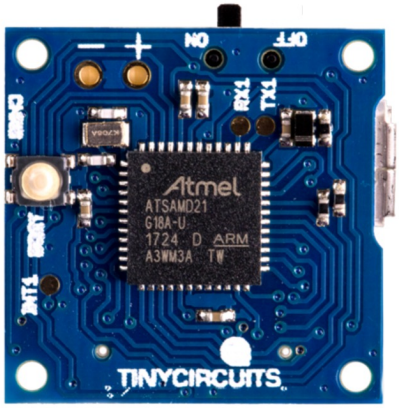
CSE190 Fall 2023

Lecture 1-19

Wireless Embedded Systems

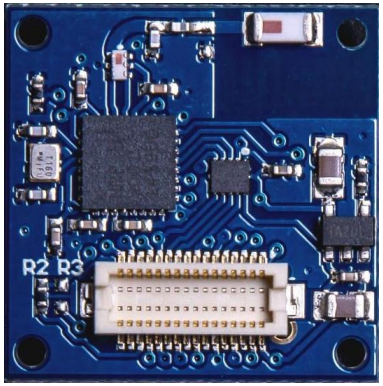
by ： Aaron Schulman

CSE190 Fall 2023



Lecture 1

Introduction



Wireless Embedded Systems

Aaron Schulman

CSE190: Wireless Embedded Systems

Goal: Provide a hands on introduction to the design of wireless embedded systems.

Class will focus broadly on how to program a fully functional wireless embedded system.

By the end of the course, you should be able to operate in a team that is building a product involving an embedded system.

Logistics

- Instructors: Aaron Schulman
  - TA: Tyler Potyodony
- Lectures: M/W/F 10:00am-10:50am
- Office hours:
  - Aaron: M/W 11:00am-12:00pm in CSE 3154
  - Tyler: Tu 11:30-2:30pm Th 2:00:5:00pm
- Web:
  - Course website: http://cseweb.ucsd.edu/classes/fa23/cse190-e
  - Discussion Forum: Piazza via Canvas

Lectures & Assignments

- Lecture slides will be posted on the course website
- Four assignments
  - Due approx. every two weeks
- Assignments will be low-level C firmware programming
  - Important to do them to keep pace in class
  - Collaboration will be limited to Piazza posts and Office Hours

Project & Exams

- Course project: build a battery-powered embedded sensor that transmits data to a smartphone.
  - Individual projects
  - You will have your own embedded systems hardware
    - **Please fill out signup sheet on Piazza**
- Two exams class (based on projects and in class)
  - Midterm Exam: Th Nov 8 (in class)
  - Final Exam: Th Dec 15 (8am-10am) :sad:

Grading Logistics

- Overall Class Grade
  - Four Assignments: 60%
  - Midterm Exam: 20%
  - Final Exam: 20%

What do computers look like today?

- Computing systems are everywhere
- Most of us think of computers as a platform for people:
  - PC's
  - Laptops
  - Servers
  - Smartphones

- But there's another type of computing system
  - They are far more common...

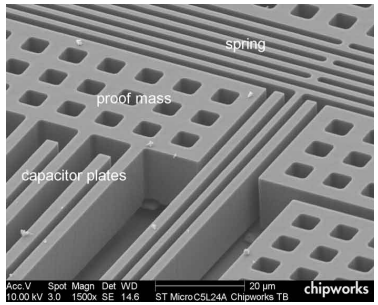
Wireless embedded systems:
The computing platform for *things*
(aka the *Internet of Things*
aka *Smart[ABC]*)

The components of an embedded system

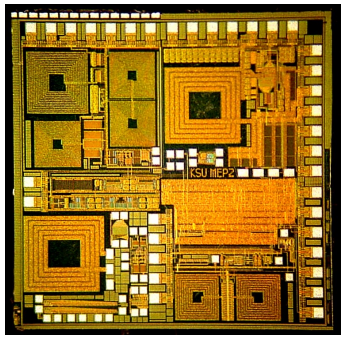
Embedded systems

Analog Systems

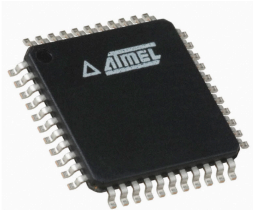
Tiny MEMS sensors



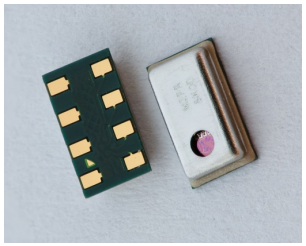
Tiny RF Integrated Circuits



Programmable Microcontrollers



Tiny sensors



Wireless networks

Bluetooth SMART

Digital Systems

Inexpensive computation

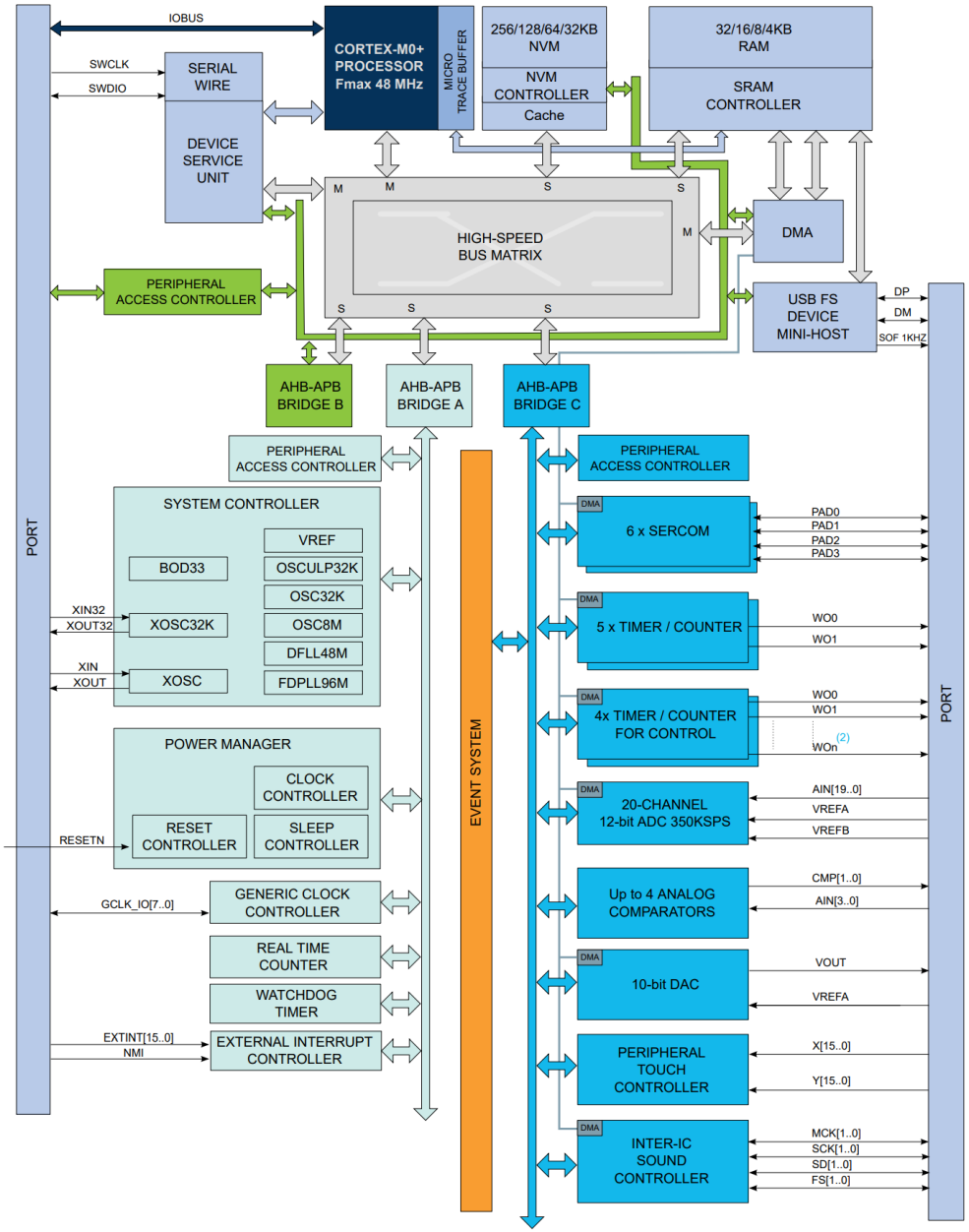
amazon web services™ EC2

Easy-to-use frameworks

hadoop

Spark

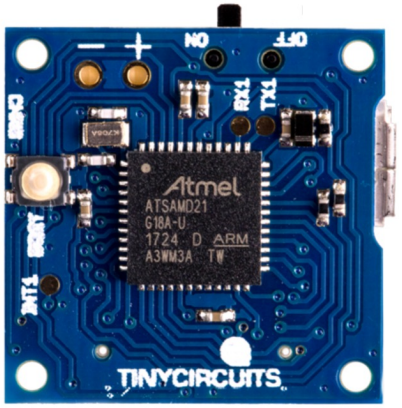
Typical Microcontroller

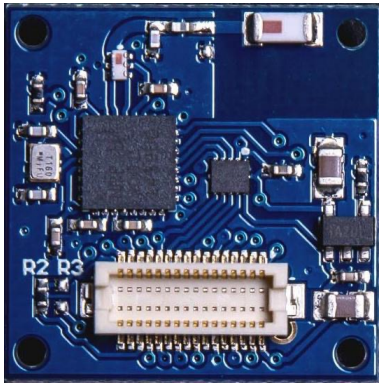


CSE190 Fall 2023

Lecture 2

Intro to Project and MCUs





Wireless Embedded Systems

Aaron Schulman

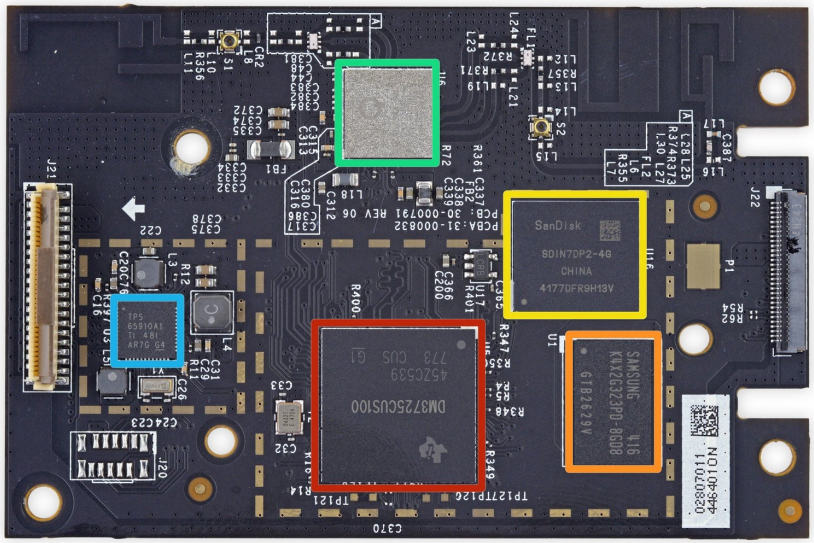
Wireless Embedded Systems

- Computing systems embedded within things/gadgets connected to the Internet
- Hard to call out any one thing/gadget, nearly everything now has a microcontroller and a wireless chipset in it
- 100s of millions of units produced yearly, versus millions of desktop/laptop units

Examples of wireless embedded systems

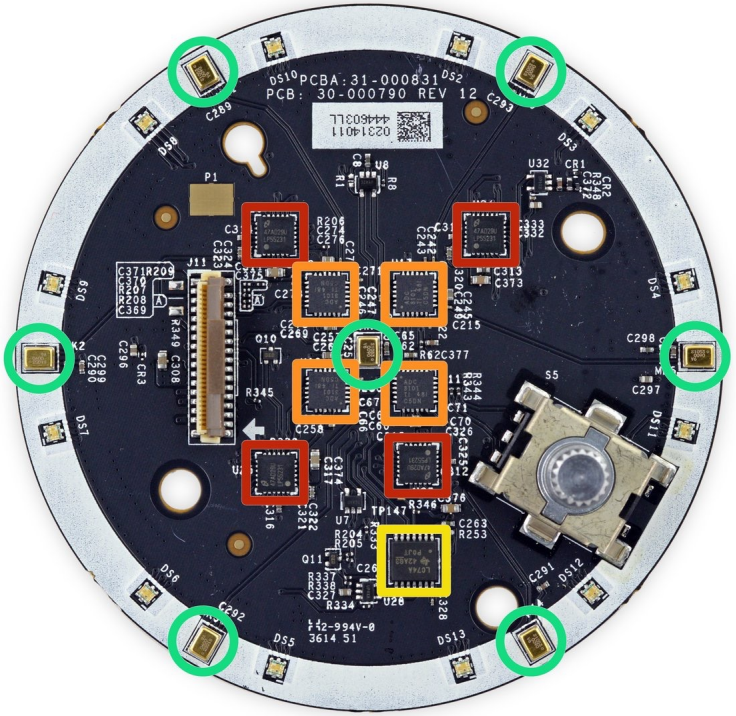
Amazon Echo (Alexa) – Voice Assistant

WiFi/Bluetooth

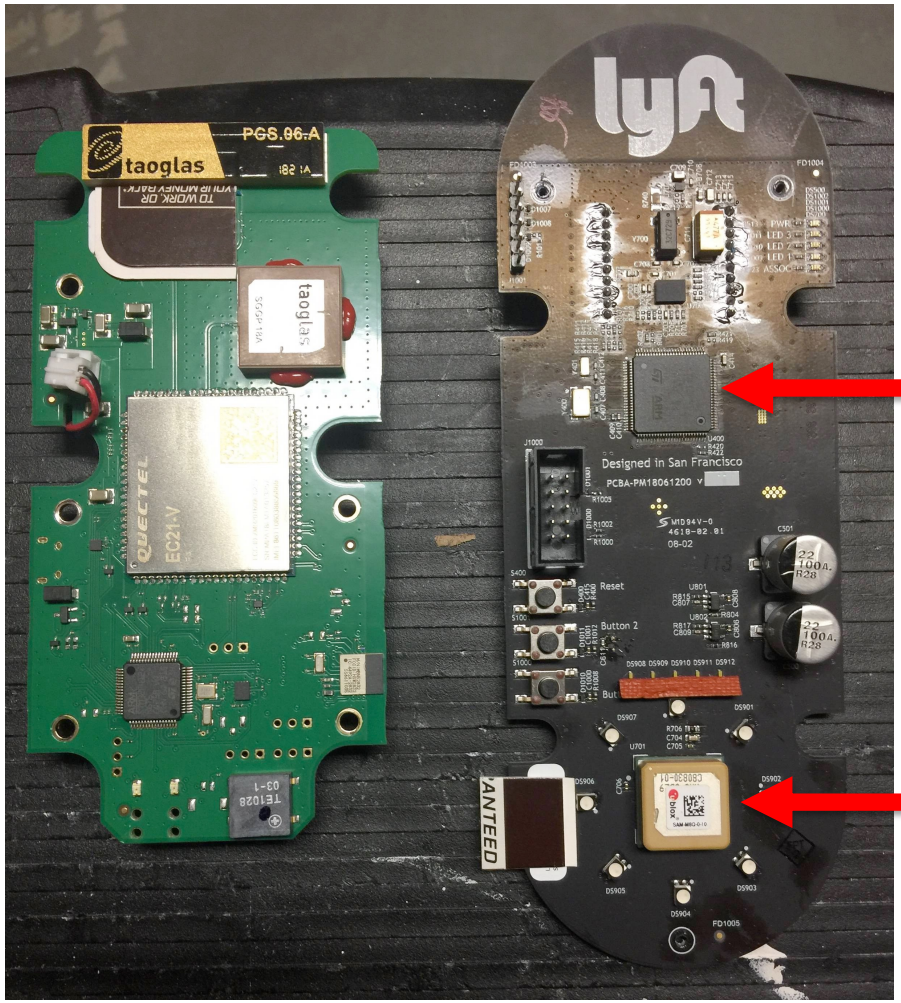


Microcontroller/DSP

8x microphone array



Examples of wireless embedded systems
Sharing Economy Scooters



YouLostIt Project Overview

Privacy-enabled Lost item tracker

Attaches to an item (e.g., backpack) and tells any nearby smartphone when it thinks it is “lost”. Provides visual identifier of user when lost, and radio notifications when it is lost for a long time.

YouLostIt:
Your privacy preserving battery-powered lost-device tracker

- Hardware
  - Accelerometer to detect when device is lost
  - LEDs to indicate who the lost owner is
    - (captured with smartphone video)
  - MCU that manages sensors, radios, and power
  - Bluetooth Low-Energy to tell a smartphone about lost device
    - Who owns it? How long has it been lost?
  - Flash memory to persistently store program binary
  - Clocks/Timers to keep track of time
  - Power from battery for several days to months.

- Software in a smartphone
  - Listen for nearby devices
  - Alert if the lost device goes out of range

What will you take away from this project?

- General knowledge of how an embedded system works
  - GPIO, Clocks, Interrupts, DMA, ADC, SPI, I2C, UART
- Low-level firmware development in C
  - Device drivers
  - Digital signal processing
- Hardware testing (and building)
  - Reading circuit layout and schematics
  - Analog circuit measurement (Oscilloscope, Multimeter)
  - Digital logic observation (Digital Logic Analyzer)

YouLostIt Hardware Platform

ST Micro Discovery board

What do you need to succeed with this project?

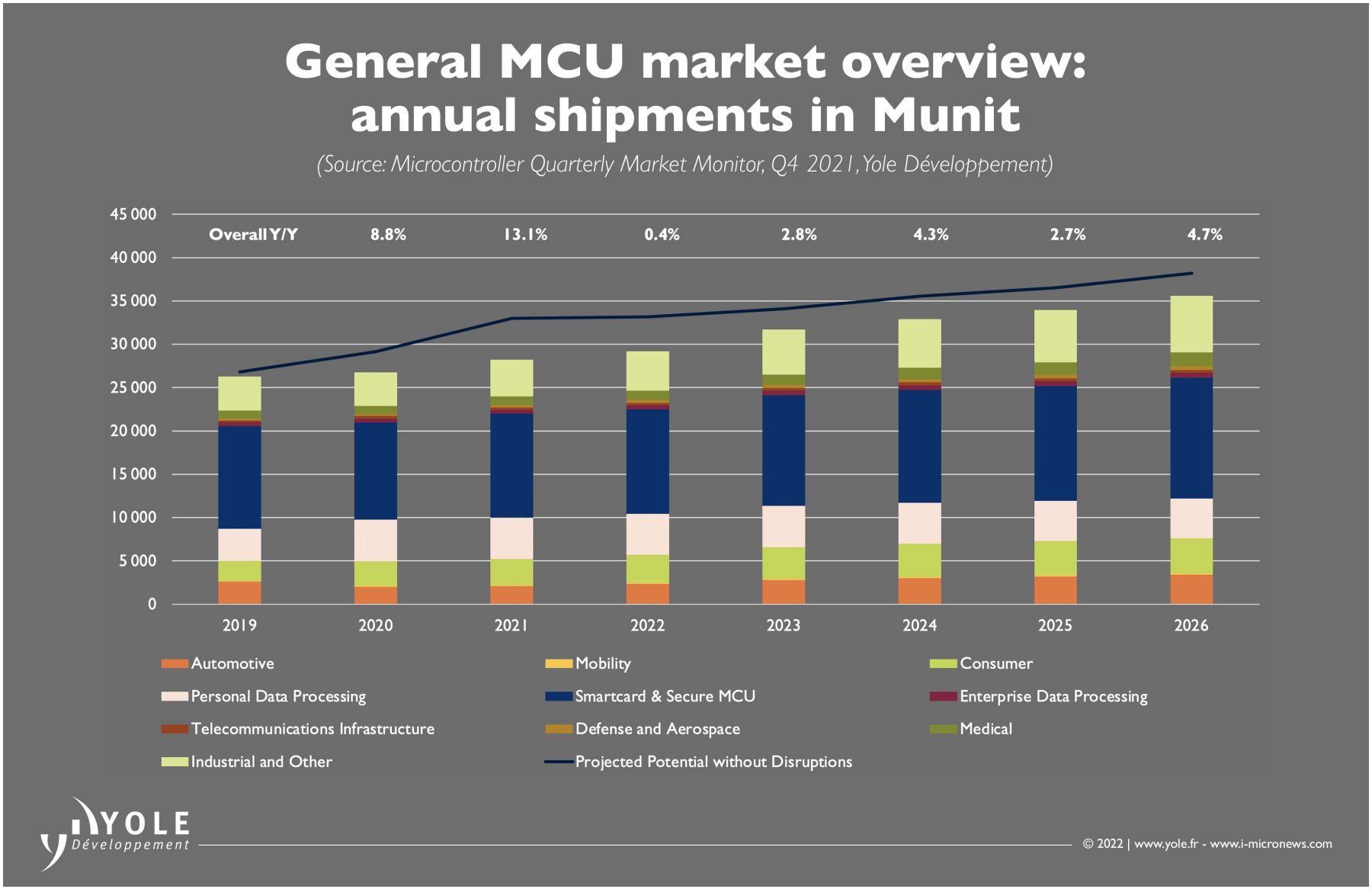
- C knowledge
- Desire to learn more about what happens inside of your computer.
- Tenacity: Hardware is hard (but fun!).
  - Need to find out how it works
  - Net to write code in C
  - Need to debug an entire computer w/no OS
- You would be surprised what you are capable of, you can make a physical thing!

Introduction to Microcontrollers

A microcontroller (MCU) is a small computer on a single integrated circuit consisting of a relatively simple central processing unit (CPU) combined with peripheral devices such as memories, I/O devices, and timers.

- By some accounts, more than half of all CPUs sold worldwide are microcontrollers

Billions of microcontrollers in use today



Microcontroller VS Microprocessor

- A microcontroller is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals.
- A microprocessor incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit.

Reading for next week

- Posted on Website – Please skim.
  - "ARM Cortex-M for Beginners"

WHITE PAPER

ARM

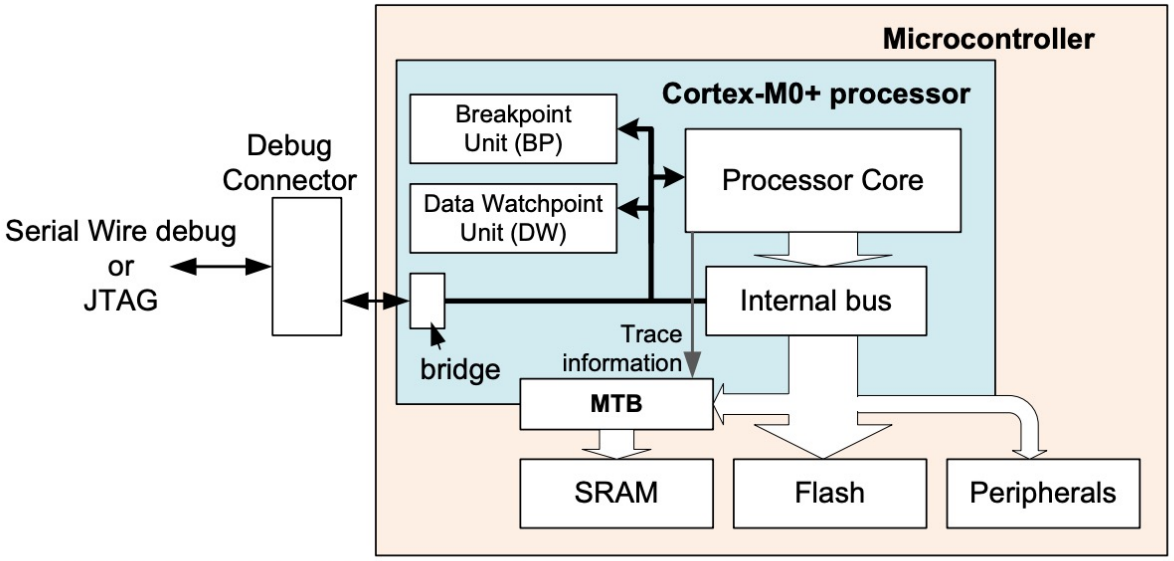
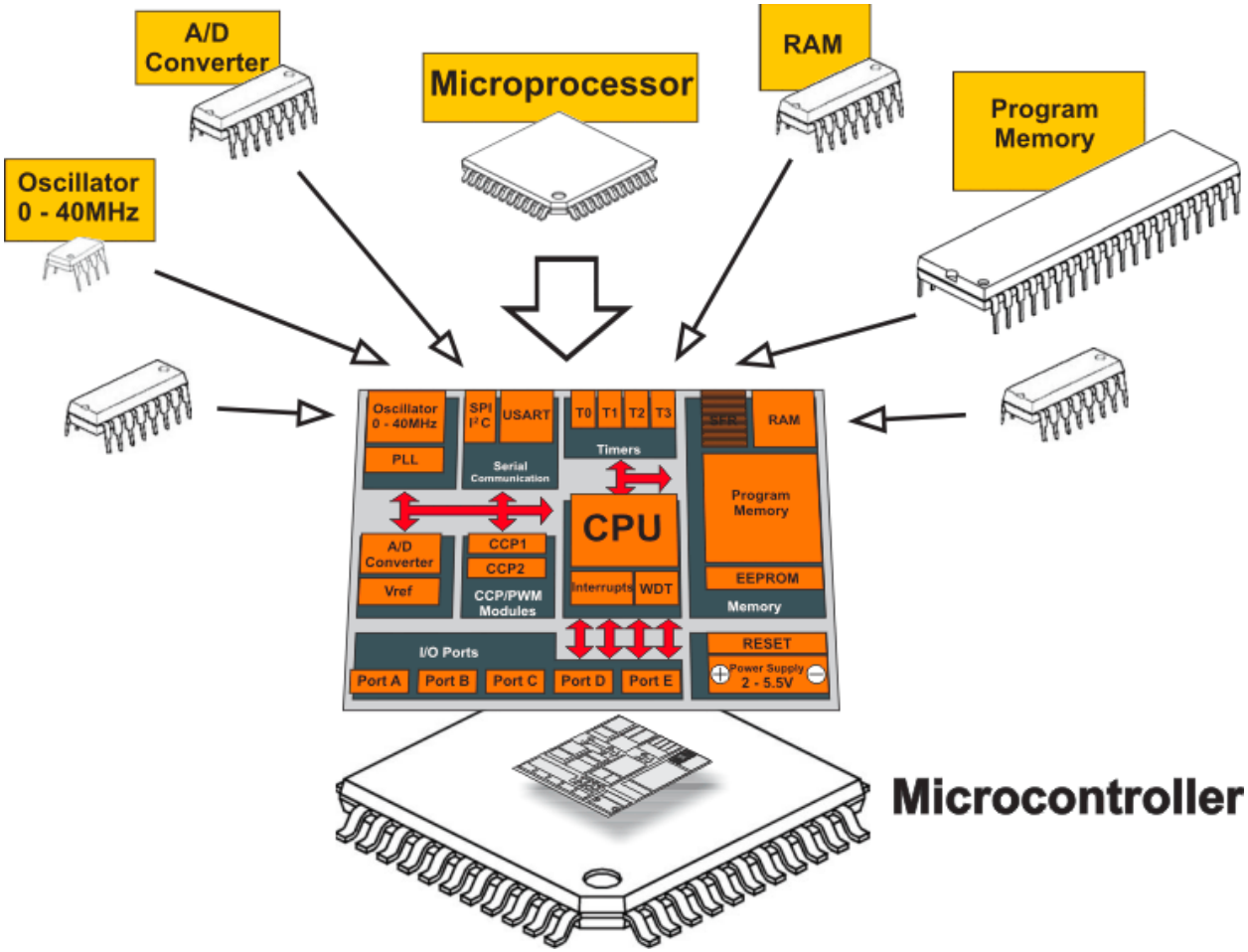
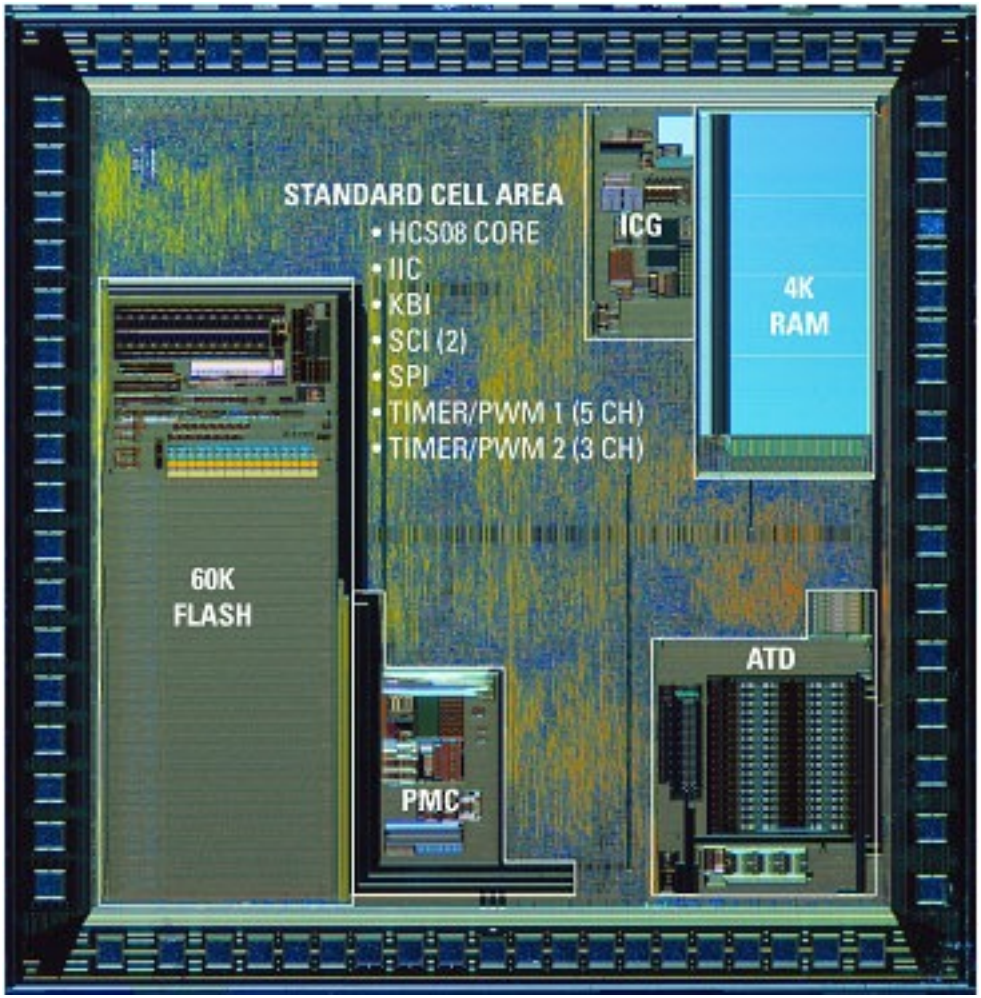


Figure 13: MTB feature in Cortex-M0+ processor provides low cost instruction trace solution

Microcontrollers are full computers integrated into a single chip



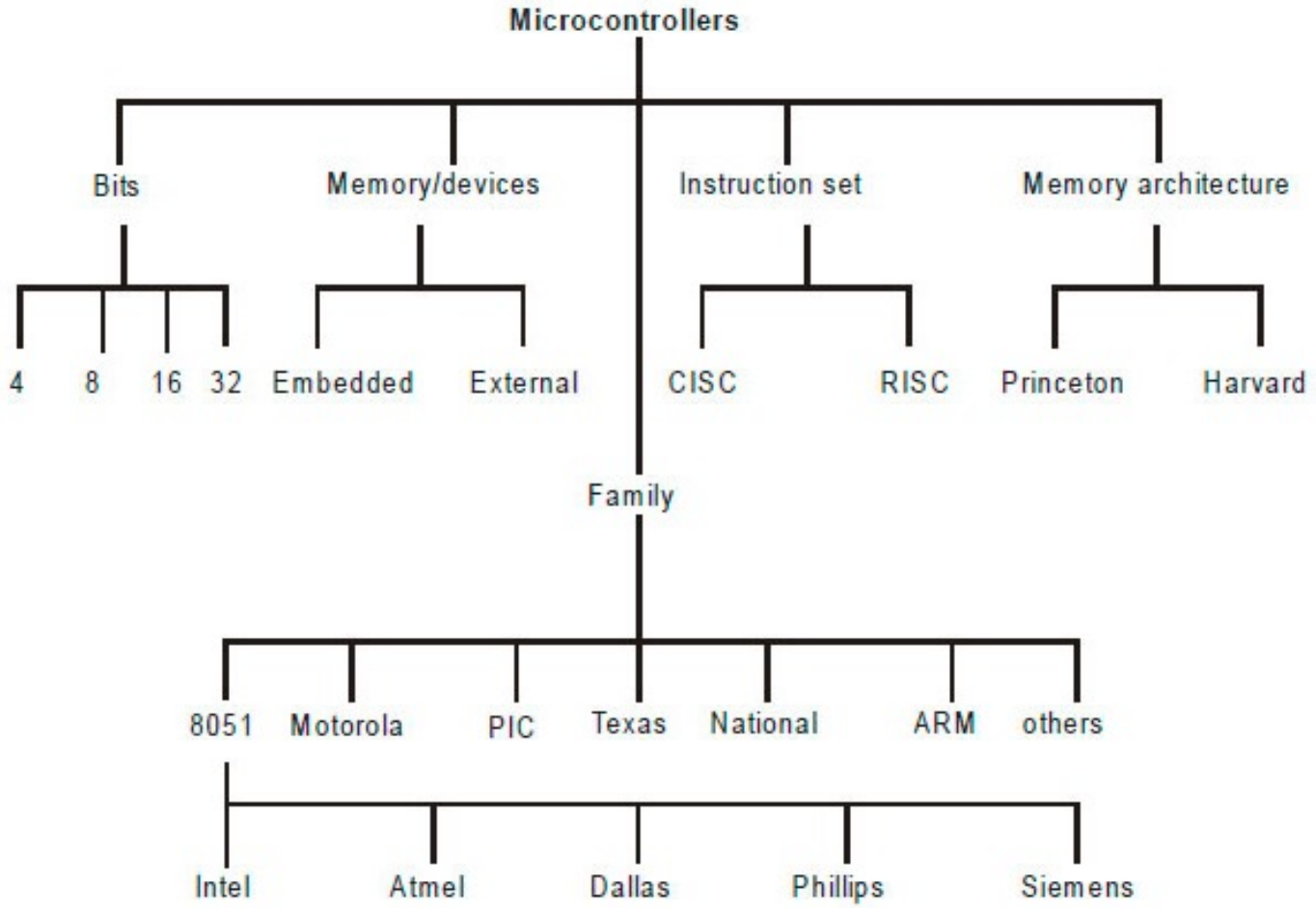
Die shot of a microcontroller



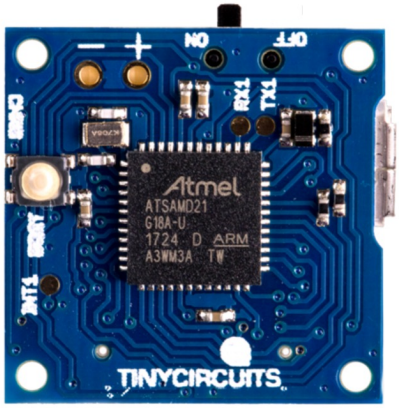
Types of Processors

- In general-purpose computing, the variety of instruction set architectures today is limited, with the Intel x86 and ARM “Thumb” ISAs overwhelmingly dominating all.
- There is no such dominance in embedded computing. On the contrary, the variety of processors can be daunting to a system designer.
- Things that matter
  - Peripherals, Concurrency & Timing, Clock Rates, Memory sizes (SRAM & flash), Package sizes

Types of Microcontrollers

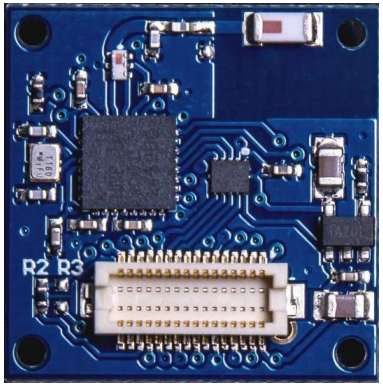


CSE190 Fall 2023



Lecture 3

MCUs (cont.)



Wireless Embedded Systems

Aaron Schulman

How to choose an MCU for a project?

- What metrics we need to consider?
  - Power consumption
  - Clock frequency
  - I/O
  - Memory
  - Internal functions
  - Size

How to choose MCU for our project?

- What metrics we need to consider?
  - Power consumption
    - E.g., we cannot afford high-power MCU because the power budget of the system requires lasting two years on one battery charge.
  - Clock frequency (speed that instructions are executed)
    - kHz is too slow…
    - 100MHz is over kill...
  - I/O
    - Lots of peripherals you can have:
      Image sensor, UART debugger, SD card, DAC, ADC, microphone, LED

How to choose MCU for our project?

- What metrics we need to consider?
  - Memory
    - We need to have sufficient memory to store:
      - Program (Non-volatile): Logic to read from sensors, communicate
      - Stack: Function calls are now expensive (no recursion)
      - Data: Constants (time periods), Sensor history, Communication state
        - » We may need non-volatile data storage for data too (e.g., Flash)
  - Performance of internal peripherals
    - E.g., Speed of copying data from the sensor to the radio (DMA)

How to choose MCU for our project?

- **Memory**
  - Store accelerometer history data
    - 12bits each for X,Y,Z acceleration (36 bits)
    - sampled 2 thousand times a second (2 KHz)
    - = 36*2,000 bits per second (72 kbit/s or 9 kByte/s)
    - How many seconds can we hold if we have only 100 kBytes of storage?
  - What types of memory are available on an MCU?
    - Internal memory: SRAM, 0.5~128 kBytes, non-volatile FRAM also available
    - External memory: Flash, high power consumption, ~5mA for read and ~10mA for erase

How to choose MCU for our project?

- **Clock frequency**
  - kHz is too slow
    - Smartphone camera frame rate is 60fps

      (1 KHz clock would leave only 60 clock cycles per frame)
  - 100MHz is too fast
    - Power consumption is high

      (power increases linearly with clock speed)
  - O(10) MHz is ideal for most embedded applications

How to choose MCU for our project?

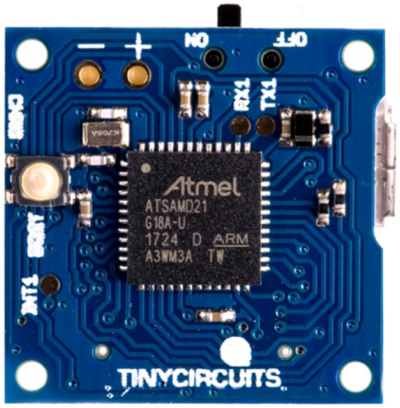
- **I/O (interface for external peripherals)**
  - Interfacing sensors, debugger, LEDs, Bluetooth radio
  - Every I/O needs physical pins on the chip
  - We often need a large number of I/O pins
  - We need various types of I/O pins

    (some pins can do more than one function)
    - Analog pins (input/output analog signals e.g., audio)
    - Digital pins (input/output digital signals e.g., busses, GPIOs)

The MCU used in our projects ($10)

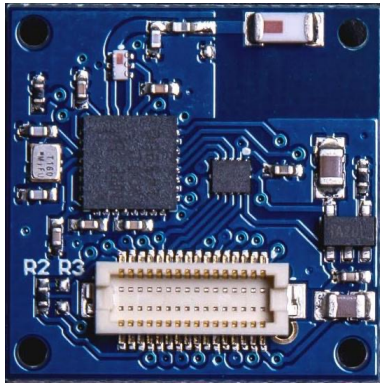
Core Processor	ARM® Cortex®-M4
Core Size	32-Bit Single-Core
Speed	80MHz
Connectivity	CANbus, EBI/EMI, I²C, IrDA, LINbus, MMC/SD, QSPI, SAI, SPI, SWPMI, UART/USART, USB OTG
Peripherals	Brown-out Detect/Reset, DMA, PWM, WDT
Number of I/O	82
Program Memory Size	**1MB (1M x 8)**
Program Memory Type	FLASH
EEPROM Size	-
RAM Size	128K x 8
/oltage - Supply (Vcc/Vdd)	1.71V ~ 3.6V
Data Converters	A/D 16x12b; D/A 2x12b
Oscillator Type	Internal
Operating Temperature	-40°C ~ 85°C (TA)
Mounting Type	Surface Mount
Package / Case	100-LQFP
Supplier Device Package	100-LQFP (14x14)
Base Product Number	**STM32L475**

CSE190 Fall 2023



Lecture 4

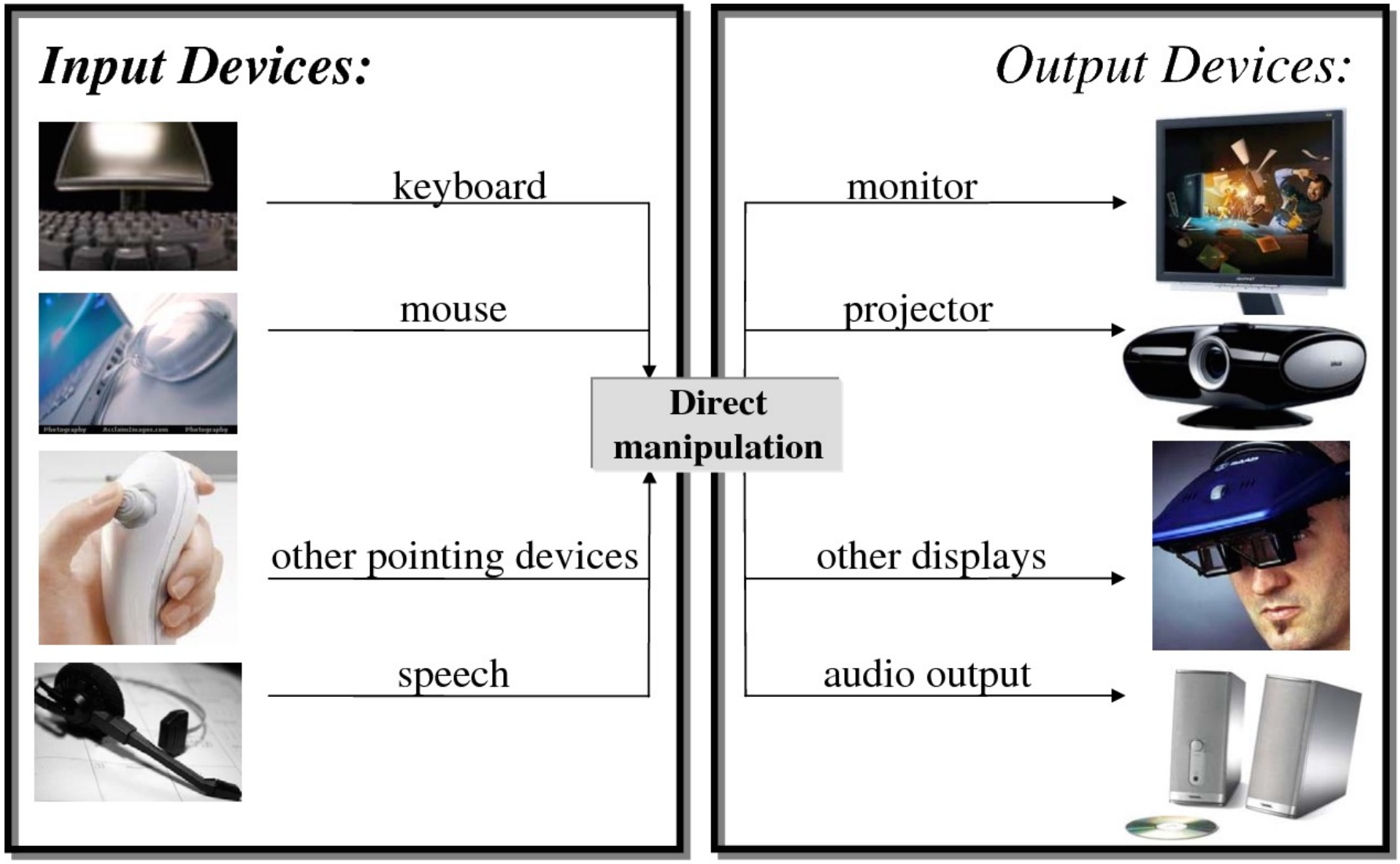
I/O



Wireless Embedded Systems

Aaron Schulman

Input and Output (I/O)



Input Peripherals are common on embedded systems (e.g., sensors)

- Keyboard, mouse, microphone, scanner, video/photo camera, etc.

- Large diversity
  - Many widely differing device types
  - Devices within each type also differs

- Speed
  - varying, often slow access & transfer compared to CPU
  - Some device-types require very fast access & transfer

- Access
  - Sequential VS random
  - read, write, read & write

What operations does software need to perform on I/O peripherals?

1. Get and set parameters
2. Receive and transmit data
3. Enable and disable functions

How can we imagine providing an interface to hardware from software?

1. Specialized CPU instructions (x86 in/out)

Port I/O

- Devices registers mapped onto “ports”; a separate address space

- Use special I/O instructions to read/write ports
- Protected by making I/O instructions available only in kernel/supervisor mode
- Used for example by IBM 360 and successors

How can we imagine providing an interface to hardware from software?

1. Specialized CPU instructions (x86 in/out)
2. Treating devices like they are memory (MMIO)

Memory Mapped IO

- Device registers mapped into regular address space

- Use regular move (assignment) instructions to read/ write a device's hardware "registers"
- Can use memory protection mechanism to protect device registers

MMIO is used for embedded systems

- Why not Ports I/O?
  - special I/O instructions would be instruction set dependent (x86, ARM Thumb, MIPS)
    - Bad if there are many different instruction sets out there
  - Need special hardware to execute and protect instructions
- Memory mapped I/O:
  - Can use all existing memory reference instructions for I/O
    - Can reuse code for reading and writing (e.g., memcpy)
  - Memory protection mechanism allows greater flexibility than protected instructions (protect specific registers)
  - Can reuse memory management / protection hardware to interface with hardware (saves space and power)

Reading and writing with MMIO is not like talking to RAM

- MMIO reads and writes hardware device registers
- Read and write to registers can cause peripherals to begin or end an operation
  - Reading is not a passive operation; it can make the hardware to do something!
  - E.g., Read to clear an interrupt flag, or to advance
  - Writing often starts operations
    E.g., Send this data over the UART bus

GPIOs are the general digital I/O device

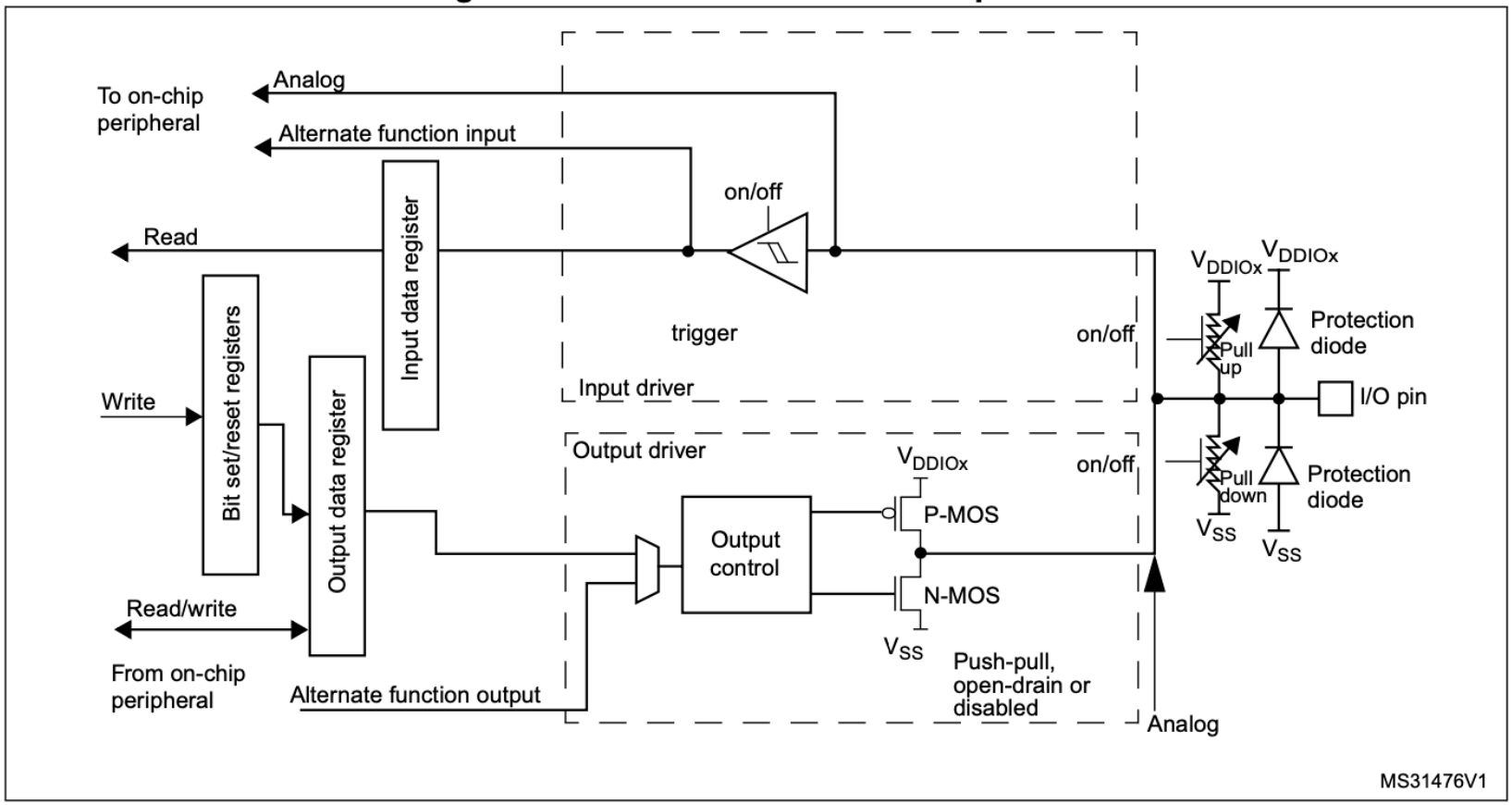
Each GPIO pin represents one bit in memory: if the pin is on it's a 1, off it's a 0

That bit can be an input or an output

- GPIOs can be used to control lights (light on or off), but even more
- Indicating that an event just happened
  - Interrupt the radio to tell it to transmit data
  - Interrupt the CPU to tell it a button was pressed
  - Read pin status to receive configuration messages
- Debugging
  - Did this one part of my code actually execute?
  - Is the timer firing at the interval that I expect it to fire (connect GPIO to oscilloscope)?
  - Why using GPIO?
    - GPIO ops are lightweight

Hardware Schematic of a GPIO pin

Figure 23. Basic structure of an I/O port bit



GPIO Output Configurations

Table 39. Port bit configuration table(1)

MODE(i) [1:0]	OTYPER(i)	OSPEED(i) [1:0]	PUPD(i) [1:0]		I/O configuration	
01	0	SPEED [1:0]	0	0	GP output	PP
	0		0	1	GP output	PP + PU
	0		1	0	GP output	PP + PD
	0		1	1	Reserved	
	1		0	0	GP output	OD
	1		0	1	GP output	OD + PU
	1		1	0	GP output	OD + PD
	1		1	1	Reserved (GP output OD)	
10	0	SPEED [1:0]	0	0	AF	PP
	0		0	1	AF	PP + PU
	0		1	0	AF	PP + PD
	0		1	1	Reserved	
	1		0	0	AF	OD
	1		0	1	AF	OD + PU
	1		1	0	AF	OD + PD
	1		1	1	Reserved	

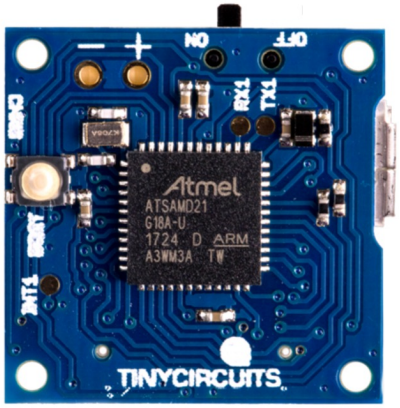
GPIO Input configuration

Table 39. Port bit configuration table[1] (continued)

MODE(i) [1:0]	OTYPER(i)	OSPEED(i) [1:0]		PUPD(i) [1:0]		I/O configuration	
00	x	x	x	0	0	Input	Floating
	x	x	x	0	1	Input	PU
	x	x	x	1	0	Input	PD
	x	x	x	1	1	Reserved (input floating)	
11	x	x	x	0	0	Input/output	Analog
	x	x	x	0	1	Reserved	
	x	x	x	1	0		
	x	x	x	1	1		

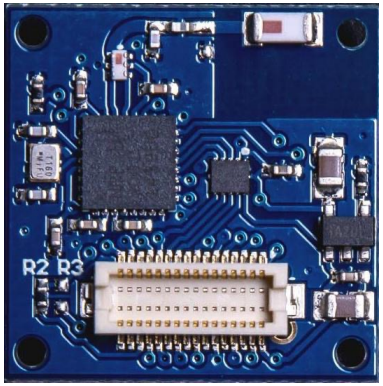
1. GP = general-purpose, PP = push-pull, PU = pull-up, PD = pull-down, OD = open-drain, AF = alternate function.

