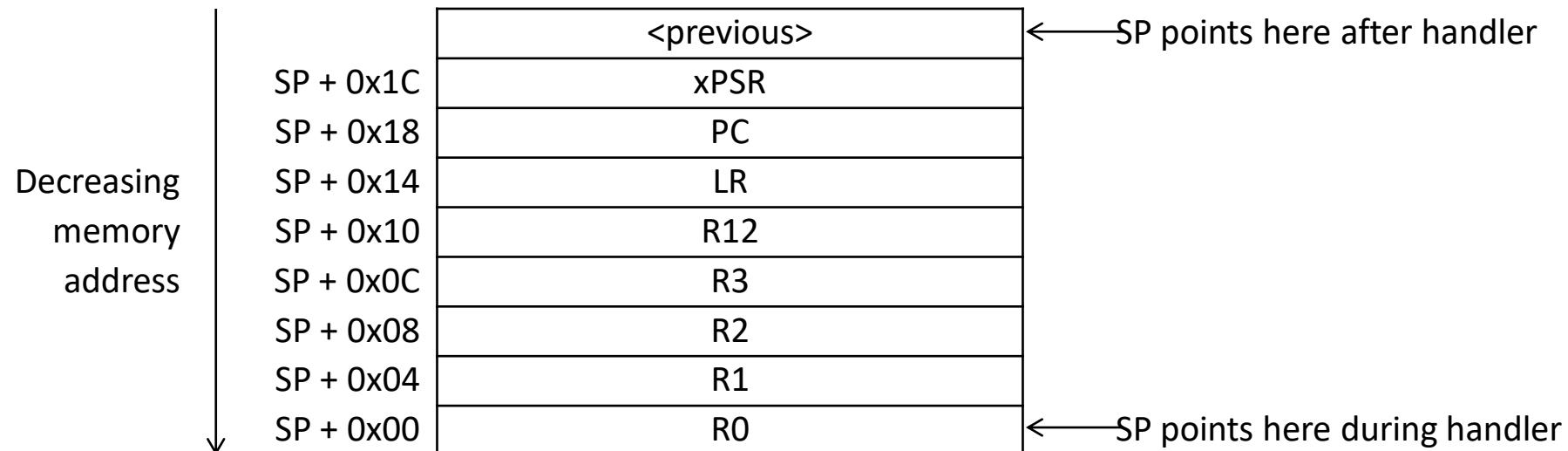
- Saved R4 (at 0x20000FD0)
- Saved LR (at 0x20000FE4)
- Saved R0 (at 0x20000FF8)
- Saved R1 (at 0x2000100C)
- Saved R2 (at 0x20001020)
- Saved R3 (at 0x20000FD0)
- Saved R12 (at 0x20000FE4)
- Saved PC (at 0x20000FF8)
- Saved xPSR (at 0x2000100C)

## 2. Select stack, restore context

- Check EXC\_RETURN (bit 2) to determine from which SP to pop the context

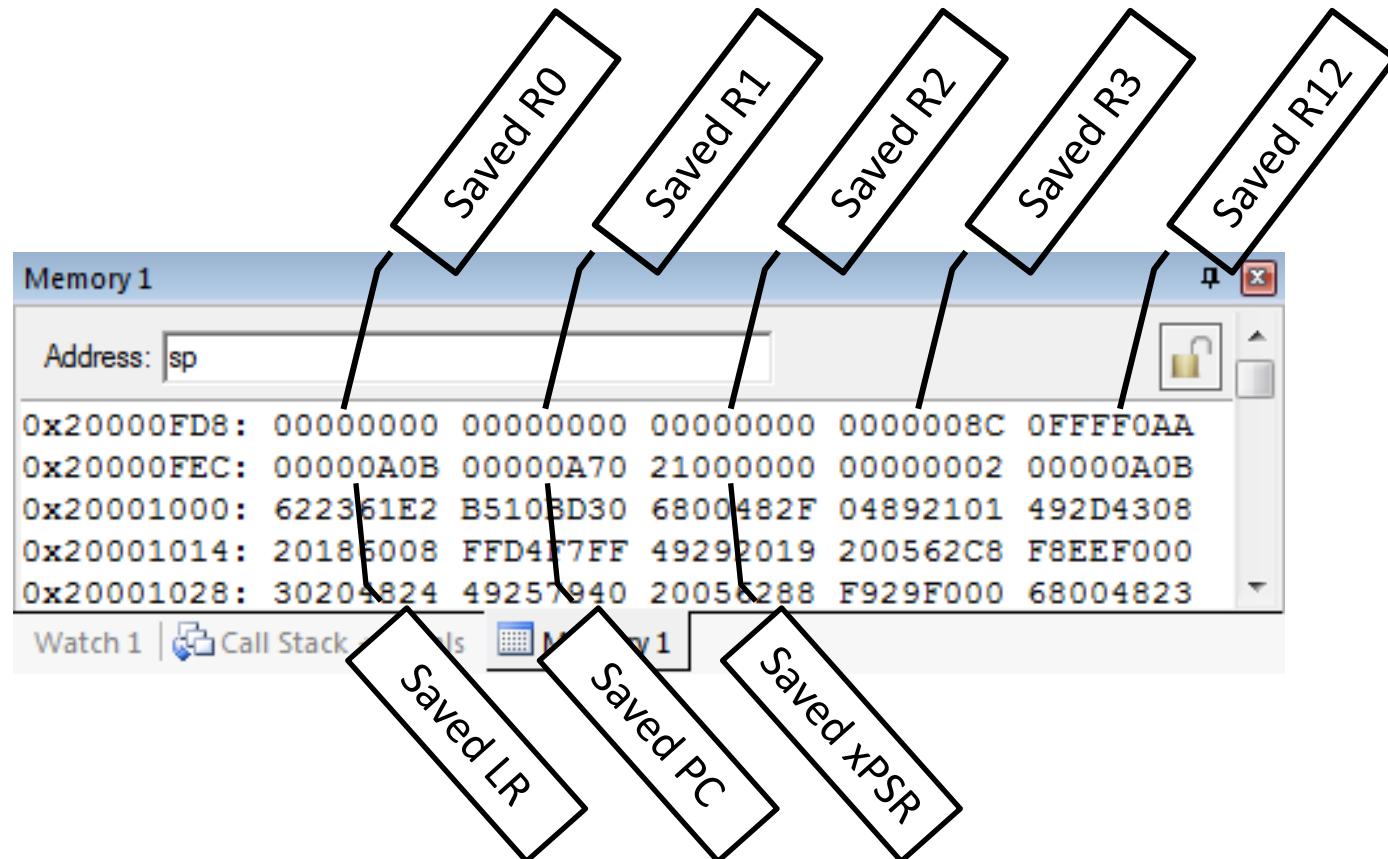
| EXC_RETURN  | Return Stack | Description                          |
|-------------|--------------|--------------------------------------|
| 0xFFFF_FFF1 | 0 (MSP)      | Return to exception handler with MSP |
| 0xFFFF_FFF9 | 0 (MSP)      | Return to thread with MSP            |
| 0xFFFF_FFFD | 1 (PSP)      | Return to thread with PSP            |

- Pop the registers from that stack



# Example

- PC=0xFFFF\_FFF9, so return to thread mode with main stack pointer
- Pop exception stack frame from stack back into registers



# Resume executing previous main thread code

- Exception handling registers have been restored: R0, R1, R2, R3, R12, LR, PC, xPSR
- SP is back to previous value
- Back in thread mode
- Next instruction to execute is at 0x0000\_0A70

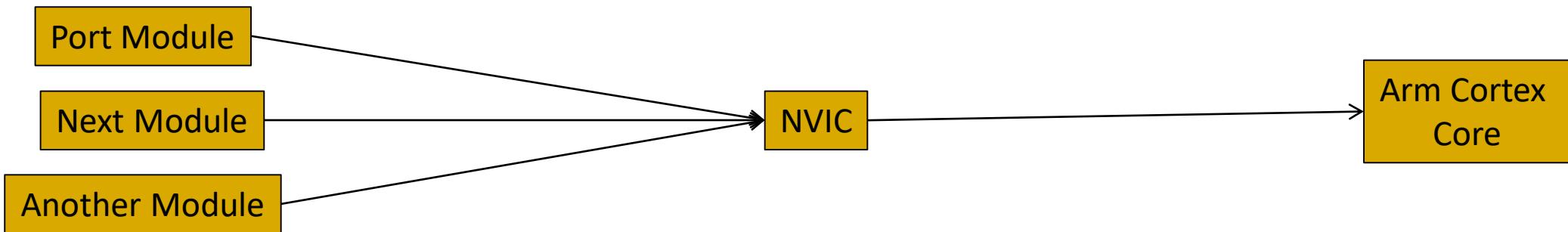
| Register | Value      |
|----------|------------|
| Core     |            |
| R0       | 0x00000000 |
| R1       | 0x00000000 |
| R2       | 0x00000000 |
| R3       | 0x0000008C |
| R4       | 0x40040000 |
| R5       | 0x00000ECC |
| R6       | 0x2EE00000 |
| R7       | 0x00000EAB |
| R8       | 0x9C7FB76F |
| R9       | 0x20000768 |
| R10      | 0x00000ECC |
| R11      | 0x00000ECC |
| R12      | 0xFFFFF0AA |
| R13 (SP) | 0x20000FF8 |
| R14 (LR) | 0x00000A0B |
| R15 (PC) | 0x00000A70 |
| xPSR     | 0x21000000 |
| Banked   |            |
| MSP      | 0x20000FF8 |
| PSP      | 0xBEF5DFF0 |
| System   |            |
| PRIMASK  | 0          |
| CONTROL  | 0x00       |
| Internal |            |
| Mode     | Thread     |
| Stack    | MSP        |

# Microcontroller interrupts

- Types of interrupts:
  - Hardware interrupts
    - *Asynchronous*: not related to what code the processor is currently executing
    - Examples: interrupt is asserted, character is received on serial port, or ADC converter finishes conversion
  - Exceptions, faults, software interrupts
    - *Synchronous*: are the result of specific instructions executing
    - Examples: undefined instructions, overflow occurs for a given instruction
  - We can enable and disable (*mask*) most interrupts as needed (*maskable*), others are *non-maskable*
- Interrupt service routine (ISR)
  - Subroutine which processor is *forced to execute* to respond to a *specific event*
  - After ISR completes, MCU goes back to previously executing code

# Nested vectored interrupt controller

- NVIC manages and prioritizes external interrupts
- Interrupts are types of exceptions
  - Exceptions 16 through 16+N
- Modes
  - Thread Mode: entered on reset
  - Handler Mode: entered on executing an exception
- Privilege level
- Stack pointers
  - Main Stack Pointer, MSP
  - Process Stack Pointer, PSP
- Exception states: Inactive, Pending, Active, A&P



# NVIC registers and state

- Enable - allows interrupt to be recognized
  - Accessed through two registers (set bits for interrupts)
    - Set enable with NVIC\_ISER, clear enable with NVIC\_ICER
  - CMSIS Interface: NVIC\_EnableIRQ(IRQnum), NVIC\_DisableIRQ(IRQnum)
- Pending - interrupt has been requested but is not yet serviced
  - CMSIS: NVIC\_SetPendingIRQ(IRQnum), NVIC\_ClearPendingIRQ(IRQnum)

# Core exception mask register

- Similar to “Global interrupt disable” bit in other MCUs
- PRIMASK - Exception mask register (CPU core)
  - Bit 0: PM Flag
    - Set to 1 to prevent activation of all exceptions with configurable priority
    - Clear to 0 to allow activation of all exception
  - Access using CPS, MSR and MRS instructions
  - Use to prevent data race conditions with code needing atomicity
- CMSIS-CORE API
  - `void __enable_irq()` - clears PM flag
  - `void __disable_irq()` - sets PM flag
  - `uint32_t __get_PRIMASK()` - returns value of PRIMASK
  - `void __set_PRIMASK(uint32_t x)` - sets PRIMASK to x

# Prioritization

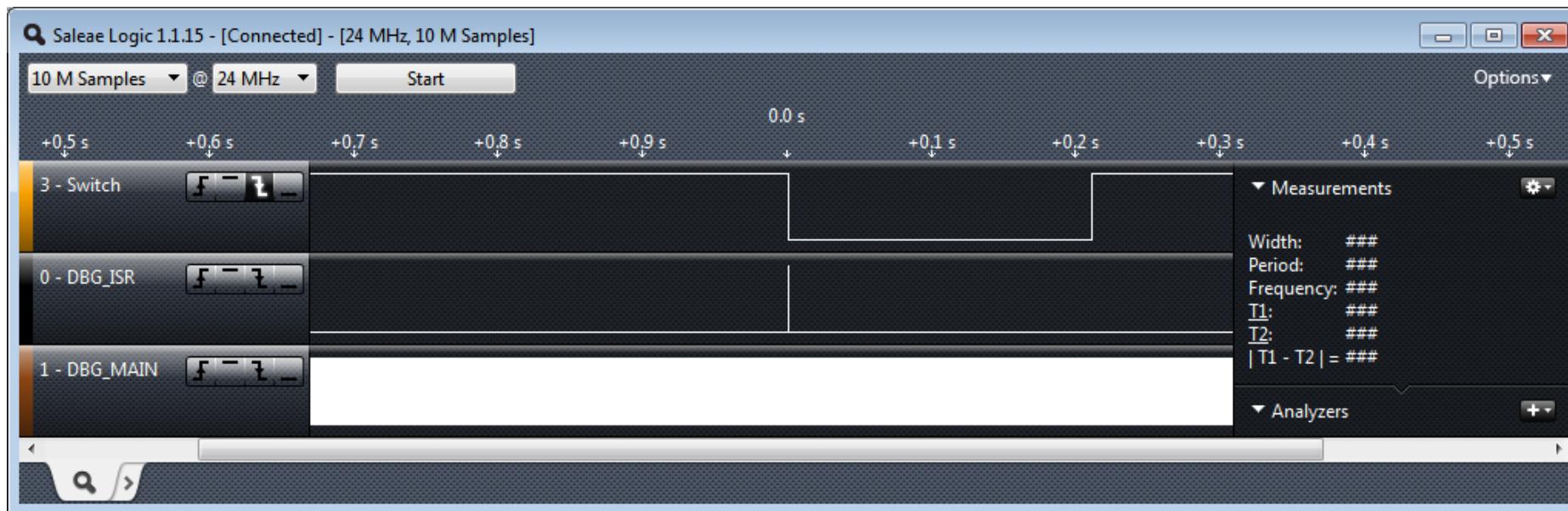
- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)
- Priorities of some exceptions are fixed
  - Reset: -3, highest priority
  - NMI: -2
  - Hard Fault: -1
- Priorities of other (peripheral) exceptions are adjustable
  - Value is stored in the interrupt priority register (IPR0-7)
  - 0x00
  - 0x40
  - 0x80
  - 0xC0

# Special cases of prioritization

- Simultaneous exception requests?
  - Lowest exception type number is serviced first
- New exception requested while a handler is executing?
  - New priority higher than current priority?
    - New exception handler preempts current exception handler
  - New priority lower than or equal to current priority?
    - New exception held in pending state
    - Current handler continues and completes execution
    - Previous priority level restored
    - New exception handled if priority level allows

# Timing analysis: big picture timing behavior

- Switch was pressed for about 0.21 s
- ISR runs in response to switch signal's falling edge
- Main seems to be running continuously (signal toggles between 1 and 0)
  - Does it really? You will investigate this in the lab exercise.



# Interrupt response latency

- Latency = time delay
- Why do we care?
  - This is overhead which wastes time, and increases as the interrupt rate rises.
  - This delays our response to external events, which may or may not be acceptable for the application, such as sampling an analog waveform.
- How long does it take?
  - Finish executing the current instruction or abandon it
  - Push various registers on to the stack, fetch vector
    - $C_{\text{IntResponseOvhd}}$ : Overhead for responding to each interrupt
  - If we have external memory with wait states, this takes longer

# Maximum interrupt rate

- We can only handle so many interrupts per second
  - $F_{Max\_Int}$ : maximum interrupt frequency
  - $F_{CPU}$ : CPU clock frequency
  - $C_{ISR}$ : Number of cycles ISR takes to execute
  - $C_{Overhead}$ : Number of cycles of overhead for saving state, vectoring, restoring state, etc.
  - $F_{Max\_Int} = F_{CPU}/(C_{ISR} + C_{Overhead})$
  - Note that model applies only when there is one interrupt in the system
- When processor is responding to interrupts, it isn't executing our other code
  - $U_{Int}$ : Utilization (fraction of processor time) consumed by interrupt processing
  - $U_{Int} = 100\% * F_{Int} * (C_{ISR} + C_{Overhead}) / F_{CPU}$
  - CPU looks like it's running the other code with CPU clock speed of  $(1 - U_{Int}) * F_{CPU}$

# Program design with interrupts

- How much work to do in ISR?
  - Trade-off: faster response for ISR code will delay completion of other code
  - In system with multiple ISRs with short deadlines, perform critical work in ISR and buffer partial results for later processing
- Should ISRs re-enable interrupts?
- How to communicate between ISR and other threads?
  - Data buffering
  - Data integrity and race conditions
    - Volatile data – can be updated outside of the program's immediate control
    - Non-atomic shared data – can be interrupted partway through read or write, is vulnerable to race conditions

# Volatile data

- Compilers assume that variables in memory don't change spontaneously, and optimize based on that belief
  - *Don't reload a variable from memory if current function hasn't changed it*
  - Read variable from memory into register (faster access)
  - Write back to memory at the end of the procedure, or before a procedure call, or when compiler runs out of free registers
- This optimization can fail
  - Example: reading from input port, polling for key press
    - While (SW\_0) ; will read from SW\_0 once and reuse that value
    - Will generate an infinite loop triggered by SW\_0 being true
- Variables for which it fails
  - Memory-mapped peripheral register – register changes on its own
  - Global variables modified by an ISR – ISR changes the variable
  - Global variables in a multithreaded application – another thread or ISR changes the variable

# The volatile directive

- We need to tell compiler which variables may change outside of its control
  - Use volatile keyword to force compiler to reload these vars from memory for each use

```
volatile unsigned int num_ints;
```

- Pointer to a volatile int

```
volatile int * var; // or
int volatile * var;
```

- Now each C source read of a variable (e.g. status register) will result in an assembly language LDR instruction
- See explanation in Nigel Jones' "Volatile," *Embedded Systems Programming* July 2001

# Non-atomic shared data

- We want to keep track of current time and date
- Use 1 Hz interrupt from timer
- System:
  - `current_time` structure tracks time and days since some reference event
  - `current_time`'s fields are updated by periodic 1Hz timer ISR

```
void GetDateTime(DateTimeType * DT) {
 DT->day = current_time.day;
 DT->hour = current_time.hour;
 DT->minute = current_time.minute;
 DT->second = current_time.second;
}
```

```
void DateTimeISR(void) {
 current_time.second++;
 if (current_time.second > 59) {
 current_time.second = 0;
 current_time.minute++;
 if (current_time.minute > 59) {
 current_time.minute = 0;
 current_time.hour++;
 if (current_time.hour > 23) {
 current_time.hour = 0;
 current_time.day++;
 ... etc.
 }
 }
 }
}
```

# Example: checking the time

- Problem: An interrupt at the wrong time will lead to half-updated data in DT
- Failure Case
  - `current_time` is {10, 23, 59, 59} (10th day, 23:59:59)
  - Task code calls `GetDateTime()`, which copies the `current_time` fields to DT: day = 10, hour = 23
  - A timer interrupt occurs, which updates `current_time` to {11, 0, 0, 0}
  - `GetDateTime()` resumes executing, copying the remaining `current_time` fields to DT: minute = 0, second = 0
  - DT now has a time stamp of {10, 23, 0, 0}.
  - *The system thinks time just jumped backwards one hour!*
- Fundamental problem: “race condition”
  - Preemption enables ISR to interrupt other code and possibly overwrite data
  - Must ensure *atomic (indivisible)* access to the object
    - Native atomic object size depends on processor’s instruction set and word size
    - 32 bits for Arm

# Examining the problem more closely

- Protect any data object which both:
  - Requires multiple instructions to read or write (non-atomic access), and
  - Is potentially written by an ISR
- How many tasks/ISRs can write to the data object?
  - If one, then we have one-way communication
    - Must *ensure the data isn't overwritten* partway through being *read*
      - Writer and reader don't interrupt each other
  - If more than one, we
    - Must *ensure the data isn't overwritten* partway through being *read*
      - Writer and reader don't interrupt each other
    - Must *ensure the data isn't overwritten* partway through being *written*
      - Writers don't interrupt each other

# Definitions

- Race condition: Anomalous behavior due to unexpected critical dependence on the relative timing of events. Result of example code depends on the relative timing of the read and write operations.
- Critical section: A section of code which creates a possible race condition. The code section can only be executed by one process at a time. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.

# Solution: briefly disable preemption

- Prevent preemption within critical section
- If an ISR can *write* to the shared data object, need to *disable interrupts*
  - save current interrupt masking state in m
  - disable interrupts
- Restore *previous state* afterwards (interrupts may have already been disabled for another reason)
- Use CMSIS-CORE to save, control and restore interrupt masking state
- Avoid disabling interrupts. Disabling delays response to all other processing requests
- Use as few instructions as possible, to make the time as short as possible

```
void GetDateTime(DateTimeType * DT) {
 uint32_t m;

 m = __get_PRIMASK();
 __disable_irq();

 DT->day = current_time.day;
 DT->hour = current_time.hour;
 DT->minute = current_time.minute;
 DT->second = current_time.second;
 __set_PRIMASK(m);
}
```



<sup>†</sup>The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

[www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks)

arm

# General Purpose I/O

# Learning Objectives

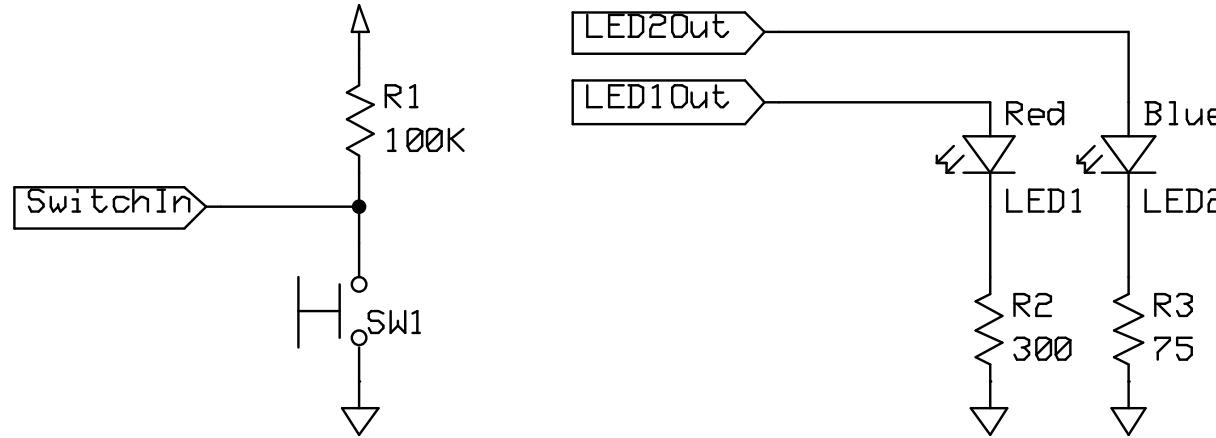
At the end of this lecture, you should be able to:

- Explain the concept of GPIO alternative functions and outline its advantages.
- Explain the functions and relevance of pull-up and pull-down resistors used in IO pins.
- Describe a simple input synchronisation circuitry consisting of two D-flipflops and explain its importance.
- Describe the GPIO code structure and outline its layers.
- Write C program to set the GPIO mode and turn on the on-board LED.

# Overview

- How do we make a program light up LEDs in response to a switch?
- GPIO
  - Basic Concepts
  - Port Circuitry
  - Alternate Functions
  - Peripheral Access In C
- Circuit Interfacing
  - Inputs
  - Outputs
- Additional Port Configuration

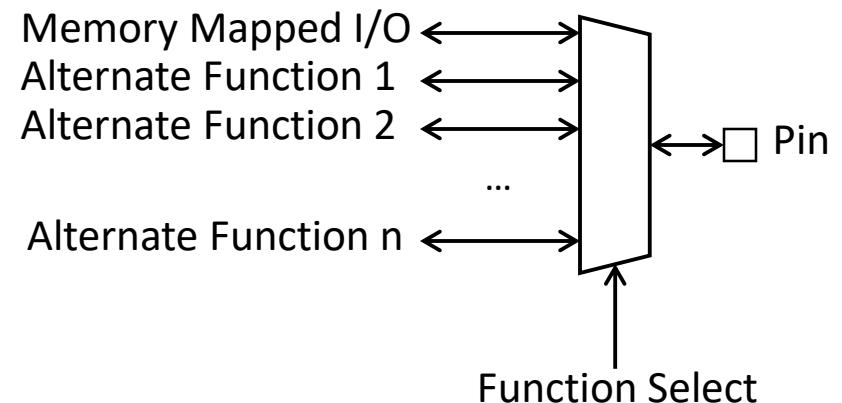
# Basic Concepts



- Goal: light either LED1 or LED2 based on switch SW1 position
- GPIO = General-purpose input and output (digital)
  - Input: program can determine if input signal is a 1 or a 0
  - Output: program can set output to 1 or 0
- Can use this to interface with external devices
  - Input: switch
  - Output: LEDs

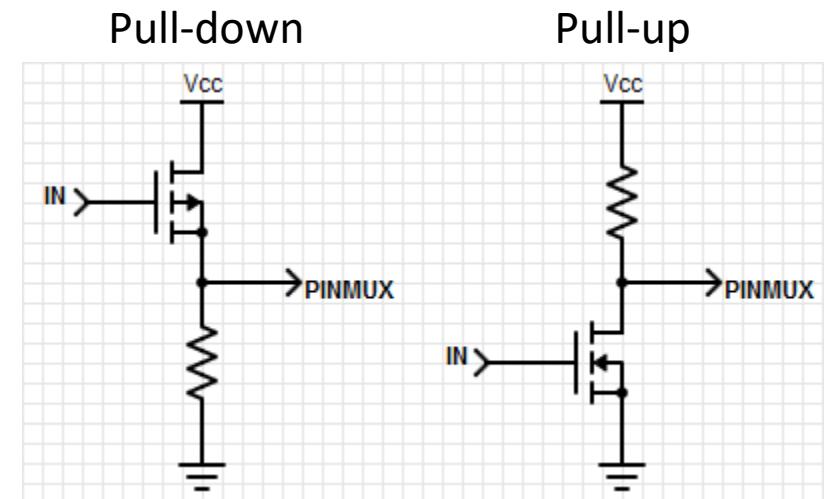
# GPIO Alternative Functions

- Pins may have different features
- To enable an alternative function, set up the appropriate register
- May also have analogue paths for ADC / DAC etc.
- Advantages:
  - Saves space on the package
  - Improves flexibility



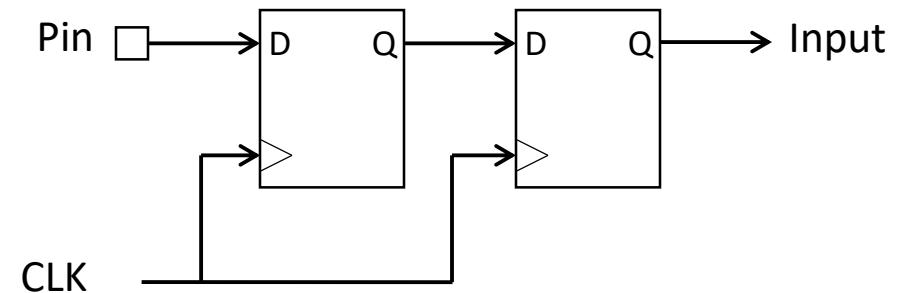
# Pull-Up & Pull-Down Resistors

- Ensure a known value on the output if a pin is left floating
- In our example, we want the switch SW1 to pull the pin to ground, so we enable the pull-up
- The pin value is:
  - High when SW1 is not pressed
  - Low when SW1 is pressed



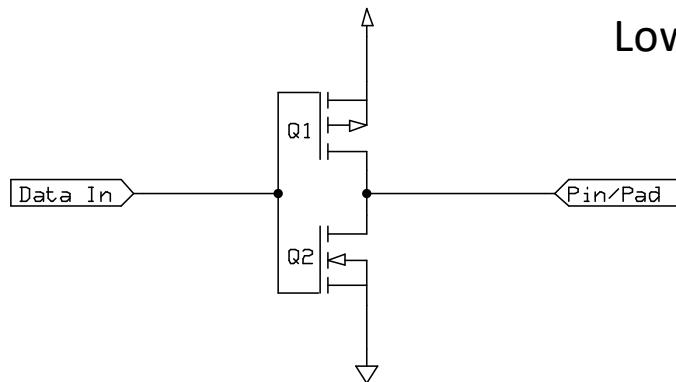
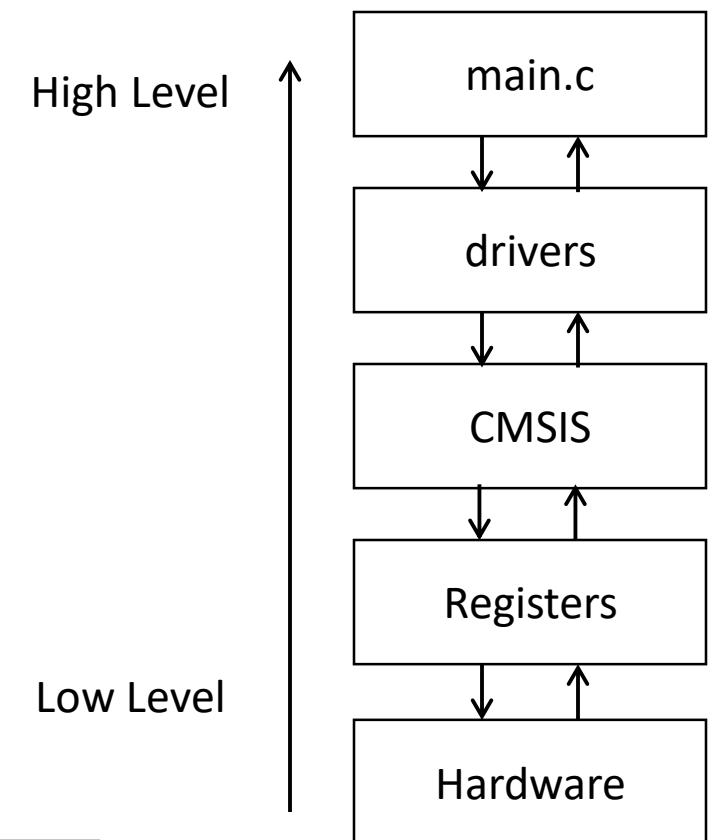
# Input Synchronization

- External signals are asynchronous to internal clock
- If an external signal changes at the same time as the clock, a flip-flop can enter a metastable state indefinitely
- Solution – synchronize the input signals with the clock
- This is done for us by hardware, no need to worry!



