EPFL

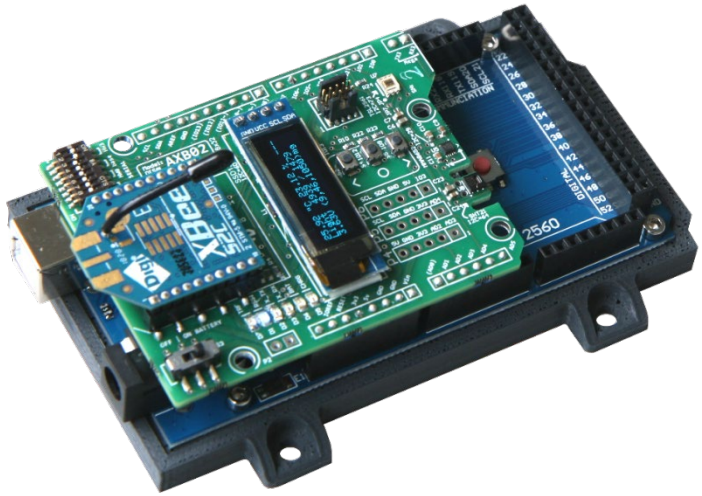
Disal

Signals, Instruments, and Systems – W7

Introduction to Embedded Systems – Communication and Computation

Outline

- Communication
  - Wired
  - Wireless
  - Communication power
- Computation
  - Dealing with limited resources: the example of memory management
  - Real-time computing in embedded systems



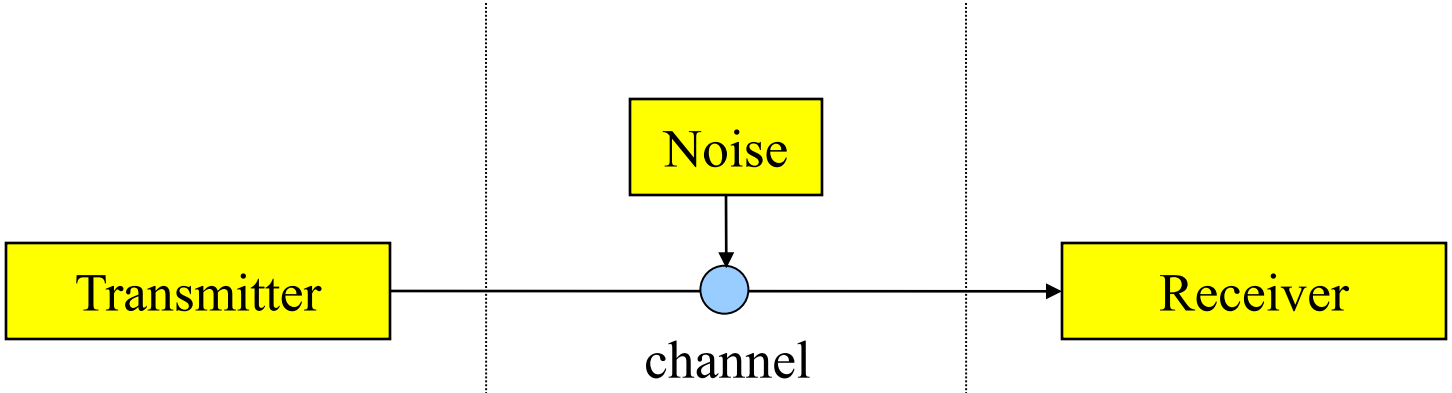
Wired Communication

Where?

- Within embedded systems (from sensor to microcontroller, from microcontroller to microcontroller, etc.)
- From an embedded system to another
- From an embedded system to a PC
- …

Communication Model

Communication Model



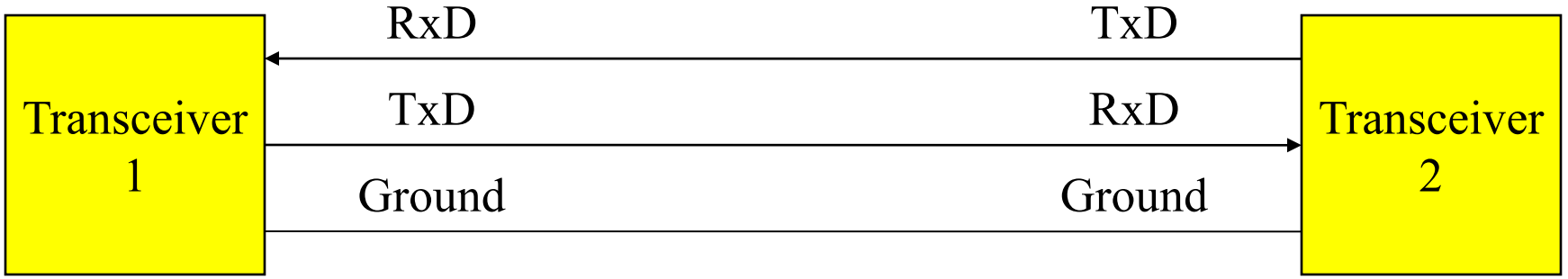
Modulation
Coding
(Compression)

Distortion
Filtering
Frequency shift
...

Demodulation
Decoding
(Decompression)

A Seminal Example: The RS-232 (serial port)

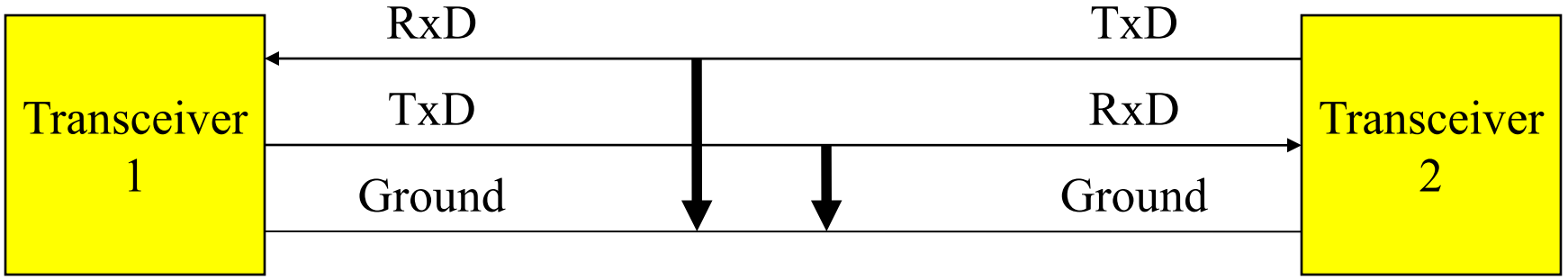
- Hardware:
  - 3 wires: TxD, RxD, Ground



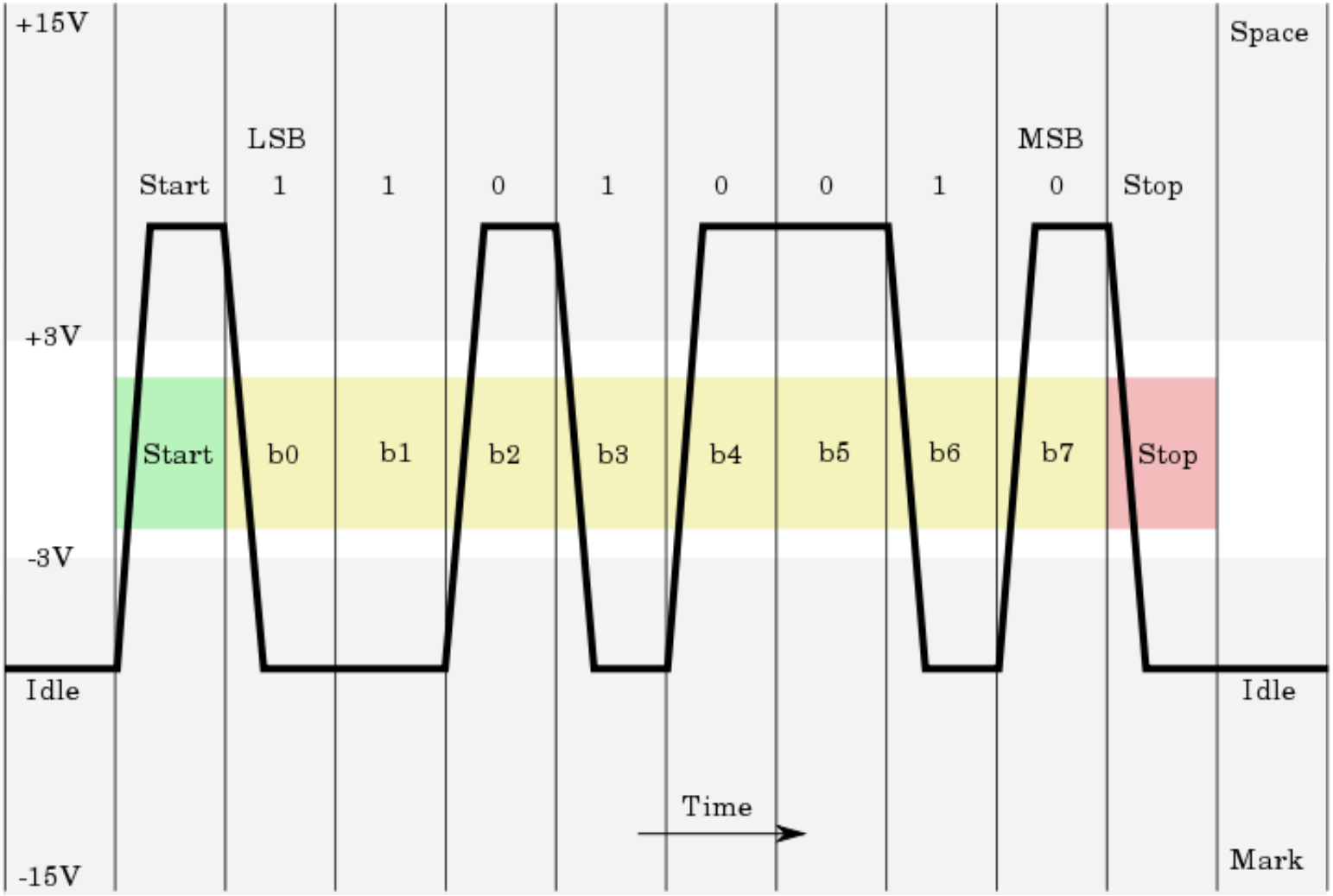
Transceiver = Transmitter + Receiver

RS-232 (serial port)

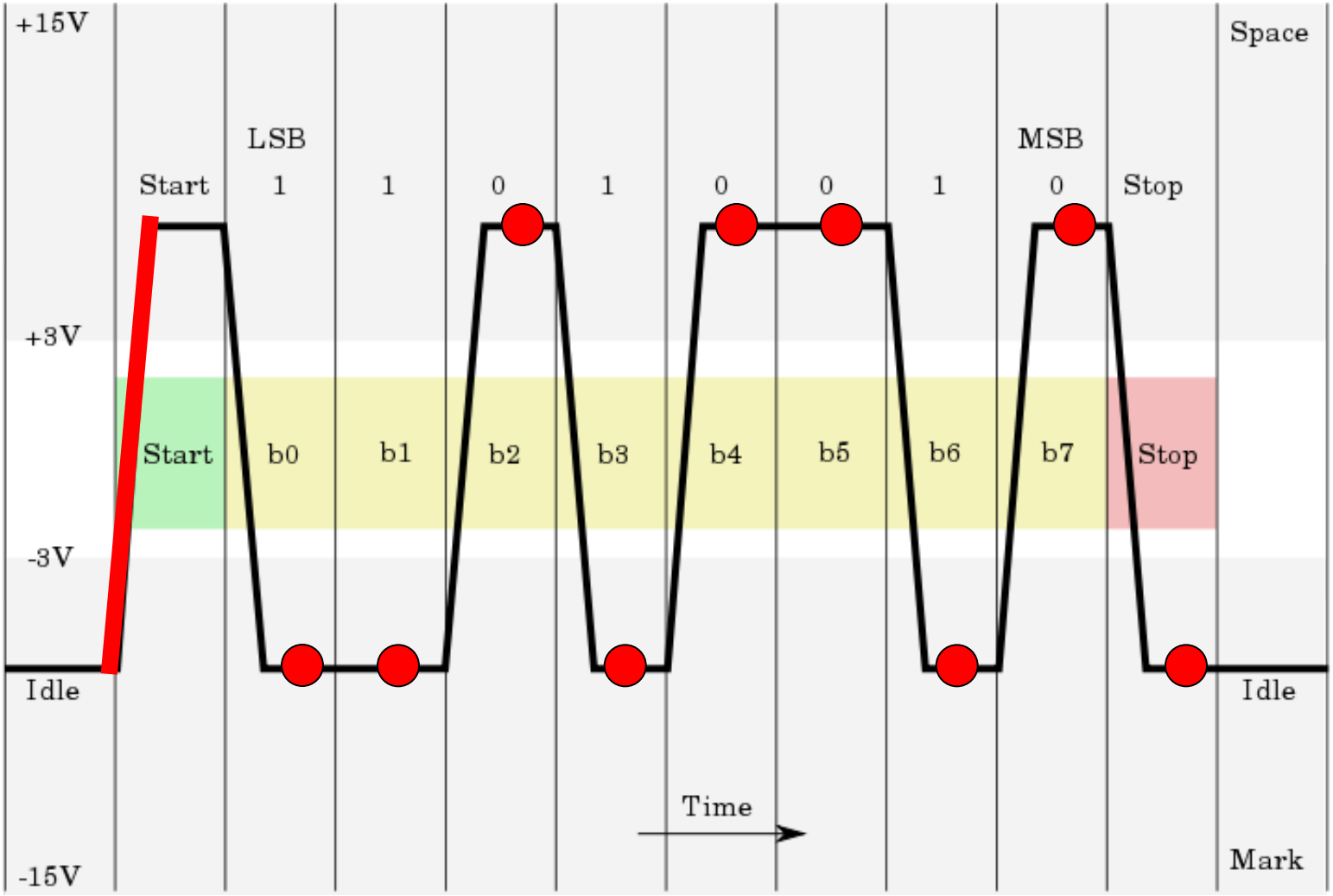
- Signal: **between** RxD/TxD and Ground