CSE190 Fall 2023



Lecture 5

Time



Wireless Embedded Systems

Aaron Schulman

What time is the Apple Watch tracking?

How often | *Granularity*

Clock (all the time | sec)
Alarm (all the time | sec)
Stopwatch (when open | msec)
Sync (all the time | sec)
UI (when open | msec)
Buzzer (when buzzing | msec)
WiFi (when communicating | usec)

Why do we need timers?

- In general, why do we need timers?
  - What time is it now?
  - How much time has elapsed since I last checked?
  - Let me know when this much time passes.
  - When did this external input occur?

What peripherals do we use to track time?

(all the time | sec) - [Alarm, Sync]

32-bit Real time clock (RTC) peripheral with interrupts

(when open/buzzing | msec) - [Stopwatch, UI, Buzzer]

Processor's *timer* peripheral with interrupts

(when communicating | usec) - [WiFi]

WiFi chip's internal timer peripheral with interrupts

What peripherals do we use to track time?

(all the time | sec) - [Alarm, Sync]

32-bit Real time clock (RTC) peripheral with interrupts

The term is used to avoid confusion with ordinary hardware clocks which are only signals that govern digital electronics, and do not count time in human units.

(when open/buzzing | msec) - [Stopwatch, UI, Buzzer]

Processor's *timer* peripheral with interrupts

(when communicating | usec) - [WiFi]

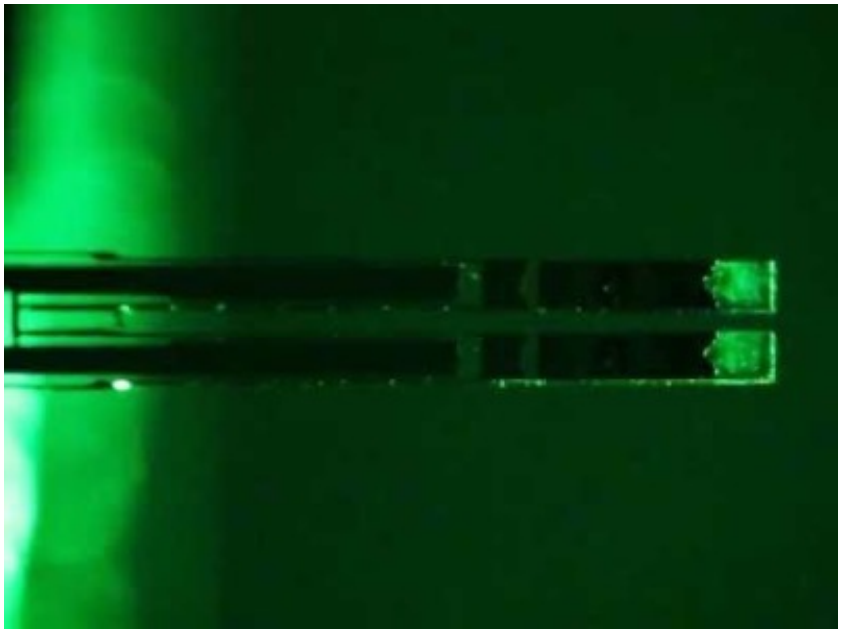
WiFi chip's internal timer peripheral with interrupts

Why do we need timers?

In the first project, what do we need timers for?

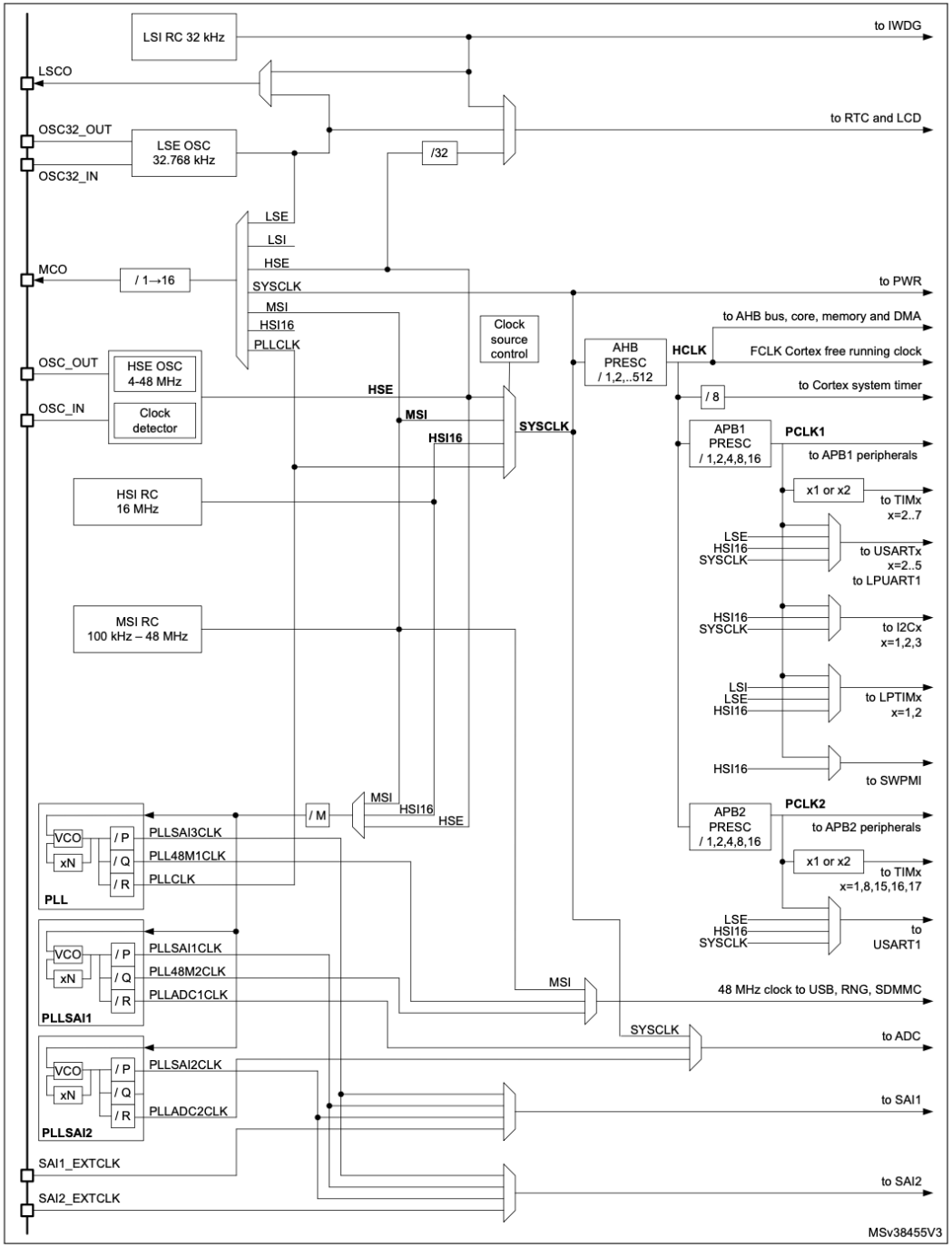
- Determining when to change LEDs
  - 20 Hz means change bits every 50 milliseconds
  - How to measure 50 ms?
  - Option 1: Use the timer hardware to let you know when 50 ms has passed.
  - Option 2: Count how many processor cycles it would take to equal 50 ms.

What is measuring the time?
Oscillators (generally, crystal oscillators)

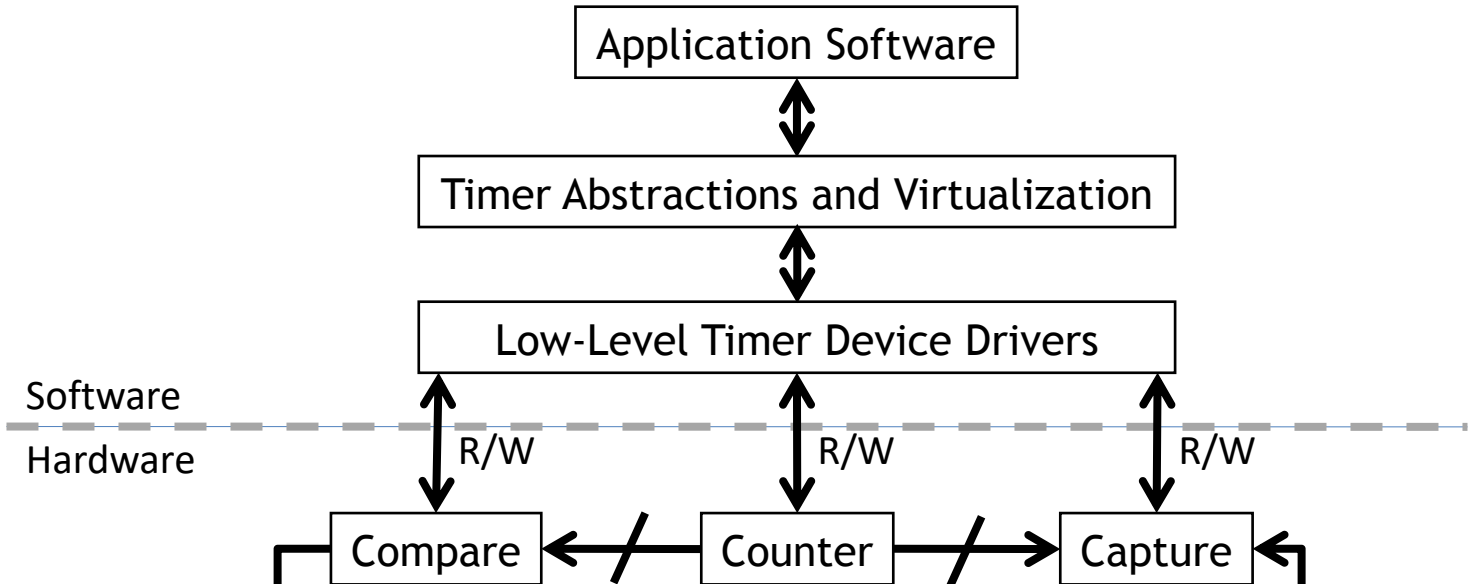


Video: Crystal oscillators "go to war"

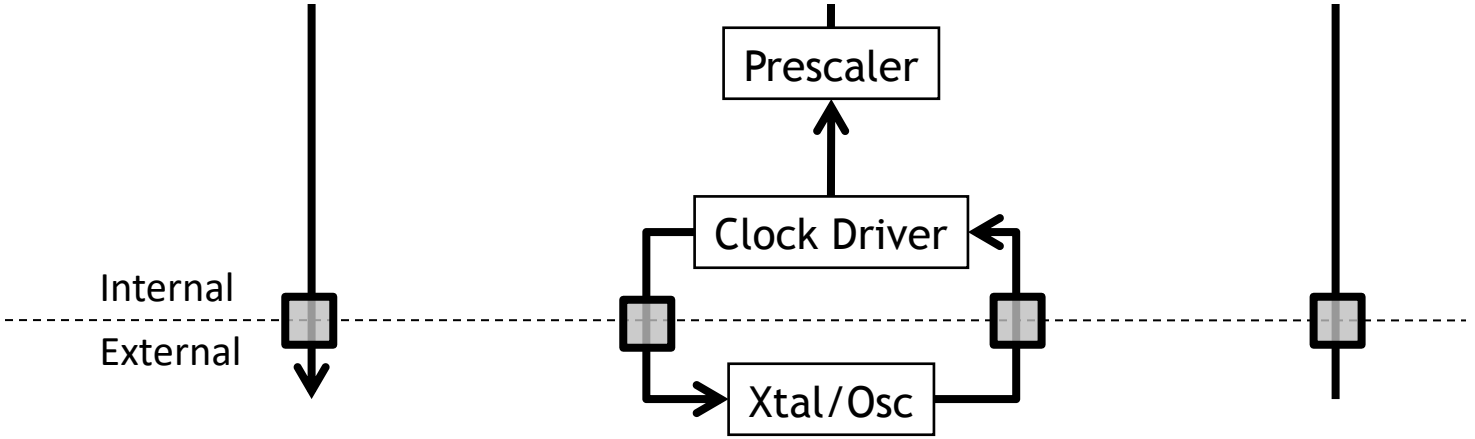
Figure 15. Clock tree (for STM32L47x/L48x devices)



Timer: A peripheral for tracking time



The purpose of the prescaler is to allow the timer to be clocked at the rate a user desires.



Frequency depends on the attached oscillator device

Figure adapted from Prabal Dutta's EE373 slides

Timers, why do we need them?

In the first project, what do we need timers for?

- Determining when to change LEDs
  - 20 Hz means change LED bits every 50 milliseconds
  - How to measure 50 ms?
  - Option 1: Use the timer hardware to let you know when 50 ms has passed.
  - Option 2: Count how many processor cycles it would take to equal 50 ms.

How does the number in the counter register correspond to wall clock time?

Frequency (Hz) = Cycles / Second

1 / Frequency (Hz) = Seconds / Cycle

The counter is incremented once per cycle.

You read 100 from the counter register which is clocked by a 1 MHz oscillator. How much time has passed since the counter was reset?

How should we choose the OSC frequency?

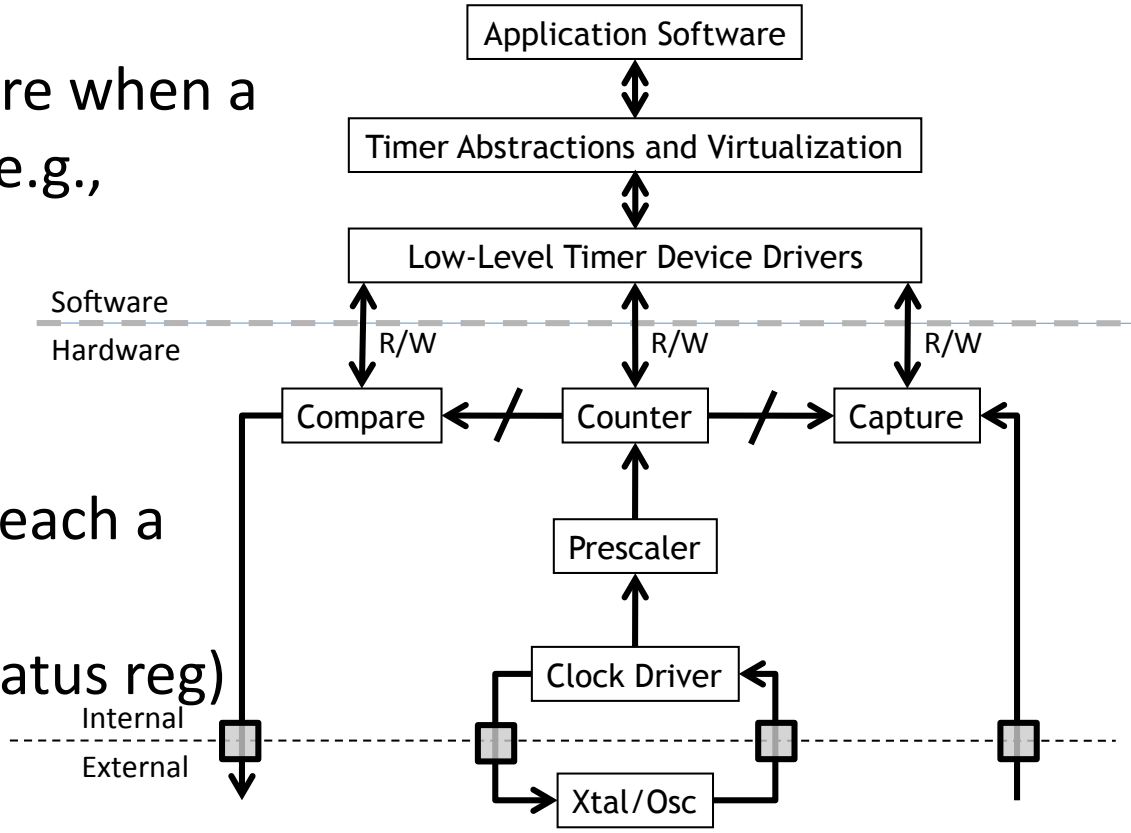
For timers, there will often be a tradeoff between resolution (high resolution requires a high clock rate) and range (high clock rates cause the timer to overflow more quickly).

1MHz OSC: resolution = 1 / 1e6 second = 1us
10MHz OSC: resolution = 1/10e6 second = 0.1us

16-bits timer:
1MHz OSC: max range = 1 / 1e6 * 2^16 = 65.536ms
10MHz OSC: max range = 1/10e6 * 2^16 = 6.5536ms

How does a firmware developer use the capture register?

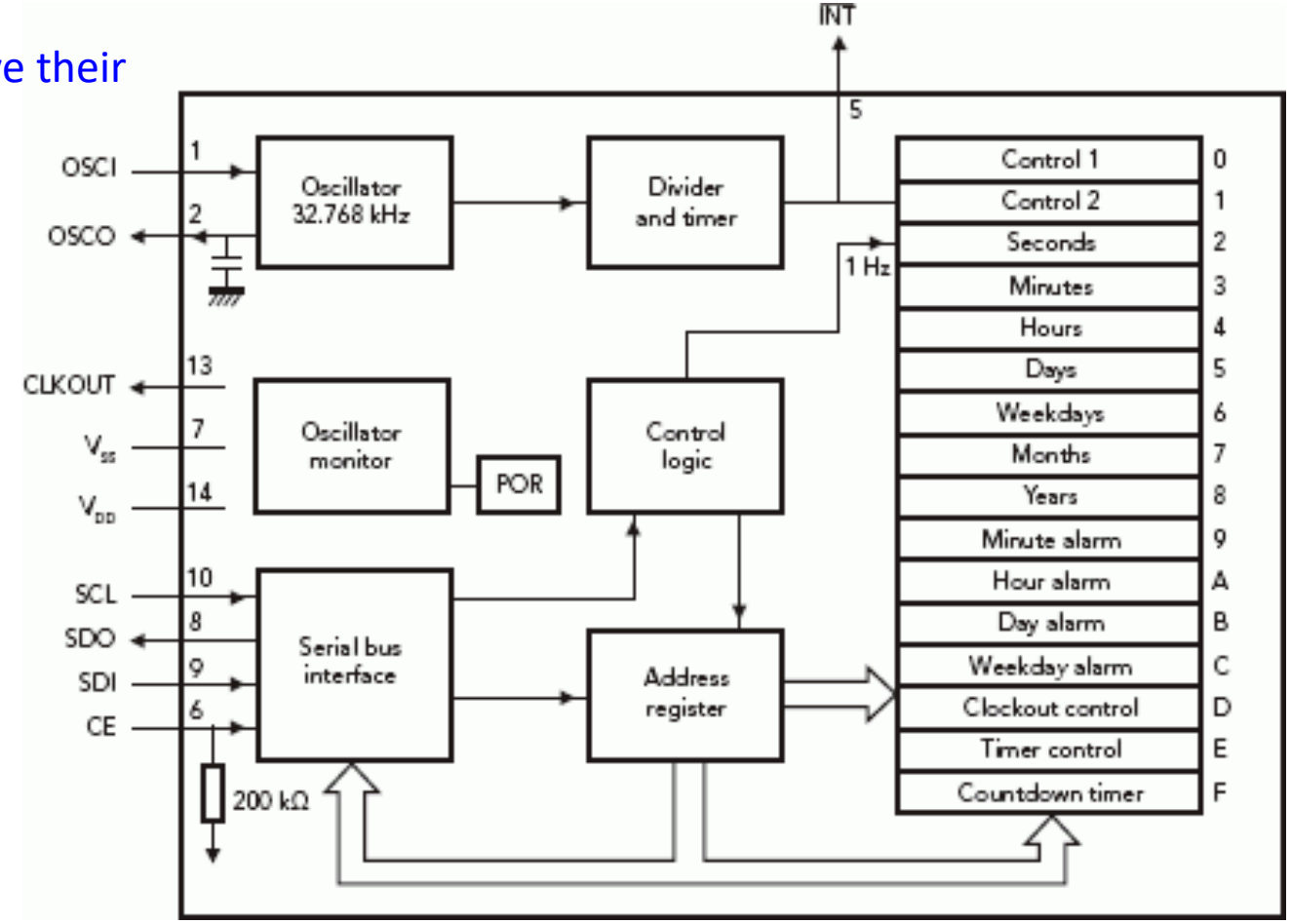
1. Stop the timer
2. Setup the timer to capture when a particular event occurs (e.g., change of GPIO pin)
3. Reset the counter
4. Start the timer
5. Wait for the counter to reach a capture event
   (via interrupt or check status reg)



A timer for keeping track of wall-clock time
Real Time Clock (RTC)

Note: RTCs have their own oscillator.

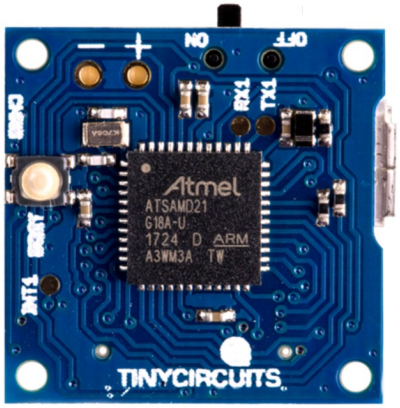
Why is it 32,768 kHz?



The reason the 32,768 Hz resonator has become so common is due to a compromise between the large physical size of low frequency crystals and the large current drain of high frequency crystals.

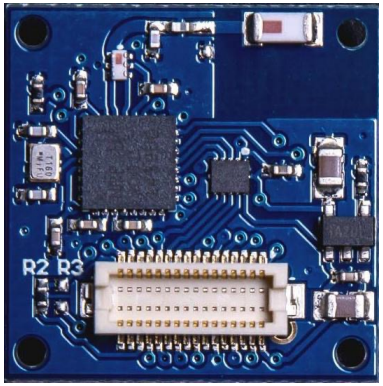
Figure adapted from Prabal Dutta's EE373 slides

CSE190 Fall 2023



Lecture 6

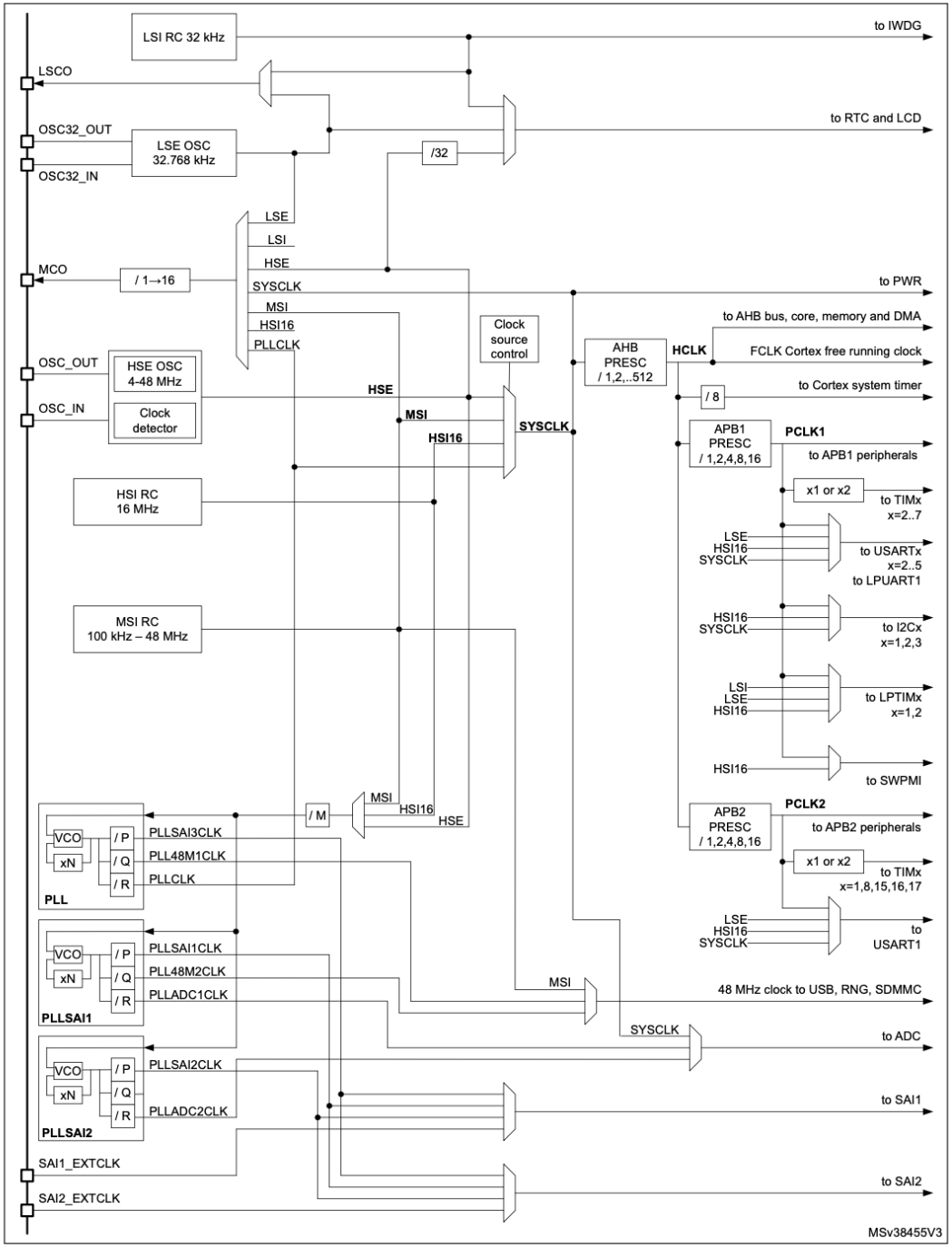
Time



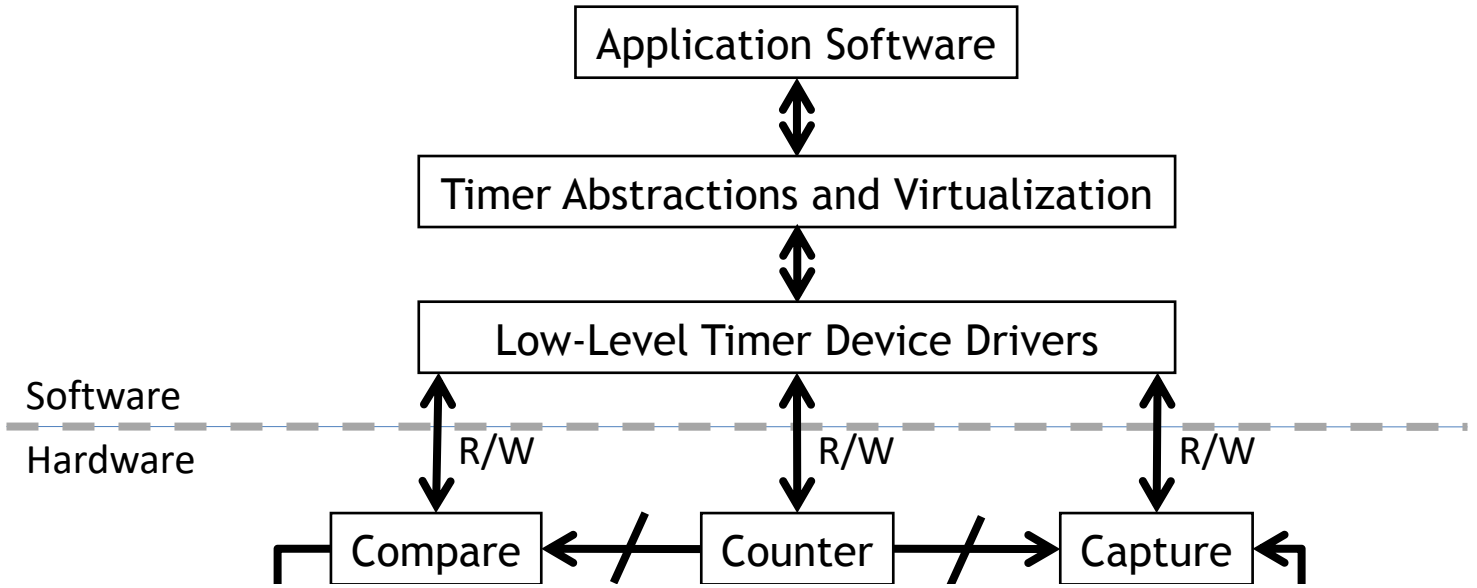
Wireless Embedded Systems

Aaron Schulman

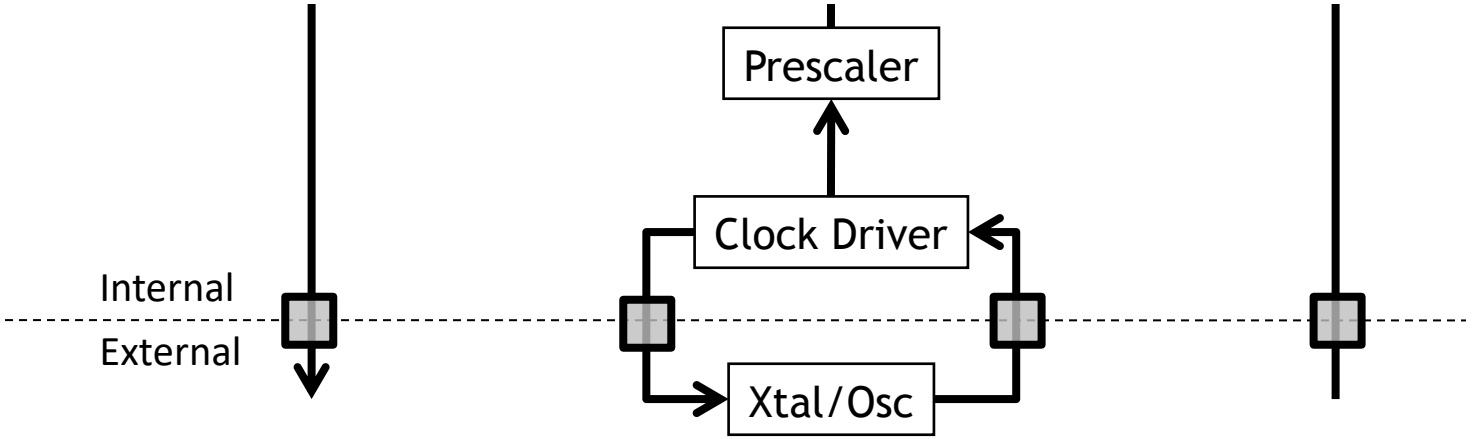
Figure 15. Clock tree (for STM32L47x/L48x devices)



Timer: A peripheral for tracking time



The purpose of the prescaler is to allow the timer to be clocked at the rate a user desires.



Frequency depends on the attached oscillator device

Figure adapted from Prabal Dutta's EE373 slides

Timers, why do we need them?

In the first project, what do we need timers for?

- Determining when to change LEDs
  - 20 Hz means change LED bits every 50 milliseconds
  - How to measure 50 ms?
  - Option 1: Use the timer hardware to let you know when 50 ms has passed.
  - Option 2: Count how many processor cycles it would take to equal 50 ms.

How does the number in the counter register correspond to wall clock time?

Frequency (Hz) = Cycles / Second

1 / Frequency (Hz) = Seconds / Cycle

The counter is incremented once per cycle.

You read 100 from the counter register which is clocked by a 1 MHz oscillator. How much time has passed since the counter was reset?

How should we choose the OSC frequency?

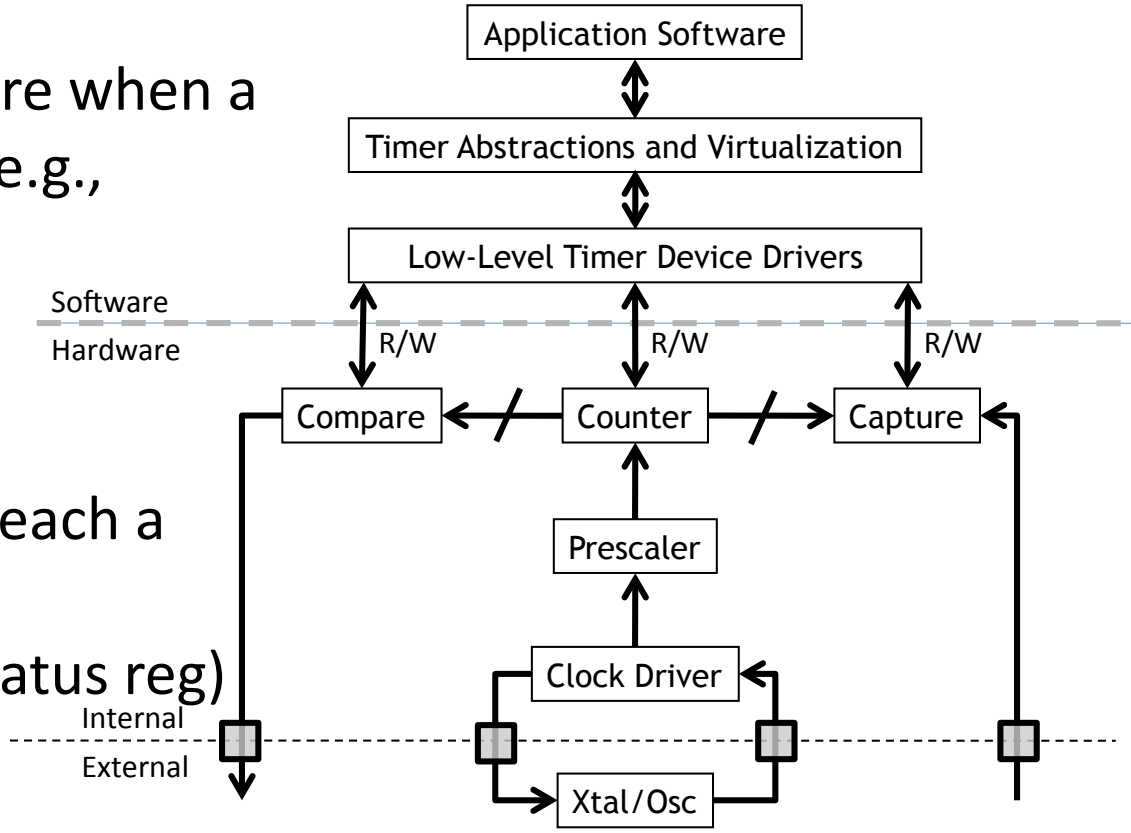
For timers, there will often be a tradeoff between resolution (high resolution requires a high clock rate) and range (high clock rates cause the timer to overflow more quickly).

1MHz OSC: resolution = 1 / 1e6 second = 1us
10MHz OSC: resolution = 1/10e6 second = 0.1us

16-bits timer:
1MHz OSC: max range = 1 / 1e6 * 2^16 = 65.536ms
10MHz OSC: max range = 1/10e6 * 2^16 = 6.5536ms

How does a firmware developer use the capture register?

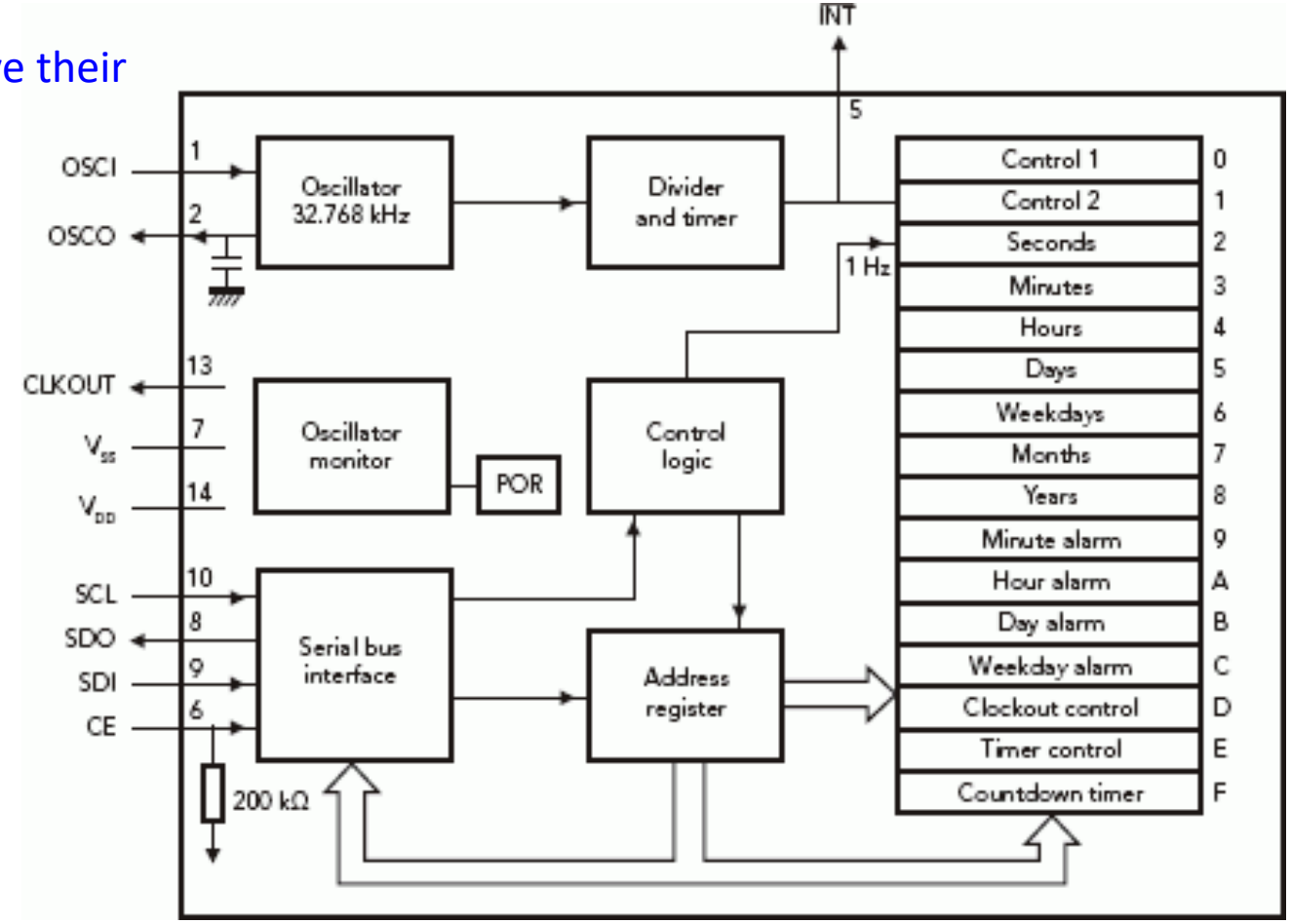
1. Stop the timer
2. Setup the timer to capture when a particular event occurs (e.g., change of GPIO pin)
3. Reset the counter
4. Start the timer
5. Wait for the counter to reach a capture event
   (via interrupt or check status reg)



A timer for keeping track of wall-clock time
Real Time Clock (RTC)

Note: RTCs have their own oscillator.

Why is it 32,768 kHz?



The reason the 32,768 Hz resonator has become so common is due to a compromise between the large physical size of low frequency crystals and the large current drain of high frequency crystals.

Figure adapted from Prabal Dutta's EE373 slides

Interrupts

How peripherals notify the CPU that an event has occurred.

Example: GPIO detected that a button was just pressed.

Interrupts

Definition

- An event external to the currently executing process that causes a change in the normal flow of software execution; usually generated by hardware peripherals, but can also be generated by the CPU.
- Key point is that interrupts are asynchronous w.r.t. current software procedure
- Typically indicate that some device needs service immediately

Why interrupts?

- MCUs have many external peripherals
  - Keyboard, mouse, screen, disk drives, scanner, printer, sound card, camera, etc.
- These devices occasionally need the CPU to act
  - But we can't predict when
- We want to keep the CPU busy (or idle for power) between these events
  - Need a way for CPU to find out when a particular peripheral needs attention

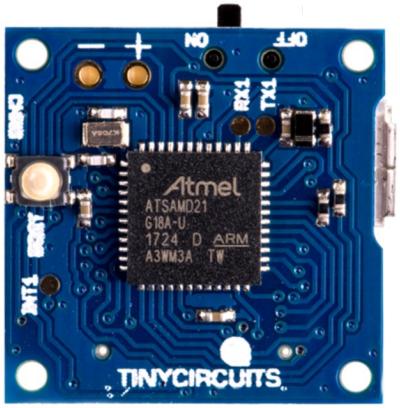
Possible Solution: Polling

- CPU periodically checks each device to see if it needs service
  - "Polling is like picking up your phone every few seconds to see if you have a call. …"

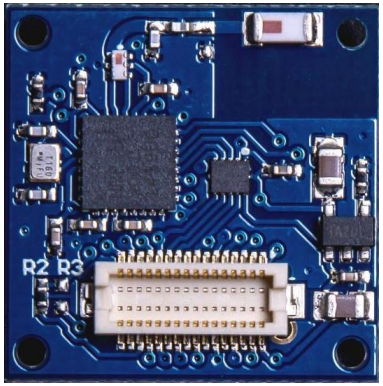
Possible Solution: Polling

- CPU periodically checks each device to see if it needs service
  - "Polling is like picking up your phone every few seconds to see if you have a call. …"
  - Cons: takes CPU time even when no requests pending
  - Pros: can be efficient if events arrive rapidly

CSE190 Fall 2023



Lecture 7
Interrupts



Wireless Embedded Systems

Aaron Schulman

Interrupts

How peripherals notify the CPU that an event has occurred.

Example: GPIO detected that a button was just pressed.

Interrupts

Definition

- An event external to the currently executing process that causes a change in the normal flow of software execution; usually generated by hardware peripherals, but can also be generated by the CPU.
- Key point is that interrupts are asynchronous w.r.t. current software procedure
- Typically indicate that some device needs service immediately

Why interrupts?

- MCUs have many external peripherals
  - Keyboard, mouse, screen, disk drives, scanner, printer, sound card, camera, etc.
- These devices occasionally need the CPU to act
  - But we can't predict when
- We want to keep the CPU busy (or idle for power) between these events
  - Need a way for CPU to find out when a particular peripheral needs attention

Possible Solution: Polling

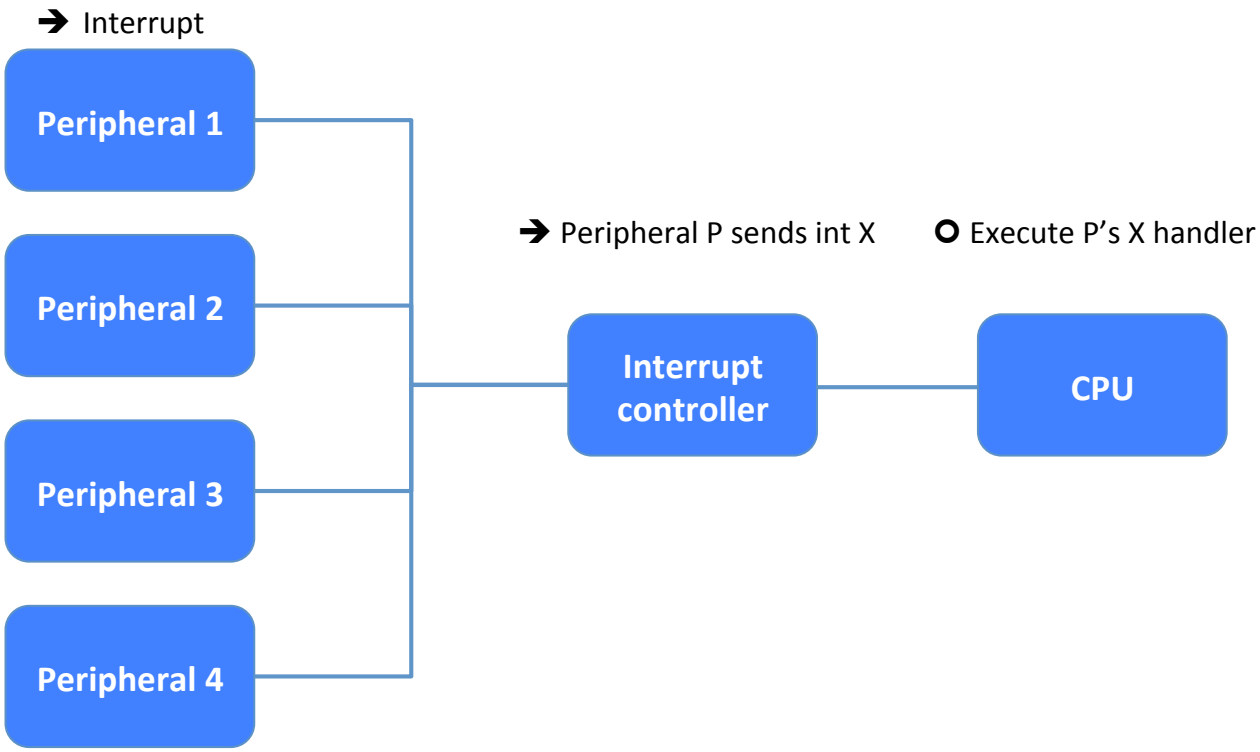
- CPU periodically checks each device to see if it needs service
  - "Polling is like picking up your phone every few seconds to see if you have a call. …"

Possible Solution: Polling

- CPU periodically checks each device to see if it needs service
  - "Polling is like picking up your phone every few seconds to see if you have a call. …"
  - Cons: takes CPU time even when no requests pending
  - Pros: can be efficient if events arrive rapidly

Alternative: Interrupts

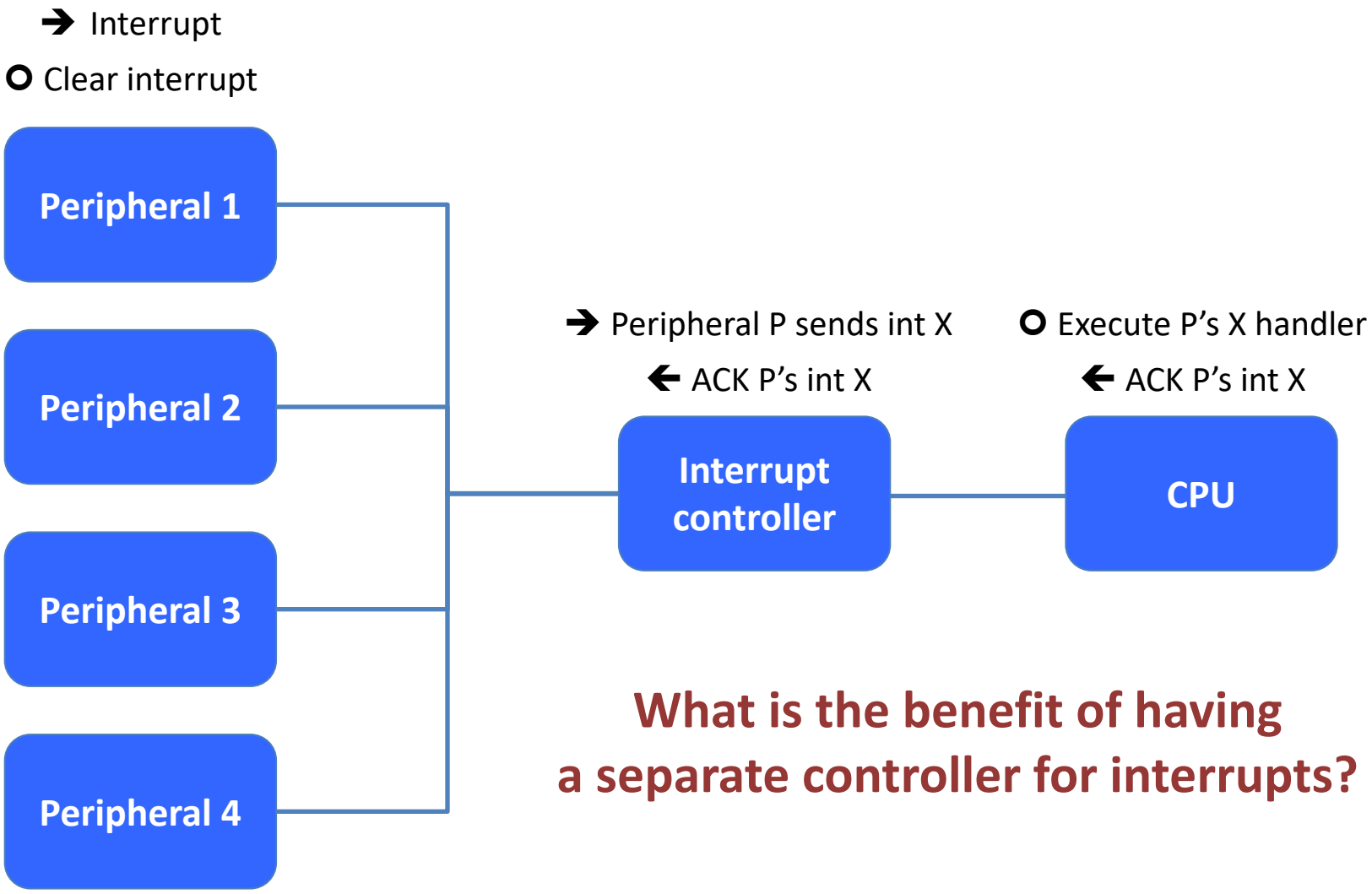
- Give each device a wire (interrupt line) that it can use to signal the processor



Alternative: Interrupts

- Give each device a wire (interrupt line) that it can use to signal the processor
  - When interrupt signaled, processor executes a routine called an interrupt handler to deal with the interrupt
  - No overhead when no requests pending

How do interrupts work?