# Code Structure

- Main code talks to the drivers, producing easy to read and understand code
  - `gpio_set_mode(P2_5, Output)`
- Drivers utilize CMSIS library and group relevant actions
  - `port_struct->direction_reg = output`
- CMSIS transforms memory mapped registers into C structs
  - `#define PORT0 ((struct PORT*)0x2000030)`
- Registers directly control hardware
  -
- Hardware drives I/O pins physically



# Drivers Layer: How It Works

```
void gpio_set(Pin pin, int value)
1) mask = 1 << pin index
2) tmp = port_struct->data_reg & ~mask
3) tmp |= value << pin index
4) port_struct->data_reg = tmp
```

- e.g. `gpio_set(P2_5, 1)` with `PORT_DATA_REGISTER = 0b01010101`
  1. Create a mask for the bit we want to set (`0b00100000`)
  2. Invert the mask (`0b11011111`) to select all the other bits in the port data register, and save the status of the other bits (`tmp = 0b01010101`)
  3. Move the new value of the bit into position, and or it with the new register value (`tmp = 0b01110101`)
  4. Write the new data register value out to the port (`PORT_DATA_REGISTER = 0b01110101`)

# Drivers Layer: How It Works

```
int gpio_get(Pin pin)
1) mask = 1 << pin index
2) tmp = port_struct->data_reg & mask
3) tmp >>= pin index
4) return tmp
```

- e.g. `gpio_get(P2_5)` with `PORT_DATA_REGISTER = 0b01110101`
  1. Create a mask for the bit we want to get (`0b00100000`)
  2. Select the bit in the port data register based on the mask (`tmp = 0b00100000`)
  3. Bitshift the value to produce a one or zero (`tmp = 0b00000001`)
  4. Return the value of the pin back to the user

# C Interface: GPIO Configuration

```
/*! This enum describes the directional setup of a GPIO pin. */
typedef enum {
 Reset, //!< Resets the pin-mode to the default value.
 Input, //!< Sets the pin as an input with no pull-up or pull-down.
 Output, //!< Sets the pin as a low impedance output.
 PullUp, //!< Enables the internal pull-up resistor and sets as input.
 PullDown //!< Enables the internal pull-down resistor and sets as input.
} PinMode;

/*! \brief Configures the output mode of a GPIO pin.
 * Used to set the GPIO as an input, output, and configure the
 * possible pull-up or pull-down resistors.
 * \param pin Pin to set.
 * \param mode New output mode of the pin.
 */
void gpio_set_mode(Pin pin, PinMode mode);
```

# C Interface: Reading and Writing

```
/*! \brief Sets a pin to the specified logic level.
 * \param pin Pin to set.
 * \param value New logic level of the pin (0 is low, otherwise high).
 */
void gpio_set(Pin pin, int value);

/*! \brief Get the current logic level of a GPIO pin.
 * \param pin Pin to read.
 * \return The logic level of the GPIO pin (0 if low, 1 if high).
 */
int gpio_get(Pin pin);
```

# Pseudocode for Program

Make LED1 and LED2 outputs

Make switch an input with a pull-up resistor

do forever {

    if switch is not pressed {

        Turn off LED1

        Turn on LED2

    } else {

        Turn off LED2

        Turn on LED1

    }

}

# C Code

```
gpio_set_mode(P_LED1, Output); // Set LED pins to outputs
gpio_set_mode(P_LED2, Output);
gpio_set_mode(P_SW, Pullup); // Switch pin to resistive pull-up

while (1) {
 if (gpio_get(P_SW)) {
 // Switch is not pressed (active-LOW), turn LED1 off and LED2 on.
 gpio_set(P_LED1, 0);
 gpio_set(P_LED2, 1);
 } else {
 // Switch is pressed, turn off LED2 and LED1 on.
 gpio_set(P_LED2, 0);
 gpio_set(P_LED1, 1);
 }
}
```

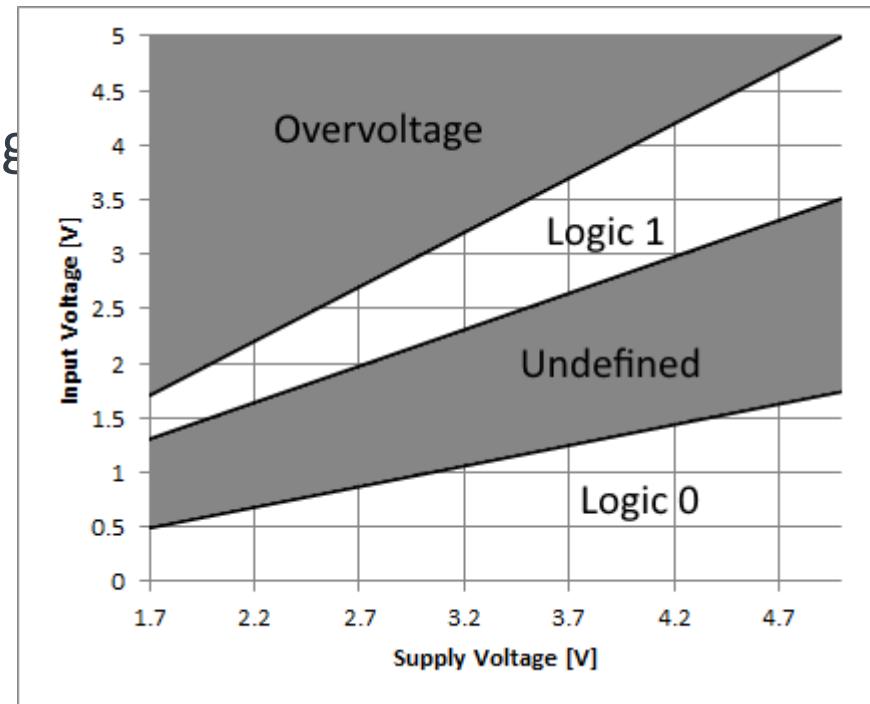
arm

# Interfacing

Inputs and Outputs, Ones and Zeros, Voltages and Currents

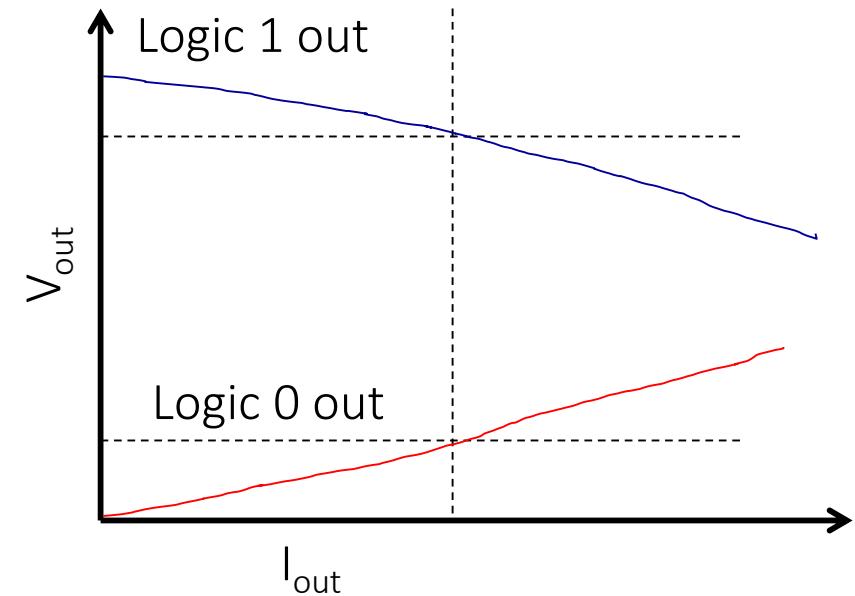
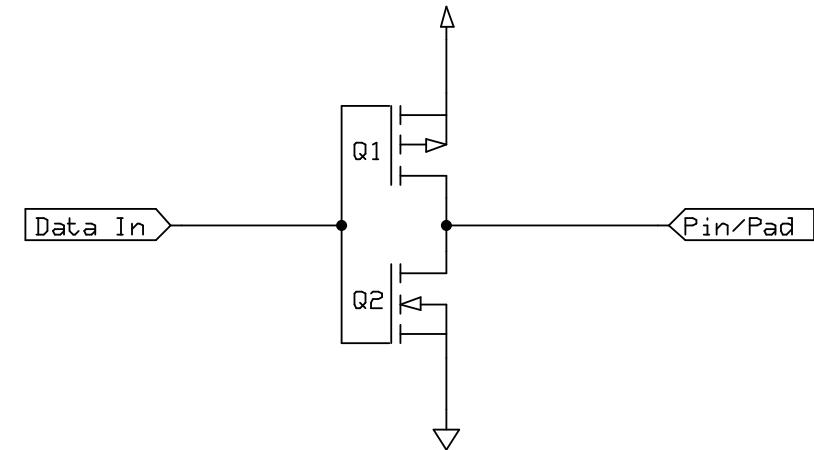
# Inputs: What's a One? A Zero?

- Input signal's value is determined by voltage
- Input threshold voltages depend on supply voltage
- Exceeding VDD or GND may damage chip



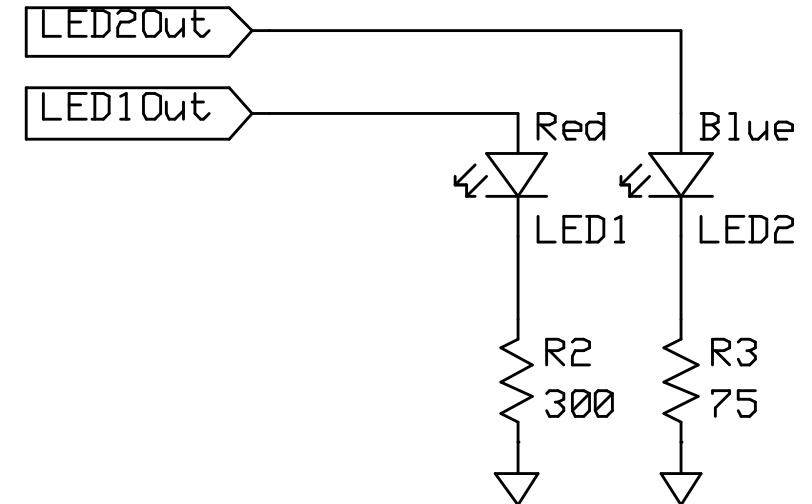
# Outputs: What's a One? A Zero?

- Nominal output voltages
  - 1: VDD-0.5 V to VDD
  - 0: 0 V to 0.5 V
- Note: Output voltage depends on current drawn by load on pin
  - Need to consider source-to-drain resistance in the transistor
  - Above values only specified when current < 5mA (18 mA for high-drive pads) and VDD > 2.7 V



# Output Example: Driving LEDs

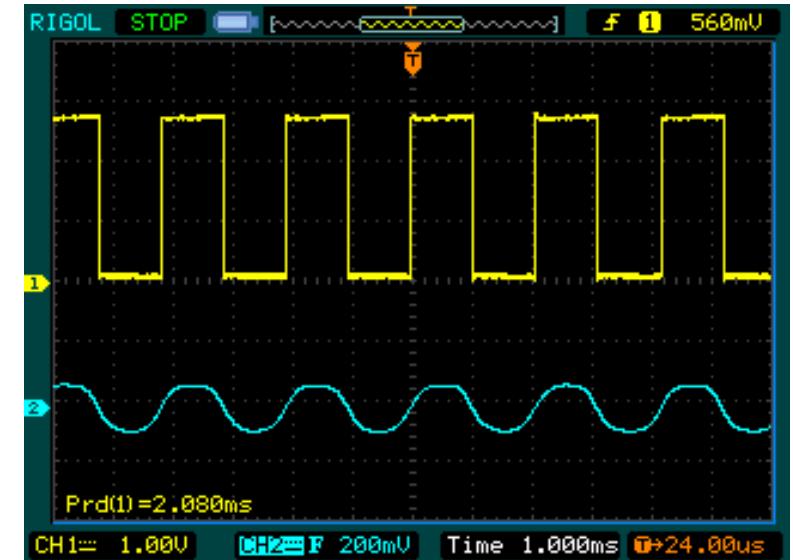
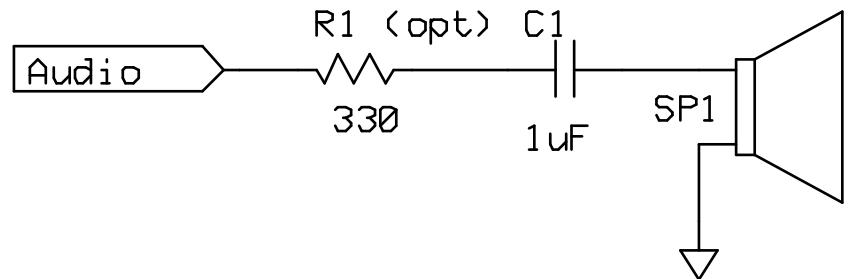
- Need to limit current to a value which is safe for both LED and MCU port driver
- Use current-limiting resistor
  - $R = (VDD - VLED)/ILED$
- Set  $ILED = 4\text{mA}$
- $VLED$  depends on type of LED (mainly color)
  - Red:  $\sim 1.8\text{V}$
  - Blue:  $\sim 2.7\text{V}$
- Solve for  $R$  given  $VDD = \sim 3.0\text{V}$ 
  - Red:  $300\text{W}$
  - Blue:  $75\text{W}$
- Demonstration code in Basic Light Switching Example



# Output Example: Driving a Speaker

- Create a square wave with a GPIO output
- Use capacitor to block DC value
- Use resistor to reduce volume if needed

```
void beep(void) {
 unsigned int period = 20;
 while (1) {
 gpio_toggle(P_SPEAKER);
 delay_ms(period/2);
 }
}
```





<sup>†</sup>The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

[www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks)

arm

# Analog Interfacing

# Learning Objectives

At the end of this lecture, you should be able to:

- Describe the theory of analog signal to digital signal conversion and vice versa.
- Describe the two common types of analog-to-digital converters (ADCs).
- Explain the properties of analog-to-digital conversion including range, resolution, quantization and sampling.
- Explain Nyquist criterion and its application in sampling frequency of the ADC.
- Explain the function of a sample and hold device in ADCs.
- Describe the mode of operations of the ADC flash and ADC successive approximation converters.
- Describe the functions of the digital-to-analog converter.
- Describe the application of ADC in power failure detection and in battery monitoring.

# Module syllabus

- Converting Between Analog and Digital Values
- Analog to Digital Conversion Concepts
- Analog Interfacing Peripherals
- Digital to Analog Converter
- Analog Comparator
- Analog to Digital Converter

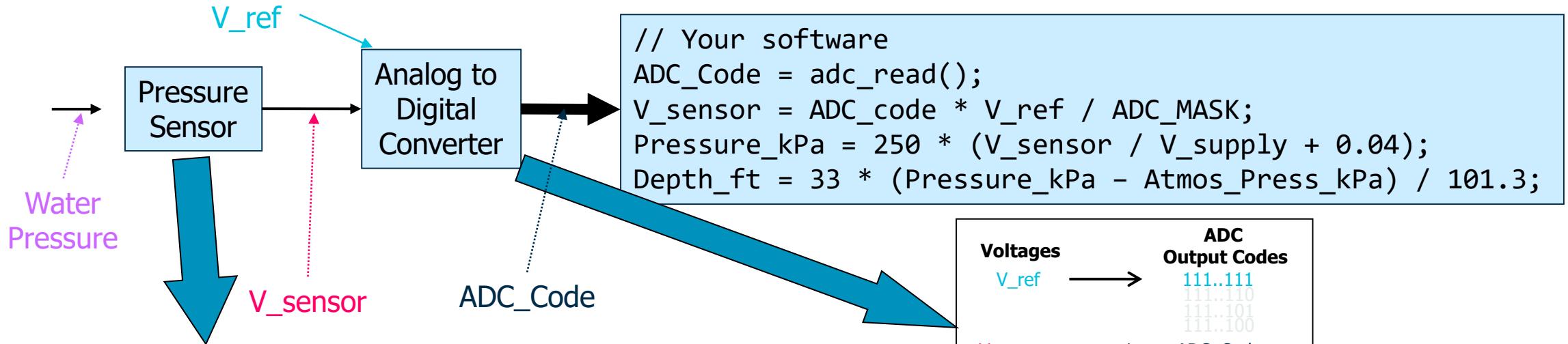
# Why It's Needed

- Embedded systems often need to measure values of physical parameters
- These parameters are usually continuous (analog) and not in a digital form which computers (which operate on discrete data values) can process
- Temperature
  - Thermometer (do you have a fever?)
  - Thermostat for building, fridge, freezer
  - Car engine controller
  - Chemical reaction monitor
  - Safety (e.g. microprocessor processor thermal management)
- Light (or infrared or ultraviolet) intensity
  - Digital camera
  - IR remote control receiver
  - Tanning bed
  - UV monitor
- Rotary position
  - Wind gauge
  - Knobs
- Pressure
  - Blood pressure monitor
  - Altimeter
  - Car engine controller
  - Scuba dive computer
  - Tsunami detector
- Acceleration
  - Air bag controller
  - Vehicle stability
  - Video game remote
- Mechanical strain
- Other
  - Touch screen controller
  - EKG, EEG
  - Breathalyzer

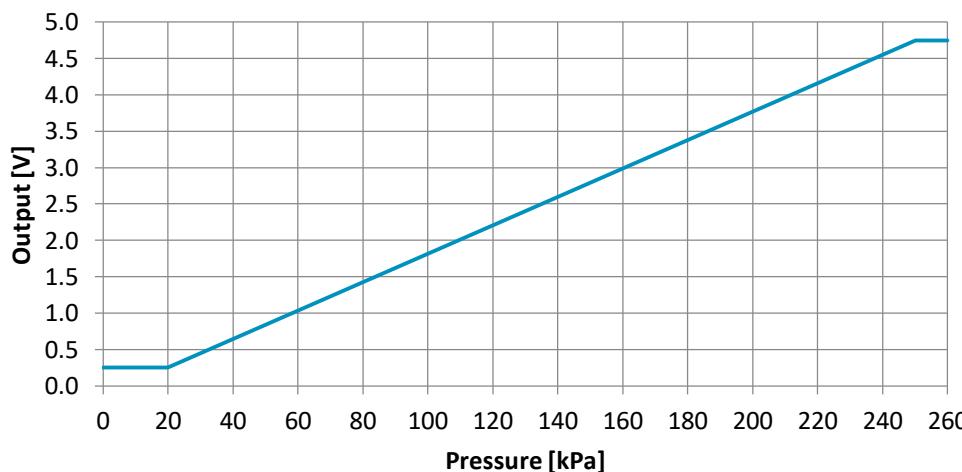


# Converting Between Analog and Digital Values

# Example Analog Sensor - Depth Gauge



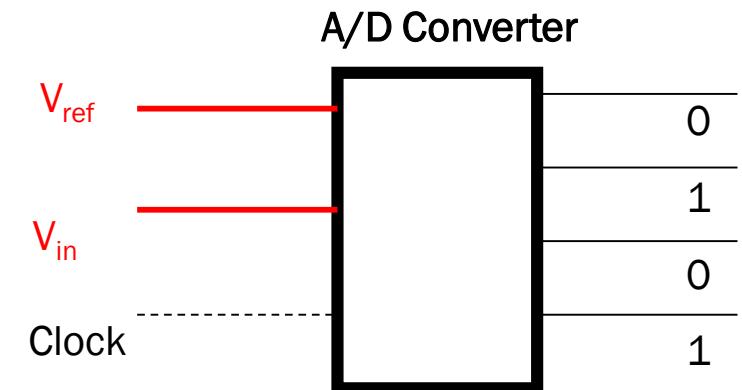
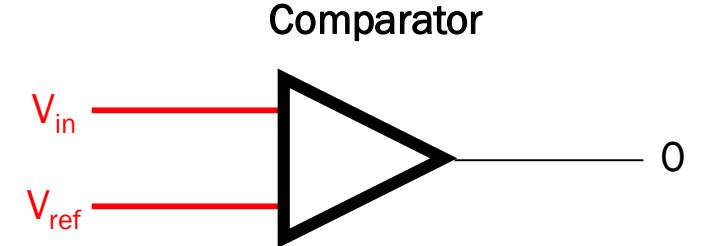
Typical Absolute Pressure vs. Output



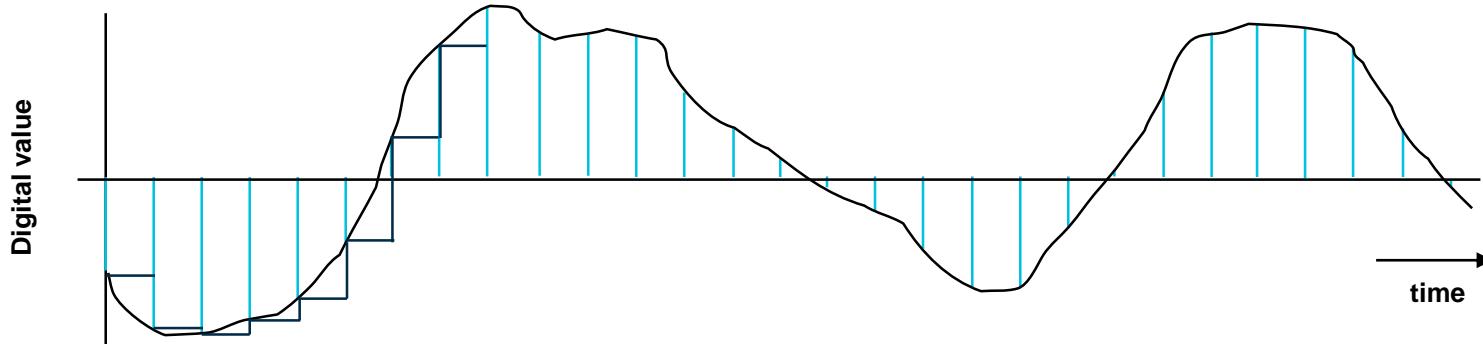
- Sensor detects pressure and generates a proportional output voltage  $V_{sensor}$
- ADC generates a proportional digital integer (code) based on  $V_{sensor}$  and  $V_{ref}$
- Code can convert that integer to something more useful
  - first a float representing the voltage,
  - then another float representing pressure,
  - finally another float representing depth

# Getting From Analog to Digital

- A Comparator tells us “Is  $V_{in} > V_{ref}$ ?”
  - Compares an analog input voltage with an analog reference voltage and determines which is larger, returning a 1-bit number
  - E.g. Indicate if depth > 100ft
  - Set  $V_{ref}$  to voltage pressure sensor returns with 100 ft depth.
- An Analog to Digital converter [AD or ADC] tells us how large  $V_{in}$  is as a fraction of  $V_{ref}$ .
  - Reads an analog input signal (usually a voltage) and produces a corresponding multi-bit number at the output.
  - E.g. calculate the depth



# Waveform Sampling and Quantization



- A waveform is sampled at a constant rate – every  $\Delta t$ 
  - Each such sample represents the instantaneous amplitude at the instant of sampling
  - “At 37 ms, the input is 1.91341914513451451234311... V”
  - Sampling converts a continuous time signal to a discrete time signal
- The sample can now be quantized (converted) into a digital value
  - Quantization represents a continuous (analog) value with the closest discrete (digital) value
  - “The sampled input voltage of 1.91341914513451451234311... V is best represented by the code 0x018, since it is in the range of 1.901 V to 1.9980 V which corresponds to code 0x018.”