RS-232 Modulation



RS-232 Demodulation



RS-232 Delay

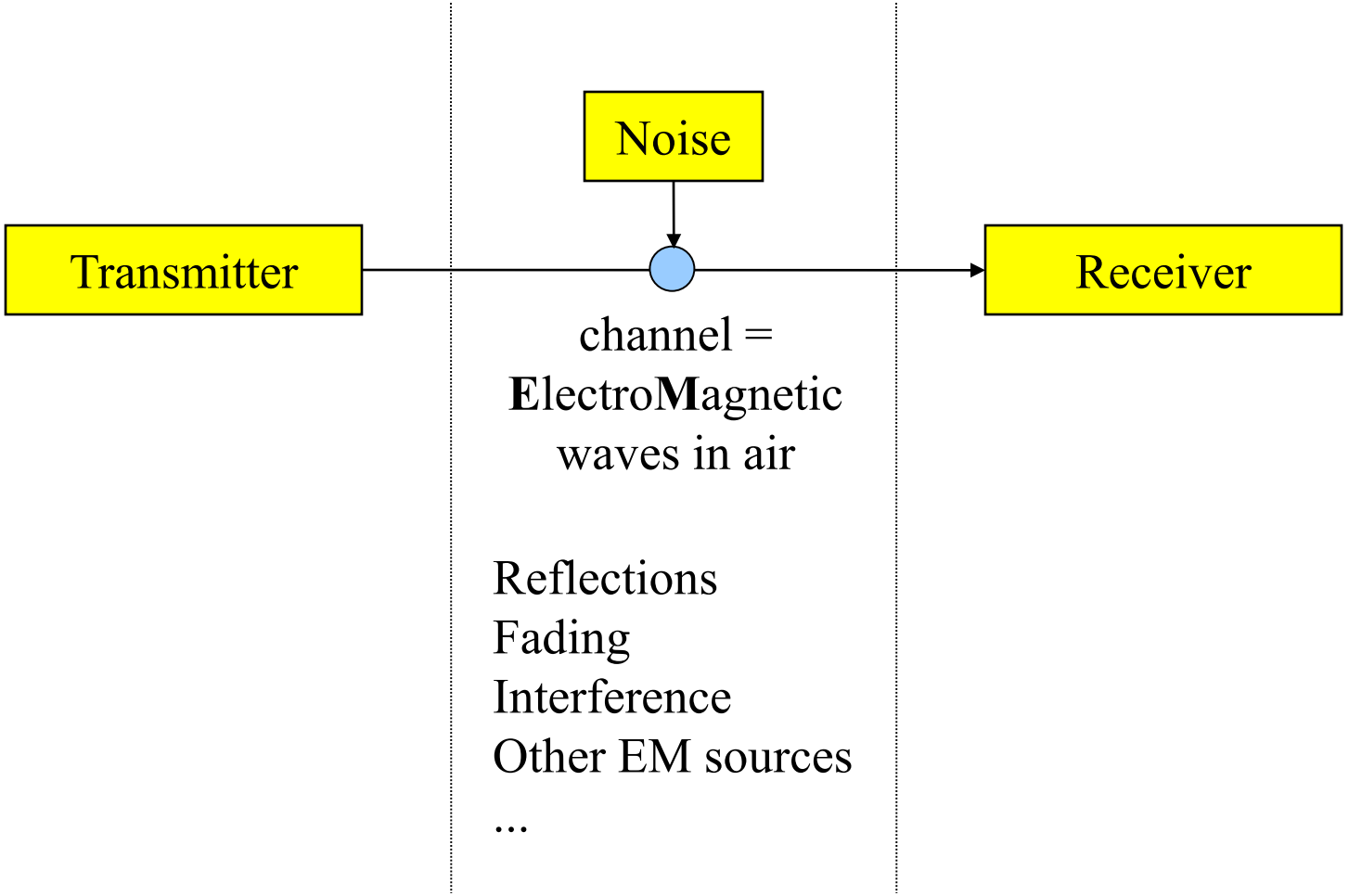
- Packet-based
  - 1 byte (i.e. 8 bits)/ packet
  - 8 data bits + 2 control bits (start/stop) = 10 bits
- Transmission speed
  - max. 115'200 bits/s (bps)
- Propagation speed:
  - approx. c (speed of light)

RS-232 Delay

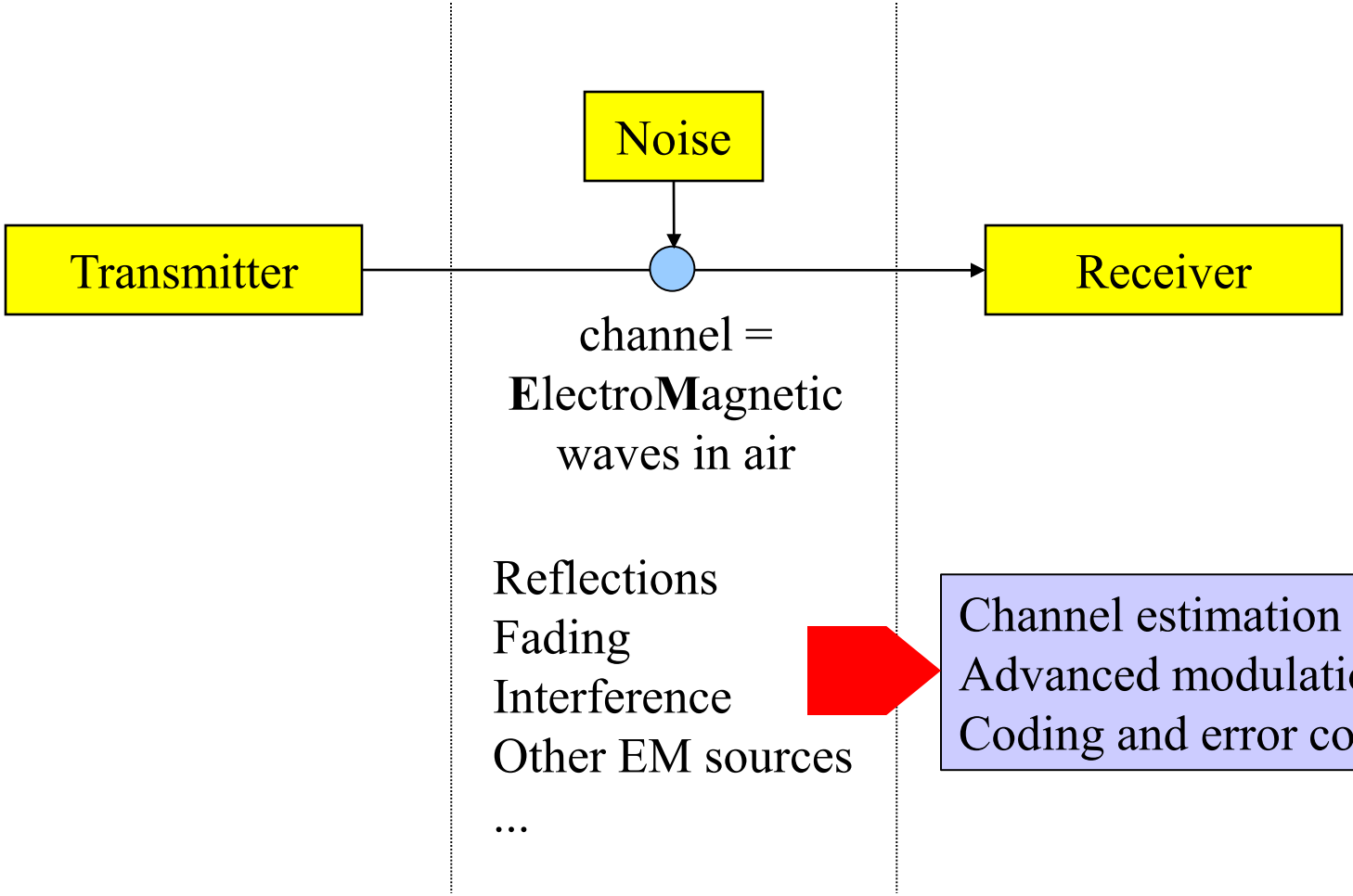
- Transmission delay
  - 10 bits / 115'200 bps = 86.8 µs
- Signal propagation delay (2 m cable)
  - 2 m / c = 6.6712819 ns
- Processing delay:
  - ~ 1 µs (modulation, demodulation, processing)
- Total: ~ 90 µs = 0.09 ms