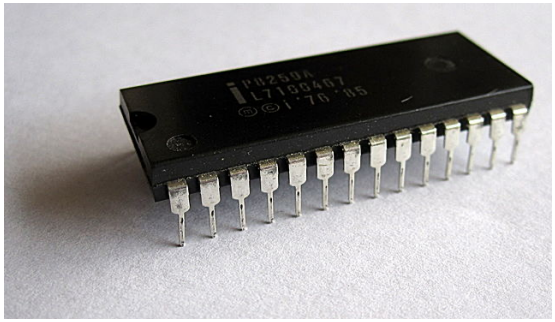


What is the benefit of having a separate controller for interrupts?

The Interrupt controller

- **Handles simultaneous interrupts**
  - Receives and sequences interrupts while the CPU handles them
- **Maintains interrupt flags**
  - CPU can poll/clear interrupt flags instead of jumping to a handler
- **Multiplexes many wires to few wires**
  - CPU doesn't need a interrupt wire to each peripheral
- This is the **NVIC (Nested Vectored Interrupt Controller)** on ARM Cortex M CPUs.

Fun fact: Interrupt controllers used to be separate chips!



Intel 8259A IRQ chip

Image by Nixdorf - Own work

CPU execution of interrupt handlers

INTERRUPT OCCURS

1. Wait for instruction to end
2. Push the program counter to the stack
3. Push all active registers to the stack
4. Jump to the interrupt handler in the interrupt vector
5. Run the interrupt handler code
6. Pop the registers off of the stack
7. Pop the program counter off of the stack

How a firmware programmer uses interrupts

1. Tell the peripheral which interrupts you want it to enable.
2. Tell the interrupt controller to enable that interrupt.
3. Tell the interrupt handler what that interrupt's priority is.
4. Tell the processor where the interrupt handler is (function address).
5. When the interrupt handler fires, do the work then clear the int flag.

How do you use your first interrupt handler in the project?

1. Setup the Timer to fire interrupt (IRQ) for TIM2 match

2. Setup NVIC to fire TIM2 interrupt

```
// Set TIM2 Interrupt Priority to Level 0
NVIC_SetPriority(TIM2_IRQn, 0);
// Enable TC3's NVIC Interrupt Line
NVIC_EnableIRQ(TIM2_IRQn);
```

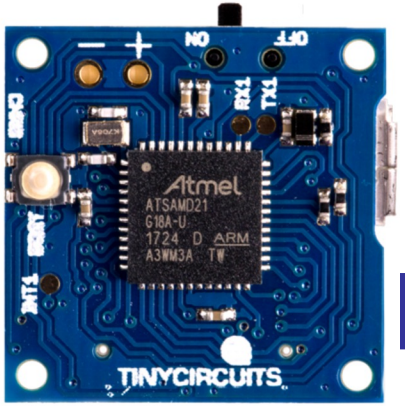
3. Write TIM2 Interrupt Handler function

```
void TIM2_IRQHandler() {
}
```

How does the CPU know to call that handler? Interrupt Vectors

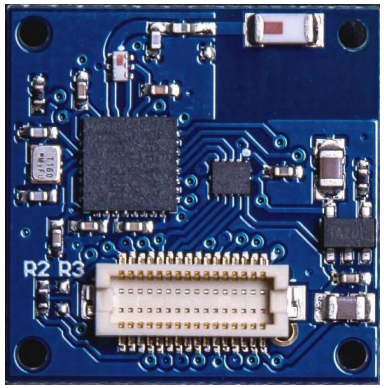
```
******************************************************************************
* @file      startup_stm32l475xx.s
* @author    MCD Application Team
* @brief     STM32L475xx devices vector table for GCC toolchain.
*            This module performs:
*                - Set the initial SP
*                - Set the initial PC == Reset_Handler,
*                - Set the vector table entries with the exceptions ISR address,
*                - Configure the clock system
*                - Branches to main in the C library (which eventually
*                  calls main()).
*            After Reset the Cortex-M4 processor is in Thread mode,
*            priority is Privileged, and the Stack is set to Main.
******************************************************************************
* @attention
*
* Copyright (c) 2017 STMicroelectronics.
* All rights reserved.
*
* This software is licensed under terms that can be found in the LICENSE file
* in the root directory of this software component.
* If no LICENSE file comes with this software, it is provided AS-IS.
*
******************************************************************************
*/
```

CSE190 Fall 2023



Lecture 8

Debugging Embedded Systems



Wireless Embedded Systems

Aaron Schulman

Qucik C development note

Volatile variables

```
volatile uint32_t a = 0;

void TIM2_IRQHandler() {
 a = 1;
}

int main() {
  if (a == 1) {
    // Will this code run?
  }
}
```

Hardware is hard.

Who do you trust when things are going wrong?

- Your software?
- Your users?
- Your test equipment?
  - Your test equipment configuration?
- The device designer?
  - The datasheet authors?
- The circuit board designer?
  - The manufacturer and assembler?
- …
- Physics?

Data sheet errors are common

Which one of these is correct?
SAM21 data sheet in 2017, and 2018 for I2C

2017

28.10.9 Address

Name: ADDR
Offset: 0x24
Reset: 0x0000
Property: Write-Synchronized

Bit	31	30	29	28	27	26	25	24
Access								
Reset								

Bit	23	22	21	20	19	18	17	16
	LEN[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8
	TENBITEN	HS	LENEN			ADDR[2:0]		
Access	R/W	R/W	R/W			R/W	R/W	R/W
Reset	0	0	0			0	0	0

Bit	7	6	5	4	3	2	1	0
Access								
Reset								

2018

28.10.9 Address

Name: ADDR
Offset: 0x24
Reset: 0x0000
Property: Write-Synchronized

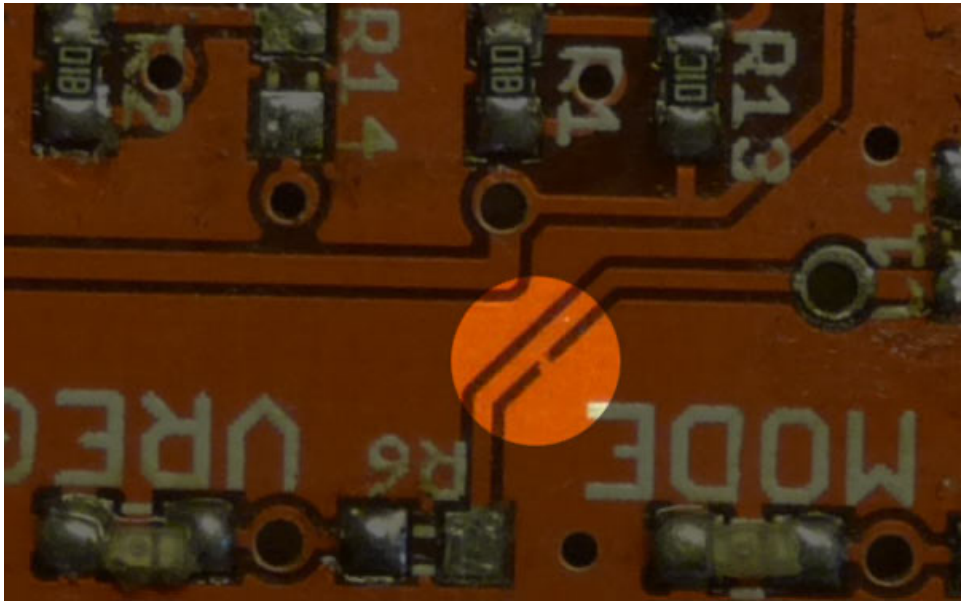
Bit	31	30	29	28	27	26	25	24
Access								
Reset								

Bit	23	22	21	20	19	18	17	16
	LEN[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

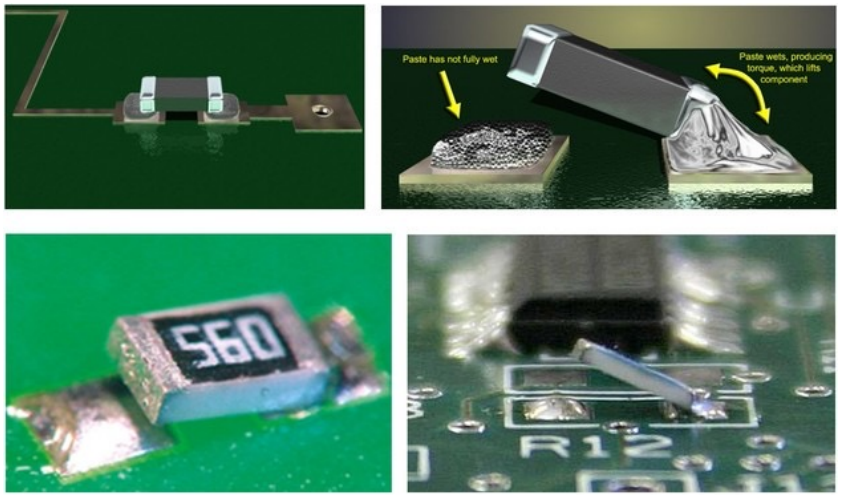
Bit	15	14	13	12	11	10	9	8
	TENBITEN	HS	LENEN			ADDR[10:8]		
Access	R/W	R/W	R/W			R/W	R/W	R/W
Reset	0	0	0			0	0	0

Bit	7	6	5	4	3	2	1	0
	ADDR[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Manufacturing errors are common



Circuit board manufacturing fault (signal shorted to GND)



PCB assembly fault (“tombstoning”) resistors

When all else fails, the bootloader can save the day!

When your software has a bug in it that causes the processor to hang or crash, you can't do anything!

- Programming over USB requires the processor to be responsive (a reset message is sent over USB at the beginning of the programming)
- The bootloader is a small bit of code run at boot time that doesn't get overwritten
  - It has just enough code in it to reflash over USB
  - Normally it just jumps into your code, but you can force it to wait for a program flash switching the power on while you hold the reset button.

https://www.st.com/resource/en/application_note/cd00167594-stm32-microcontroller-system-memory-boot-mode-stmicroelectronics.pdf

Even one LED can be critical for debugging

Think of an LED as a breakpoint (very efficient!)

- Turn it on before a line of code
  - Lets you know you reached that line
- Turn it off after a line of code
  - Lets you know you exited that code segment
- Toggle it each time a repeated event is called
  - This can show you visually how often the event happens

Need some way to introspect code: printf() is essential

Think of prints like tracing + inspection (not efficient)

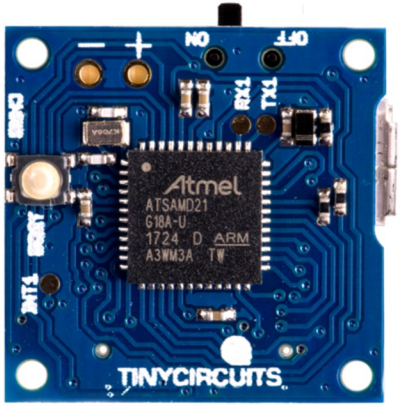
- Print variables you need to inspect
  - Register states (flags, sensor readings)
- Print when a specific function is run
  - Lets you know the order functions are run
  - Can you use printf for interrupt handlers?
  - printf("Currently inside function: %s\n", __func__);

Hardware debugger pro/cons

Ok for development, impossible after deployment

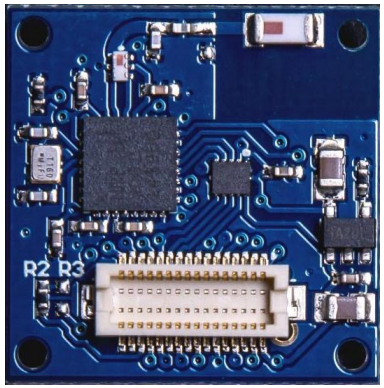
- Excellent tool for building/testing new drivers
- Lets you pause/inspect memory of CPU
- Does not pause peripherals, they keep running
  - MMIO inspection will be current peripheral state

CSE190 Fall 2023



Lecture 9

Serial Busses

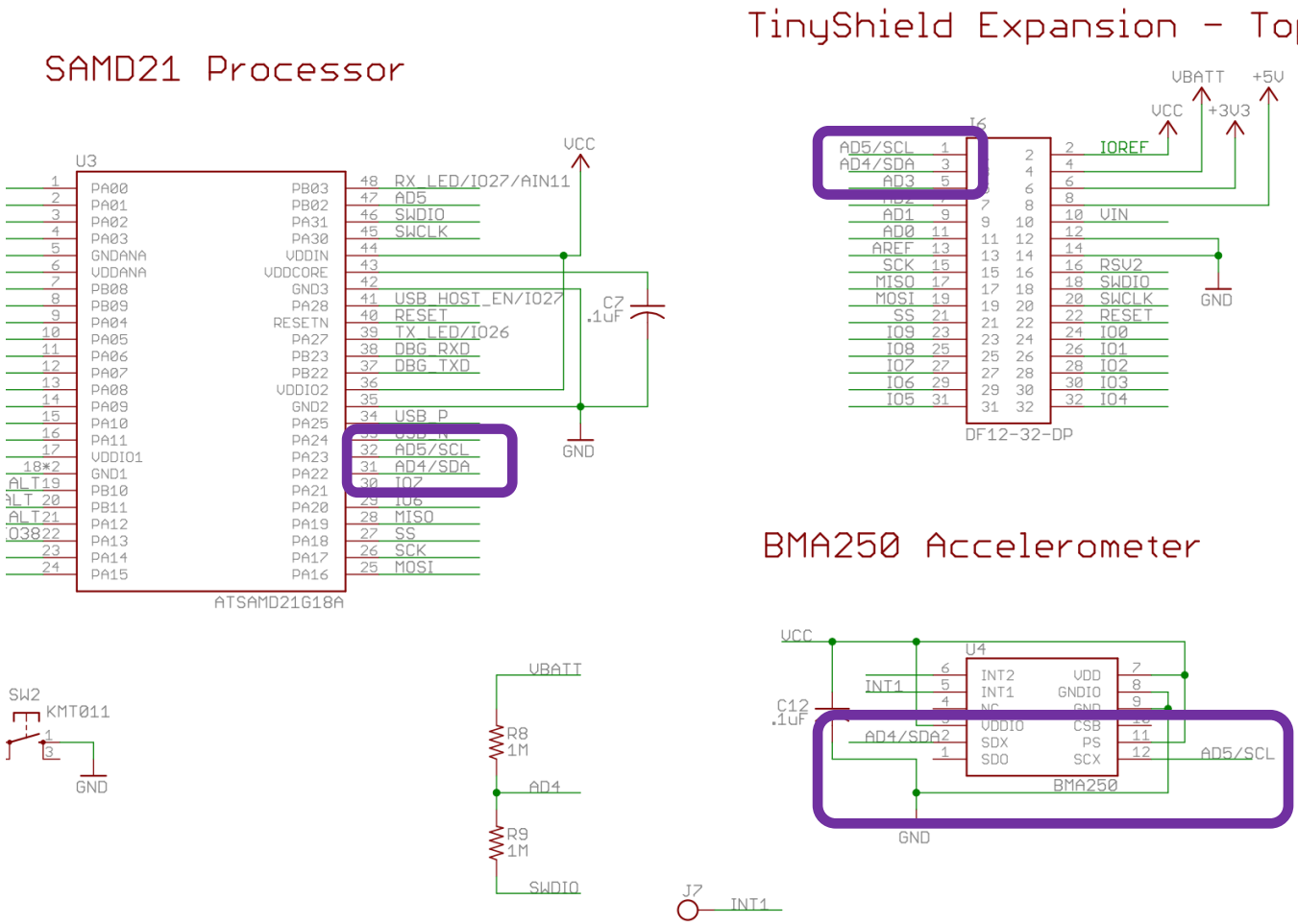


Wireless Embedded Systems

Aaron Schulman

Serial Busses

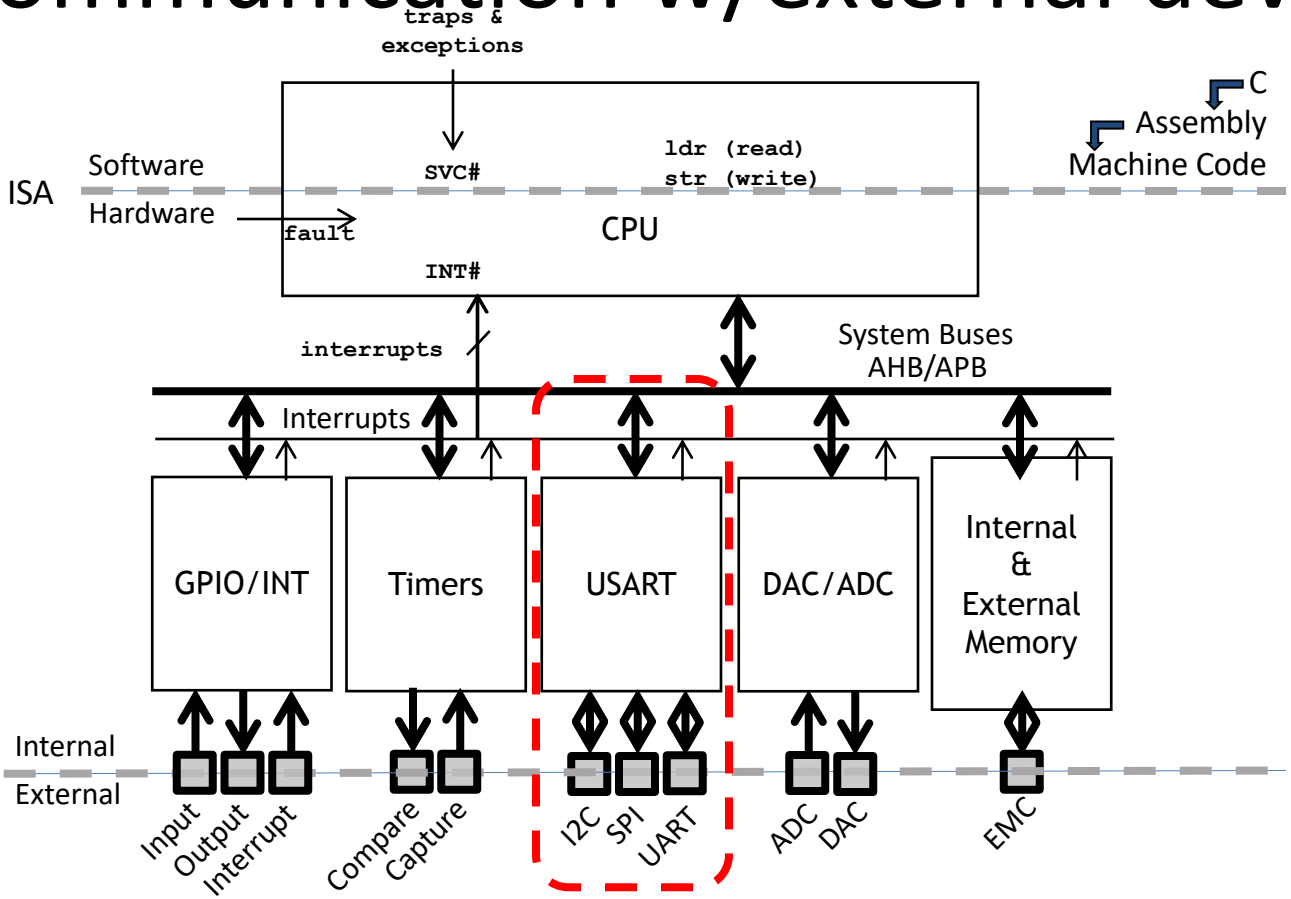
Digital data highways that *external peripherals* use to communicate with microcontrollers



Serial Buses in our project

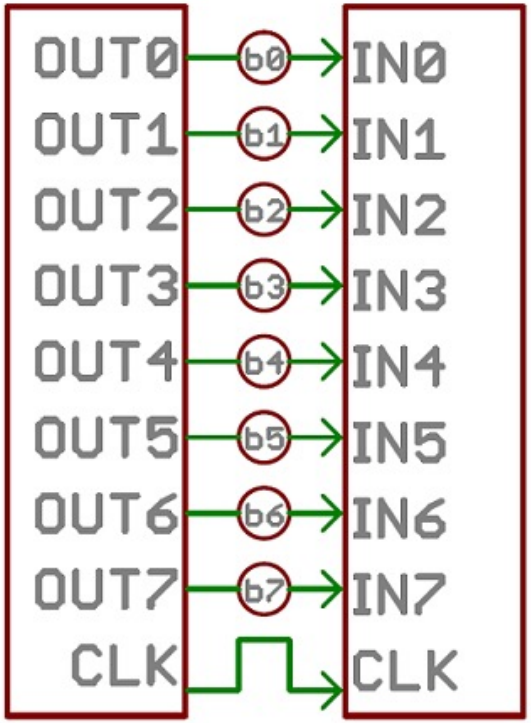
- UART serial bus for sending debug messages to your development host
- I2C serial bus for communicating with sensors (e.g., the accelerometer)
- SPI serial bus for communicating with the Bluetooth Low Energy radio

We use an internal peripheral for serial communication w/external devices

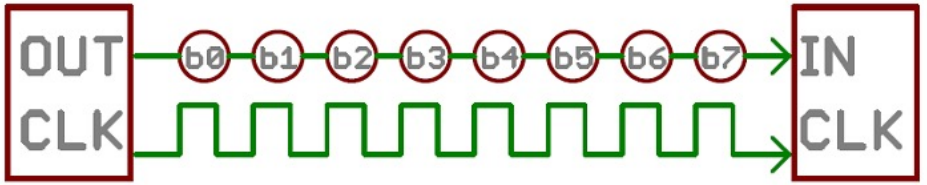


Parallel Bus vs Serial Bus

Parallel

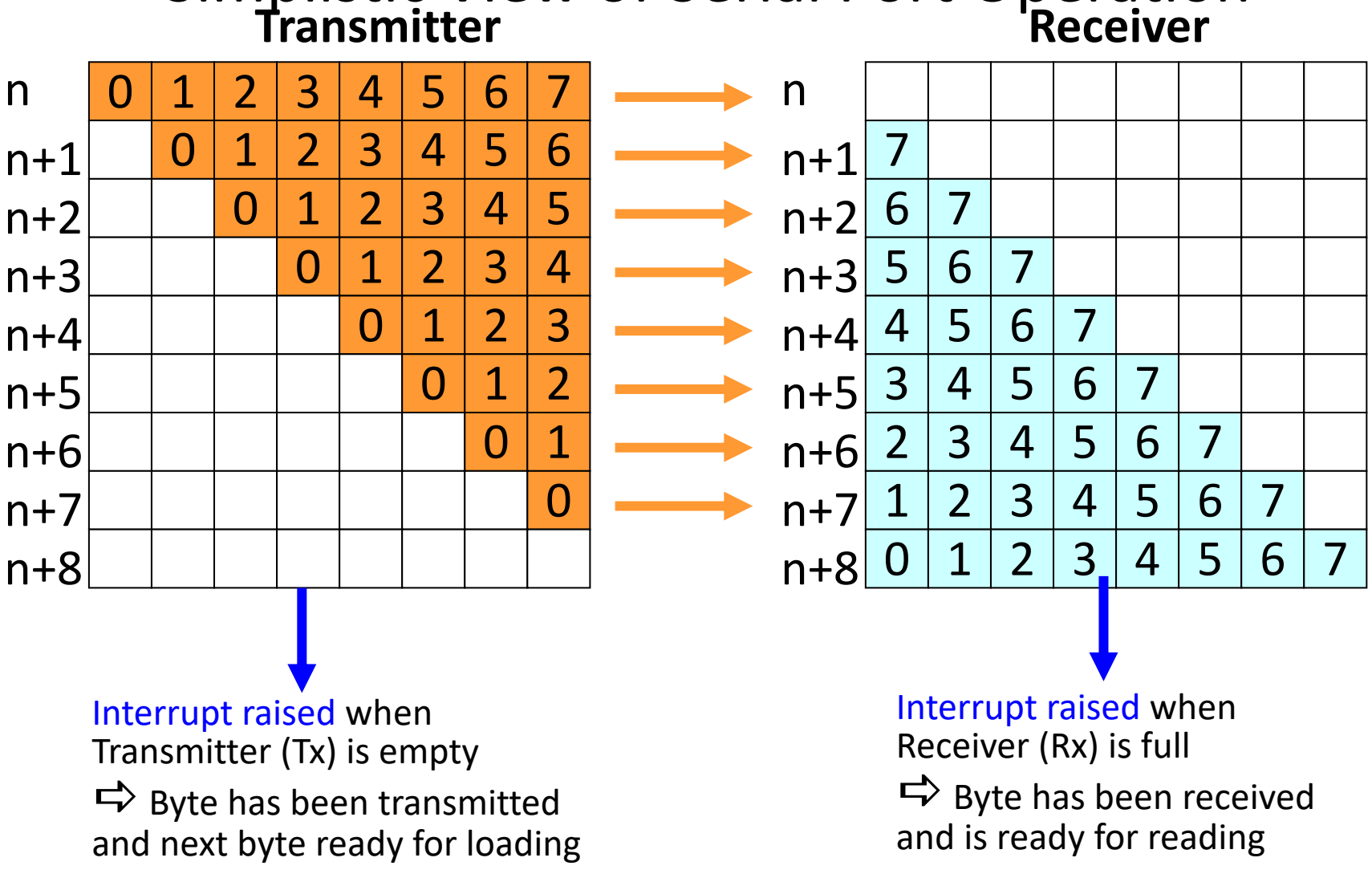


Serial



What is the benefit of a serial bus over a parallel bus (and vice versa)?

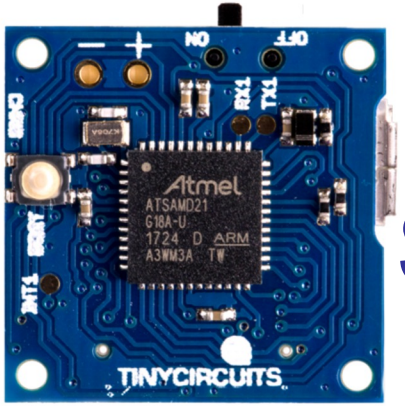
Simplistic View of Serial Port Operation



Serial Bus Interface Motivations

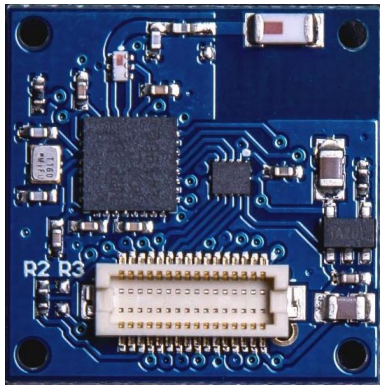
- Motivation
  - Without using a lot of I/O lines
    - I/O lines require I/O pads which cost $$$ and size
    - I/O lines require PCB area which costs $$$ and size
  - Connect different systems together
    - Two embedded systems
    - A desktop and an embedded system
  - Connect different chips together in the same embedded system
    - MCU to peripheral
    - MCU to MCU
  - Often at relatively low data rates
  - But sometimes at higher data rates
- So, what are our options?
  - Universal Synchronous/Asynchronous Receiver Transmitter
  - Also known as USART (pronounced: “you-sart”)

CSE190 Fall 2023



Lecture 10

Serial Busses (cont)



Wireless Embedded Systems

Aaron Schulman

Serial Bus Design Space

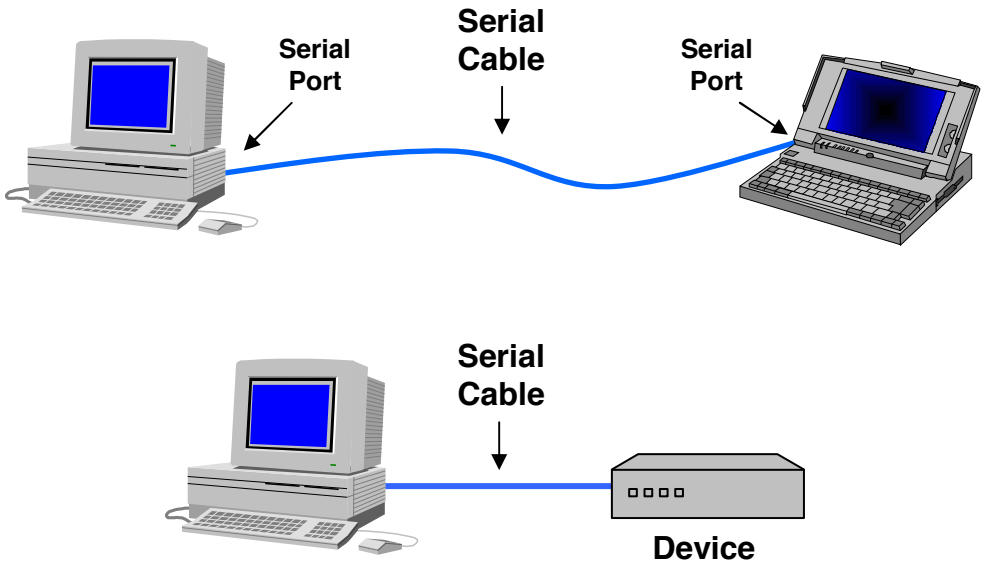
- Number of wires required?
- Asynchronous or synchronous clocking of bits?
- How fast can it transfer data?
- Can it support more than two endpoints?
- Can it support more than one primary?
- How do we support flow control?
- How does it handle errors/noise?
- How far can signals travel?

Serial Bus Examples

	S/A	Type	Duplex	#Devices	Speed (kbps)	Distance (ft)	Wires
RS232	A	Peer	Full	2	20	30	2+
RS422	A	Multi-drop	Half	10	10000	4000	1+
RS485	A	Multi-point	Half	32	10000	4000	2
I2C	S	Multi-primary	Half	?	3400	<10	2
SPI	S	Multi-primary	Full	?	>1000	<10	3+
Microwire	S	Peer	Full	?	>625	<10	3+
1-Wire	A	Peer	half	?	16	1000	1+

UART Uses

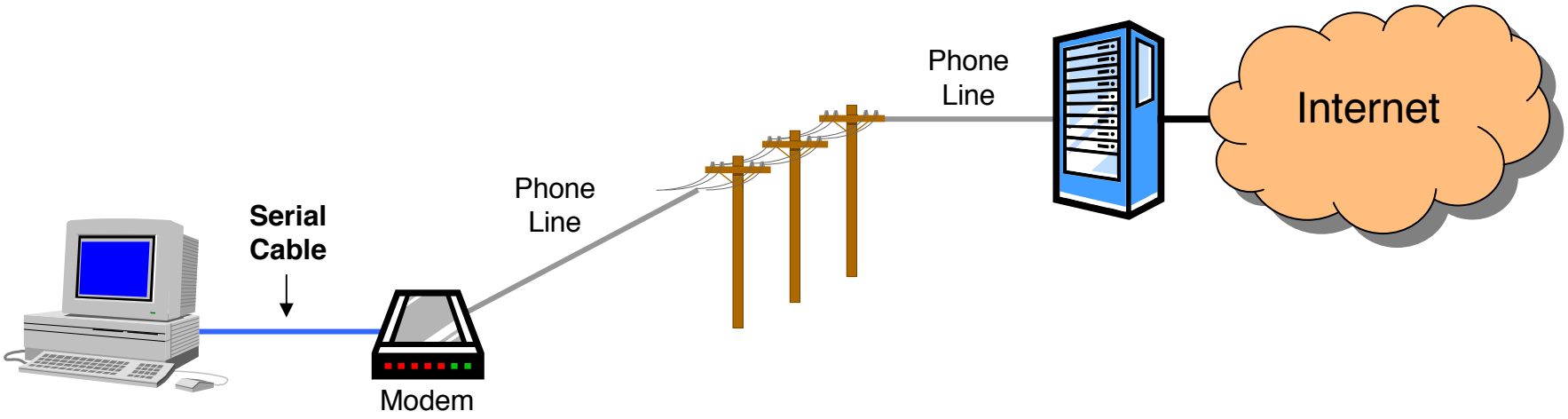
- The PC serial port is a UART!
- Serializes data to be sent over a serial cable
  - De-serializes received data



Slides from BYU CS 224

UART Uses

Used to be commonly used for internet access

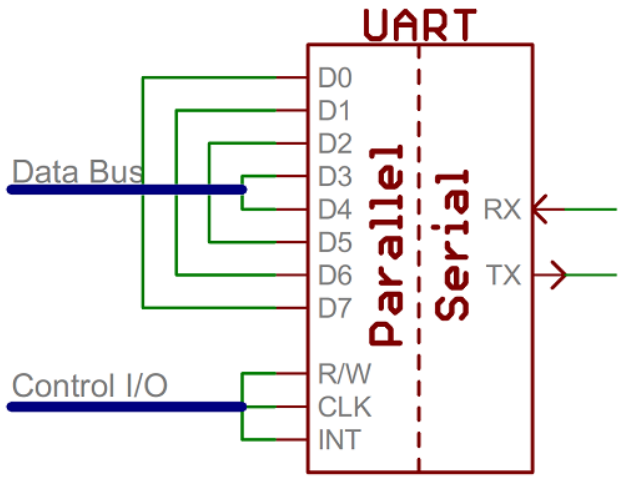


Now often used for **debugging interfaces** on embedded systems (as a debugging terminal)

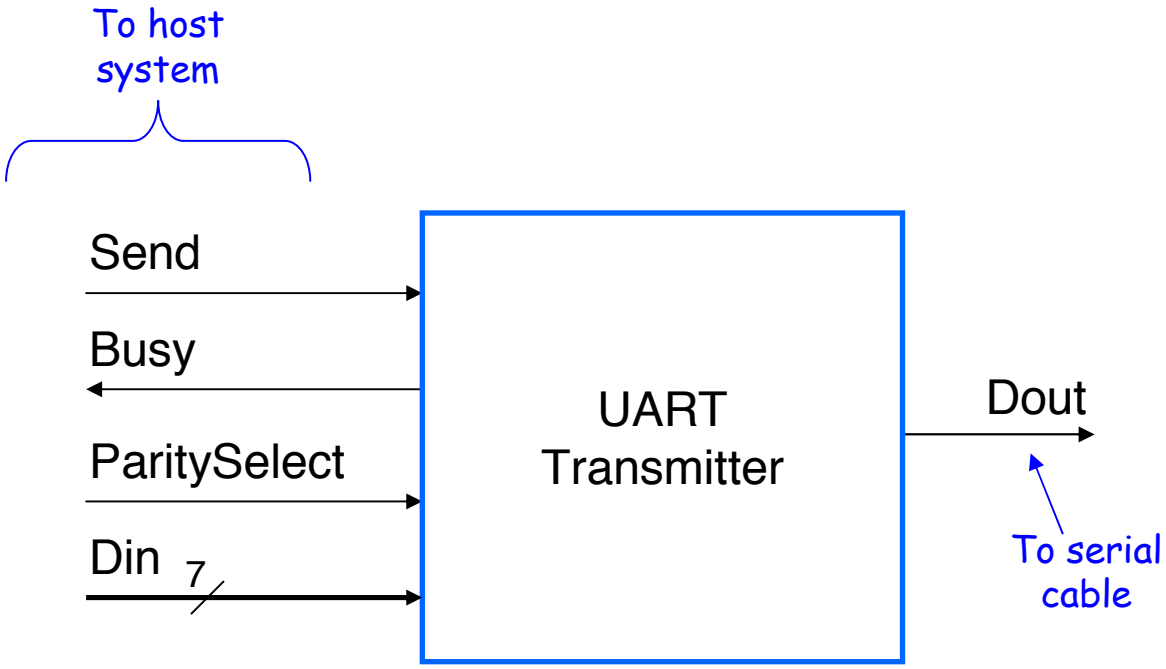
Slides from BYU CS 224

UART

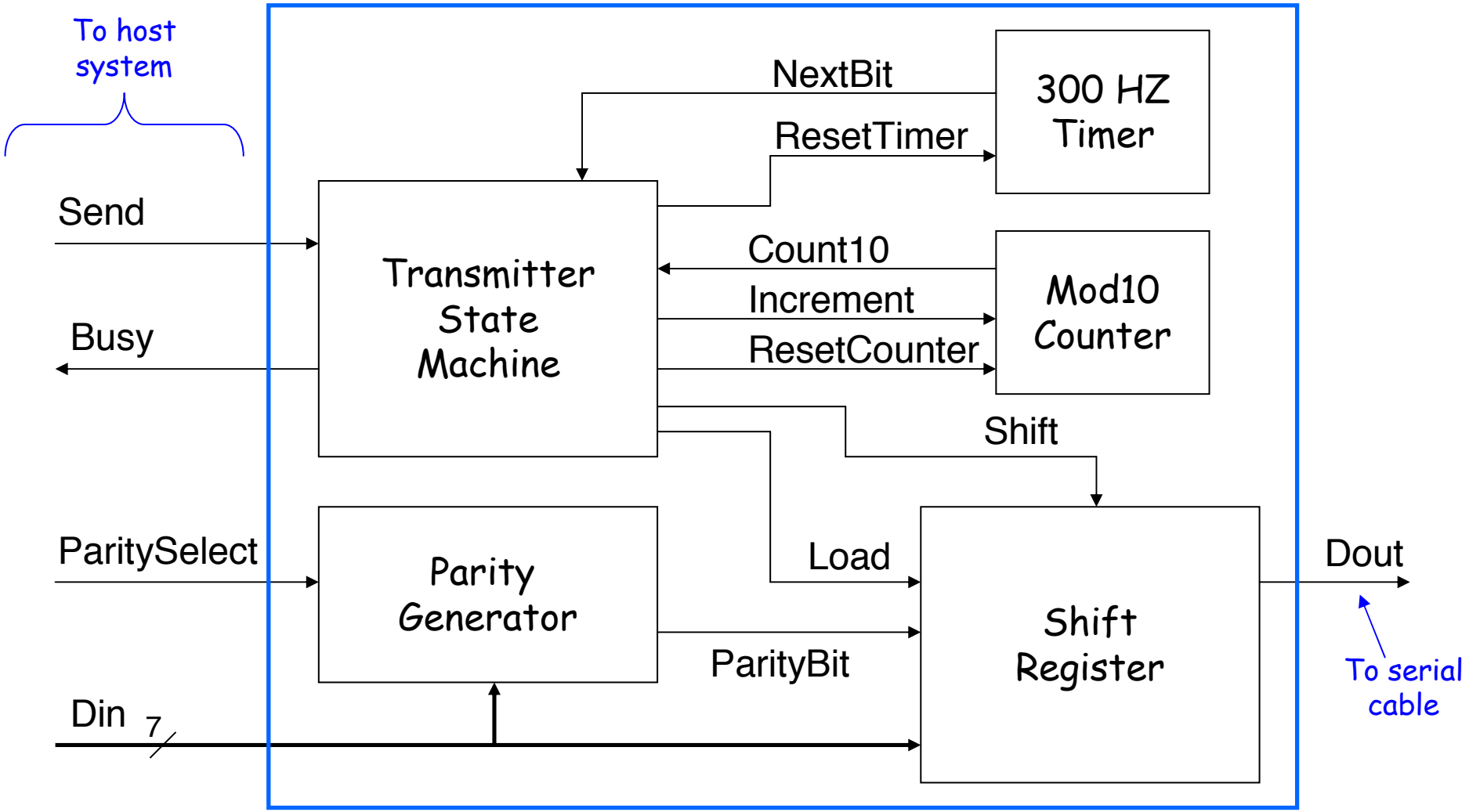
- Universal Asynchronous Receiver/Transmitter
- Hardware that translates between parallel and serial forms
- Commonly used in conjunction with communication standards such as EIA, RS-232, RS-422 or RS-485



Let us design a UART transmitter



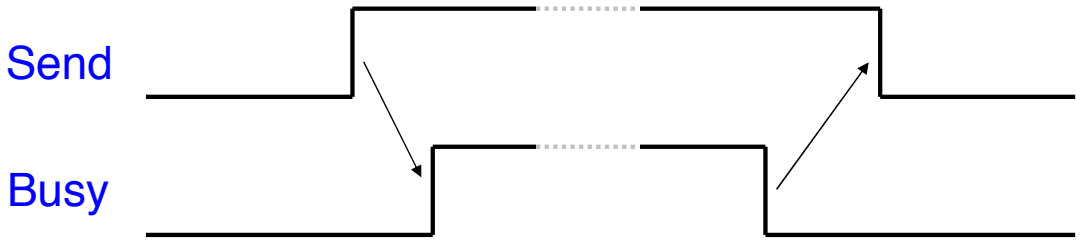
UART Transmitter Block Diagram



Slides from BYU CS 224

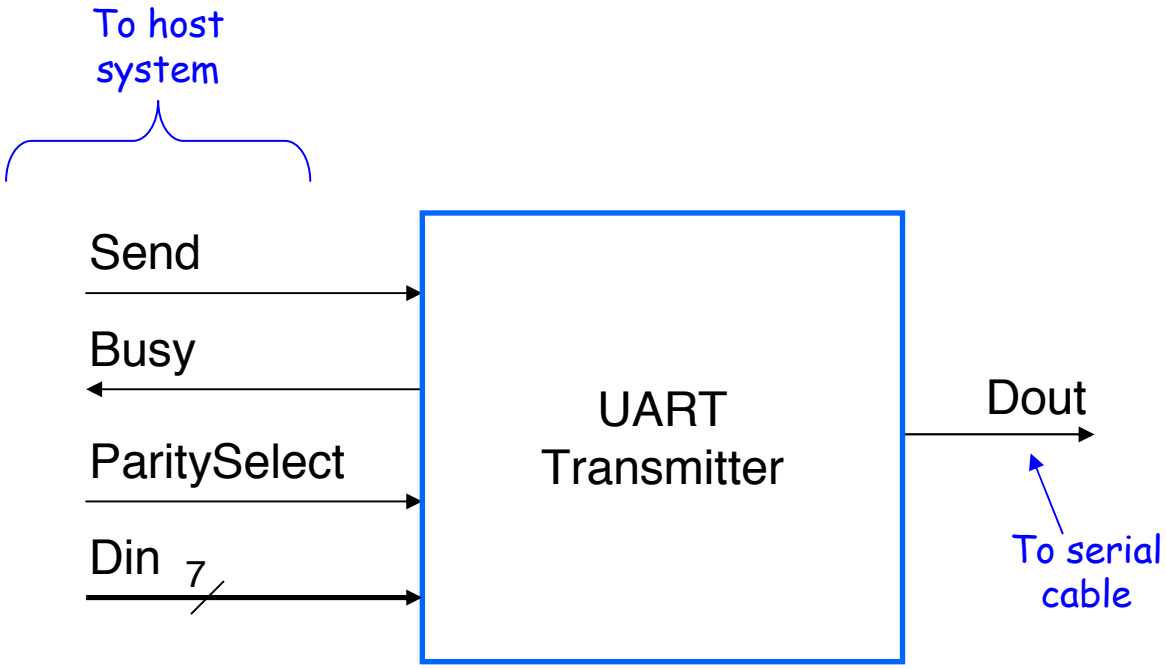
Transmitter/Microcontroller Handshaking

- Microcontroller asserts Send flag and holds it high when it wants to send a byte
- UART asserts Busy flag in response
- When UART has finished transfer, UART de-asserts Busy flag
- (sometimes) system de-asserts Send flag



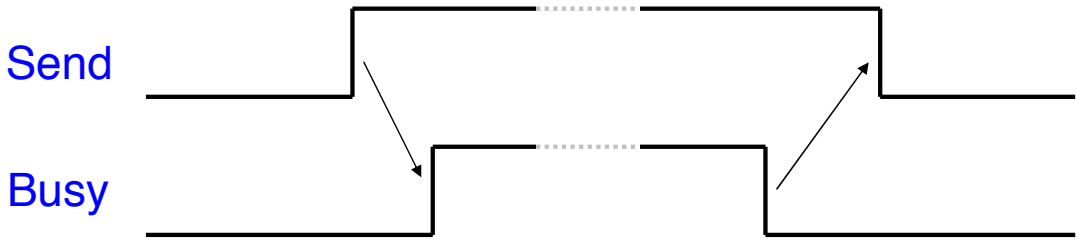
Slides from BYU CS 224

Let us design a UART transmitter



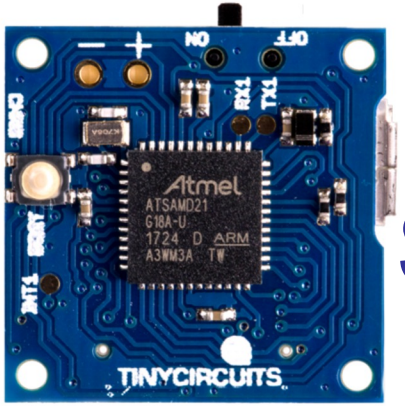
Transmitter/System Handshaking

- System asserts Send and holds it high when it wants to send a byte
- UART asserts Busy signal in response
- When UART has finished transfer, UART de-asserts Busy signal
- System de-asserts Send signal



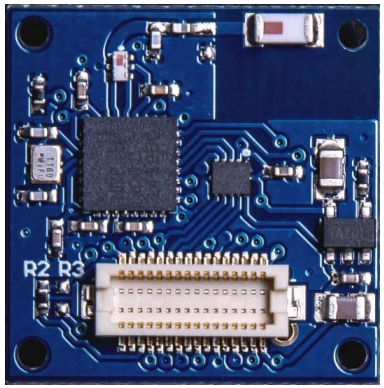
Slides from BYU CS 224

CSE190 Fall 2023



Lecture 11

Serial Busses (cont)



Wireless Embedded Systems

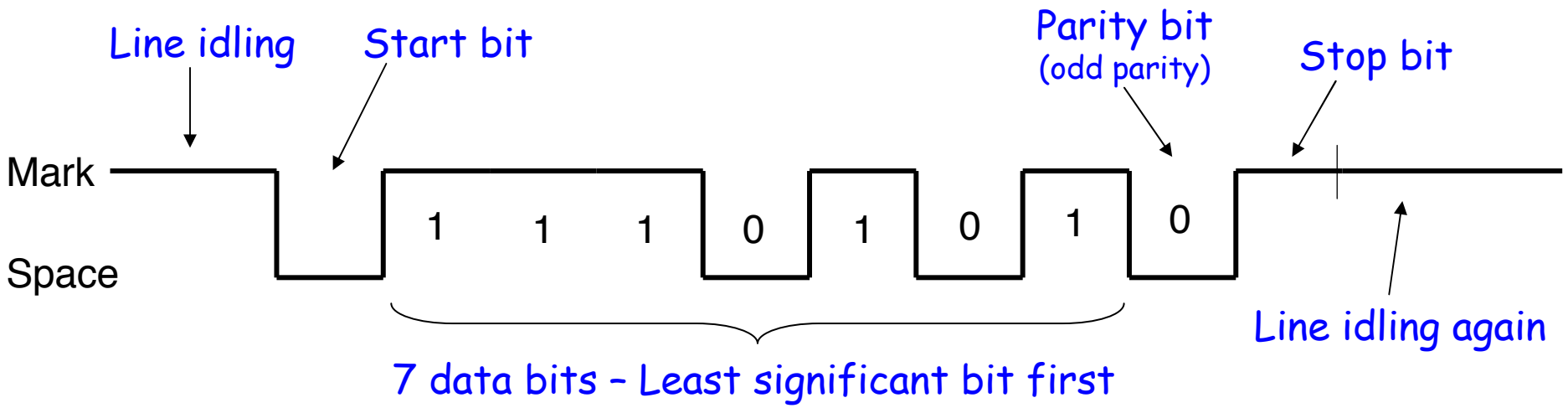
Aaron Schulman

Protocol

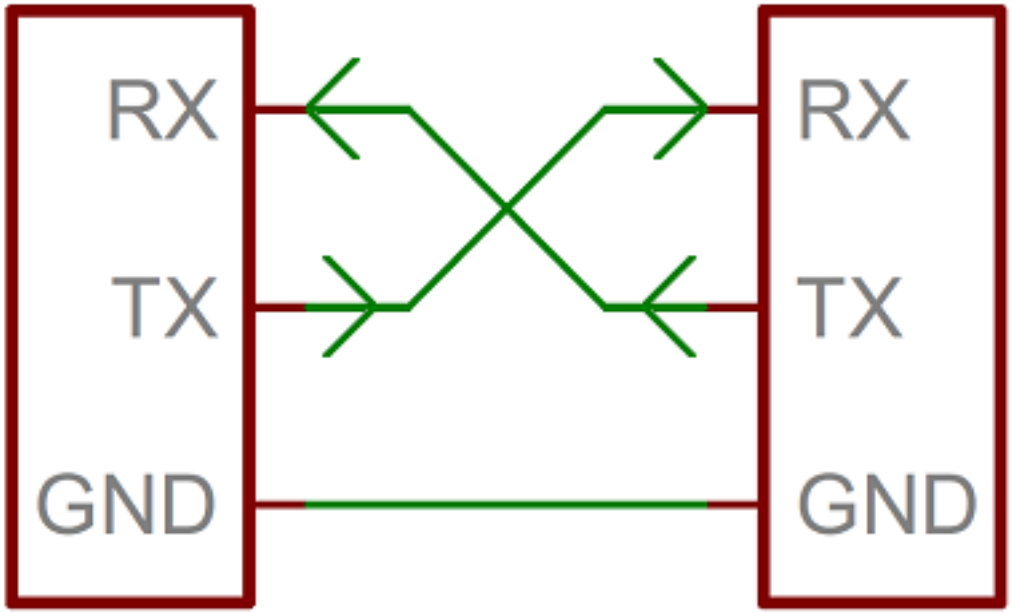
- Each character is sent as
  - a logic *low* **start** bit
  - a configurable number of data bits (usually 7 or 8, sometimes 5)
  - an optional parity bit
  - *one or more logic high* **stop** bits
  - with a particular bit timing (“baud”)

UART Example

- Send the ASCII letter ‘W’ (1010111)

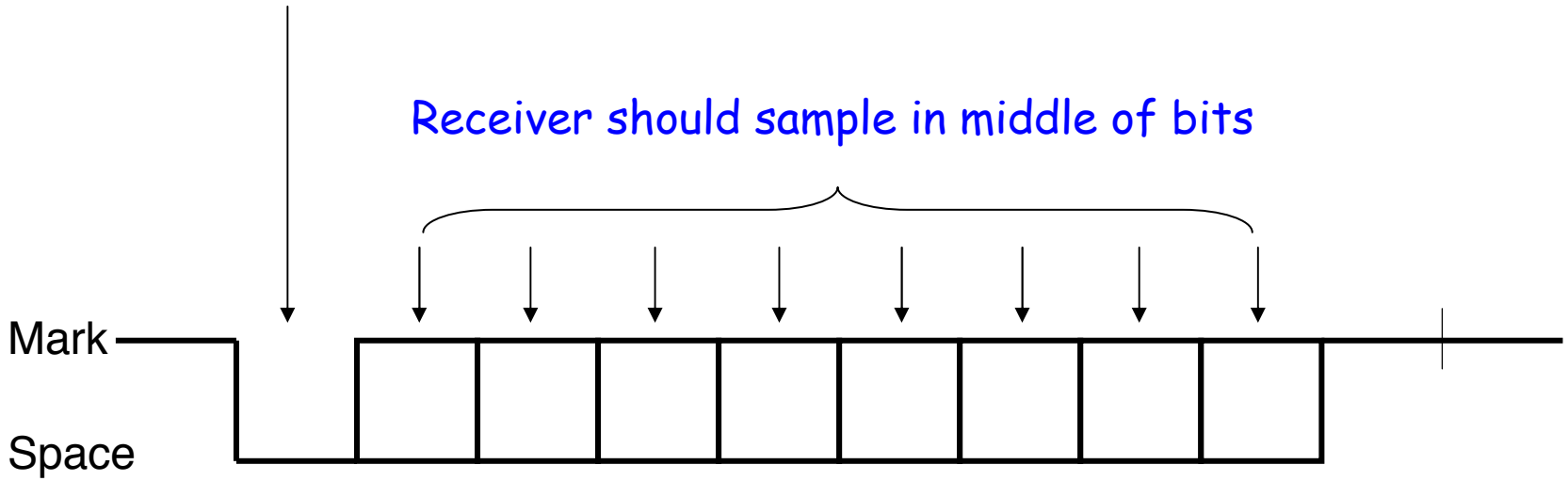


UART Hardware Connection



UART Character Reception

Start bit says a character is coming,
receiver resets its timers

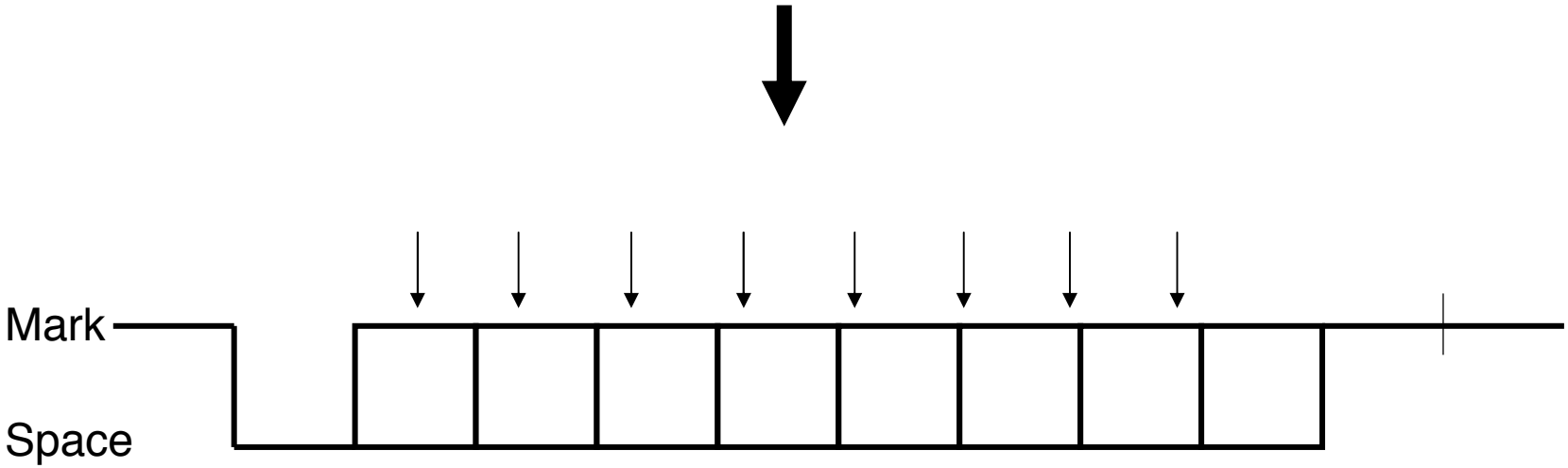


Receiver uses a timer (counter) to time when it samples.
Transmission rate (i.e., bit width) must be known!