# Forward Transfer Function Equations

## General Equation

What code  $n$  will the ADC use to represent voltage  $V_{in}$ ?

n = converted code

## **Simplification with $V_{ref} = 0$ V**

**Vin = sampled input voltage**

V+ref = upper voltage reference

## V-ref = lower voltage reference

N = number of bits of resolution in ADC

$$n = \left\lfloor \frac{V_{in}}{V_{+ref}} 2^N + 1/2 \right\rfloor$$

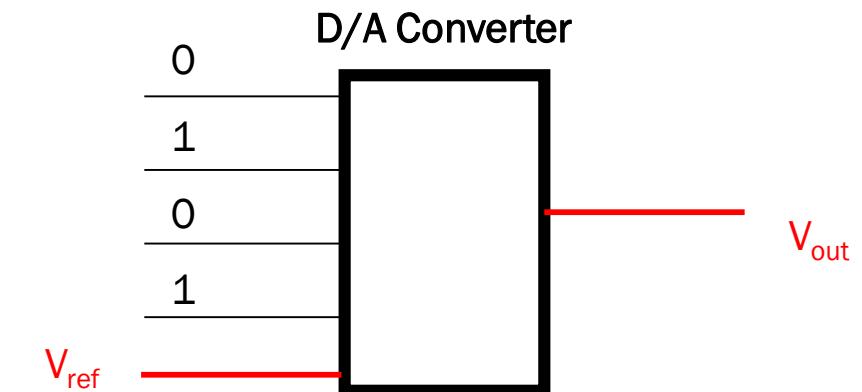
$$n = \left\lfloor \frac{3.3V}{5V} 2^{10} + 1/2 \right\rfloor = 388$$

$$n = \left\lfloor \frac{V_{in} - V_{-ref}}{V_{+ref} - V_{-ref}} 2^N + 1/2 \right\rfloor$$

$$\lfloor X \rfloor = \text{floor}(X) \quad \begin{aligned} & \text{floor}(X) \text{ nearest integer } I \text{ such that } I \leq X \\ & \text{floor}(x+0.5) \text{ rounds } x \text{ to the nearest integer} \end{aligned}$$

# Digital to Analog Conversion

- May need to generate an analog voltage or current as an output signal
  - E.g. audio signal, video signal brightness
- DAC: “Generate the analog voltage which is this fraction of Vref”
- Digital to Analog Converter equation
  - $n$  = input code
  - $N$  = number of bits of resolution of converter
  - $V_{ref}$  = reference voltage
  - $V_{out}$  = output voltage. Either
    - $V_{out} = V_{ref} * n/(2N)$  or
    - $V_{out} = V_{ref} * (n+1)/(2N)$
    - The offset +1 term depends on the internal tap configuration of the DAC – check the datasheet to be sure



# Inverse Transfer Function

## General Equation

*What range of voltages  $V_{in\_min}$  to  $V_{in\_max}$  does code n represent?*

n = converted code

$V_{in\_min}$  = minimum input voltage for code n

$V_{in\_max}$  = maximum input voltage for code n

$V_{+ref}$  = upper voltage reference

$V_{-ref}$  = lower voltage reference

N = number of bits of resolution in ADC

***Simplification with  $V_{-ref} = 0\text{ V}$***

$$V_{in\_min} = \frac{n - \frac{1}{2}}{2^N} (V_{+ref})$$

$$V_{in\_max} = \frac{n + \frac{1}{2}}{2^N} (V_{+ref})$$

$$V_{in\_min} = \frac{n - \frac{1}{2}}{2^N} (V_{+ref} - V_{-ref}) + V_{-ref}$$

$$V_{in\_max} = \frac{n + \frac{1}{2}}{2^N} (V_{+ref} - V_{-ref}) + V_{-ref}$$

# What if the Reference Voltage is not known?

- Example - running off an unregulated battery (to save power)
- Measure a known voltage and an unknown voltage

$$V_{\text{unknown}} = V_{\text{known}} \frac{n_{\text{unknown}}}{n_{\text{known}}}$$

- Many MCUs include an internal fixed voltage source which ADC can measure for this purpose
- Can also solve for Vref

$$V_{\text{ref}} = V_{\text{known}} \frac{2^N}{n}$$

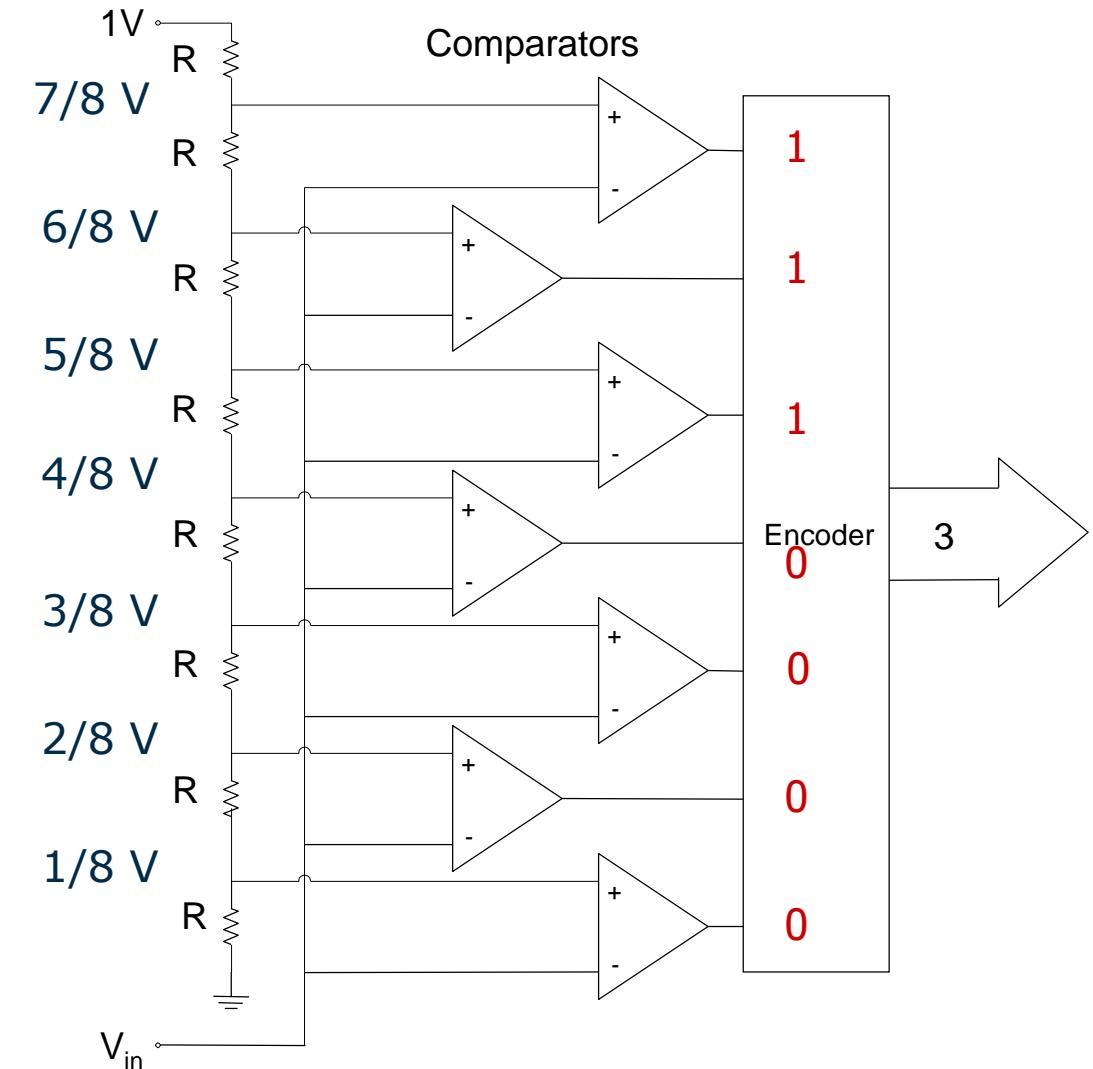
“My ADC tells me that channel 27 returns a code of 0x6543, so I can calculate that  $V_{\text{REFSH}} = 1.0V * 2^{16}/0x6543 = ...$



# Analog to Digital conversion concepts

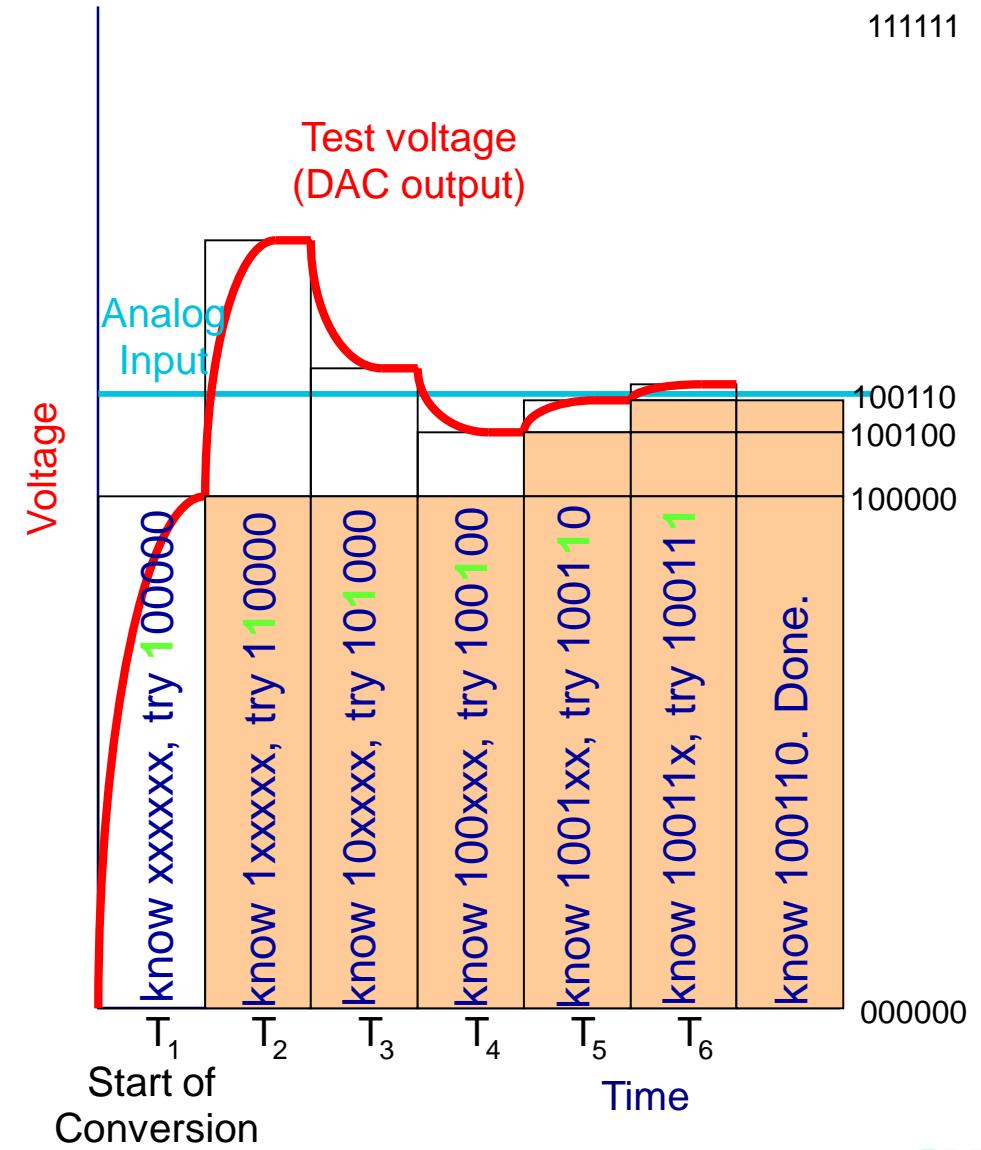
# A/D – Flash Conversion

- A multi-level voltage divider is used to set voltage levels over the complete range of conversion.
- A comparator is used at each level to determine whether the voltage is lower or higher than the level.
- The series of comparator outputs are encoded to a binary number in digital logic (a priority encoder)
- Components used
  - $2^N$  resistors
  - $2^N - 1$  comparators
- Note
  - This particular resistor divider generates voltages which are not offset by  $\frac{1}{2}$  bit, so maximum error is 1 bit
  - We could change this offset voltage by using resistors of values  $R, 2R, 2R \dots 2R, 3R$  (starting at bottom)



# ADC - Successive Approximation Conversion

- Successively approximate input voltage by using a binary search and a DAC
- SA Register holds current approximation of result
- Set all DAC input bits to 0
- Start with DAC's most significant bit
- Repeat
  - Set next input bit for DAC to 1
  - Wait for DAC and comparator to stabilize
  - If the DAC output (test voltage) is smaller than the input then set the current bit to 1, else clear the current bit to 0



# ADC Performance Metrics

- Number of bits determines overall accuracy.
- Linearity measures how well the transition voltages lie on a straight line.
- Differential linearity measure the equality of the step size.
- Conversion time: between start of conversion and generation of result.
- Conversion rate = inverse of conversion time.

# Sampling Problems

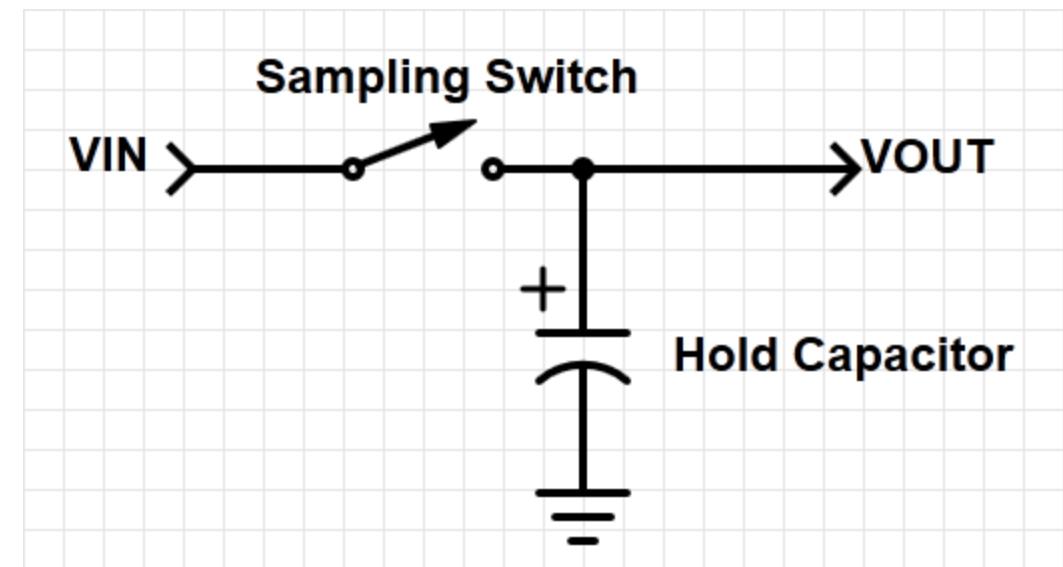
- When sampling varying signals there is an upper limit on the bandwidth of the converter which is set by the ‘Nyquist criterion’
- Nyquist criterion
  - $F_{\text{sample}} \geq 2 * F_{\text{max}}$  frequency component
  - Frequency components above  $\frac{1}{2} F_{\text{sample}}$  are aliased, distorting the measured signal
- Nyquist and the real world
  - This theorem assumes we have a perfect filter with “brick wall” roll-off
  - Real world filters have more gentle roll-off
  - Inexpensive filters are even worse (e.g. first order filter is 20 dB/decade, aka 6 dB/octave)
  - So we have to choose a sampling frequency high enough that our filter attenuates aliasing components adequately

# Inputs

- Differential
  - Use two channels, and compute difference between them
  - Very good noise immunity
  - Some sensors offer differential outputs (e.g. Wheatstone Bridge)
- Multiplexing
  - Typically share a single ADC among multiple inputs
  - Need to select an input, allow time to settle before sampling
- Signal Conditioning
  - Amplify and filter input signal
  - Protect against out-of-range inputs with clamping diodes

# Sample and Hold Devices

- Some A/D converters require the input analog signal to be held constant during conversion, (e.g. successive approximation devices)
- In other cases, peak capture or sampling at a specific point in time necessitates a sampling device.
- This function is accomplished by a sample and hold device as shown to the right.
- These devices are incorporated into some A/D converters.

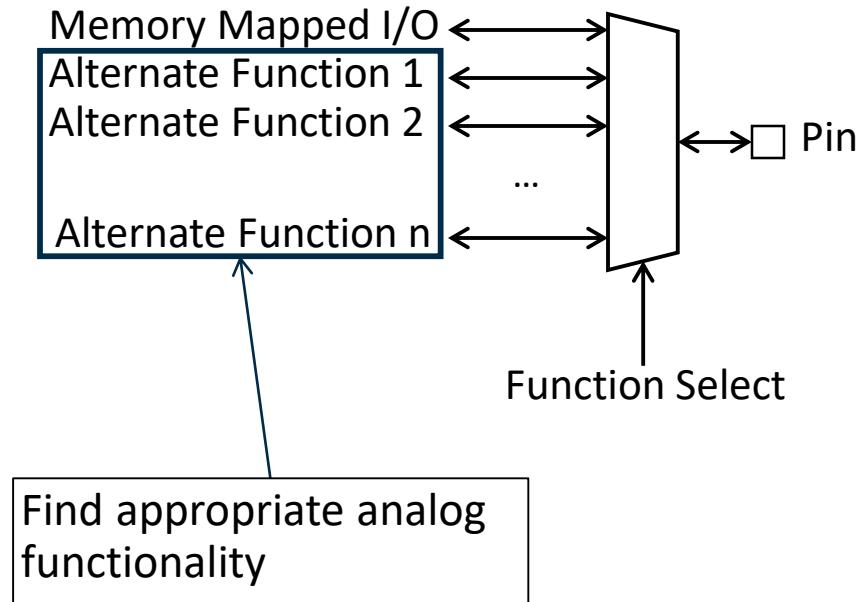




# Analog Interfacing Peripherals

# GPIO Alternative Functions

- Pins may have different features
- To enable an alternative function, set up the appropriate register
- May also have analog paths for ADC / DAC etc.
- Advantages:
  - Saves space on the package
  - Improves flexibility

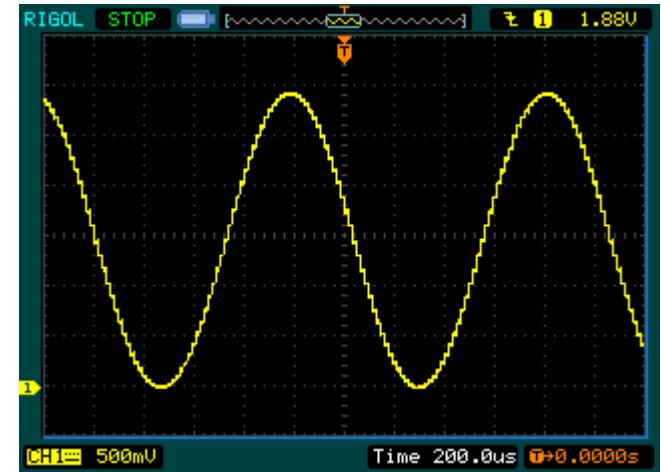
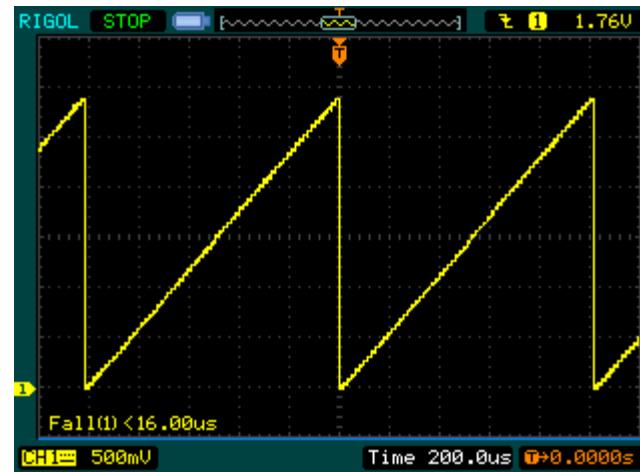
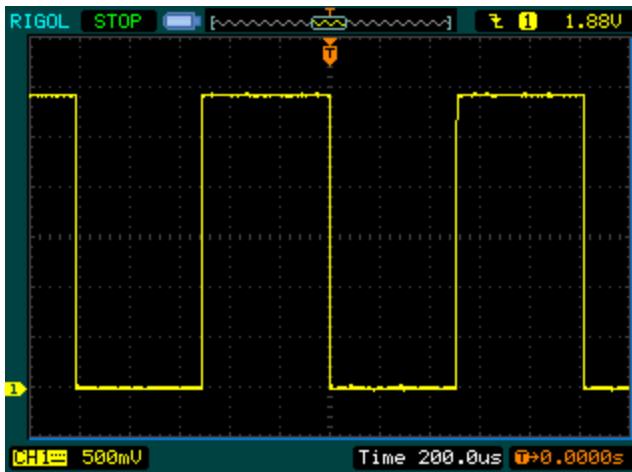


arm

# Digital to Analog Converter

# Example: Waveform Generation

- DAC can be used to generate arbitrary waveforms
  - Pre-generate lookup table
  - Update DAC output value
  - Delay
  - Repeat



## C Code – Initialization

```
void tone_init(void) {
 dac_init();
 sinewave_init();
}

void sinewave_init(void) {
 int n;
 for (n = 0; n < NUM_STEPS; n++) {
 sine_table[n] = MAX_DAC_CODE * (1 + sin(n*2*PI/NUM_STEPS)) / 2;
 }
}
```

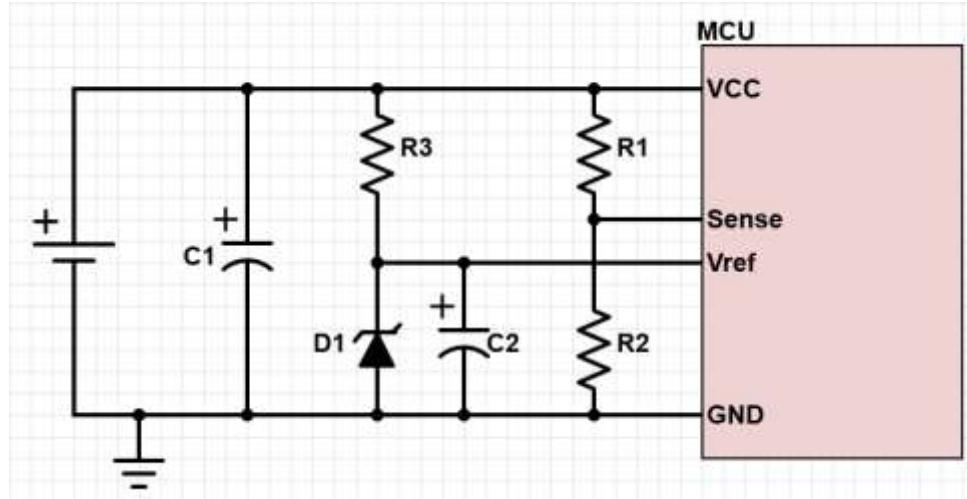
## C Code – Playback

```
void tone_play(int period_us, int num_cycles, wavetype wave) {
 int sample, step;
 while(num_cycles-- > 0) {
 for (step = 0; step < NUM_STEPS; step++) {
 switch(wave) {
 case SINE: sample = sine_table[step]; break;
 case SQUARE: sample = step < NUM_STEPS / 2 ? 0 : MAX_DAC_CODE;
 break;
 case RAMP: sample = (step * MAX_DAC_CODE) / NUM_STEPS; break;
 }
 dac_set(sample);
 delay_us(period_us);
 }
 }
}
```

arm

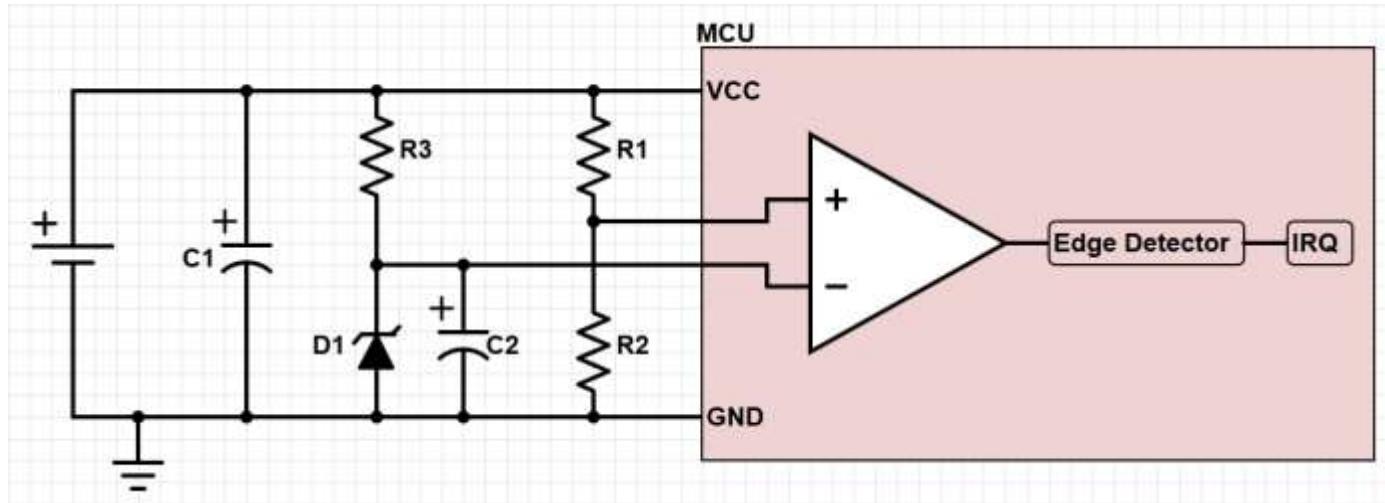
# Analog Comparator

# Example: Power Failure Detection



- Need warning of when power has failed
  - Very limited amount of time before capacitor C1 discharges
  - Save critical information
  - Turn off output devices
  - Put system into safe mode
- Can use a comparator to compare Vcc against a fixed reference voltage Vref

# Comparator Overview



- Comparator compares Sense and Vref
- Comparator output indicates if Sense > Vref (1) or Sense < Vref (0)
- Can generate an interrupt request (+, -, or +- edges)
- If the Sense input drops below Vref, fire an interrupt

## C Code – Comparator

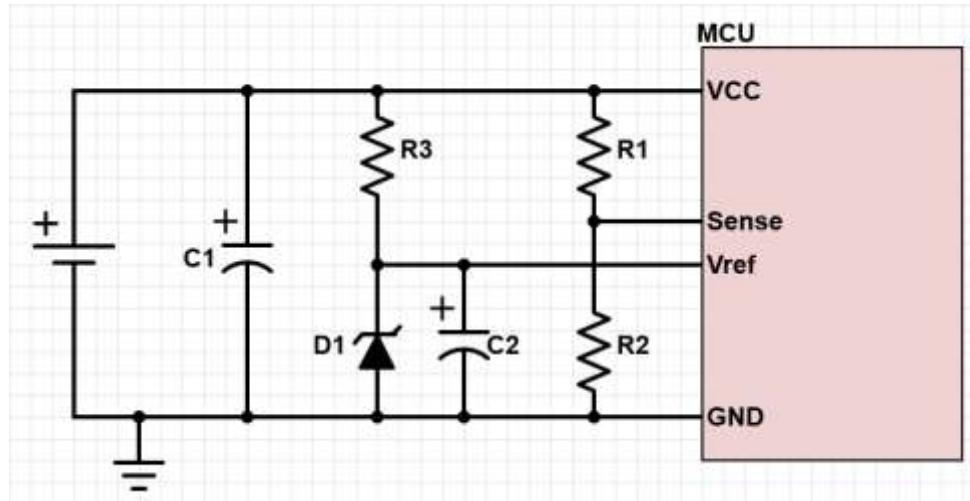
```
void comparator_isr(int state) {
 if (state) {
 // Sense > Vref, turn off LEDs.
 leds_set(0, 0, 0);
 } else {
 // Sense < Vref, turn on red LED.
 leds_set(1, 0, 0);
 }
}

int main(void) {
 comparator_init();
 comparator_set_trigger(CompBoth); // ISR on both rising and falling edges.
 comparator_set_callback(comparator_isr);
 ...
}
```

arm

# Analog to Digital Converter

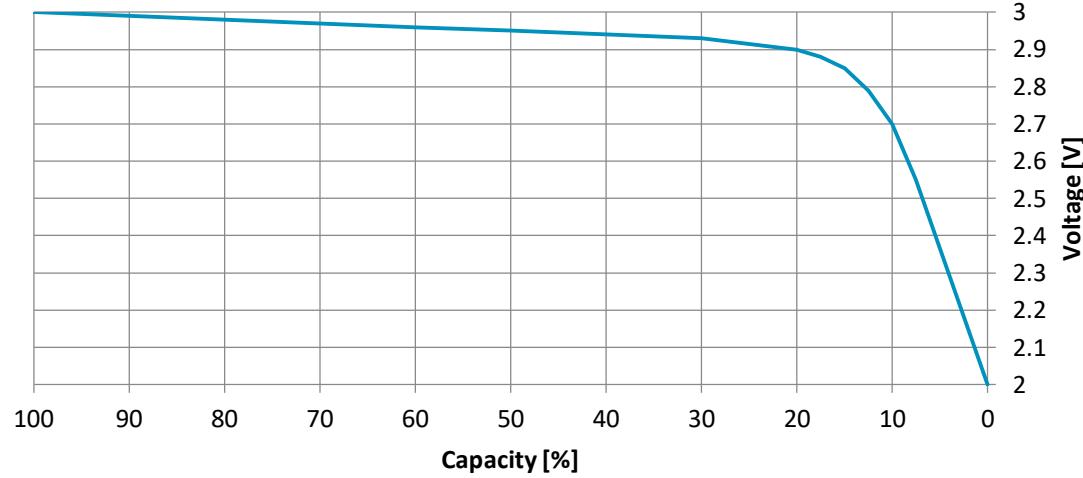
# Example: Battery Monitoring



- Operates similarly to comparator based system
- Measures battery voltage, better indication of battery life than comparator
- Can provide information about battery discharge rate

# Battery Discharge

Example Battery Discharge Curve



- As the battery discharges, the voltage decreases
- Measure with respect to  $V_{ref}$  (fixed voltage reference)
- Convert to capacity with look-up table and interpolation



<sup>†</sup>The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

[www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks)

arm

# Timer Peripherals

# Learning Objectives

At the end of this lecture, you should be able to:

- Describe the mode of operation and function of the interrupt timer.
- Explain the three modes of operation of the standard timer including ‘compare mode’, ‘capture mode’ and pulse width modulation mode.