Wireless Communication
–
Sharing the Medium

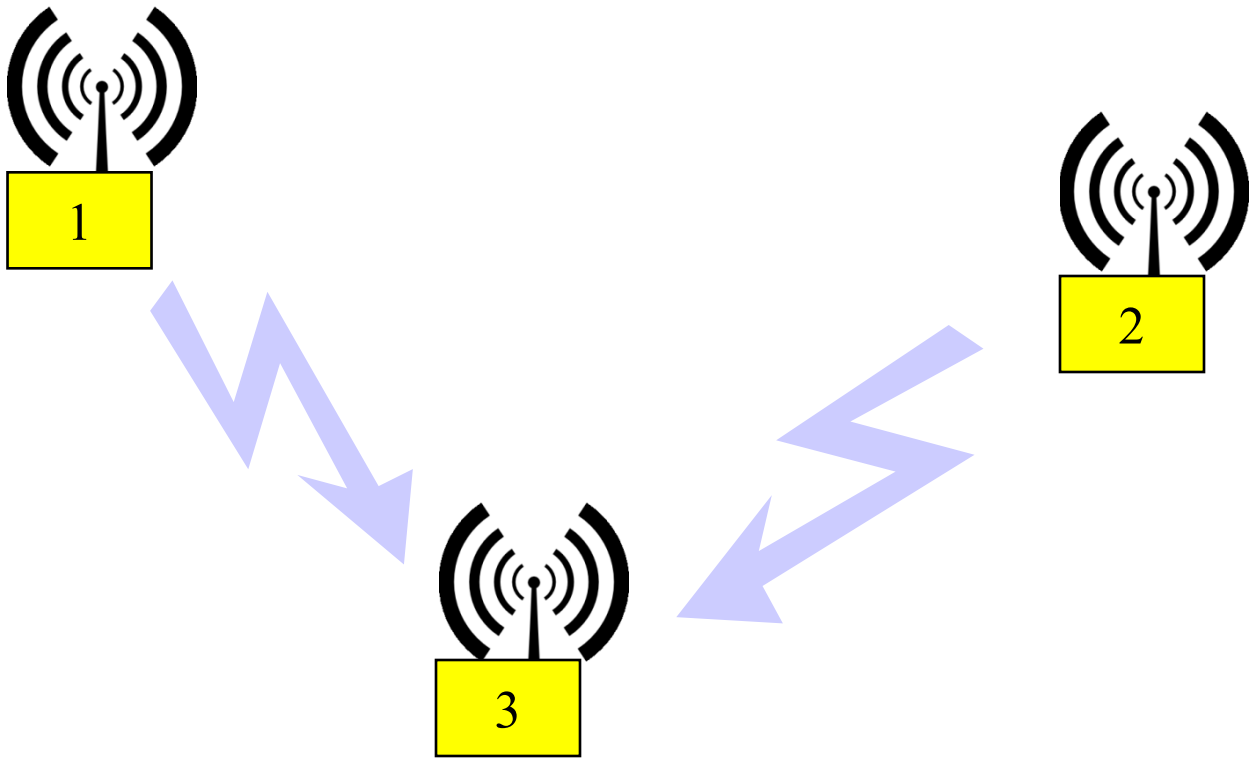
Communication Model



Communication Model



Sharing the Medium

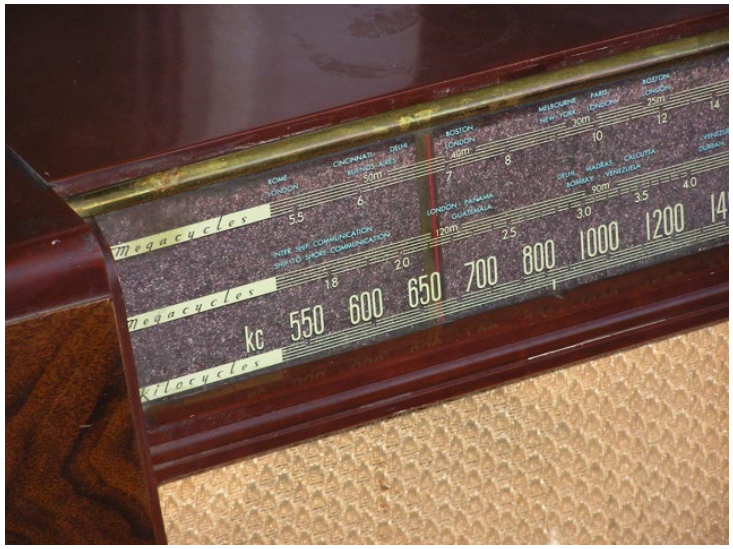


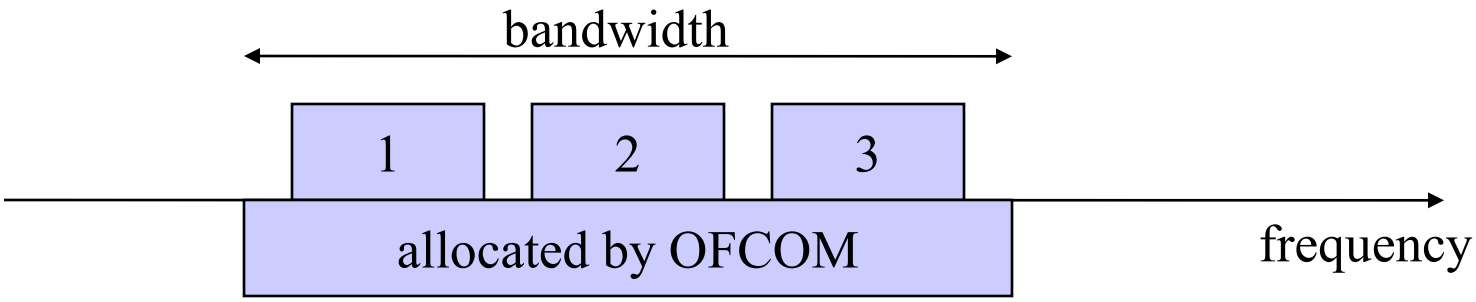
Sharing the Medium

- TDMA
  - Time-Division Multiple Access
  - “You shut up while I talk“
  - Time allocation
    - Fixed, synchronized
      e.g., mobile phones (GSM)
    - Dynamic (check if channel is free)
      e.g., Wireless LAN (802.11b/g/n)

Sharing the Medium

- FDMA
  - Frequency-Division MA
  - e.g., FM radio channels
  - Frequency regulation
    - OFCOM (CH)





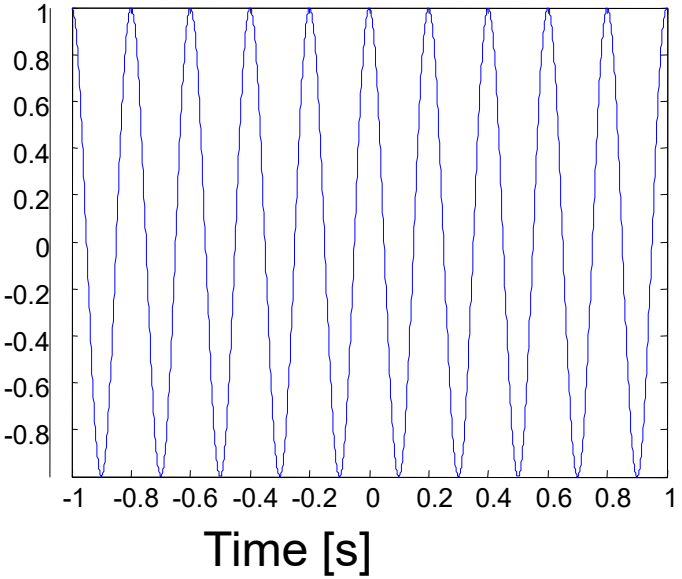
Properties

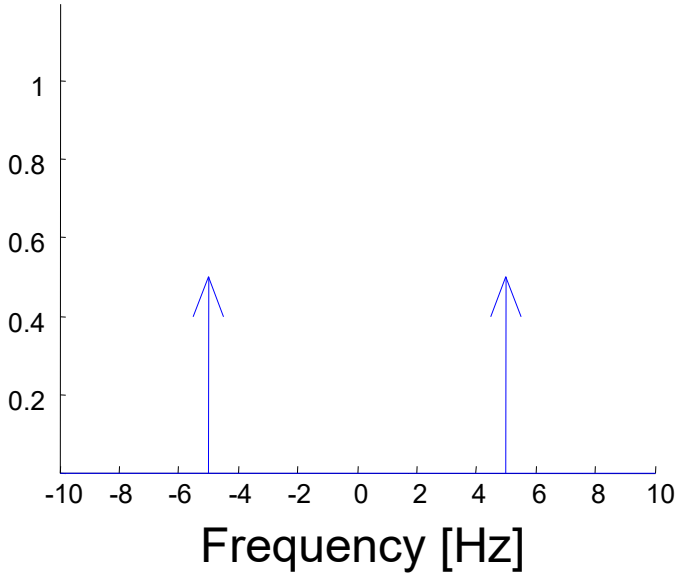
Week 2, s. 28

Linearity	$\mathrm{h}(t) = af(t) + bg(t) \Rightarrow \hat{h}(\xi) = a\hat{f}(\xi) + b\hat{g}(\xi)$
Translation	$h(\mathrm{t}) = f(t - t_0) \Rightarrow \hat{h}(\xi) = e^{-2\pi i t_0 \xi}\hat{f}(\xi)$
Modulation	$h(t) = e^{2\pi i t \xi_0} f(t) \Rightarrow \hat{h}(\xi) = \hat{f}(\xi - \xi_0)$
Scaling	$h(t) = f(at) \Rightarrow \hat{h}(\xi) = \frac{1}{\lvert a \rvert}\hat{f}(\frac{\xi}{a})$
Convolution	$h(t) = (f * g)(t) \Rightarrow \hat{h}(\xi) = \hat{f}(\xi) \cdot \hat{g}(\xi)$

FT of Real Sinusoidal Function

Week 2, s. 26



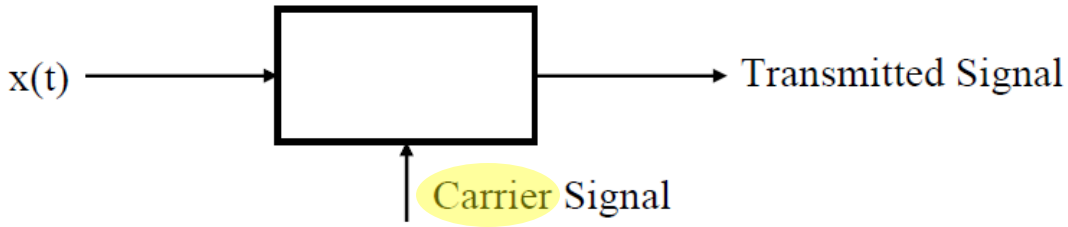


$$f(t) = \cos(2\pi a t)$$

$$\hat{f}(\xi) = \frac{\delta(\xi - a) + \delta(\xi + a)}{2}$$

Modulation

The Concept of Modulation

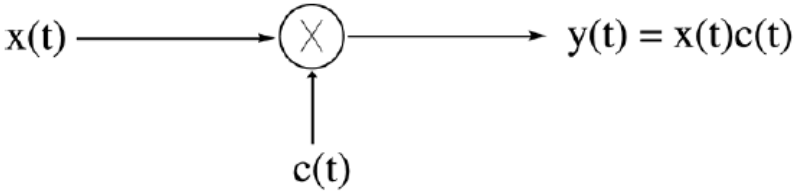


Why?

- More efficient to transmit E&M signals at higher frequencies
- Transmitting multiple signals through the same medium using different carriers
- Transmitting through "channels" with limited passbands
- Others...

How?

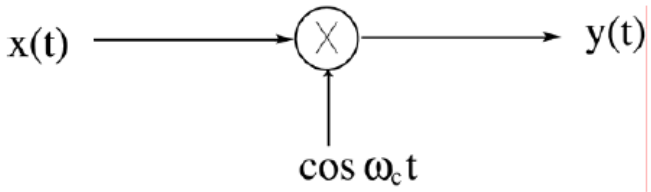
- *Many* methods
- Focus here for the most part on *Amplitude Modulation* (AM)



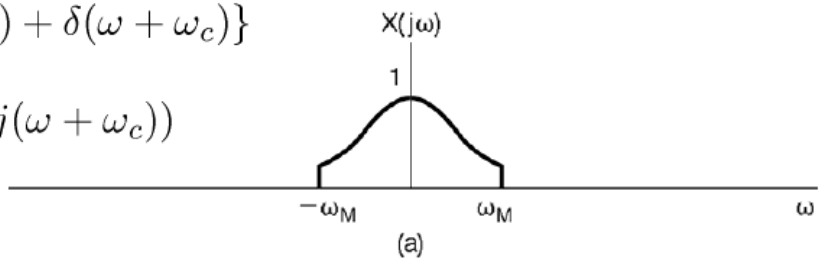
From A. S. Willsky, Signals and Systems Course, MIT

Amplitude Modulation

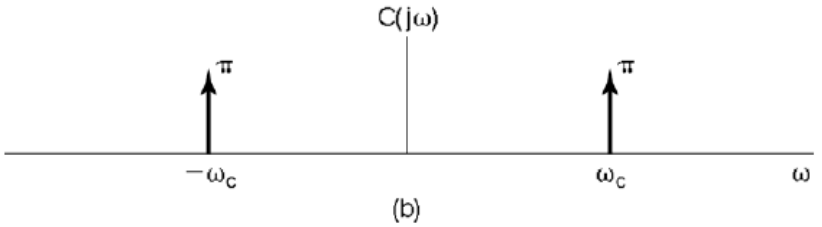
Sinusoidal AM

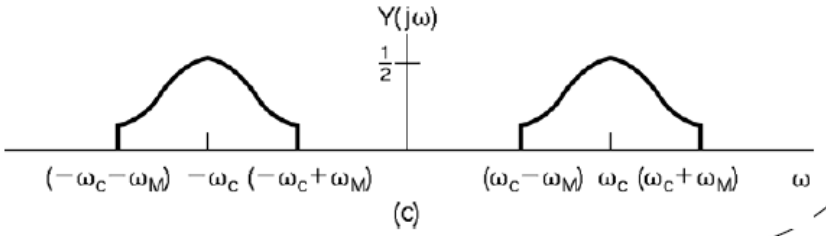


$$Y(j\omega) = \frac{1}{2\pi} X(j\omega) * \pi\{\delta(\omega-\omega_c)+\delta(\omega+\omega_c)\}$$
$$= \frac{1}{2} X(j(\omega-\omega_c)) + \frac{1}{2} X(j(\omega+\omega_c))$$



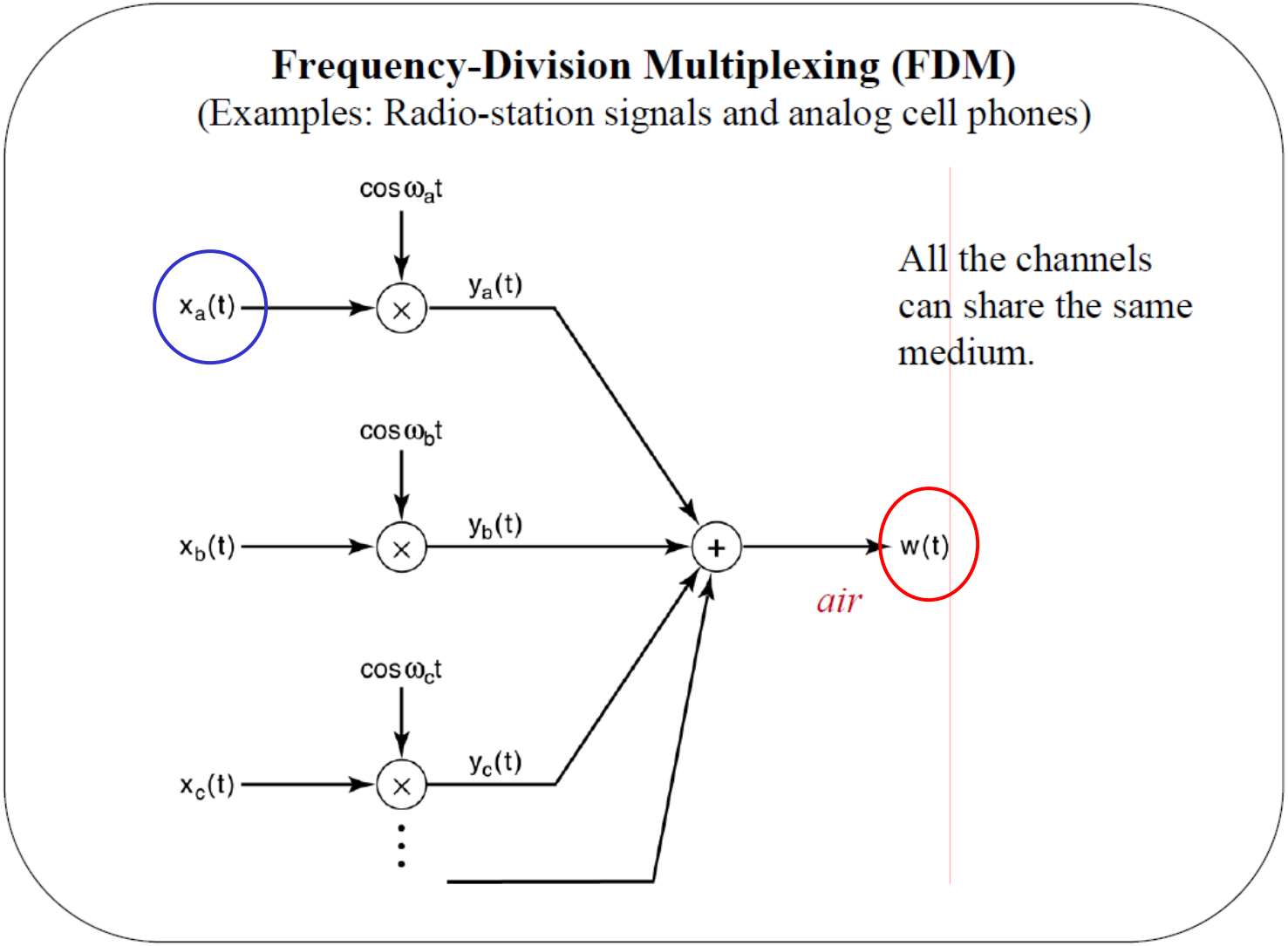
Drawn assuming
$\omega_c > \omega_M$





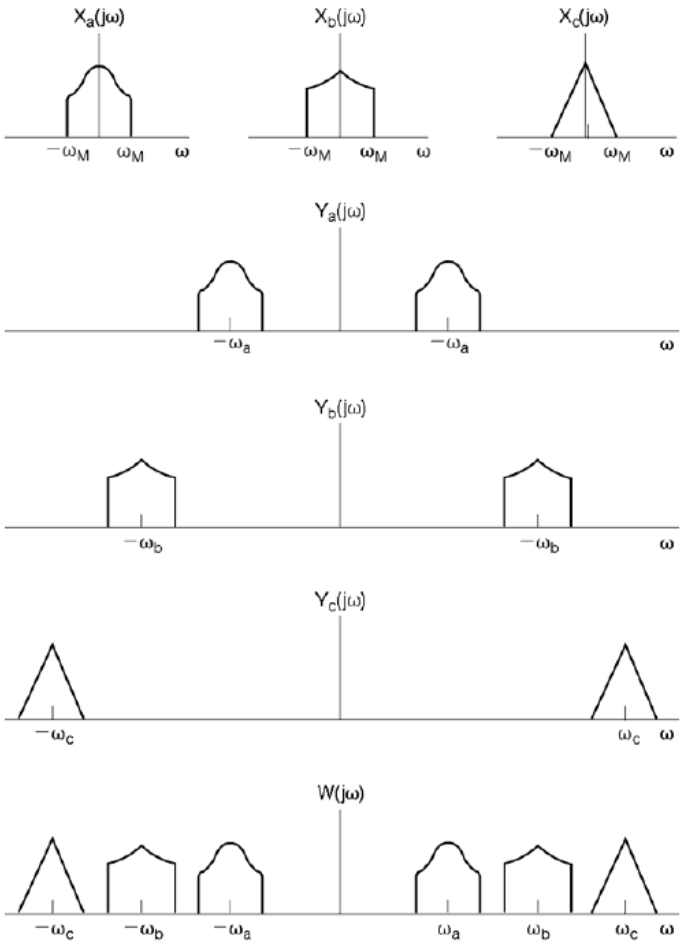
From A. S. Willsky, Signals and Systems Course, MIT

Frequency Division Multiplexing



From A. S. Willsky, Signals and Systems Course, MIT

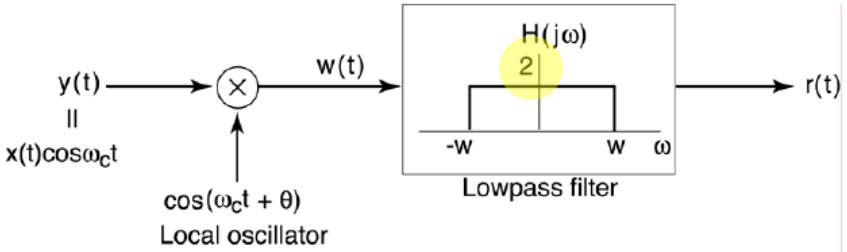
Frequency Division Multiplexing



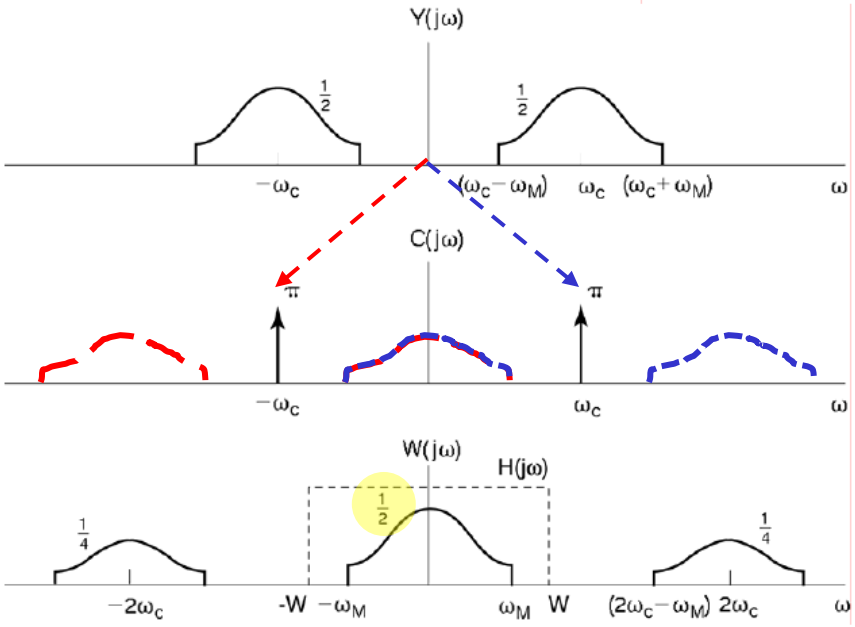
From A. S. Willsky, Signals and Systems Course, MIT

Amplitude Demodulation

Synchronous Demodulation of Sinusoidal AM



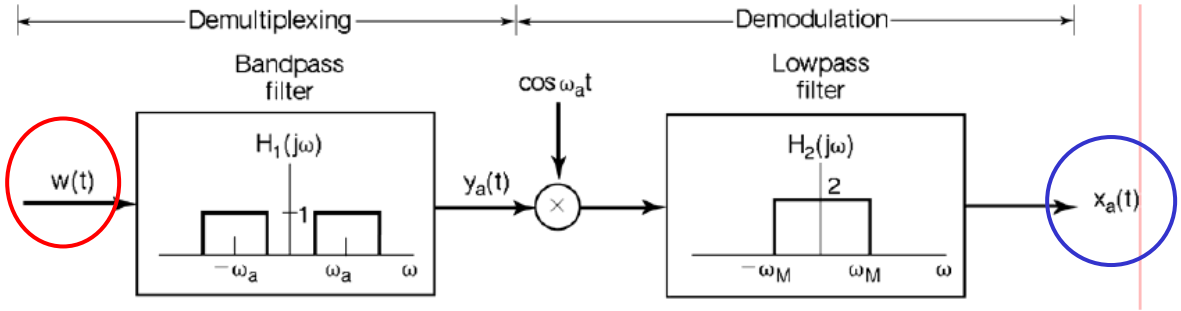
Suppose $\theta = 0$ for now,
$\Rightarrow$ Local oscillator is in phase with the carrier.



From A. S. Willsky, Signals and Systems Course, MIT

Demultiplexing and Demodulation

Demultiplexing and Demodulation



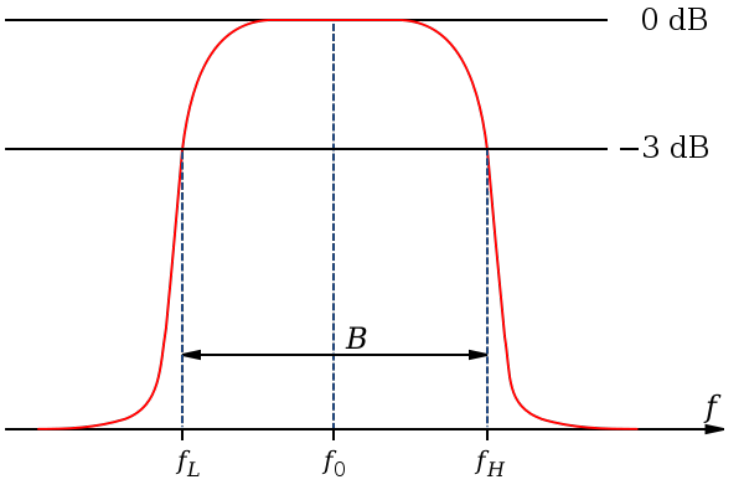
ω_a needs to be tunable

- Channels must not overlap ⇒ Bandwidth Allocation
- It is difficult (and expensive) to design a highly selective bandpass filter with a tunable center frequency

From A. S. Willsky, Signals and Systems Course, MIT

Bandwidth

- Can be defined by the OFCOM for multiple channels for a given purpose (in the overall spectrum)
- Can be defined for a single channel as follow:



- B = bandwidth
- f_0 = **carrier** (channel) frequency
- f_L = low cut-off frequency (typically defined at -3dB)
- f_H = high cut-off frequency (typically defined at -3dB)

Note (s. 39):
- carrier frequency DISAL Arduino Zigbee 2.4 GHz
- carrier frequency SensorScope stations ≈ 900 MHz

-3dB = 50% power (spectral density)
i.e. 10 log(0.5) = -3dB

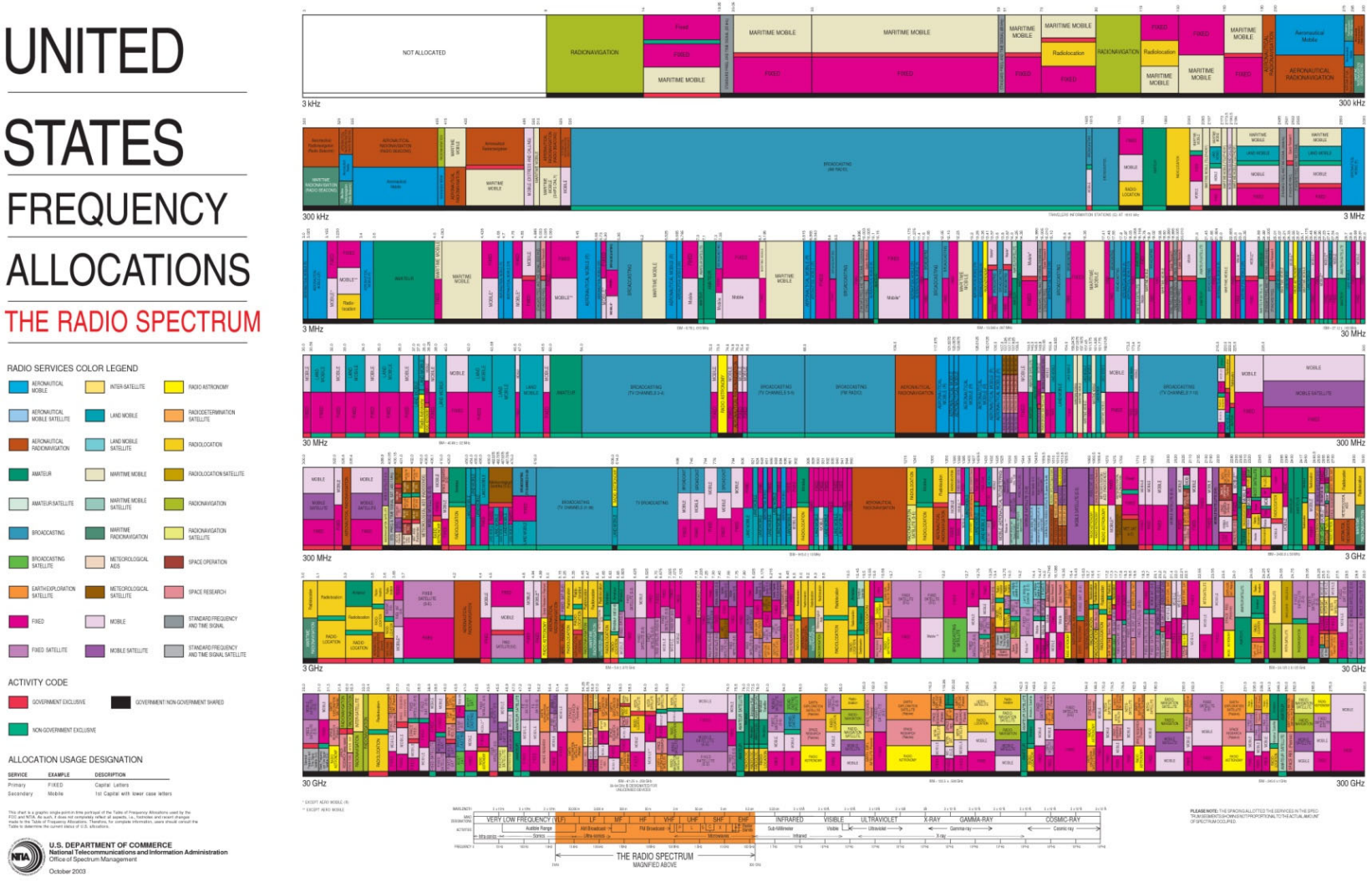
Bandwidth

- FM station broadcasting at 106,4 MHz
  → actually occupies 106,3 MHz – 106,5 MHz
  → Bandwidth = 200 kHz
- Mobile phone (GSM, 2G): 200 kHz (around 900 MHz)
- WLAN/WiFi: 5 MHz (around 2,4 GHz)
- Analog TV station: 6 MHz (around 180 MHz)

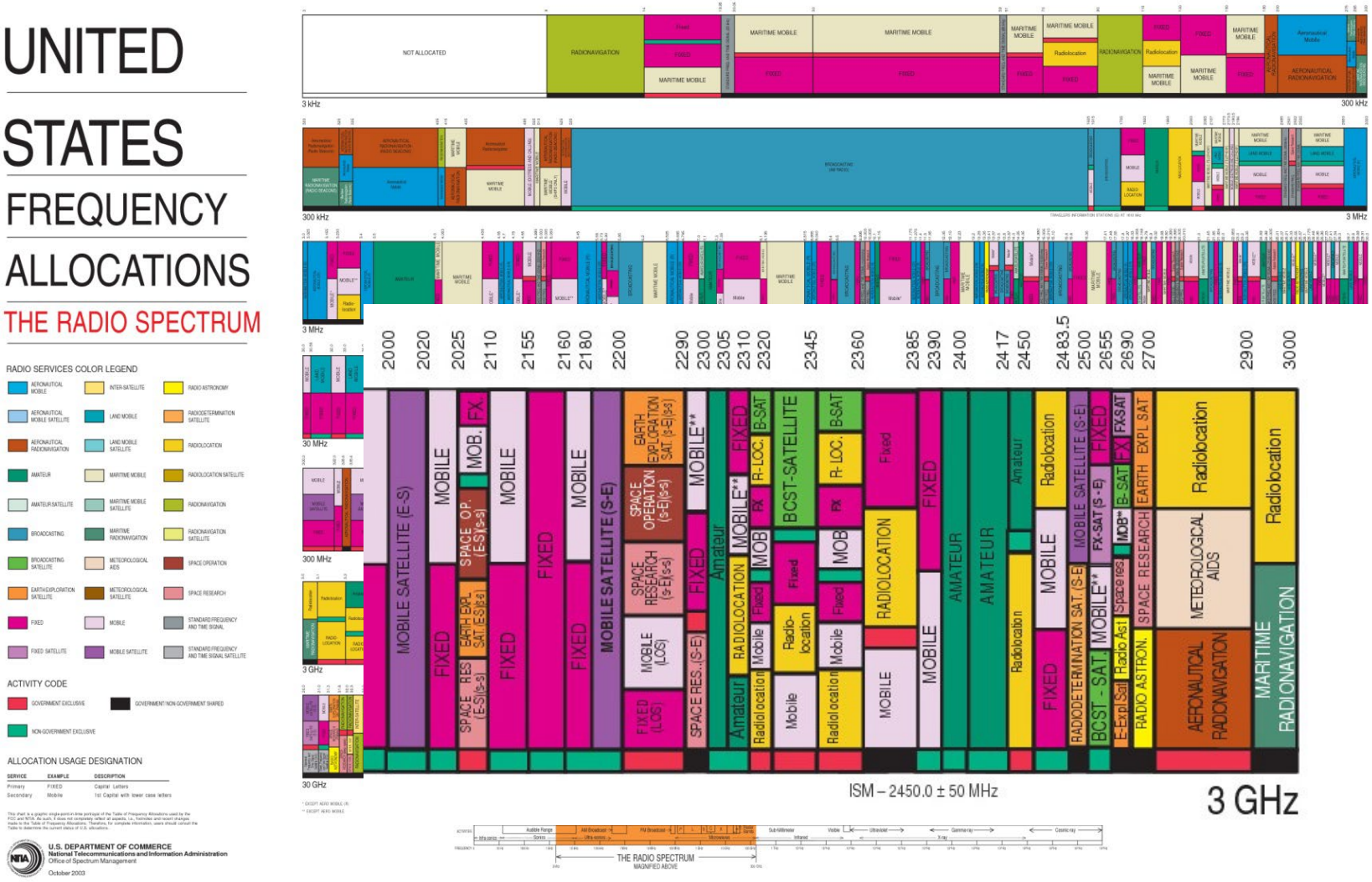
What does the bandwidth depend on?

Bandwidth [Hz] ↑ → Data rate (Throughput) [bits/s] ↑

Bandwidth


UNITED
STATES
FREQUENCY
ALLOCATIONS
THE RADIO SPECTRUM
RADIO SERVICES COLOR LEGEND
ACTIVITY CODE
ALLOCATION USAGE DESIGNATION
U.S. DEPARTMENT OF COMMERCE
National Telecommunications and Information Administration
Office of Spectrum Management
October 2003
3 kHz
300 kHz
3 MHz
30 MHz
300 MHz
3 GHz
30 GHz
300 GHz
THE RADIO SPECTRUM

Bandwidth

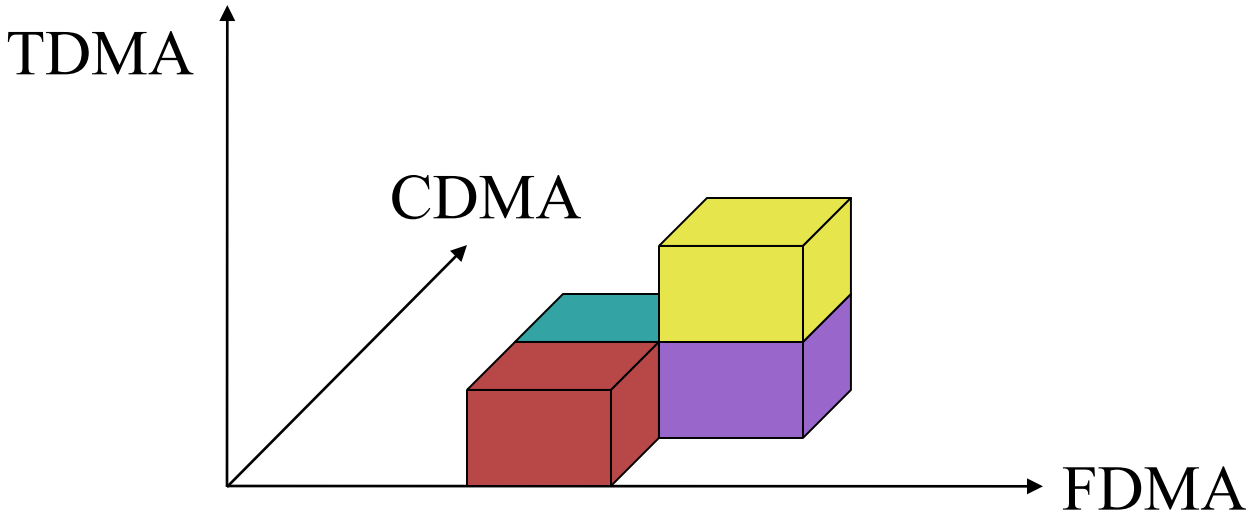

UNITED
STATES
FREQUENCY
ALLOCATIONS
THE RADIO SPECTRUM
RADIO SERVICES COLOR LEGEND
ACTIVITY CODE
ALLOCATION USAGE DESIGNATION
U.S. DEPARTMENT OF COMMERCE
National Telecommunications and Information Administration
Office of Spectrum Management
October 2003
3 kHz
300 kHz
3 MHz
30 MHz
300 MHz
3 GHz
30 GHz
2000
2020
2025
2110
2155
2160
2180
2200
2290
2300
2305
2310
2320
2345
2360
2385
2390
2400
2417
2450
2483.5
2500
2655
2690
2700
2900
3000
MOBILE SATELLITE (E-S)
FIXED
MOBILE
SPACE RES. (E-S)
SPACE OP. (E-S)
EARTH EXPL. SAT (E-S)
MOB.
FX
MOBILE SATELLITE (S-E)
EARTH EXPLORATION SAT. (S-E)
SPACE OPERATION (S-E)
SPACE RESEARCH (S-E)
MOBILE (LOS)
FIXED (LOS)
SPACE RES. (S-E)
MOBILE**
Amateur
RADIOLOCATION
Radiolocation
Mobile
FIXED
R-LOC.
B-SAT
MOB
FX
BCST-SATELLITE
Fixed
Radio-location
RADIOLOCATION
MOBILE
AMATEUR
Amateur
Radiolocation
Radiolocation
MOBILE
RADIODETERMINATION SAT. (S-E)
MOBILE SATELLITE (S-E)
FIXED
BCST - SAT.
MOBILE**
FX-SAT (S-E)
FX
B-SAT
MOB**
Space Res.
E-Expl.Sat
Radio Ast
EARTH EXPL SAT
SPACE RESEARCH
RADIO ASTRON.
Radiolocation
METEOROLOGICAL AIDS
AERONAUTICAL RADIONAVIGATION
Radiolocation
MARITIME RADIONAVIGATION
ISM – 2450.0 ± 50 MHz
3 GHz
THE RADIO SPECTRUM
MAGNIFIED ABOVE

Sharing the Medium

- CDMA (spread spectrum)
  - Code-Division MA
  - Using different transmission codes
  - e.g., GPS, WiFi, smart phones (3G/4G/5G), Zigbee
  - Interesting properties
    - Wide channels (less fading)
    - Concurrent communication
  - More complex demodulation

Throughput (bits/s)

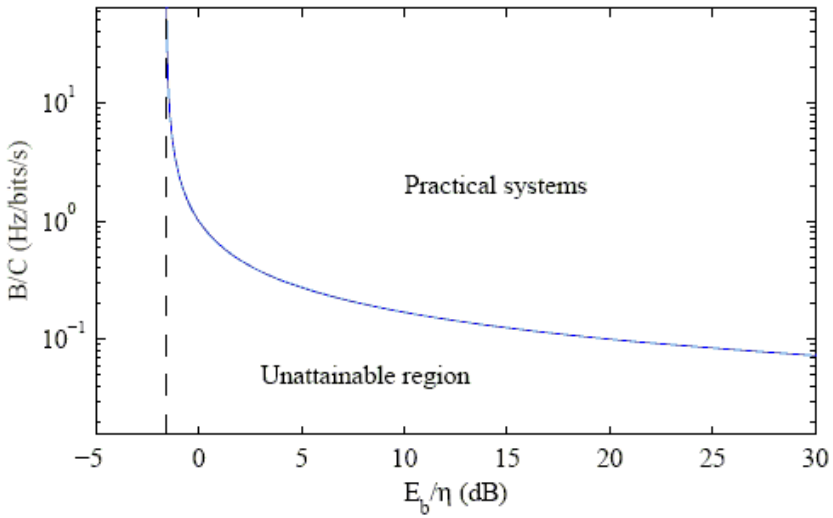
- TDMA, FDMA, CDMA can be combined
- Total throughput is shared



Shannon-Hartley Limit

- Hard theoretical limit on throughput
  - More bandwidth = higher throughput
  - More power (SNR) = higher throughput

Bandwidth "dilution"



$$C = B \log_2 \left(1 + \frac{S}{N}\right)$$

C: capacity (throughput)
B: bandwidth
S: signal power (W)
N: noise power (W)

Bit energy to noise-power spectral density ~ S/N

Wireless Communication
–
Power

Power

- Increased power
  - higher throughput
  - higher range
  - mobile systems: shorter battery life
  - increased health risk (?)
- Regulation
  - CH: OFCOM
  - e.g., WLAN: 100 mW

Power

- Unit: W (Watt)
  - Often written in dBm (decibels to 1 mW)

$$P_{dBm} = 10 \log_{10}(P_{mW})$$

- Gain / loss: factors
  - Often written in dB (decibels)

$$F_{dB} = 10 \log_{10}(F)$$

P_{dBm} and Gain/Loss factors

$$P_{dBm} = 10 \log\left(\frac{P_W}{1\text{mW}}\right)$$

- 1mW → 10 log(1mW/1mW) → 10 log(1) = 10·0 = 0 dBm
- 2mW → 10 log(2mW/1mW) → 10 log(2) ≈10·0.3 = 3 dBm
- 10mW → 10 log(10mW/1mW) → 10 log(10) = 10·1 = 10 dBm
- 100mW → 10 log(100mW/1mW) → 10 log(100) = 10·2 = 20 dBm

Factors in the chain become sums in a log form:

$$\log(x \cdot y) = \log(x) + \log(y)$$

Link Budget

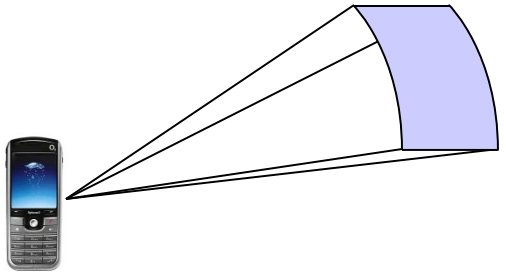
Typical WLAN link budget (100 m, dipole antennas):

TX power	100 mW	20 dBm
TX losses	*0.5	-3 dB
TX antenna gain	*1.6	+2 dB
Free space path loss	*1.0106*10^{-8}	-80 dB
RX antenna gain	*1.6	+2 dB
RX losses	*0.5	-3 dB
RX power	0.00000064 mW	*-62 dBm*
RX sensitivity	0.000000003 mW	*-85 dBm*
Margin	200	23 dB

Free Space Path Loss (Friis Law)

- Signal power decay in air:

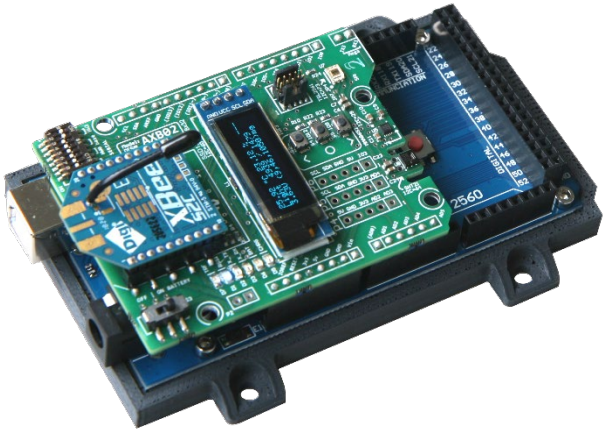
$$L = \left(\frac{4\pi df}{c}\right)^2$$

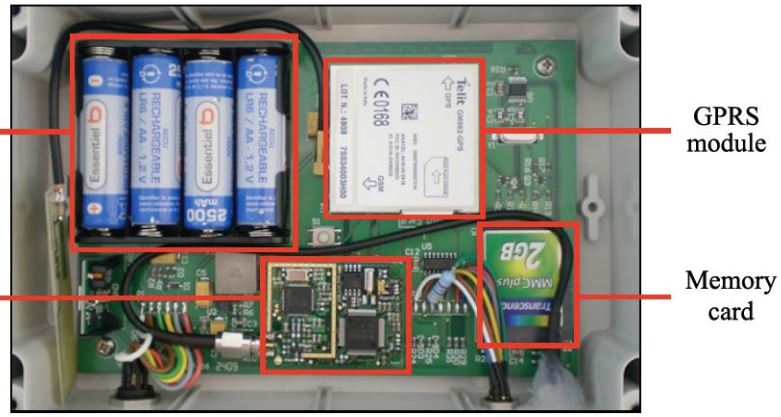


$$L_{dB} = 20\log_{10}(d) + 20\log_{10}(f) - 147.56$$

- Proportional to the square of the distance d
- Proportional to the square of the frequency f
  - high frequency = high loss
  - low frequency = low throughput

DISAL Arduino Xbee vs. SensorScope Station





DISAL Arduino Xbee kit

- Microcontroller:
  - ATMega 2560
- Transceiver:
  - Silicon Labs EM357 (part of the Xbee 802.15.4 module)
  - 2.4 GHz carrier
  - Throughput: up to 250 kbps
  - Range: up to 90 m

SensorScope Data Logger

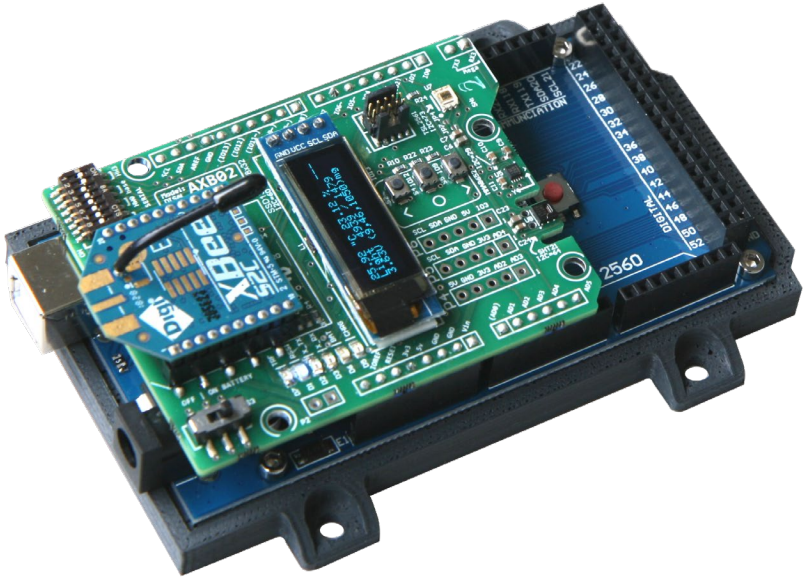
- Microcontroller:
  - TI MSP430
- Transceiver:
  - Semtech XE 1205
  - 868 and 915 MHz carriers
  - Throughput: up to 153 kbps
  - Range: up to 2 km

Computation

Programming Embedded Systems

- Programming embedded systems is all about **resource management**, i.e. dealing with delays and time constraints imposed by the environment and limitations of the hardware
- **Energy, memory and computation** are often very limited and precious in real-time embedded system applications
- When programming an embedded system, you must bear in mind **its limitations** and find appropriate techniques for dealing with them (e.g., finding application-specific **trade-offs** for the use of the available resources)

DISAL Arduino Xbee Node



Week 6, s. 27

- Computation: Arduino Mega 2560 board (ATMega 2560 microcontroller)
- Communication: Zigbee-complaint transceiver (Xbee)
- Actuation: on-board mini-display
- Perception:
  - Light sensor (TSL2561-T, ams)
  - Humidity and Temperature sensor (SHT20, Sensirion)
  - Digital Accelerometer (MMA8652, NXP)
- Power: 14 hours autonomy fully on (70 mA on 1000 mAh Li-Po battery)
- Can be programmed in C leveraging Arduino libraries

Arduino Mega 2560 Board

- 8-bit microcontroller
- 16 MHz clock, up to 16 MIPS
- 256 kB Flash memory (code)
- 8 kB SRAM (data)
- 16 analog inputs, 54 digital I/O
- 4 UARTs, USB connector

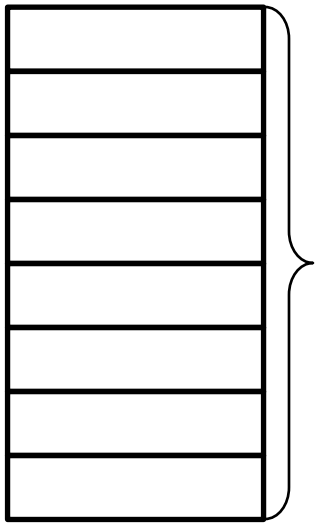
Week 6, s. 28

Memory on the Arduino Kit

- All memory **inside the** ATMega 2560 microcontroller
- **On-board memory**: **8 kB** of RAM (**random-access**, **volatile**) for data and 256 kB of Flash Memory (**sequential-access**, **non-volatile**) for programs
- **Permanent storage of data:** communication with the base station, which will take care of it
- Memory is a **very precious resource** in embedded systems!
- It can become a **serious bottleneck**, for instance in case the embedded system is endowed with a rich sensor such as **a camera**

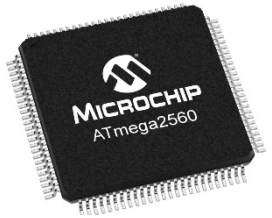
The Arduino Kit with a Camera

1 image = 922 kB



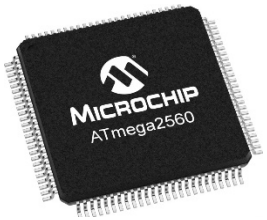
On-board memory

ATMega 2560 memory = 8kB



- Camera: 640x480 pixels, RGB color (3 bytes), 30 frames/s
- The microcontroller has not **enough memory** available for storing even a single image (640 x 480 x 3 B = 922 kB)

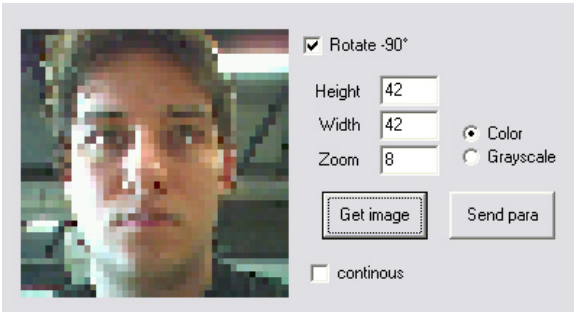
The Arduino Kit with a Camera

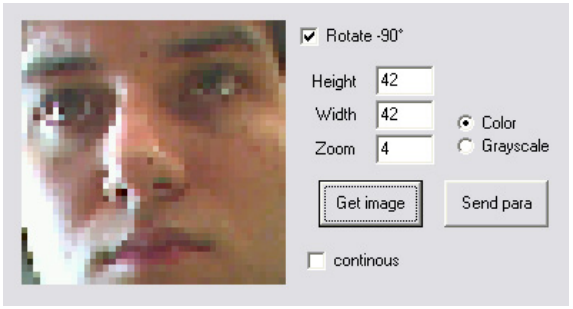


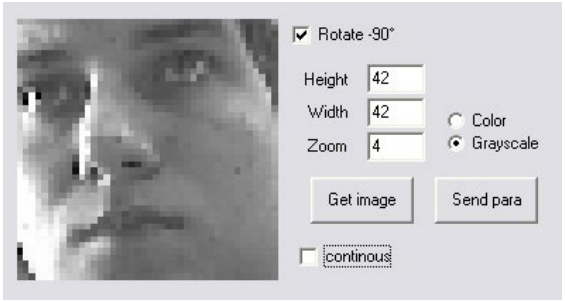
- The on-board computation resources are also not following the data flow:
  - Pixels H x V x RGB x fps
  - 640 x 480 x 3 x 30 = 27 MB/second
  - An ATMega 2560 can execute max 16 MIPS (millions of instructions/second, i.e. 1 instruction takes at least 66 ns)

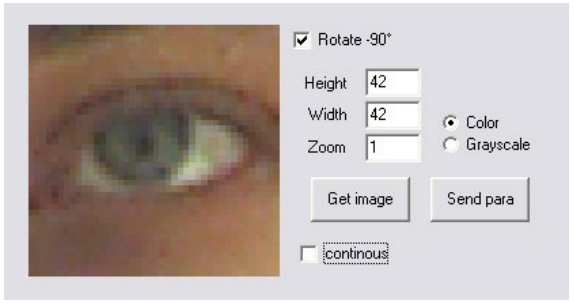
Solution: Image Processing

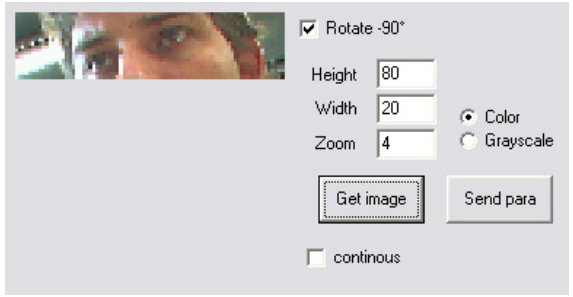
- By using **downsampling**, you can reduce the size of the image while retaining useful information.
- Typically, we can acquire a 40x40 downsampled color image at 4 frames per second
- Without color, we can go up to 8 frames per second
- This will allow up to roughly 1000 IPS per image processing











Real-Time Computing

Real-Time: Definition

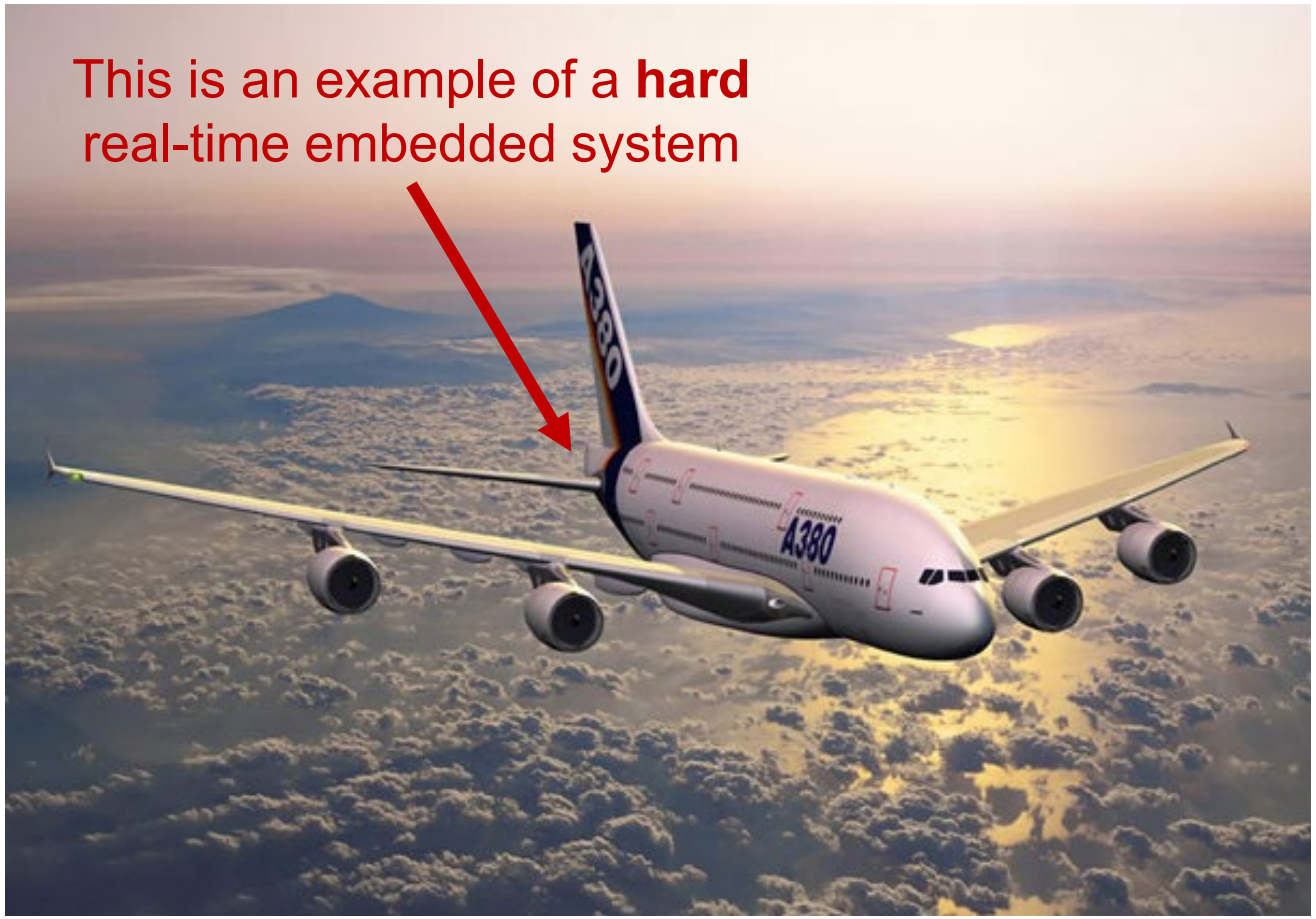
- A system is said to be **real-time** if the total correctness of an operation depends not only upon its logical correctness, but also upon **the time** in which it is performed (Wikipedia).
- One can distinguish two types of real-time systems:
  - **Hard real-time systems**: the completion of an operation after its deadline is considered useless (ultimately, this may lead to a critical failure or the destruction of the system).
  - **Soft real-time systems:** the completion of an operation after its deadline decreases the service quality (e.g., dropping frames in a video streaming).

- **Note:** a real-time system is not necessarily a high-performance system, and vice versa!
  - An Arduino-based board can be a real-time system (if properly programmed), but it will never be a high-performance system!

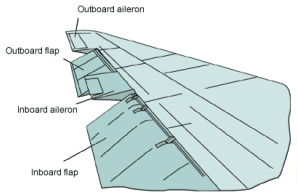
Real-time: It Matters

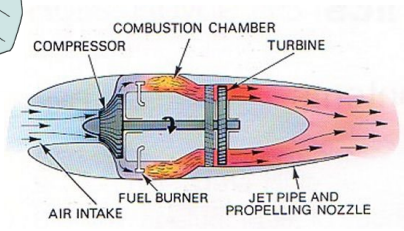
time

Read and digitalize instruments data	Filter data	Calculate updated roll, yaw and pitch	Update flaps and throttle



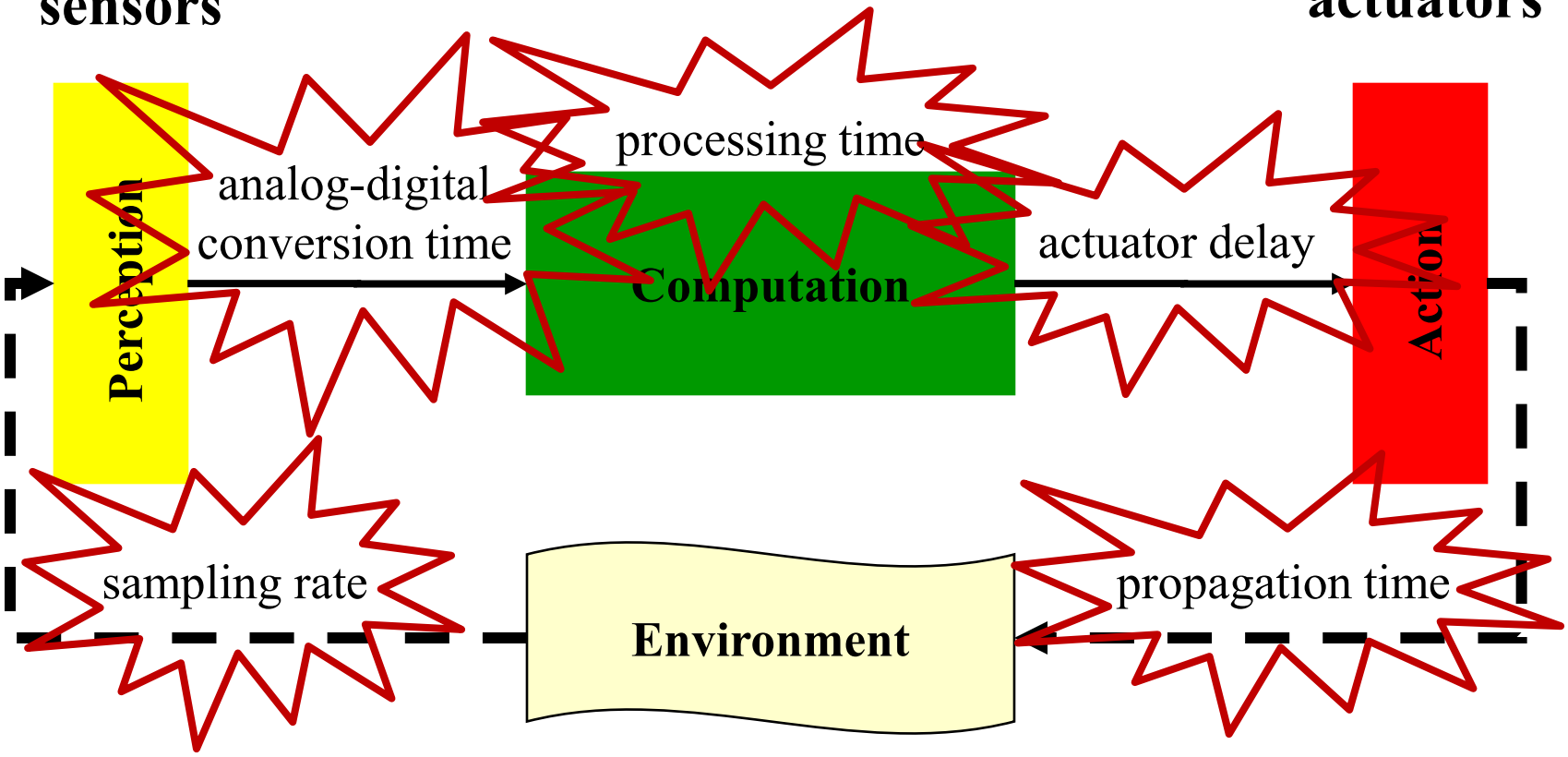
Perception-to-Action Loop





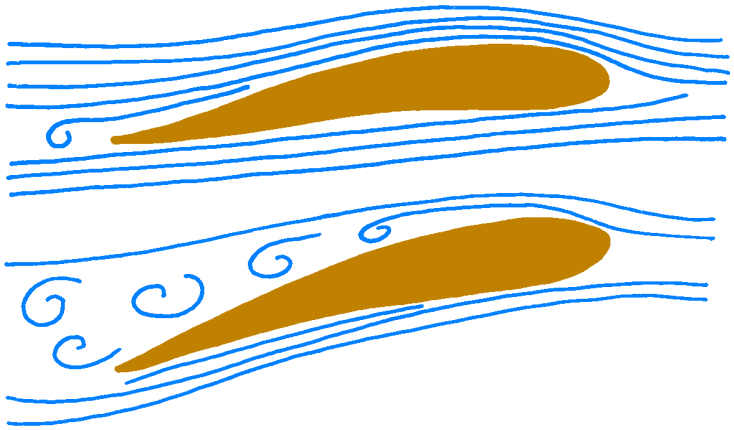
sensors

actuators



Perception-to-Action Delay

Sideairbag in a car: perception-to-action delay < 10 mSec



Wing vibration in a plane: perception-to-action delay < 5 mSec

Interrupts

Interrupt - Motivation

- A computer is managing multiple peripherals, e.g., mouse, screen , keyboard, camera, etc.
- An embedded system is similarly managing multiple devices: e.g., sensors, transceivers, actuators, buttons, etc.
- These devices need occasionally some attention from the microcontroller or the microprocessor but we cannot predict when these events occur
- We need a way to let the Central Process Unit (CPU) working (awake mode) or resting (asleep mode) between events but offer service to the devices above when requested

Possible Solutions for Handling Attention Requests

- Polling:
  - Regular checking of all connected devices
  - Can be efficient with events arriving rapidly
  - Inefficient use of CPU time when there is no request
- Interrupts
  - Each device has an interrupt line (typically a wired connection)
  - When the device needs attention, a signal (interrupt) is emitted on the line and the CPU execute the service

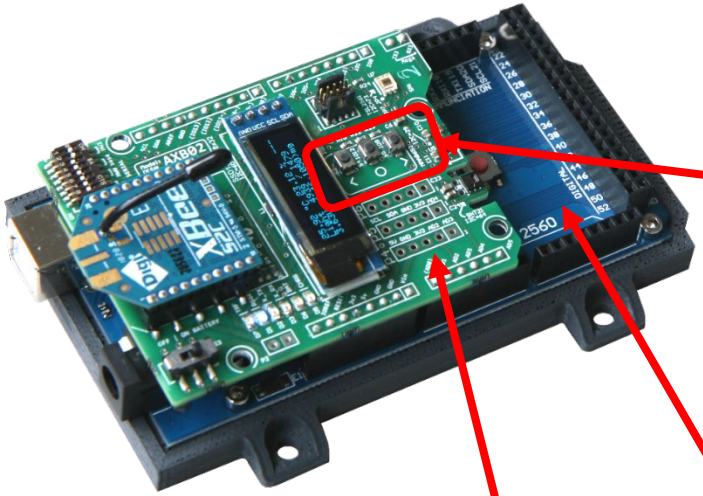
Interrupts – Definition and Types

- An interrupt is a special signal that triggers a change in execution, i.e. a call to a subroutine which is usually referred to as an **Interrupt Service Routine** (ISR)
- The interrupt does **not** wait for the current program to finish. It is unconditional and immediate
- The **processor state is saved** before handling the interrupt; and the current program can be therefore resumed from there once the interrupt has been handled
- Interrupts are very useful for interacting with **hardware devices (hardware interrupts)**, but there are also **software interrupts**, which are triggered by a program

An example of Interrupt Use for Reading a Button State

Reading a Button on the Arduino Kit

Interrupts can be triggered by interactions with the hardware

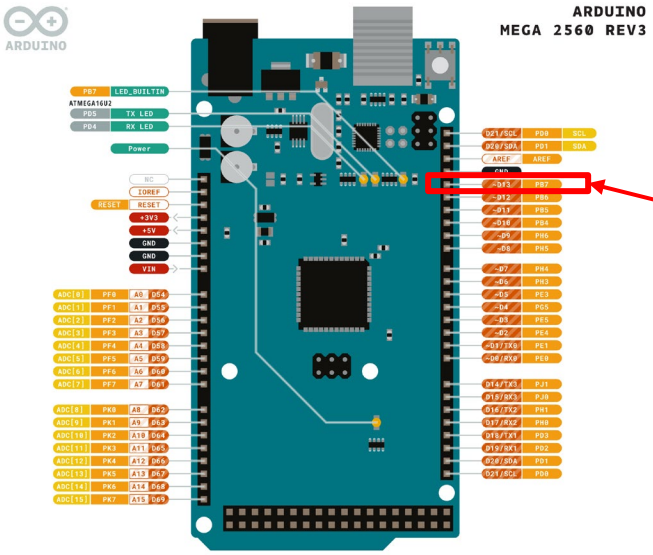


Ex. pressing a button on the deck → triggers ISR in software

Arduino Mega 2560 board

Arduino custom deck

Configuring Interrupt Pins



We must configure the digital pin connected to the button to be an interrupt pin

Identify which digital pin is connected to the button

Interrupt Software Architecture

```
void setup(){
cli();
PCICR |= 0b00000001; // Enables Port C Pin Change Interrupts
PCMSK0 |= 0b10000000; // PCINT7 -> Digital pin 13 = "UP" button
sei();
}
```

Clear all registers

Set registers

Enable interrupts

Interrupt set-up and initialization:

- Clear **registers**
- Consult microcontroller manual to find which **register** to set to enable interrupt on the selected digital pin
- Enable interrupts

Write ISR -> what happens when interrupt is triggered

Note: The code syntax used above is referring to the instruction set of a 8-bit ATMMega 2560 microcontroller by Microchip Technology, using Arduino-specific functions

An example of Interrupt Use for Ensuring Accurate Timing

"Super" Loop Software Architecture

- Assume that you want X() to be executed at precisely 18 kHz (T = 55 µs)
- You know that X() duration is about 21 µs. Therefore, you wait 34 µs (using a calibrated active loop)
- The *super loop* or *endless loop* is often required because **we have no operating system to return to** at the end of the program!
  - ✓ Simple, efficient and portable!
  - × Timing is inaccurate!
  - × High power consumption!

```
void main() {
  // prepare for task X
  X_init();

  // super loop
  while (1) {
    X(); // perform task X
    wait(34); // active wait
  }
}
```

- This type of software architecture **does not guarantee** accurate timing (particular critical for hard real-time systems)

Interrupts as Solution

- Interrupts are also very useful when coupled with a **timer**. They allow a function or subroutine to run periodically with **an accurate timing.**
- Another benefit of using interrupts is that in some microcontrollers you can use a **wake-from-sleep** interrupt. This allows the microcontroller to go into a low power mode, and be wakened later on by an interrupt.
- **Note:** the syntax used on s. 64 is referring to the instruction set of a **16-bit PIC microcontroller** by Microchip Technology. The _ISRFAST macro is defined in this context for declaring a ISR (e.g., can automatically be associated to one of the four available shadow **registers** of a PIC microcontroller usable for implementing a timer).

Interrupt Software Architecture

```
void _ISRFAST _T1Interrupt(void) {
  IFS0bits.T1IF = 0;  // clear interrupt flag

  X(); // perform task X (e.g., read sensor,
       // save in a buffer, increment the index)
}
```

Interrupt Service Routine

```
void InitTMR1(void) {
  T1CON = 0;
  T1CONbits.TCKPS = 0;      // prescaler = 256
  TMR1 = 0;                 // clear timer 1
  PR1 = (MILLISEC/18.00);   // 18KHz interrupt (T = 55 us)
  IFS0bits.T1IF = 0;        // clear interrupt flag
  IEC0bits.T1IE = 1;        // set interrupt enable bit
  T1CONbits.TON = 1;        // start Timer1
}
```

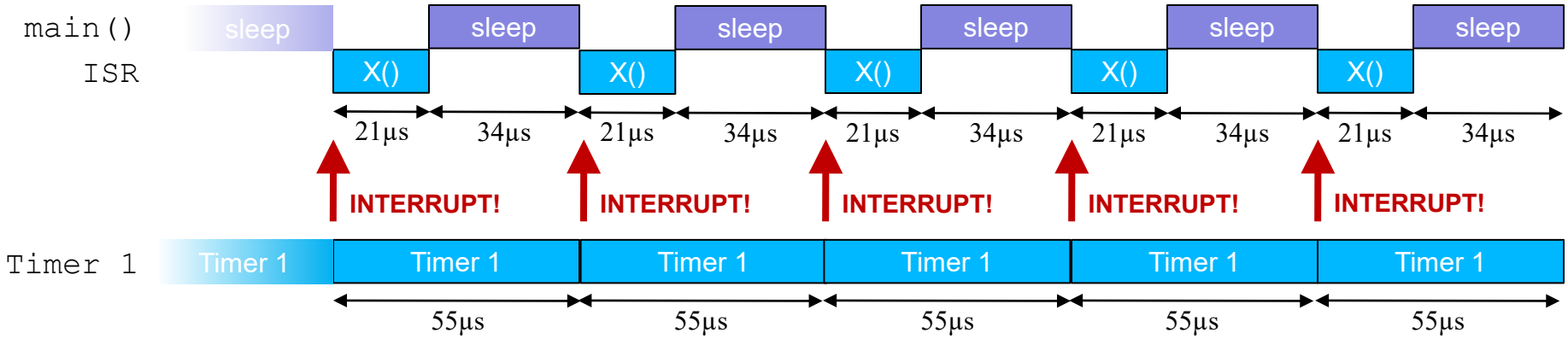
Interrupt set-up and initialization

```
void main() {
  // initialize Timer 1
  InitTMR1();

  sleep(); // go to sleep, waiting for interrupt

}
```

Initialize timer

Sleep forever, waiting for interrupt

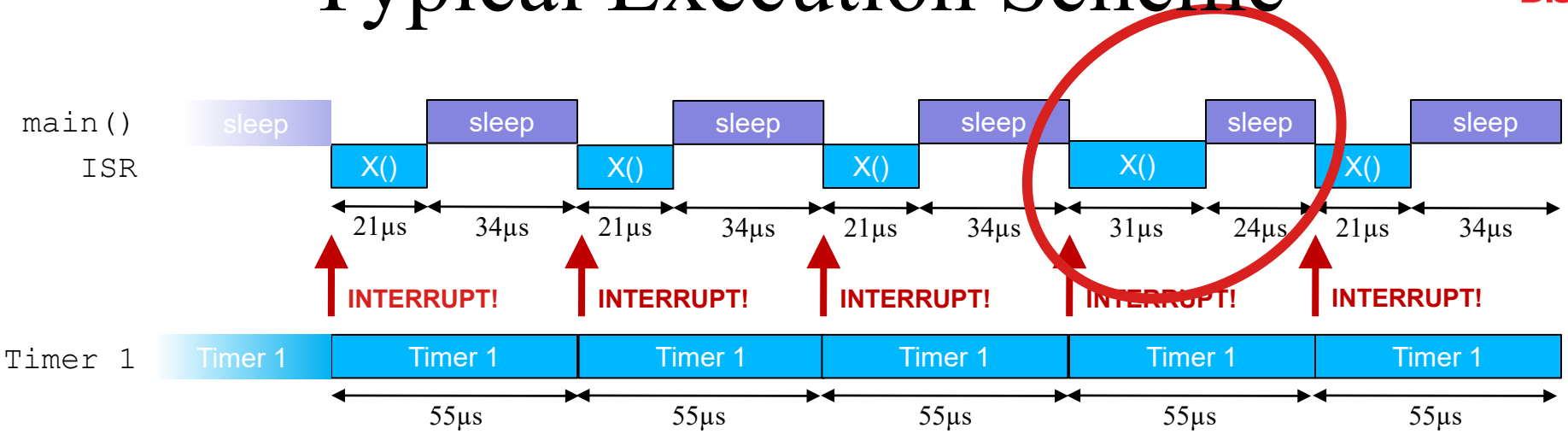
Typical Execution Scheme



- The main loop gets executed, but the only thing it does is to put the microcontroller in sleep mode
- In sleep mode, timers and interrupts are still active
- Each 55µs, `Timer 1` triggers an interrupt, thus wakening the microcontroller
- The task `X()` gets executed during approximately 21µs; then, the microcontroller can get back to sleep for 34µs.

What if the task X() lasts longer than expected?

Typical Execution Scheme



The timing remains unchanged:
X() still runs every 55 µs!

Bonus: the microcontroller sleeps about 60% of the time (in other words, you can save a lot of energy, depending on the system).

Conclusion

Take Home Messages

- Communication plays a key role in embedded systems; in particular radio wireless communication is crucial for field instruments
- Some key concepts in communication systems
  - Bandwidth, throughput, TDMA, FDMA, CDMA
  - Shannon-Hartley limit on actual throughput
- Communication power can be very significant and is a nonlinear function of the communication distance and the carrier frequency
- Programming embedded systems is all about resource management: memory and input/output devices are two examples of resources
- A key method for handling real-time constraints in embedded systems is using interrupts
- Interrupts can be generated by hardware (e.g., an external button, a sensor, an internal timer flag) and software (e.g., device drivers, program execution errors).

Additional Literature – Week 7

Books

- U. Madhow, “Introduction to Communication Systems”, Cambridge University Press, 2014.
- E. A. Lee and S. A. Seshia, “Introduction to Embedded Systems, A Cyber-Physical Systems Approach”, MIT Press, 2017.
- D. J. Russell, “Introduction to Embedded Systems Using ANSI C and the Arduino Development Environment”, Morgan & Claypool 2010, reprinted by Springer 2022.

Course pointers at EPFL

- https://edu.epfl.ch/coursebook/en/microprogrammed-embedded-systems-EE-310
- https://edu.epfl.ch/coursebook/en/embedded-system-design-CS-476