Slides from BYU CS 224

UART Character Reception

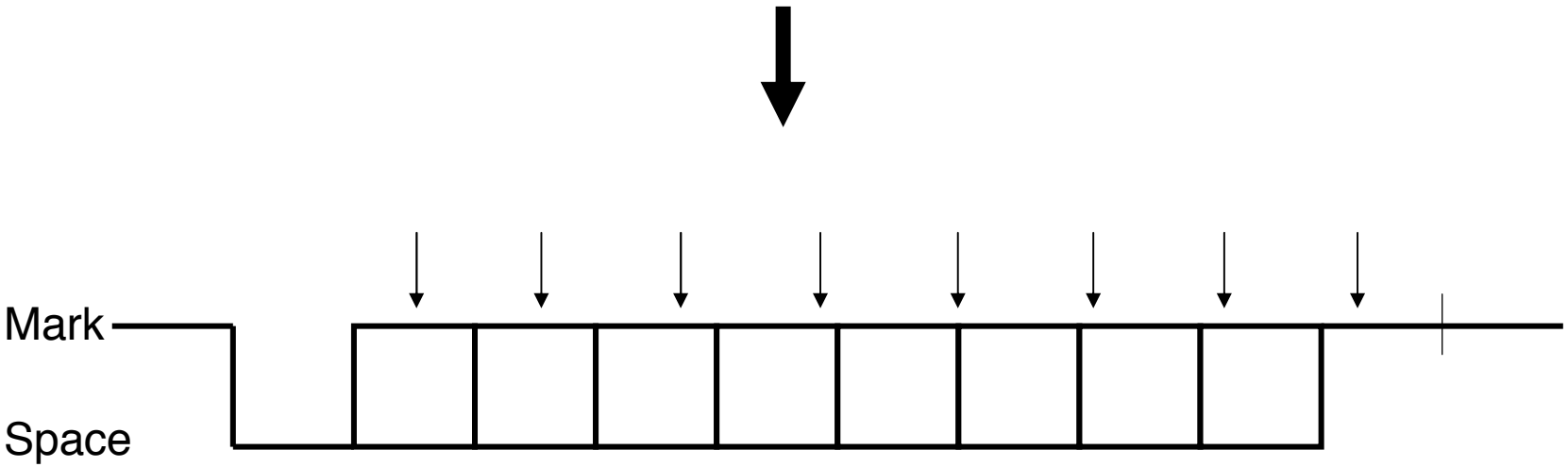
If receiver samples too quickly, see what happens…



Slides from BYU CS 224

UART Character Reception

If receiver samples too slowly, see what happens…



Receiver resynchronizes on every start bit.
Only has to be accurate enough to read 9 bits.

Slides from BYU CS 224

UART Character Reception

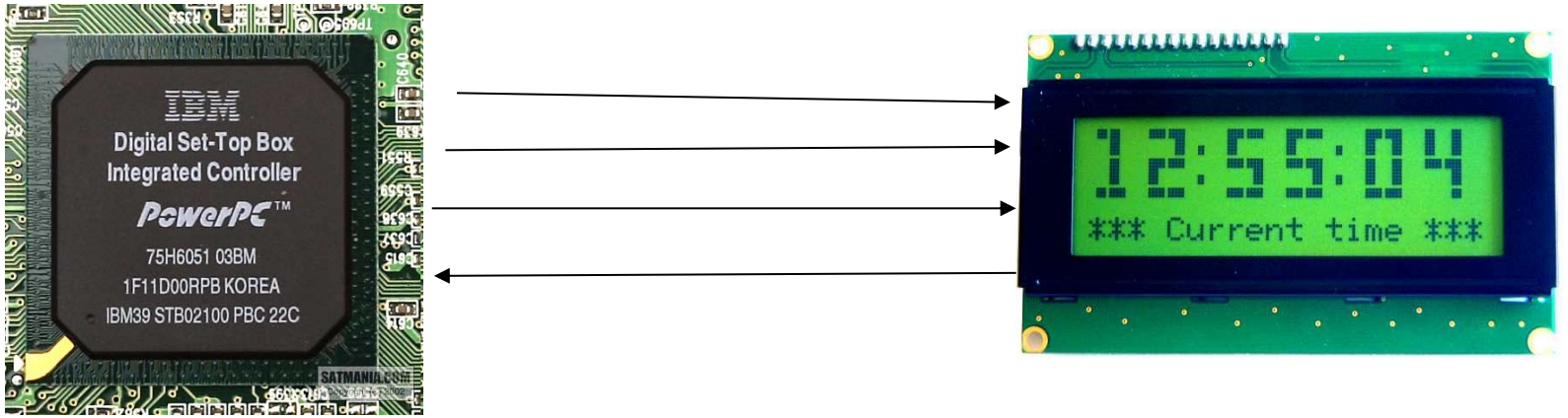
- Receiver also verifies that stop bit is ‘1’
  - If not, reports “framing error” to host system
- New start bit can appear immediately after stop bit
  - Receiver will resynchronize on each start bit

Serial Peripheral Interconnect (SPI)

- Another kind of serial protocol in embedded systems (proposed by Motorola)
- Four-wire protocol
  - SCLK — Serial Clock
  - MOSI/SIMO — Master Output, Slave Input
  - MISO/SOMI — Master Input, Slave Output
  - SS — Slave Select
- Single master device and with one or more slave devices
- Higher throughput than I2C and can do “stream transfers”
- No arbitration required
- But
  - Requires more pins
  - Has no hardware flow control
  - No slave acknowledgment (master could be talking to thin air and not even know it)

What is SPI?

- Serial Peripheral Interface (SPI) protocol [1979]
- Fast (Mbps), easy to use (few wires), simple
- Nearly all microcontrollers support it



SPI Basics

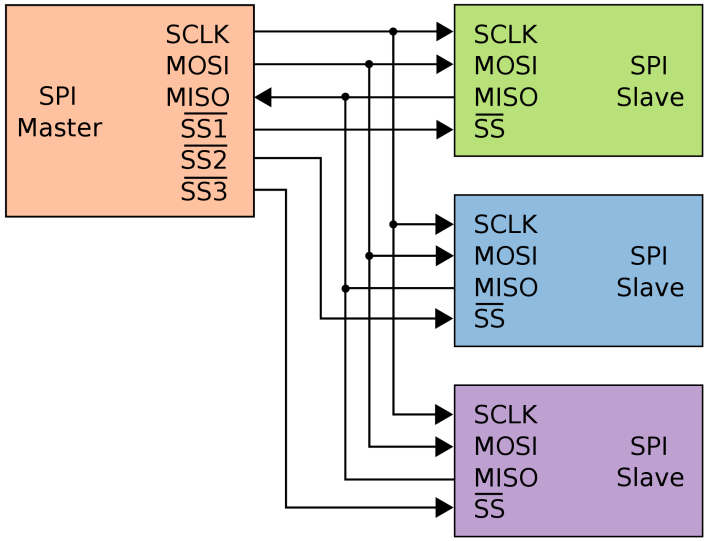
- Uses 4 wires (compared to UART's 2-wires)
  - Also known as a "4 wire" bus
- Used to communicate across short distances
- Multiple Secondaries, Single Primary
- Synchronized

SPI Capabilities

- Always Full Duplex
  - Communicating in two directions at the same time
  - Transmission need not be meaningful
- Multiple Mbps transmission speed
- Transfers data in 4 to 32 bit characters
- Multiple slaves
  - Daisy-chaining possible

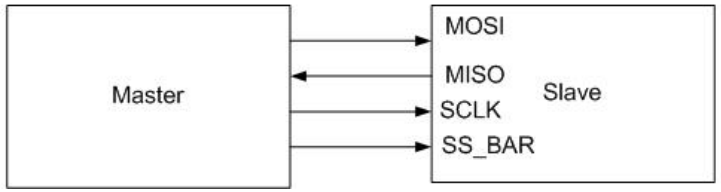
SPI Protocol

- Wires:
  - Master Out Slave In (MOSI)
  - Master In Slave Out (MISO)
  - System Clock (SCLK)
  - Slave Select 1…N
- Master Set Slave Select low
- Master Generates Clock
- Shift registers shift in and out data

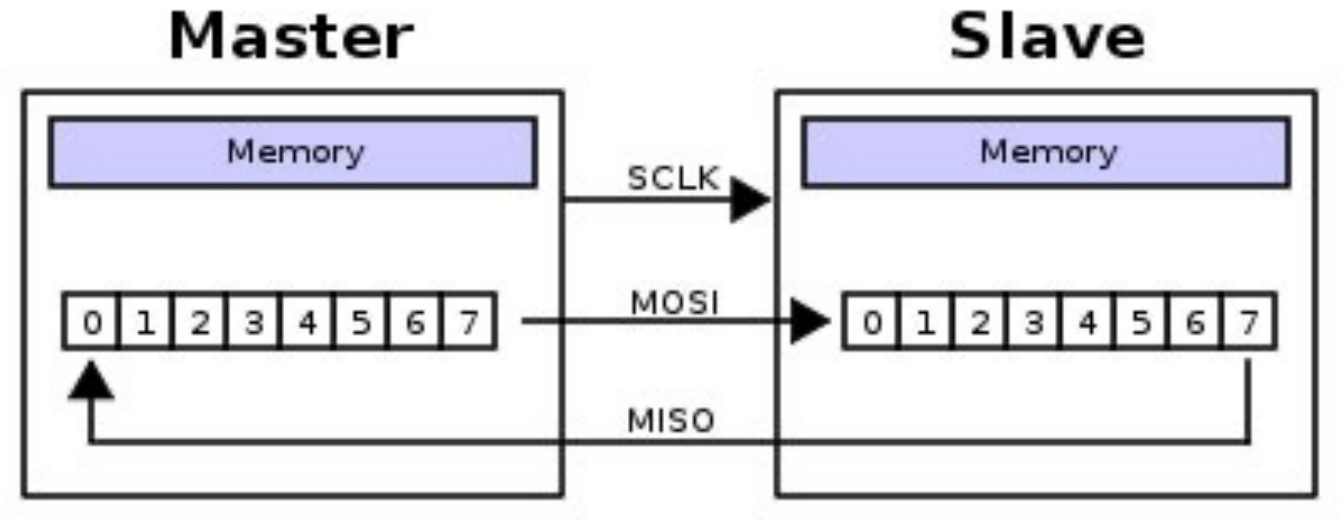


SPI Wires in Detail

- MOSI – Carries data out of Primary to Secondary
- MISO – Carries data from Secondary to Primary
  - Both signals happen for every transmission
- SS_BAR – Unique line to select a secondary
- SCLK – Primary-produced clock to synchronize data transfer



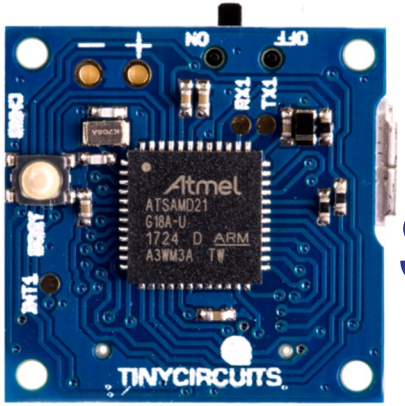
SPI is the quintessential “shift register” communication bus



Primary shifts out data to Secondary, and shifts in data from Secondary

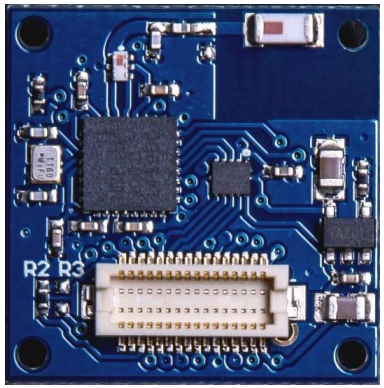
http://upload.wikimedia.org/wikipedia/commons/thumb/b/bb/SPI_8-bit_circular_transfer.svg/400px-SPI_8-bit_circular_transfer.svg.png

CSE190 Fall 2023



Lecture 12

Serial Busses (cont)

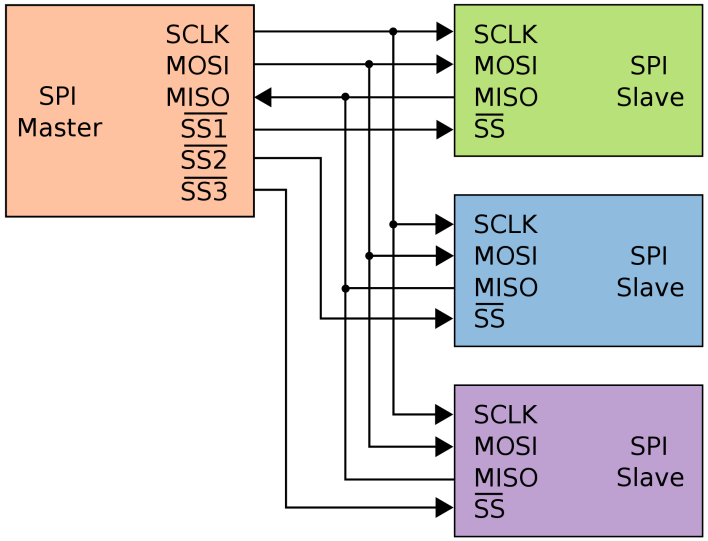


Wireless Embedded Systems

Aaron Schulman

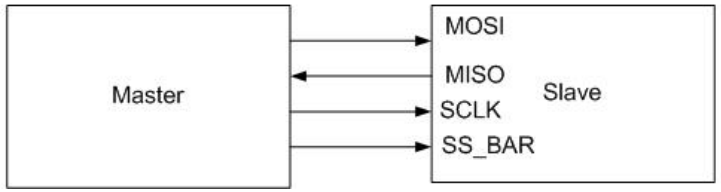
SPI Protocol

- Wires:
  - Master Out Slave In (MOSI)
  - Master In Slave Out (MISO)
  - System Clock (SCLK)
  - Slave Select 1…N
- Master Set Slave Select low
- Master Generates Clock
- Shift registers shift in and out data

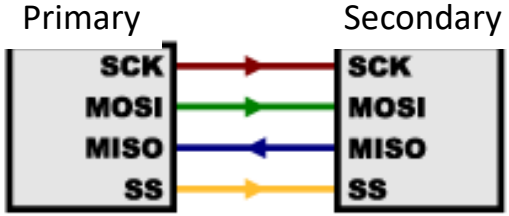


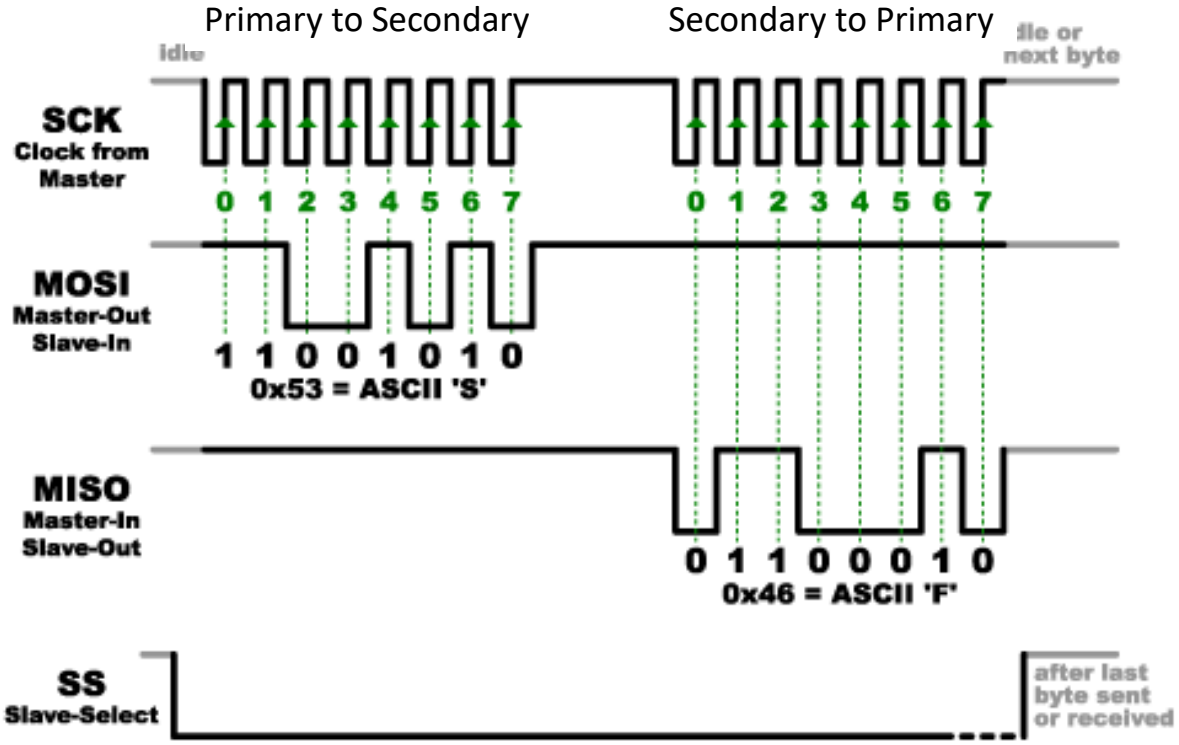
SPI Wires in Detail

- MOSI – Carries data out of Primary to Secondary
- MISO – Carries data from Secondary to Primary
  - Both signals happen for every transmission
- SS_BAR – Unique line to select a secondary
- SCLK – Primary-produced clock to synchronize data transfer



SPI Communication

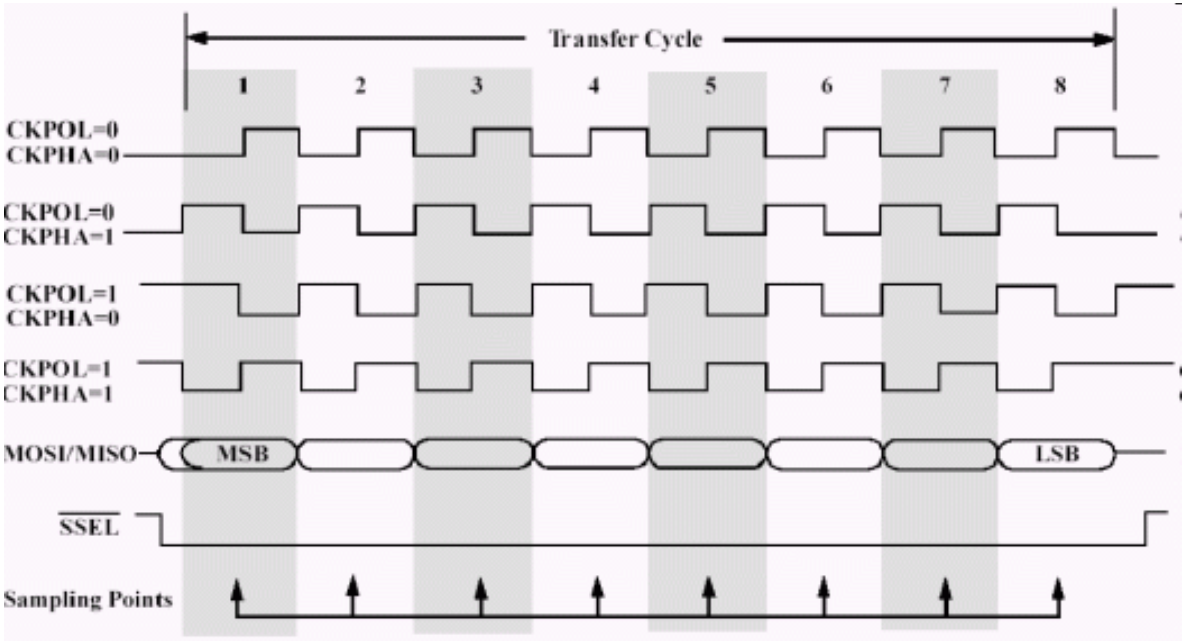




SPI clocking: **there is no "standard way"**

- Four clocking "modes"
  - Two phases
  - Two polarities
- Primary and *selected* secondary must be in the same mode
- During transfers with secondary's A and B, primary must
  - Configure clock to secondary A's clock mode
  - Select secondary A
  - Do transfer
  - Deselect secondary A
  - Configure clock to secondary B's clock mode
  - Select secondary B
  - Do transfer
  - Deselect secondary B
- primary reconfigures clock mode on-the-fly!

SPI timing diagram showing different clock modes



Timing Diagram – Showing Clock polarities and phases

http://www.maxim-ic.com.cn/images/appnotes/3078/3078Fig02.gif

SPI Pros and Cons

- Pros:
  - Fast and easy
    - Fast for point-to-point connections
    - Easily allows streaming/Constant data inflow
    - No addressing/Simple to implement
  - Everyone supports it

- Cons:
  - SS makes multiple secondaries very complicated
  - No acknowledgement ability
  - No inherent arbitration
  - No flow control

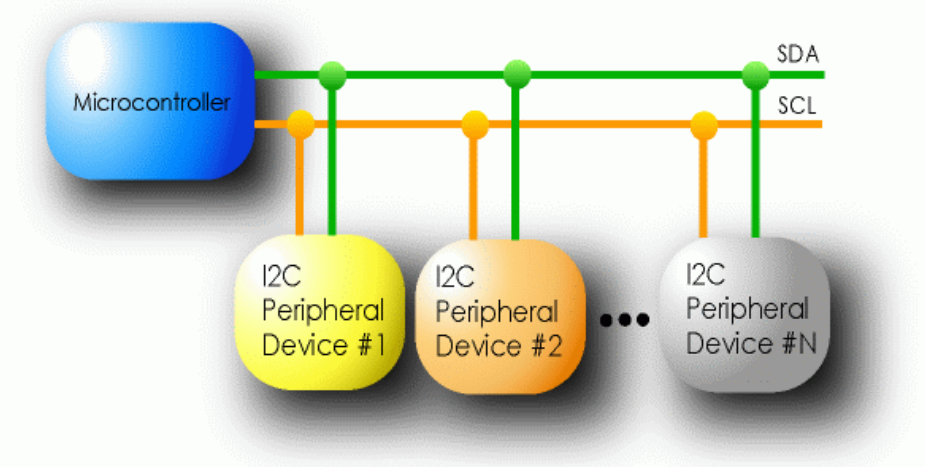
I2C bus (in our next project)

- Communication with the accelerometer
  - Read acceleration values and configure interrupts
- **Pros**
  - Two wires bus that can connect multiple peripherals with the MCU
- **Cons**
  - Overhead is significantly higher, and bus is slower

I2C Details

- Two lines
  - Serial data line (SDA)
  - Serial clock line (SCL)

- Only two wires for connecting multiple devices

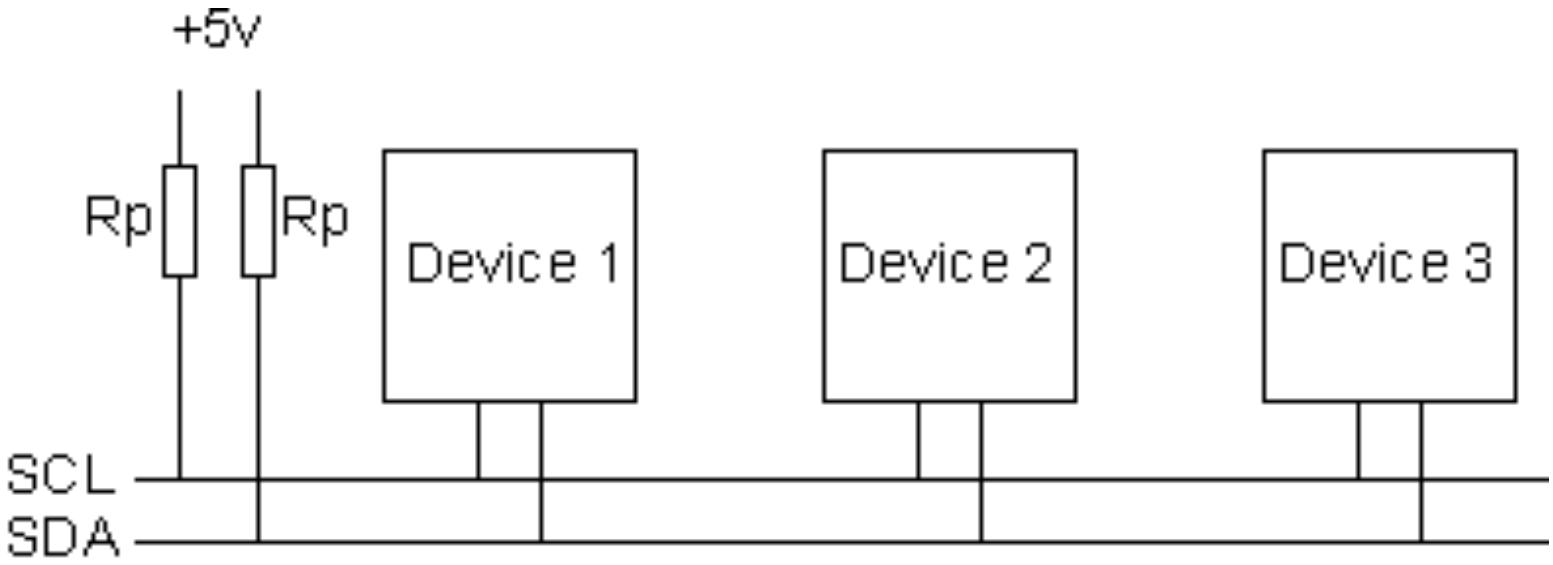


I2C Details

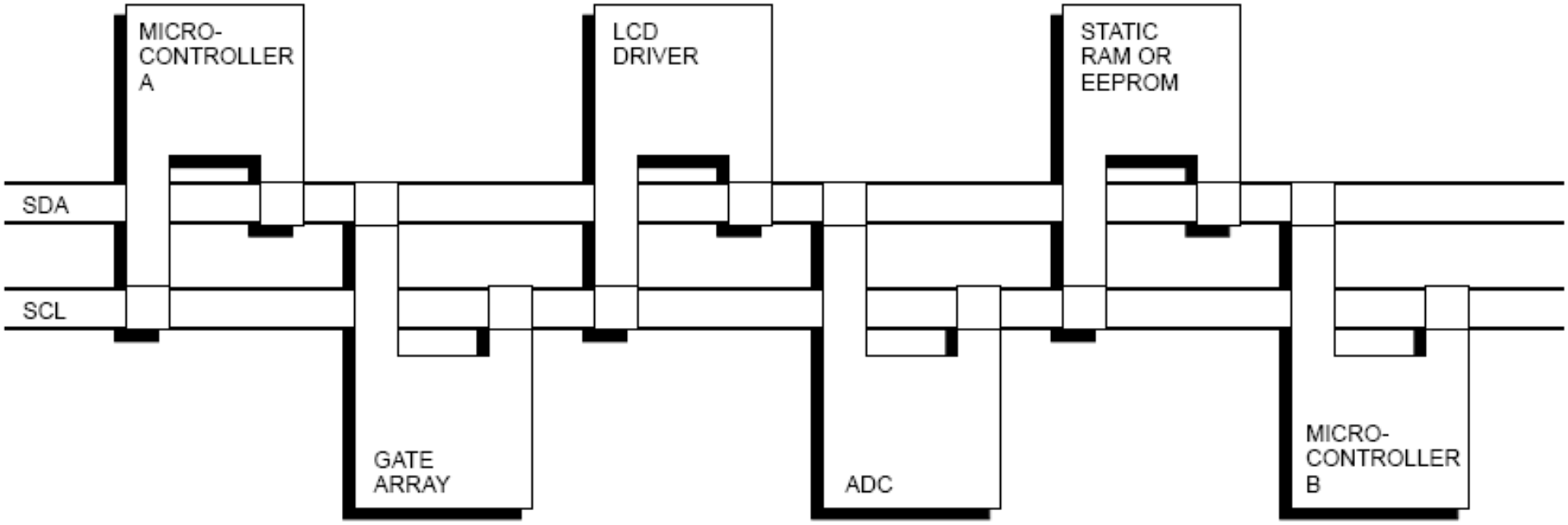
- Each I2C device recognized by a unique address
- Each I2C device can be either a transmitter or receiver
- I2C devices can be primarys or secondarys for a data transfer
  - primary (usually a microcontroller): Initiates a data transfer on the bus, generates the clock signals to permit that transfer, and terminates the transfer
  - secondary: Any device addressed by the primary at that time

How can any device transfer or receive on the same two wires?

- Pull ups and high-impedance mode pins
  - Wires default to being “high”, any device can make a wire go “low”.
  - This is super clever. SPI and UART can’t do this, why?



I2C-Connected System



Example I2C-connected system with two microcontrollers

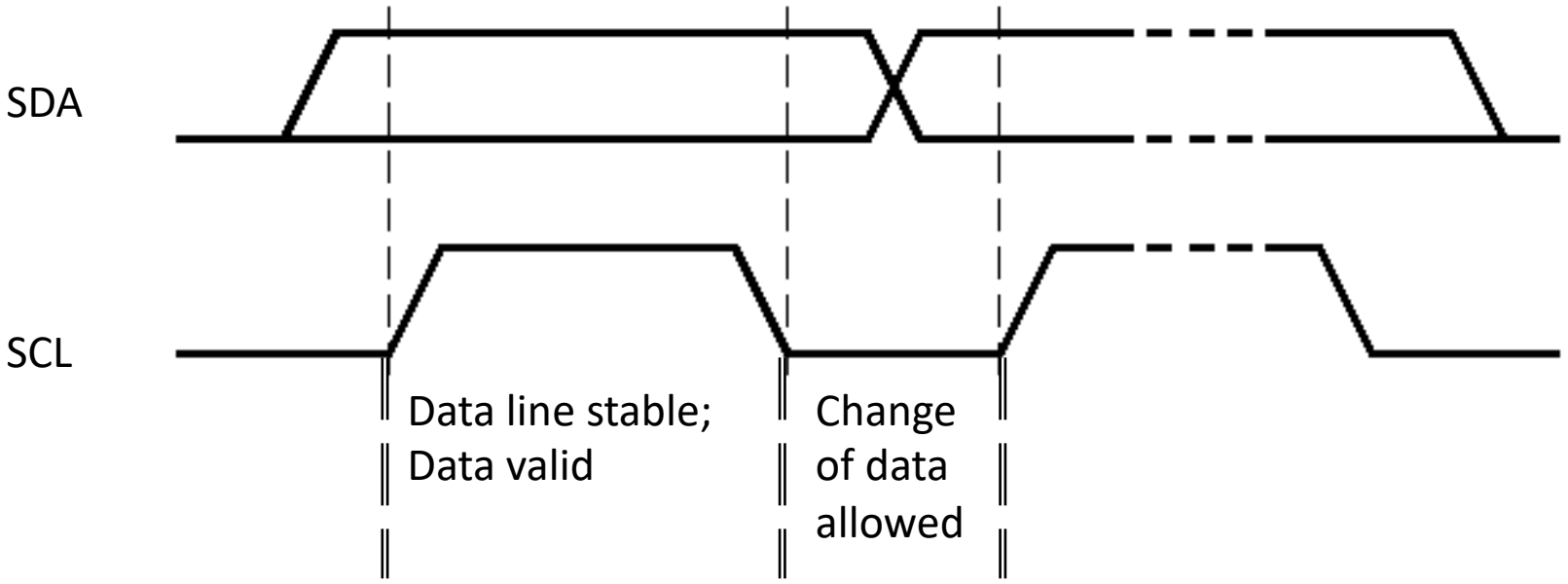
(Source: I2C Specification, Philips)

Primary/Secondary Relationships

- Who is the primary?
  - primary-transmitters
  - primary-receivers
- Suppose microcontroller A wants to send information to microcontroller B
  - A (primary) addresses B (secondary)
  - A (primary-transmitter), sends data to B (secondary-receiver)
  - A terminates the transfer.
- If microcontroller A wants to receive information from microcontroller B
  - A (primary) addresses microcontroller B (secondary)
  - A (primary-receiver) receives data from B (secondary-transmitter)
  - A terminates the transfer
- In both cases, the primary (microcontroller A) generates the timing and terminates the transfer

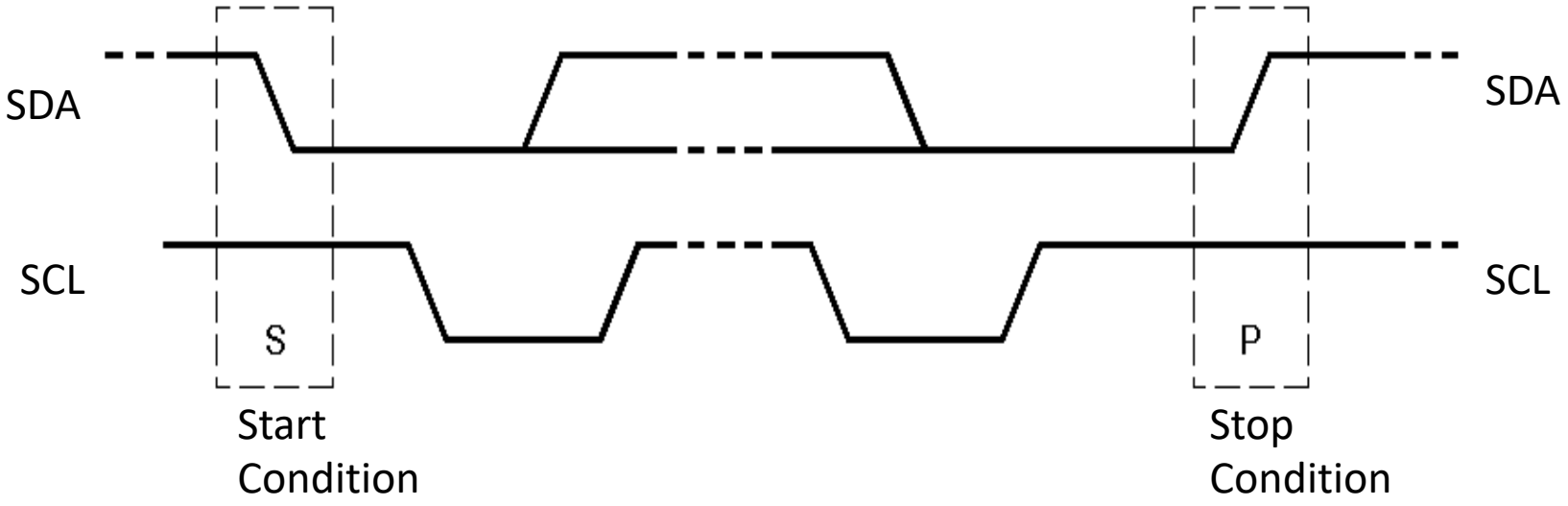
Bit Transfer on the I^2C Bus

- In normal data transfer, the data line only changes state when the clock is low



Start and Stop Conditions

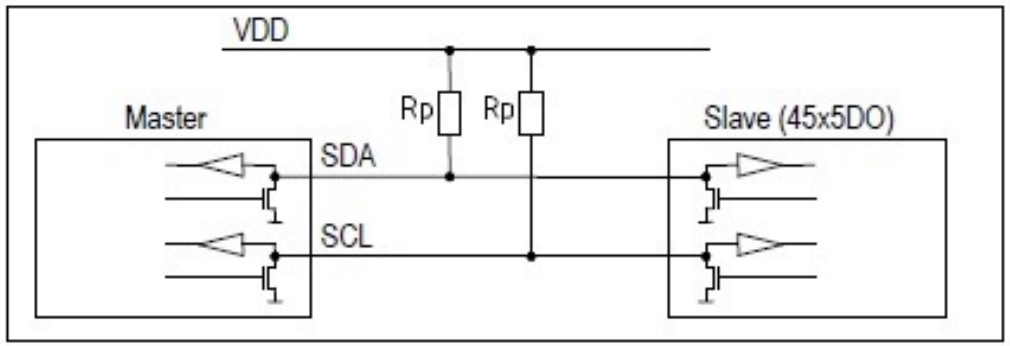
- A transition of the data line while the clock line is high is defined as either a start or a stop condition.
- Both start and stop conditions are generated by the bus primary
- The bus is considered busy after a start condition, until a stop condition occurs

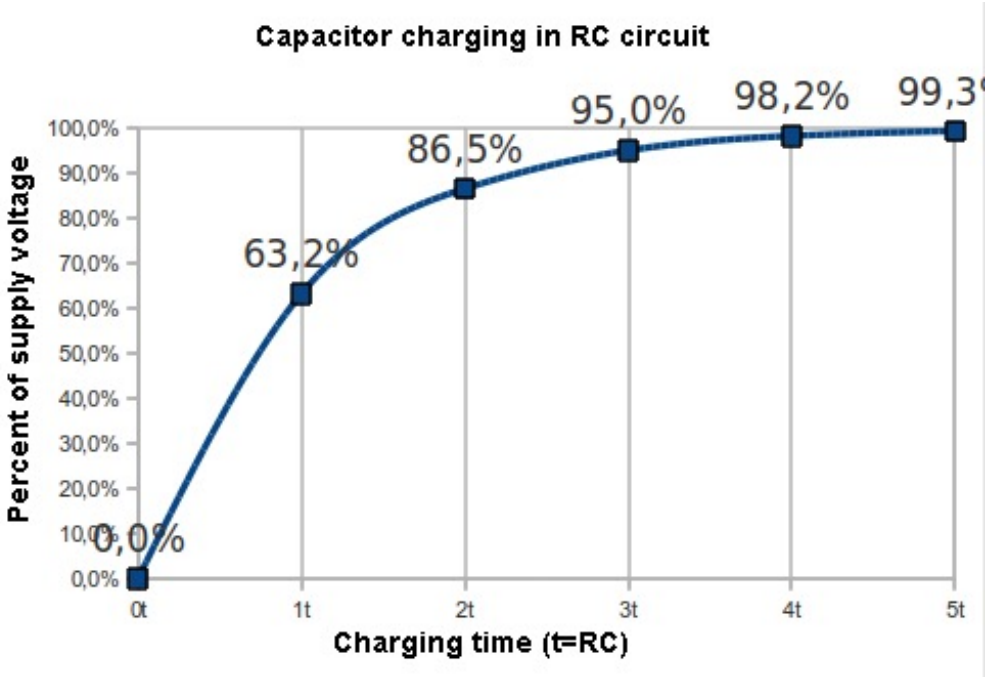


I^2C Addressing

- Each node has a unique 7 (or 10) bit address
- Peripherals often have fixed and programmable address portions
- Addresses starting with 0000 or 1111 have special functions:-
  - 0000000 Is a General Call Address
  - 0000001 Is a Null (CBUS) Address
  - 1111XXX Address Extension
  - 1111111 Address Extension – Next Bytes are the Actual Address

How fast can I2C run?





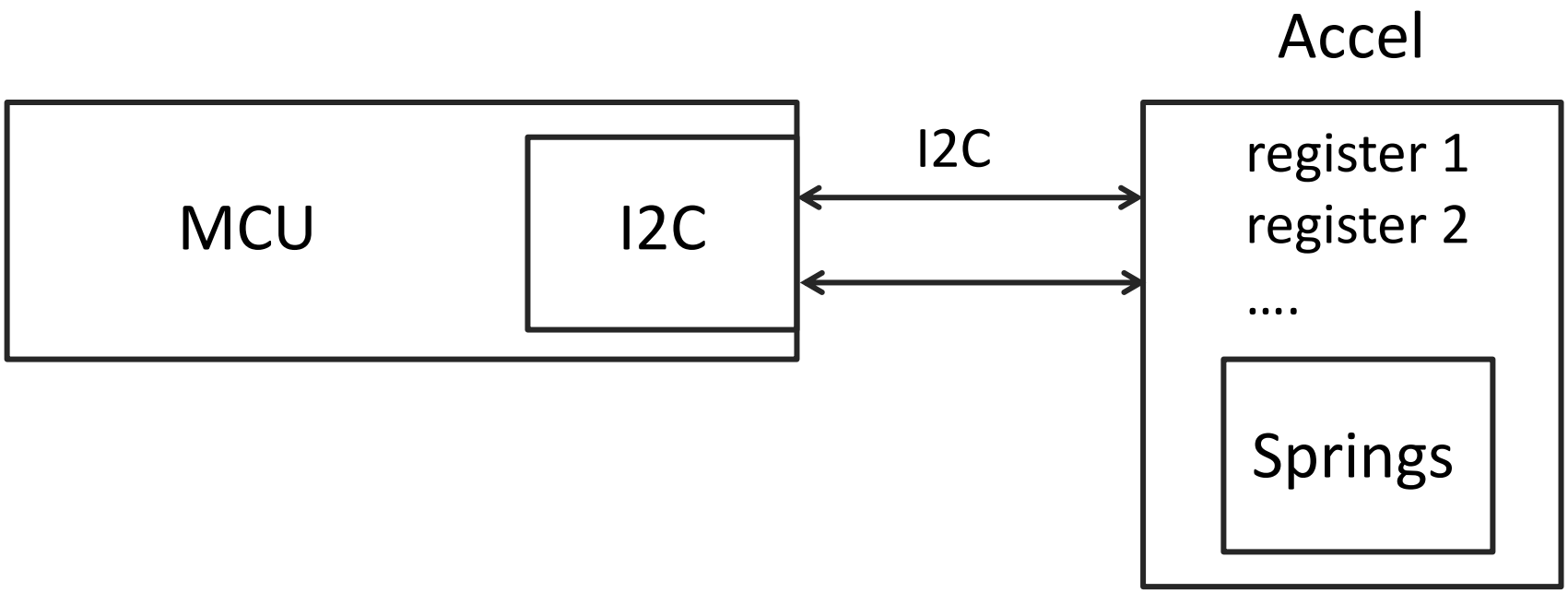
- How fast can you run it?
- Assumptions
  - 0's are driven
  - 1's are "pulled up"
- Some working figures
  - R_p = 10 kΩ
  - C_{cap} = 100 pF
  - V_{DD} = 5 V
  - V_{in_high} = 3.5 V
- Recall for RC circuit
  - $V_{cap}(t) = V_{DD}(1-e^{-t/\tau})$
  - Where $\tau = RC$

Practically I2C can do at most 400kbps

Exercise: Bus bit rate vs Useful data rate

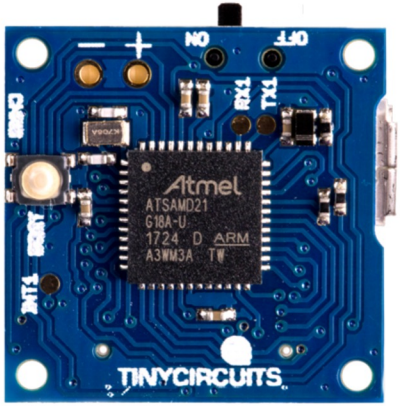
- An I2C “transactions” involves the following bits
  - <S><A6:A0><R/W><A><D7:D0><A><F>
- Which of these actually carries useful data?
  - <S><A6:A0><R/W><A><D7:D0><A><F>
- So, if a bus runs at 400 kHz
  - What is the clock period?
  - What is the data throughput (i.e. data-bits/second)?
  - What is the bus “efficiency”?

How to operate the accelerometer?