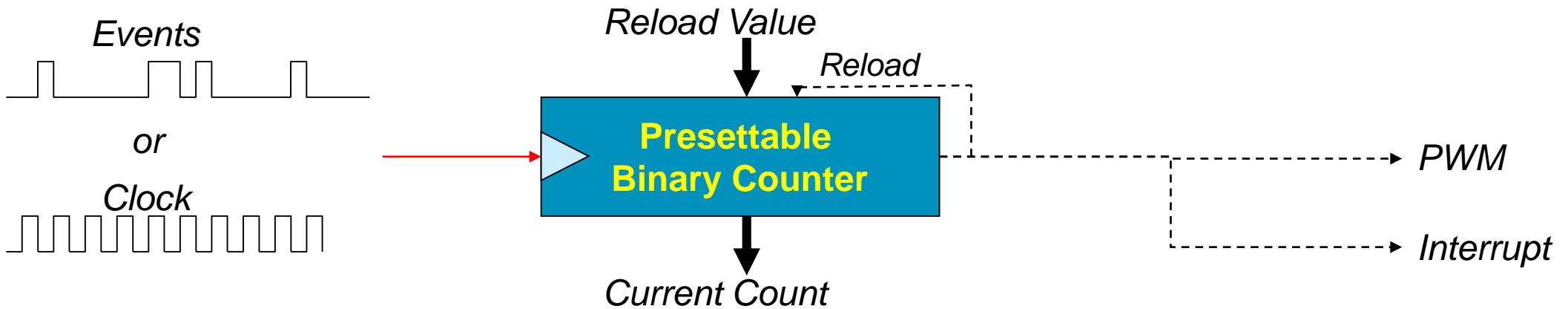
# Outline

- Types of Timer Peripherals
  - Interrupt Timer
  - PWM Module
  - Low-Power Timer
  - Real-Time Clock
  - SYSTICK

# Types of Timer Peripherals

- Interrupt Timer
  - Can generate periodically generate interrupts or trigger DMA (direct memory access) transfers
- PWM Module
  - Connected to I/O pins, has input capture and output compare support
  - Can generate PWM signals
  - Can generate interrupt requests
- Low-Power Timer
  - Can operate as timer or counter in all power modes
  - Can wake up system with interrupt
  - Can trigger hardware
- Real-Time Clock
  - Powered by external 32.768 kHz crystal
  - Tracks elapsed time (seconds) in register
  - Can set alarm
  - Can generate 1Hz output signal and/or interrupt
  - Can wake up system with interrupt
- SysTick
  - Part of CPU core's peripherals
  - Can generate periodic interrupt

# Timer/Counter Peripheral Introduction

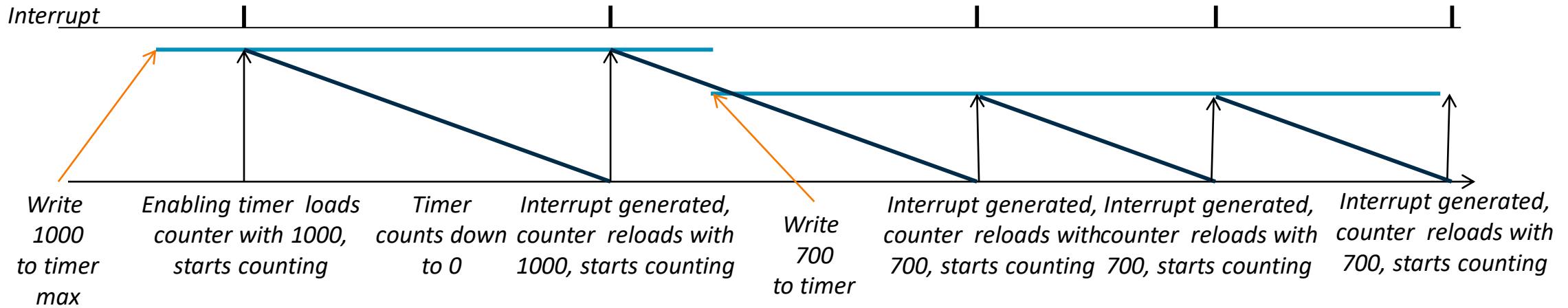


- Common peripheral for microcontrollers
- Based on presettable binary counter, enhanced with configurability
  - Count value can be read and written by MCU
  - Count direction can often be set to up or down
  - Counter's clock source can be selected
    - Counter mode: count pulses which indicate events (e.g. odometer pulses)
    - Timer mode: clock source is periodic, so counter value is proportional to elapsed time (e.g. stopwatch)
  - Counter's overflow/underflow action can be selected
    - Generate interrupt
    - Reload counter with special value and continue counting
    - Toggle hardware output signal

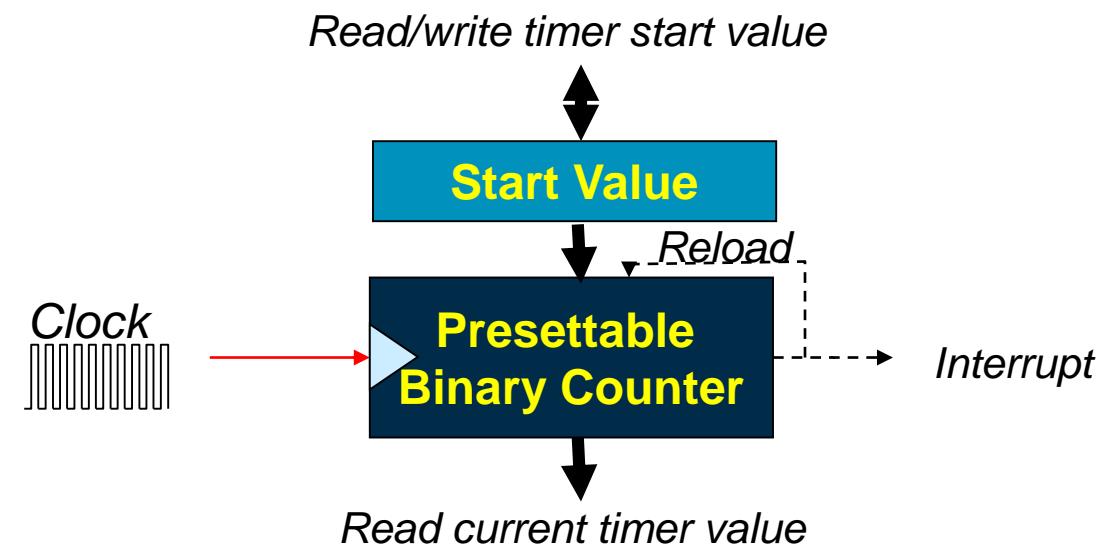
arm

# Interrupt Timer

# Interrupt Timer



- Load start value from register
- Counter counts down with each clock pulse
- When timer value reaches zero
  - Generates interrupt
  - Reloads timer with start value



# Calculating Max Value

- Goal: generate an interrupt every T seconds
- Max value =  $\text{round}(T * \text{Freq})$ 
  - Round since register is an integer, not a real number
    - Rounding provides closest integer to desired value, resulting in minimum timing error
- Example: Interrupt every 137.41ms, assuming clock frequency 24MHz
  - $137.41\text{ms} * 24\text{MHz} = 3297840$
- Example: Interrupt with a frequency of 91Hz with a 12MHz clock
  - $(1/91 \text{ Hz}) * 12\text{MHz} = \text{round}(131868.1318) = 131868$
- Use macros, interrupt 1000 times per second:
  - `CLK_FREQ / 1000`

# Configuring the Interrupt Timer

- Setup timer, set to tick at 10Hz
  - `timer_init(CLK_FREQ / 10);`
- Set interrupt
  - `timer_set_callback(timer_isr);`
- Enable module
  - `timer_enable();`
- Disable module
  - `timer_disable();`

# Example: Stopwatch

- Measure time with 100 us resolution
- Display elapsed time, updating screen every 10ms
- Controls
  - S1: toggle start/stop
- Use interrupt timer
  - Counter increment every 100 us
    - Set to timer to expire every 100 us
    - Calculate max value, e.g. at 24 MHz = round (100 us \* 24MHz -1) = 2399
  - LCD Update every 10 ms
    - Update LCD every nth ISR
    - $n = 10 \text{ ms}/100\text{us} = 100$
    - Don't update LCD in ISR! Too slow.
    - Instead set flag in ISR, poll it in main loop

arm

# Timer / PWM Module

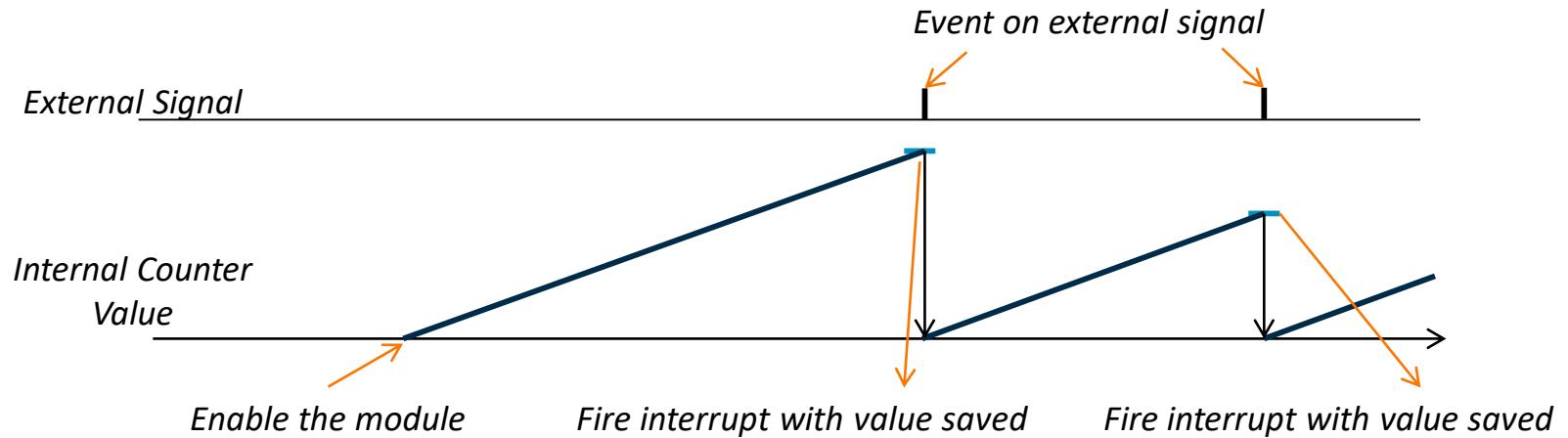
# Timer / PWM Module

- Core Counter
  - Clock options - external or internal
  - Prescaler to divide clock
  - Can reload with set value, or overflow and wrap around
- N channels
  - 3 modes
    - Capture Mode: Capture timer's value when input signal changes
    - Output Compare: Change an output signal when timer reaches certain value
    - PWM: Generate pulse-width-modulated signal. Width of pulse is proportional to specified value.
  - Possible triggering of interrupt, hardware trigger on overflow
  - One I/O pin per channel

# Major Channel Modes

- Input Capture Mode
  - Capture timer's value when input signal changes
    - Rising edge, falling edge, both
  - How long after I started the timer did the input change?
    - Measure time delay
- Output Compare Mode
  - Modify output signal when timer reaches specified value
    - Set, clear, pulse, toggle (invert)
  - Make a pulse of specified width
  - Make a pulse after specified delay
- Pulse Width Modulation
  - Make a series of pulses of specified width and frequency

# Input Capture Mode



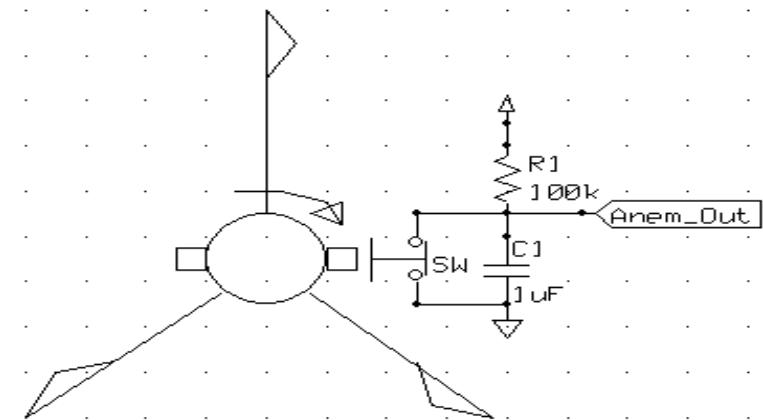
- I/O pin operates as input on edge
- When valid edge is detected on pin...
  - Current value of counter is stored
  - Interrupt is called

# Wind Speed Indicator (Anemometer)

- Rotational speed (and pulse frequency) is proportional to wind velocity
- Two measurement options:
  - Frequency (best for high speeds)
  - Width (best for low speeds)
- Can solve for wind velocity  $v$

$$v_{wind} = \frac{K * f_{clk}}{T_{anemometer}}$$

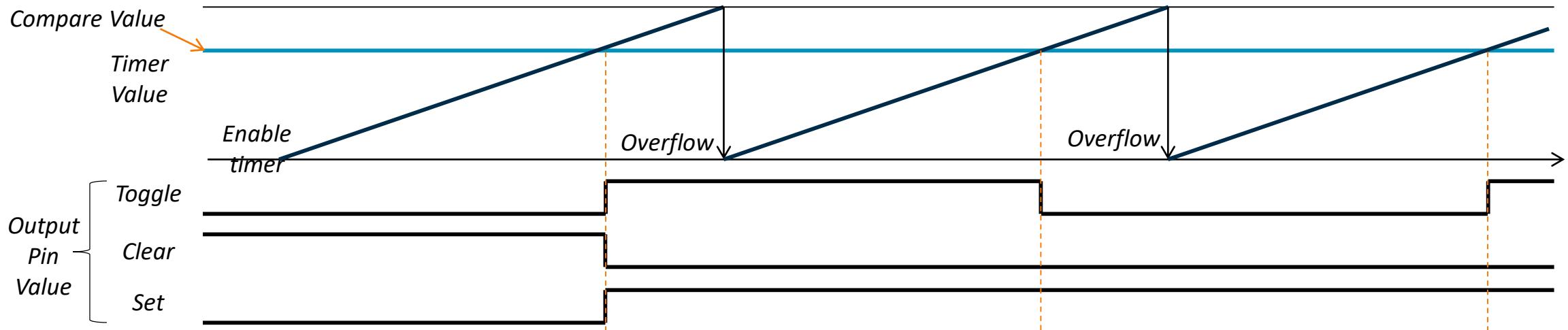
- How can we use the timer for this?
  - Use Input Capture Mode to measure period of input signal



# TPM Capture Mode for Anemometer

- Configuration
  - Set up module to count at given speed from internal clock
  - Set up channel for input capture on rising edge
- Operation: Repeat
  - First interrupt - on rising edge
    - Reconfigure channel for input capture on falling edge
    - Clear counter, start it counting
  - Second interrupt - on falling edge
    - Read capture value, save for later use in wind speed calculation
    - Reconfigure channel for input capture on rising edge
    - Clear counter, start it counting

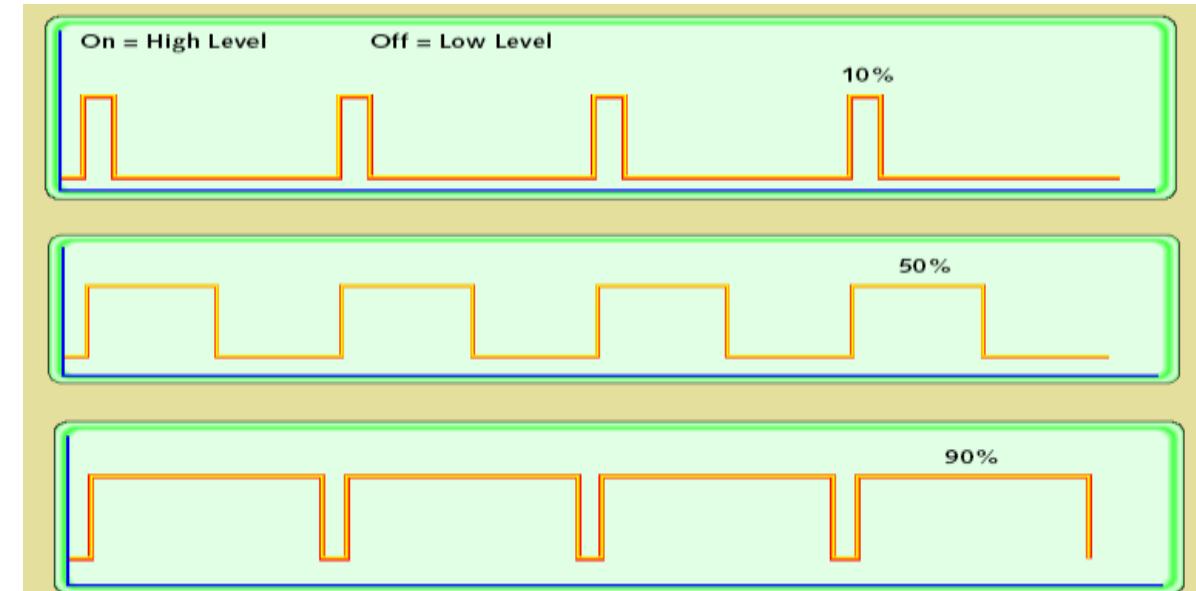
# Output Compare Mode



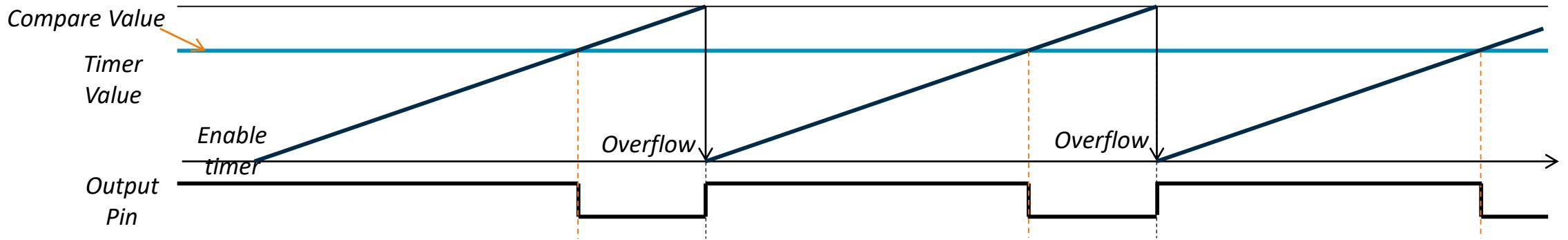
- Action on match
  - Toggle
  - Clear
  - Set
- When counter matches value ...
  - Output signal is generated
  - Interrupt is called (if enabled)

# Pulse-Width Modulation

- Uses of PWM
  - Digital power amplifiers are more efficient and less expensive than analog power amplifiers
    - Applications: motor speed control, light dimmer, switch-mode power conversion
    - Load (motor, light, etc.) responds slowly, averages PWM signal
  - Digital communication is less sensitive to noise than analog methods
    - PWM provides a digital encoding of an analog value
    - Much less vulnerable to noise
- PWM signal characteristics
  - Modulation frequency – how many pulses occur per second (fixed)
  - Period –  $1/(\text{modulation frequency})$
  - On-time – amount of time that each pulse is on (asserted)
  - Duty-cycle – on-time/period
  - Adjust on-time (hence duty cycle) to represent the analog value



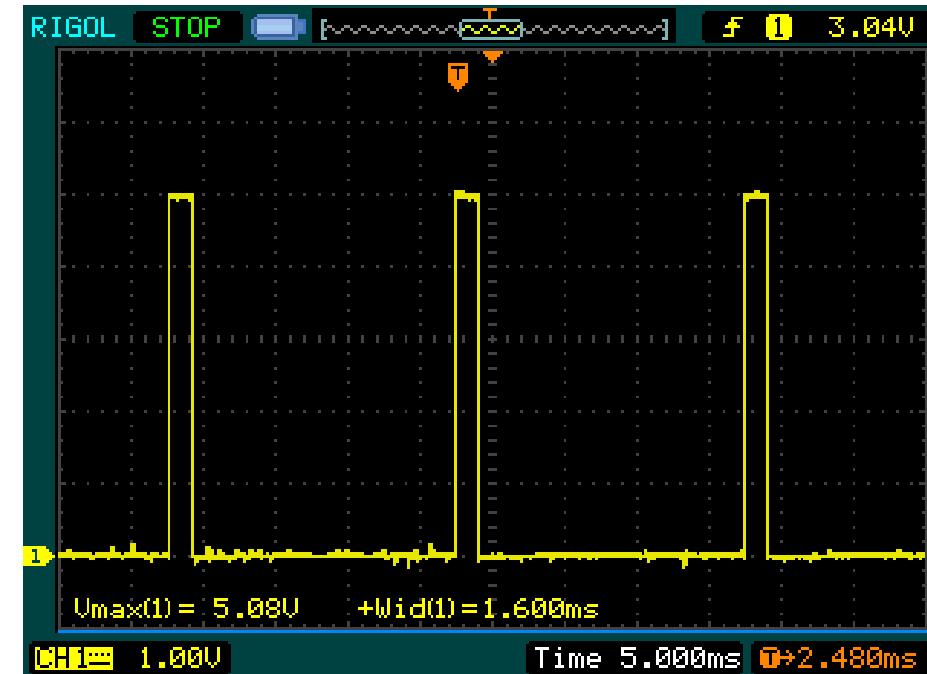
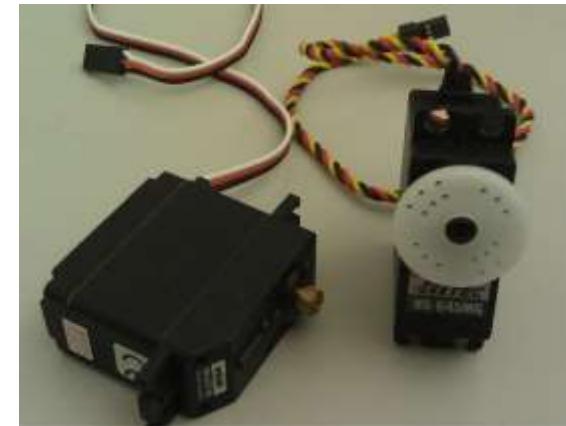
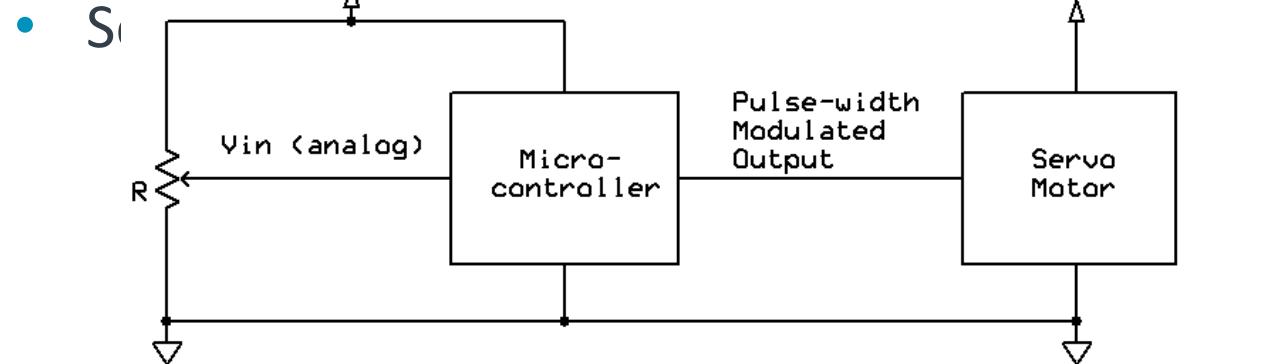
# PWM Mode



- PWM duty cycle proportional to compare value
  - Period = max timer value
  - Pulse width = compare value

$$\text{Duty Cycle} = \frac{\text{Compare Value}}{\text{Max Value}} \cdot 100\%$$

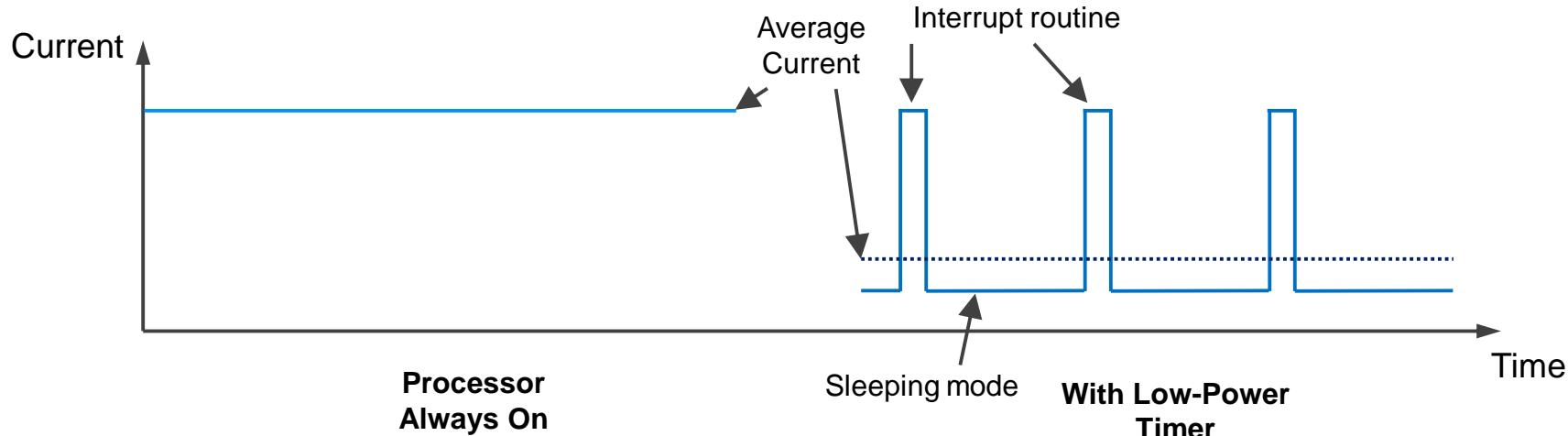
# PWM to Drive Servo Motor



arm

# Low Power Timer

# Low Power Timer Overview



- Features
  - Count time or external pulses
  - Generate interrupt when counter matches compare value
  - Interrupt wakes MCU from any low power mode
- Current draw can be reduced to microamps or even nanoamps!
- Use the WFI instruction (Wait For Instruction)
  - Puts CPU in low power mode until interrupt request



<sup>†</sup>The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

[www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks)



# Serial Communications

# Learning Objectives

At the end of this lecture, you should be able to:

- Outline the concepts of both serial and parallel communication and give examples of their applications.
- Explain the difference between synchronous and asynchronous serial communication.
- Compare serial and parallel communication including their advantages and disadvantages.
- Outline the difference between synchronous half-duplex and full-duplex serial buses
- Describe UART and its communication protocol.
- Describe SPI communication protocol.
- Describe I2C communication protocol.

# Overview

- Serial communications
  - Concepts
  - Tools
  - Software: polling, interrupts, and buffering
- Protocols:
  - UART
  - SPI
  - I2C

# Why Communicate Serially?

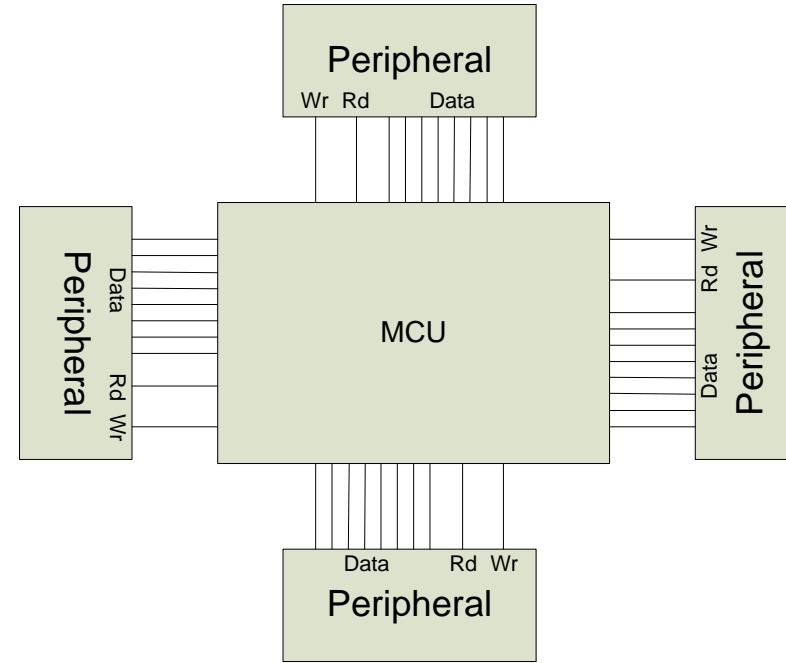
- Native word size is multi-bit (8, 16, 32, etc.)
- Often it's not feasible to support sending all the word's bits at the same time
  - Cost and weight: more wires needed, larger connectors needed
  - Mechanical reliability: more wires => more connector contacts to fail
  - Timing Complexity: some bits may arrive later than others due to variations in capacitance and resistance across conductors
  - Circuit complexity and power: may not want to have 16 different radio transmitters + receivers in the system

# Example System: Voyager Spacecraft



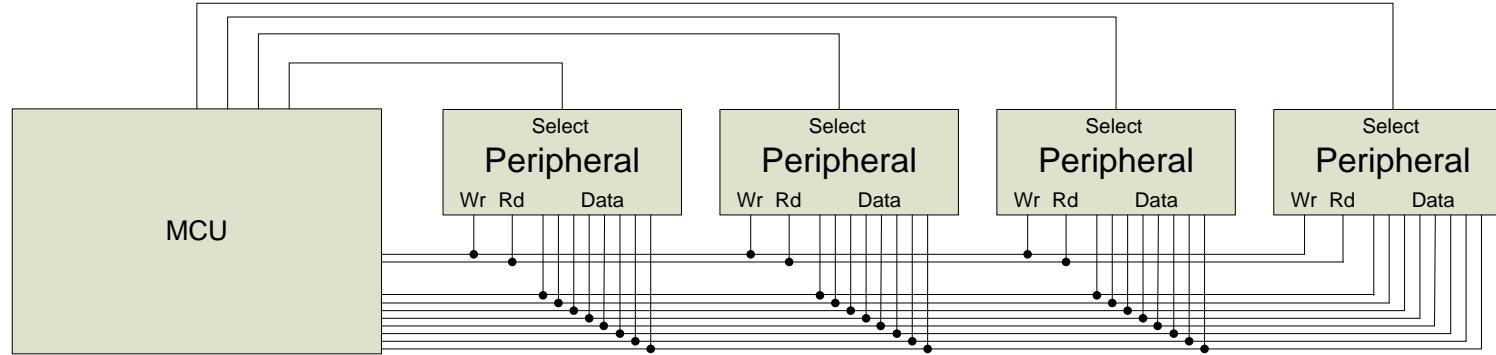
- Launched in 1977
- Constraints: Reliability, power, size, weight, reliability, reliability, etc.
- “Uplink communications are via S-band (16-bits/sec command rate) while an X-band transmitter provides downlink telemetry at 160 bits/sec normally and 1.4kbps for playback of high-rate plasma wave data. All data are transmitted from and received at the spacecraft via the 3.7 meters high-gain antenna (HGA).”  
<http://voyager.jpl.nasa.gov/spacecraft/index.html>
  - Uplink – to spacecraft
  - Downlink – from spacecraft

# Example System



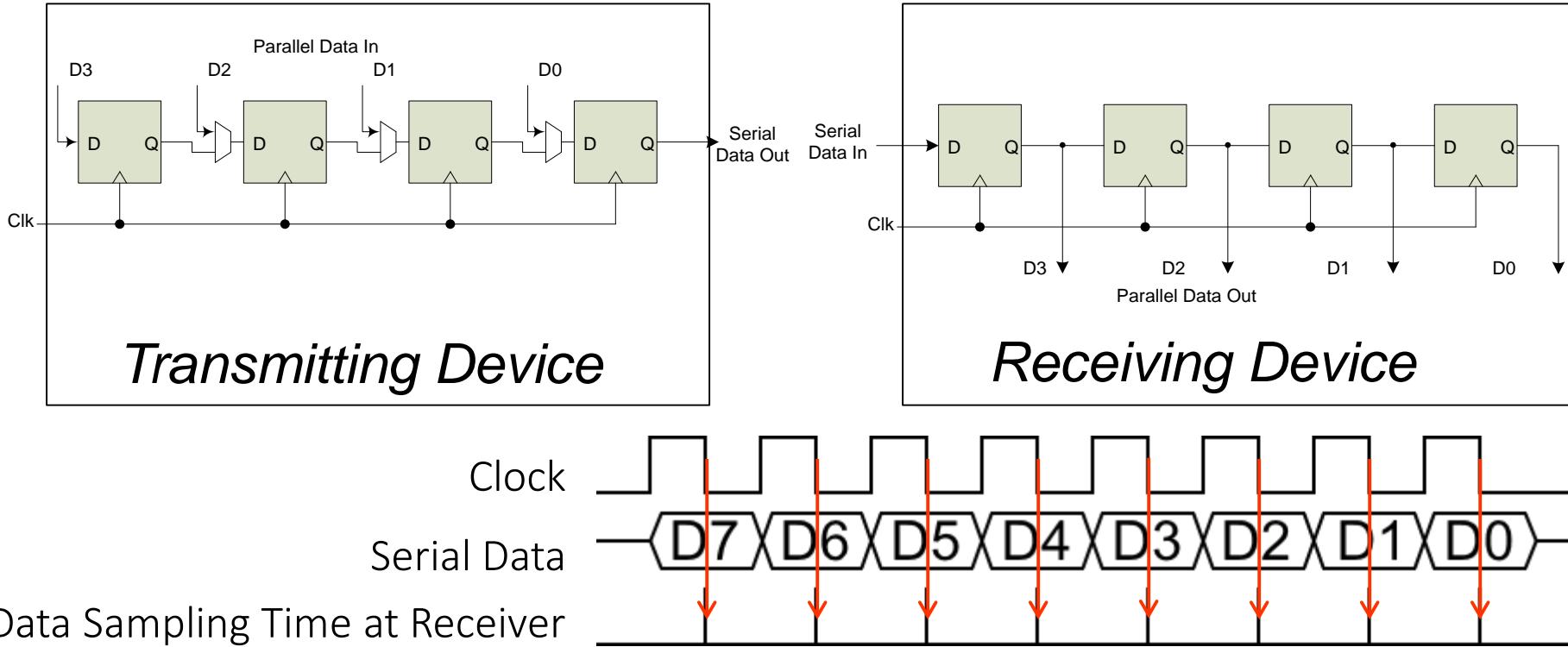
- Dedicated point-to-point connections
  - Parallel data lines, read and write lines between MCU and each peripheral
- Fast, allows simultaneous transfers
- Requires many connections, PCB area, scales badly

# Parallel Buses



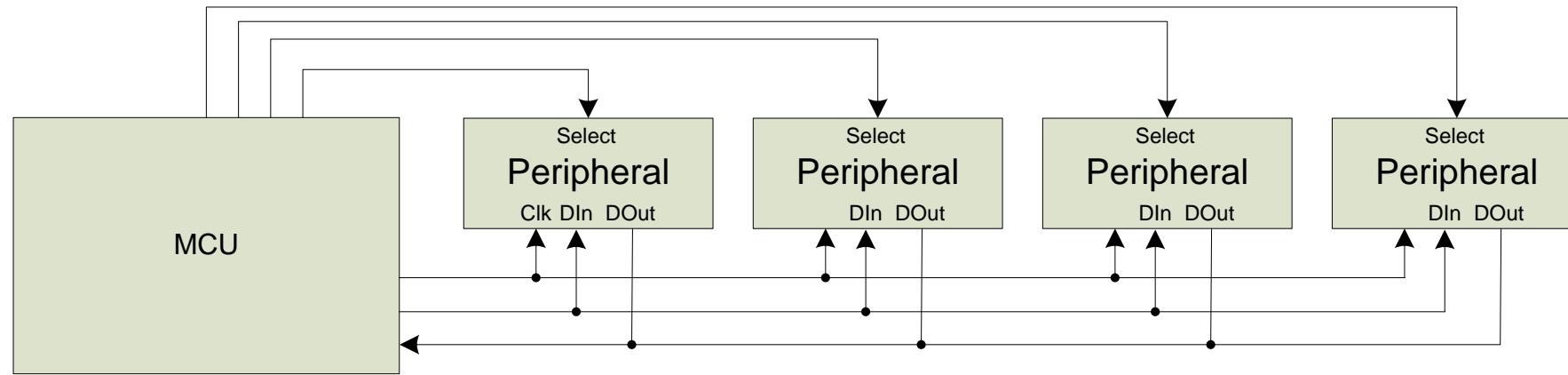
- All devices use buses to share data, read, and write signals
- MCU uses individual select lines to address each peripheral
- MCU requires fewer pins for data, but still one per data bit
- MCU can communicate with only one peripheral at a time

# Synchronous Serial Data Transmission



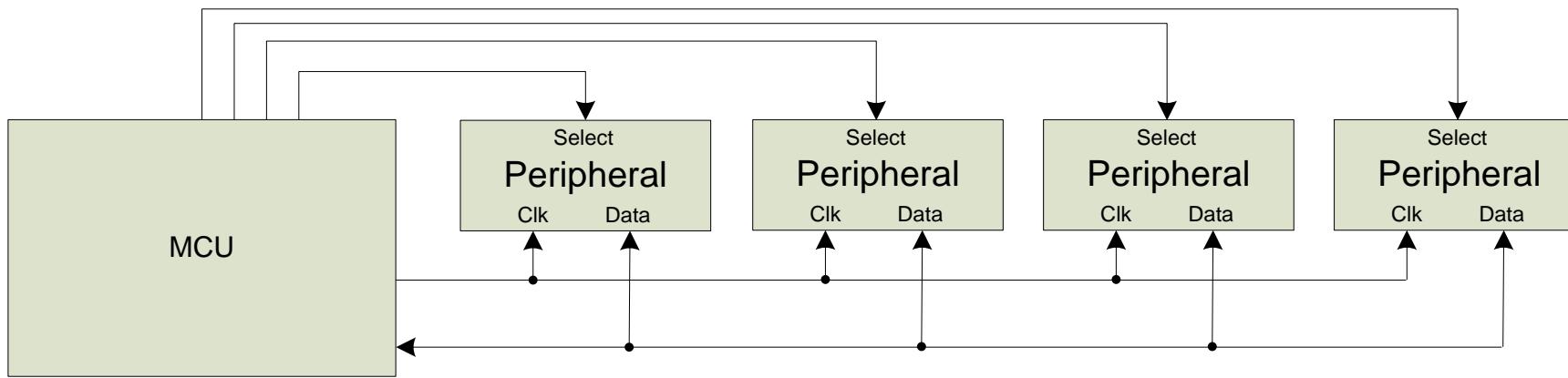
- Use shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal is along with the data signal

# Synchronous Full-Duplex Serial Data Bus



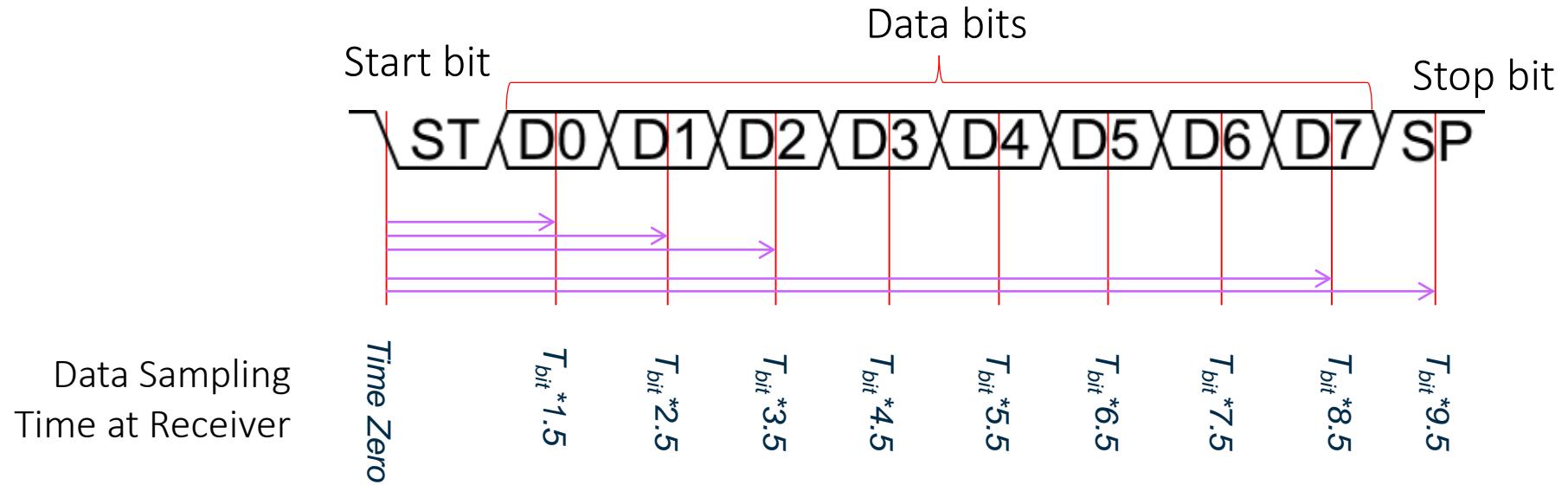
- Now can use two serial data lines - one for reading, one for writing.
  - Allows simultaneous send and receive full-duplex communication

# Synchronous Half-Duplex Serial Data Bus



- Share the serial data line
- Doesn't allow simultaneous send and receive - is half-duplex communication

# Asynchronous Serial Communication



- Eliminate the clock line!
- Transmitter and receiver must generate clock locally
- Transmitter must add start bit (always same value) to indicate start of each data frame
- Receiver detects leading edge of start bit, then uses it as a timing reference for sampling data line to extract each data bit N at time  $T_{bit} * (N+1.5)$
- Stop bit is also used to detect some timing errors

# Serial Communication Specifics

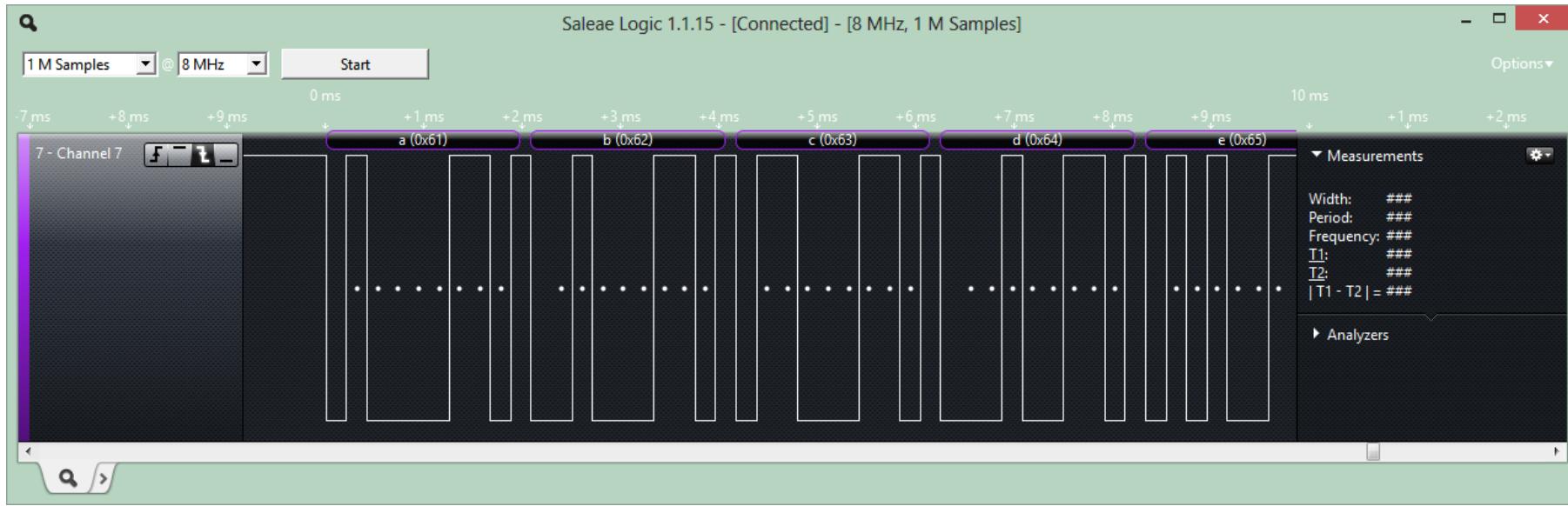
- Data frame fields
  - Start bit (one bit)
  - Data (LSB first or MSB, and size – 7, 8, 9 bits)
  - Optional parity bit is used to make total number of ones in data even or odd
  - Stop bit (one or two bits)
- All devices must use the same communications parameters
  - E.g. communication speed (300 baud, 600, 1200, 2400, 9600, 14400, 19200, etc.)
- Sophisticated network protocols have more information in each data frame
  - Medium access control – when multiple nodes are on bus, they must arbitrate for permission to transmit
  - Addressing information – for which node is this message intended?
  - Larger data payload
  - Stronger error detection or error correction information
  - Request for immediate response (“in-frame”)



# Error Detection

- Can send additional information to verify data was received correctly
- Need to specify which parity to expect: even, odd or none.
- Parity bit is set so that total number of “1” bits in data and parity is even (for even parity) or odd (for odd parity)
  - 01110111 has 6 “1” bits, so parity bit will be 1 for odd parity, 0 for even parity
  - 01100111 has 5 “1” bits, so parity bit will be 0 for odd parity, 1 for even parity
- Single parity bit detects if 1, 3, 5, 7 or 9 bits are corrupted, but doesn’t detect an even number of corrupted bits
- Stronger error detection codes (e.g. Cyclic Redundancy Check) exist and use multiple bits (e.g. 8, 16), and can detect many more corruptions.
  - Used for CAN, USB, Ethernet, Bluetooth, etc.

# Tools for Serial Communications Development



- Tedious and slow to debug serial protocols with just an oscilloscope
- Instead use a logic analyzer to decode bus traffic
- Worth its weight in gold!
- Saelae 8-Channel Logic Analyzer
  - \$150 ([www.saleae.com](http://www.saleae.com))
  - Plugs into PC's USB port
  - Decodes SPI, asynchronous serial, I2C, 1-Wire, CAN, etc.
- Build your own: with Logic Sniffer or related open-source project



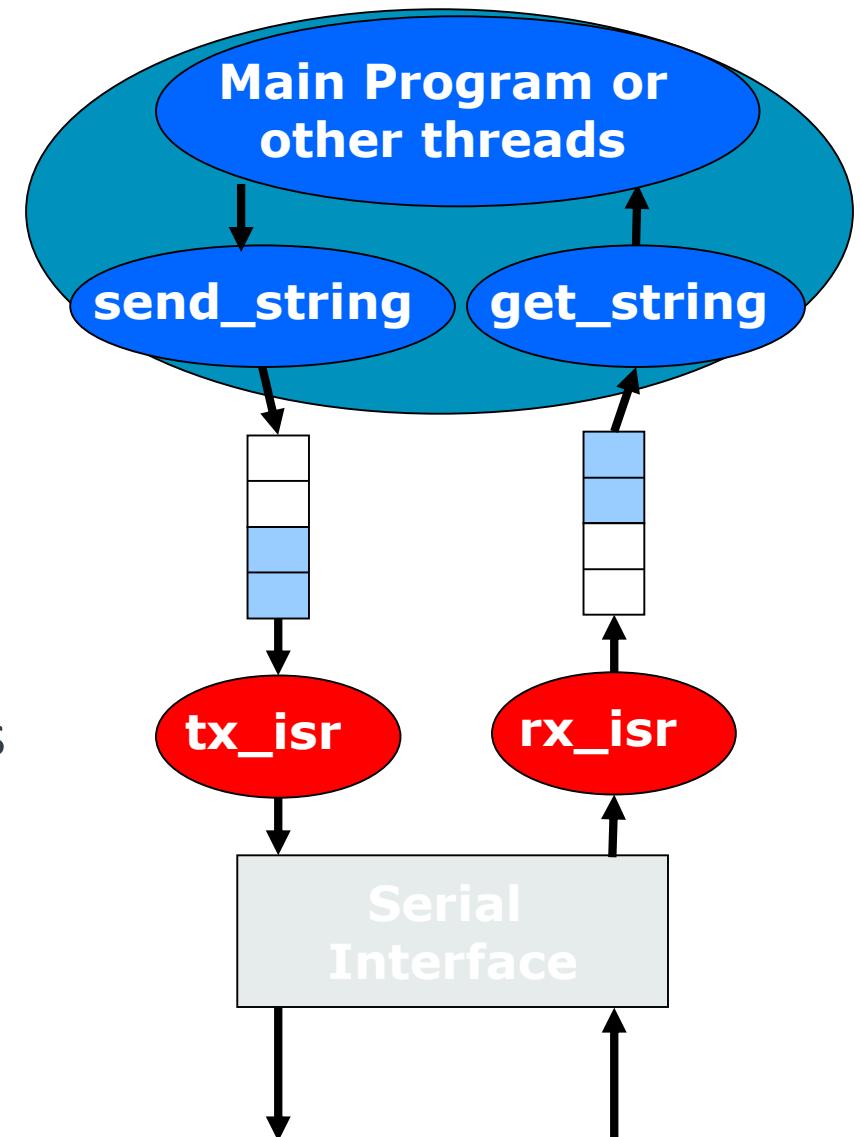
# Software Structure – Handling asynchronous Communication

# Software Structure

- Communication is *asynchronous* to program
  - Don't know what code the program will be executing ...
    - when the next item arrives
    - when current outgoing item completes transmission
    - when an error occurs
  - Need to synchronize between program and serial communication interface somehow
- Options
  - Polling
    - Wait until data is available
    - Simple but inefficient of processor time
  - Interrupt
    - CPU interrupts program when data is available
    - Efficient, but more complex

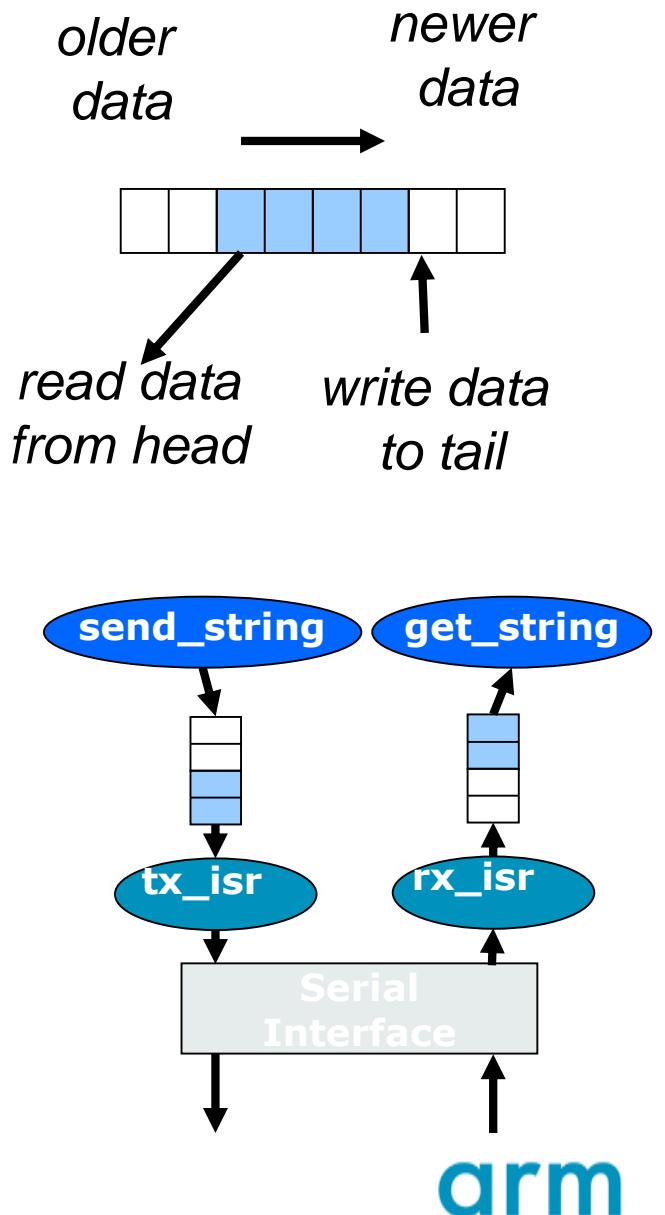
# Serial Communications and Interrupts

- Want to provide multiple threads of control in the program
  - Main program (and subroutines it calls)
  - Transmit ISR – executes when serial interface is ready to send another character
  - Receive ISR – executes when serial interface receives a character
  - Error ISR(s) – execute if an error occurs
- Need a way of buffering information between threads
  - Solution: circular queue with head and tail pointers
  - One for tx, one for rx



# Code to Implement Queues

- Enqueue at tail: tail is the index of the next free entry
- Dequeue from head: head is the index of the item to remove
- Queue size is initialized and stored in size
- One queue per direction
  - tx ISR unloads tx\_q
  - rx ISR loads rx\_q
- Other threads (e.g. main) load tx\_q and unload rx\_q
- Need to wrap pointer at end of buffer to make it circular,
  - Use % (modulus, remainder) operator if queue size is not power of two
  - Use & (bitwise and) if queue size is a power of two
- Queue is empty if tail == head
- Queue is full if  $(tail + 1) \% size == head$



# Defining the Queues

```
typedef struct {
 uint8_t *data; //!< Array of data, stored on the heap.
 uint32_t head; //!< Index in the array of the oldest element.
 uint32_t tail; //!< Index in the array of the youngest element.
 uint32_t size; //!< Size of the data array.
} Queue;
```

# Initialization and Status Inquiries

```
int queue_init(Queue *queue, uint32_t size) {
 queue->data = (uint8_t*)malloc(sizeof(uint8_t) * size);
 queue->head = 0;
 queue->tail = 0;
 queue->size = size;

 // If malloc returns NULL (0) the allocation has failed.
 return queue->data != 0;
}

int queue_is_full(Queue *queue) {
 return ((queue->tail + 1) % queue->size) == queue->head;
}

int queue_is_empty(Queue *queue) {
 return queue->tail == queue->head;
}
```

# Enqueue and Dequeue

```
int queue_enqueue(Queue *queue, uint8_t item) {
 if (!queue_is_full(queue)) {
 queue->data[queue->tail++] = item;
 queue->tail %= queue->size;
 return 1;
 } else {
 return 0;
 }
}

int queue_dequeue(Queue *queue, uint8_t *item) {
 if (!queue_is_empty(queue)) {
 *item = queue->data[queue->head++];
 queue->head %= queue->size;
 return 1;
 } else {
 return 0;
 }
}
```

# Using the Queues

- Sending data:

```
queue_enqueue(..., c)
```

- Receiving data:

```
queue_dequeue(..., &c)
```



# Software Structure – Parsing Messages

# Decoding Messages

- Two types of messages
  - Actual binary data sent
    - First identify message type
    - Second, based on this message type, copy binary data from message fields into variables
      - May need to use pointers and casting to get code to translate formats correctly and safely
  - ASCII text characters representing data sent
    - First identify message type
    - Second, based on this message type, translate (parse) the data from the ASCII message format into a binary format
    - Third, copy the binary data into variables

# Example Binary Serial Data: TSIP

```
switch (id) {
 case 0x84:
 lat = *((double *)(&msg[0]));
 lon = *((double *)(&msg[8]));
 alt = *((double *)(&msg[16]));
 clb = *((double *)(&msg[24]));
 tof = *((float *)(&msg[32]));
 break;
 case 0x4A: ...

 default:
 break;
}
```

Report Packet (0x84) Data Structure

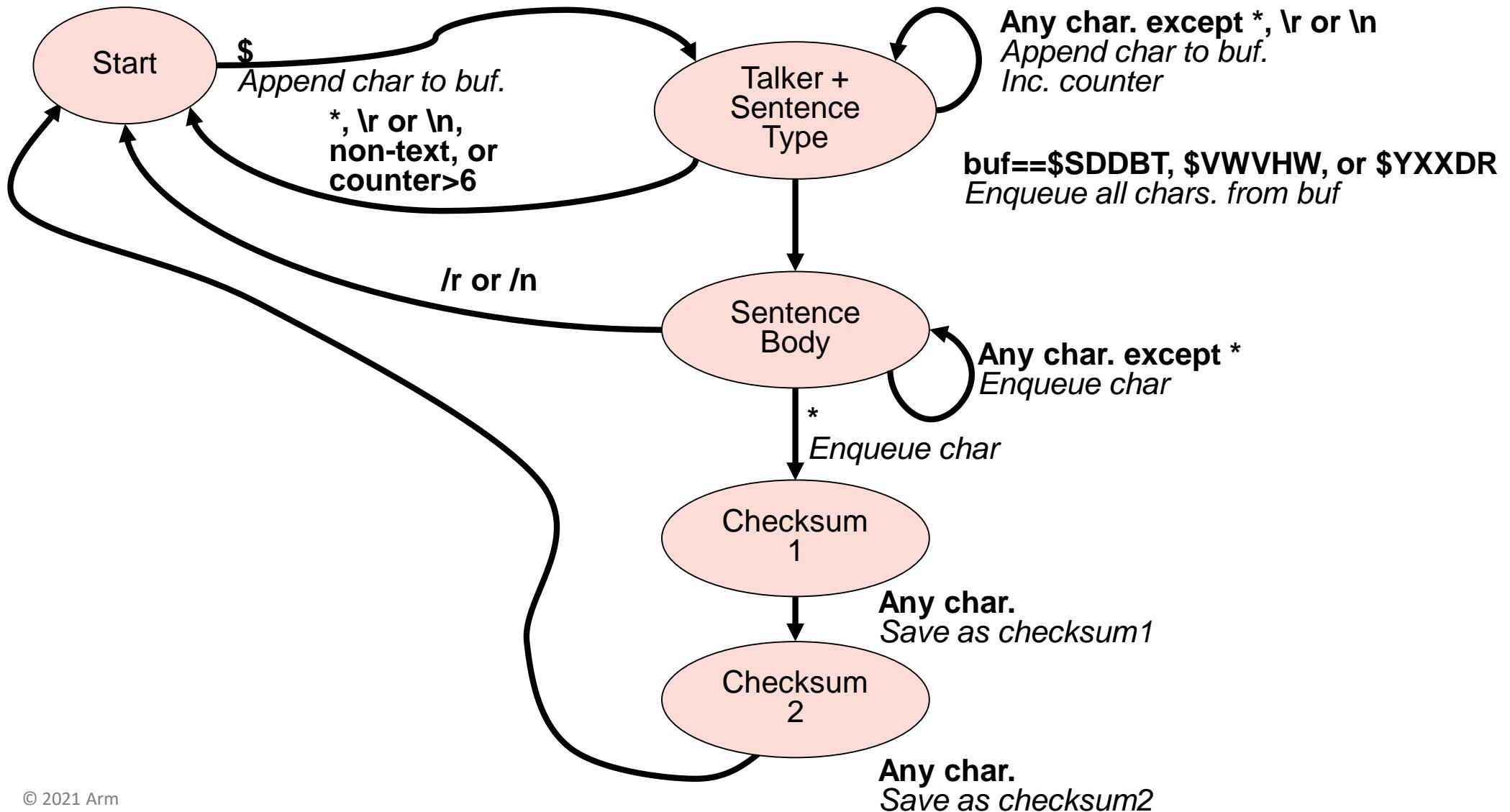
| Type   | sizeof(Type) | Item        | Units                             |
|--------|--------------|-------------|-----------------------------------|
| Double | 8            | Latitude    | Radians; + for north, - for south |
| Double | 8            | Longitude   | Radians; + for east, - for west   |
| Double | 8            | Altitude    | Meters                            |
| Double | 8            | Clock Bias  | Meters                            |
| Single | 4            | Time-of-fix | Seconds                           |

# Example ASCII Serial Data: NMEA-0183

\$IDMSG,D1,D2,D3,D4,...,Dn\*CS\r\n

- \$ denotes the start of a message
- ID is a two letter mnemonic to describe the source of data, e.g. GP signifies GPS
- MSG is a three letter mnemonic to describe the message content.
- Commas are used to delaminate the data fields.
- Dn represents each of the data fields.
- \* is used to separate the data from the checksum.
- CS contains two ASCII characters representing the hex value of the checksum.
- \r\n is the carriage return character followed by the new line character to denote the end of a message.