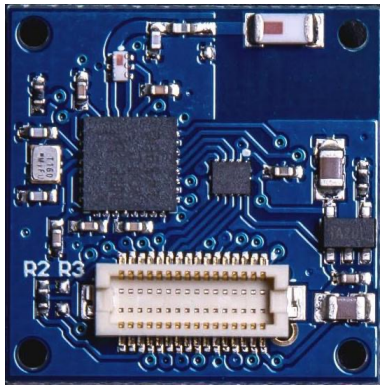
https://www.youtube.com/watch?v=eqZgxR6eRjo

CSE190 Fall 2023



Lecture 13

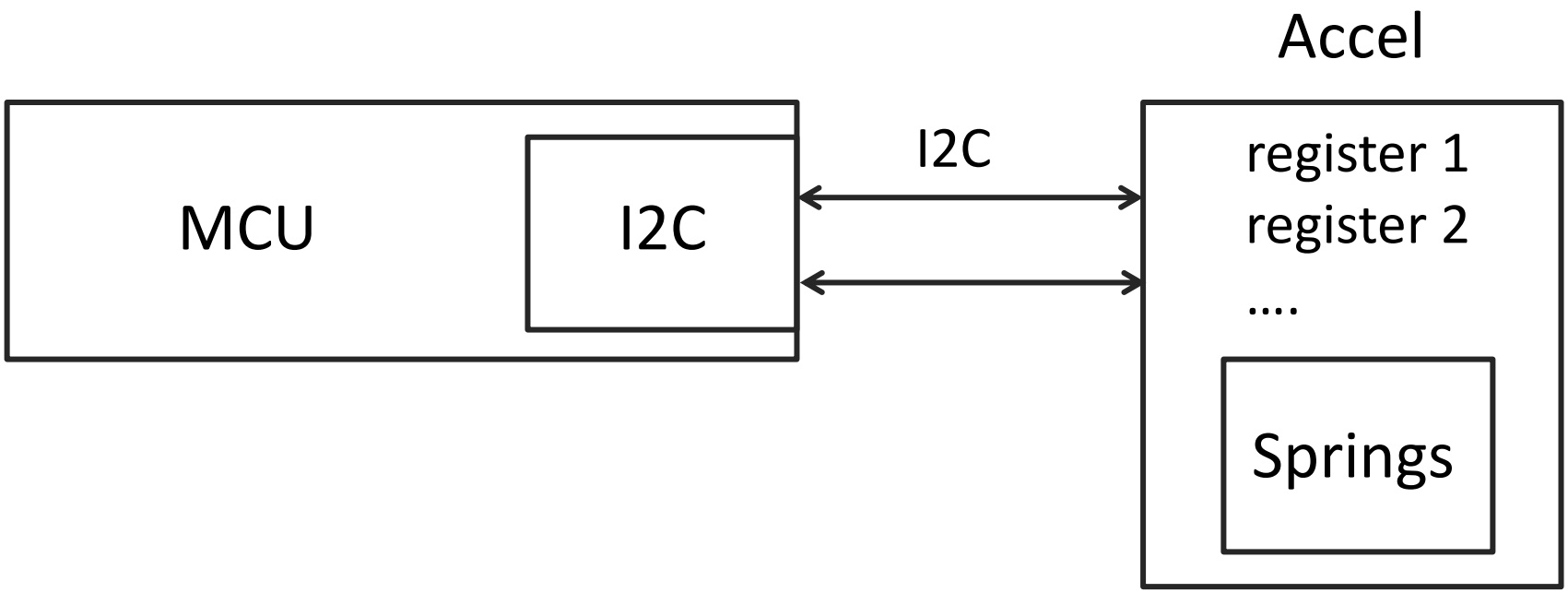
Peripheral example (Accel)



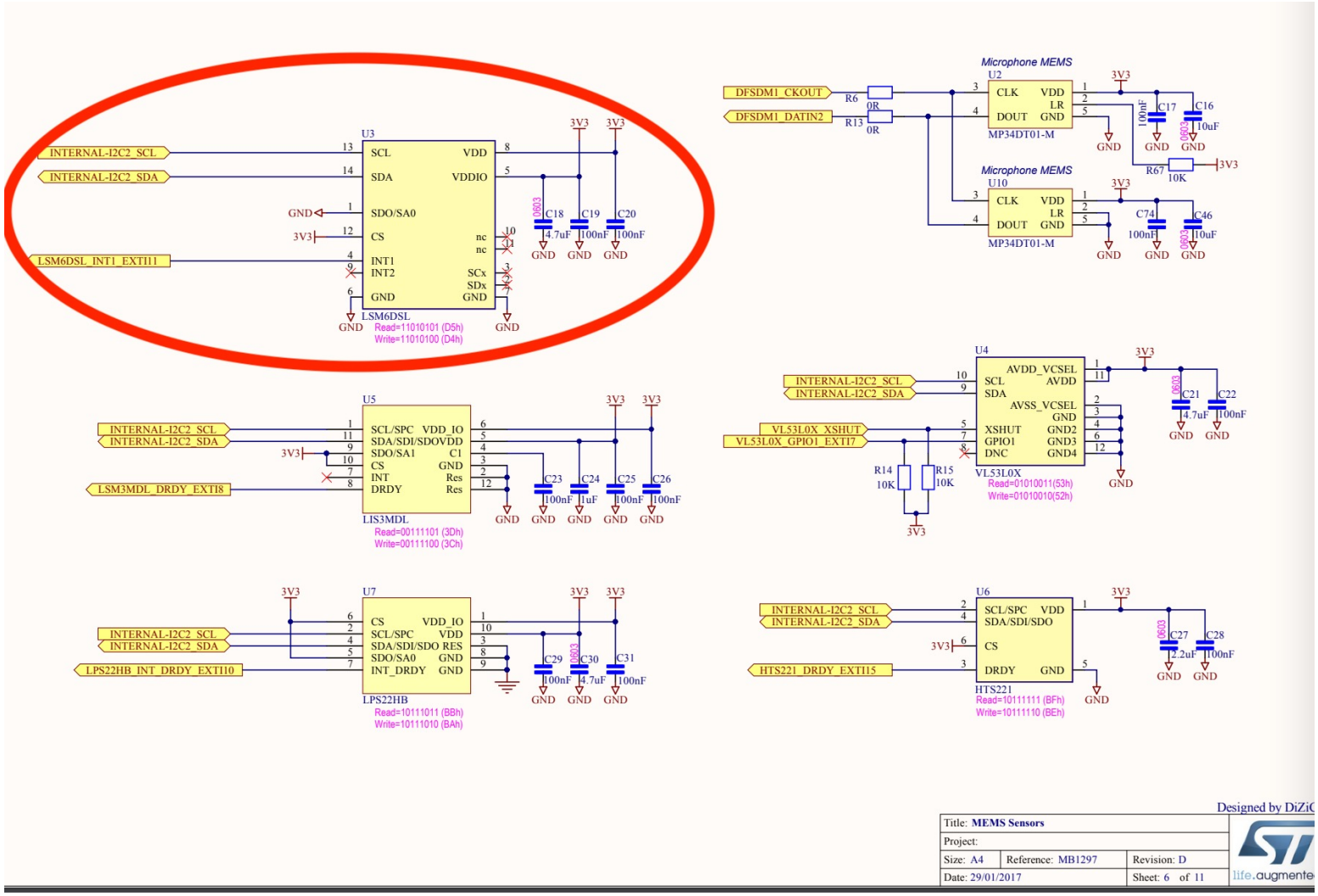
Wireless Embedded Systems

Aaron Schulman

How to operate the accelerometer?



I2C bus connected to accel



Bus-connected peripherals have registers

8 Register mapping

The table given below provides a list of the 8/16-bit registers embedded in the device and the corresponding addresses.

Table 19. Registers address map

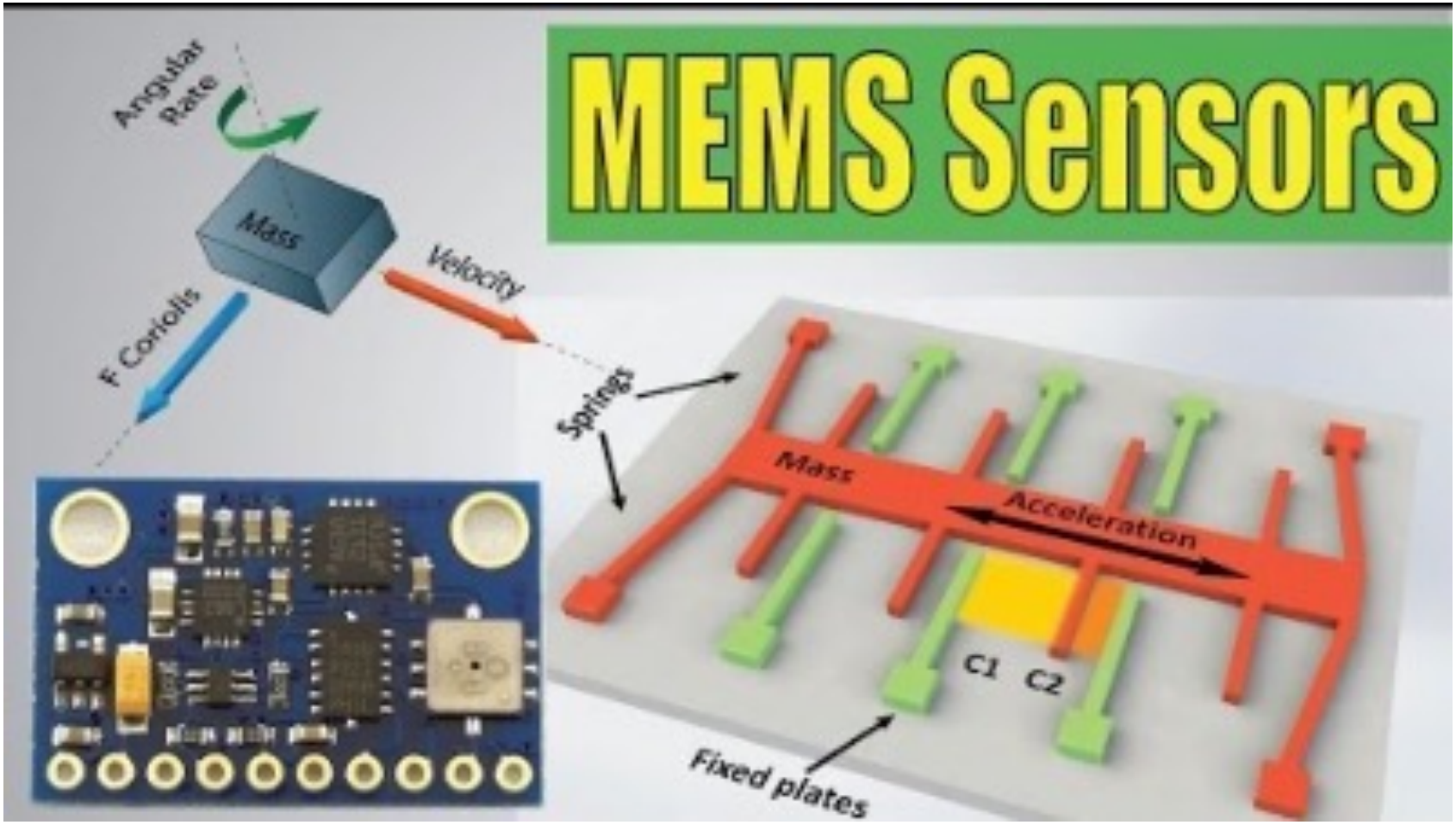
Name	Type	Register address		Default	Comment
		Hex	Binary		
RESERVED	-	00	00000000	-	Reserved
FUNC_CFG_ACCESS	r/w	01	00000001	00000000	Embedded functions configuration register
RESERVED	-	02	00000010	-	Reserved
RESERVED	-	03	00000011	-	Reserved
SENSOR_SYNC_TIME_FRAME	r/w	04	00000100	00000000	Sensor sync configuration register
SENSOR_SYNC_RES_RATIO	r/w	05	00000101	00000000	
FIFO_CTRL1	r/w	06	00000110	00000000	FIFO configuration registers
FIFO_CTRL2	r/w	07	00000111	00000000	
FIFO_CTRL3	r/w	08	00001000	00000000	
FIFO_CTRL4	r/w	09	00001001	00000000	
FIFO_CTRL5	r/w	0A	00001010	00000000	
DRDY_PULSE_CFG_G	r/w	0B	00001011	00000000	
RESERVED	-	0C	00001100	-	Reserved
INT1_CTRL	r/w	0D	00001101	00000000	INT1 pin control
INT2_CTRL	r/w	0E	00001110	00000000	INT2 pin control
WHO_AM_I	r	0F	00001111	01101010	Who I am ID
CTRL1_XL	r/w	10	00010000	00000000	Accelerometer and gyroscope control registers
CTRL2_G	r/w	11	00010001	00000000	
CTRL3_C	r/w	12	00010010	00000100	
CTRL4_C	r/w	13	00010011	00000000	
CTRL5_C	r/w	14	00010100	00000000	
CTRL6_C	r/w	15	00010101	00000000	
CTRL7_G	r/w	16	00010110	00000000	
CTRL8_XL	r/w	17	0001 0111	00000000	
CTRL9_XL	r/w	18	00011000	00000000	
CTRL10_C	r/w	19	00011001	00000000	

Table 19. Registers address map (continued)

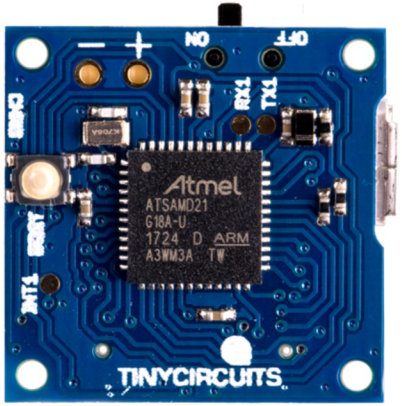
Name	Type	Register address		Default	Comment
		Hex	Binary		
MASTER_CONFIG	r/w	1A	00011010	00000000	I^2C master configuration register
WAKE_UP_SRC	r	1B	00011011	output	Interrupt registers
TAP_SRC	r	1C	00011100	output	
D6D_SRC	r	1D	00011101	output	
STATUS_REG	r	1E	00011110	output	Status data register for user interface
RESERVED	-	1F	00011111	-	
OUT_TEMP_L	r	20	00100000	output	Temperature output data registers
OUT_TEMP_H	r	21	00100001	output	
OUTX_L_G	r	22	00100010	output	Gyroscope output registers for user interface
OUTX_H_G	r	23	00100011	output	
OUTY_L_G	r	24	00100100	output	
OUTY_H_G	r	25	00100101	output	
OUTZ_L_G	r	26	00100110	output	
OUTZ_H_G	r	27	00100111	output	
OUTX_L_XL	r	28	00101000	output	Accelerometer output registers
OUTX_H_XL	r	29	00101001	output	
OUTY_L_XL	r	2A	00101010	output	
OUTY_H_XL	r	2B	00101011	output	
OUTZ_L_XL	r	2C	00101100	output	
OUTZ_H_XL	r	2D	00101101	output	
SENSORHUB1_REG	r	2E	00101110	output	Sensor hub output registers
SENSORHUB2_REG	r	2F	00101111	output	
SENSORHUB3_REG	r	30	00110000	output	
SENSORHUB4_REG	r	31	00110001	output	
SENSORHUB5_REG	r	32	00110010	output	
SENSORHUB6_REG	r	33	00110011	output	
SENSORHUB7_REG	r	34	00110100	output	
SENSORHUB8_REG	r	35	00110101	output	
SENSORHUB9_REG	r	36	00110110	output	
SENSORHUB10_REG	r	37	00110111	output	
SENSORHUB11_REG	r	38	00111000	output	
SENSORHUB12_REG	r	39	00111001	output	

Table 19. Registers address map (continued)

Name	Type	Register address		Default	Comment
		Hex	Binary		
FIFO_STATUS1	r	3A	00111010	output	FIFO status registers
FIFO_STATUS2	r	3B	00111011	output	
FIFO_STATUS3	r	3C	00111100	output	
FIFO_STATUS4	r	3D	00111101	output	
FIFO_DATA_OUT_L	r	3E	00111110	output	FIFO data output registers
FIFO_DATA_OUT_H	r	3F	00111111	output	
TIMESTAMP0_REG	r	40	01000000	output	Timestamp output registers
TIMESTAMP1_REG	r	41	01000001	output	
TIMESTAMP2_REG	r/w	42	01000010	output	
RESERVED	-	43-48		-	Reserved
STEP_TIMESTAMP_L	r	49	0100 1001	output	Step counter timestamp registers
STEP_TIMESTAMP_H	r	4A	0100 1010	output	
STEP_COUNTER_L	r	4B	01001011	output	Step counter output registers
STEP_COUNTER_H	r	4C	01001100	output	
SENSORHUB13_REG	r	4D	01001101	output	Sensor hub output registers
SENSORHUB14_REG	r	4E	01001110	output	
SENSORHUB15_REG	r	4F	01001111	output	
SENSORHUB16_REG	r	50	01010000	output	
SENSORHUB17_REG	r	51	01010001	output	
SENSORHUB18_REG	r	52	01010010	output	
FUNC_SRC1	r	53	01010011	output	Interrupt registers
FUNC_SRC2	r	54	01010100	output	
WRIST_TILT_IA	r	55	01010101	output	Interrupt register
RESERVED	-	56-57		-	Reserved
TAP_CFG	r/w	58	01011000	00000000	Interrupt registers
TAP_THS_6D	r/w	59	01011001	00000000	
INT_DUR2	r/w	5A	01011010	00000000	
WAKE_UP_THS	r/w	5B	01011011	00000000	
WAKE_UP_DUR	r/w	5C	01011100	00000000	
FREE_FALL	r/w	5D	01011101	00000000	
MD1_CFG	r/w	5E	01011110	00000000	
MD2_CFG	r/w	5F	01011111	00000000	
MASTER_CMD_CODE	r/w	60	01100000	00000000	

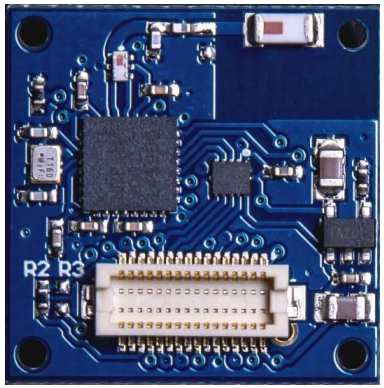

MEMS Sensors
Angular Rate
Mass
Velocity
F Coriolis
Springs
Mass
Acceleration
C1
C2
Fixed plates

CSE190 Fall 2023



Lecture 14

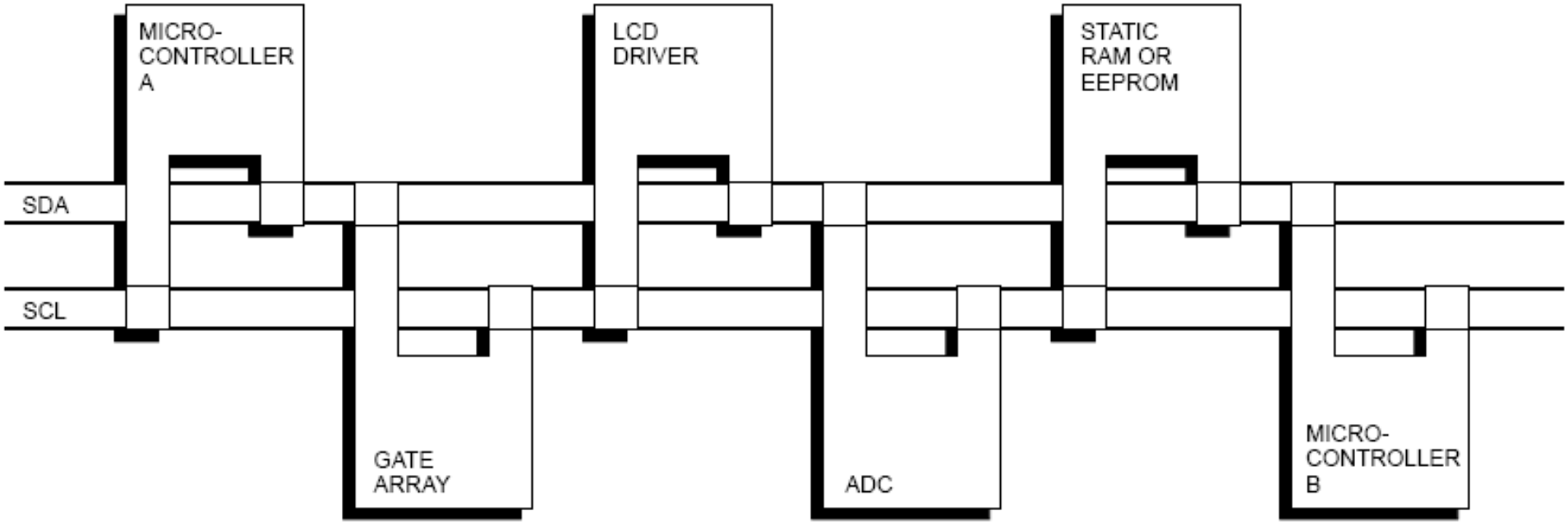
How to use I2C



Wireless Embedded Systems

Aaron Schulman

I2C-Connected System



Example I2C-connected system with two microcontrollers

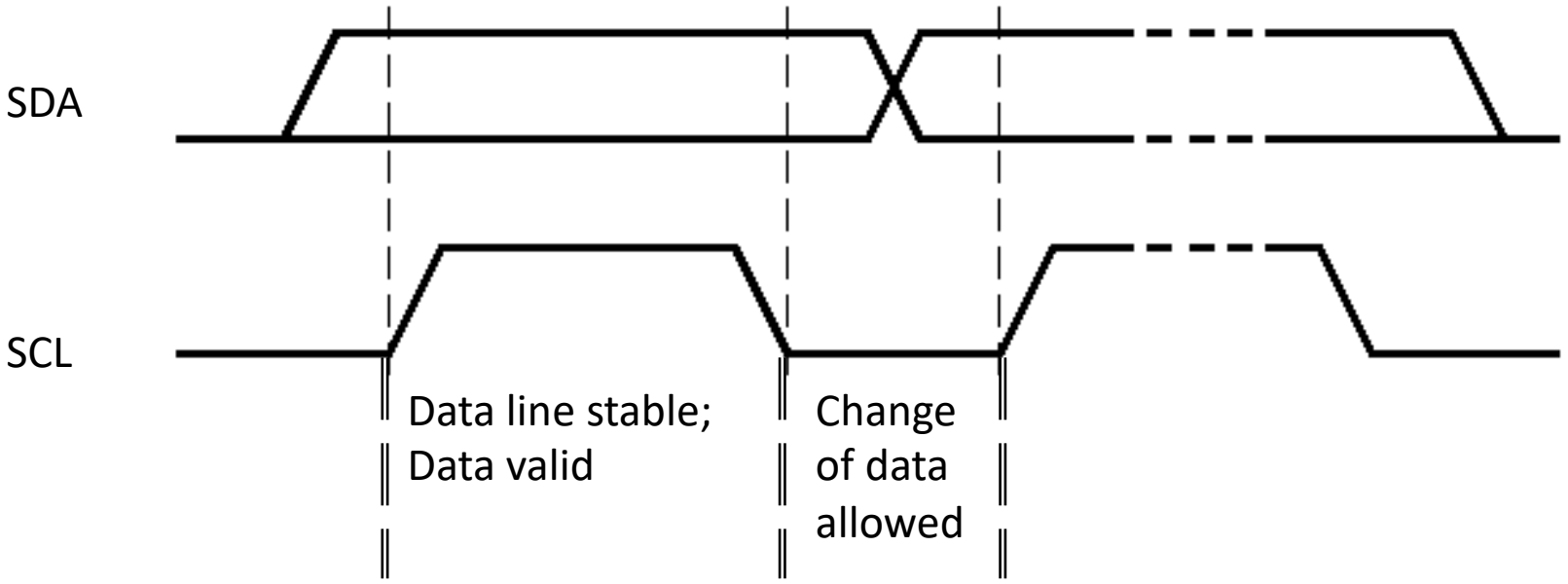
(Source: I2C Specification, Philips)

Primary/Secondary Relationships

- Who is the primary?
  - primary-transmitters
  - primary-receivers
- Suppose microcontroller A wants to send information to microcontroller B
  - A (primary) addresses B (secondary)
  - A (primary-transmitter), sends data to B (secondary-receiver)
  - A terminates the transfer.
- If microcontroller A wants to receive information from microcontroller B
  - A (primary) addresses microcontroller B (secondary)
  - A (primary-receiver) receives data from B (secondary-transmitter)
  - A terminates the transfer
- In both cases, the primary (microcontroller A) generates the timing and terminates the transfer

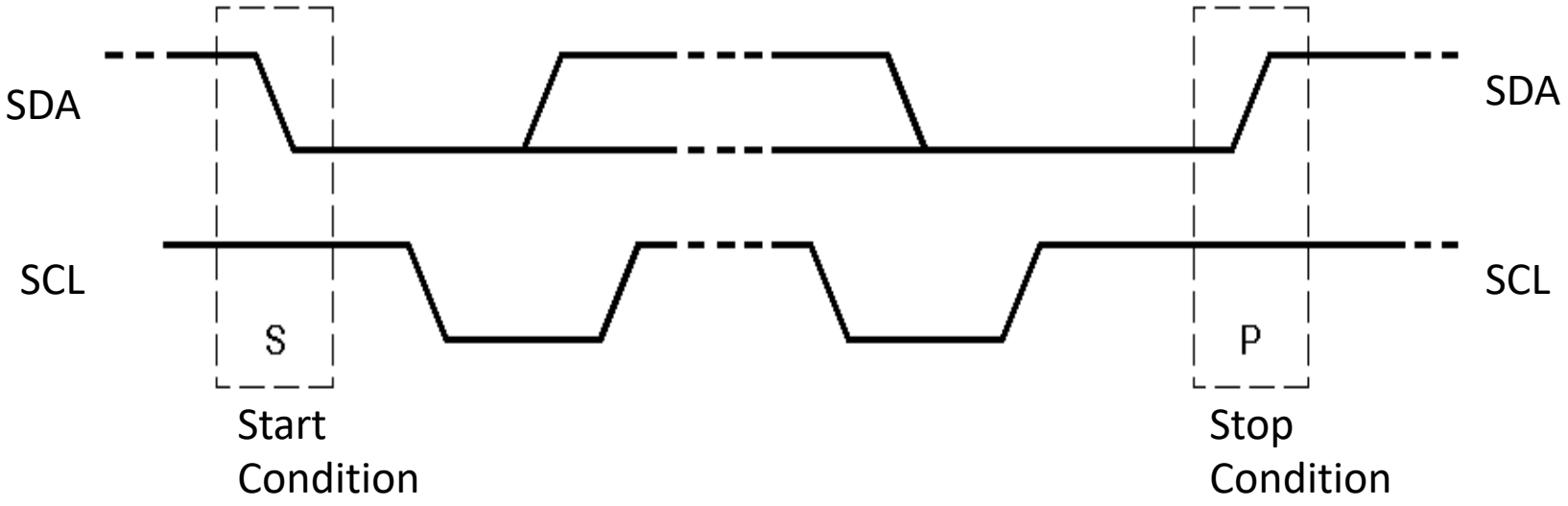
Bit Transfer on the I^2C Bus

- In normal data transfer, the data line only changes state when the clock is low



Start and Stop Conditions

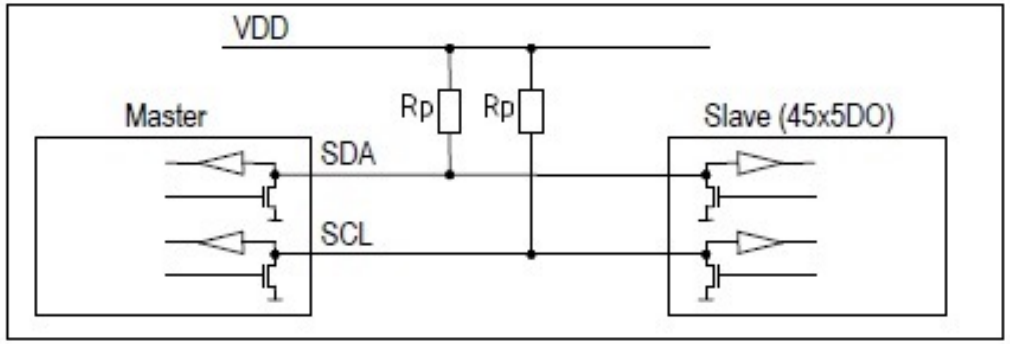
- A transition of the data line while the clock line is high is defined as either a start or a stop condition.
- Both start and stop conditions are generated by the bus primary
- The bus is considered busy after a start condition, until a stop condition occurs



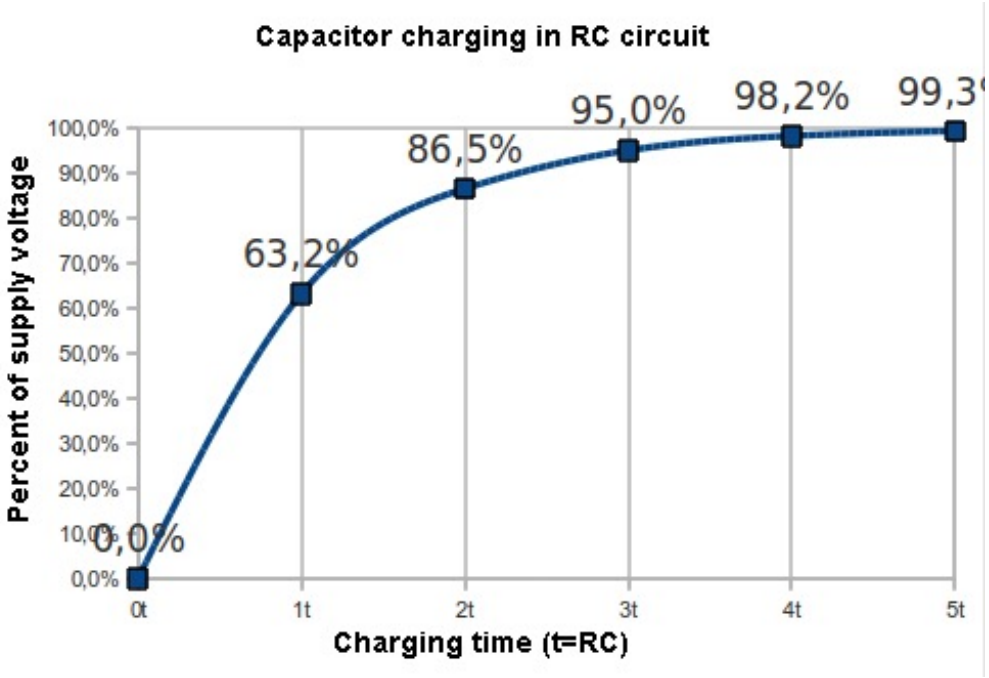
I^2C Addressing

- Each node has a unique 7 (or 10) bit address
- Peripherals often have fixed and programmable address portions
- Addresses starting with 0000 or 1111 have special functions:-
  - 0000000 Is a General Call Address – E.g. system reset
  - 1111XXX Address Extension
  - 1111111 Address Extension – Next Bytes are the Actual Address

How fast can I2C run?



- How fast can you run it?
- Assumptions
  - 0's are driven
  - 1's are "pulled up"
- Some working figures
  - R_p = 10 kΩ
  - C_{cap} = 100 pF
  - V_{DD} = 5 V
  - V_{in_high} = 3.5 V
- Recall for RC circuit
  - $V_{cap}(t) = V_{DD}(1-e^{-t/\tau})$
  - Where $\tau = RC$

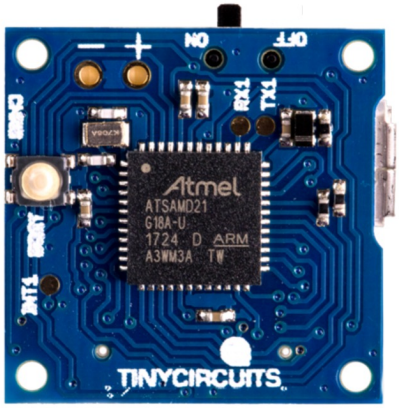


Practically I2C can do at most 400kbps

Exercise: Bus bit rate vs Useful data rate

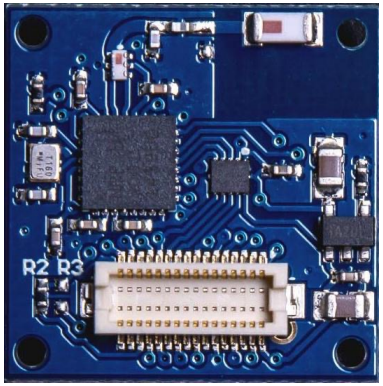
- An I2C “transactions” involves the following bits
  - <S><A6:A0><R/W><A><D7:D0><A><F>
- Which of these actually carries useful data?
  - <S><A6:A0><R/W><A><D7:D0><A><F>
- So, if a bus runs at 400 kHz
  - What is the clock period?
  - What is the data throughput (i.e. data-bits/second)?
  - What is the bus “efficiency”?

CSE190 Fall 2023



Lecture 16

Direct Memory Access



Wireless Embedded Systems

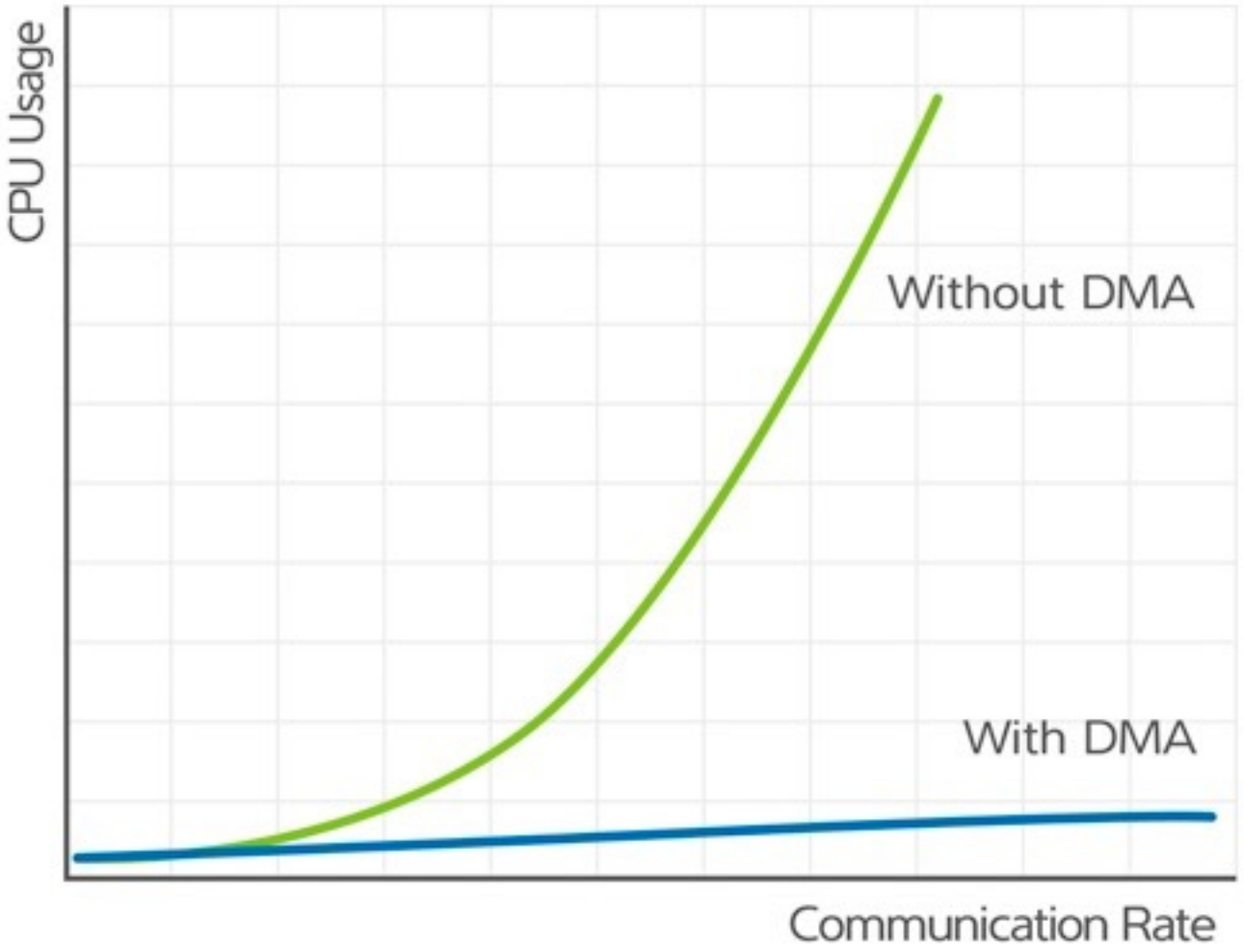
Aaron Schulman

How do you move data to and from peripherals in MMIO?

- Moving data to a peripheral?
  - CPU instructions that write data from RAM to MMIO
  - `I2C->DATA = X[1];`

- Moving data from a peripheral?
  - CPU instructions that read MMIO to RAM
  - `X[2] = I2C->DATA;`

Why do we need DMA?



Why do we need DMA?

Polling and Interrupt driven I/O concentrates on data transfer between the processor and I/O devices.

An instruction to transfer (`mov data,R0`) only occurs after the processor determines that the device is ready

- Either by polling a status flag in the device register or
- Waits for the device to send an interrupt request.

Why do we need DMA?

Considerable overhead is incurred by MMIO, because several program instructions must be executed for each data word transferred.

Moving things is a waste of CPU instructions:

– Instructions are needed to increment memory address and keeping track of how many bytes are moved.

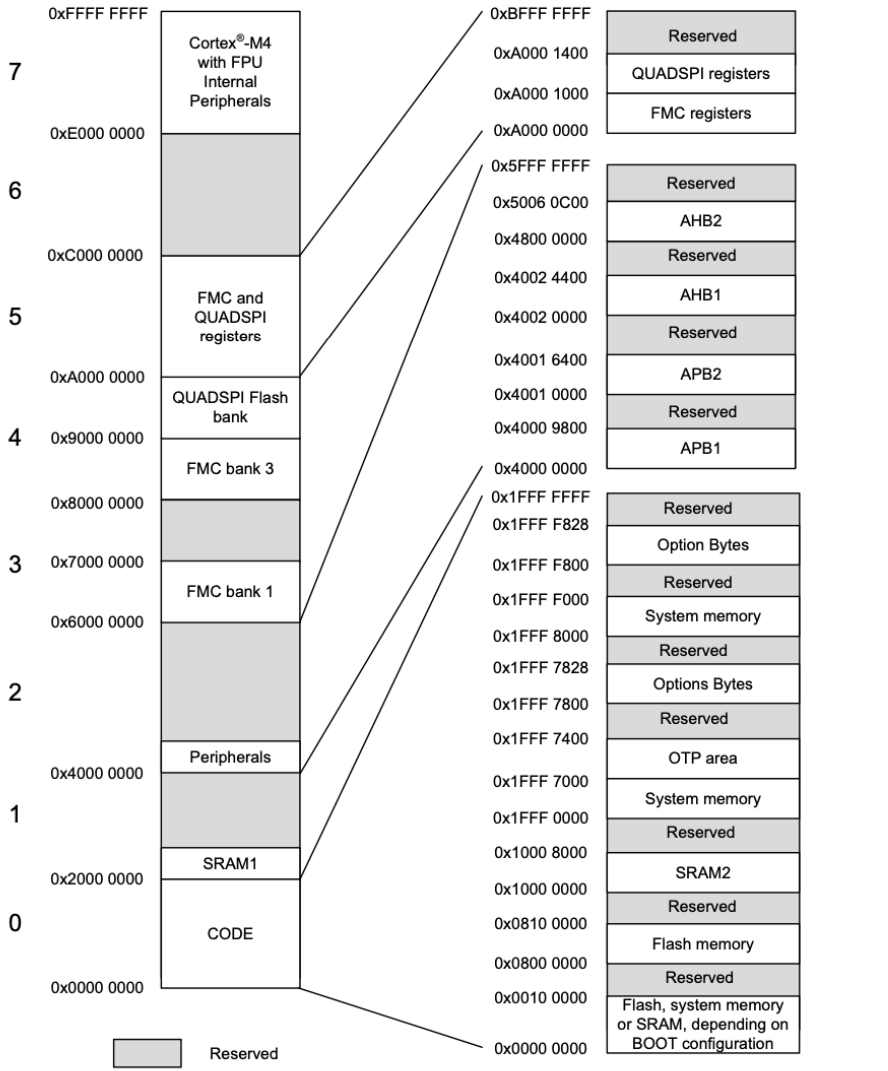
Direct Memory Access (DMA)

- To transfer large blocks of data at high speed, an alternative approach is used, the DMA periperhal.

- Blocks of data are transferred between an external device and the main memory, without continuous intervention by the processor.

- ***It's just another peripheral, but it's only job is moving data.***

DMA is possible because of linear memory addressing



MS46961V

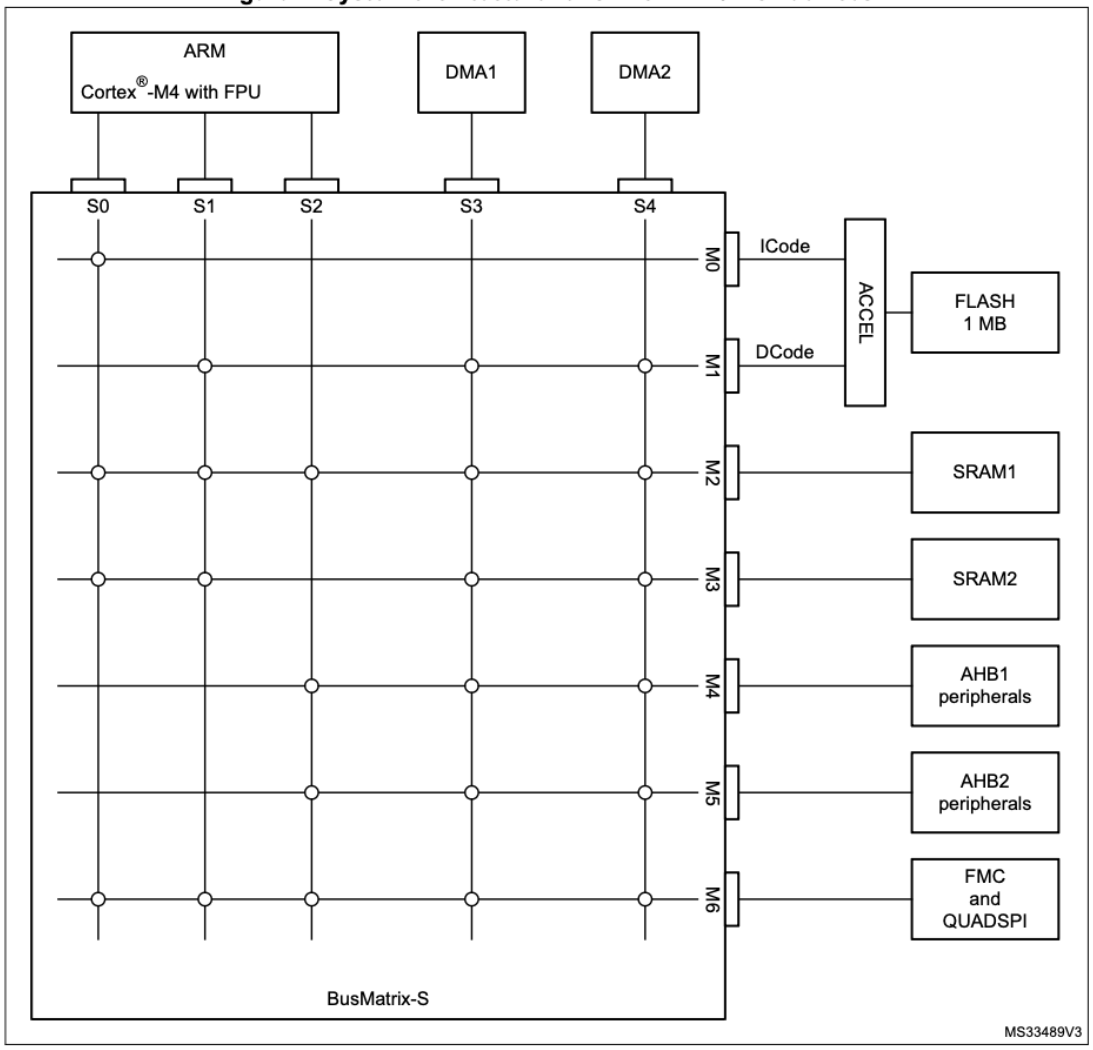
DMA Controller

- DMA controller is connected to the internal I/O bus.

- Performs the functions that would normally be carried out by the processor when access main memory. For each word transferred, it provides the memory address and all the bus signals that control data transfer.

Memory Architecture

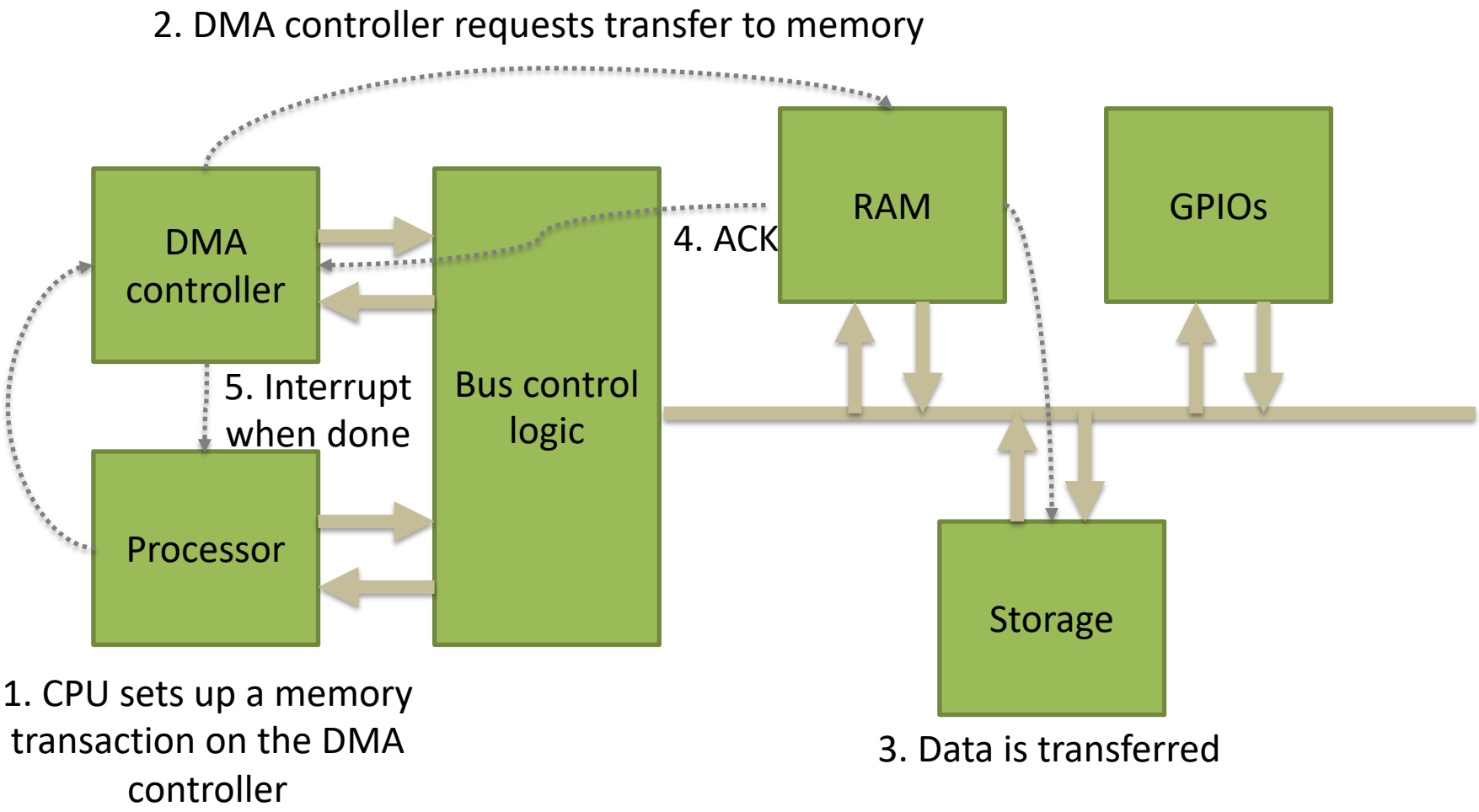
Figure 1. System architecture for STM32L47x/L48x devices



The DMA Transaction in a nutshell

1. Device wishing to perform DMA asserts the microcontroller's bus request signal.
2. Processor completes the current bus cycle and then asserts the bus grant signal to the device.
3. The device then asserts the bus grant ack signal.
4. The DMA device performs the transfer from the source to destination address.
5. Once the DMA operations have been completed, the device releases the bus by asserting the bus release signal.
6. Microcontroller acknowledges the bus release and resumes its bus cycles from the point it left off.

Use of DMA Controllers



Buffers and Arbitration

- Most DMAs have a data storage buffer – peripherals can send a burst of data faster than RAM memory can handle (as long as this happens infrequently)

- Bus Arbitration is needed to resolve conflicts with more than one device (2 DMAs or DMA and processor, etc..) try to use the bus to access main memory.

Bus Arbitration

- Bus Primary – the device that is allowed to initiate bus transfers on the bus at any given time. When the current primary relinquishes control, another device can acquire this status.

- Bus Arbitration – the process by which the next device to become bus primary is selected and bus primaryship is transferred to it.

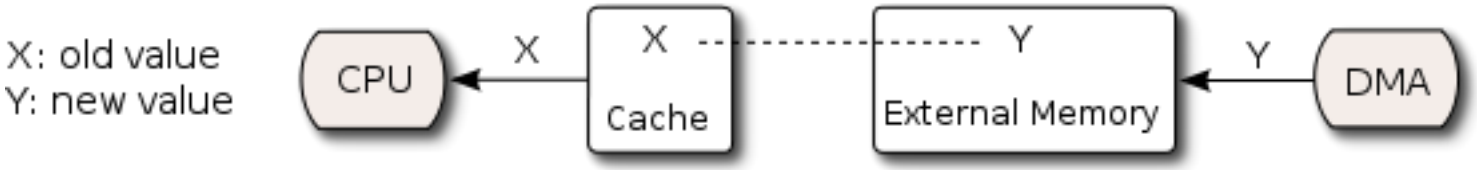
Arbitration Approaches

- Centralized – a single arbiter performs the arbitration.