# State Machine for Parsing NMEA-0183



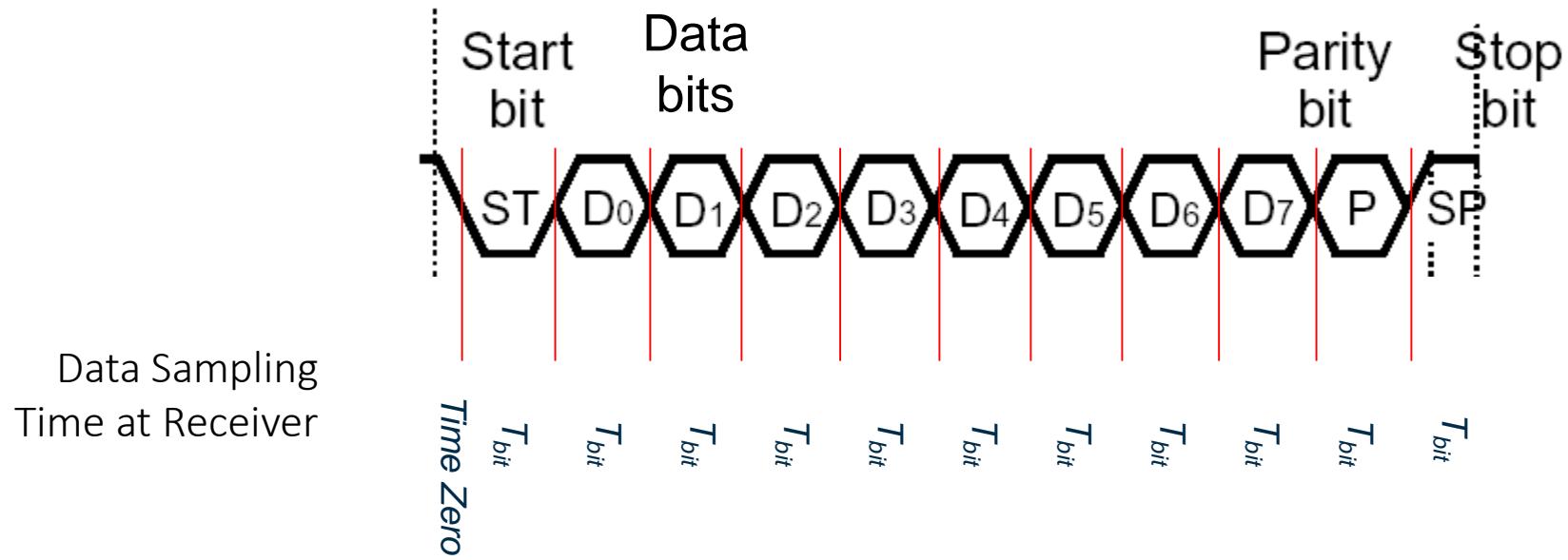
# Parsing

```
switch (parser_state) {
 case TALKER_SENTENCE_TYPE:
 switch (msg[i]) {
 '*' :
 '\r' :
 '\n' :
 parser_state = START;
 break;
 default:
 if (Is_Not_Character(msg[i]) || n>6) {
 parser_state = START;
 } else {
 buf[n++] = msg[i];
 }
 break;
 }
 if ((n==6) & ...){
 parser_state = SENTENCE_BODY;
 }
 break;
 case SENTENCE_BODY:
 break;
```



# Asynchronous serial (UART) Communications

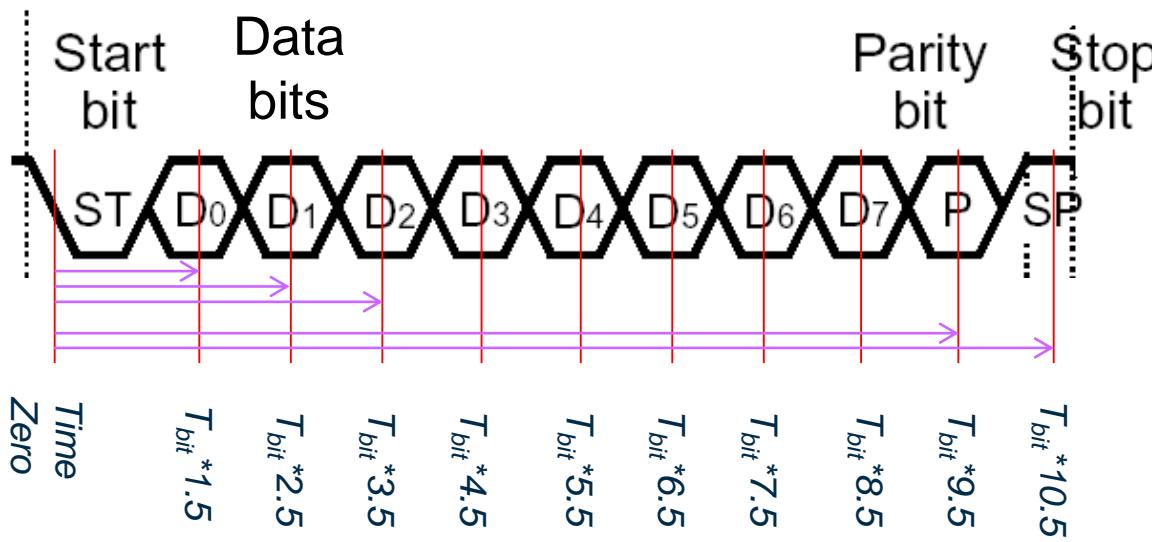
# Transmitter Basics



- If no data to send, keep sending 1 (stop bit) – idle line
- When there is a data word to send
  - Send a 0 (start bit) to indicate the start of a word
  - Send each data bit in the word (use a shift register for the transmit buffer)
  - Send a 1 (stop bit) to indicate the end of the word

# Receiver Basics

Data Sampling  
Time at  
Receiver

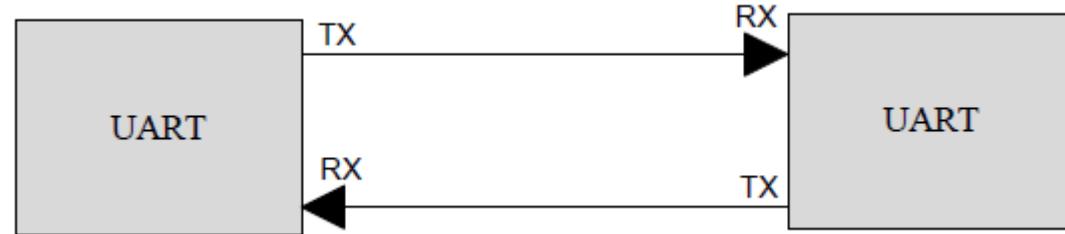


- Wait for a falling edge (beginning of a Start bit)
  - Then wait  $\frac{1}{2}$  bit time
  - Do the following for as many data bits in the word
    - Wait 1 bit time
    - Read the data bit and shift it into a receive buffer (shift register)
  - Wait 1 bit time
  - Read the bit
    - if 1 (Stop bit), then OK
    - if 0, there's a problem!

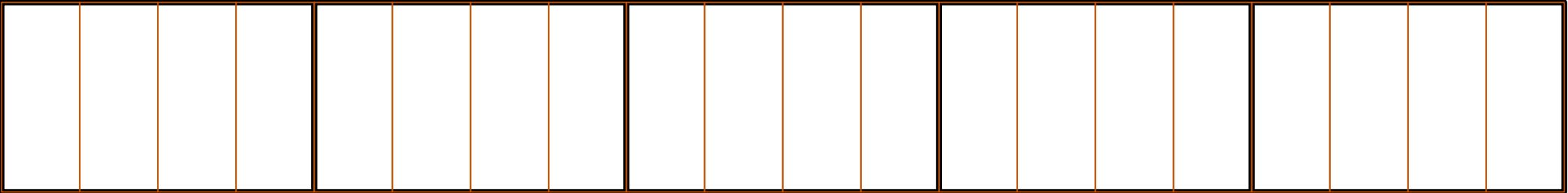
# For this to work...

- Transmitter and receiver must agree on several things (protocol)

- Order of data bits
- Number of data bits
- What a start bit is (1 or 0)
- What a stop bit is (1 or 0)
- How long a bit lasts
  - Transmitter and receiver clocks must be reasonably close, since the only timing reference is the start of the start bit



# Input Data Oversampling



- When receiving, UART *oversamples* incoming data line
  - Extra samples allow voting, improving noise immunity
  - Better synchronization to incoming data, improving noise immunity
- Configurable oversampling from 8x to 32x
  - Put desired oversampling factor minus one into SCB\_CTRL, SCB\_OVS bits.

# Baud Rate

- Need to divide high frequency clock down to desired baud rate \* oversampling factor
- Example
  - 24MHz -> 4800 baud with 16x oversampling
  - Division factor =  $24E6/(4800*16) = 312.5$ . Must round to closest integer value ( 312 or 313), will have a slight frequency error.

# Using the UART

- When can we transmit?
  - Transmit peripheral must be ready for data
  - Can poll the status register
  - Or we can use an interrupt, in which case we will need to queue up data
- When can we receive a byte?
  - Receive peripheral must have data
  - Can poll the status register
  - Or we can use an interrupt, and again we will need to queue the data

# Software for Polled Serial Comm.

```
void test_polled() {
 uart_init(9600);
 uart_enable();

 while(1) {
 uart_tx(uart_rx()); // echoes the received character back
 }
}
```

# Example Receiver: Display Data on LCD

```
line = col = 0;
while (1) {
 c = uart_rx();
 lcd_set_cursor(col, line);
 lcd_put_char(c);
 col++;
 if (col > 7) {
 col = 0;
 line++;
 if (line > 1) {
 line = 0;
 }
 }
}
```

# Software for Interrupt-Driven Serial Comm.

- Use interrupts
- First, initialize peripheral to generate interrupts
- Second, create single ISR with for the receive callback
- Third, enable the peripheral

# Interrupt Handler

```
Queue rx_queue;

void uart_rx_isr(uint8_t rx) {
 // Store the received character
 queue_enqueue(&rx_queue, rx);
}

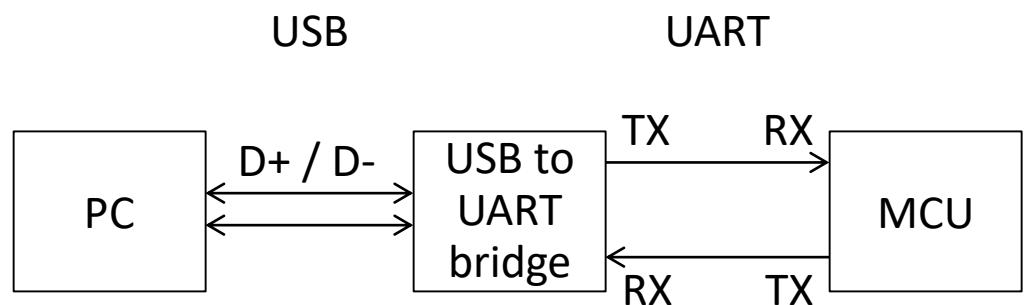
int main() {
 queue_init(&rx_queue, 128);
 uart_init(9600);
 uart_set_rx_callback(uart_rx_isr);
 uart_enable();
 ...
}
```

# USB to UART Interface

- PCs haven't had external asynchronous serial interfaces for a while, so how do we communicate with a UART?

- USB to UART interface

- USB connection to PC
  - Logic level (0-3.3V) to microcontroller's UART (not RS232 voltage levels)
  - USB01A USB to serial adapter
    - <http://www.pololu.com/catalog/product/391>
    - Can also supply 5V, 3.3V from USB



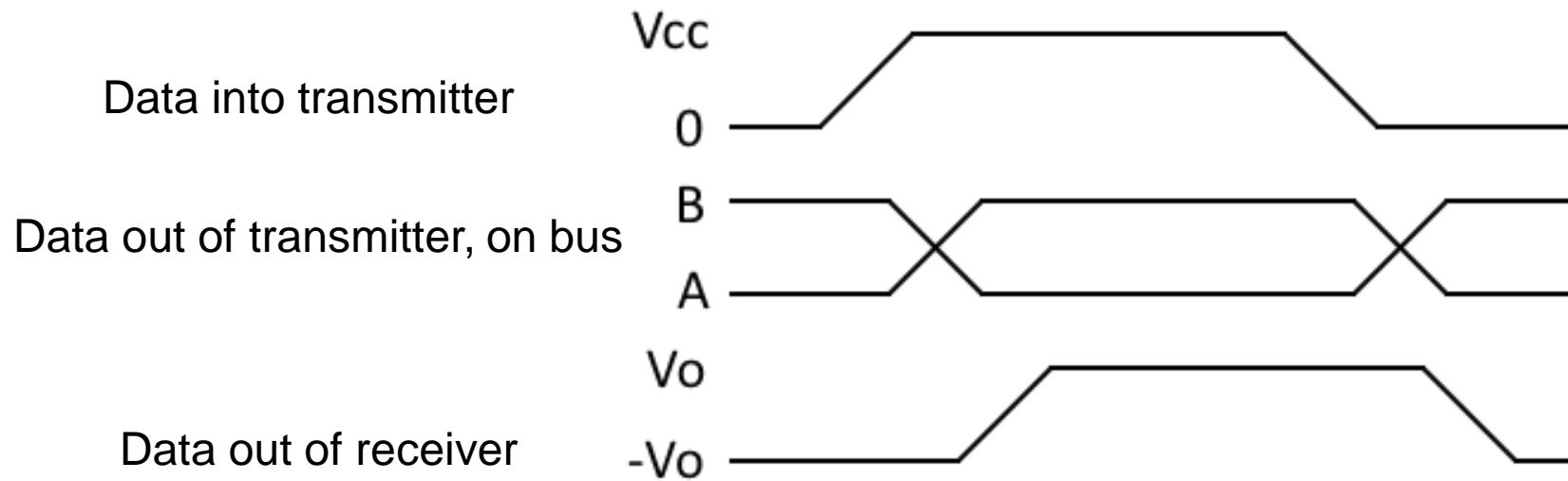
# Building on Asynchronous Comm.

- Problem #1
  - Logic-level signals (0 to 1.65V, 1.65V to 3.3V) are sensitive to noise and signal degradation
- Problem #2
  - Point-to-point topology does not support a large number of nodes well
    - Need a dedicated wire to send information from one device to another
    - Need a UART channel for each device the MCU needs to talk to
    - Single transmitter, single receiver per data wire

# Solution to Noise: Higher Voltages

- Use higher voltages to improve noise margin:  
+3 V to +15 V, -3 V to -15 V
- Example IC (Maxim MAX3232) uses charge pumps to generate higher voltages from 3.3V supply rail

# Solution to Noise: Differential Signaling



- Use differential signaling
  - Send two signals: Buffered data (A), buffered complement of data (B)
  - Receiver compares the two signals to determine if data is a one ( $A > B$ ) or a zero ( $B > A$ )

# Solutions to Poor Scaling

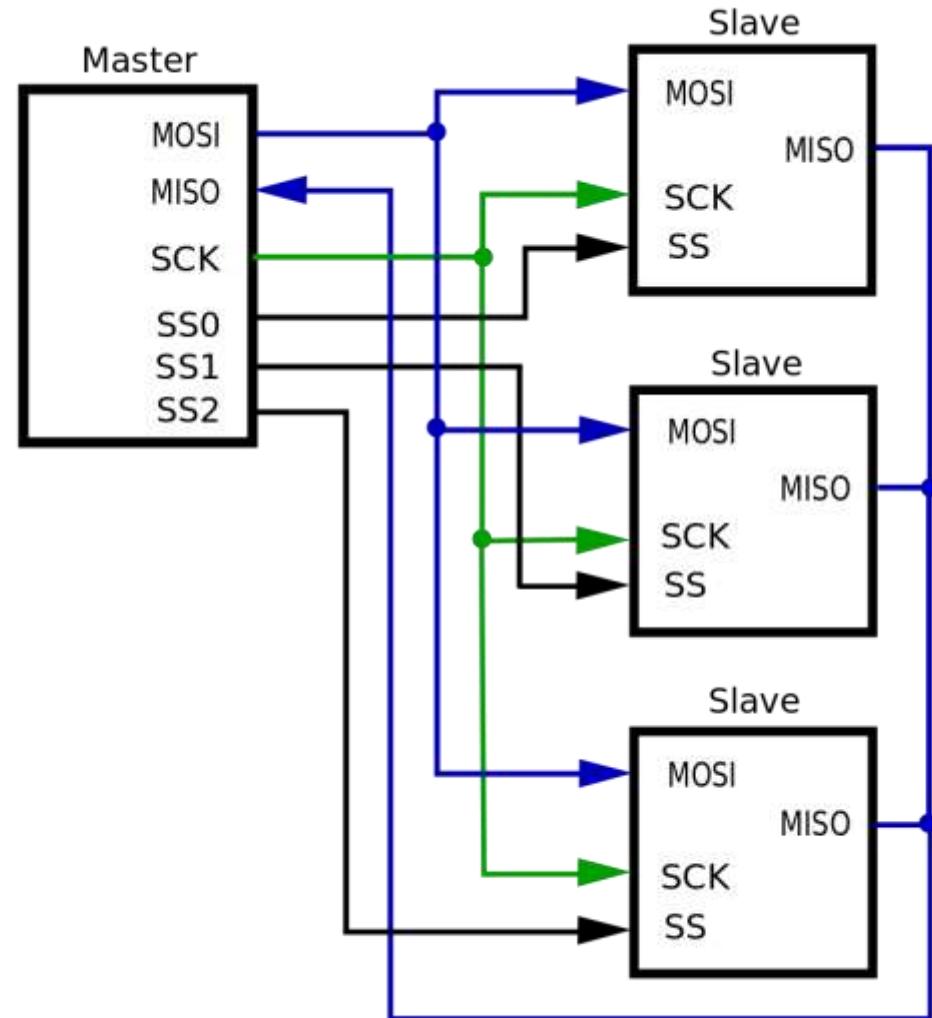
- Approaches
  - Allow one transmitter to drive multiple receivers (multi-drop)
  - Connect all transmitters and all receivers to same data line (multi-point network). Need to add a medium access control technique so all nodes can share the wire
- Example Protocols
  - RS-232: higher voltages, point-to-point
  - RS-422: higher voltages, differential data transmission, multi-drop
  - RS-485: higher voltages, multi-point



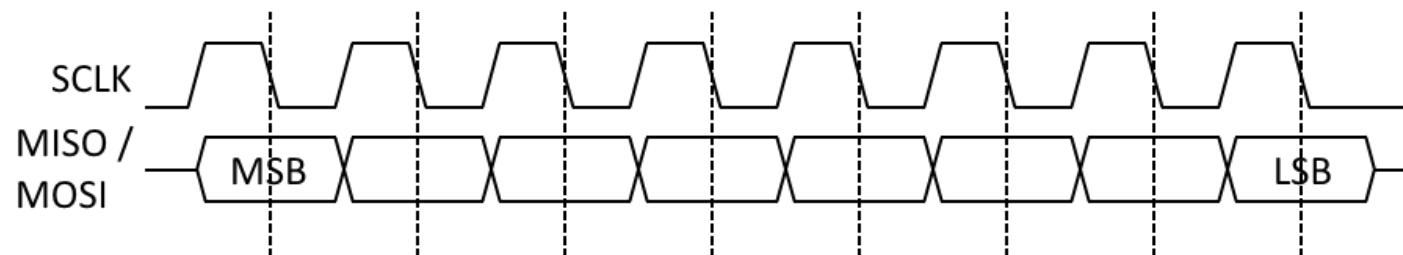
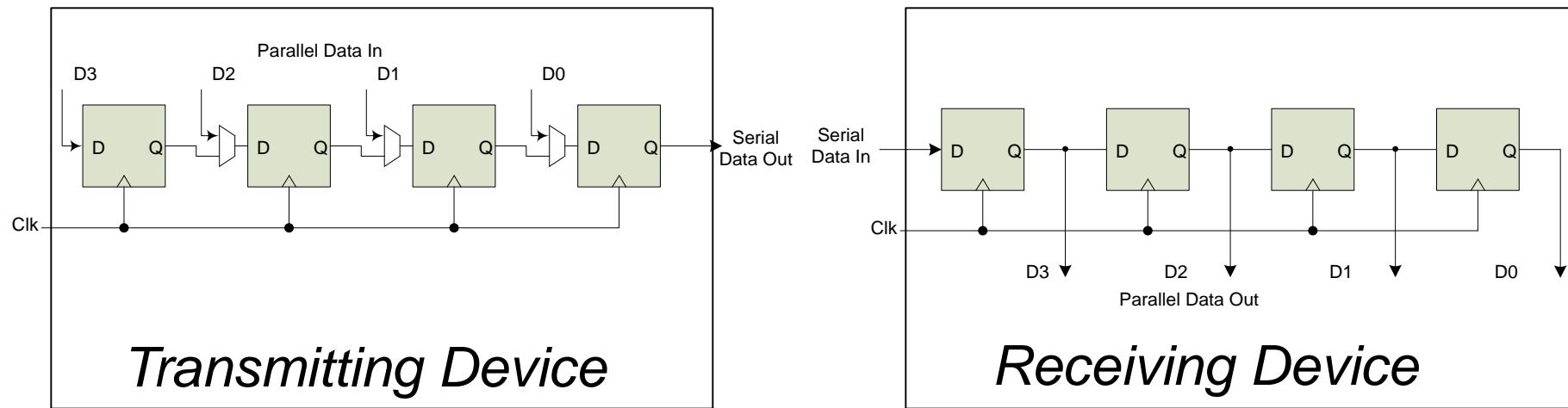
# SPI Communications

# Hardware Architecture

- All chips share bus signals
  - Clock SCK
  - Data lines MOSI (master out, slave in) and MISO (master in, slave out)
- Each peripheral has its own chip select line (CS)
  - Master (MCU) asserts the CS line of only the peripheral it's communicating with



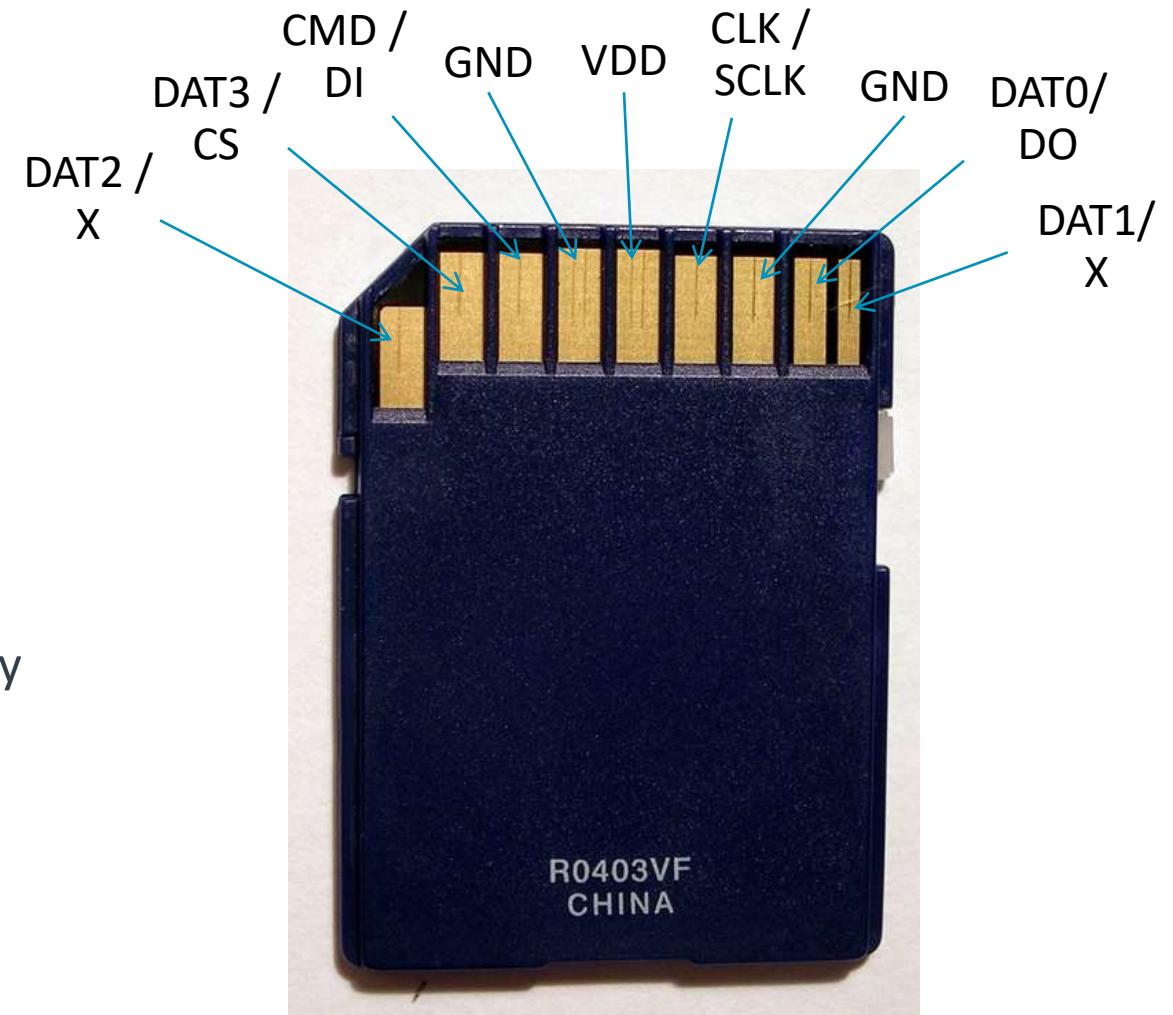
# Serial Data Transmission



- Use shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal is along with the data signal

# SPI Example: Secure Digital Card Access

- SD cards have two communication modes
    - Native 4-bit
    - Legacy SPI 1-bit
  - VDD from 2.7 V to 3.6 V
  - CS: Chip Select (active-LOW)
- 
- Host sends a six-byte command packet to card
    - Index, argument, CRC
  - Host reads bytes from card until card signals it is ready
    - Card returns
      - 0xff while busy
      - 0x00 when ready without errors
      - 0x01-0x7f when error has occurred

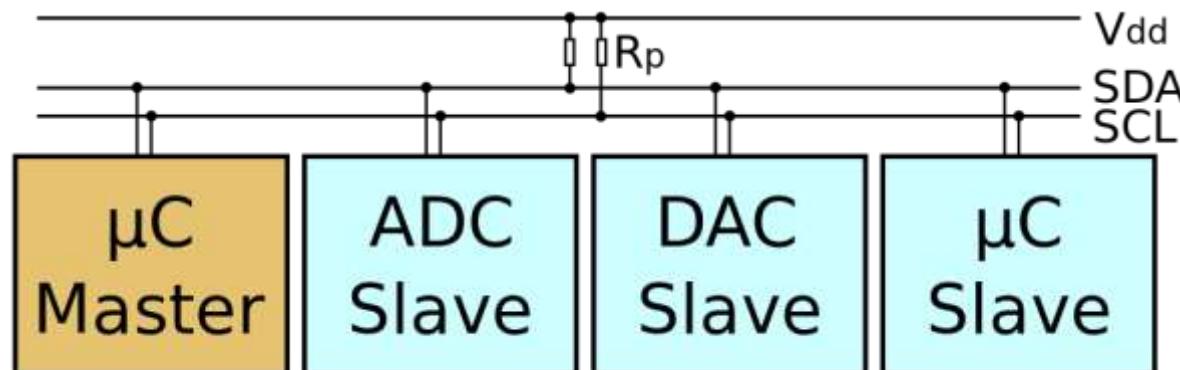




# I2C Communications

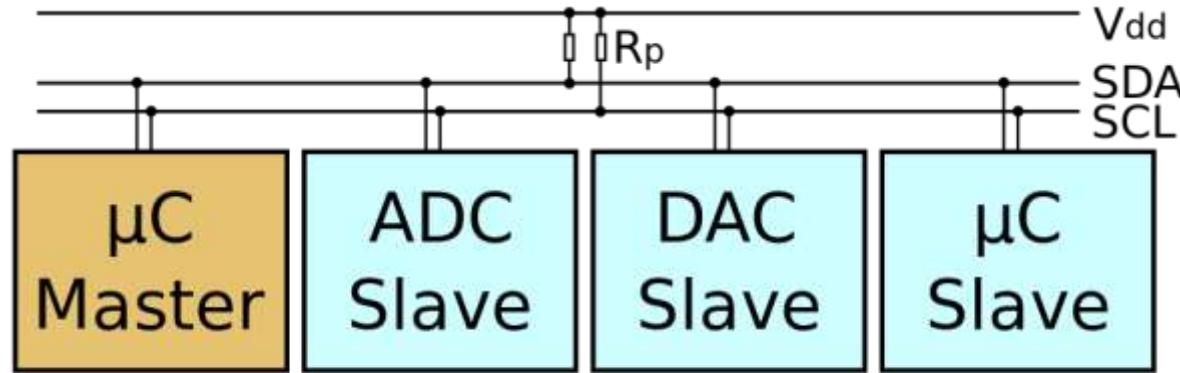
# I2C Bus Overview

- “Inter-Integrated Circuit” bus
- Multiple devices connected by a shared serial bus
- Bus is typically controlled by master device, subordinates respond when addressed
- I2C bus has two signal lines
  - SCL: Serial clock
  - SDA: Serial data
- Full details available in “The I2C-bus Specification”



Author: Colin M.L. Burnett, Source: Wikimedia

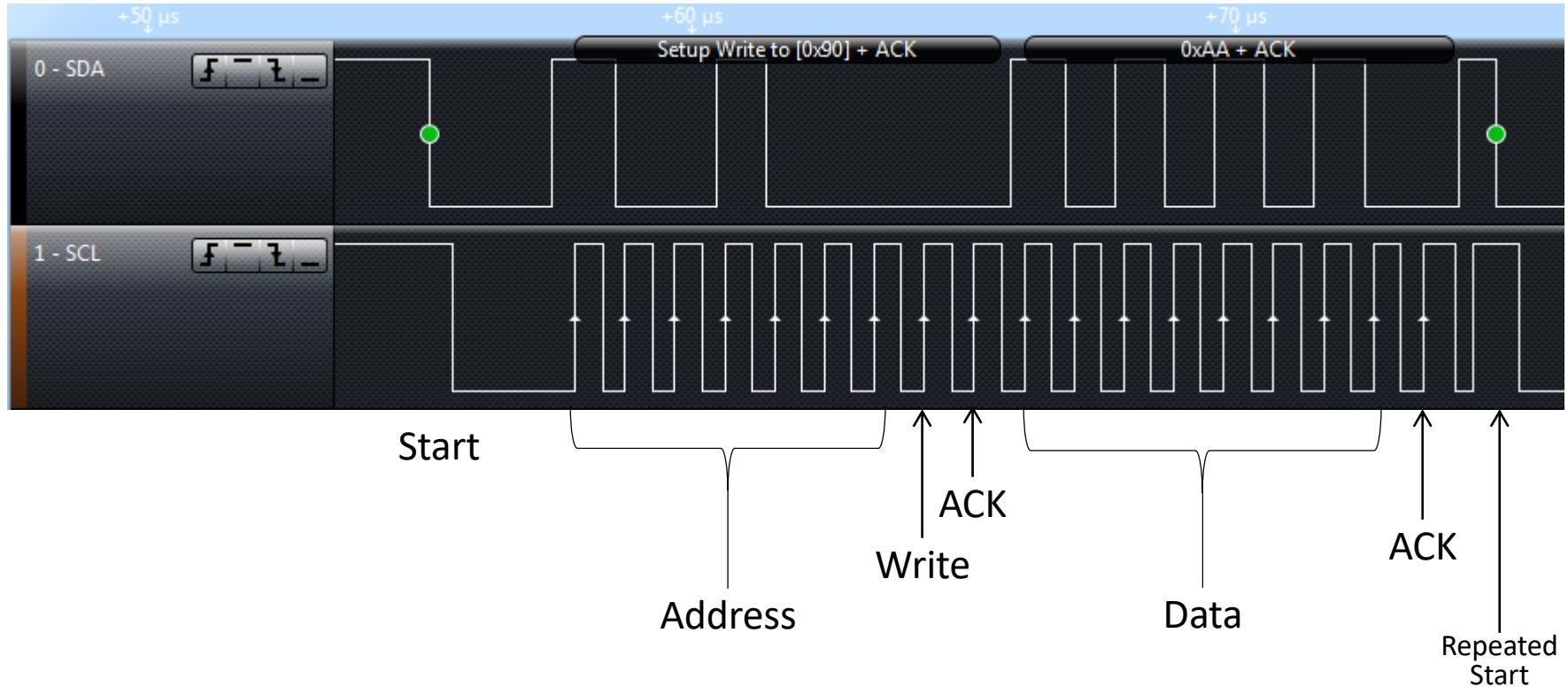
# I2C Bus Connections



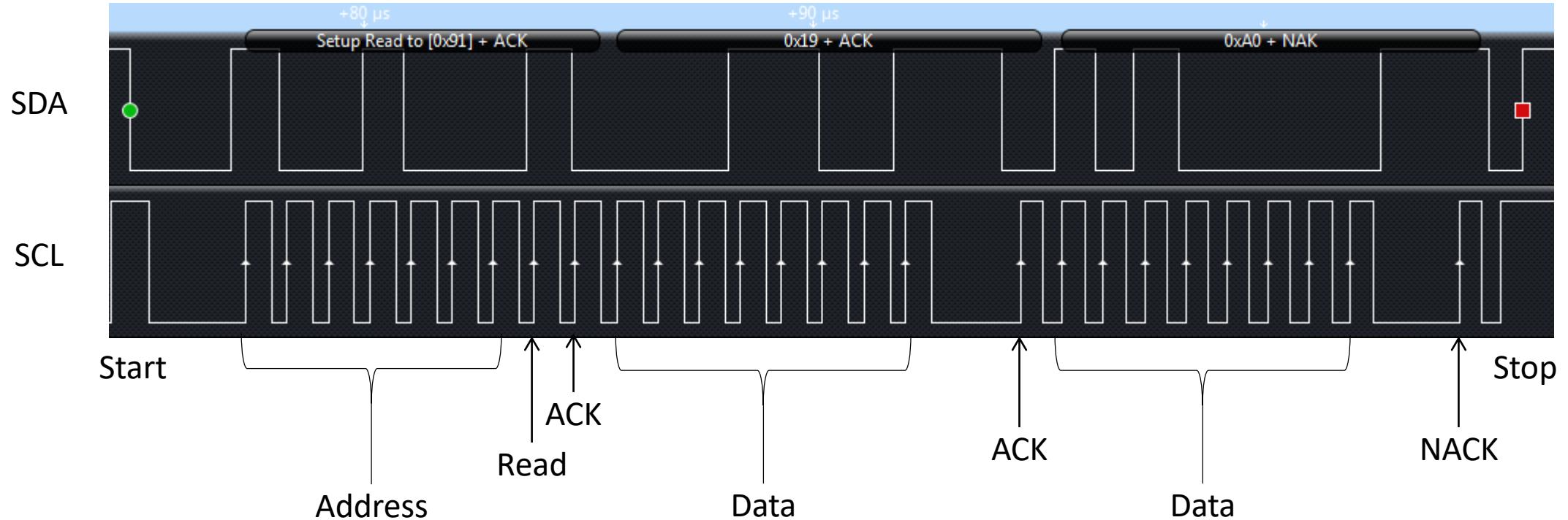
Author: Colin M.L. Burnett, Source: Wikimedia

- Resistors pull up lines to VDD
- Open-drain transistors pull lines down to ground
- Master generates SCL clock signal
  - Can range up to 400 kHz, 1MHz, or more

# Master Writing Data to Subordinate



# Master Reading Data from Subordinate



# I2C Addressing

- Each device (IC) has seven-bit address
  - Different types of device have different default addresses
  - Sometimes can select a secondary default address by tying a device pin to a different logic level
- What if we treat the first byte of data as a register address?
  - Can specify registers within a given device: 8 bits

# Enabling i2c

```
void temperature_init(void) {
 i2c_init();
 i2c_enable();
}

void temperature_enable(void) {
 // Start conversion.
 i2c_start();
 i2c_tx(subordinate_ADDRESS | WRITE);
 i2c_tx(START_CONVERT_T);
 i2c_stop();
}
```

# Reading 12 bits from a temperature sensor

```
float temperature_read(void) {
 short temp;
 i2c_start();
 i2c_tx(subordinate_ADDRESS | WRITE);
 i2c_tx(READ_TEMPERATURE);
 i2c_start();
 i2c_tx(subordinate_ADDRESS | READ);
 temp = i2c_rx() << 8;
 i2c_ack();
 temp |= i2c_rx();
 i2c_nack();
 i2c_stop();
 // Sign extend from 16-bit to 12-bit.
 temp >>= 4;
 // Convert from fixed-point to floating point.
 return temp / (float)16;
```

# PROTOCOL COMPARISON

# Factors to Consider

- How fast can the data get through?
  - Depends on raw bit rate, protocol overhead in packet
- How many hardware signals do we need?
  - May need clock line, chip select lines, etc.
- How do we connect multiple devices (topology)?
  - Dedicated link and hardware per device - point-to-point
  - One bus for manager transmit/subordinate receive, one bus for subordinate transmit/manager receive
  - All transmitters and receivers connected to same bus – multi-point
- How do we address a target device?
  - Discrete hardware signal (chip select line)
  - Address embedded in packet, decoded internally by receiver
- How do these factors change as we add more devices?

# Protocol Trade-Offs

| Protocol                     | Speed                                                                     | Signals Req. for Bidirectional Communication with N devices | Device Addressing                      | Topology                                              |
|------------------------------|---------------------------------------------------------------------------|-------------------------------------------------------------|----------------------------------------|-------------------------------------------------------|
| <b>UART (Point to Point)</b> | Fast – Tens of Mbit/s                                                     | 2*N (TxD, RxD)                                              | None                                   | Point-to-point full duplex                            |
| <b>UART (Multi-drop)</b>     | Fast – Tens of Mbit/s                                                     | 2 (TxD, RxD)                                                | Added by user in software              | Multi-drop                                            |
| <b>SPI</b>                   | Fast – Tens of Mbit/s                                                     | 3+N for SCLK, MOSI, MISO, and one SS per device             | Hardware chip select signal per device | Multi-point full-duplex, multi-drop half-duplex buses |
| <b>I<sup>2</sup>C</b>        | Moderate – 100kbit/s, 400 kbit/s, 1Mbit/s, 3.4Mbit/s.<br>Packet overhead. | 2 (SCL, SDA)                                                | In packet                              | Multi-point half-duplex bus                           |



<sup>†</sup>The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

[www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks)



# Programming for Power-Efficient Computing : High Level Techniques

# Learning Objectives

At the end of this lecture, you should be able to:

- Describe the risks with optimizing code at high level.
- Describe the advantages of high-level code optimisation.
- Identify different types of data structures and their mode of data access.
- Explain binary search algorithm.
- Describe the advantages and drawbacks of using optimized libraries in programming.

# Outline

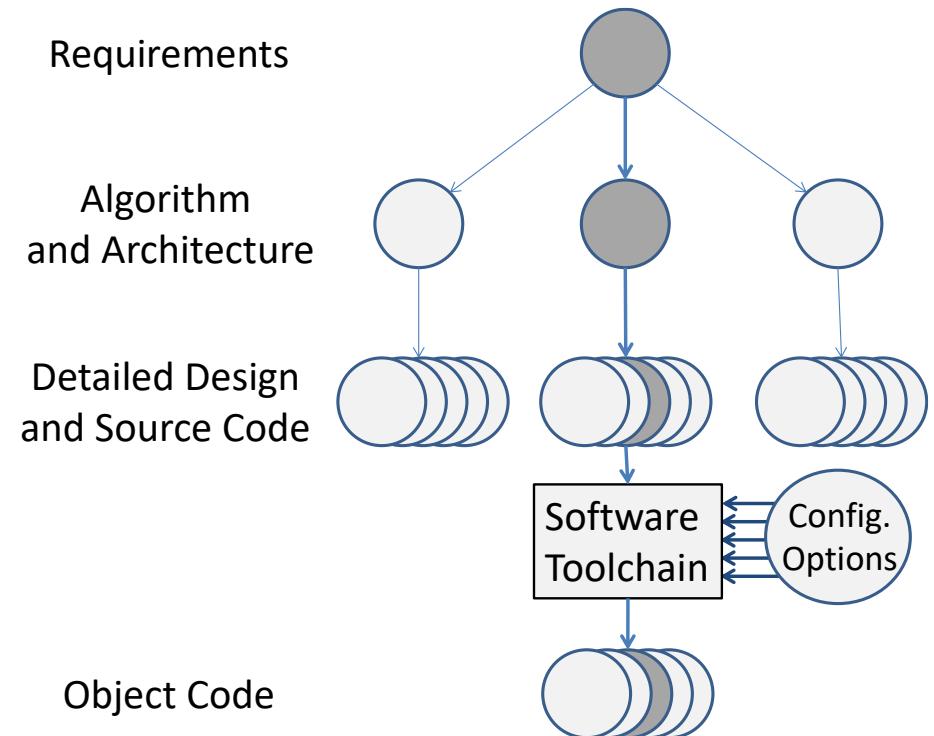
- Low Power Computing
- Optimization & The Software Development Process
- Optimization Risks
- High Level Optimizations with Examples
- Better Algorithms – Search Example
  - Making Searches Faster
  - More Data Structures
  - Optimization: Binary Search
  - Use of Optimized Libraries
  - Precision
  - Open-source Software

# Low Power Computing

- Power = Energy / Time
- Personal Computers operate to a maximum power budget.
- Mobile Computers operate to maximum power and energy budgets.
- There are many ways in which we can reduce power consumption:
  - Reduce the amount of computations necessary e.g. optimize programs to remove redundant operations, use optimized libraries, reduce memory transactions by reusing local data, use better algorithms.
  - Toolchain optimization e.g. help the compiler do a better job though directives to exploit hardware-specific features.
  - Exploit power efficient hardware-assisted techniques e.g. big.LITTLE multiprocessing, Dynamic Voltage and Frequency Scaling (DVFS).
  - Improve hardware occupancy e.g. better scheduling algorithms, hide latency.

# Optimization & The Software Development Process

- Many opportunities for optimization
  - Multiple levels of design hierarchy.
  - Requirements, algorithm, architecture, design, implementation...
- Some computer systems are more amenable/open to optimizations than others.
- Higher-level optimizations may span multiple levels
  - Impacts more code than lower level optimizations, requiring more development effort.
  - May add more risk.



# Optimization Risks

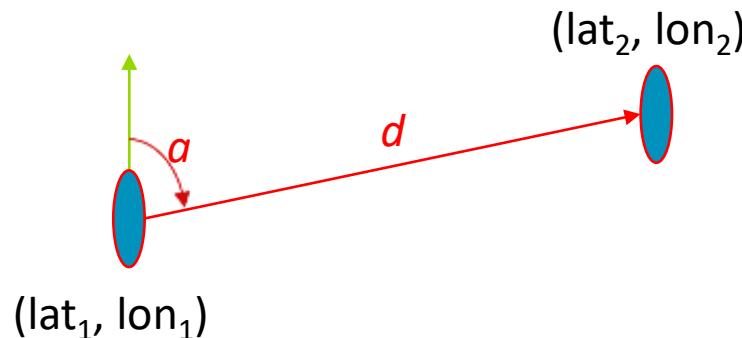
- Unpredictability of development effort needed
  - Balancing act
    - Pro: expected performance gain
    - Cons: additional development time required, increased schedule risk
  - Difficulties in prediction
    - How much gain will we get after this optimization? Will it be optimized enough so we can stop optimizing the program?
    - How long will it take to perform this optimization?
    - How many more optimizations will we need?
- Impact on code maintainability
  - Code will be used in future
    - Bug fixes, feature changes, feature additions, upgrades
    - Basis for follow-up and evolved products, platform for range of products
    - What if you've forgotten how your optimized code works?
    - What if someone else needs to maintain your optimized code?
    - Optimization often hurts code maintainability
    - Need to optimize in a way which retains maintainability

# High Level Optimizations: Do Less Work

- Fundamental concept: perform less computation
  - Lazy (or deferred) execution: don't compute data until needed
  - Early decisions: for decisions based on computations, may be able to use intermediate results
- Applied broadly
  - Many algorithms implement these concepts
  - Compilers try to apply these in optimization passes
- Role of developer
  - Implement concepts directly in source code
  - Help the compiler apply these concepts

# Example Program: “Nearby Points of Interest”

- Find distance and bearing from current position to the closest position of a fixed set of positions.
- Positions are described as coordinates on the surface of the Earth (latitude, longitude)



$$d = \text{acos}((\sin(\text{lat}_1) * \sin(\text{lat}_2) + (\cos(\text{lat}_1) * \cos(\text{lat}_2)) * \cos(\text{lon}_2 - \text{lon}_1)) * 6371$$

$$a = \text{atan2}(\cos(\text{lat}_1) * \sin(\text{lat}_2) - \sin(\text{lat}_1) * \cos(\text{lat}_2) * \cos(\text{lon}_2 - \text{lon}_1), \sin(\text{lon}_2 - \text{lon}_1) * \cos(\text{lat}_2)) * \frac{180}{\pi}$$

## Example Program: “Nearby Points of Interest” - Core Code: Calculate Distance

```
float calc_Distance(PT_T * p1, const PT_T * p2) {
 // calculates distance in kilometers between locations
 return acos(sin(p1->Lat*PI/180)*
 sin(p2->Lat*PI/180) +
 cos(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
 cos(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
}
```

# Example Program: “Nearby Points of Interest”

- Calc\_Distance is called on every point, returning closest\_d (distance in km)
- This distance has two uses
  - To identify closest point
  - To be returned to calling function
- Can we split these up?

```
void Find_Nearest_Point(. . .) {
 . . .
 while (strcmp(points[i].Name, "END")) {
 d = Calc_Distance (&ref, &(points[i]));
 if (d<closest_d) {
 closest_d = d;
 closest_i = i;
 }
 i++;
 }
```

# Distance Calculation

- Distance is  $\text{acos}(\text{big\_expression}) * 6371$
- What is 6371?
  - Scaling factor to convert angle in radians to distance in km
  - $6371\text{km} = \text{Earth's circumference}/2\pi \text{ radians}$
- Can compare angle (returned by  $\text{acos}$ ) rather than distance
  - Angle is still proportional to distance between points

```
float Calc_Distance(PT_T * p1, const PT_T * p2) {
 // calculates distance in kilometers between
 locations
 return acos(p1->SinLat * p2->SinLat +
 p1->CosLat * p2->CosLat
 *cos(p2->Lon - p1->Lon)) * 6371;
}
```

# Optimized Code

- Eliminates  $N_{\text{Points}} - 1$  floating point multiplies

```
float Calc_Distance_in_Radians(PT_T * p1, const PT_T * p2) {
 // calculates distance in radians between locations
 return acos(p1->SinLat * p2->SinLat +
 p1->CosLat * p2->CosLat * cos(p2->Lon - p1->Lon)); // no *6371 here
}

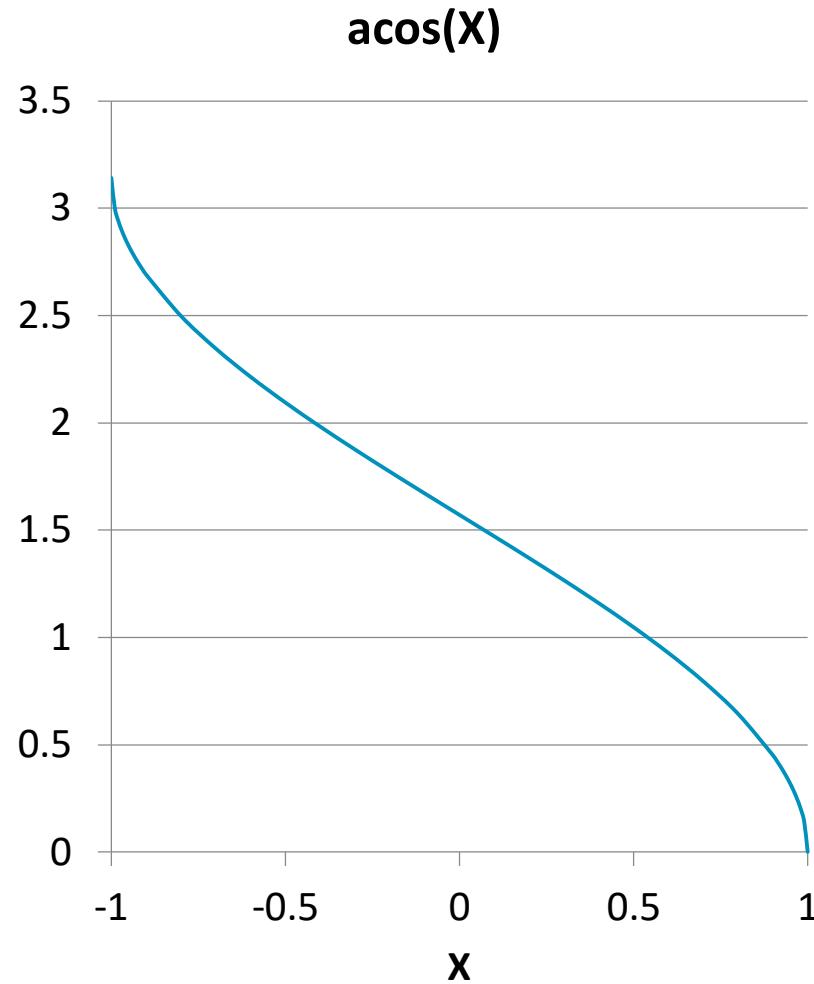
void Find_Nearest_Point(. . .) {
 while (strcmp(points[i].Name, "END")) {
 d = Calc_Distance_in_Radians(&ref, &(points[i]));
 . . .
 }
 *distance = d*6371;
 . . .
}
```

# Taking it Further

- Can we make distance comparisons *without* using acos?
  - How is **acos** related to its argument X?
- Can we call acos just once – to compute the distance to the closest point?

```
float Calc_Distance_in_Radians(PT_T * p1, const PT_T * p2) {
 // calculates distance in radians between locations
 return acos(p1->SinLat * p2->SinLat +
 p1->CosLat * p2->CosLat * cos(p2->Lon - p1->Lon));
}
```

# Taking it Further



```
float Calc_acos_arg (PT_T * p1,
const PT_T * p2) {

 return (
 p1->SinLat * p2->SinLat +
 p1->CosLat * p2->CosLat
 *cos(p2->Lon - p1->Lon));
}
```

- $\text{acos}$  always decreases when input  $X$  increases
- Nearest point will have minimum distance and maximum  $X$
- So search for point with maximum argument to  $\text{acos}$  function
- After finding nearest point (max  $X$ ), compute  $\text{distance\_km} = \text{acos}(X) * 6371$