- Distributed – all devices participate in the selection of the next bus master.

Cache coherency problems

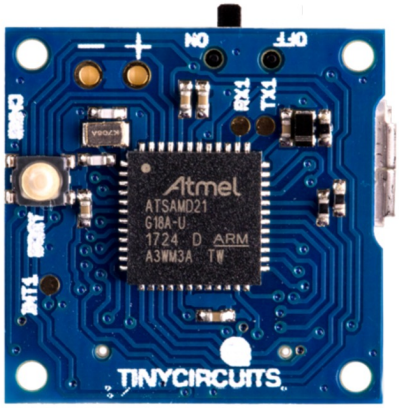
Imagine a CPU equipped with a cache and an external memory that can be accessed directly by devices using DMA. When the CPU accesses location X in the memory, the current value will be stored in the cache. Subsequent operations on X will update the cached copy of X, but not the external memory version of X, assuming a write-back cache. If the cache is not flushed to the memory before the next time a device tries to access X, the device will receive a stale value of X.

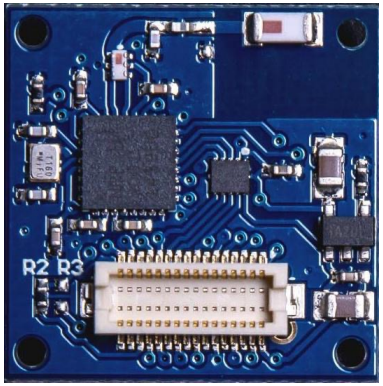


CSE190 Fall 2023

Lecture 17

Interfacing with The Analog World





Wireless Embedded Systems

Aaron Schulman

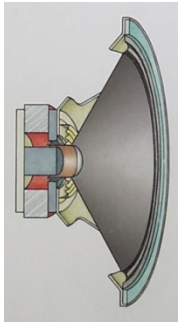
What does digital mean?

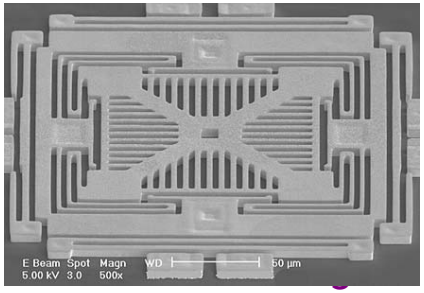
- Digital computer / microcontroller
- Digital watch
- Digital accelerometer
- Digital TV
- Digital cell phone?

- Operates on discrete signal “e.g., 0s and 1s”
- Not continuous!

We live in an analog world

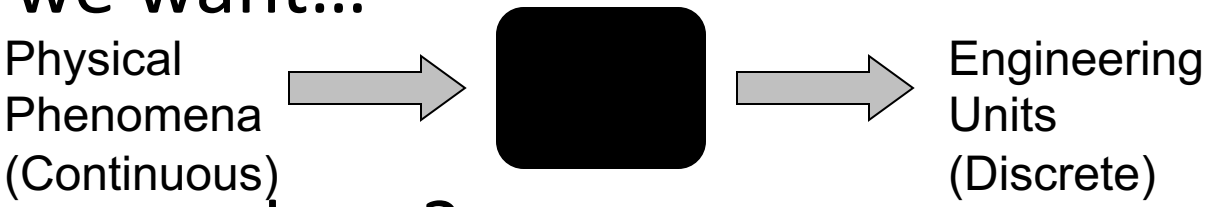
- Everything in the physical world is an analog signal
  - Continuous!
  - Sound, light, temperature, pressure
- Need to convert into electrical signals
  - Transducers: converts one type of energy to another
    - Electro-mechanical, Photonic, Electrical, …
  - Examples
    - Microphone/speaker
    - Thermocouples
    - Accelerometers



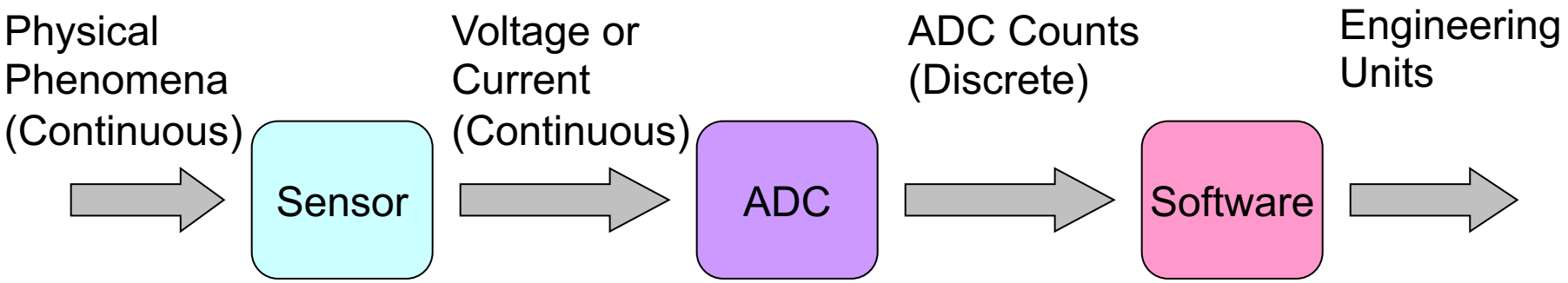


Going from analog to digital

- What we want…



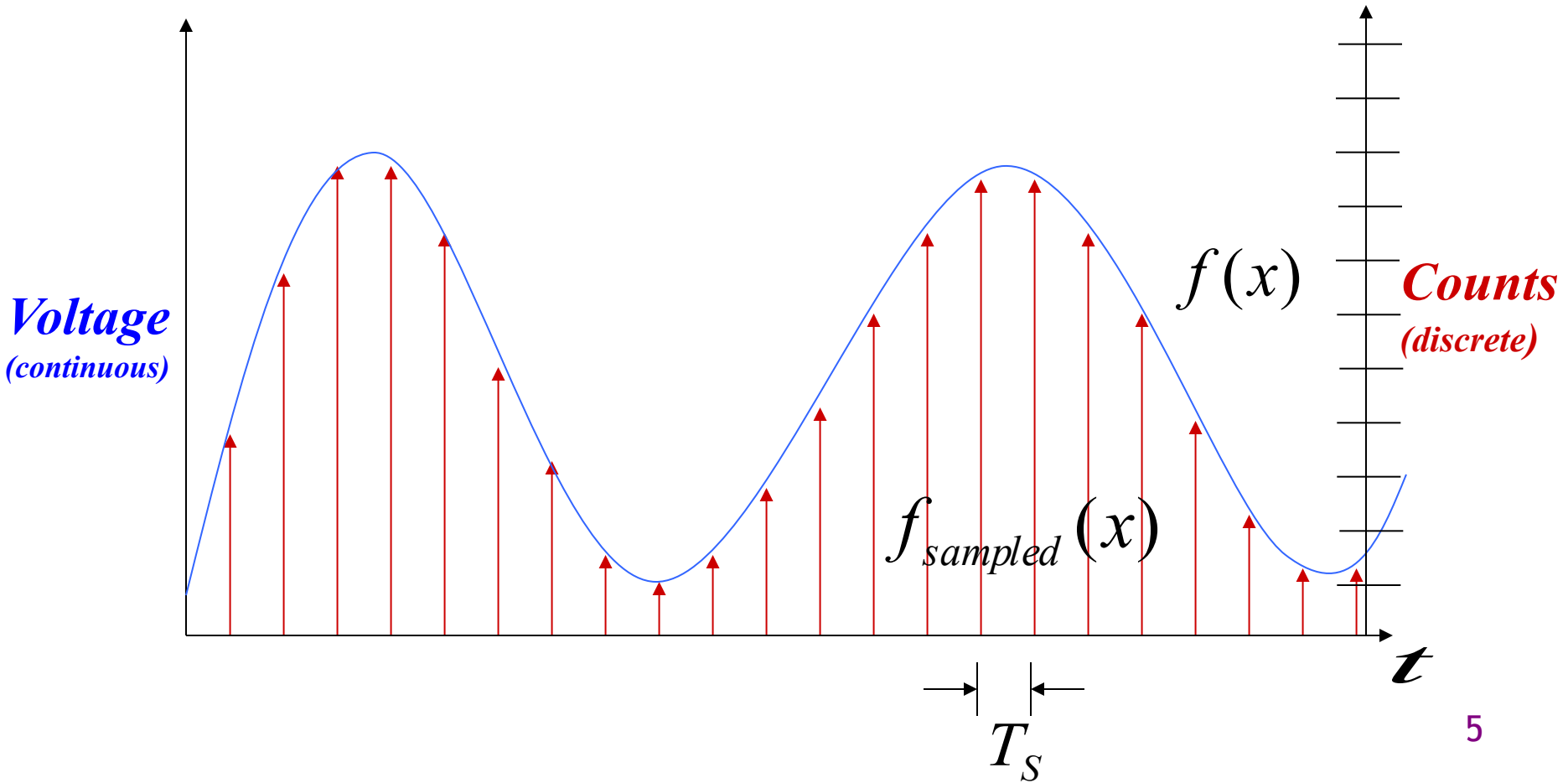
- How we get there?



Representing an analog signal in a digital microcontroller

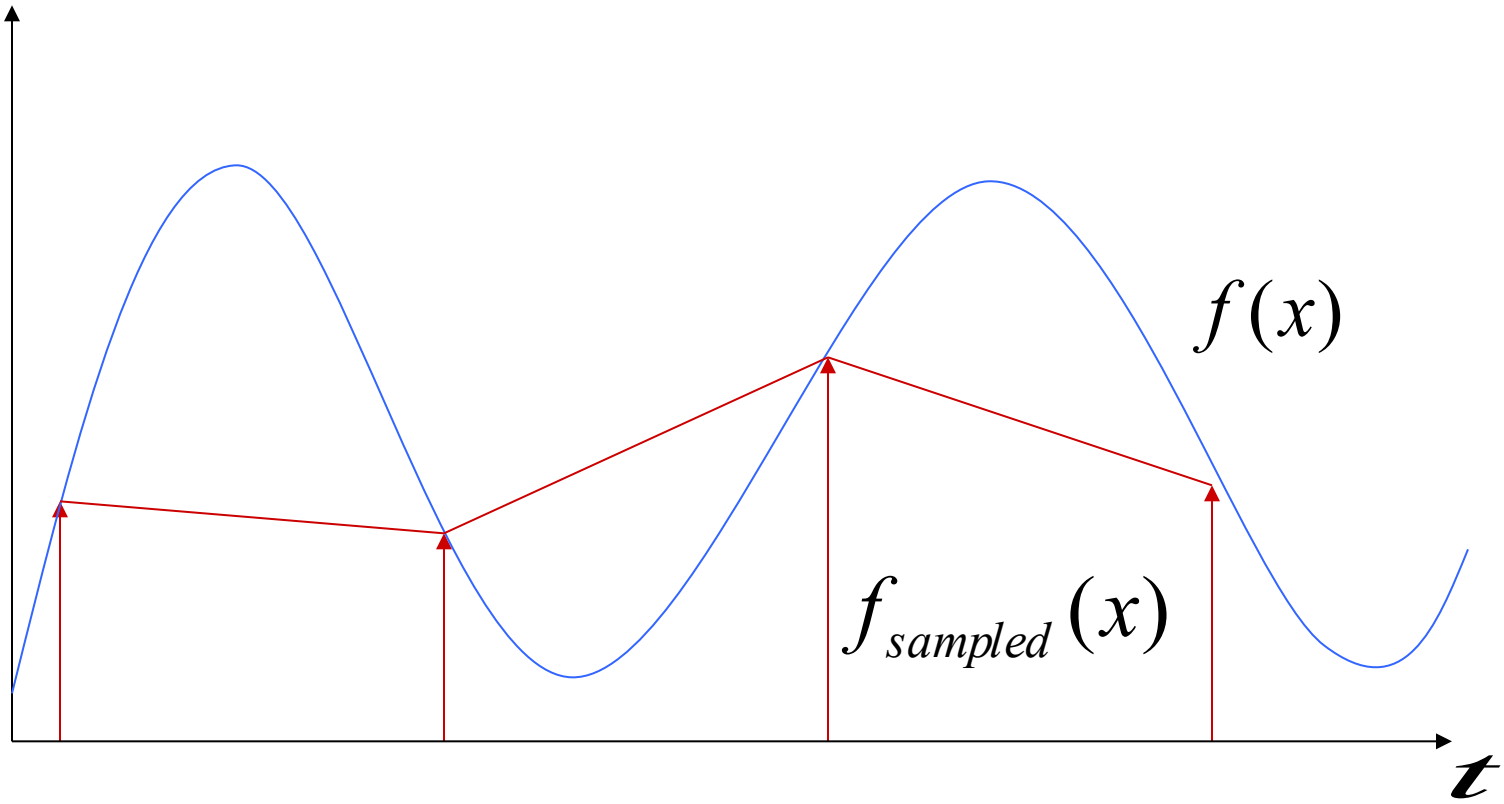
- How do we represent an analog signal (e.g. continuous voltage)?
  - As a time series of discrete values (binary integers)

    → On MCU: read ADC data register (counts) periodically (T_s)



Choosing the sample rate

- What sample rate do we need?
  - Too little: we can't reconstruct the signal we care about
  - Too much: waste computation, energy, resources



Shannon-Nyquist sampling theorem

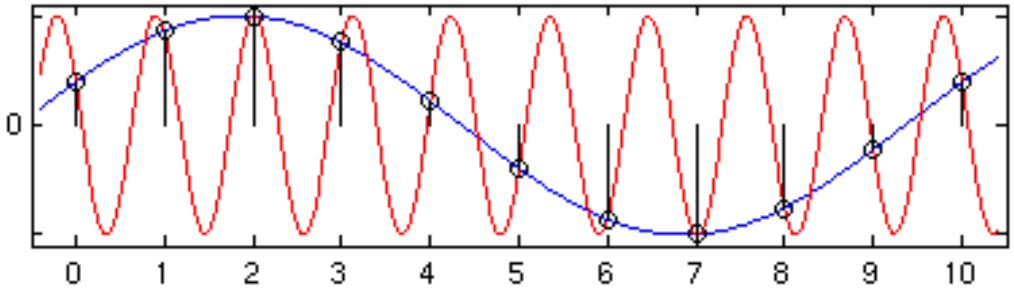
- *If a continuous-time signal $f(x)$ contains no frequencies higher than $f_{\max}$, it can be completely determined by discrete samples taken at a rate:*

$$f_{\text{samples}} > 2 f_{\max}$$

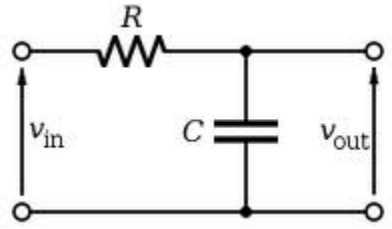
- Example:
  - Humans can process audio signals 20 Hz – 20 KHz
  - Audio CDs: sampled at 44.1 KHz

Use anti-aliasing filters on ADC inputs to ensure that Shannon-Nyquist is satisfied

- Aliasing (sampling too slow!)
  - Different frequencies are indistinguishable when they are sampled.



- Solution: Condition the input signal using a low-pass filter
  - Removes high-frequency components
  - (a.k.a. anti-aliasing filter)



Do I really need to filter my input signal?

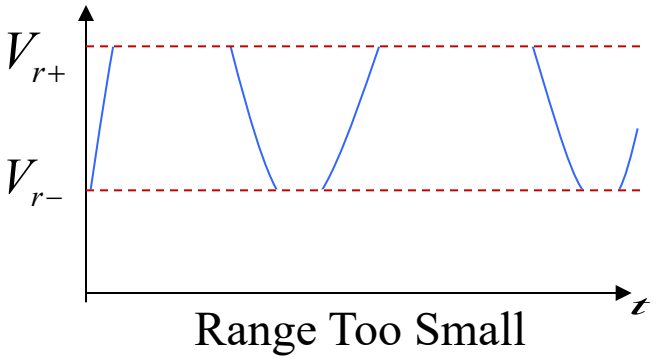
- Short answer: Yes.

- Longer answer: Yes, but sometimes it's already done for you.
  - Many (most?) ADCs have a pretty good analog filter built in.
  - Those filters typically have a cut-off frequency just above ½ their ***maximum*** sampling rate.
    - Which is great if you are using the maximum sampling rate, less useful if you are sampling at a slower rate.

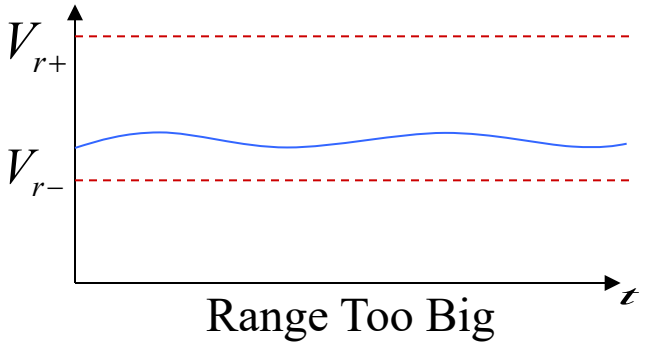
2nd digital problem: Choosing the ADC's range (amplitude)

- Fixed (discrete) # of bits (e.g. 8-bit ADC)
- Span a particular continuous input voltage range
- What do the sample values represent?
  - Some fraction within the range of values

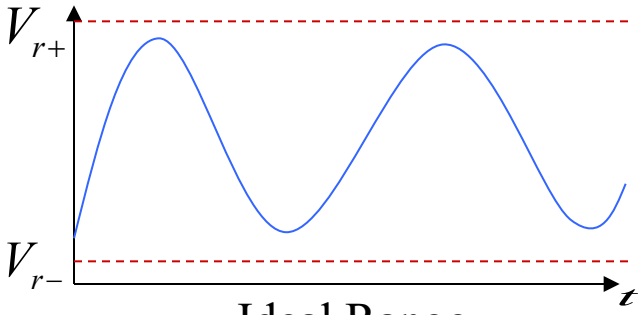
  → What range to use?



Range Too Small



Range Too Big



Ideal Range

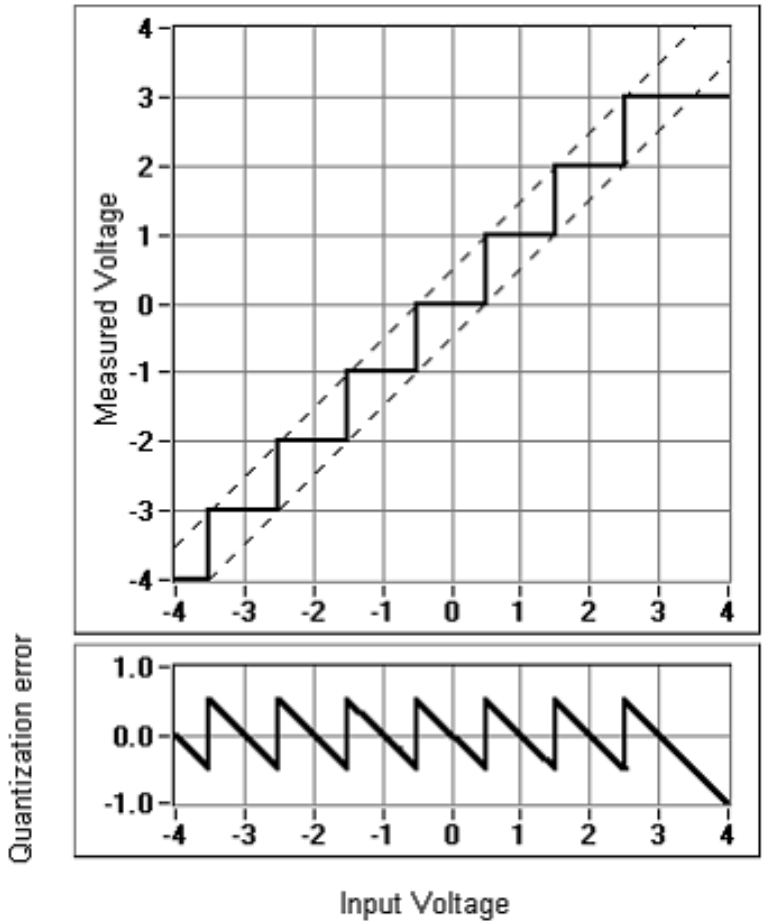
Choosing the granularity

- Resolution
  - Number of discrete values that represent a range of analog values
  - 12-bit ADC
    - 4096 values
    - Range / 4096 = Step

  Larger range ➔ less info / bit

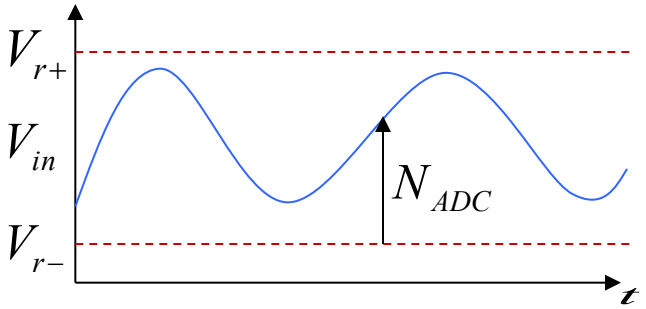
- Quantization Error
  - How far off discrete value is from actual
  - ½ LSB → Range / 8192

  Larger range ➔ larger error



Converting between voltages, ADC counts, and engineering units

- Converting: ADC counts ⇔ Voltage



$$N_{ADC} = 4095 \times \frac{V_{in} - V_{r-}}{V_{r+} - V_{r-}}$$

$$V_{in} = N_{ADC} \times \frac{V_{r+} - V_{r-}}{4095}$$

- Converting: Voltage ⇔ Engineering Units

$$V_{\text{TEMP}} = 0.00355(\text{TEMP}_{\text{C}}) + 0.986$$

$$\text{TEMP}_{\text{C}} = \frac{V_{\text{TEMP}} - 0.986}{0.00355}$$

A note about sampling and arithmetic*

- Converting values in fixed-point MCUs

$$V_{\text{TEMP}} = N_{ADC} \times \frac{V_{r+} - V_{r-}}{4095} \qquad \text{TEMP}_{\text{C}} = \frac{V_{\text{TEMP}} - 0.986}{0.00355}$$

```
float vtemp = adccount/4095 * 1.5;
float tempc = (vtemp-0.986)/0.00355;
```

→ vtemp = 0! Not what you intended, even when vtemp is a float!

→ tempc = -277 C

- Fixed point operations
  - Need to worry about underflow and overflow

- Floating point operations
  - They can be costly on the embedded system

Try it out for yourself…

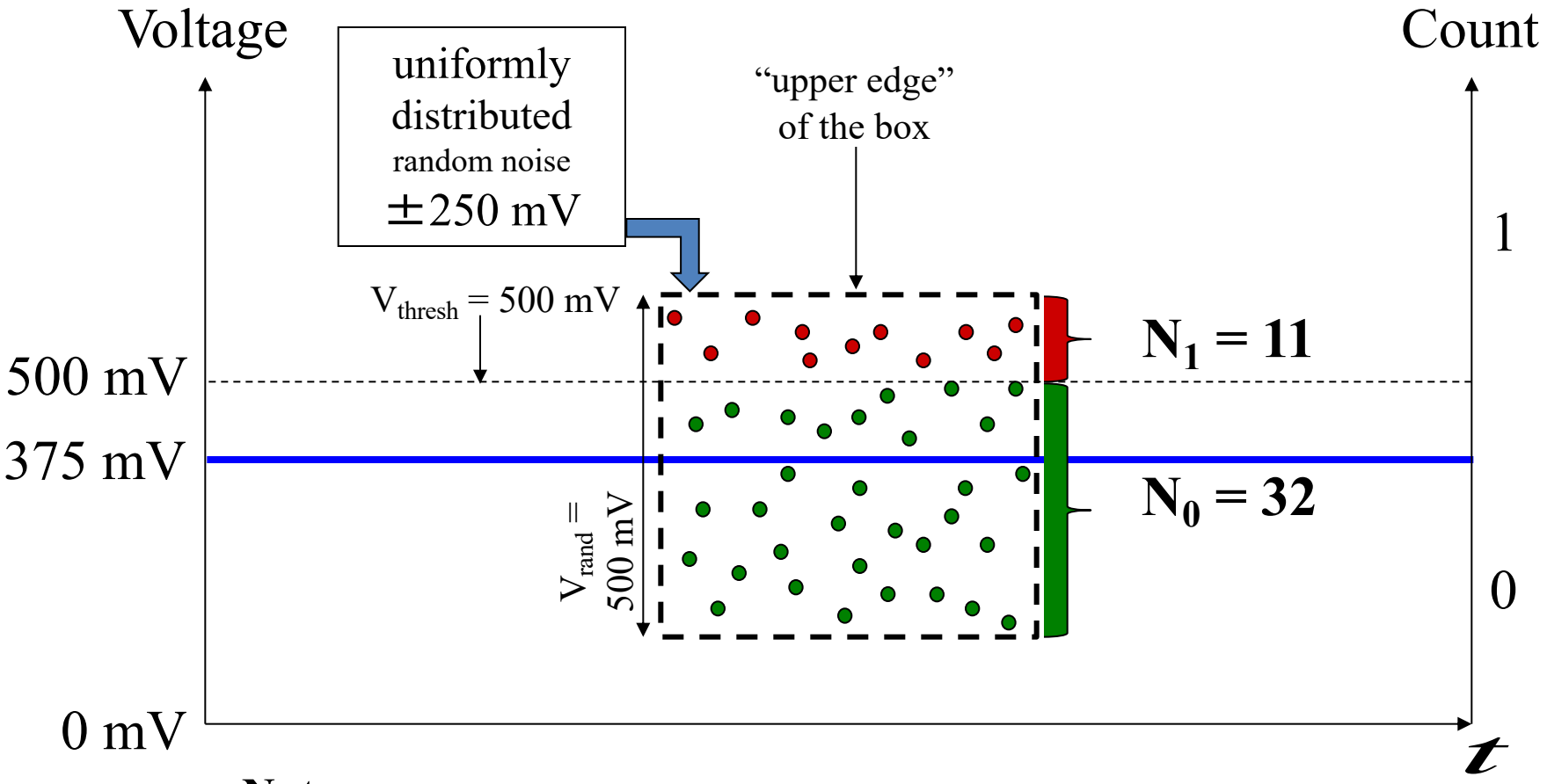
```
$ cat arithmetic.c
#include <stdio.h>

int main() {

  int adccount = 2048;
  float vtemp;
  float tempc;

  vtemp = adccount/4095 * 1.5;
  tempc = (vtemp-0.986)/0.00355;

  printf("vtemp: %f\n", vtemp);
  printf("tempc: %f\n", tempc);
}

$ gcc arithmetic.c

$ ./a.out
vtemp: 0.000000
tempc: -277.746490
```

Oversampling (sampling faster than Nyquist)

One interesting trick is that you can use oversampling to help reduce the impact of quantization error.

- Let's look at an example of oversampling plus dithering to get a 1-bit converter to do a much better job…

Oversampling a 1-bit ADC w/ noise & dithering (cont)



Note:

N_1 is the # of ADC counts that = 1 over the sampling window

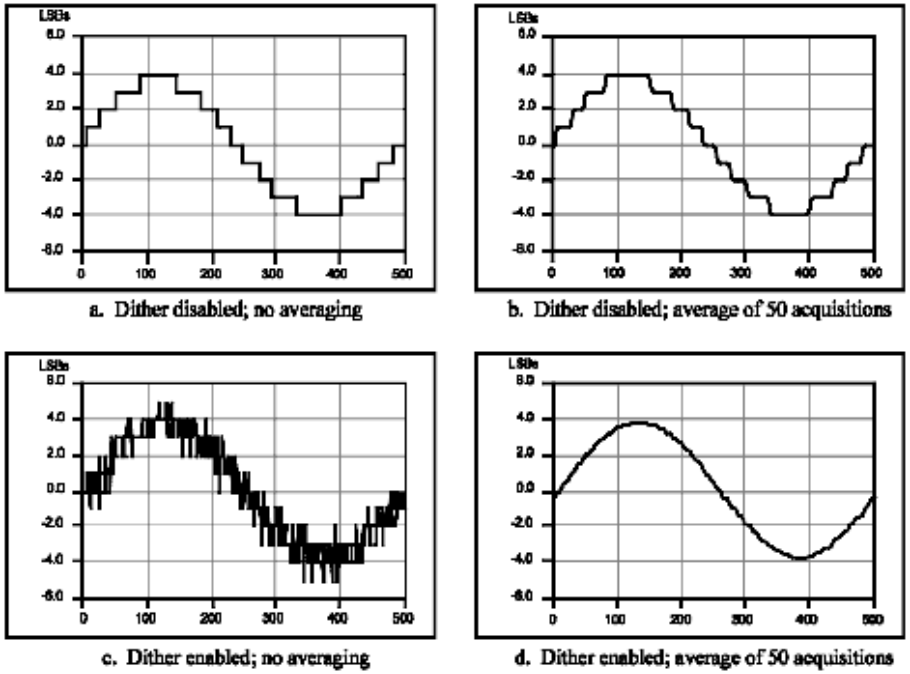
N_0 is the # of ADC counts that = 0 over the sampling window

Oversampling a 1-bit ADC w/ noise & dithering (cont)

- How to get more than 1-bit out of a 1-bit ADC?
- Add some noise to the input
- Do some math with the output
- Example
  - 1-bit ADC with 500 mV threshold
  - Vin = 375 mV → ADC count = 0
  - Add ±250 mV uniformly distributed random noise to Vin
  - Now, roughly
    - 25% of samples (N_1) ≥ 500 mV → ADC count = 1
    - 75% of samples (N_0) < 500 mV → ADC count = 0

Can use dithering to deal with quantization

- Dithering (introducing noise)
  - Quantization errors can result in large-scale patterns that don't accurately describe the analog signal
  - Oversample and dither
  - Introduce random (white) noise to randomize the quantization error.

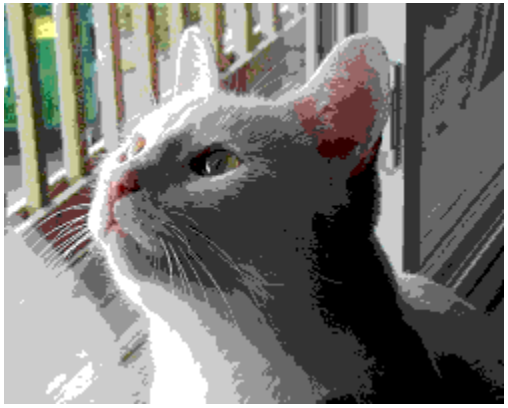


a. Dither disabled; no averaging

b. Dither disabled; average of 50 acquisitions

c. Dither enabled; no averaging

d. Dither enabled; average of 50 acquisitions



Direct Samples

Dithered Samples

Selection of a DAC (digital to analog converter)

- **Error/Accuracy/Resolution:** Quantizing error represents the difference between an actual analog value and its digital representation. Ideally, the quantizing error should not be greater than ± 1⁄2 LSB.

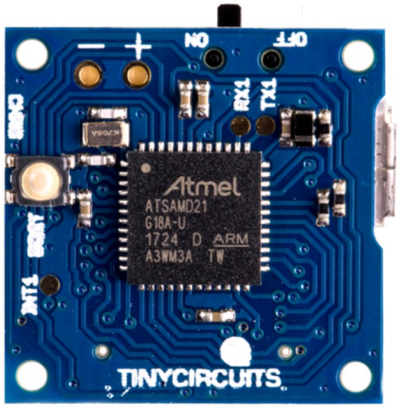
Output Voltage Range -> Input Voltage Range

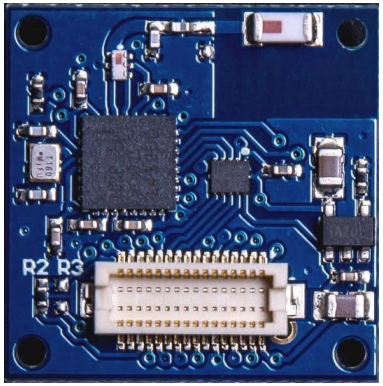
- **Output Settling Time** -> Conversion Time
- **Output Coding** (usually binary)

CSE190 Fall 2023

Lecture 18

Analog and Wireless



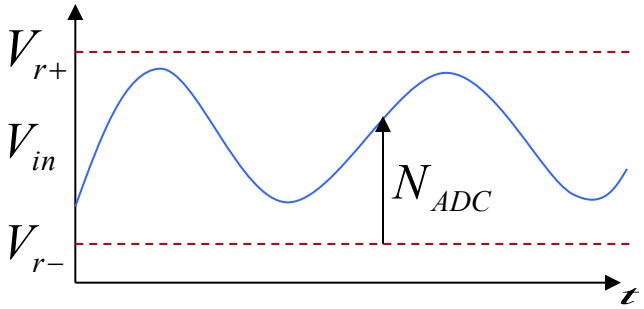


Wireless Embedded Systems

Aaron Schulman

Converting between voltages, ADC counts, and engineering units

- Converting: ADC counts ⇔ Voltage



$$N_{ADC} = 4095 \times \frac{V_{in} - V_{r-}}{V_{r+} - V_{r-}}$$

$$V_{in} = N_{ADC} \times \frac{V_{r+} - V_{r-}}{4095}$$

- Converting: Voltage ⇔ Engineering Units

$$V_{\text{TEMP}} = 0.00355(\text{TEMP}_{\text{C}}) + 0.986$$

$$\text{TEMP}_{\text{C}} = \frac{V_{\text{TEMP}} - 0.986}{0.00355}$$

A note about sampling and arithmetic*

- Converting values in fixed-point MCUs

$$V_{\text{TEMP}} = N_{ADC} \times \frac{V_{r+} - V_{r-}}{4095} \qquad \text{TEMP}_{\text{C}} = \frac{V_{\text{TEMP}} - 0.986}{0.00355}$$

```
float vtemp = adccount/4095 * 1.5;
float tempc = (vtemp-0.986)/0.00355;
```

→ vtemp = 0! Not what you intended, even when vtemp is a float!

→ tempc = -277 C

- Fixed point operations
  - Need to worry about underflow and overflow

- Floating point operations
  - They can be costly on the embedded system

Try it out for yourself…

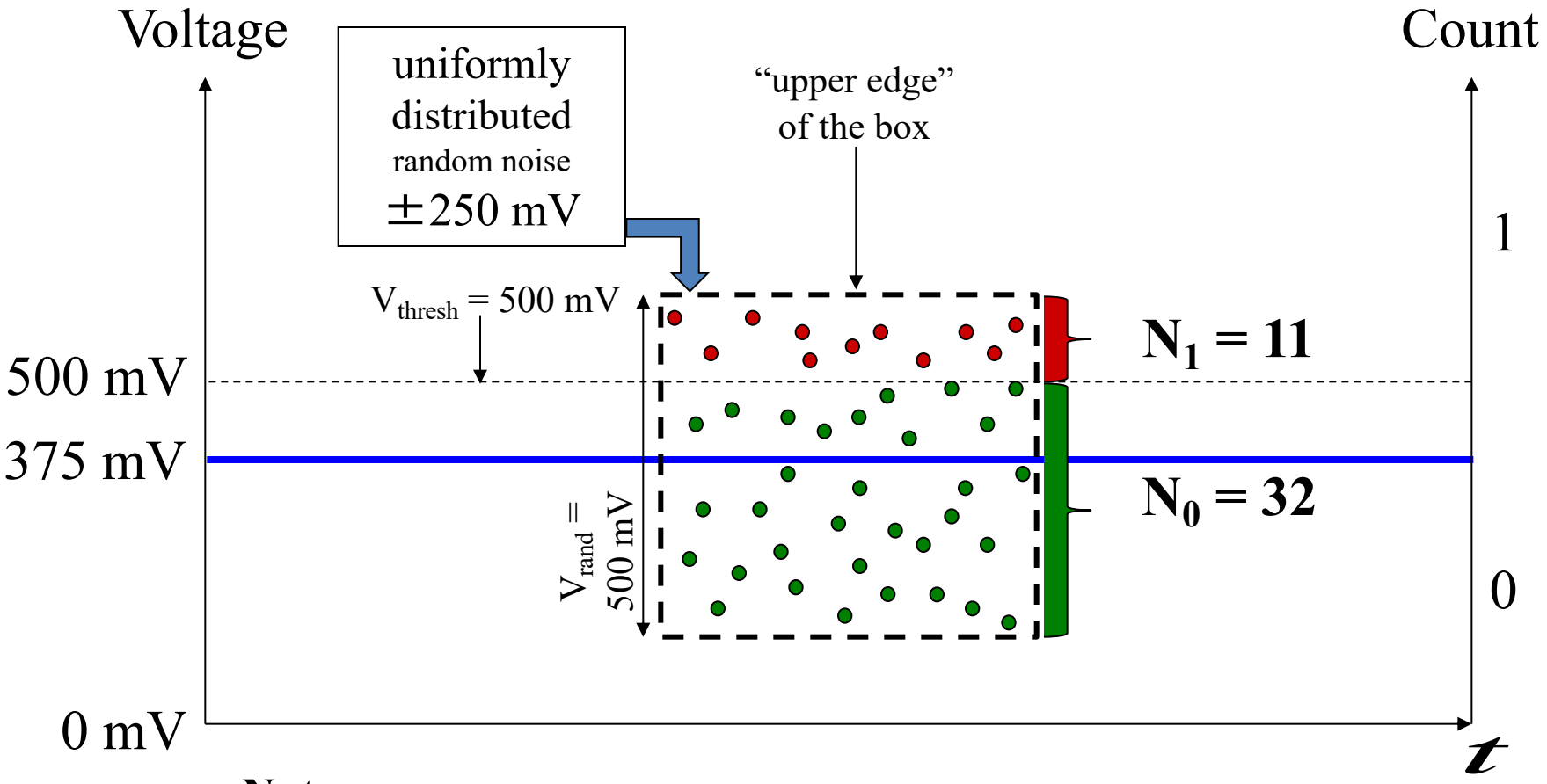
```
$ cat arithmetic.c
#include <stdio.h>

int main() {

  int adccount = 2048;
  float vtemp;
  float tempc;

  vtemp = adccount/4095 * 1.5;
  tempc = (vtemp-0.986)/0.00355;

  printf("vtemp: %f\n", vtemp);
  printf("tempc: %f\n", tempc);
}

$ gcc arithmetic.c

$ ./a.out
vtemp: 0.000000
tempc: -277.746490
```

Oversampling (sampling faster than Nyquist)

One interesting trick is that you can use oversampling to help reduce the impact of quantization error.

- Let's look at an example of oversampling plus dithering to get a 1-bit converter to do a much better job…

Oversampling a 1-bit ADC w/ noise & dithering (cont)



Note:

N_1 is the # of ADC counts that = 1 over the sampling window

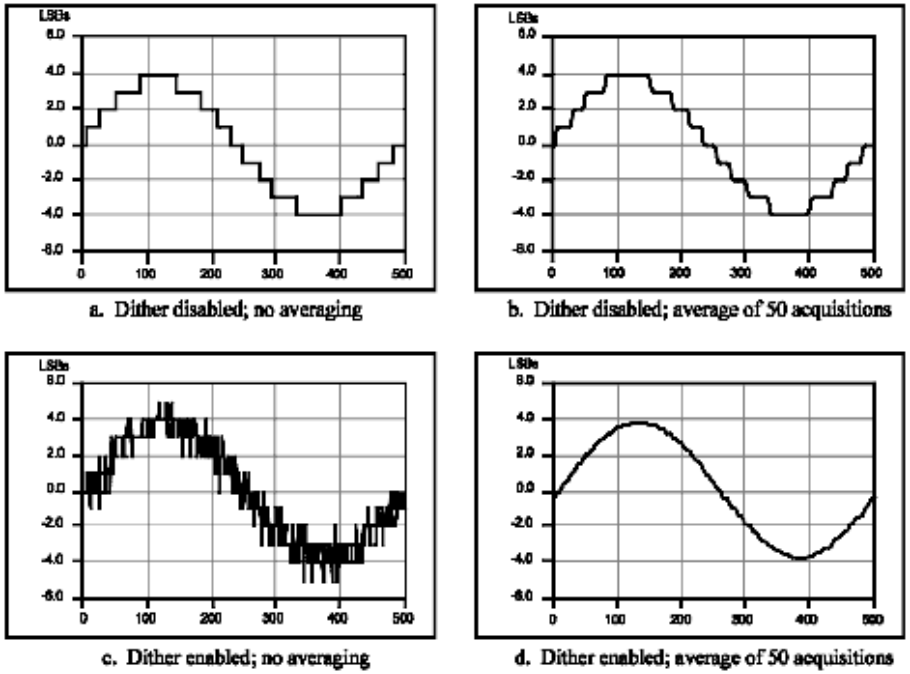
N_0 is the # of ADC counts that = 0 over the sampling window

Oversampling a 1-bit ADC w/ noise & dithering (cont)

- How to get more than 1-bit out of a 1-bit ADC?
- Add some noise to the input
- Do some math with the output
- Example
  - 1-bit ADC with 500 mV threshold
  - Vin = 375 mV → ADC count = 0
  - Add ±250 mV uniformly distributed random noise to Vin
  - Now, roughly
    - 25% of samples (N_1) ≥ 500 mV → ADC count = 1
    - 75% of samples (N_0) < 500 mV → ADC count = 0

Can use dithering to deal with quantization

- Dithering (introducing noise)
  - Quantization errors can result in large-scale patterns that don't accurately describe the analog signal
  - Oversample and dither
  - Introduce random (white) noise to randomize the quantization error.

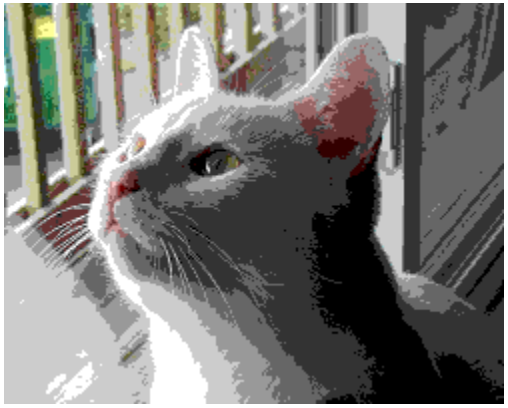


a. Dither disabled; no averaging

b. Dither disabled; average of 50 acquisitions

c. Dither enabled; no averaging

d. Dither enabled; average of 50 acquisitions



Direct Samples

Dithered Samples

Selection of a DAC (digital to analog converter)

- **Error/Accuracy/Resolution:** Quantizing error represents the difference between an actual analog value and its digital representation. Ideally, the quantizing error should not be greater than ± 1⁄2 LSB.

Output Voltage Range -> Input Voltage Range

- **Output Settling Time** -> Conversion Time
- **Output Coding** (usually binary)

Wireless communication is ubiquitous

- Bluetooth/WiFi/Cellular
- Device-to-Device (Bluetooth-Low-Energy)
  Device-to-infrastructure (LTE or WiFi)
- Bluetooth Low Energy
  the most popular wireless protocol.

Outline

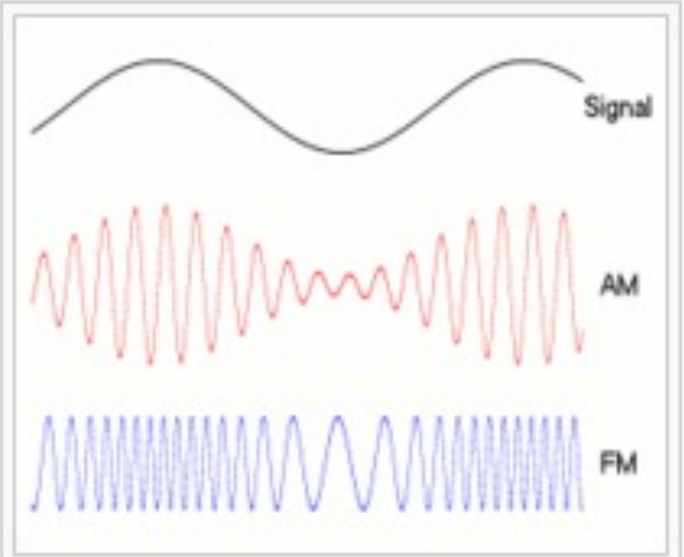
- What are radios
  - How do they work?
- Fundamental characteristics
  - Design tradeoffs
- Common radio standards/protocols for indoor applications
  - Where characteristics fall under (above)
- Emerging radio standards/protocols for outdoor Internet-of-Things applications
  - Why the design requirement of IoT radios is different

What are Radios?

- A device that enables wireless transmission of signals
  - Electromagnetic wave
  - Transmitter encodes signal and receiver decodes it

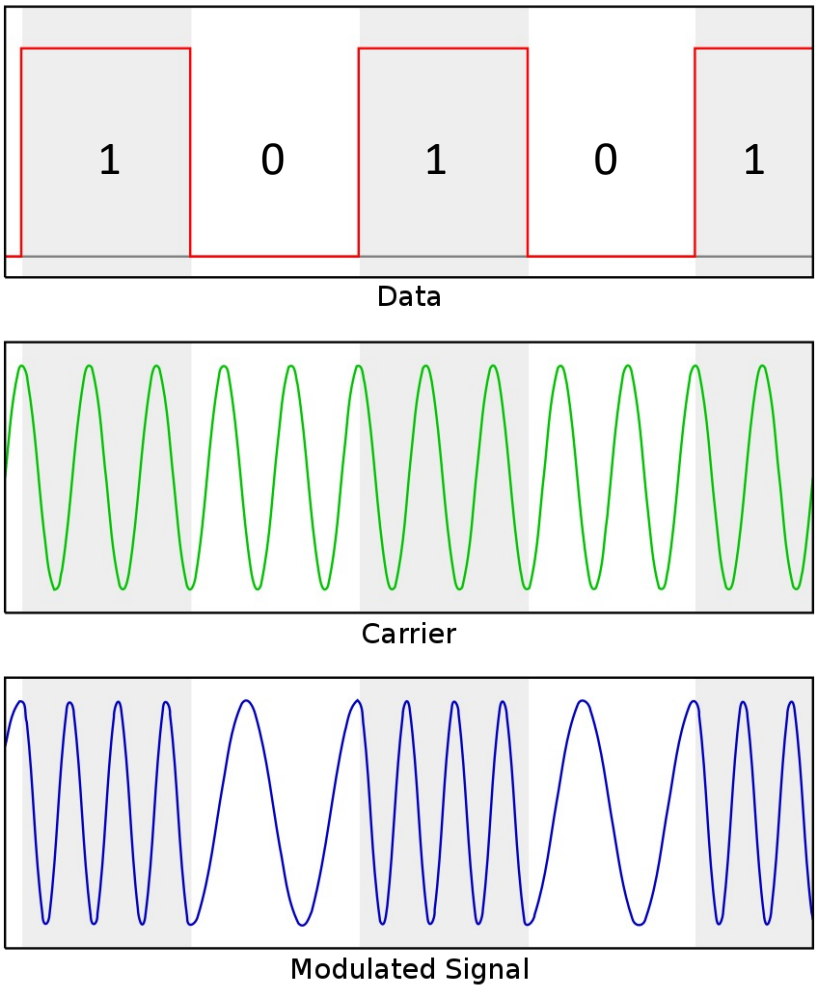
How Radios Work - Transmitting

- Modulation
  - Converts digital bits to an analog signal
  - Encodes bits as changes in a **carrier frequency**:
  - Frequency, Amplitude, Phase, etc.



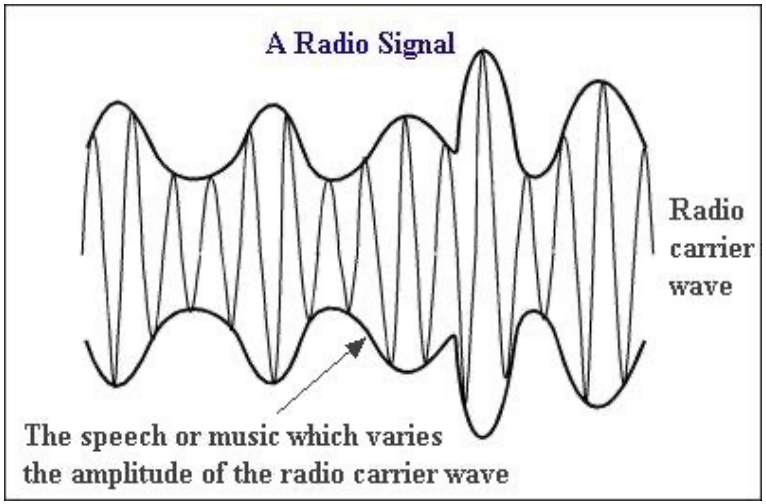
A low-frequency message signal (top) may be carried by an AM or FM radio wave.

Example of modulating digital data onto an analog signal



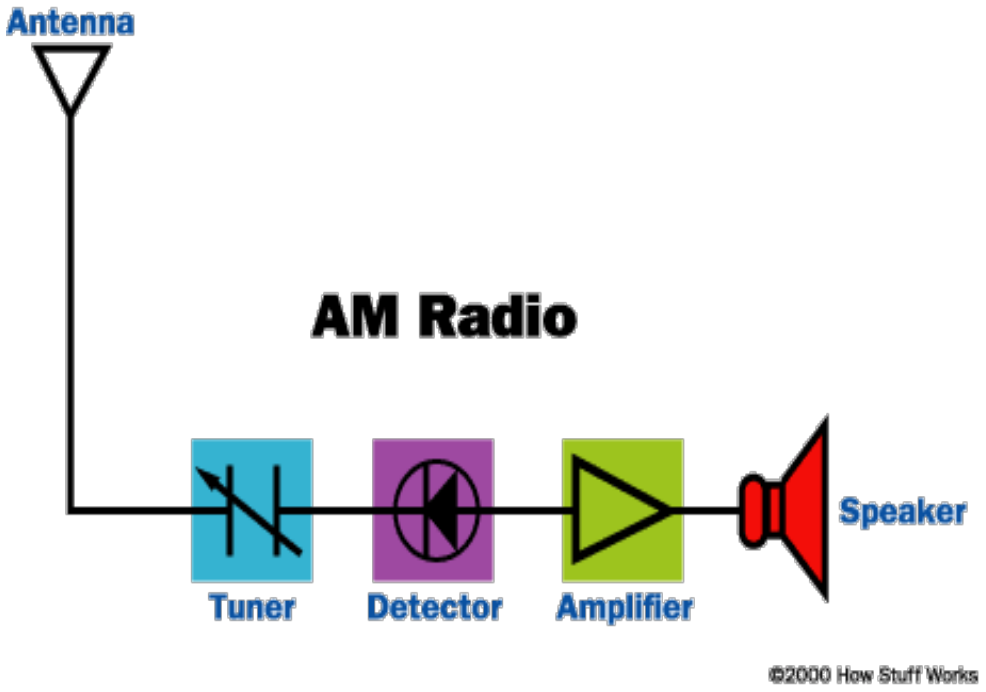
How Radios Work - Receiving

- Simplest is Envelope Detection
  - Detect changes in carrier freq.
- Complex require synchronization
- All require filtering
- Signals must be demodulated



How Radios Work – Receiving (AM Radio Example)

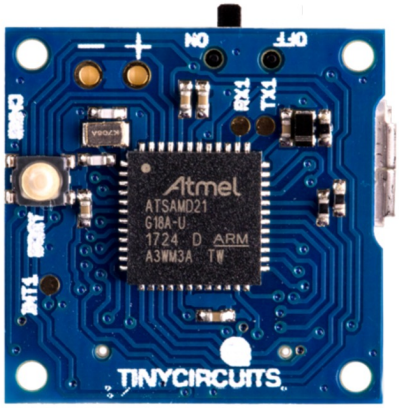
- Antenna picks up modulated radio waves
- Tuner filters out specific frequency ranges
- Amplitude variations detected with demodulation
- Amplifier strengthens the clipped signal and sends it through the speaker



Radio Characteristics

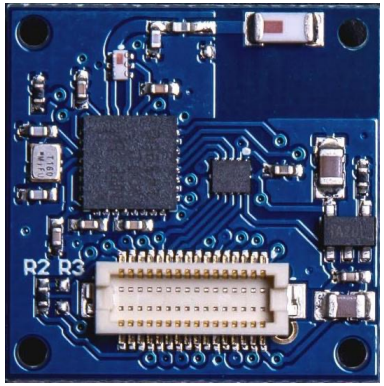
- Why so many protocols for indoor and outdoor applications?
- All radios have to make tradeoffs
  - Short vs. long distance
  - High vs. low power/energy
  - High vs. low speeds
  - Large vs. small number of devices
  - Device-to-device, device-to-infrastructure
  - Indoor vs. outdoor usages

CSE190 Winter 2023



Lecture 19

Wireless



Wireless Embedded Systems

Aaron Schulman

Wireless communication is ubiquitous

- Bluetooth/WiFi/Cellular
- Device-to-Device (Bluetooth-Low-Energy)
  Device-to-infrastructure (LTE or WiFi)
- Bluetooth Low Energy
  the most popular wireless protocol.

Outline

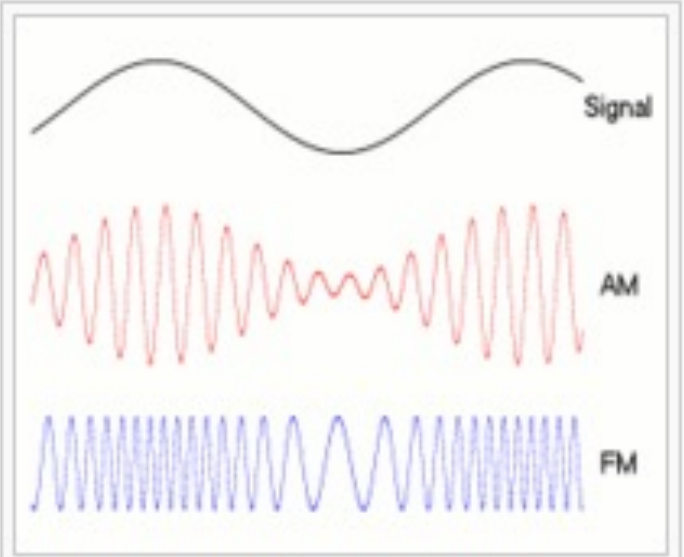
- What are radios
  - How do they work?
- Fundamental characteristics
  - Design tradeoffs
- Common radio standards/protocols for indoor applications
  - Where characteristics fall under (above)
- Emerging radio standards/protocols for outdoor Internet-of-Things applications
  - Why the design requirement of IoT radios is different

What are Radios?

- A device that enables wireless transmission of signals
  - Electromagnetic wave
  - Transmitter encodes signal and receiver decodes it

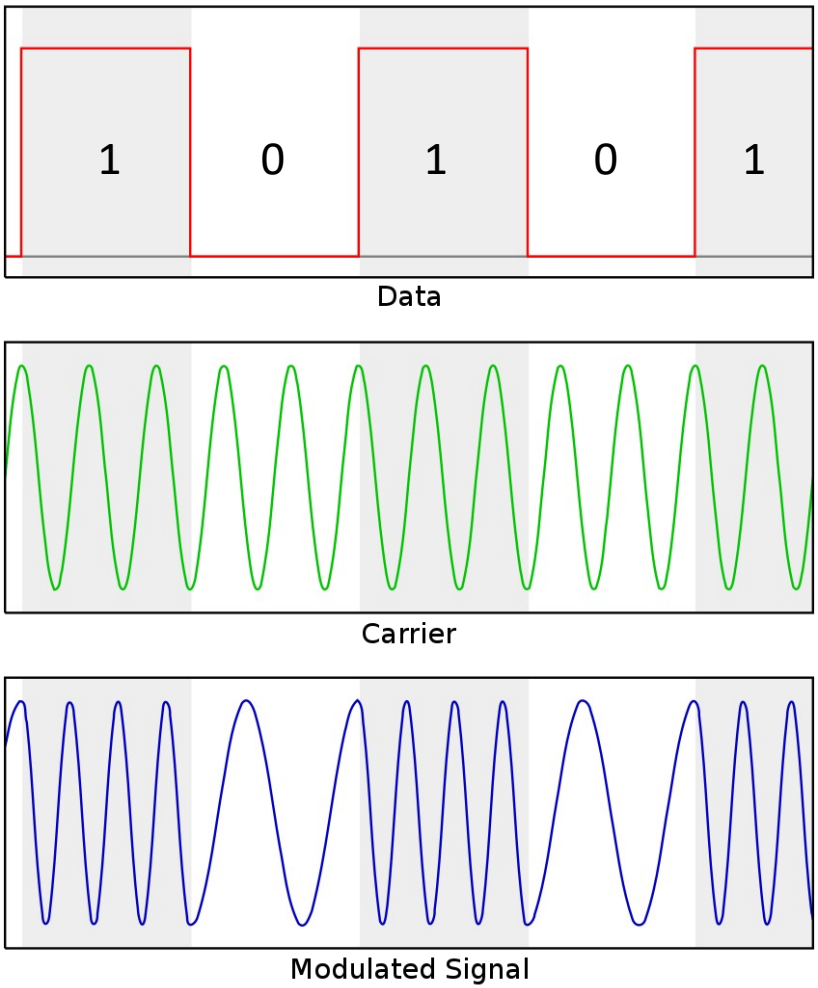
How Radios Work - Transmitting

- Modulation
  - Converts digital bits to an analog signal
  - Encodes bits as changes in a **carrier frequency**:
  - Frequency, Amplitude, Phase, etc.



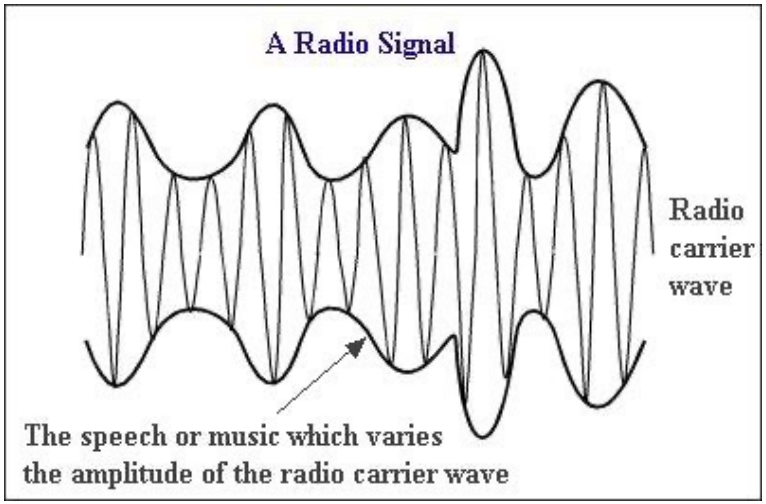
A low-frequency message signal (top) may be carried by an AM or FM radio wave.

Example of modulating digital data onto an analog signal



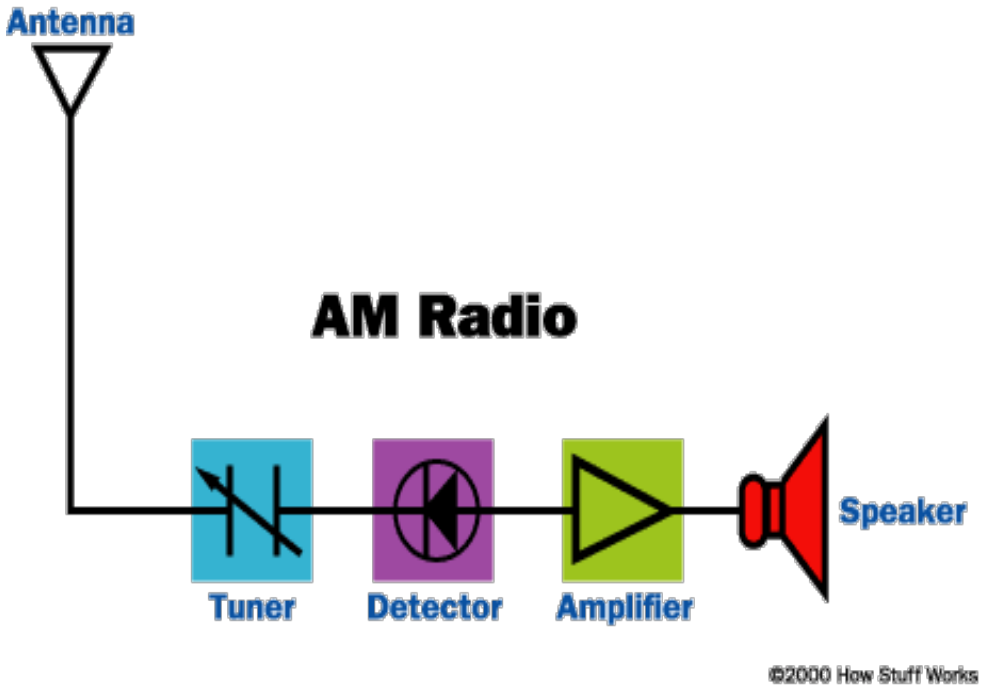
How Radios Work - Receiving

- Transmitted signals must be demodulated
- Simplest is Envelope Detection
  - Detect changes in carrier freq. amplitude
- Most complex receivers require synchronization
- All require filtering



How Radios Work – Receiving (AM Radio Example)

- Antenna picks up modulated radio waves
- Tuner filters out specific frequency ranges
- Amplitude variations detected with demodulation
- Amplifier strengthens the clipped signal and sends it through the speaker



Wireless Protocol Characteristics

- Why so many protocols for indoor and outdoor applications?
- All radios have to make tradeoffs
  - Short vs. long distance
  - High vs. low power/energy
  - High vs. low speeds
  - Large vs. small number of devices
  - Device-to-device, device-to-infrastructure
  - Indoor vs. outdoor usages

Common Radio Protocols

Radios for indoor IoT applications

- Design requirements
  - Short range
  - High data rate
  - Small number of devices
- Common protocols
  - Bluetooth/Low Energy
  - ZigBee
  - Ant
  - WiFi

Radios for outdoor IoT applications

- Design requirements
  - Long range
  - Low data rate
  - Large number of devices
  - Low energy consumption
- Common protocols
  - GSM/GPRS
  - LTE
- Emerging protocols
  - Sigfox/LoRA
  - Narrow band LTE
  - Backscatter

Bluetooth

- Radio band: 2.4-2.48 GHz
- Average 1 Mbps - Up to 3 Mbps
- Supports point-to-point and point-to-multipoint
  - Creates personal area networks (PANs/Piconets)
  - Connects up to 8 devices simultaneously
- Minimal interference between devices
  - **Devices alter frequencies arbitrarily after packet exchanges - up to 1600 times/second - frequency hopping**
- 3 classes of Bluetooth transmit power
- Frequency hopping communication

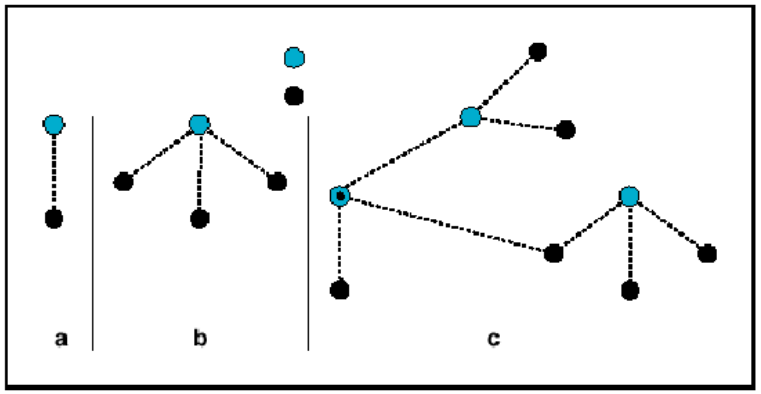


Figure 1.2: Piconets with a single slave operation (a), a multi-slave operation (b) and a scatternet operation (c).

Class	Maximum Power	Operating Range
Class 1	100mW (20dBm)	100 meters
Class 2	2.5mW (4dBm)	10 meters
Class 3	1mW (0dBm)	1 meter

Frequency hopping communication was invented by actress Hedy Lamar

UNITED STATES PATENT OFFICE

2,292,387

SECRET COMMUNICATION SYSTEM

Hedy Kiesler Markey, Los Angeles, and George Antheil, Manhattan Beach, Calif.

Application June 10, 1941, Serial No. 397,412

6 Claims. (Cl. 250—2)

This invention relates broadly to secret communication systems involving the use of carrier waves of different frequencies, and is especially useful in the remote control of dirigible craft.

Fig. 2 is a schematic diagram of the apparatus at a receiving station;

Fig. 3 is a schematic diagram illustrating a starting circuit for starting the motors at the

Bluetooth Applications

- Wireless communication between devices
  - Mobile phones, laptops, cameras, gaming controllers, computer peripherals, etc
- Short range sensor transmission
- Share multimedia - pictures, video, music
- A2DP - Advanced Audio Distribution Profile
  - Stream audio wirelessly

Bluetooth Low Energy

From 2001 – 2006 Nokia asked:

How do we design a radio that can transmit short bursts of data for months or years *only being powered by a coin cell battery?*

The answer is: Keep the radio asleep mode most of the time!

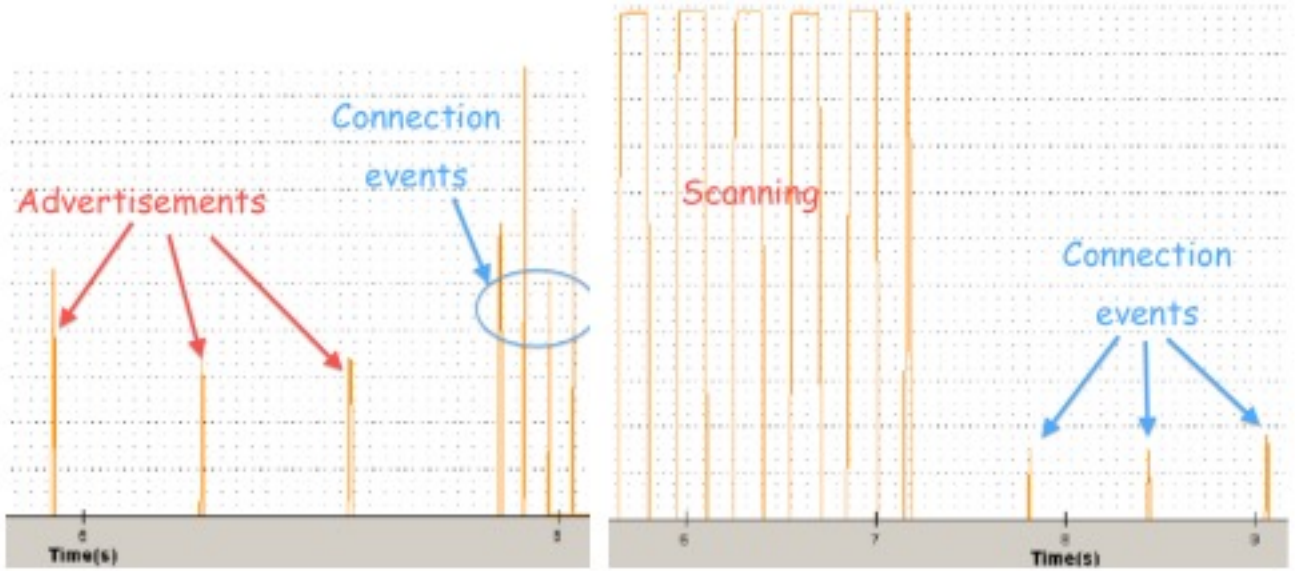
1. Advertise on only one of three channels (less freq. hopping)
2. Transmit quickly at 1 Mbit/s
3. Make the minimum time to send data only 3 msec
4. Make a very predictable time when the device accepts connections
5. Limit the max transmit power to 10 mW
6. However, don't sacrifice security: AES 128-bit

What tradeoffs were made?

The protocol is designed for transmitting tiny data

- 4 operations: Read, Write, Notify, Indicate
- Maximum of 20 bytes of data per packet

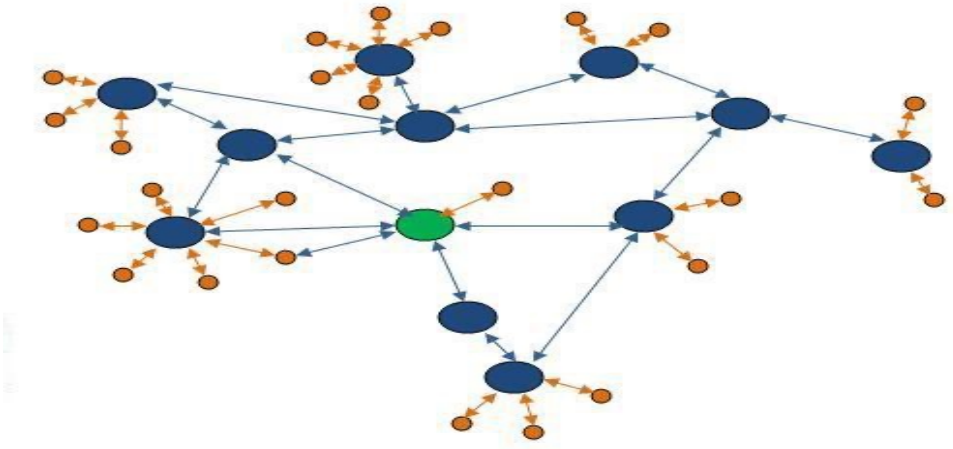
(Plots are power over time)



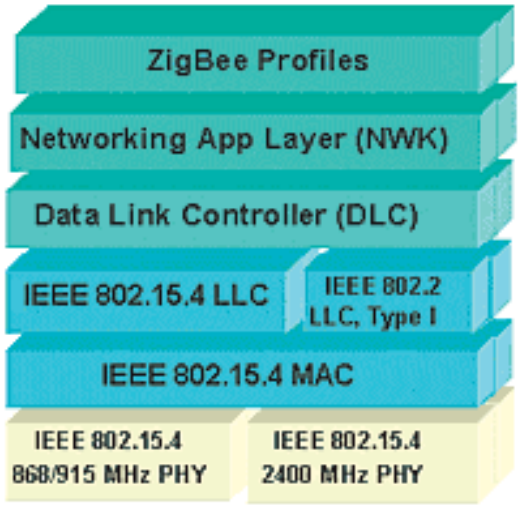
From: How Low Energy is Bluetooth Low Energy - Siekkinen et al.

Zigbee/802.15.4

- Zigbee is built on top of 802.15.4
- Radio bands: 868MHz in Europe, 915MHz in US and Australia. 2.4GHz else worldwide.
- Low data-rate - 250 kbps, low power – Up to 1000 days
- Transmits over longer distances through mesh networks



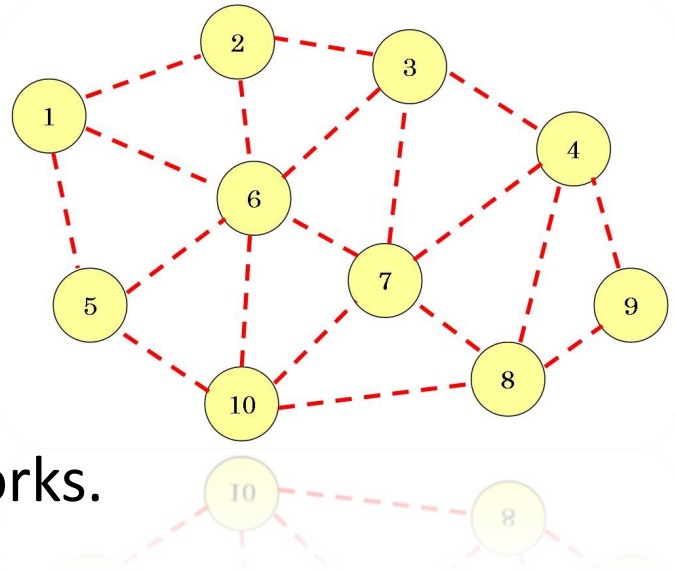
Zigbee/802.15.4 mesh network and device types



Zigbee is usually used in mesh networks

- A mesh network consists of a series of nodes.

- Each node must acquire and transmit its own data, as well as act as a relay for other nodes to propagate data.

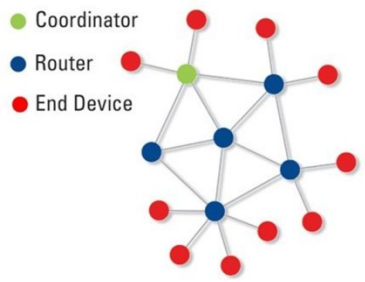


- ZigBee devices often form Mesh Networks.

- Examples: Wireless light switching, Music school practice rooms.

Image Source: <http://kf5czo.blogspot.com/2012/03/ham-radio-and-mesh-networks.html>

Mesh Networking



- Advantages of Mesh Networking:
  - Allows devices to communicate to multiple other dev the network.
  - Multiple paths to destination – greater flexibility against interference.
  - Allows overall network to grow to larger physical sizes than possible with point-to-point networks.

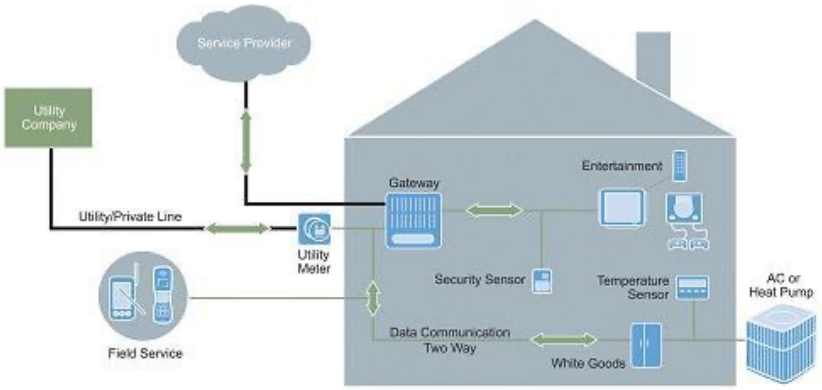
- Mesh Characteristics:
  - *Self-forming* – ZigBee devices can establish communication pathways when new devices appear.
  - *Self-healing* – If a node is removed from the network (either intentionally or not) the remaining network will look to establish alternate routes of communication.

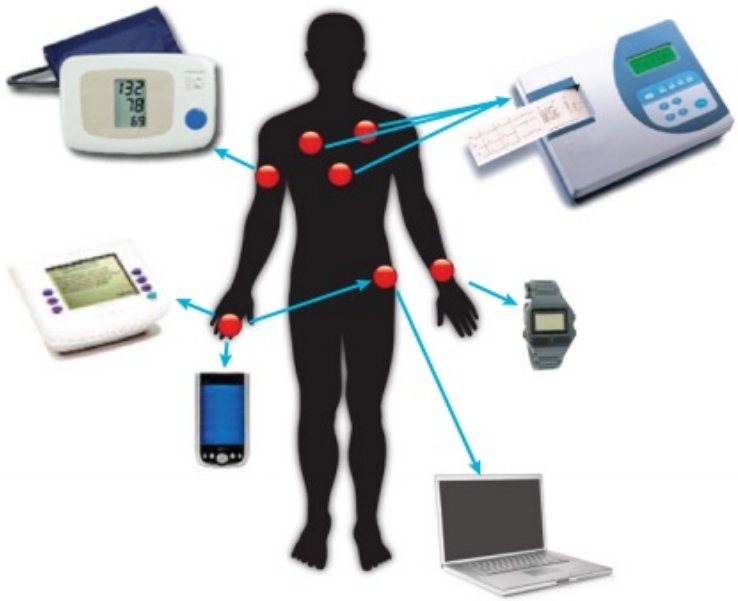
Why ZigBee?

- Low Power, Cost, and Size
- Straightforward configuration
- Good support and documentation
  - Lots of products already on the market
- Mesh Networking
- Lends itself well to a variety of applications
- Very low wakeup time
  - 30mS (Zigbee) vs. up to 3S (Bluetooth)

Zigbee/802.15.4 Applications

- Wireless environmental sensors
  - Temperature, pressure, sound, luminous intensity
- Medical devices
  - Glucose meters, heart monitors
- Household automation
  - Security/temperature controllers
  - Smoke/motion detectors





Bluetooth, Zigbee, and WiFi contend

- Competes with Wi-Fi for bandwidth..
  - Only four usable bands in Wi-Fi intensive scenarios

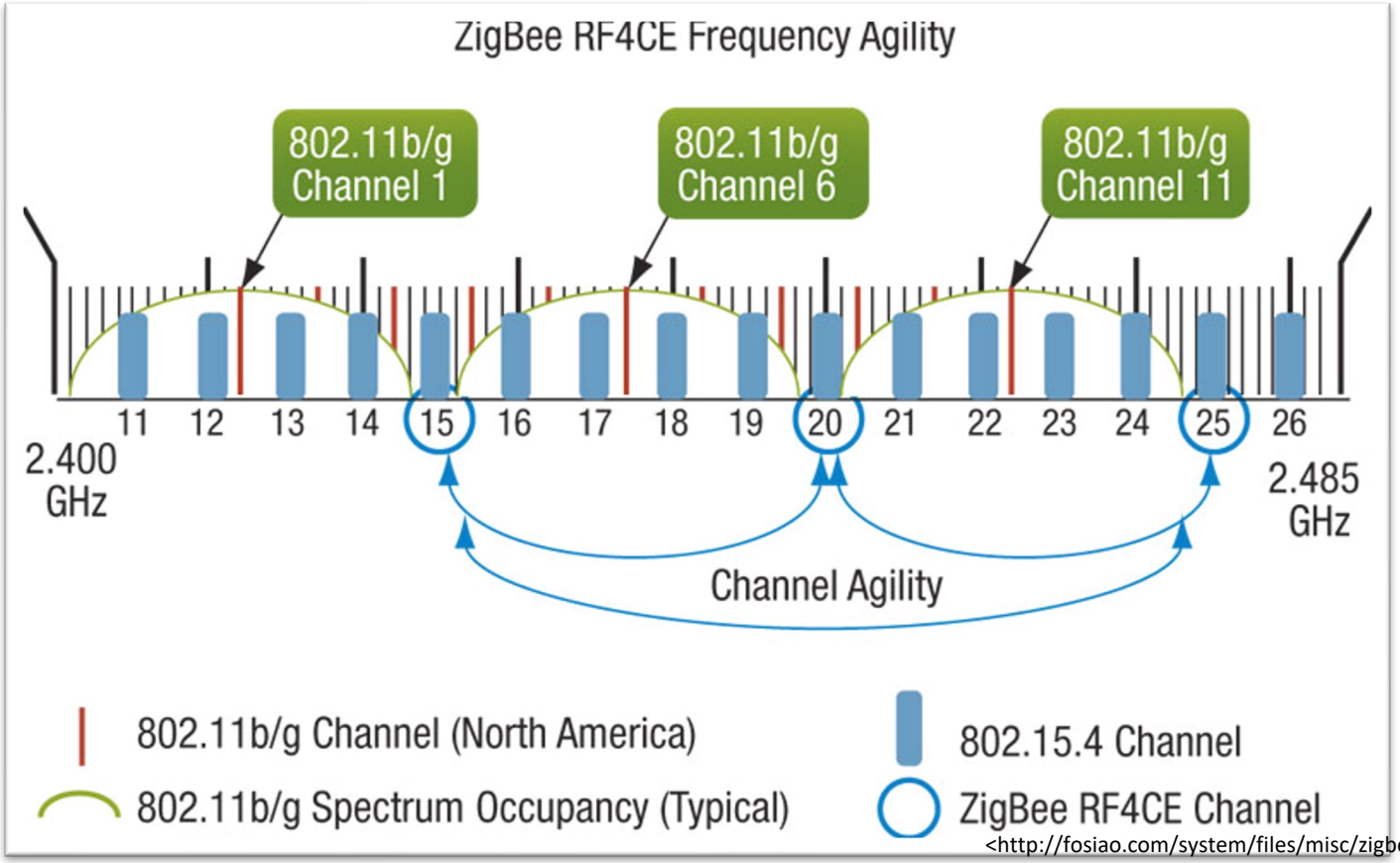
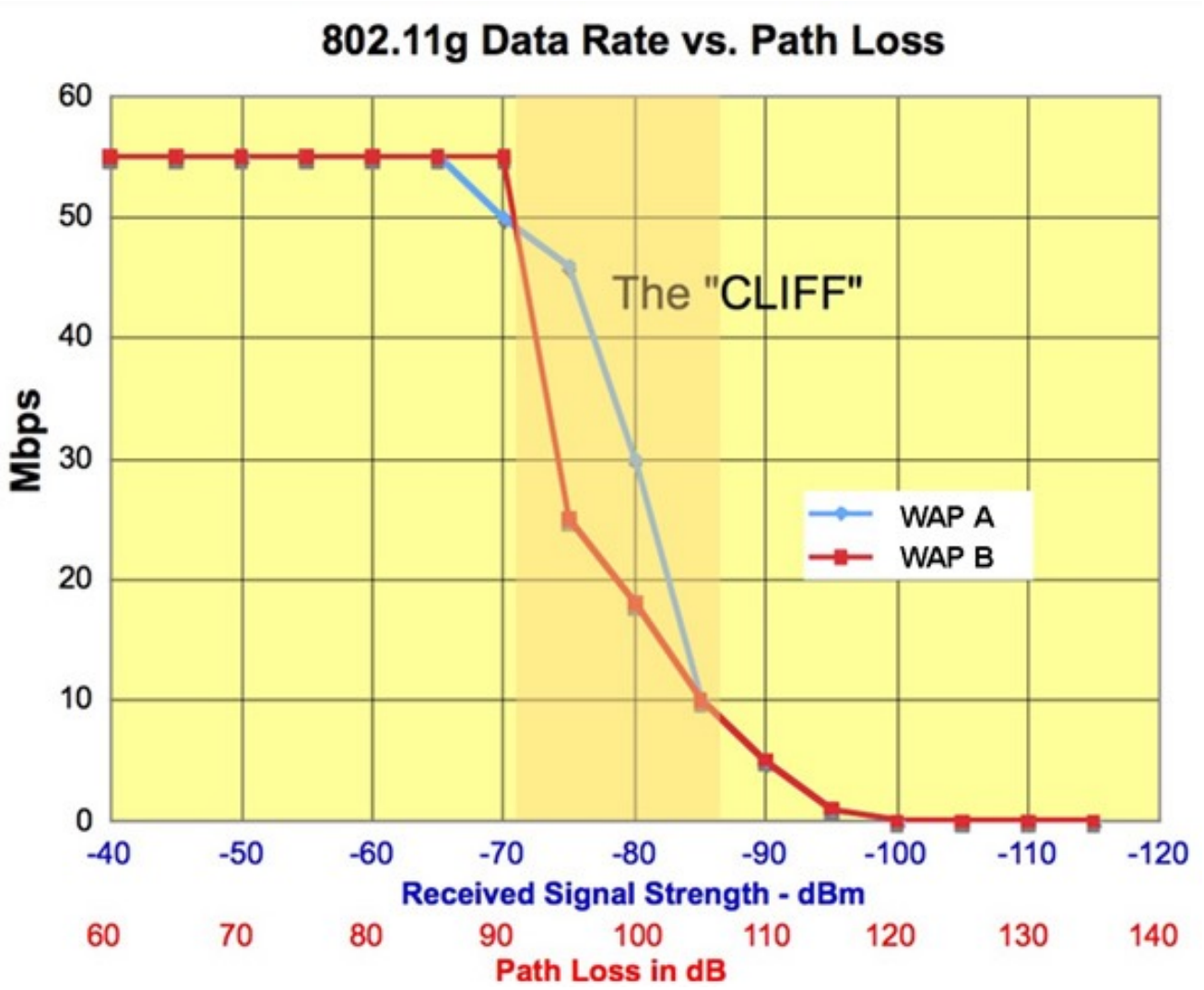


Image & Data Source <http://fosiao.com/system/files/misc/zigbee.wifi_.channel.jpg>

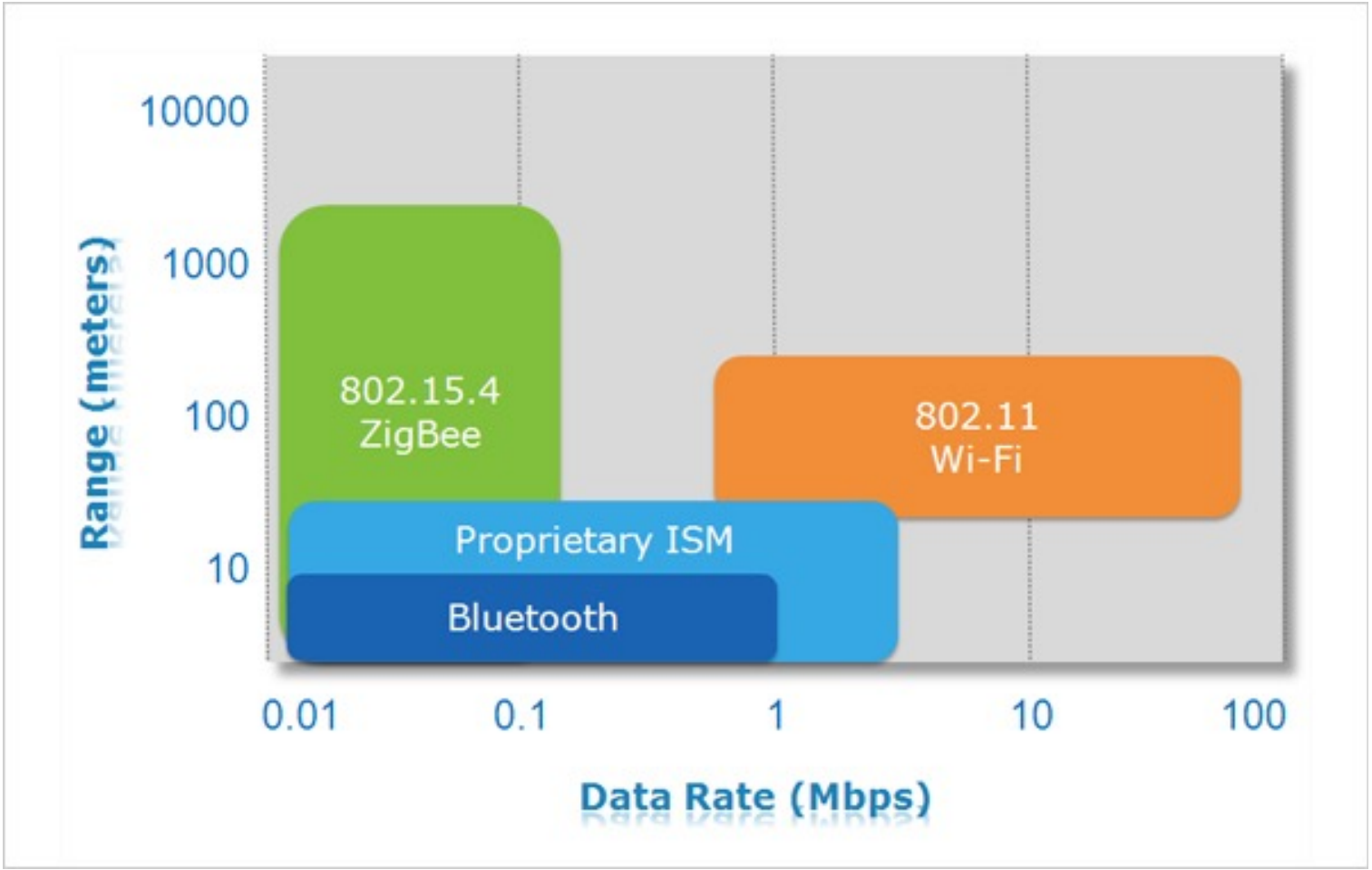
WiFi

- Dual Bands: 2.4GHz and 5GHz
- 802.11a/b/g/n
  - Cost vs Speed vs Interference (2.4/5.8 GHz) tradeoff
- Roaming
- Global standard
- High speed
  - Up to 300 Mbps
- High power consumption
  - Concern for mobile devices
- Range
  - Up to 100m

WiFi adapts speed to signal (802.11g)



Protocol Comparisons



Protocol Comparisons

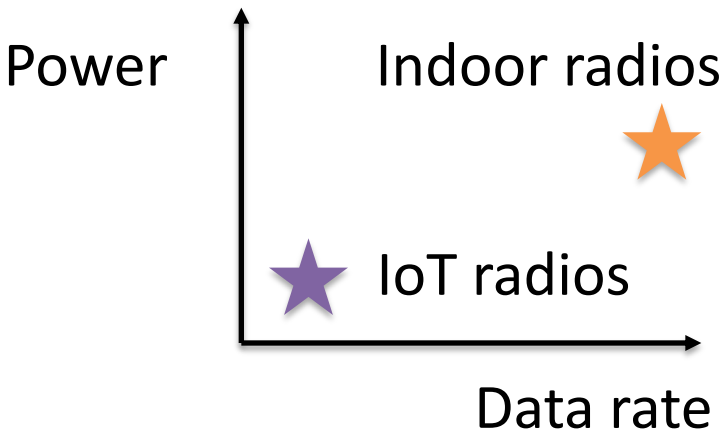
	Bluetooth	**Zigbee/802.15.4**	**WiFi**
Speed	Moderate	Low	High
Range	Moderate - High	High	High
Power Consumption	Low - Moderate	Low	High

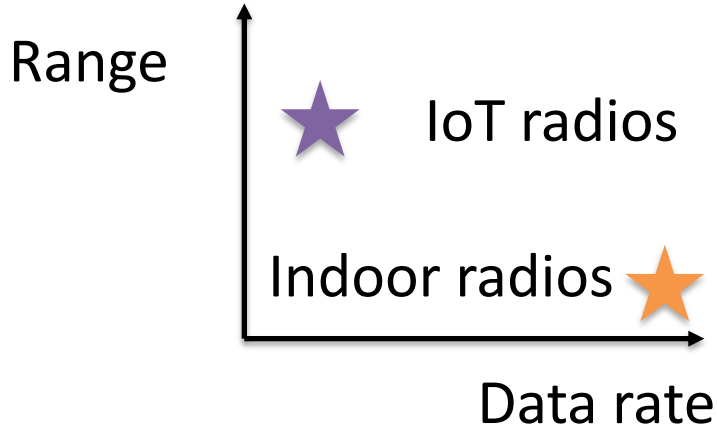
Design requirement of outdoor radios for IoT applications

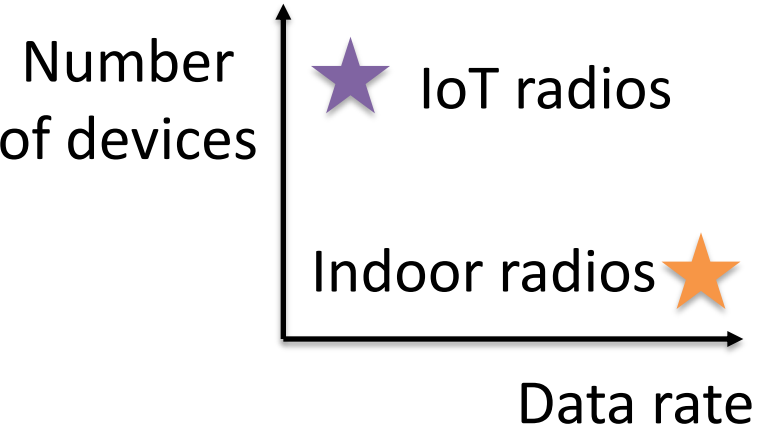
- Can we use WiFi/Bluetooth/ZigBee/Ant radios to support IoT applications deployed outdoor?
  - Can we achieve kilometer communication distance?
  - Can we support 3~5 years lifetime with a coin battery?
  - Can we support the communication with thousands of IoT devices with the coverage of a base station?
  - We only need to transmit 100 bits per second data compared to the mega bits per second case in WiFi

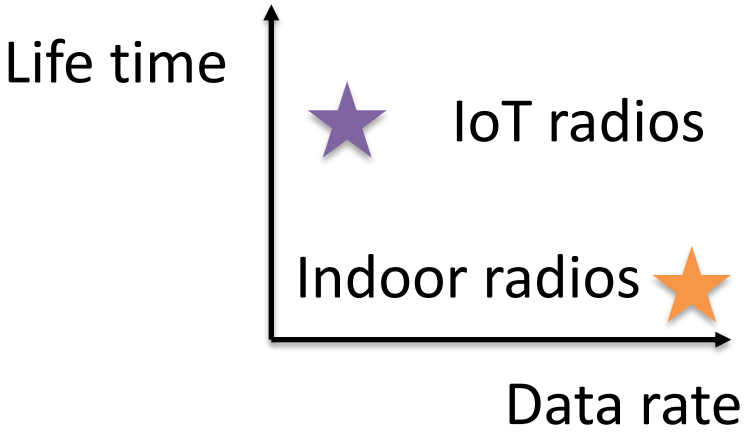
We are wiling to trade data rate for range, lifetime, and the number of devices supported.

Design requirement of outdoor radios for IoT applications









SIGFOX

- Deploy its own base stations to support IoT applications
  - Kilometer communication distance
  - Connect thousands of devices
  - 100 bits per second date rate
  - 5 years battery life time

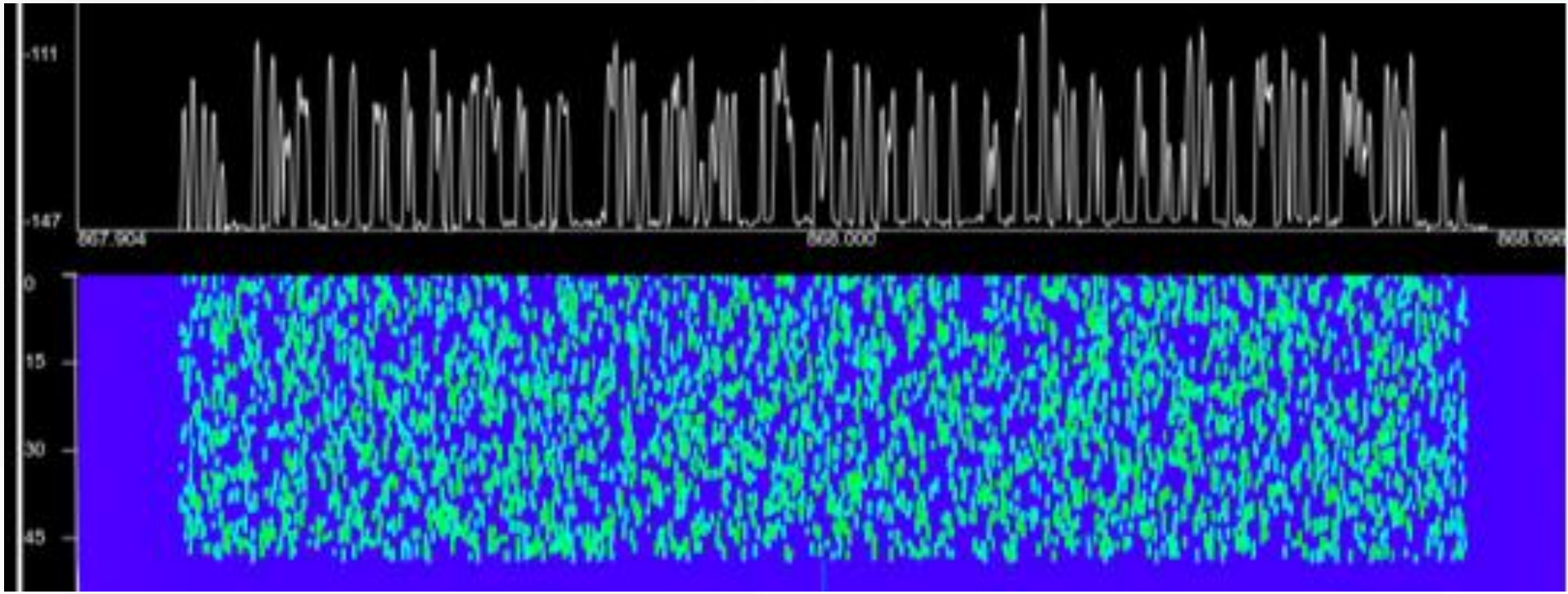
SIGFOX is Extremely Reliable

- REPETITION OF THE MESSAGES
  - Each message sent 3 times
  - Repetition at 3 different time slot = time diversity
  - Repetition at 3 different frequencies = frequency diversity
- COLLABORATIVE NETWORK
  - Network deployed and operated to have 3 base stations coverage at all times = space diversity
- MINIMIZATION OF COLLISIONS
  - Probability of collisions are highly reduced
  - Ultra Narrow Band
  - 3 base stations at 3 different locations

Ultra Narrow Band

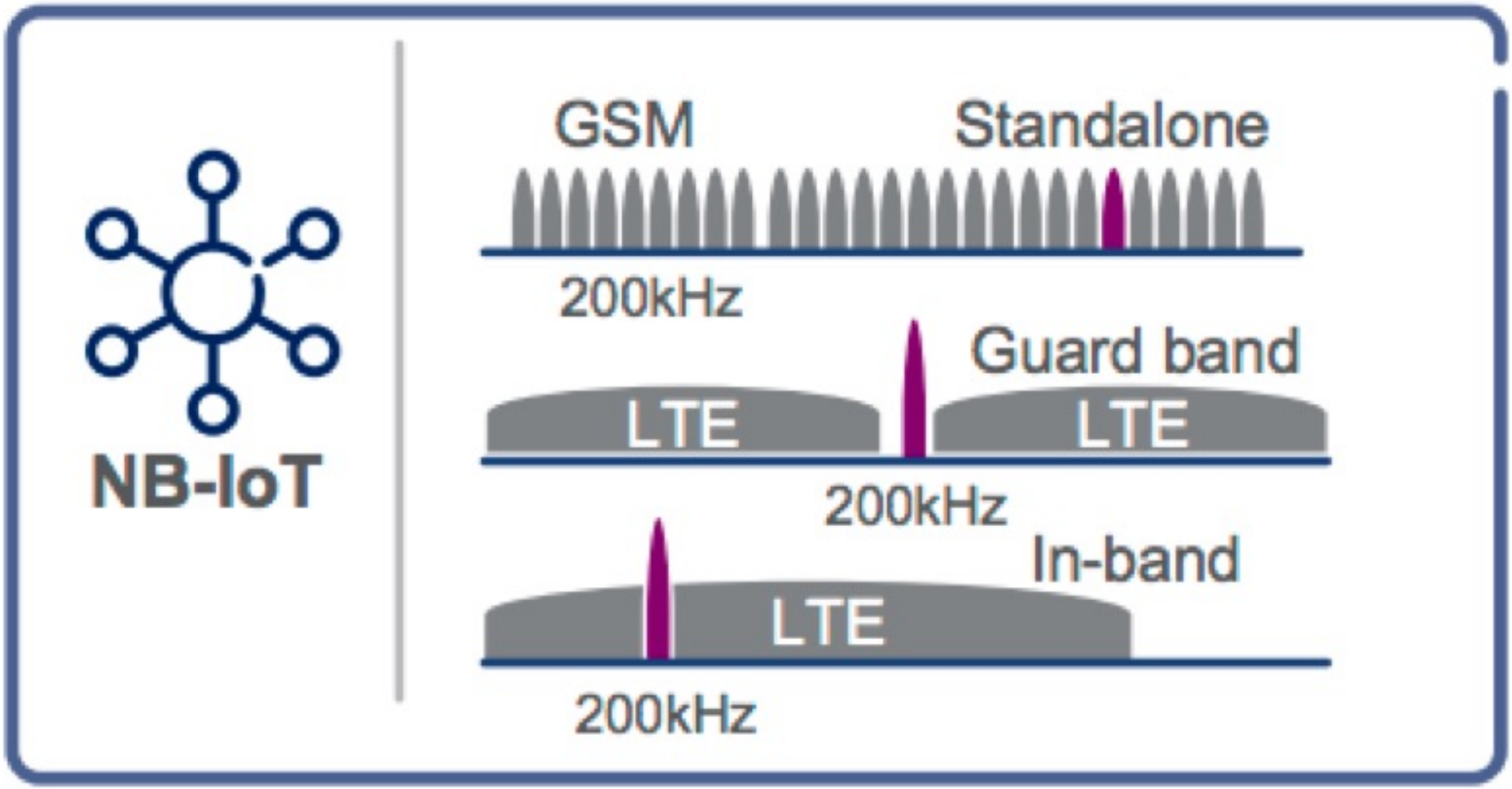
- Reduce the transmitted signal bandwidth
  - Reduced noise power
  - Therefore, we can reduce the transmission power
  - Therefore, we can reduce the power consumption of radio communication

Ultra Narrow Band

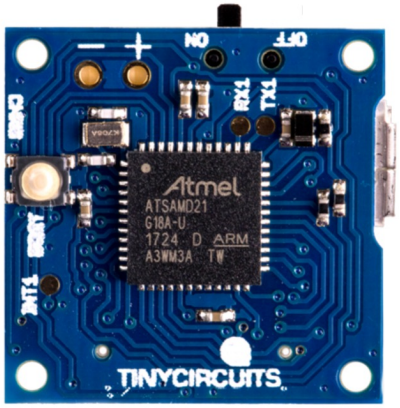


200 simultaneous messages within a 200kHz channel

NB-IoT LTE

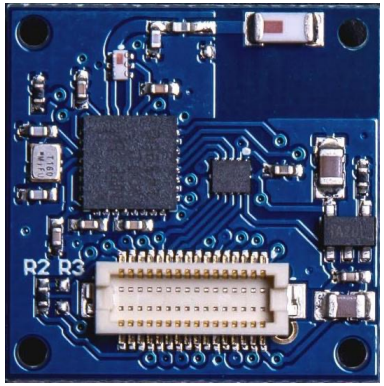

NB-IoT
GSM
Standalone
200kHz
Guard band
LTE
LTE
200kHz
In-band
LTE
200kHz

CSE190 Winter 2023



Lecture 19

Wireless



Wireless Embedded Systems

Aaron Schulman

Wireless communication is ubiquitous

- Bluetooth/WiFi/Cellular
- Device-to-Device (Bluetooth-Low-Energy)
  Device-to-infrastructure (LTE or WiFi)
- Bluetooth Low Energy
  the most popular wireless protocol.

Outline

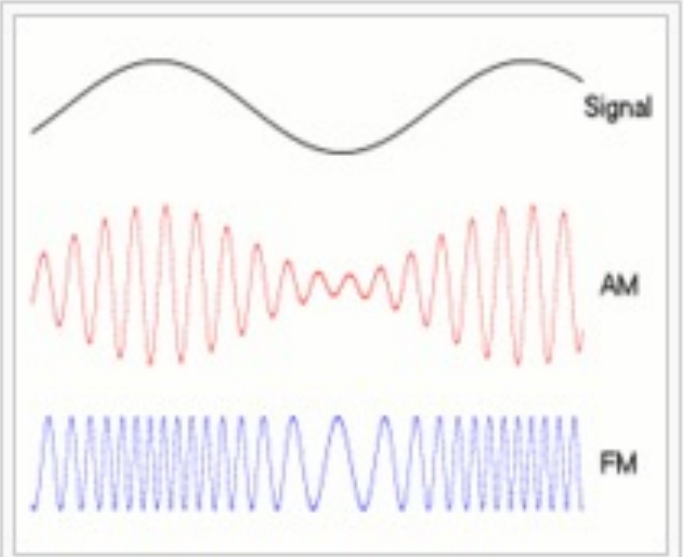
- What are radios
  - How do they work?
- Fundamental characteristics
  - Design tradeoffs
- Common radio standards/protocols for indoor applications
  - Where characteristics fall under (above)
- Emerging radio standards/protocols for outdoor Internet-of-Things applications
  - Why the design requirement of IoT radios is different

What are Radios?

- A device that enables wireless transmission of signals
  - Electromagnetic wave
  - Transmitter encodes signal and receiver decodes it

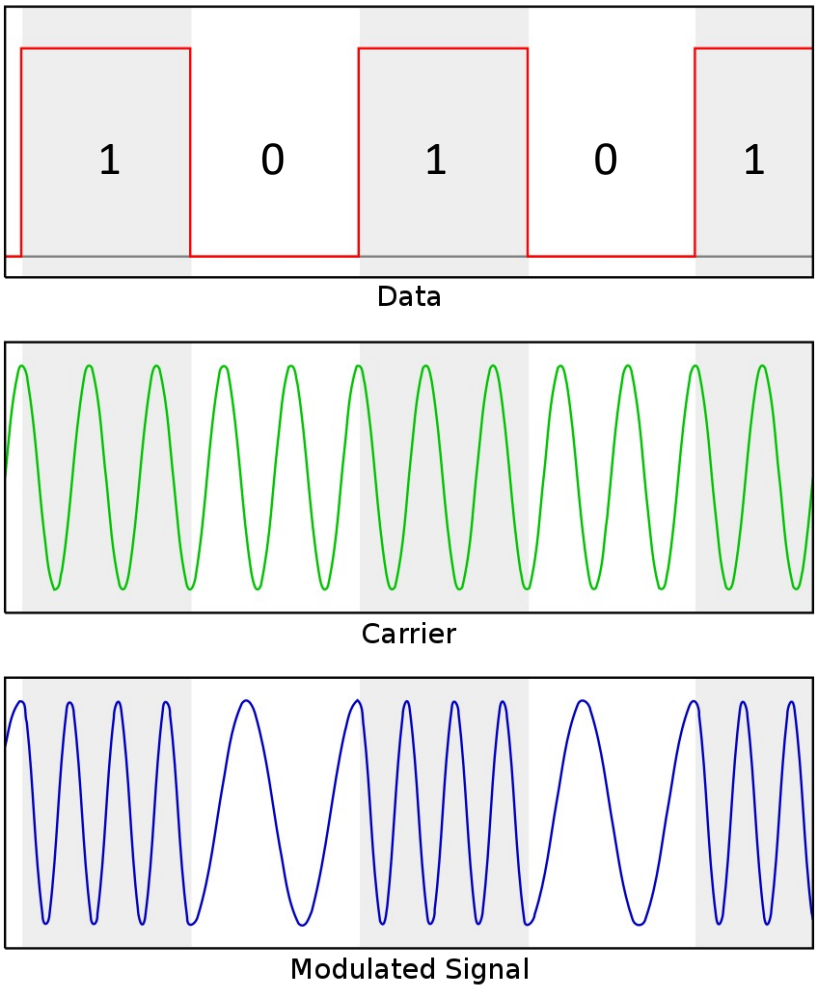
How Radios Work - Transmitting

- Modulation
  - Converts digital bits to an analog signal
  - Encodes bits as changes in a **carrier frequency**:
  - Frequency, Amplitude, Phase, etc.



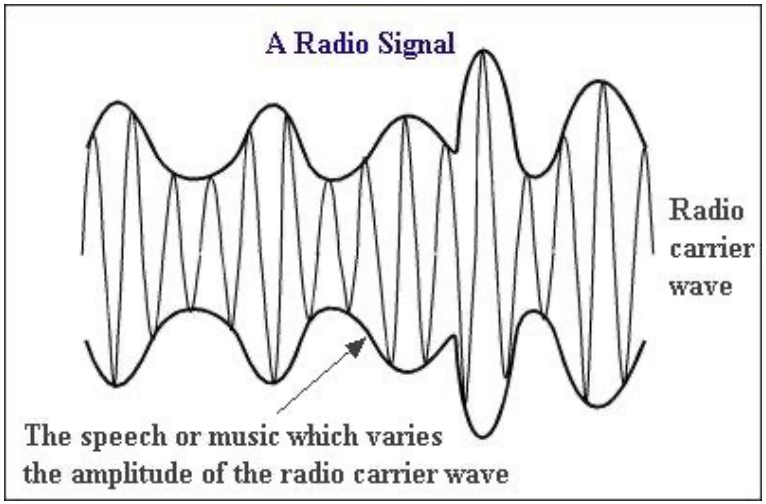
A low-frequency message signal (top) may be carried by an AM or FM radio wave.

Example of modulating digital data onto an analog signal



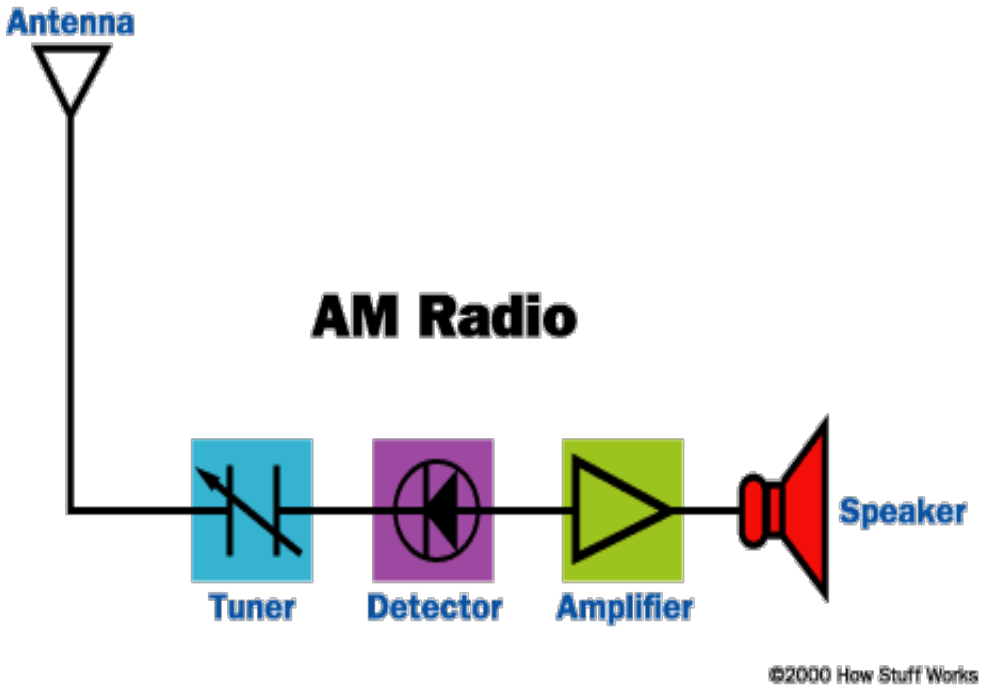
How Radios Work - Receiving

- Transmitted signals must be demodulated
- Simplest is Envelope Detection
  - Detect changes in carrier freq. amplitude
- Most complex receivers require synchronization
- All require filtering



How Radios Work – Receiving (AM Radio Example)

- Antenna picks up modulated radio waves
- Tuner filters out specific frequency ranges
- Amplitude variations detected with demodulation
- Amplifier strengthens the clipped signal and sends it through the speaker



Wireless Protocol Characteristics

- Why so many protocols for indoor and outdoor applications?
- All radios have to make tradeoffs
  - Short vs. long distance
  - High vs. low power/energy
  - High vs. low speeds
  - Large vs. small number of devices
  - Device-to-device, device-to-infrastructure
  - Indoor vs. outdoor usages

Common Radio Protocols

Radios for indoor IoT applications

- Design requirements
  - Short range
  - High data rate
  - Small number of devices
- Common protocols
  - Bluetooth/Low Energy
  - ZigBee
  - Ant
  - WiFi

Radios for outdoor IoT applications

- Design requirements
  - Long range
  - Low data rate
  - Large number of devices
  - Low energy consumption
- Common protocols
  - GSM/GPRS
  - LTE
- Emerging protocols
  - Sigfox/LoRA
  - Narrow band LTE
  - Backscatter

Bluetooth

- Radio band: 2.4-2.48 GHz
- Average 1 Mbps - Up to 3 Mbps
- Supports point-to-point and point-to-multipoint
  - Creates personal area networks (PANs/Piconets)
  - Connects up to 8 devices simultaneously
- Minimal interference between devices
  - **Devices alter frequencies arbitrarily after packet exchanges - up to 1600 times/second - frequency hopping**
- 3 classes of Bluetooth transmit power
- Frequency hopping communication

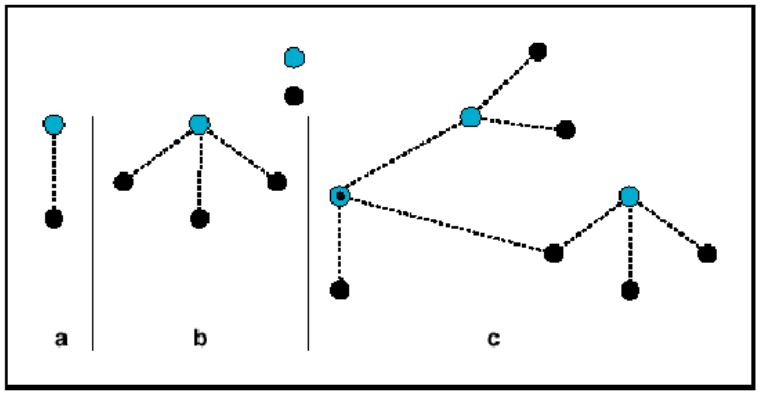


Figure 1.2: Piconets with a single slave operation (a), a multi-slave operation (b) and a scatternet operation (c).

Class	Maximum Power	Operating Range
Class 1	100mW (20dBm)	100 meters
Class 2	2.5mW (4dBm)	10 meters
Class 3	1mW (0dBm)	1 meter

Frequency hopping communication was invented by actress Hedy Lamar

UNITED STATES PATENT OFFICE

2,292,387

SECRET COMMUNICATION SYSTEM

Hedy Kiesler Markey, Los Angeles, and George Antheil, Manhattan Beach, Calif.

Application June 10, 1941, Serial No. 397,412

6 Claims. (Cl. 250—2)

This invention relates broadly to secret communication systems involving the use of carrier waves of different frequencies, and is especially useful in the remote control of dirigible craft.

Fig. 2 is a schematic diagram of the apparatus at a receiving station;

Fig. 3 is a schematic diagram illustrating a starting circuit for starting the motors at the

Bluetooth Applications

- Wireless communication between devices
  - Mobile phones, laptops, cameras, gaming controllers, computer peripherals, etc
- Short range sensor transmission
- Share multimedia - pictures, video, music
- A2DP - Advanced Audio Distribution Profile
  - Stream audio wirelessly

Bluetooth Low Energy

From 2001 – 2006 Nokia asked:

How do we design a radio that can transmit short bursts of data for months or years *only being powered by a coin cell battery?*

The answer is: Keep the radio asleep mode most of the time!

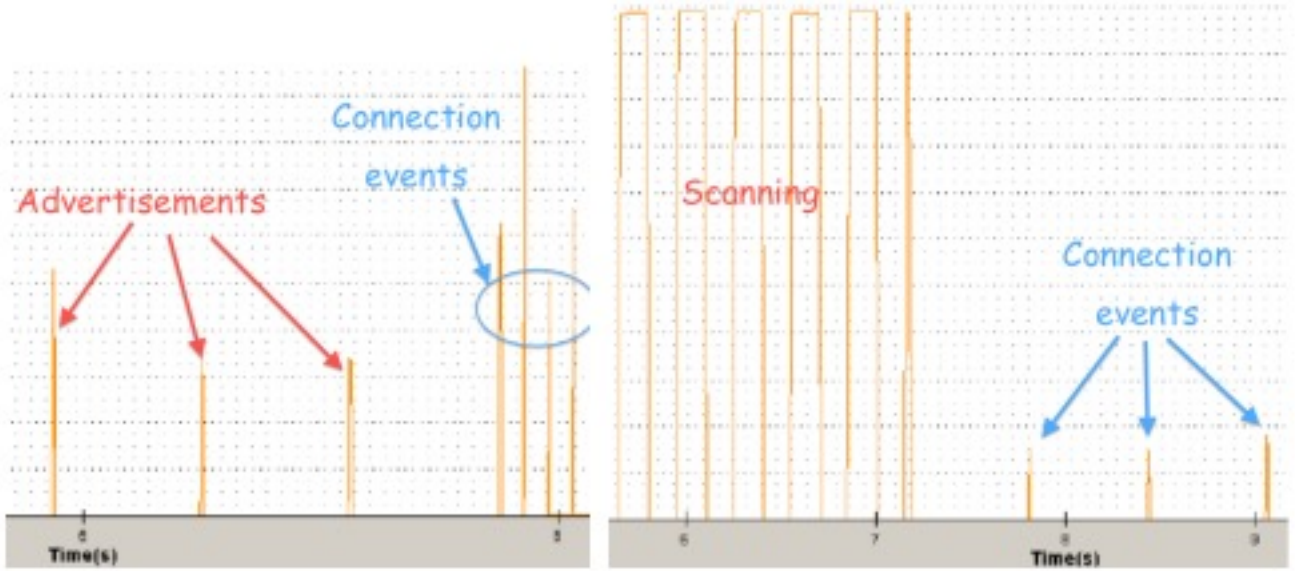
1. Advertise on only one of three channels (less freq. hopping)
2. Transmit quickly at 1 Mbit/s
3. Make the minimum time to send data only 3 msec
4. Make a very predictable time when the device accepts connections
5. Limit the max transmit power to 10 mW
6. However, don't sacrifice security: AES 128-bit

What tradeoffs were made?

The protocol is designed for transmitting tiny data

- 4 operations: Read, Write, Notify, Indicate
- Maximum of 20 bytes of data per packet

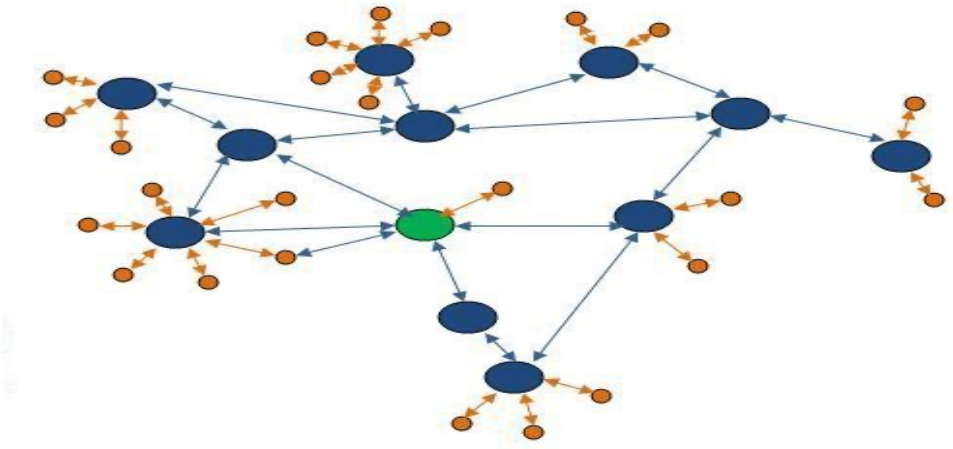
(Plots are power over time)



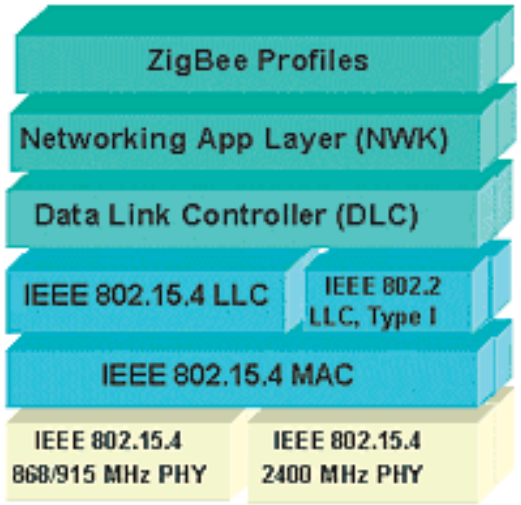
From: How Low Energy is Bluetooth Low Energy - Siekkinen et al.

Zigbee/802.15.4

- Zigbee is built on top of 802.15.4
- Radio bands: 868MHz in Europe, 915MHz in US and Australia. 2.4GHz else worldwide.
- Low data-rate - 250 kbps, low power – Up to 1000 days
- Transmits over longer distances through mesh networks



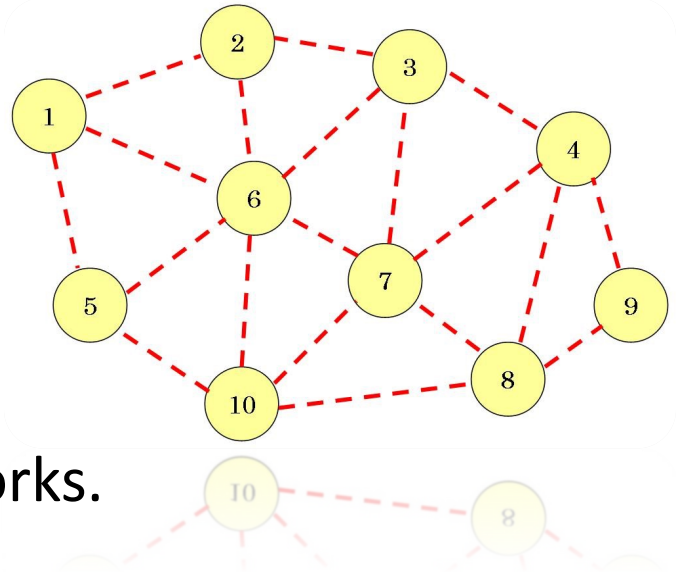
Zigbee/802.15.4 mesh network and device types



Zigbee is usually used in mesh networks

- A mesh network consists of a series of nodes.

- Each node must acquire and transmit its own data, as well as act as a relay for other nodes to propagate data.

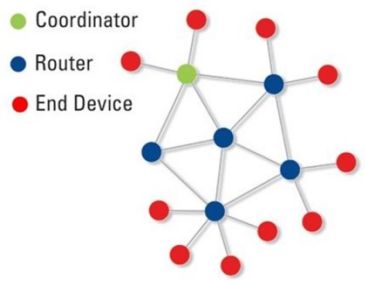


- ZigBee devices often form Mesh Networks.

- Examples: Wireless light switching, Music school practice rooms.

Image Source: <http://kf5czo.blogspot.com/2012/03/ham-radio-and-mesh-networks.html>

Mesh Networking



- Advantages of Mesh Networking:
  - Allows devices to communicate to multiple other dev the network.
  - Multiple paths to destination – greater flexibility against interference.
  - Allows overall network to grow to larger physical sizes than possible with point-to-point networks.

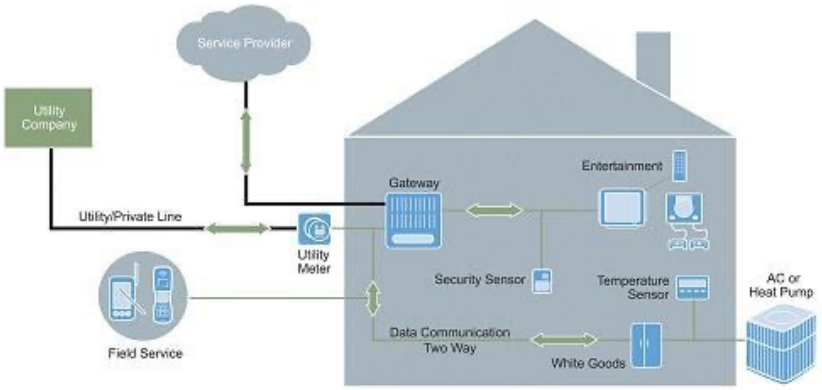
- Mesh Characteristics:
  - *Self-forming* – ZigBee devices can establish communication pathways when new devices appear.
  - *Self-healing* – If a node is removed from the network (either intentionally or not) the remaining network will look to establish alternate routes of communication.

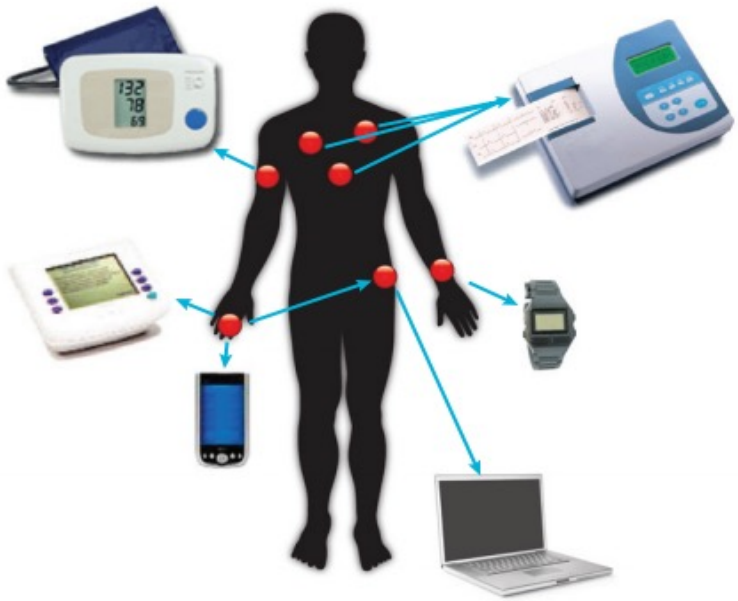
Why ZigBee?

- Low Power, Cost, and Size
- Straightforward configuration
- Good support and documentation
  - Lots of products already on the market
- Mesh Networking
- Lends itself well to a variety of applications
- Very low wakeup time
  - 30mS (Zigbee) vs. up to 3S (Bluetooth)

Zigbee/802.15.4 Applications

- Wireless environmental sensors
  - Temperature, pressure, sound, luminous intensity
- Medical devices
  - Glucose meters, heart monitors
- Household automation
  - Security/temperature controllers
  - Smoke/motion detectors





Bluetooth, Zigbee, and WiFi contend

- Competes with Wi-Fi for bandwidth..
  - Only four usable bands in Wi-Fi intensive scenarios

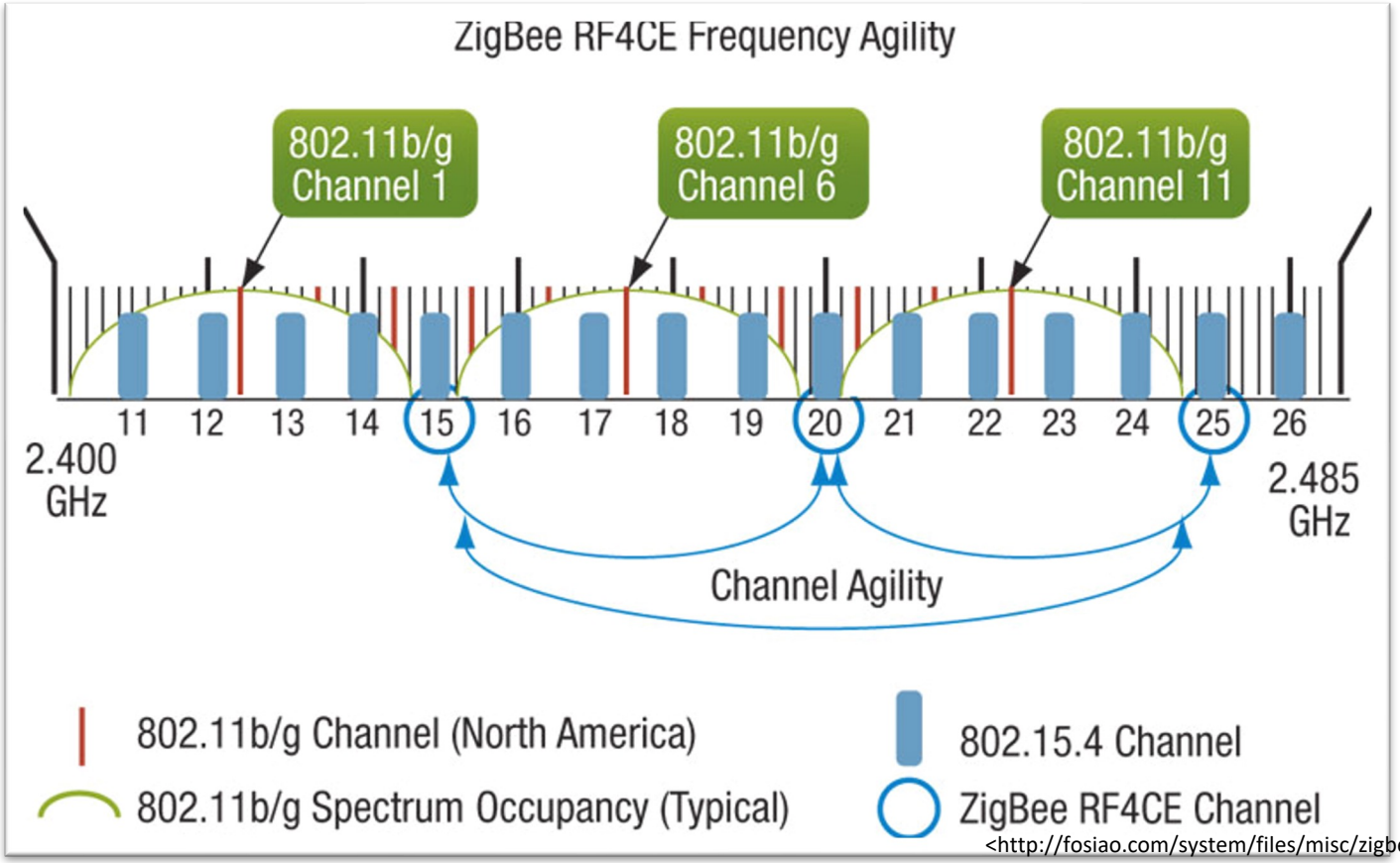
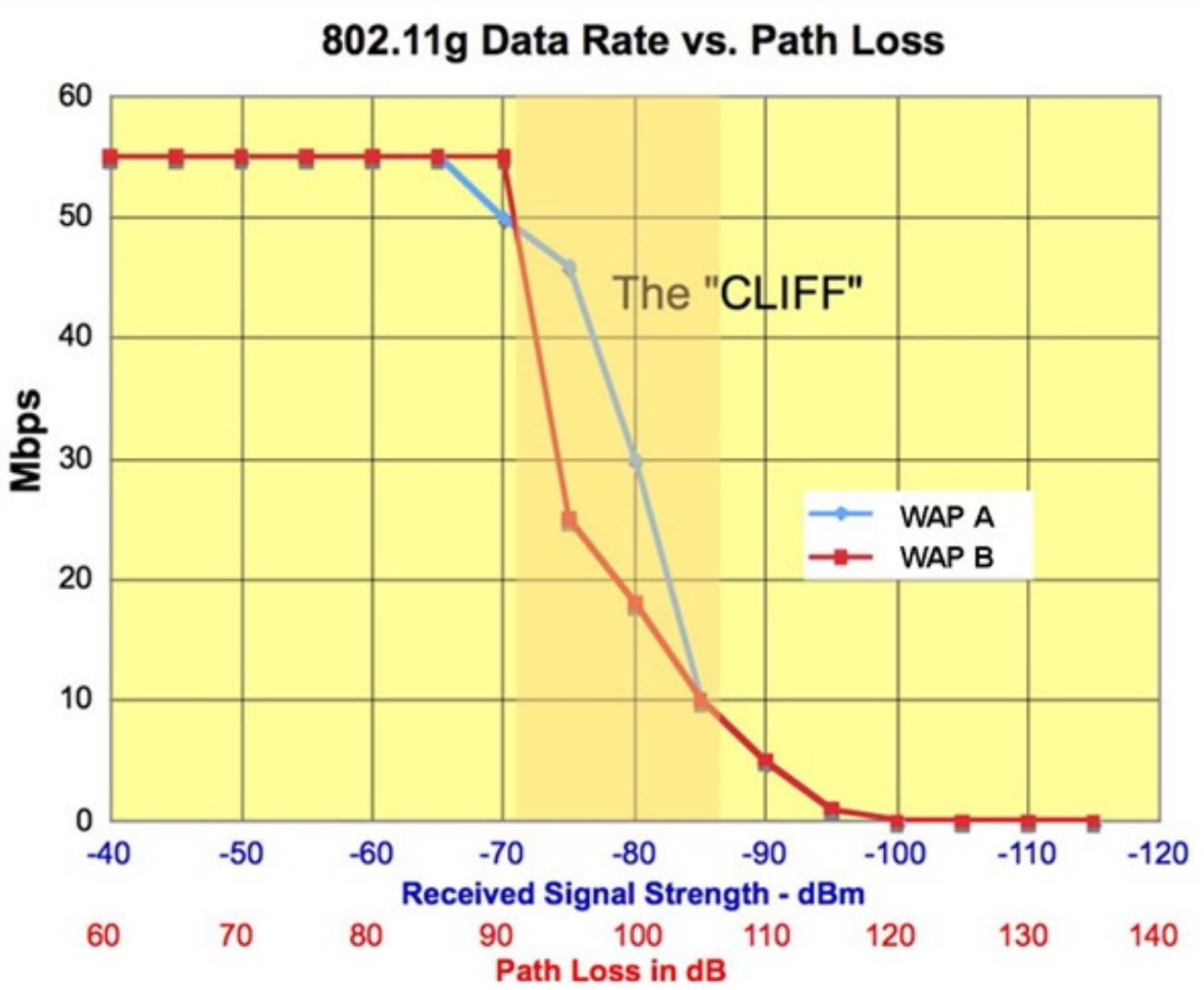


Image & Data Source <http://fosiao.com/system/files/misc/zigbee.wifi_.channel.jpg>

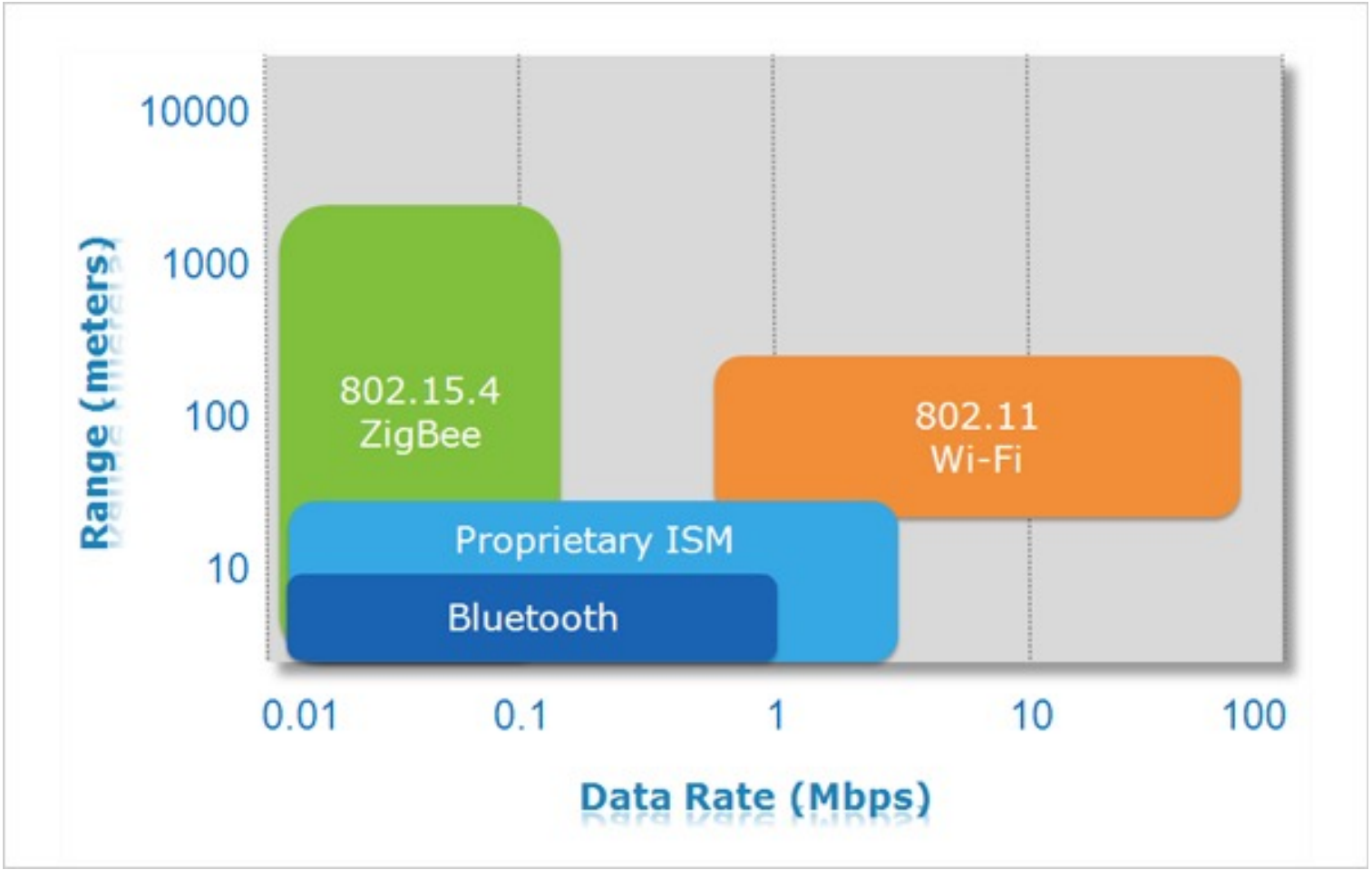
WiFi

- Dual Bands: 2.4GHz and 5GHz
- 802.11a/b/g/n
  - Cost vs Speed vs Interference (2.4/5.8 GHz) tradeoff
- Roaming
- Global standard
- High speed
  - Up to 300 Mbps
- High power consumption
  - Concern for mobile devices
- Range
  - Up to 100m

WiFi adapts speed to signal (802.11g)



Protocol Comparisons



Protocol Comparisons

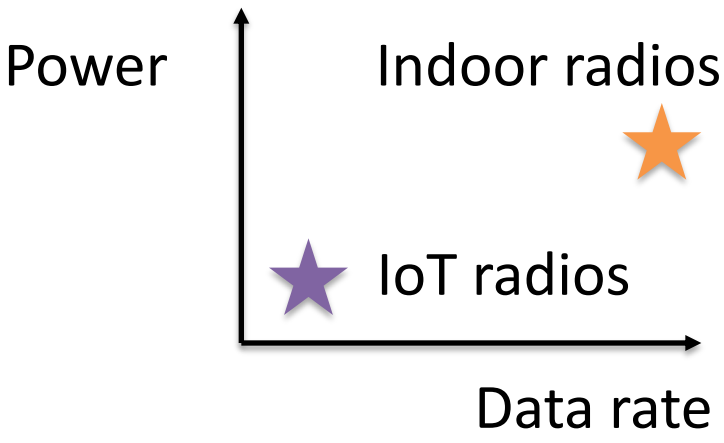
	Bluetooth	**Zigbee/802.15.4**	**WiFi**
Speed	Moderate	Low	High
Range	Moderate - High	High	High
Power Consumption	Low - Moderate	Low	High

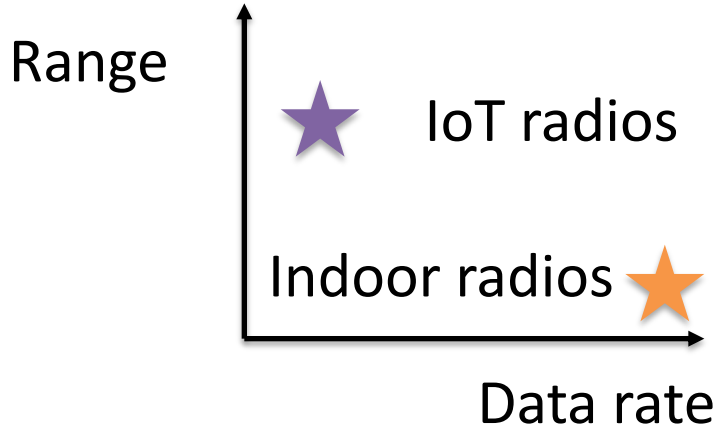
Design requirement of outdoor radios for IoT applications

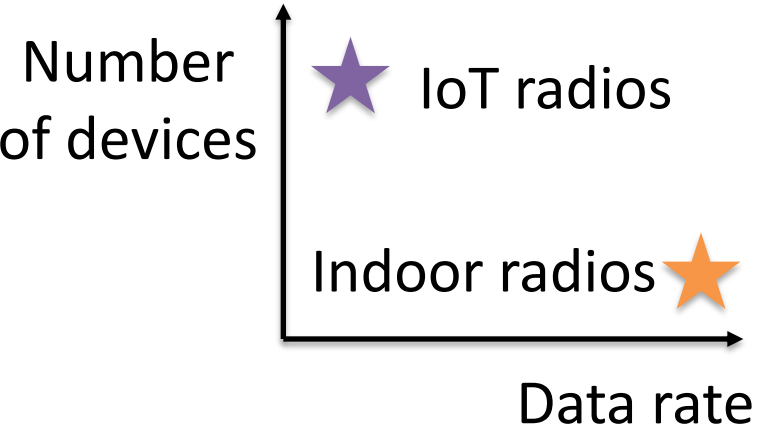
- Can we use WiFi/Bluetooth/ZigBee/Ant radios to support IoT applications deployed outdoor?
  - Can we achieve kilometer communication distance?
  - Can we support 3~5 years lifetime with a coin battery?
  - Can we support the communication with thousands of IoT devices with the coverage of a base station?
  - We only need to transmit 100 bits per second data compared to the mega bits per second case in WiFi

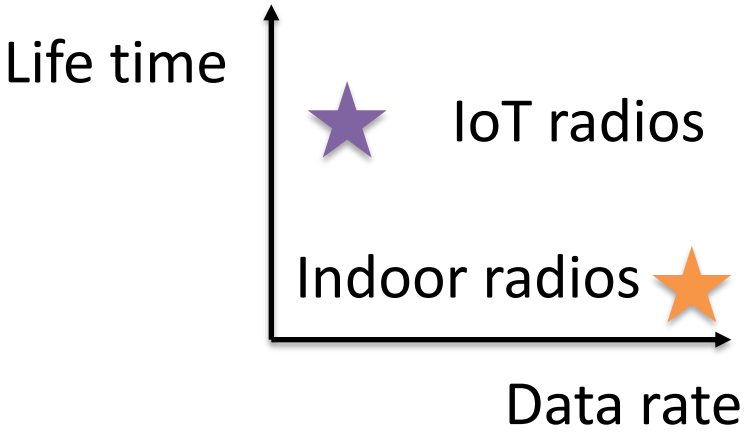
We are wiling to trade data rate for range, lifetime, and the number of devices supported.

Design requirement of outdoor radios for IoT applications









SIGFOX

- Deploy its own base stations to support IoT applications
  - Kilometer communication distance
  - Connect thousands of devices
  - 100 bits per second date rate
  - 5 years battery life time

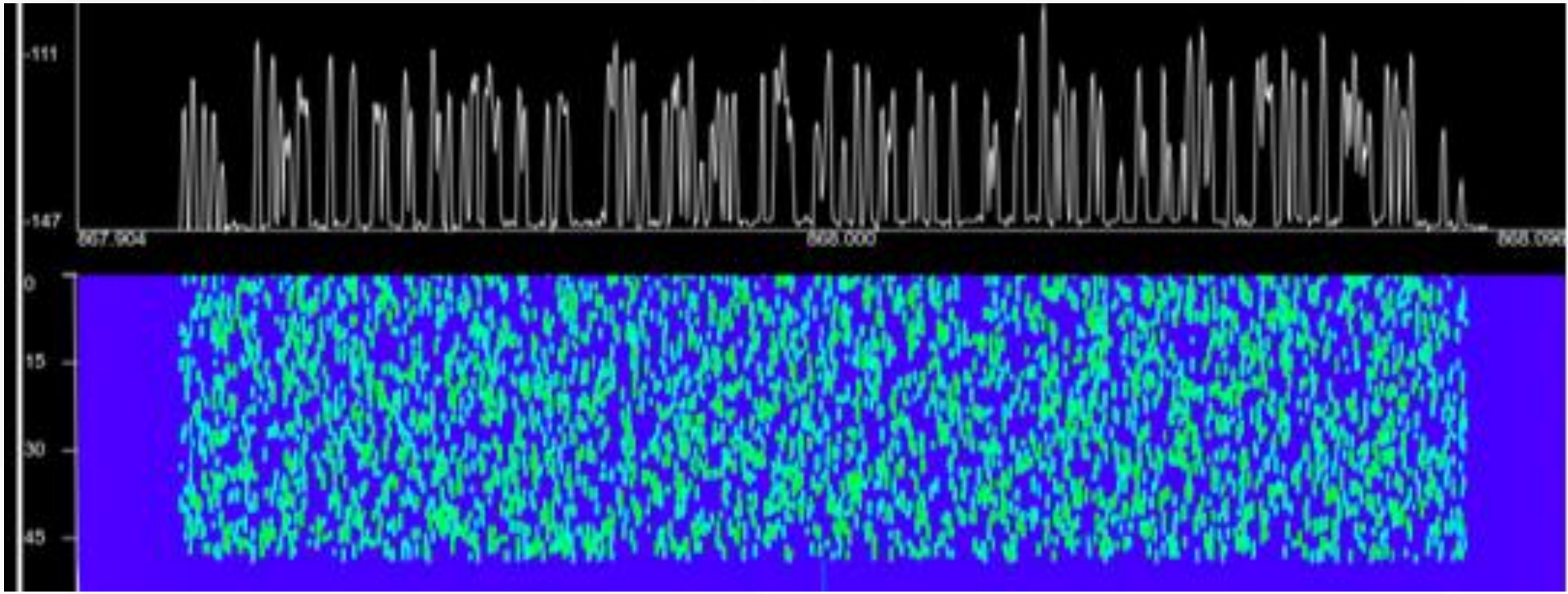
SIGFOX is Extremely Reliable

- REPETITION OF THE MESSAGES
  - Each message sent 3 times
  - Repetition at 3 different time slot = time diversity
  - Repetition at 3 different frequencies = frequency diversity
- COLLABORATIVE NETWORK
  - Network deployed and operated to have 3 base stations coverage at all times = space diversity
- MINIMIZATION OF COLLISIONS
  - Probability of collisions are highly reduced
  - Ultra Narrow Band
  - 3 base stations at 3 different locations

Ultra Narrow Band

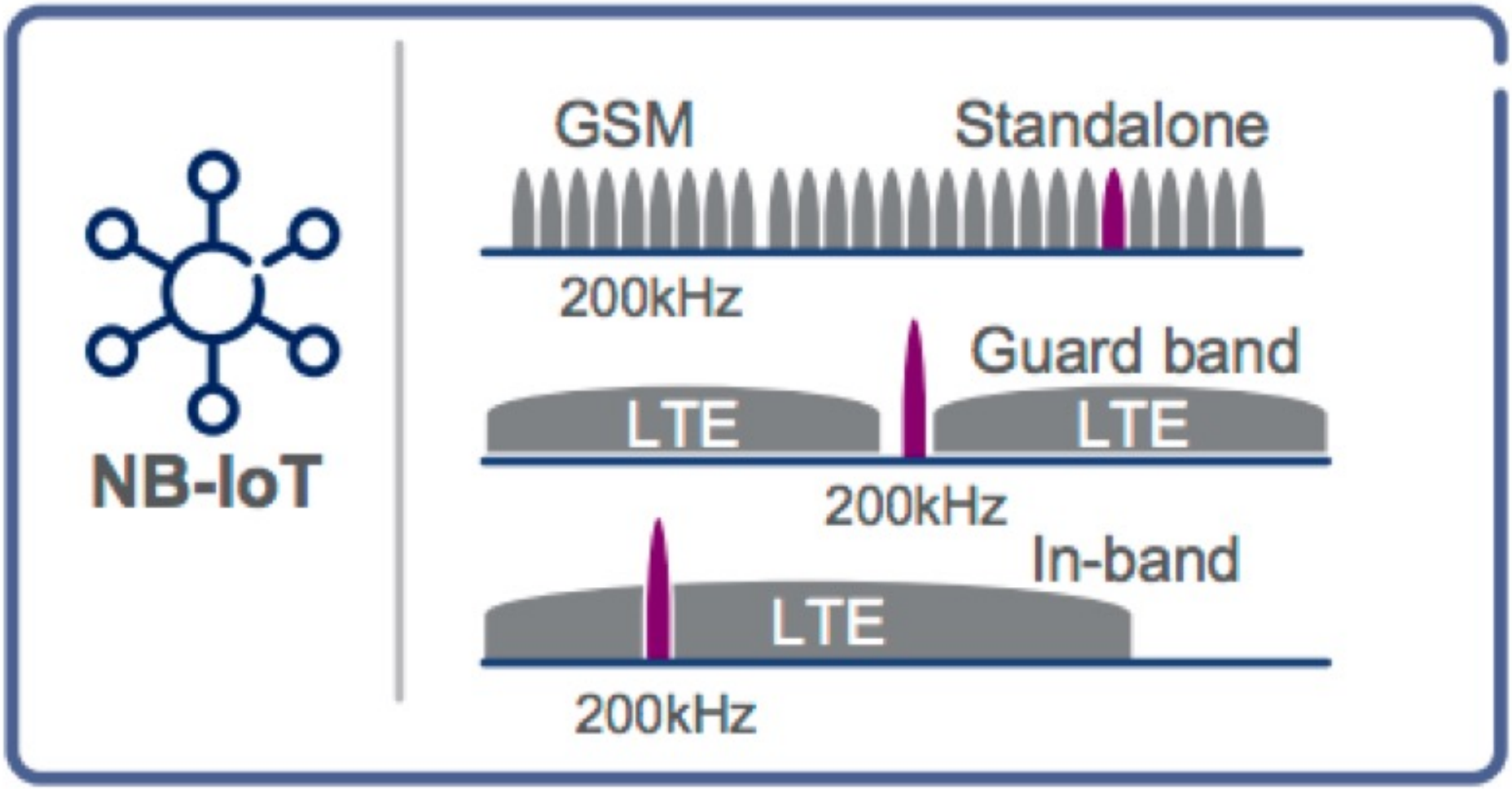
- Reduce the transmitted signal bandwidth
  - Reduced noise power
  - Therefore, we can reduce the transmission power
  - Therefore, we can reduce the power consumption of radio communication

Ultra Narrow Band



200 simultaneous messages within a 200kHz channel

NB-IoT LTE


NB-IoT
GSM
Standalone
200kHz
Guard band
LTE
LTE
200kHz
In-band
LTE
200kHz