# Even More Optimized Code

- Eliminates  $N_{\text{Points}} - 1$  floating point acos calculations.

```
float Calc_Distance_inverse (PT_T * p1, const PT_T * p2) {
 // calculates distance in radians between locations
 return (p1->SinLat * p2->SinLat +
 p1->CosLat * p2->CosLat * cos(p2->Lon - p1->Lon)); // no acos here
}

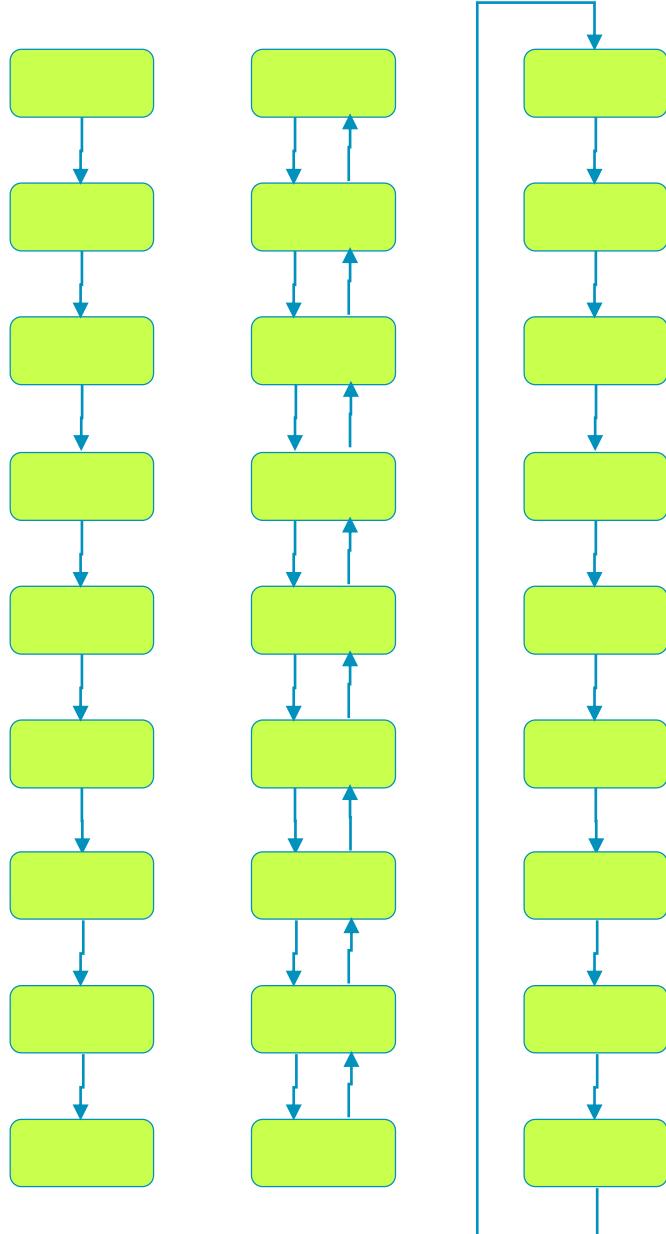
void Find_Nearest_Point(. . .) {
 while (strcmp(points[i].Name, "END")) {
 d = Calc_Distance_inverse(&ref, &(points[i]));
 if (d>closest_d) closest_d = d;
 i++;
 }
 *distance = acos(d)*6371;
 . . .
}
```



# Better Algorithms – Search Example

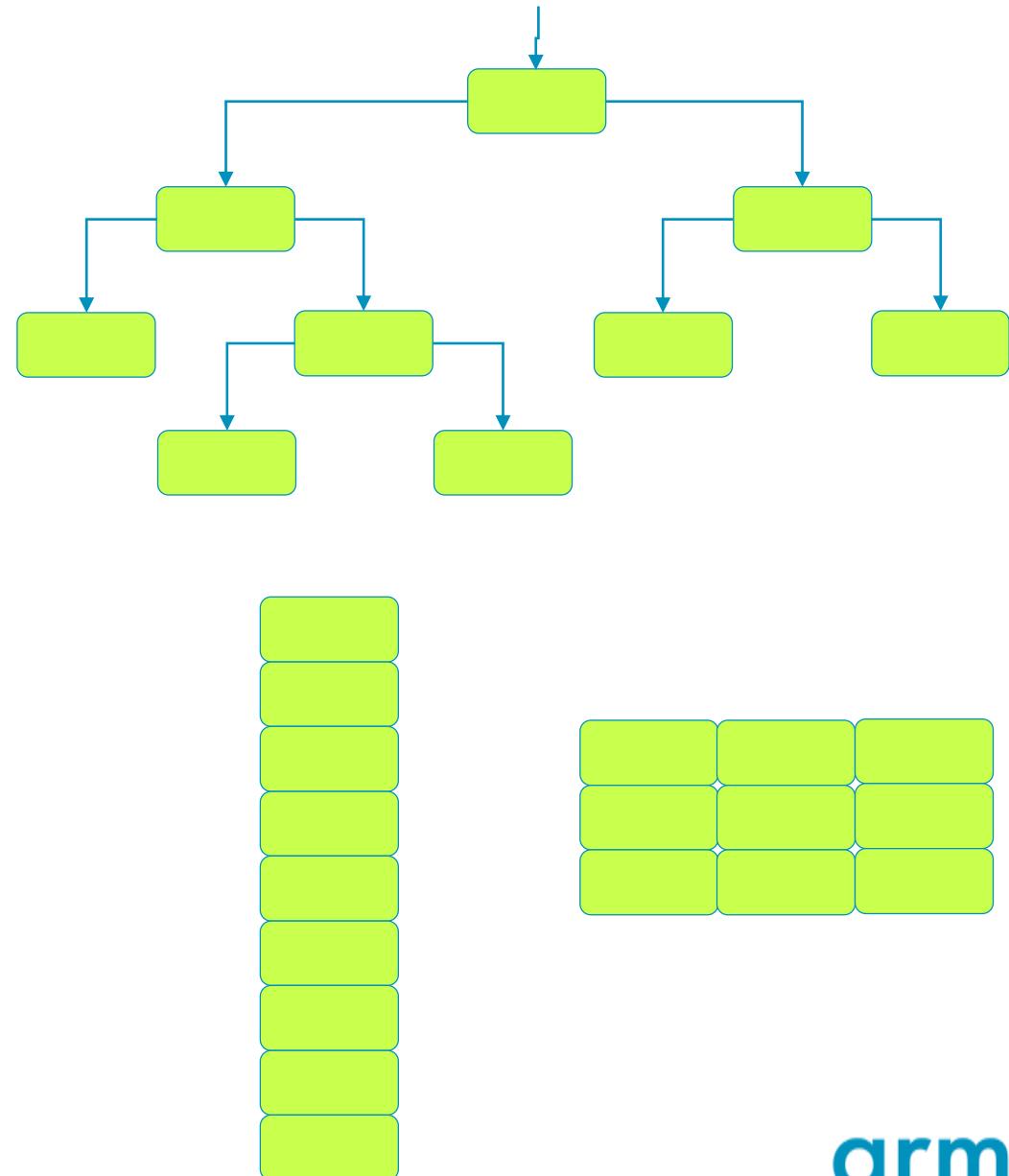
# Making Searches Faster

- Improve the data organization, possibly also enabling a better algorithm
- Example data structure: List
  - Each node holds a data element.
    - May be connected to one or two other nodes with pointers:
      - One successor (next)
      - Optional: one predecessor (prev)
  - Sequential access to data.
    - Must start at current node (or start node) and traverse list by visiting nodes via next or prev pointers.
  - Examples: linked list, queue, circular queue, double-ended queue.



# More Data Structures

- Tree - hierarchical
  - Each node holds a data element. May be connected to other nodes with pointers:
    - Up to one parent
    - Down to N children
  - Sequential access to data. Must traverse by visiting nodes, but additional connections reduce number of intermediate nodes.
  - Hierarchical structure. May be represented explicitly with pointers or implicitly with index location of element.
- Array – random access
  - Same time to access each element
  - Flat structure
  - May be multidimensional



# Optimization: Binary Search

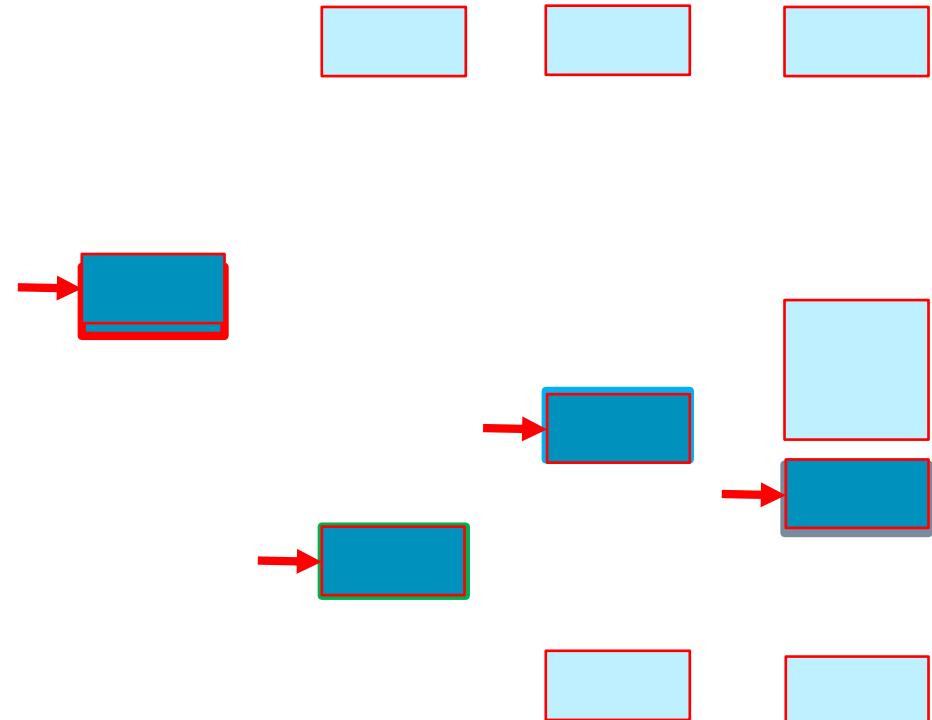
Step 1

Step 2

Step 3

Step 4

- *Divide and Conquer* approach
- Requirements
  - Regions in table must be *sorted by increasing starting address*
  - For each entry, start address  $\leq$  end address
- Start in middle of the region table
- Compare entry's start and end addresses with search address
  - If search address is *before* start address, then repeat with *upper* portion of table
  - If search address is *after* end addresses, then repeat with *lower* portion of table
  - Otherwise, we have found the region, so search is done
- Repeat until finding matching region or there's no table left to search
- The time complexity is reduced to  $O(\log(n))$  compared to  $O(n)$  for a brute force search ( $n$  is the number of elements).



# Use of Optimized Libraries

- The use of optimized libraries can achieve your goals at a relatively small extra investment.
- The use of optimized libraries however can make your code less portable across different platforms.
- Some of the most widely used optimized libraries are Math libraries
  - NAG (Numerical Algorithm Group) Numerical Library: Software library of numerical analysis routines e.g. linear algebra, optimization, differential equations, and regression analysis.
  - Intel Math Kernel Library (MKL): optimized Maths routines (e.g. linear algebra, sparse solvers, FFTs) for science, engineering, and financial applications. Optimized specially for Intel processors.
  - CMSIS-DSP: Software library with a suite of common signal processing functions (e.g. filters, transforms, matrix algebra) for use on energy-efficient Cortex-M processor based devices.

# Precision

- Match the data types and approximations method to the range and accuracy needs of your algorithm.
- You can trade-off accuracy/precision for speed, code size, energy/power.
- Floating point arithmetic is needed when you are dealing with a large range of data values that fixed-point arithmetic cannot deal with.
- Single precision floating point uses 32 bits (8 bits for exponent, 24 for fraction mantissa) whereas double-precision floating point precision uses 64 bits (11 bits for exponent, 53 for fraction mantissa).
- Floating point is slow if there is no hardware support (must be emulated in software)
- The IEEE standard for floating point arithmetic (IEEE 754) is a technical standard for floating-point computation widely implemented in hardware.

# Open Source Software

- Many optimized libraries are available as Open Source Software (OSS).
- You need to know about the type of license you are signing up to before you use any OSS.

## Permissive

- License requirements are minimal.
- Broad grant of rights (with no conditions for particular licensing terms).
- Includes MIT, BSD, and Apache 2.0 licenses.

## Copyleft

- Source code must be made available for binary distribution.
- Original work, any modifications, any derivative work must remain under the same license.
- Include GPL, LGPL, and MPL licenses.



<sup>†</sup>The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

[www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks)

arm

# Programming for Power-Efficient Computing : Low Level Techniques

# Learning Objectives

At the end of this lecture, you should be able to:

- Outline the stages of the compiler
- Identify the two types of scalar optimizations and their features.
- Explain machine-independent code optimization and its advantages.
- Explain machine-dependent code optimization and its advantages.
- Describe how the following could hinder compiler optimization: excessive variable scope and Automatic Promotions in Arithmetic Expressions.
- Compare and contrast optimisation for power, energy, speed and code size.

# Outline

- What should the compiler be able to do?
- What could stop the compiler from optimizing?
- How can we tell if the compiler has applied the optimizations?
- How can we modify the source code to help the compiler optimize?

# Starting Points for Efficient Code

- Write correct code, then optimize.
- Use a top-down approach.
- Know your microprocessor's architecture, compiler, and programming language.
  - Use the right tool for the problem
- Consider the use of optimized libraries for critical paths in your program.

# Review of Compiler Stages

- Parser
  - reads in source code e.g. C code, checks for lexical and syntax errors,
  - forms intermediate code (tree representation)
- High-Level Optimizer
  - Modifies intermediate code (processor-independent)
- Code Generator
  - Creates assembly code step-by-step from each node of the intermediate code
  - Allocates variable uses to registers
- Low-Level Optimizer
  - Modifies assembly code (parts are processor-specific)
- *Assembler*
  - *Creates object code (machine code)*
- *Linker/Loader*
  - *Creates executable image from object file*

arm

don't handcuff the  
compiler

# Approach

- Which optimizations is the compiler capable of?
- What might stop the compiler from applying them?
- How can we tell if the compiler applied them?
- How can we modify the source code to help the compiler optimize?



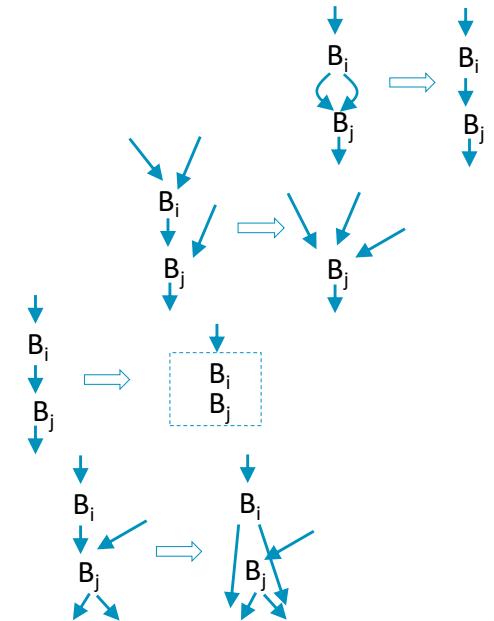
What should the  
compiler be able to do?

# Scalar Optimizations

- What should you expect your compiler to be able to do?
- Machine-Independent (MI)
  - Eliminate code with no effect
  - Specialize computations
  - Eliminate redundant computations
- Machine-Dependent (MD)
  - Take advantage of special hardware features
  - Manage or hide latency
  - Manage limited machine resources

# MI: Eliminate code with no effect

- Useless code (based on data-flow graph - DFG)
  - Mark *critical operations*
    - sets return value for procedure
    - input/output statement
    - modifies non-local data
  - Find operations which define data (operands) used by these critical operations
  - Repeat until set of critical operations doesn't grow
  - Delete remaining operations
- Useless control flow (based on control flow graph - CFG)
  - Fold redundant branches
  - Remove empty blocks
  - Combine blocks
  - Branch hoisting
  - Unreachable code
    - Examine CFG for nodes without predecessors
  - Simplification of algebraic identities
    - $x * 1 \rightarrow x$
    - $x + 0 \rightarrow x$



# MI: Specialize computations

- Operator strength reduction
  - Replace multiplication with addition or shifting, division with subtraction or shifting
- Constant propagation
  - if a variable has a known value at a given point in the program, it might be possible to specialize operations based on this knowledge e.g. perform calculations at compile time rather than runtime.
- Peephole optimization
  - Recognize patterns of assembly instructions which can be replaced with a faster set

# MI: Enabling Transformations

- Goal is to make code more amenable to *other* optimizations

- Loop unrolling

- replicate loop body
- Increases speed by reducing loop overhead
- At the expense of increase binary code size

```
for (i = 0; i < 100; i++) →
 function-call ();
for (i = 0; i < 100; i+=2){
 function-call ();
 function-call ();
}
}
```

- Loop unswitching

- hoist loop-invariant control-flow operations out of loop

```
for (i=0; i<N; i++) { →
 if(x>y) {
 a[i] = b[i] * x;
 } else {
 a[i] = b[i] * y;
 }
}
if(x>y){
 for (i=0; i<N; i++)
 a[i] = b[i] * x;
} else {
 for (i=0; i<N; i++)
 a[i] = b[i] * y;
}
```

- Renaming

- value numbering gives independent name to each value defined
- optimizer can recognize already-computed values which are still live

a  $\leftarrow$  x + y  
b  $\leftarrow$  x + y  
a  $\leftarrow$  17  
c  $\leftarrow$  x + y

$a_0 \leftarrow x_0 + y_0$   
 $b_0 \leftarrow x_0 + y_0$   
 $a_1 \leftarrow 17$   
 $c_0 \leftarrow x_0 + y_0$

# Machine-Dependent Optimizations

- Take advantage of special instructions, addressing modes and hardware features
  - Instruction pre-fetch, branch prediction (special processor circuitry)
  - Conditional instruction execution (Arm instructions)
  - Advanced addressing modes (processor ISA dependent)
- Manage or hide latency (pipeline, memory system, ALU)
  - Large memory latency – reschedule operations to keep the processor busy
  - Rearrange loop iteration order for cache locality (e.g. to use GPU local memory)
  - Change data layout to improve cache locality
- Manage limited machine resources
  - Register allocation



What could stop the  
compiler from  
optimizing?

# Excessive Variable Scope

- Avoid declaring variables as globals or statics when they could be locals
- Globals and statics are allocated permanent storage in memory, not reusable stack space
- Compiler assumes that any function may access a global variable
  - Function must write back any globals it has modified before calling a subroutine
  - Function must reload any globals to use after calling the routine

# Automatic Promotions in Arithmetic Expressions

- How are expressions with mixed data types evaluated?

```
float f;
char c;
int r;
```

```
r = f * c;
```

- ANSI C Standard has rules for arithmetic conversions
  - If either operand is a long double, promote the other to a long double
  - Else if either is a double, promote the other to a double
  - Else if either is a float, promote the other to a float
  - Else if either is a unsigned long int, promote the other to a unsigned long int
  - Else if either is a long int, promote the other to a long int
  - Else if either is an unsigned int, promote the other to an unsigned int
  - Else both are promoted to int: short, char, bit field
  - *special rules for dealing with signed/unsigned differences left out*

# Resulting Object Code

```
float f;
char c;
int r;

r = f * c;
```

- Call routine to convert c to float
- Call routine to perform floating point multiply with f
- Call routine to convert result from floating point to integer
- store result in r

- Time and code space overhead of conversion routines
- Avoid mixed type expressions

# ANSI C Standard for Argument Promotions

- Integral function arguments smaller than an int for non-prototyped functions are promoted to ints
  - Extra time converting to int
  - Extra space on stack
- So prototype all functions
  - Function:

```
int Find_Average(char a, char b, char c, char d) {
 ...
}
```
  - Correct, complete prototype:

```
int Find_Average(char a, char b, char c, char d);
```
  - Parameter names are optional but good for documentation/maintainability.
- Where should the prototype go?
  - If program is broken into modules, put prototype in header (.h) file
  - Otherwise put prototype near top of C code file, before the function is called

# Compiler Optimization Levels e.g. Arm Compiler

- Optimizing for code vs. speed
  - –Otime: the compiler aggressively optimizes for time e.g. aggressive inlining, at the expense of a possible increase in image (binary) size.
  - –Ospace: the compiler optimizes for image size at the expense of a possible increase in execution time.
- Optimization levels:
  - -O0 : Minimum optimization. Turns off most optimizations. Best possible debug view.
  - -O1 : Restricted optimization e.g. removes unused inline functions. Turns off optimizations that seriously degrade the debug view.
  - -O2 : High optimization. e.g. compiler automatically inlines functions if optimizing for time. The compiler may perform optimizations that cannot be described by debug information.
  - -O3 : Maximum optimization e.g. loop unrolling if optimizing for time. This can give significant performance benefits at a small code size cost, but at the risk of a longer build time.



How Can WE TELL IF THE  
COMPILER HAS APPLIED  
THE OPTIMIZATIONS?

# Profiling Computer Programs

- Code profilers are tools that instrument computer programs (either the program source code or its binary form(s)).
- Code profilers can measure the frequency and execution time of certain instructions or function calls, memory accesses and memory usage, event trace etc.
- Code profilers are crucial for code optimization e.g. for speed, memory footprint, energy/power.
- Some of the most popular profilers include:
  - Statistical profilers such as Shark (OSX), oprofile (Linux), VTune (Intel) and Streamline (Arm).
  - Intermediate language instrumentation tools such as OpenPAT.
  - Runtime instrumentation tools such as Valgrind and DynamoRIO.

arm

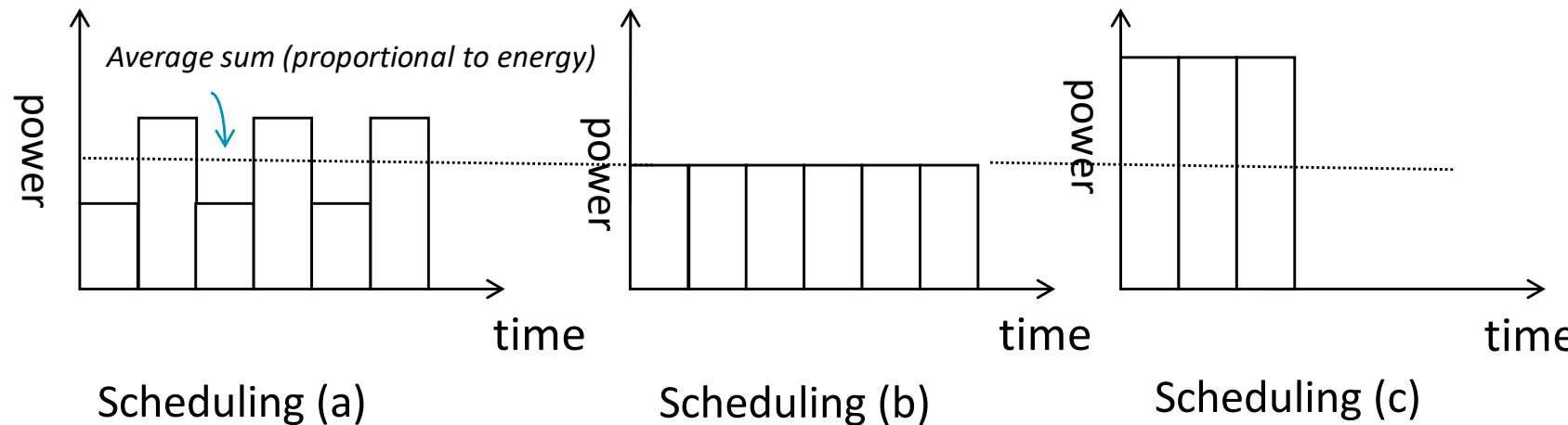
How can we modify the  
source code to help the  
compiler optimize?

# Iterative Optimization Process

1. Read the Compiler Manual
2. Apply the necessary hints/directives for the compiler to achieve the sought optimizations
3. Inspect the results
4. Go back to 1 and/or 2 if the results are not satisfactory

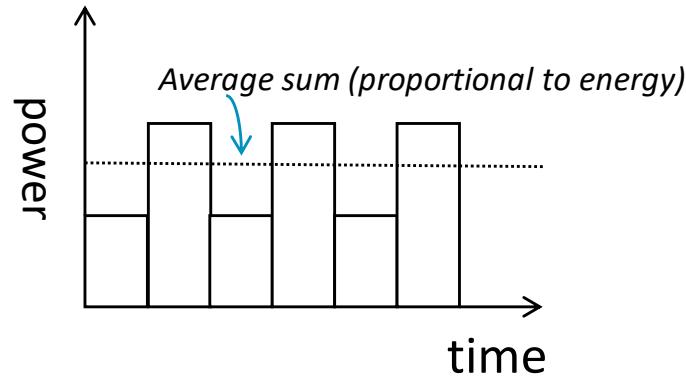
# Is Optimizing For Power The Same As Optimizing For Energy? (1/2)

- Not always. Optimizing for power aims to reduce the amount of energy consumed over an interval of time whereas optimizing for energy aims to optimize the total energy consumed.
- Remember: power is linked to heat, energy is the total you pay for in the end.
- Assume the power consumption is defined by the number of instructions executed at any moment in time\*\*. By rescheduling instructions on a VLIW processor for instance, or any parallel computer system, we can have different power and energy consumption profiles.

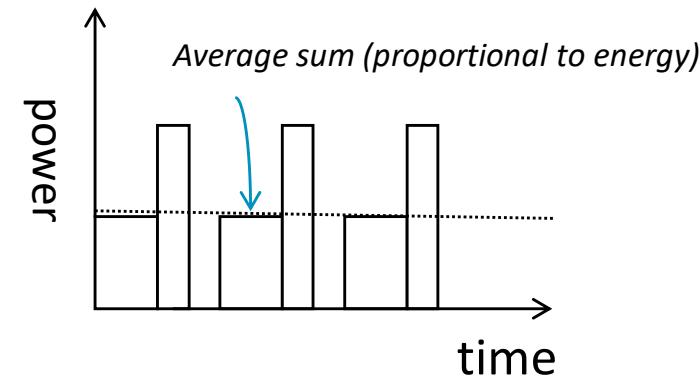


# Is Optimizing For Power The Same As Optimizing For Energy? (2/2)

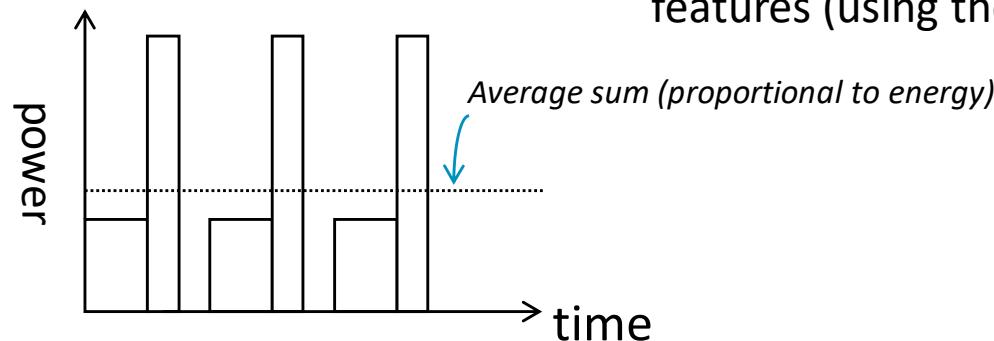
- Suppose we can execute certain instructions faster (e.g. in half the time) by using certain hardware features e.g. exploiting special SIMD hardware.



Scheduling (a)



Scheduling (d) exploits certain hardware features (using the same power budget)



Scheduling (e) exploits certain hardware features (at a higher power budget this time) – still less energy consumed overall compared to (a)

# Is Optimizing For Power/Energy The Same As Optimizing For Speed? (1/2)

- Not necessarily - often higher performance leads to more energy consumption e.g. through increasing clock frequency.
- However, eliminating redundant instructions or memory hierarchy optimizations, for instance, reduce the overall execution time (i.e. increase execution speed) and often result in energy reductions consequently. But even this is not always guaranteed.
- Consider the following code:

```
for (i=0; i<10; i++) {
 x = 2 * y
 z[i] = w[i] + 1;
}
```

- “ $x = 2 * y$ ” is loop invariant so we can take it out of the loop and execute it just once to save time.

# Is Optimizing For Power/Energy The Same As Optimizing For Speed? (2/2)

```
for (i=0; i<10; i++) {
 x = 2 * y
 z[i] = w[i] + 1;
}
```

```
x = 2 * y
for (i=0; i<10; i++) {
 z[i] = w[i] + 1;
}
```

- In a VLIW architecture, however, the code to the left might well be quicker to run as the processor could well be able to execute  $x = 2 * y$  in parallel with  $z[i] = w[i] + 1$  so there would be no need to execute it separately as in the case of the code to the right.
- Code to the left however would be more energy consuming as  $x = 2 * y$  would be executed 10 times. So here, quicker execution does not mean less energy consumption as redundant computation is needed to run the code quicker!
- Other similar examples could include memory prefetching and branch prediction.
- In general, optimizing for power/energy on the back of optimizing for speed is more demanding as we often have to compensate for the power/energy cost of extra hardware, or extra redundancy.

# Is Optimizing For Power/Energy The Same As Optimizing For Code Size?

- Not always. Yes, reducing code size results in less memory usage which can reduce power/energy consumption especially that memory/memory access is often power/energy hungry.
- Eliminating redundant variables for instance reduces code size and power/energy consumption.
- But reducing code size might also be achieved at the expense of more computations, which consume extra power/energy as illustrated in the variable value-swap codes below.

```
temp= x;
x=y;
y=temp;
```

```
x= x + y;
y= x - y;
x= x - y;
```

- Only two variables are used in the code to the right (instead of three in the code to the left).



<sup>†</sup>The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

[www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks)