

Embedded Systems

1 - Introduction

© Lothar Thiele

Computer Engineering and Networks Laboratory

Michele Magno

D-ITET Center for project-based Learning

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Organization Summary

- **Lectures** are held on Mondays from 14:15 to 16:00 in ETF C1 until further notice. Life streaming and slides are available via the web page of the lecture **from the previous year (some part will be new!!)**. In addition, you find audio and video recordings of most of the slides as well as recordings of this years and last years life streams on the web page of the lecture.
- **Exercises:** The exercises are made available before the date of the exercise. During an exercise, teaching assistants will summarize the lecture material required to solve the exercise questions, give hints how to approach the solution, answer your questions and discuss the correct solution at the end of the exercise. The electronic version of the exercise questions and solutions will be placed online after the exercise has been conducted. Exercises will take place from 4pm - 6pm on Wednesdays and Fridays as indicated in the timetable.
- **Laboratory:** Laboratory exercises will be conducted in teams of 2 people. Due to the large number of students, there will be two successive ES-Lab sessions at the days of the ES-Lab (see timetable): Session A from 4pm - 6pm on Wednesday and session B from 4pm - 6pm on Friday. Both sessions will take place in rooms D61.1 and D96. If you still need to sign up you can do this in the lecture (preferred) or via email to the lecture coordinator.

Organization

WWW: <https://pbl.ee.ethz.ch/education/embedded-systems.html>

Lecture: Michele Magno <michele.magno@pbl.ee.ethz.ch>; Lothar Thiele, thiele@ethz.ch;

Coordination: Seonyeong Heo (ETZ D97.7) <seoheo@ethz.ch>

References:

- *P. Marwedel*: Embedded System Design, Springer, ISBN 978-3-319-85812-8/978-3-030-60909-2, 2018/2021. **You can find it also as open access!**
- *G.C. Buttazzo*: Hard Real-Time Computing Systems. Springer Verlag, ISBN 978-1-4614-0676-1, 2011.
- *Edward A. Lee and Sanjit A. Seshia*: Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition, MIT Press, ISBN 978-0-262-53381-2, 2017.

Sources: The slides contain ideas and material of J. Rabaey, K. Keutzer, M. Wolf, P. Marwedel, P. Koopman, E. Lee, P. Dutta, S. Seshia, and from the above cited books.

1 - 2

Lecture Organization

Autumn Semester 2022			
227-0124-00L Embedded Systems			
Data for restriction on registration			
Number of places	Current registration	Priority for the primary target group until	Waiting list until
250	250	25.09.2022	02.10.2022
			134

1 - 3

1 - 4

Further Material via the Web Page (under development)

Lecture Slides

All lecture slides are available for download as a bundle:

- Embedded Systems lecture slides (single page format) [↓](#)
- Embedded Systems lecture slides (4on1 page format) [↓](#)

Lecture Recordings

Life Recordings Autumn 2021

The life recordings of the lectures in Autumn Semester are available at the following link:
Embedded Systems Life Recordings AS 2021.

Life Recordings Autumn 2020

The life recordings of last years lecture are available at the following links:

- Lecture 1: Chapters 1. Introduction and 2. Software Development
- Lecture 2: Chapters 2. Software Development and 3. Hardware-Software Interface

Audio and Videos of Selected Chapters

Some of the chapters are documented via carefully recorded videos. They contain some of the slides as well as audio explanations.

- 1. Introduction
- 2. Software Development
- 3. Hardware Software Interface

Exercises and Laboratory	
Generic Documents	
Embedded System Companion	Supplementary Material
Remote Installation Instructions	
Documents for Lab 0	
Handout	Source (code)
Slides and videos	Solution (code and handout)
Documents for Lab 1	
Handout	Source (code)
Slides and videos	Solution (code and handout)
Documents for Lab 2	
Handout	Source (code)
Slides and videos	Solution (code and handout)
Documents for Lab 3	

1 - 5

When and where?

	When	Where
Lectures	Monday 14:15 - 16:00	ETF C1
Exercises	Wednesday 16:15 - 18:00 Friday 16:15 - 18:00	ETF E1 ETF E1
Labs	Wednesday 16:15 - 18:00 Friday 16:15 - 18:00	ETZ D61.1, ETZ D96.1 ETZ D61.1, ETZ D96.1

Date	Lecture	Exercise	Lab
26.09.2022	1. Introduction 2. Software Development		
28.09./30.09.2022			0. Prelab [SH]
03.10.2022	2. Software Development 3. Hardware-Software Interface (Processor vs Microcontroller)		

1 - 6

What will you learn?

- Theoretical foundations and principles of the analysis and design of embedded systems.
- Practical aspects of embedded system design, mainly software design.

The course has three components:

- Lecture**: Communicate principles and practical aspects of embedded systems.
- Exercise**: Use paper and pencil to deepen your understanding of analysis and design principles . **EXAMS** 😊
- Laboratory (ES-Lab)**: Introduction into practical aspects of embedded systems design. Use of state-of-the-art hardware and design tools.

1 - 7

Please read carfully!!

- <https://pbl.ee.ethz.ch/education/embedded-systems.html>

Exercises and Laboratory

We urgently ask all students to install required software for the labs on their own hardware. You can find the installation instructions on [GitLab](#). We have tested this setup on PCs and Laptops with an USB port that run Windows 11, as well as Linux Mint and Linux Ubuntu 20.04 and 22.04. You are not allowed to enter ETZ D61.1 or ETZ D96.1 during the laboratory hours if you do not have an allocated slot.

1 - 8

Practical part of this course.

Discovery kit for IoT node with STM32U5 series



- Ultra-low-power STM32U585AI6Q microcontroller based on the **Arm® Cortex®-M33** core with Arm® TrustZone®, 2 Mbytes of Flash memory and 786 Kbytes of SRAM, and SMPS in UFBGA169 package
- 512-Mbit Quad-SPI Flash memory, 64-Mbit Octo-SPI PSRAM, 256-Kbit PC EEPROM
- USB FS, Sink and Source power, 2.5 W power capability
- 802.11 b/g/n compliant Wi-Fi module from MACHIP
- Bluetooth® Low Energy from STMicroelectronics
- **MEMS sensors** from STMicroelectronics
 - 2 digital microphones
 - Relative humidity and temperature sensor
 - 3-axis magnetometer
 - 3D accelerometer and 3D gyroscope
 - Pressure sensor, 260-1260 hPa absolute digital output barometer
 - Time-of-flight and gesture-detection sensor
- Ambient-light sensor
- Authentication and security for peripherals and IoT devices from STMicroelectronics
- 2 user LEDs
- User push-button
- Reset push-button
- Board connectors
 - USB Type-C®
 - **ARDUINO® Uno V3 expansion connectors**
 - Camera module expansion connector
 - 2x STMod+ expansion connectors
 - Pmod™ expansion connector
- **Flexible power-supply options:** ST-LINK USB V_{DD} USB connector, or external sources
- On-board ST-LINK-V3E debugger/programmer with USB re-enumeration capability: mass storage, Virtual COM port, and debug port
- Comprehensive free software libraries and examples available with the STM32CubeUS MCU Package
- Support of a wide choice of Integrated Development Environments (IDEs) including IAR Embedded Workbench®, MDK-ARM, and STM32CubeIDE

1 - 9

Be careful and please do not ...



1 - 10

You have to return the board at the end!



1 - 11

Embedded Systems - Impact

1 - 12

Embedded Systems

Embedded systems (ES) = **information processing systems embedded into a larger product**

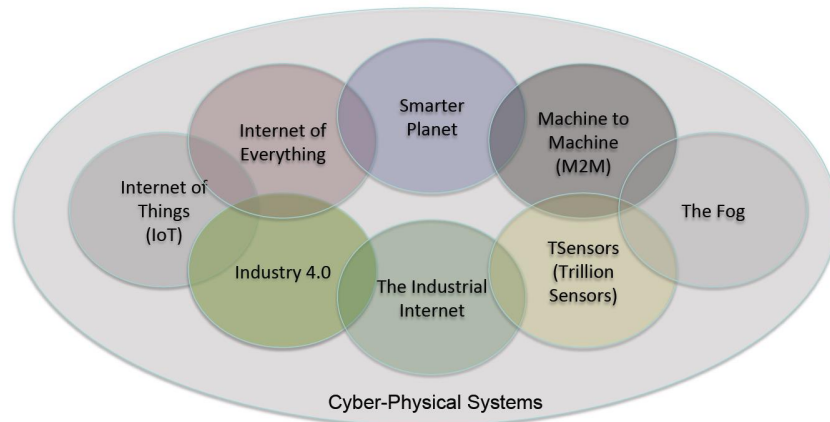
Examples:



Often, the main reason for buying is not information processing

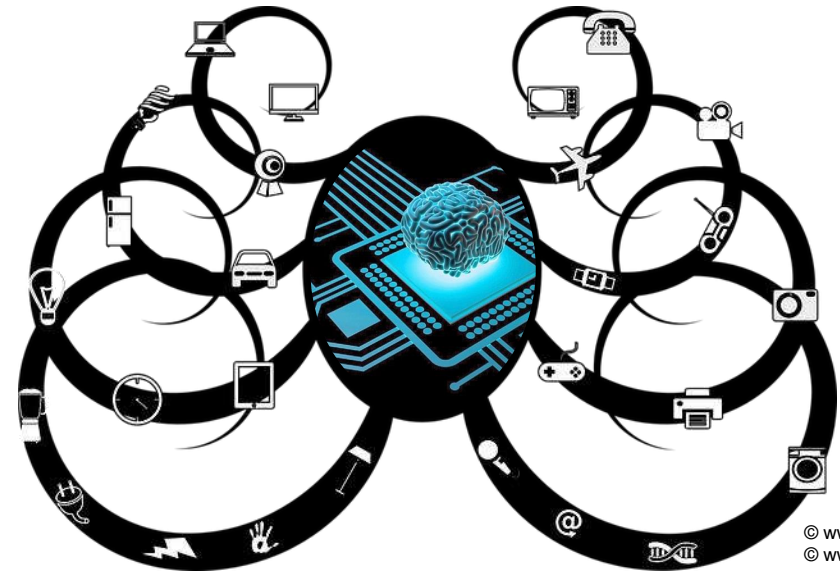
1 - 13

Many Names – Similar Meanings



© Edward Lee

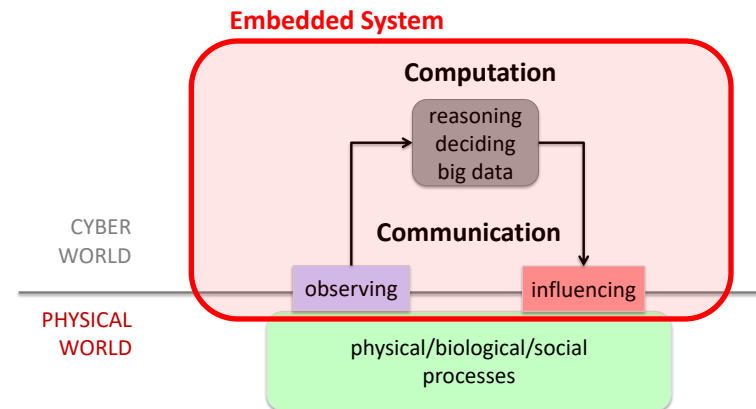
1 - 15



© www.braingrid.org
© www.openpr.com

1 - 14

Embedded System



Use feedback to influence the dynamics of the physical world by taking smart decisions in the cyber world

1 - 19

Reactivity & Timing



Embedded systems are often reactive:

- Reactive systems must **react to stimuli** from the system environment :

„A reactive system is one which is in continual interaction with its environment and executes at a pace determined by that environment“ [Bergé, 1995]

Embedded systems often must meet **real-time constraints**:

- For hard real-time systems, right answers arriving too late are wrong. All other time-constraints are called soft. A **guaranteed system response** has to be explained without statistical arguments.

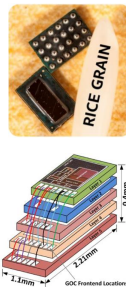
„A real-time constraint is called hard, if not meeting that constraint could result in a catastrophe“ [Kopetz, 1997].

1 - 20

Efficiency & Specialization

- Embedded systems must be **efficient**:

- Energy** efficient
- Code-size** and **data memory** efficient
- Run-time** efficient
- Weight** efficient
- Cost** efficient

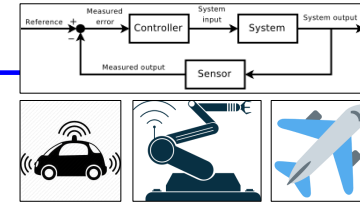


Embedded Systems are often **specialized** towards a certain application or application domain:

- Knowledge about the expected behavior and the system environment at design time is exploited to **minimize resource usage** and to **maximize predictability and reliability**.

1 - 22

Predictability & Dependability



CPS = cyber-physical system

“It is essential to **predict** how a CPS is going to behave under any circumstances [...] **before** it is deployed.”^{Maj14}

“CPS must **operate dependably**, safely, securely, efficiently and in real-time.”^{Raj10}

^{Maj14} R. Majumdar & B. Brandenburg (2014). Foundations of Cyber-Physical Systems.

^{Raj10} R. Rajkumar et al. (2010). Cyber-Physical Systems: The Next Computing Revolution.

1 - 21

Comparison

Embedded Systems:

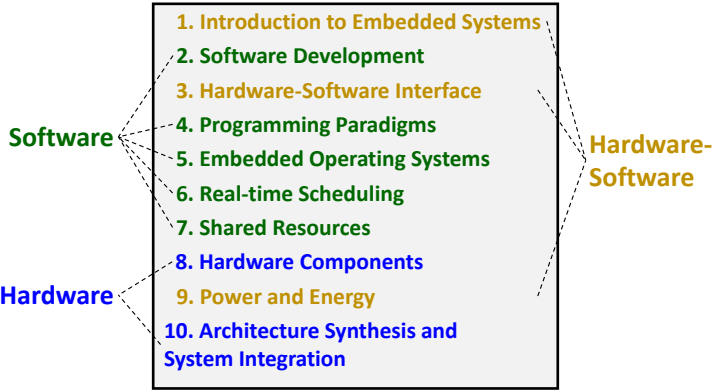
- Few applications that are known at design-time.
- Not programmable by end user.
- Fixed run-time requirements (additional computing power often not useful).
- Typical criteria:
 - cost
 - power consumption
 - size and weight
 - dependability
 - worst-case speed

General Purpose Computing

- Broad class of applications.
- Programmable by end user.
- Faster is better.
- Typical criteria:
 - cost
 - power consumption
 - average speed

1 - 23

Lecture Overview



Agenda Embedded Systems

DATE	Topic
26.09 / Magno	Lecture 1: Chapters 1. Introduction
03.10/ Magno	Lecture 2: Chapters 2. Software Development and Chapter 3. Hardware-Software Interface (Processor VS MCU, Core ARM cortex-M 33,
10.10 / Magno	Lecture 3: Chapter 3 Hardware-Software Interface (Clocks, Memory, Interrupts of ARM Cortex M33)
17.10 / Magno	Lecture 4: Chapter 3 Hardware-Software Interface (Serial Interface)
24.10 / Magno	Lecture 5: Chapter 3 Hardware-Software Interface (ADC/ DAC/Timers/PWM)
31.10 / Dr. Heo	Summary, Labs, exercises
07.11 / Thiele	Lecture 6: Chapter 4 Programming Models and Chapter 5 Operating systems
14.11 / Thiele	Lecture 7: Operating Systems and. Chapter 6 Aperiodic and Periodic Scheduling
21.11 / Thiele	Lecture 8 L Chapter 6. Aperiodic and Periodic Scheduling
28.11 / Thiele	Lecture 9: Chapter 7 Shared Resources
05.12 / Magno	Lecture 11: Chapter 8 Power and Energy
12.12 / Michele	Lecture 12: Chapter 9 Sensing; energy considerations, data rates, buffering, spatial vs. temporal density.
19.12 / Michele	Lecture 13: Chapter 10 Architecture Synthesis and System Integration

1 - 24

1 - 25

Components and Requirements by Example



1 - 26

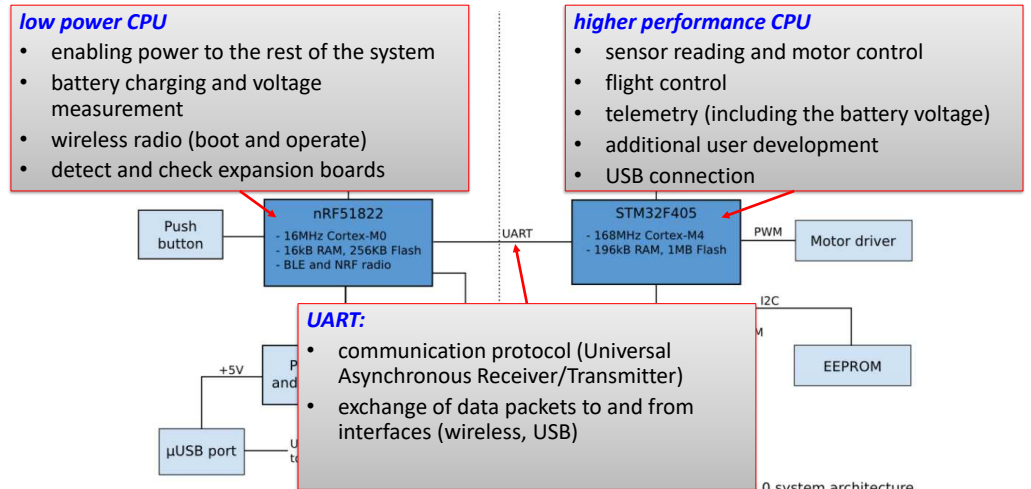


Components and Requirements by Example

- Hardware System Architecture -



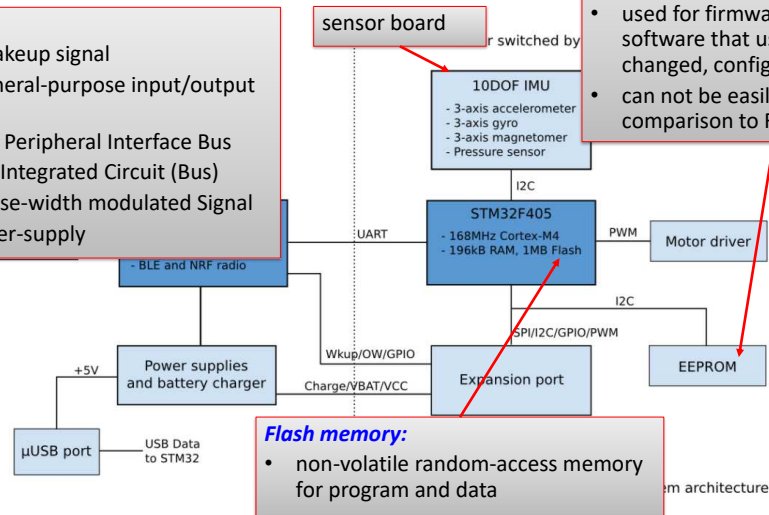
High-Level Block Diagram View



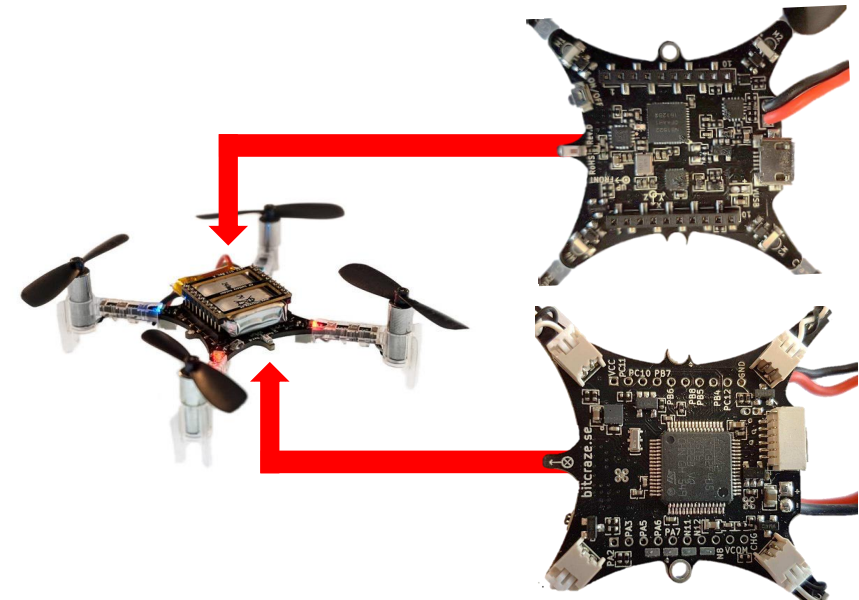
High-Level Block Diagram View

Acronyms:

- Wkup: Wakeup signal
- GPIO: General-purpose input/output signal
- SPI: Serial Peripheral Interface Bus
- I2C: Inter-Integrated Circuit (Bus)
- PWM: Pulse-width modulated Signal
- VCC: power-supply

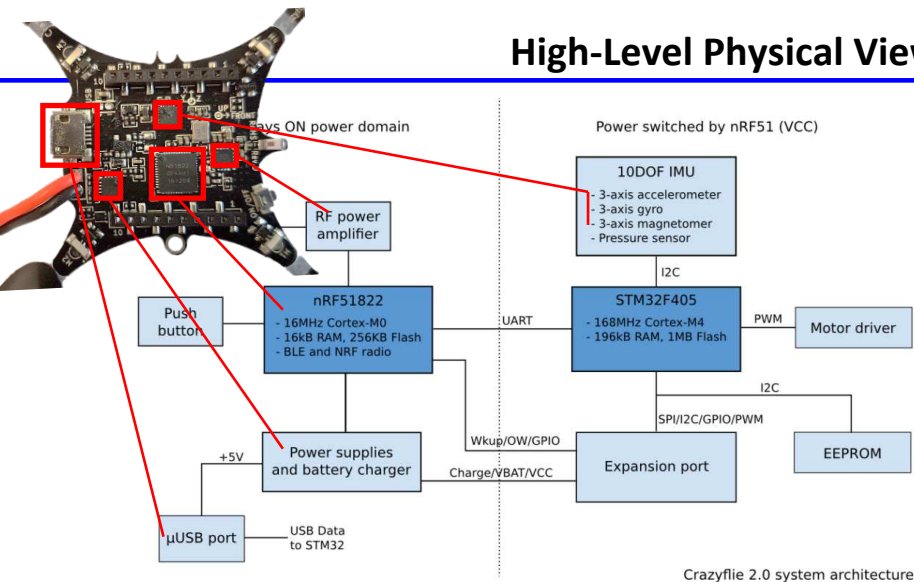


1 - 32



1 - 33

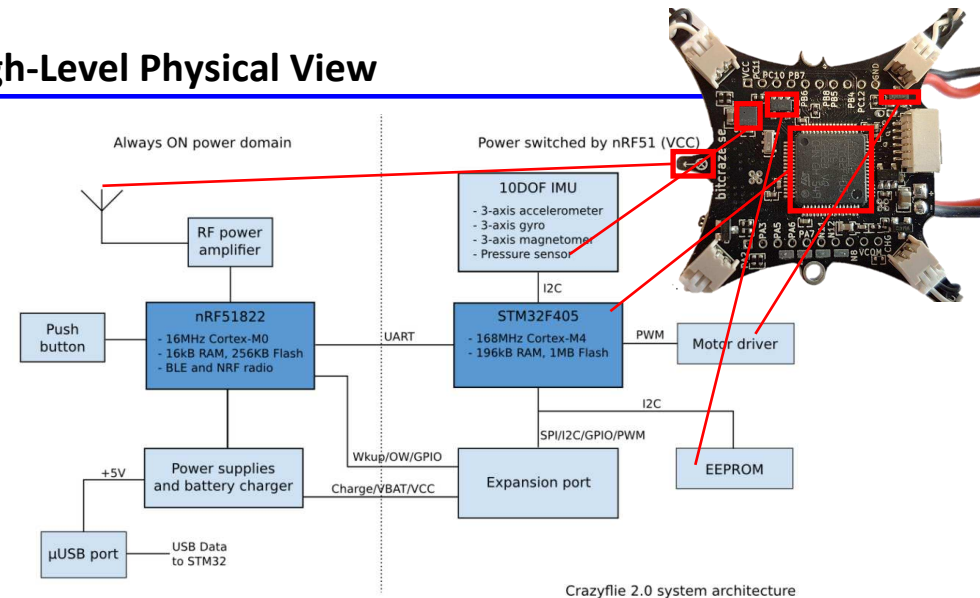
High-Level Physical View



Crazyflie 2.0 system architecture

1 - 34

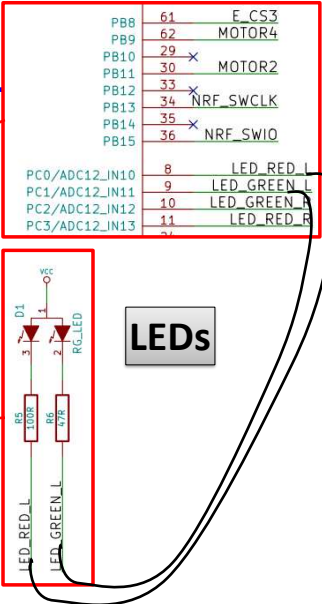
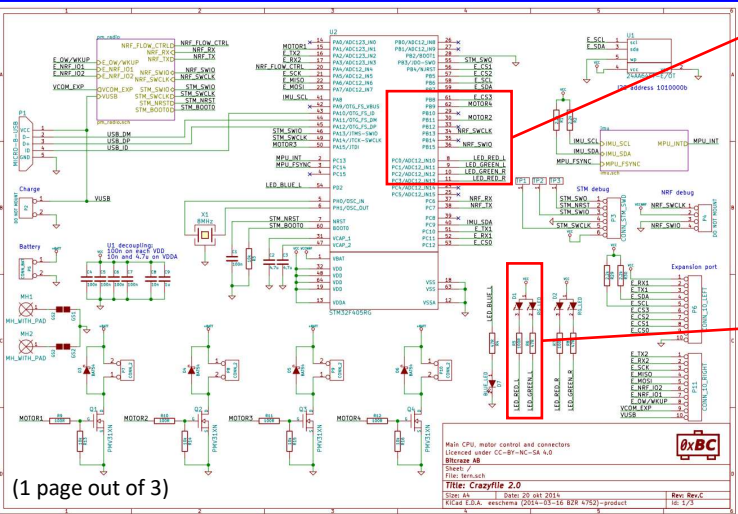
High-Level Physical View



Crazyflie 2.0 system architecture

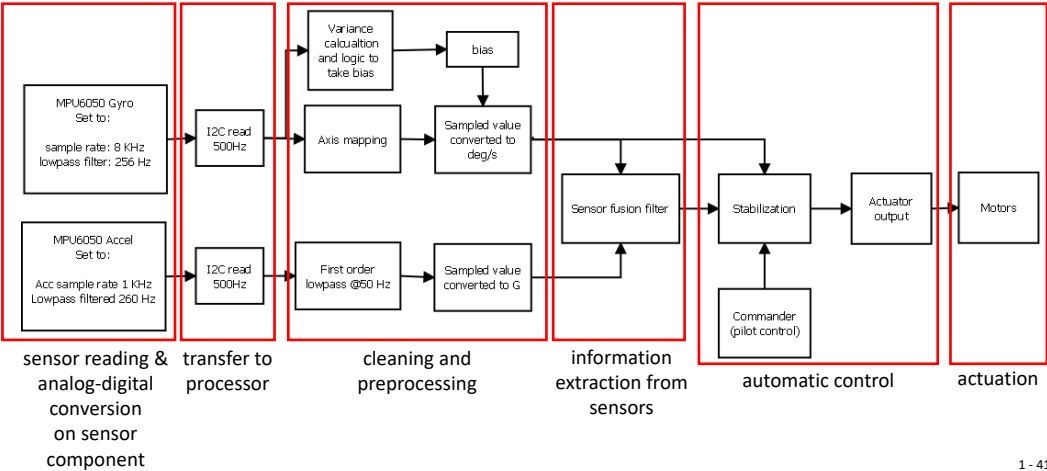
1 - 35

Low-Level Schematic Diagram View



High-Level Software View

Block diagram of the stabilization system:



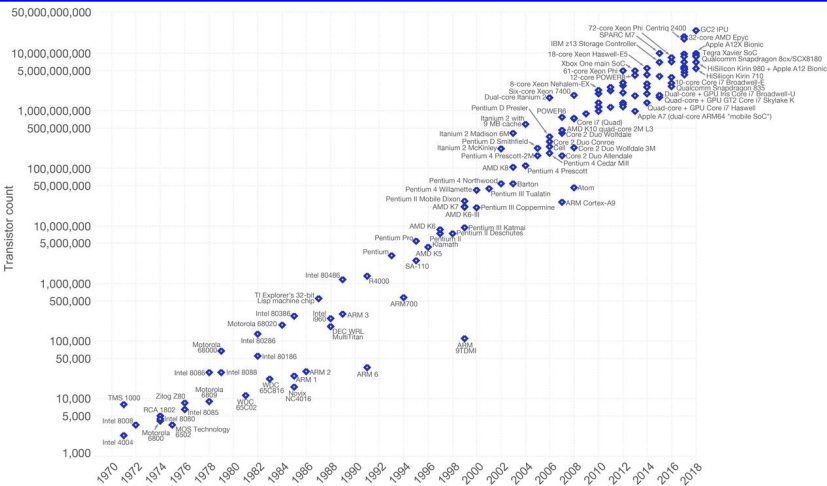
Components and Requirements by Example

- Processing Elements -



What can you do to increase performance?

From Computer Engineering



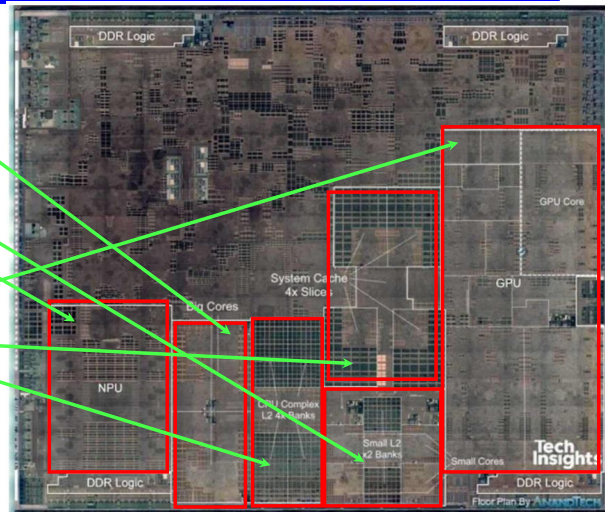
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldInData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

From Computer Engineering

iPhone Prozessor A12

- 2 processor cores - high performance
- 4 processor cores - less performant
- Acceleration for Neural Networks
- Graphics processor
- Caches



What can you do to decrease power consumption?

1 - 47

1 - 48

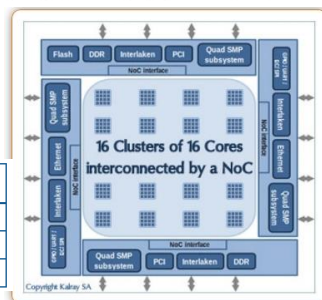
Embedded Multicore Example

Trends:

- Specialize multicore processors towards real-time processing and low power consumption (parallelism can decrease energy consumption)
- Target domains:



Core Generation	Number of Processing Cores	GFLOPS/W	GOPS/W
Andy	256	25	75
Boston (2014)	256	50	80
Coolidge (2015)	64/256/1024	75	115



Why does higher parallelism help in reducing power?

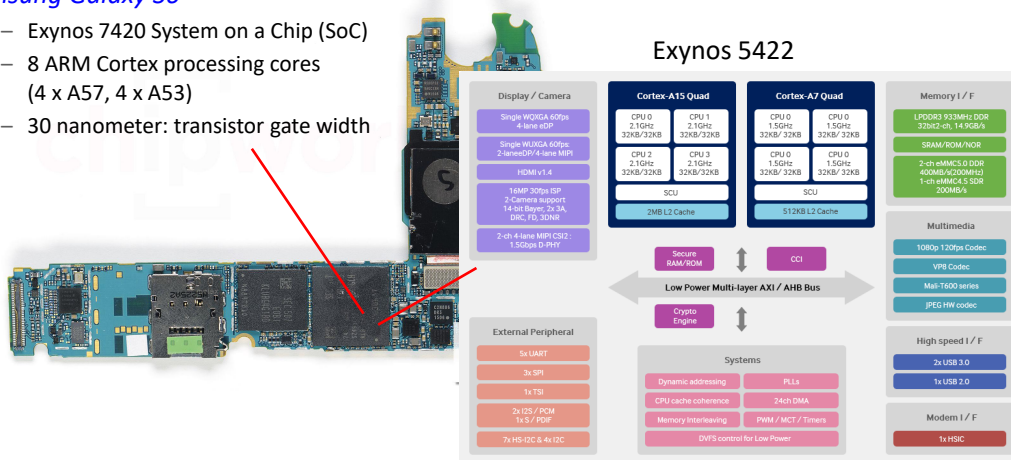
1 - 49

1 - 50

System-on-Chip

Samsung Galaxy S6

- Exynos 7420 System on a Chip (SoC)
- 8 ARM Cortex processing cores (4 x A57, 4 x A53)
- 30 nanometer: transistor gate width



1 - 51

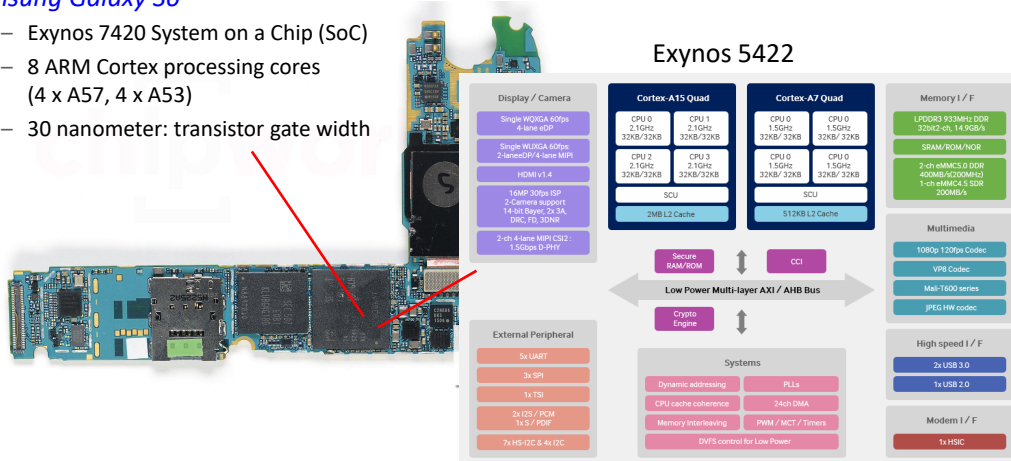
How to manage extreme workload variability?

1 - 52

System-on-Chip

Samsung Galaxy S6

- Exynos 7420 System on a Chip (SoC)
- 8 ARM Cortex processing cores (4 x A57, 4 x A53)
- 30 nanometer: transistor gate width

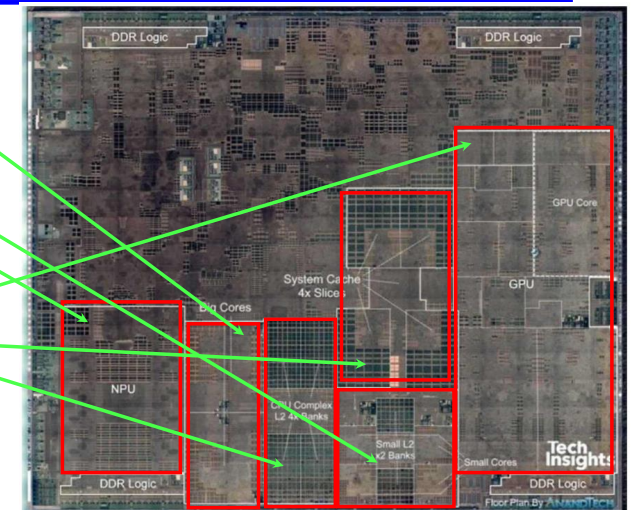


1 - 53

From Computer Engineering

iPhone Prozessor A12

- 2 processor cores - high performance
- 4 processor cores - less performant
- Acceleration for Neural Networks
- Graphics processor
- Caches



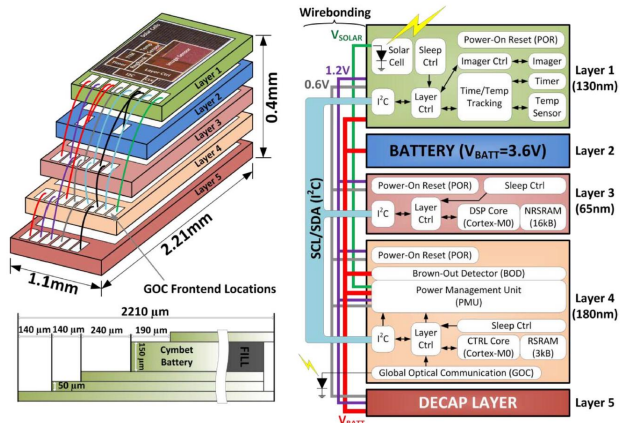
1 - 54

Components and Requirements by Example - Systems -

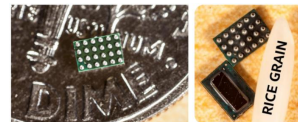
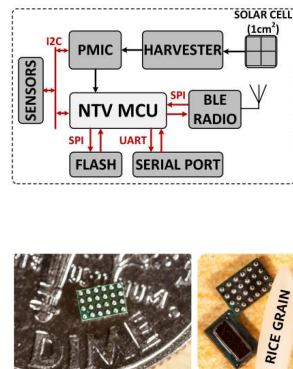


1 - 56

Zero Power Systems and Sensors



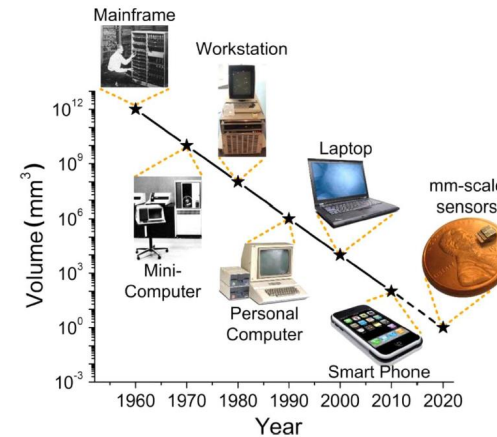
IEEE Journal of Solid-State Circuits,
Jan 2013, 229-243.



IEEE Journal of Solid-State
Circuits, April 2017, 961-971.

1 - 58

Zero Power Systems and Sensors



Streaming information to
and from the physical world:

- “Smart Dust”
- Sensor Networks
- Cyber-Physical Systems
- Internet-of-Things (IoT)

1 - 57

Trends ...

- *Embedded systems are communicating with each other*, with servers or with the cloud. Communication is increasingly wireless.
- *Higher degree of integration* on a single chip or integrated components:
 - Memory + processor + I/O-units + (wireless) communication.
 - Use of networks-on-chip for communication between units.
 - Use of homogeneous or heterogeneous multiprocessor systems on a chip (MPSoC).
 - Use of integrated microsystems that contain energy harvesting, energy storage, sensing, processing and communication (“zero power systems”).
 - The complexity and amount of software is increasing.
- *Low power and energy constraints* (portable or unattended devices) are increasingly important, as well as temperature constraints (overheating).
- There is increasing interest in *energy harvesting* to achieve long term autonomous operation.

1 - 59

Learning objective of Today Class

- Where we are what we will do.
- What are Embedded Systems and CPS
- Both hardware and software are important and you will learn
- Processors and evolution of them
- Trends and topics

Research focus: Embedded Systems and Wireless Smart Sensors for a better and greener world

- An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is **designed for a specific function** or for specific functions within a larger system.



Main Teaching at ETH

- Embedded systems – 300 students per year
- Machine Learning on Microcontrollers – 100 students per year
- Over 100 students projects per year.



1 - 60

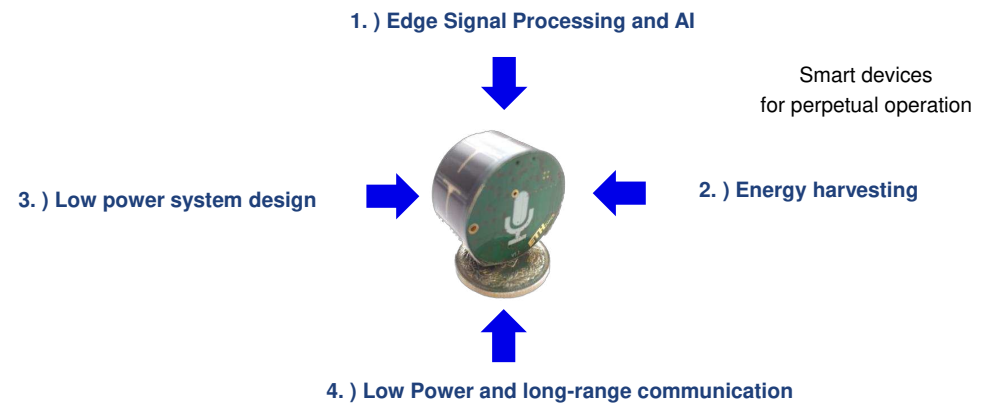
How we do learning at PBL?



1 - 61

29.05.2020

Next Generation of IoT devices: **Always-on Smart Sensors** PBL research activities in all the 4 areas.



You are kindly invited for the PBL OPEN day... right now 😊



1 - 64

See you later or next week!



9/26/2022

1 - 65

Embedded Systems

Lecture 2.

Hardware Software Architecture and Software Dev.

© Lothar Thiele
Computer Engineering and Networks Laboratory
Michele Magno
D-ITET center for project based learning

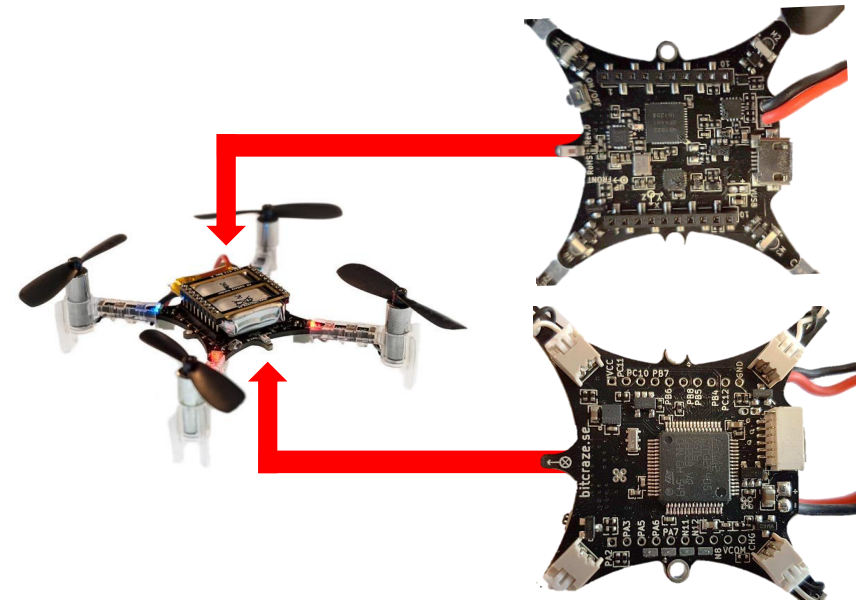
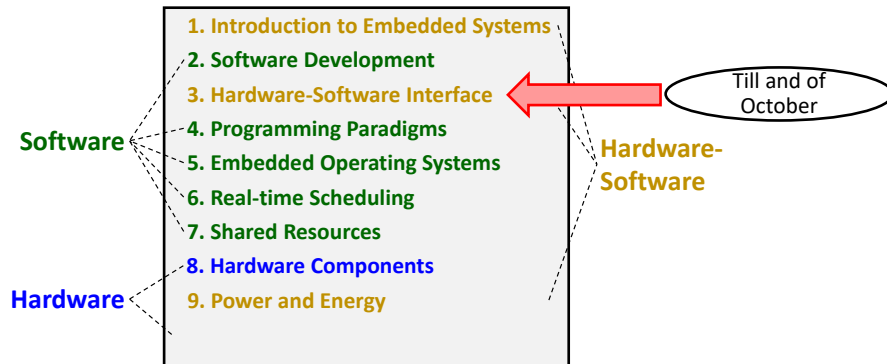
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Source the Slides or adaptation from:
Embedded Systems. P. Marwedel

Do you Remember ?

3 - 2

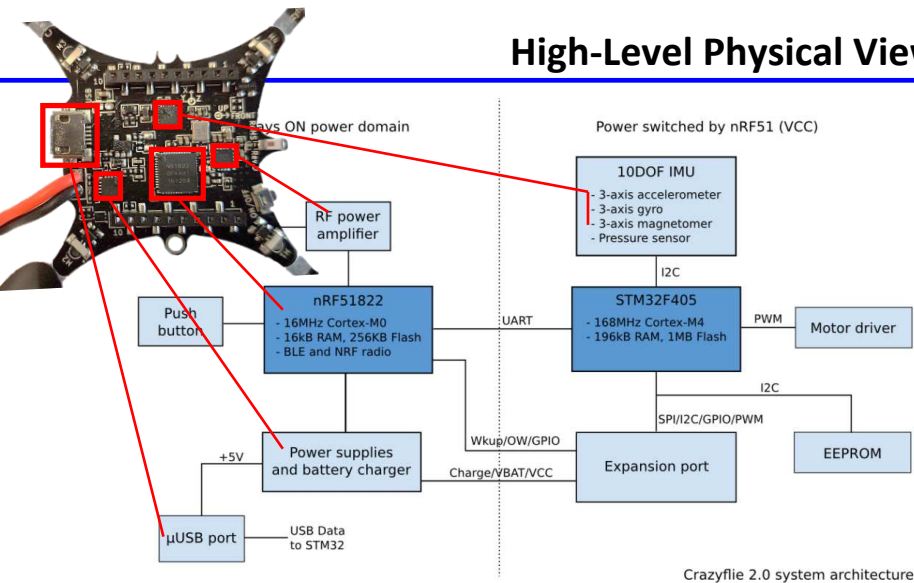
Where we are ...



3 - 3

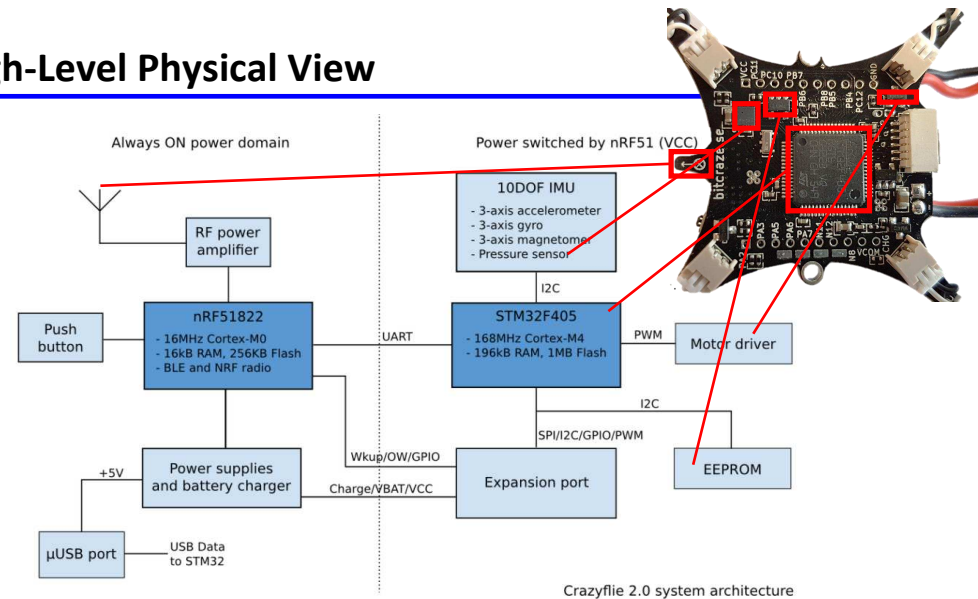
3 - 4

High-Level Physical View



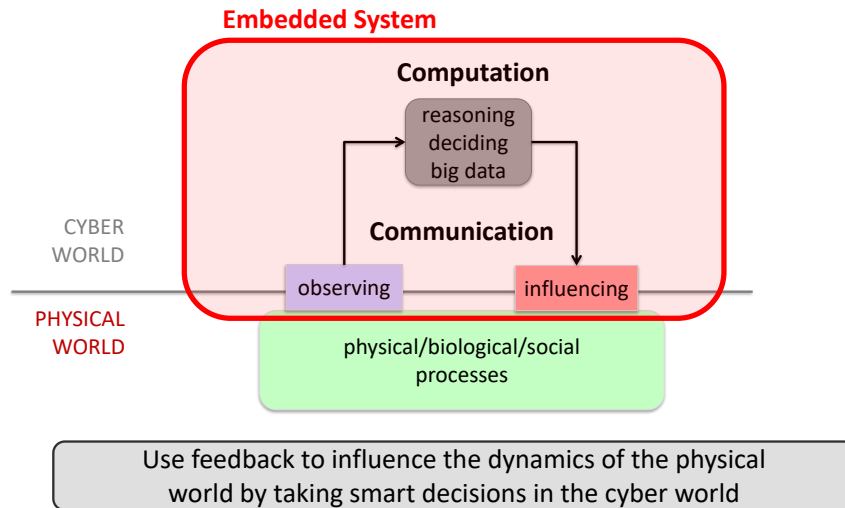
3 - 5

High-Level Physical View



3 - 6

Remember? Embedded System Overview

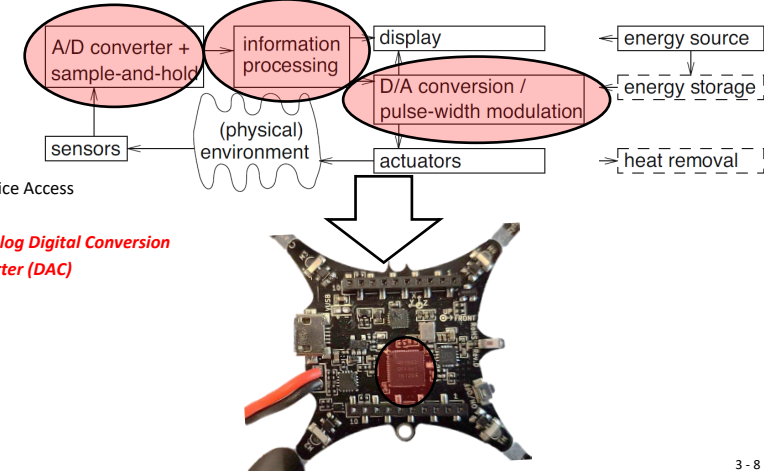


3 - 7

What you will learn In next 4 weeks.

Hardware-Software Architecture and Interfaces in Embedded Systems

- **Processing Unit**
- **Storage**
 - SRAM / DRAM / Flash
 - Memory Map
- **Input and Output**
 - UART Protocol
 - Memory Mapped Device Access
 - SPI Protocol
 - **Sensors Interface Analog Digital Conversion**
 - **Digital Analog Converter (DAC)**
- **Interrupts**
- **Clocks and Timers**
 - Clocks
 - Watchdog Timer
 - System Tick
 - Timer and PWM



3 - 8

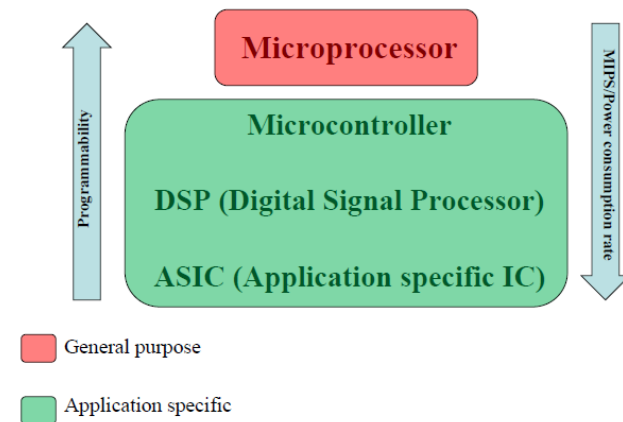
What type of Processing unit?

- **Application-specific integrated circuits (ASICs)**
 - Energy efficient
 - Low flexibility, designed for a specific task-application high cost to market
- **Digital Signal Processor**
 - Specialized Microprocessor (i.e. to audio streams, video stream etc.)
 - Cost-energy effective
- **Microcontroller**
 - Low cost and fully integrated solutions including control
 - Toward general purpose but still general purpose
- **Microprocessors**
 - General purpose (i.e. Intel Processors)
 - High performance
- **SoC/GPU/Others**
 - Can include all the above architecture with also GPUs.
- **FPGA**
 - Field Programmable Gate Array



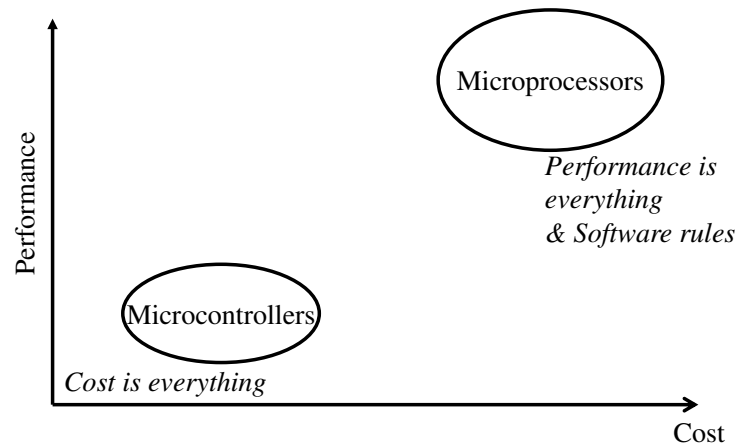
3 - 9

Digital Electronic Integrated circuit



3 - 10

The Processor Design Space



11
3 - 11

Microcontrollers

3 - 12

What is a microcontroller ?

A **Microcontroller** is a small CPU with many support devices built into the chip

- Self Contained (CPU, Memory, I/O)
- Application or Task Specific (Not a general-purpose computer)
- Appropriately scaled for the job
- Small power consumption
- Low costs (\$0.50 to \$5.00.)

Market overview

Microchip Technology™ Delivers 10 Billionth PIC® Microcontroller to Samsung Electronics Co.

2011, Sept 15th

*approximately 10 months
after delivering its
9 billionth*

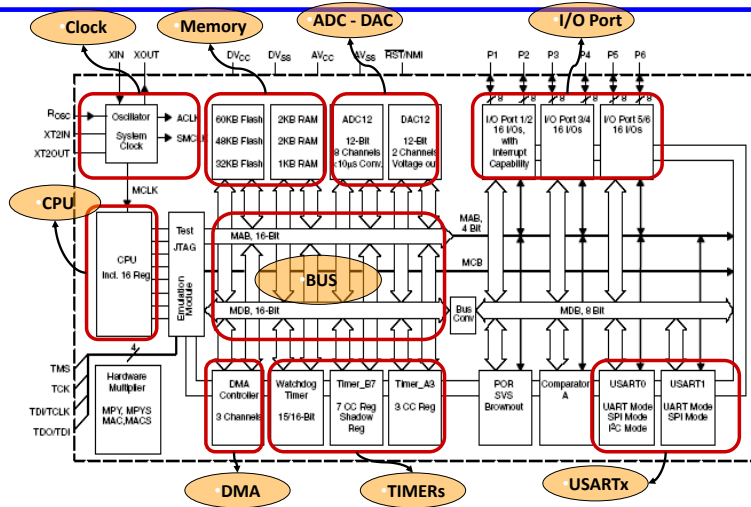
This accounts only the microprocessors of a single manufacturer



3 - 13

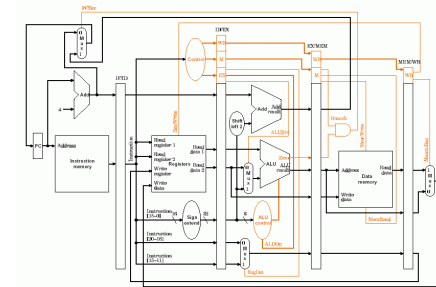
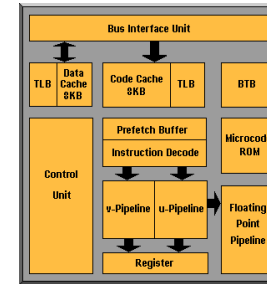
3 - 14

Example of MCU Architecture



3 - 15

CPU – Central Processing Unit



Characteristic

Instruction set

- CISC Complex Instruction Set Computing (Intel x86 family; Motorola 680x0 Family)
- RISC Reduced Instruction Set computer (AIM Power PC, ARM family, ATMEAL AVR Family)

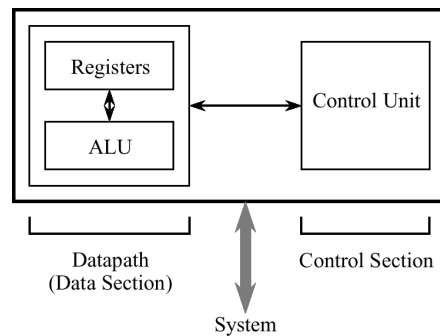
Architecture (respect integer operand maximum dimension)

- 8 bit (Intel 8051, Motorola 6800, ATMEAL AVR)
- 16 bit (Intel 8088, Motorola 68000, TI MSP430)
- 32 bit (x86 family, Motorola 680x0 Family, Power PC, ARM)
- 64 bit (x86-64 family, Power PC)

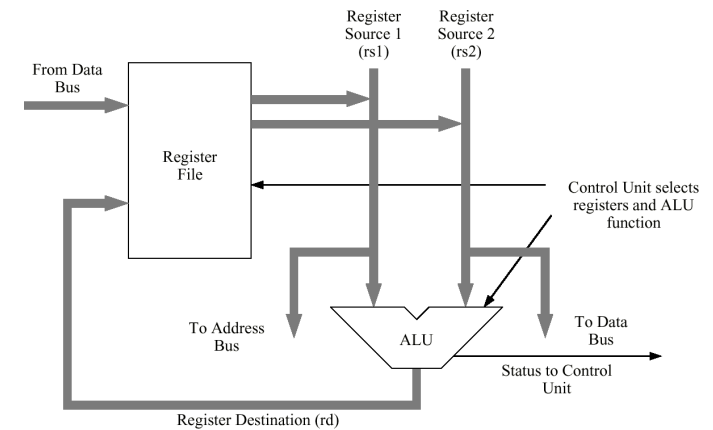
3 - 16

Abstract View of a CPU

The CPU consists of a data section containing registers and an ALU (Arithmetic and Logic Unit), and a control section, which **interprets instructions** and effects register transfers. The data section is also known as the datapath.



An Example Datapath



- The datapath usually consists of a collection of registers known as the register file and the arithmetic and logic unit (ALU).

3 - 17

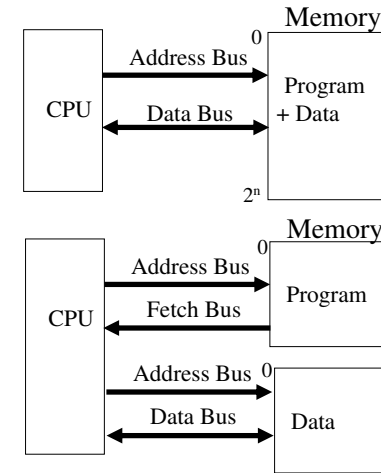
3 - 18

Memory System Architectures

- Two types of information are found in a typical program code:
 - Instruction codes for execution
 - Data that is used by the instruction codes
- The **instruction cycle** (also known as the **fetch–decode–execute cycle**)
- Two classes of memory systems designed to store the information:
 - von Neumann architecture
 - Harvard architecture

3 - 19

CPU Architectures



Von Neumann Architecture

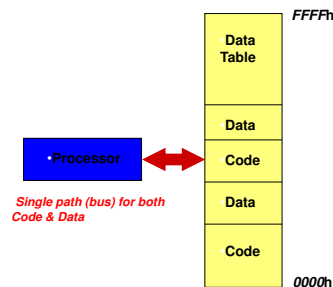
Harvard Architecture

3 - 20

von Neumann Architecture

The von Neumann architecture utilizes only one memory bus for both instruction fetching and data access

- simplifies the hardware and glue logic design
- code and data located in the same address space



3 - 21

von Neumann Features

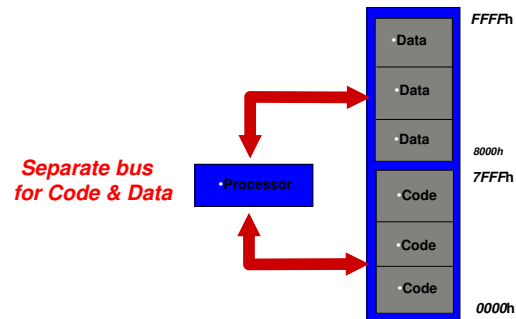
- Single memory interface bus
 - simplifies the hardware and glue logic design
- More efficient use of memory
 - code and data can reside in the same physical memory chip
- More flexible programming style
 - e.g., can include self-modified code
- But data may overwrite code (e.g. due to program bug)
 - need memory protection (e.g. hardware-based MPU)
- Bottleneck in code and data transfer
 - only one memory bus for both data and code fetching

3 - 22

Harvard Architecture

The Harvard architecture utilizes separate instruction bus and data bus

- code and data may still share the same memory space



Harvard Features

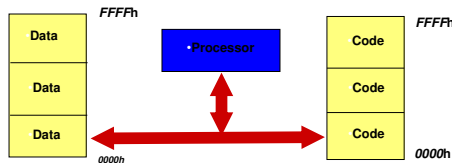
- Separate instruction and data buses
 - allow code and data access at the same time which gives improved performance
 - provide better support for instruction pipeline operations and shorter instruction execution time
 - allow different sizes of data and instructions to be used which results in more flexibility
 - do not incur any code corruption by data which makes the operations more robust
- But more sophisticated hardware glue logic is required to support multiple interface buses

3 - 23

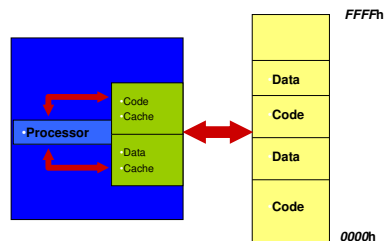
3 - 24

Architecture Variations

Independent data and code memory but with one shared bus (e.g. 8051)



Two separate internal bus for code & data (e.g. ARM9)



von Neumann vs. Harvard

- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
 - greater memory bandwidth;
 - more predictable bandwidth.
- Harvard cannot deal with self-modified code
 - Code that alters its own instructions while it is executing (We don't see it in this course ☺)

3 - 25

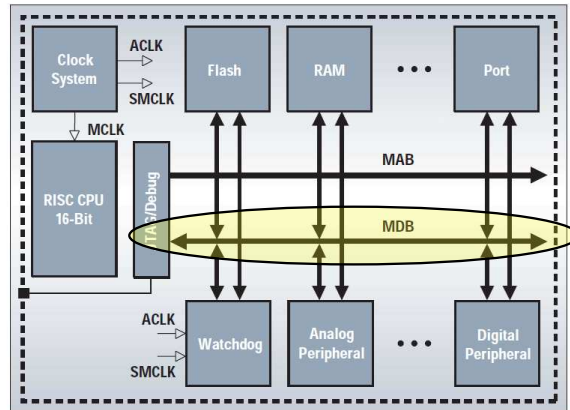
3 - 26

von Neumann Architecture an example

MSP430
Texas Instruments
von-Neumann architecture

All program, data memory and peripherals share a common bus structure.

Consistent CPU instructions and addressing modes are used.



3 - 27

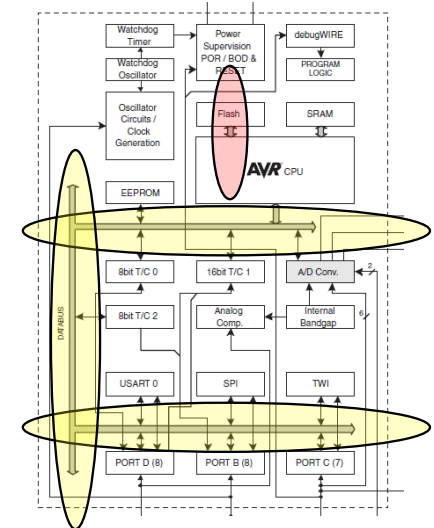
Processor Size

- Processor size is described in terms of 'bits' (e.g. an 8-bit or 32-bit processor)
 - corresponds to the data size that can be manipulated at a time by the processor
 - typically reflected in the size of the processor (internal) data path and register bank
- An 8-bit processor can only manipulate one byte of data at a time, while a 32-bit processor can handle one 32-bit double word sized data at a time even though the data content may only be of single byte size

3 - 29

Harvard Architecture an example

- AVR (atmel)
- Harvard architecture
- Separate memories and buses for program and data.
- Instructions in the program memory are executed with a single level pipelining.



3 - 28

Performance Metrics

- How we compare and classify CPU and microcontrollers?
 - Performance Metrics NOT easy to define and mostly application depended.

Electrical:

- Power Consumptions
- Voltage Supply
- Noise Immunity
- Sensitivity

**Goal: best tradeoff
power consumptions Vs
performances**

Computation:

- Clock Speed
- MIPS (instructions per sec)**
- Latency**
 - Lateness of the response
 - Lag between the begin and the end of the computation
- Throughput**
 - Tasks per second
 - Byte per second

3 - 30

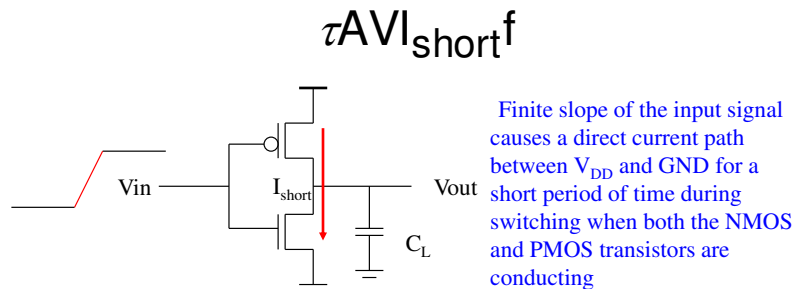
Power as a Design Constraint

- Why worry about power?
 - Battery life in portable and mobile platforms
 - Power consumption in desktops, server farms
 - Cooling costs, packaging costs, reliability, timing
 - Power density: 30 W/cm² in Alpha 21364 (3x of typical hot plate)

Where does power go in CMOS?

$$P = \underbrace{ACV^2f}_{\text{Dynamic power consumption}} + \underbrace{\tau AVI_{\text{short}}f}_{\text{Power due to short-circuit current during transition}} + \underbrace{VI_{\text{leak}}}_{\text{Power due to leakage current}}$$

Short-circuit Power Consumption



- Reducing Short-circuit
- Lower the supply voltage V
- Slope engineering – match the rise/fall time of the input and output signals

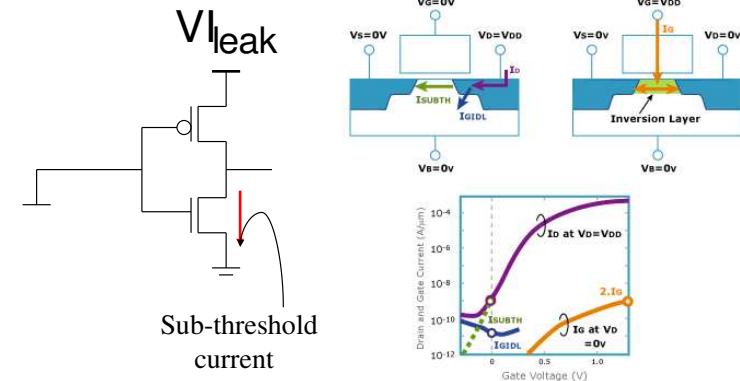
Dynamic Power Consumption

$$ACV^2f$$

C – Total capacitance seen by the gate's outputs
 Function of wire lengths, transistor sizes, ...
 V – Supply voltage
 Trend: has been dropping with each successive fab
 A – Activity of gates
 How often on average do wires switch?
 f – clock frequency
 Trend: increasing ...

- Reducing Dynamic Power
- Reducing V has quadratic effect; Limits?
- Lower C - shrink structures, shorten wires
- Reduce switching activity - Turn off unused parts or use design techniques to minimize number of transitions

Leakage Power



Sub-threshold current grows exponentially with increases in temperature and decreases in V_t

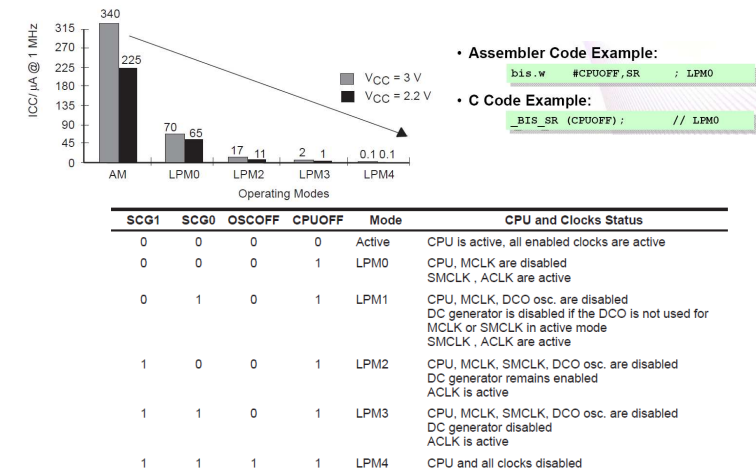
How can we reduce power consumption?

- Dynamic power consumption
 - Reduce the rate of charge/discharge of highly loaded nodes
 - Reduce spurious switching (glitches)
 - Reduce switching in idle states (clock gating)
 - Decrease frequency
 - Decrease voltage (and frequency)
- Static power Consumption
 - Smaller area (!)
 - Reduce device leakage through power gating
 - Reduce device leakage through body biasing
 - Use higher-threshold transistors when possible



Power performance tradeoffs!

Operating Modes



3 - 35

3 - 36

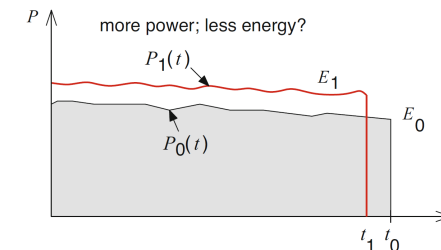
Why *Ultra-low Power* Is so Important

- Longer battery life
- Smaller products
- Simpler power supplies
- Less EMI simplifies PCB
- Permanent** battery
- Reduced liability



What about the energy?

- Energy is $E = \int P dt$
 - Where P is the power consumption in a dt Time.
- Example we have 2 tasks with different power P_0 and time t_0
 - Energy for a specific task 0 is $E_0 = \int_0^{t_0} P_0(t) dt$
 - Energy for a specific task 1 is $E_1 = \int_0^{t_1} P_1(t) dt$



3 - 37

3 - 38

Parallel Architectures

- Assuming we have a CPU working with a task that takes t time to execute at the frequency f
- Moving toward executing β operations in parallel
 - Extend the time for each operation by a factor of β
 - We can reduce the original frequency of fact β to execute at the same time $\implies f' = \frac{f}{\beta} \implies$ **Lower power and eventually lower voltage**
 - Or we can run the task with the some frequency but for less time $t' = \frac{t}{\beta} \xRightarrow{\text{Saving Energy}} E' = \frac{E}{\beta}$

Table 2 Basic parameters of CPUs and GPUs

Processor Type	CPU		GPU	
Version	Intel-E5-2650 v1	Intel-E5-2695 v3	Nvidia Tesla K-40	Nvidia Tesla V-100
Core number	8 cores, 16 threads	18 cores, 36 threads	2880	5120*
Clock rate	2.0GHz	2.3GHz	0.875GHz	1.5GHz
Memory	32GB	32GB	12GB	16GB*
Compute ability	---	---	3.5	7.0
Year of production	2011	2016	2013	2017

Liu, J., Hu, F. Q., & Li, X. (2018). Performance Comparison on Parallel CPU and GPU Algorithms for Unified Gas-Kinetic Scheme. *arXiv preprint arXiv:1810.08137*.

3 - 39

ARM- Processors

The ARM Processor Architecture

- ARM stands for “Advanced RISC Machine”
- based on Reduced Instruction Set Computer (RISC) architecture
 - trading simpler hardware circuitry with software complexity (and size)
 - but latest ARM processors utilize more than 100 instructions

A Bit of ARM History

- Originally conceived to be a processor for the desktop system (Acorn®)
- now entrenched in embedded markets
- First well-known product:
 - Apple’s Newton™ PDA (1993) based on an ARM6™ core
- Significant breakthrough:
 - Apple’s iPod® (2001) based on an ARM7™ core



Michele Magno 10/10/2022 41
3 - 41

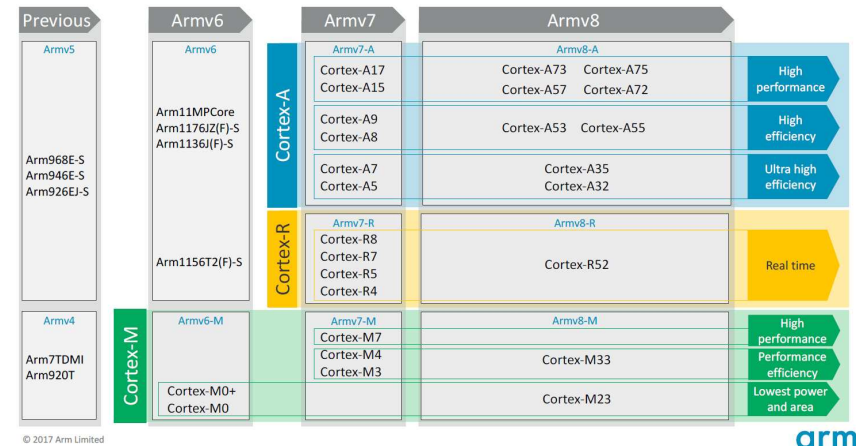
Michele Magno 10/10/2022 40
3 - 40

Michele Magno 10/10/2022 42
3 - 42

The Microprocessor Market

- In 2007,
 - 13 billion microprocessors were shipped
 - 3 billion were embedded processors based on the ARM architecture
 - 150 million were for the PC, notebook, and workstation
- By February 2008,
 - 10 billion ARM-based processors have been produced
- Only in 2017
 - 20 Billions ARM-Based processors have been produced!

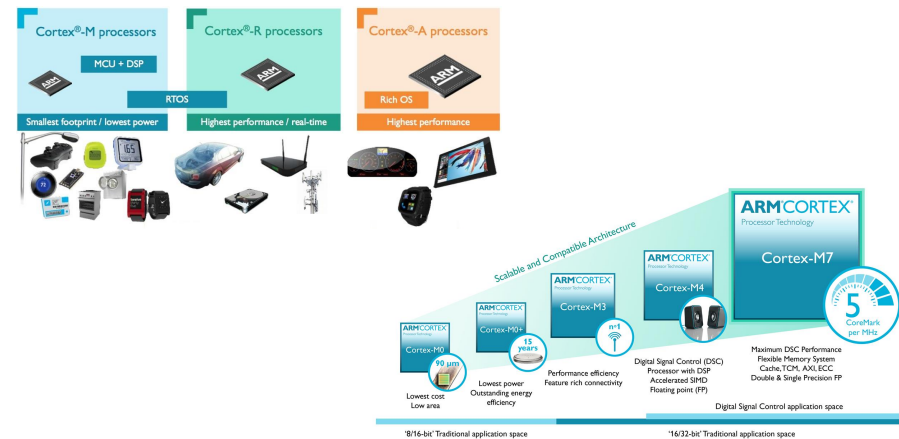
ARM Processors Families



ARM Processors Architectures (2)

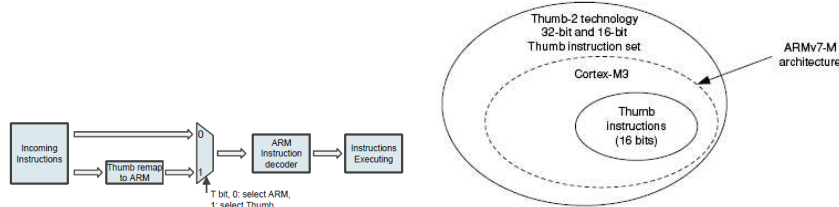
- Key attributes: Implementation size, performance, and very low power.
- Architectures types:
 - **ARMv4T** architecture introduced the 16-bit Thumb® instruction set alongside the 32-bit ARM instruction set.
 - **ARMv5TEJ** architecture introduced arithmetic support for digital signal processing (DSP) algorithms.
 - **ARMv6** architecture introduced an array of new features including the **Single Instruction Multiple Data (SIMD) operations**.
 - **ARMv7** architecture implements Thumb-2 technology.
 - **Cortex-A** implements a virtual memory system architecture based on an MMU, an optional NEON processing unit for multimedia applications and advanced hardware Floating Point.
 - **Cortex-R** – implements a protected memory system architecture based on an MPU (memory protection unit).
 - **Cortex-M** – Microcontroller profile designed for fast interrupt processing.
 - **ARMv8** implementing 64bit instruction set

ARM Processors Families



Thumb-2 Instruction Set

- Mixes 16 and 32 bits instructions
 - Enhancements: eg division, bit-field operators, wrt traditional ARMv4T
 - No need to mode switch, can be mixed freely
- **Not** backwards binary compatible
 - But porting is «easy»



3 - 47

Cortex-M4 Processor Overview

- Cortex-M4 Processor
 - Introduced in 2010
 - Designed with a large variety of highly efficient signal processing features
 - Features extended single-cycle multiply accumulate instructions, optimized SIMD arithmetic, saturating arithmetic and an optional Floating Point Unit (**When is an Cortex-M4F!**).
- High Performance Efficiency
 - 1.25 DMIPS/MHz (Dhrystone Million Instructions Per Second / MHz) at the order of μ Watts / MHz
- Low Power Consumption
 - Longer battery life – especially critical in mobile products
- Enhanced Determinism
 - The critical tasks and interrupt routines can be served quickly in a known number of cycles

3 - 48

Cortex-M4 Processor Features

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

3 - 49

Cortex-M4 Processor Features

- Supports Sleep Modes
 - Up to 240 Wake-up Interrupts
 - Integrated WFI (Wait For Interrupt) and WFE (Wait For Event) Instructions and Sleep On Exit capability (to be covered in more detail later)
 - Sleep & Deep Sleep Signals
 - Optional Retention Mode with ARM Power Management Kit
- **Enhanced Instructions**
 - Hardware Divide (2-12 Cycles)
 - **Single-Cycle 16, 32-bit MAC (Multiply and Accumulator), Single-cycle dual 16-bit MAC**
 - **8, 16-bit SIMD arithmetic**
- Debug
 - Optional JTAG & Serial-Wire Debug (SWD) Ports
 - Up to 8 Breakpoints and 4 Watchpoints
- Memory Protection Unit (MPU)
 - Optional 8 region MPU with sub regions and background region

3 - 50

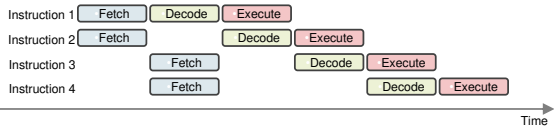
Cortex-M4 Processor Features

- Cortex-M4 processor is designed to meet the challenges of low dynamic power constraints while retaining light footprints
 - 180 nm ultra low power process –157 µW/MHz
 - 90 nm low power process – 33 µW/MHz
 - 40 nm G process – 8 µW/MHz

ARM Cortex-M4 Implementation Data			
Process	180ULL (7-track, typical 1.8v, 25C)	90LP (7-track, typical 1.2v, 25C)	40G (9-track, typical 0.9v, 25C)
Dynamic Power	157 µW/MHz	33 µW/MHz	8 µW/MHz
Floorplanned Area	0.56 mm²	0.17 mm²	0.04 mm²

Cortex-M4 Block Diagram

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

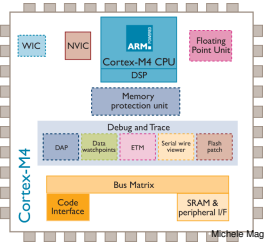


Embedded ARM Cortex Processors (M4)

Cortex M4

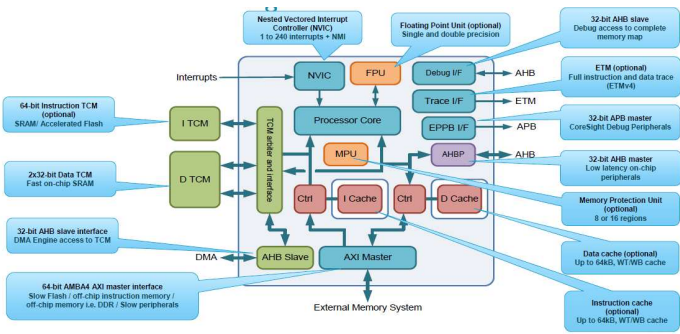
Embedded processor for DSP with FPU (M4F)

ISA Support	Thumb® / Thumb-2
DSP Extensions	Single cycle 16/32-bit MAC Single cycle dual 16-bit MAC 8/16-bit SIMD arithmetic Hardware Divide (2-12 Cycles)
Floating Point Unit	Single precision floating point unit IEEE 754 compliant
Pipeline	3-stage + branch speculation
Performance Efficiency	3.40 CoreMark/MHz**
Performance Efficiency	Without FPU: 1.25 / 1.52 / 1.91 DMIPS/MHz** With FPU: 1.27 / 1.55 / 1.95 DMIPS/MHz**
Memory Protection	Optional 8 region MPU with sub regions and background region
Interrupts	Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
Interrupt Priority Levels	8 to 256 priority levels
Wake-up Interrupt Controller	Up to 240 Wake-up Interrupts
Sleep Modes	Integrated WFI and WFE Instructions and Sleep On Exit capability. Sleep & Deep Sleep Signals. Optional Retention Mode with ARM Power Management Kit
Bit Manipulation	Integrated Instructions & Bit Banding
Debug	Optional JTAG & Serial Wire Debug Ports. Up to 8 breakpoints and 4 watchpoints.



Michele Magno

Cortex M7



2x Perf of M4

Future Processors: What are you expecting?

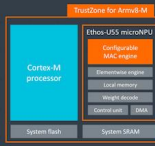
Cortex-M55: The Most AI-capable Cortex-M Processor

- First CPU based on Arm Helium technology
 - Energy-efficient and configurable with vector processing capabilities
 - Delivers up to 5x DSP performance and up to 15x ML performance*
- Versatile capability for both classical ML and NN inference
- Advanced memory interfaces for fast access to ML data and weights
- Arm TrustZone security, accelerating the route to PSA Certified

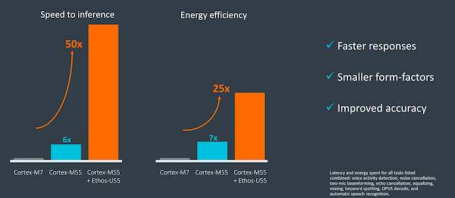


Ethos-U55: The First microNPU for Cortex-M

- Highest efficiency and small memory footprint
- 32, 64, 128, or 256 unit multiply-accumulate (MAC) engine
- Weight decoder and DMA for on-the-fly weight decompression
- Tooling available for offline optimization
- Works with a range of Cortex-M processors:
 - Cortex-M55
 - Cortex-M7
 - Cortex-M33
 - Cortex-M4

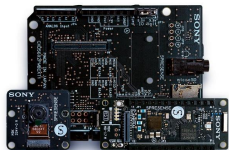


Example: Typical ML Workload for a Voice Assistant



Memory and Parallel are also important

Spresense based on 6 cores Cortex-M4F



Spresense main board	
Model name	CXD5602PWBMAIN1
Size	50.0 mm x 20.6 mm
CPU	ARM® Cortex®-M4F x 6 cores
Maximum clock frequency	156 MHz
SRAM	1.5 MB
Flash memory	8 MB
Digital input / output	GPIO, SPI, I2C, UART, I2S
Analog input	2 ch (0.7 V range)
GNSS	GPS(L1-C/A), QZSS(L1-C/A), GLONASS(L1), WAAS, QZSS(L1-S)
Camera input	Dedicated parallel interface

Ambiq Apollo 4 Cortex-M4F



Apollo 4	
Model name	Ambiq Apollo 4(blue)
CPU	ARM® Cortex®-M4F x 1 cores
Maximum clock frequency	192 MHz
SRAM	3.5 MB
Flash memory	Info ND
Digital input / output	GPIO, SPI, I2C, UART, I2S
Analog input	Several option
Current Consumption	3 µA/MHz
Wireless Interface	BTLE

GreenWave GAP8/9 RISC-V5 Based PULP

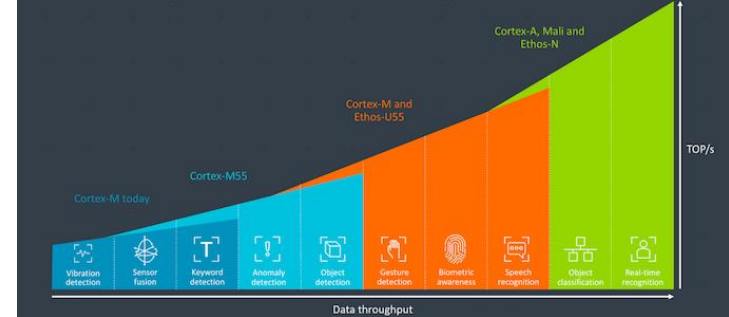


	GAP8	GAP9
Processing power	22.65 GOPS	150.8 GOPS
Power efficiency	4.24 mW/GOP	0.33 mW/GOP
Memory		
L1	80 kB	128 kB
RAM	512 kB	1.5 MB
Non Volatile	None	2 MB
External	QSPI/ HyperBus	2x QSPI/OCTO-SPI/HyperBus/SDIO
MAX Frequency		
FC*	250	400
Cluster**	175	400
Fixed Point	8, 16, 32-bit	8, 16, 32, 64-bit***
Floating Point	None	16/16ah/32, 64-bit***
Sound Interface	2 Rx-Only I2S interfaces	3 master/slave SAI full duplex, I2S and TDM 4/8/16 ch capable
Camera Interface	8-bit CPI (Camera Parallel Interface)	8-bit CPI, 2-lane CSI-2

3 - 57

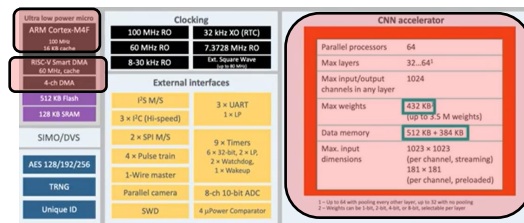
Where they will be applied?

Broadest Range of ML-optimized Processing Solutions

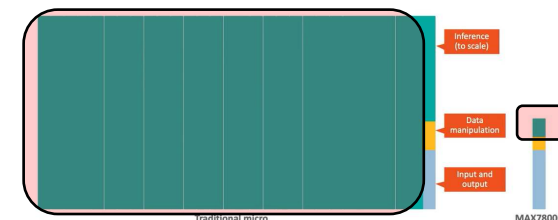


New generation of Embedded Processing (MAX78000 example)

- Multi cores and Convolutional Neural Networks accelerator



Making Inference Energy Practically Irrelevant



3 - 58

Embedded Systems

Lecture 3.

Hardware Software Architecture and Software Dev. Memory, Clock, GPIOs.

© Lothar Thiele

Computer Engineering and Networks Laboratory

Michele Magno

D-ITET center for project based learning

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Source the Slides or adaptation from:
Embedded Systems. P. Marwedel

Do you Remember ?

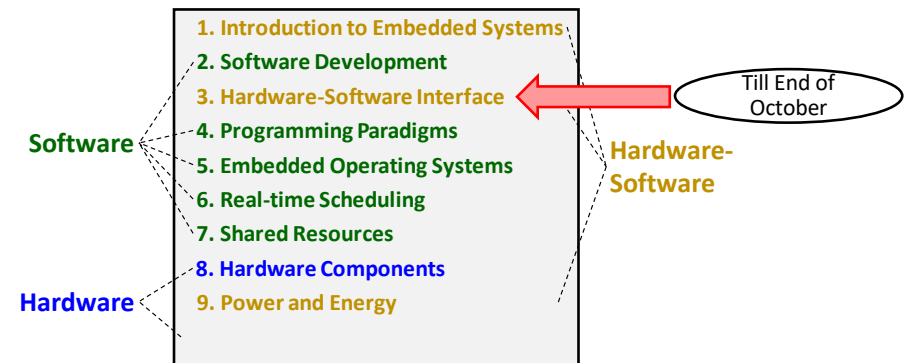
- Variety of possibility for embedded processing
- CPU has an instruction set.
 - Different architecture according to the bus
- According to the architecture we can run more fast or less fast a task
- **Energy is different than power Possible exam exercise!**
 - **Different architecture for different execution time**
 - **Frequency affect the time and the energy**
 - **Parallel processing can affect both**
- ARM processors
 - What we will use as commodity platform for the lab?

Exams information.

- 2h written exam
 - One in February and one in August.
- They will be 3 main parts.
 - Energy and latency linked with hardware-software interface
 - Real time and scheduling
 - Architecture Synthesis and systems overview.
 - Labs will be not explicitly part of the exam but can help to understand better the questions and perform.
- Exercise
 - By the 31.10 home work with some more exercise will be provided as home work.
 - By end of year more home work will be given for practice before the exam
 - Previous years exams are useful for training
 - Especially last year one.

3 - 2

Where we are ...



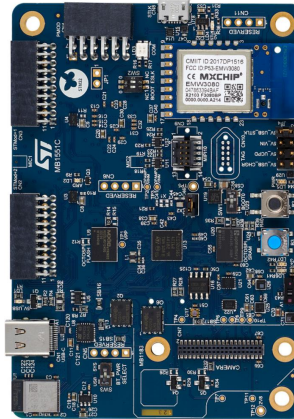
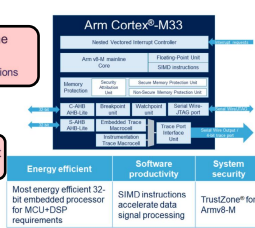
3 - 3

3 - 4

A recent architecture from ARM – CortexM33.

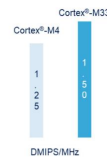
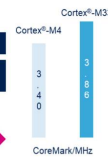
Cortex®-M33 processor overview

- Armv8-M architecture
- Harvard architecture, 3-stage pipeline
 - Floating-Point Unit
 - Single Instruction Multiple Data Instructions
- Memory Protection
 - Security Attribution Unit (SAU)
 - 2x Memory Protection Units (MPUs)
- Integrated Nested Vectored Interrupt Controller (NVIC)
- Invasive and non-invasive debug



- Seamless architecture across all applications

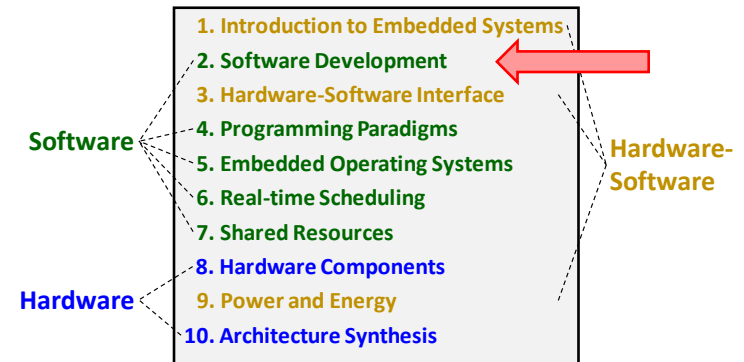
Cortex-M0 & M0+	Cortex-M3	Cortex-M4	Cortex-M33	Cortex-M7
Ultra low power	First Cortex®-M CPU		High performance	



Binary and tool compatible → Highest Performance

3 - 5

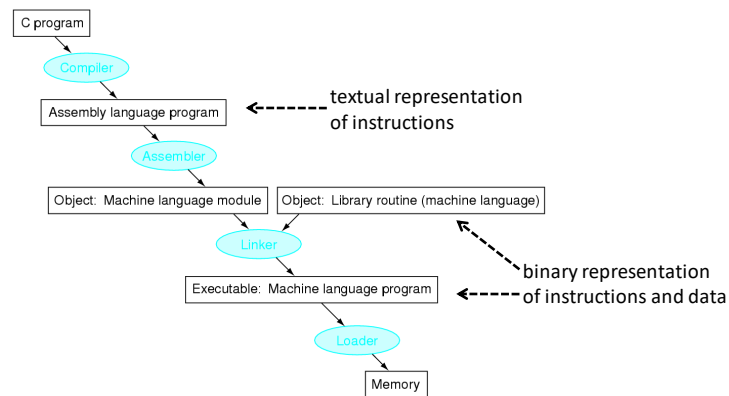
Where we are ...



3 - 6

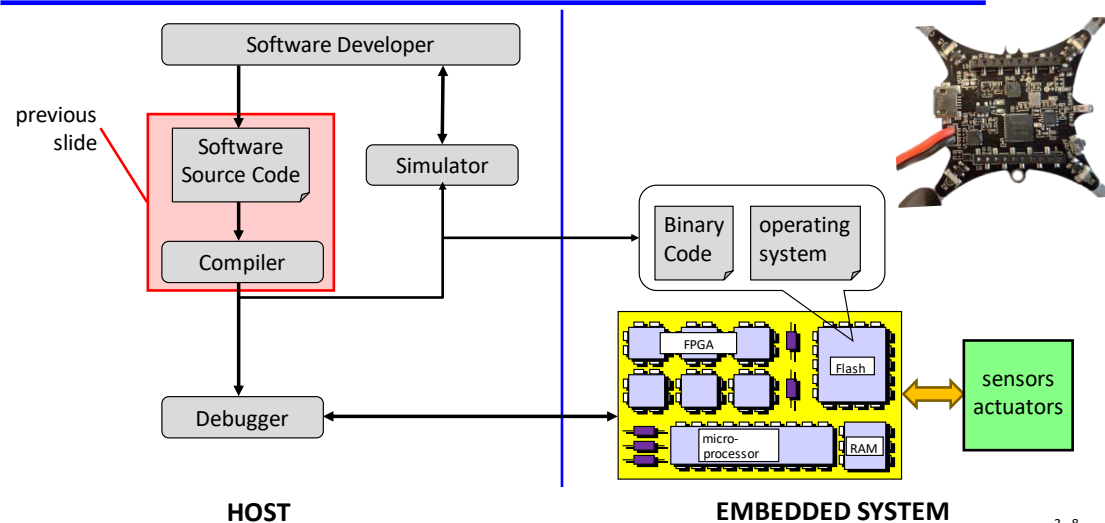
Remember: Computer Engineering I

Compilation of a C program to machine language program:



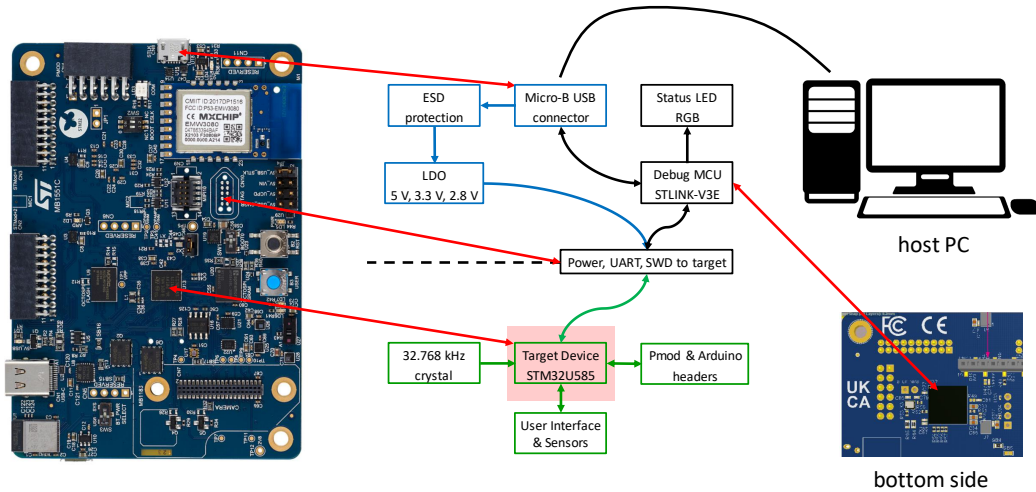
3 - 7

Embedded Software Development



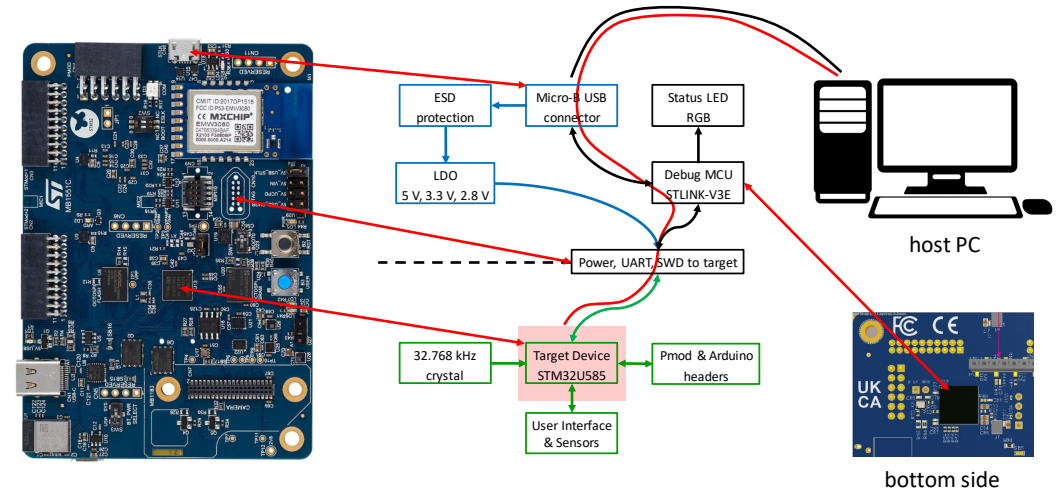
3 - 8

Software Development with MSP432 (ES-Lab)



3 - 9

Software Development with MSP432 (ES-Lab)

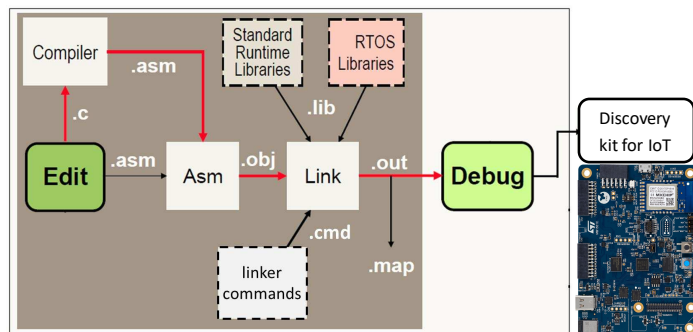


3 - 10

Software Development (ES-Lab)

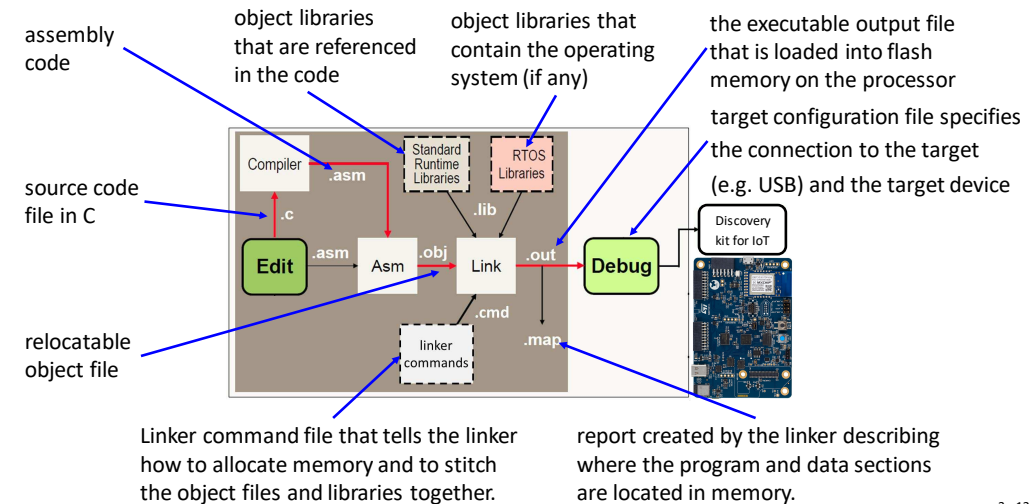
Software development is nowadays usually done with the support of an IDE (Integrated Debugger and Editor / Integrated Development Environment)

- edit and build the code
- debug and validate



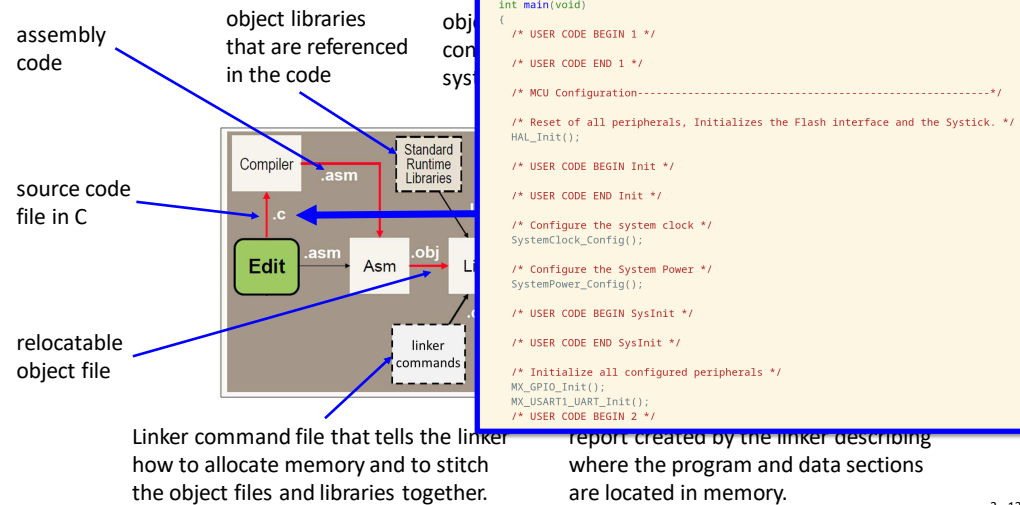
3 - 11

Software Development (ES-Lab)



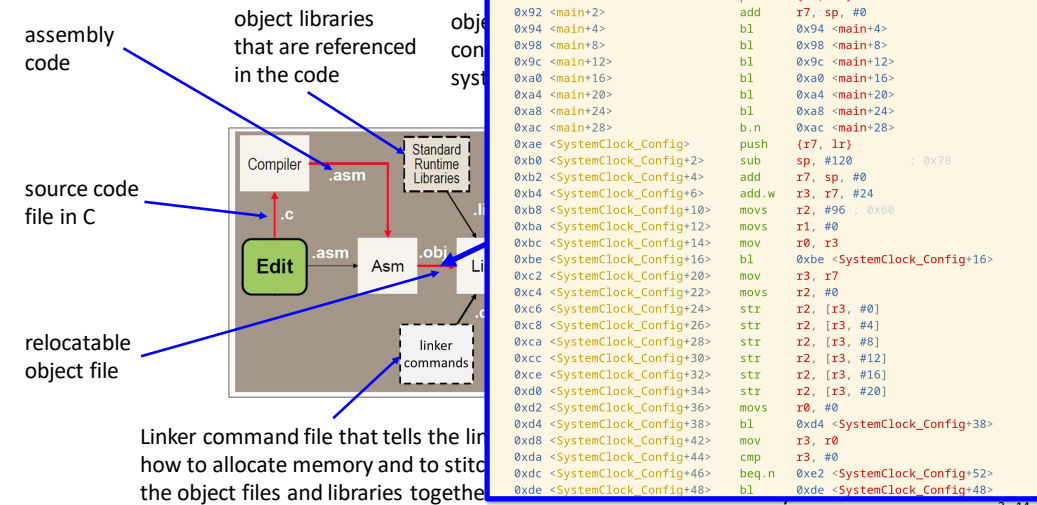
3 - 12

Software Development (ES-Lab)



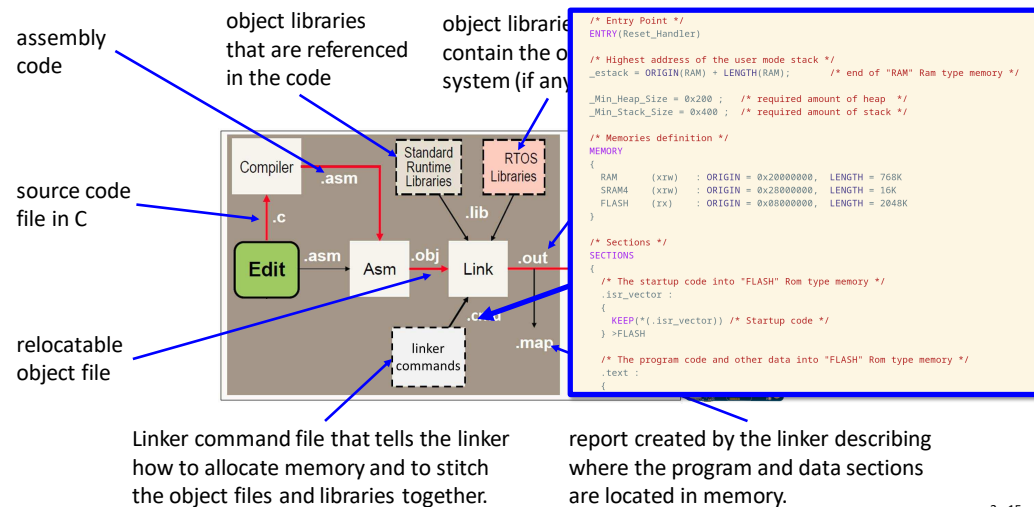
3 - 13

Software Development (ES-Lab)



3 - 14

Software Development (ES-Lab)

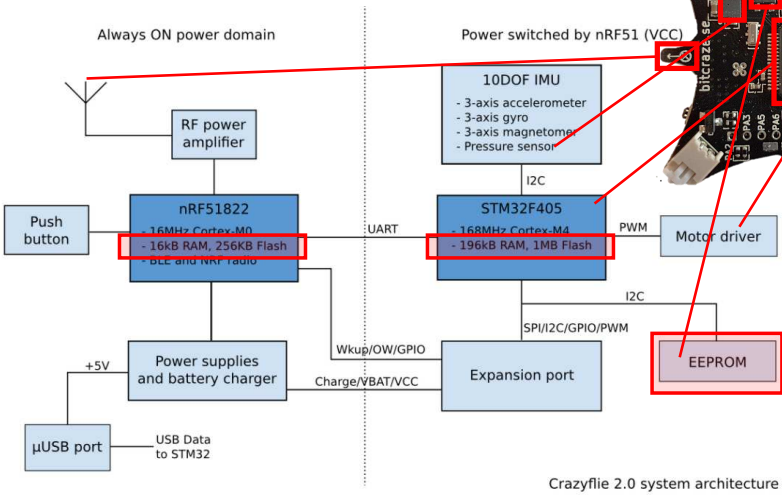


3 - 15

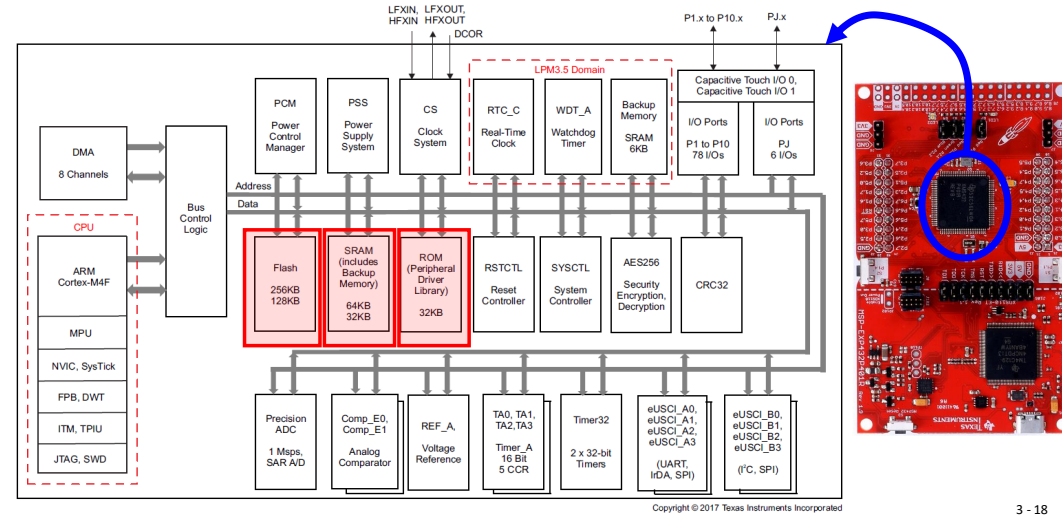
Storage

3 - 16

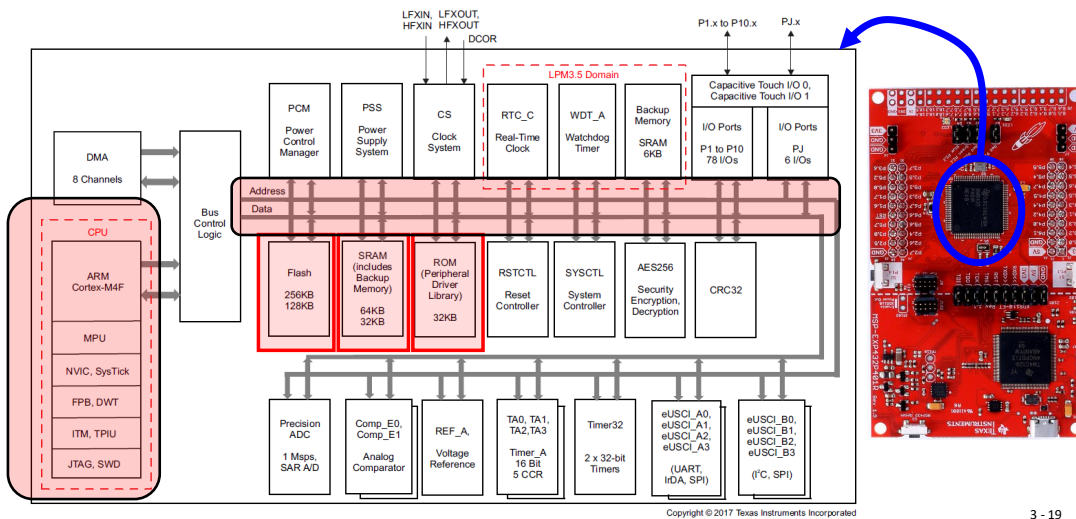
Remember ... ?



MSP432P401R (Last year lab - ES-Lab)



von Neumann or Harvard ? Why?



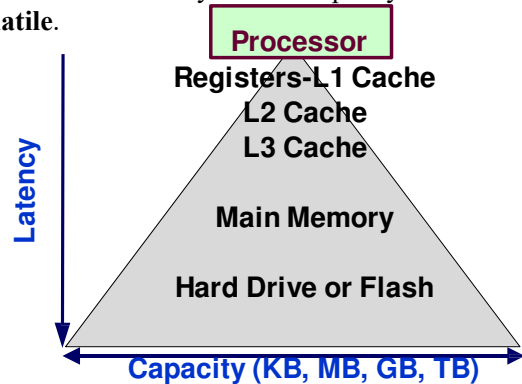
Storage

Registers/SRAM / DRAM / Flash

Memory Hierarchies and persistent and volatile memory

- Processor registers can be seen as the fastest level in the memory hierarchy, with only a limited capacity of at most a few hundred words.
- The **working memory** (or **main memory**) of computer systems implements the storage implied by processor memory addresses. Usually it has a capacity between a few megabytes and some gigabytes and is **volatile**.

- Hard drive or Flash
 - Persistent



3 - 21

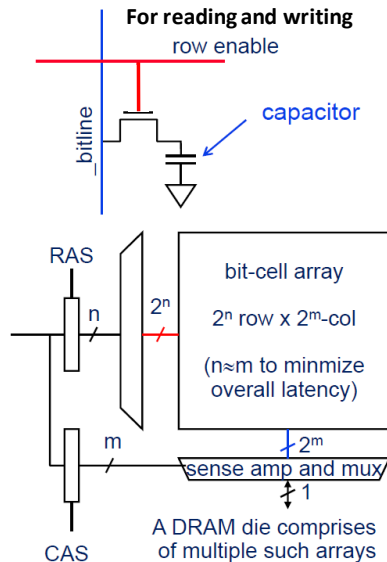
Dynamic Random Access (DRAM)

Single bit is stored as a charge in a capacitor

- Bit cell loses charge when read, bit cell drains over time
- Slower access than with SRAM due to small storage capacity in comparison to capacity of bit-line.
- Higher density than SRAM (1 vs. 6 transistors per bit)

DRAMs require *periodic refresh* of charge

- Performed by the memory controller
- Refresh interval is tens of ms
- DRAM is unavailable during refresh

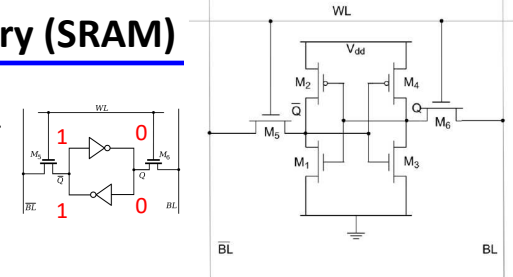


(RAS/CAS = row/column address select)

3 - 23

Static Random Access Memory (SRAM)

- Single bit is stored in a bi-stable circuit
- Static Random Access Memory is used for
 - caches
 - register file within the processor core
 - small but fast memories

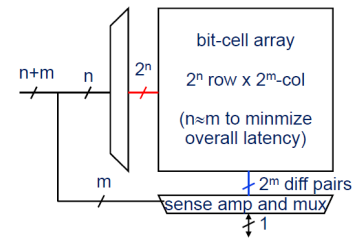


Read:

- Pre-charge all bit-lines to average voltage
- decode address ($n+m$ bits)
- select row of cells using 2^n single-bit word lines (WL)
- selected bit-cells drive all bit-lines BL (2^m pairs)
- sense difference between bit-line pairs and read out

Write:

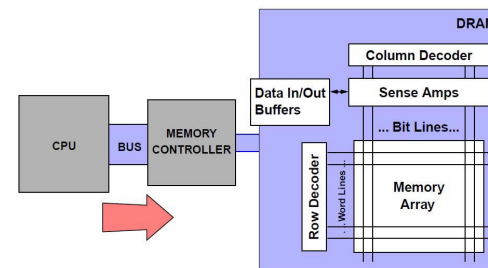
- select row and overwrite bit-lines using strong signals



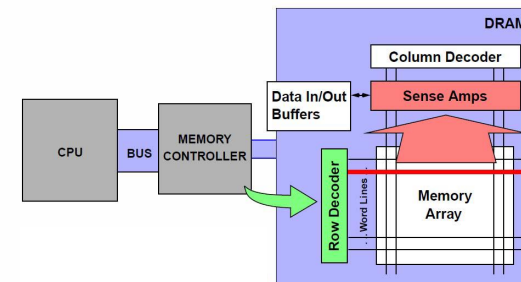
3 - 22

DRAM – Typical Access Process

1. Bus Transmission



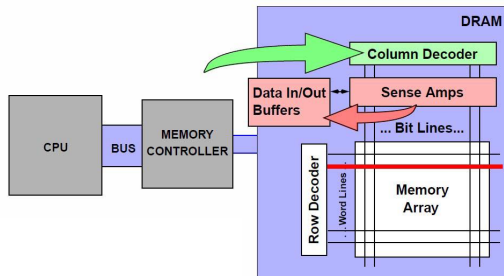
2. Precharge and Row Access



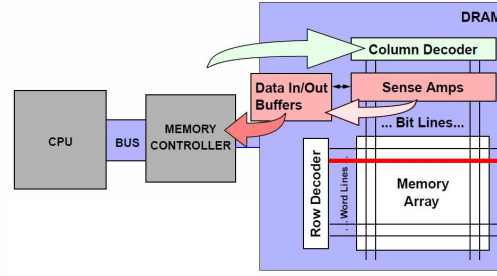
3 - 24

DRAM – Typical Access Process

3. Column Access



4. Data Transfer and Bus Transmission



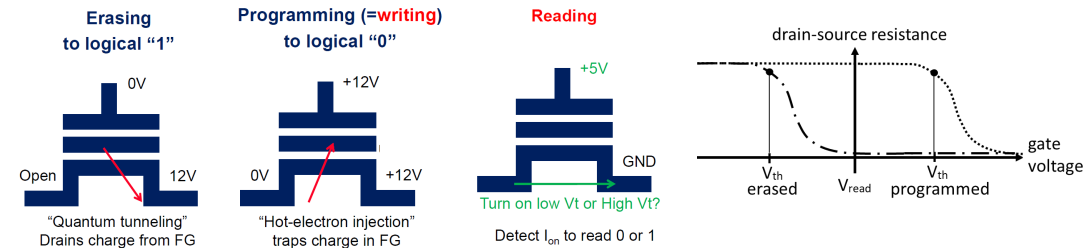
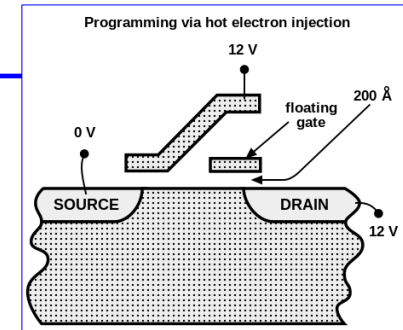
3 - 25

Flash Memory

Electrically modifiable, non-volatile storage

Principle of operation:

- Transistor with a second “floating” gate
- Floating gate can trap electrons
- This results in a detectable change in threshold voltage



3 - 26

NAND and NOR Flash Memory

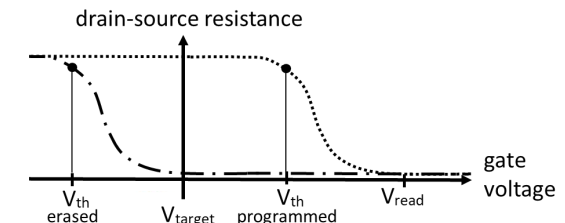
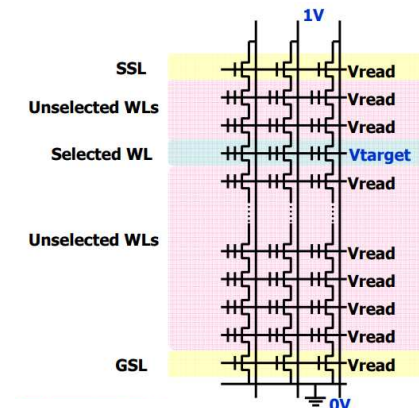
	NAND	NOR
Cell Array & Size		
Cross-section		
Features	Small Cell Size, High Density Low Power → Mass Storage	Fast random access → Code Storage

3 - 27

Example: Reading out NAND Flash

Selected word-line (WL) : Target voltage (V_{target})

Unselected word-lines : V_{read} is high enough to have a low resistance in all transistors in this row

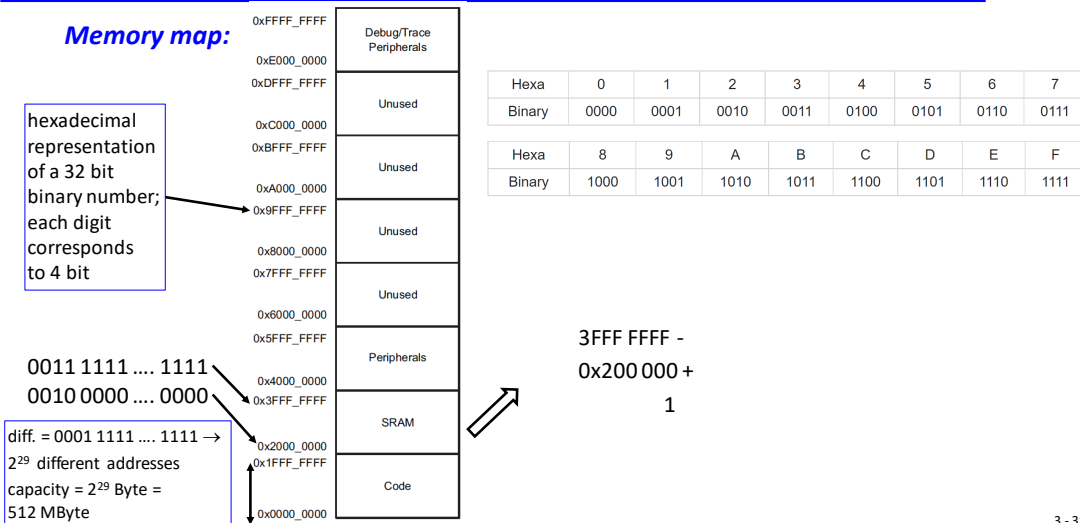


3 - 28

Storage Memory Map

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



3 - 29

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Available memory:

- The processor used in the lab (MSP432P401R) has built in 256kB flash memory, 64kB SRAM and 32kB ROM (Read Only Memory).

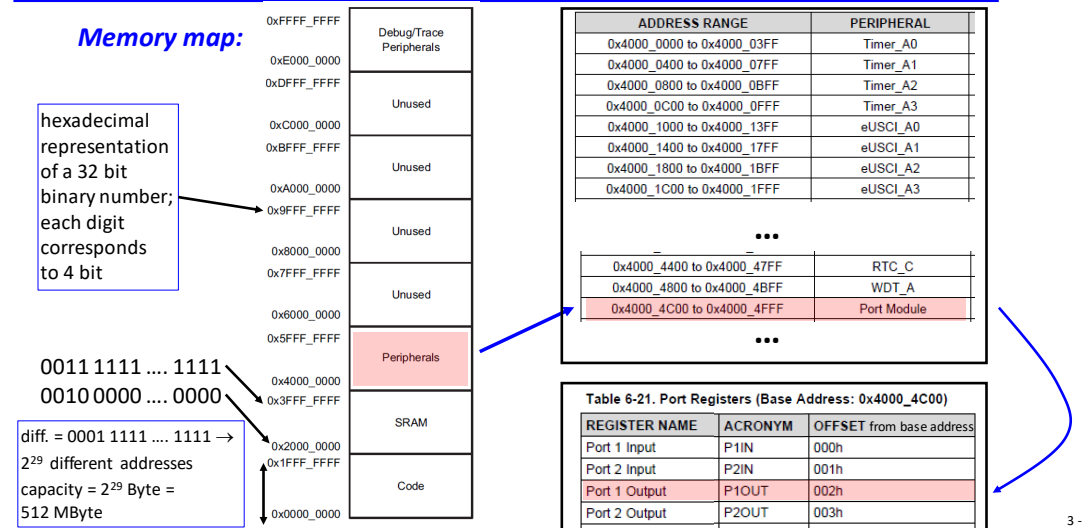
Address space:

- The processor uses 32 bit addresses as all the **ARM Cortex-M Microcontrollers**. Therefore, the addressable memory space is 4 GByte (= 2³² Byte) as each memory location corresponds to 1 Byte (for other ARM cortex-M4 usually is used 32 bits) .
- The address space is used to address the memories (reading and writing), to address the peripheral units, and to have access to debug and trace information (memory mapped microarchitecture).
- The address space is partitioned into zones, each one with a dedicated use. The following is a simplified description to introduce the basic concepts.

3 - 30

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:

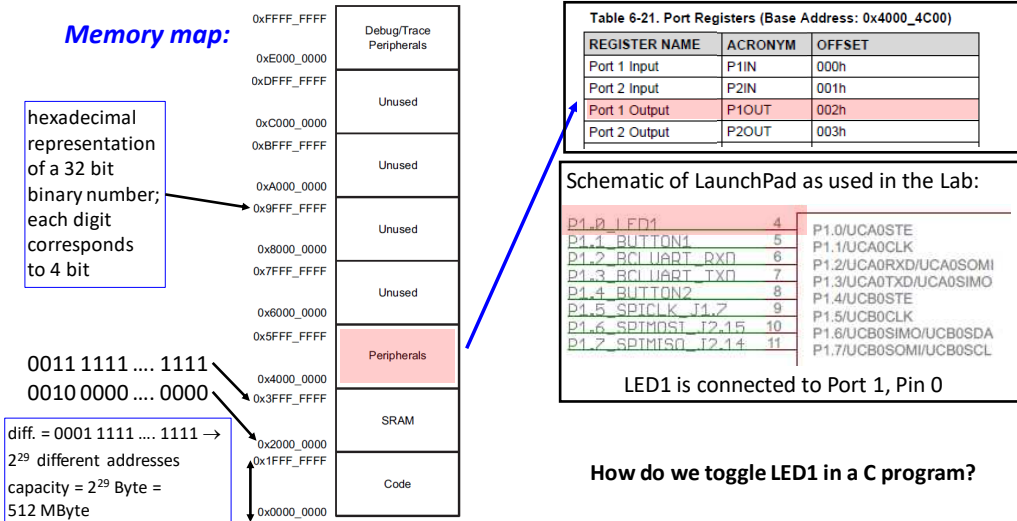


3 - 31

3 - 32

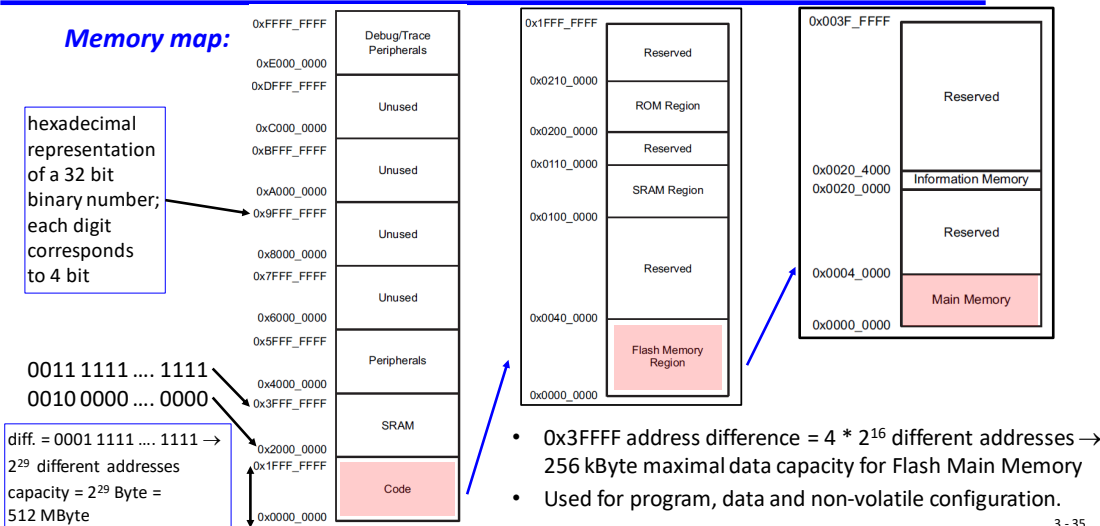
Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

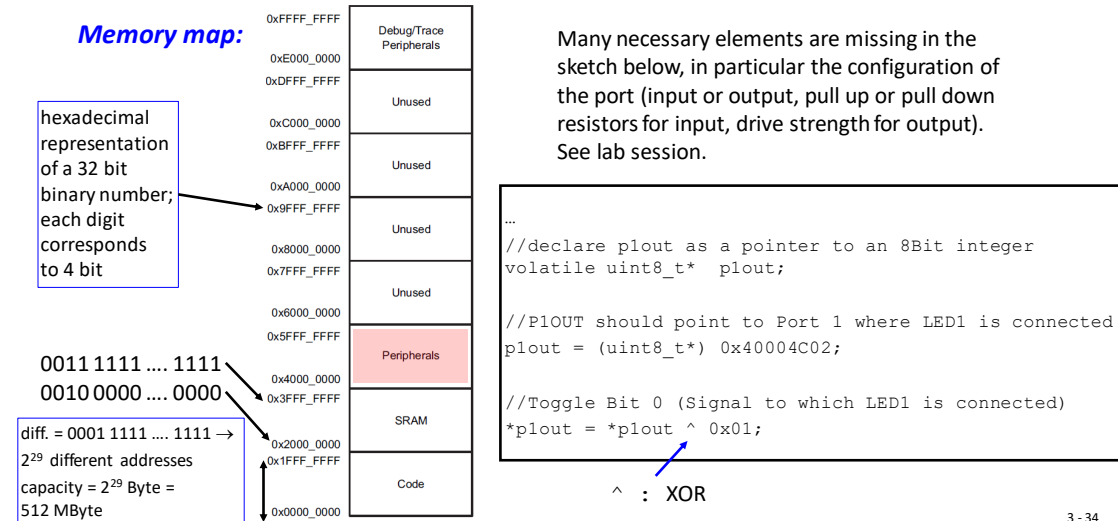
Memory map:



3 - 33

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

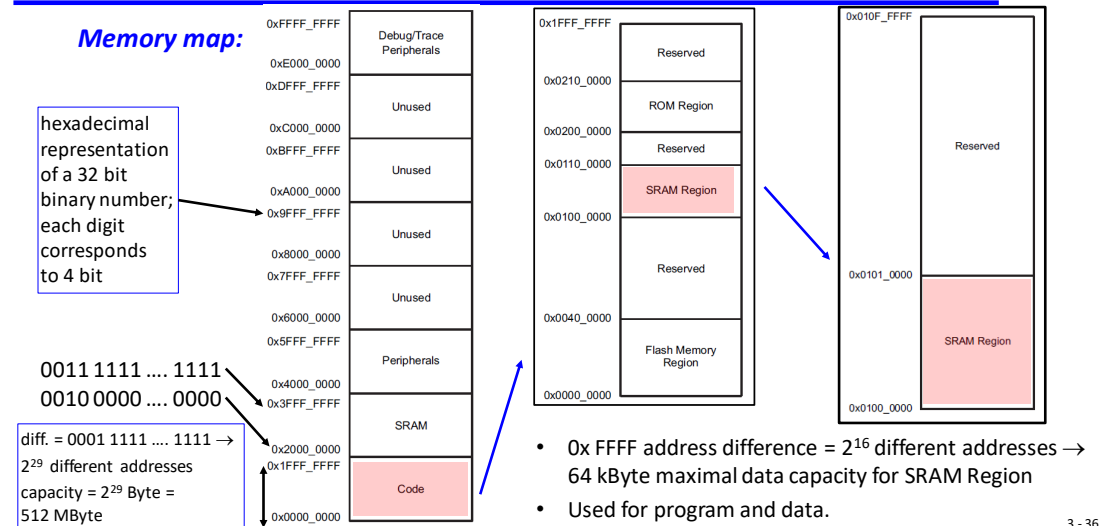
Memory map:



3 - 34

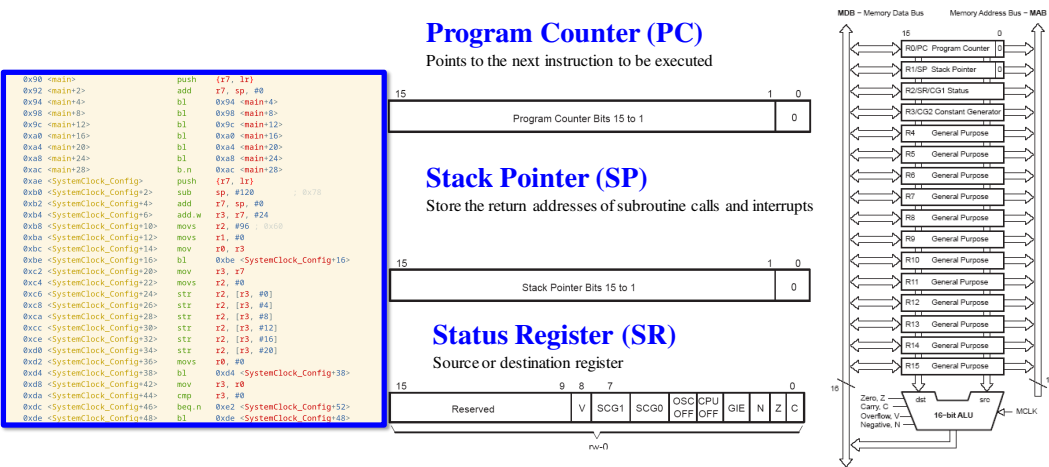
Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



3 - 36

CPU Internal Registers



3 - 37

Dedicated Registers

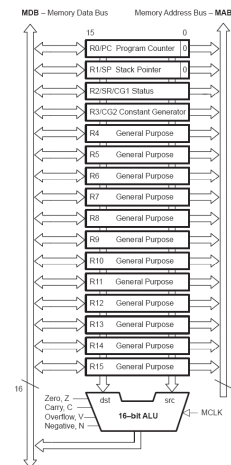
Dedicated Registers:

- Program Counter (PC):
 - Points to the next instruction to be read from memory and executed by the CPU.
- Stack Pointer (SP):
 - stack can be used by user to store data for later use (instructions: store by PUSH, retrieve by POP);
 - can be used by user or by compiler for subroutine parameters (PUSH, POP in calling routine; addressed via offset calculation on stack pointer (SP) in called subroutine);
 - used by subroutine calls to store the program counter value for return at subroutine's end (RET).
- Status Register (SR):
 - Stores status, control bits and system flags, updated automatically by the CPU.

3 - 38

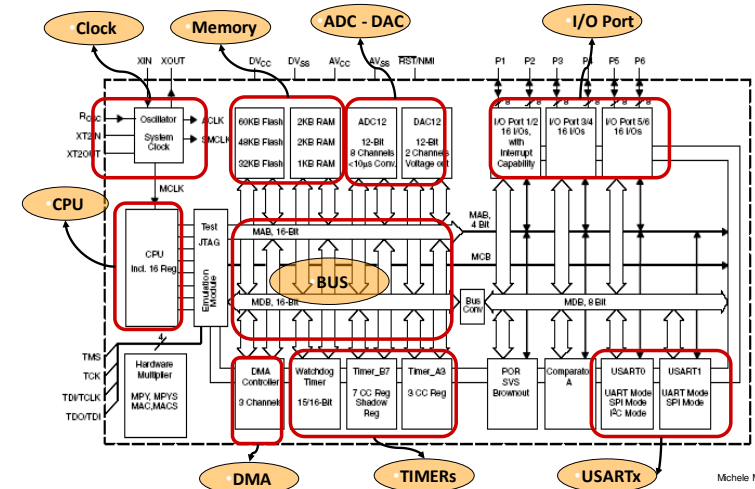
General Purpose Registers

- General-Purpose Registers:
 - The general-purpose registers are adequate to store data registers, address pointers, or index values and can be accessed with byte or word instructions.
 - Used to execute arithmetical operations or to read/write memory.



3 - 39

Example of MCU Architecture

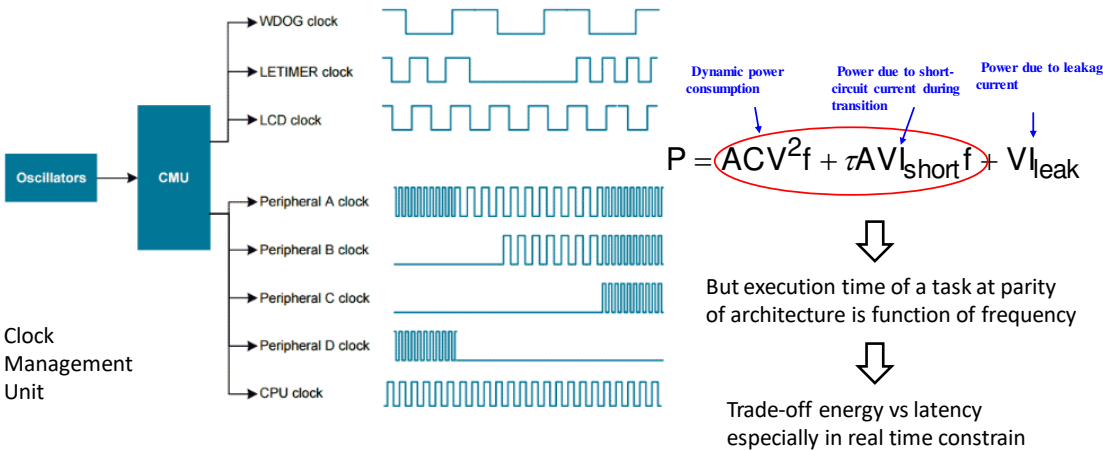


Clocks

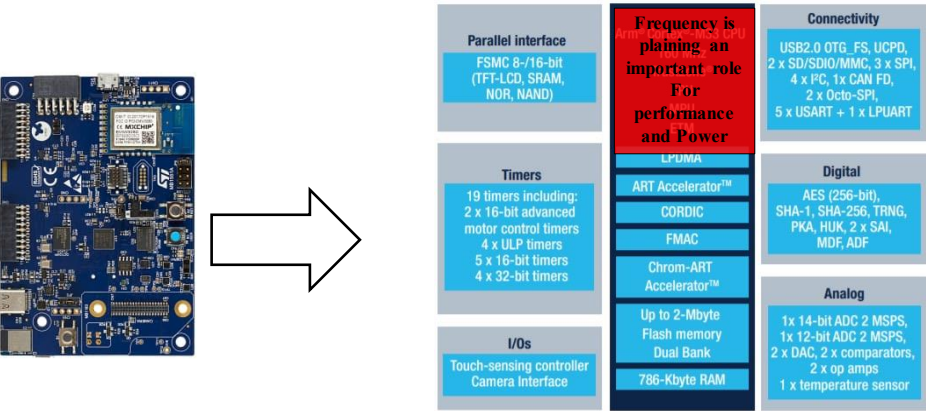
- ◆ **Fast Clocks** CPU, Communications, Burst Processing
- ◆ **Low-power** RTC, Remote, Battery, Energy Harvesting
- ◆ **Accurate** Stable over θ/V , Communications, RTC, Sensors
- ◆ **Failsafe** Robust—keeps system running in case of failure
- ◆ **Cheap** ... goes without saying ...

... or some combination of these features?

Is there a single clock in an embedded system?



STM32U5- Block Diagram (ES-LAB)



STM32U5: Reset and Clock Controller. (ES-LAB)

Six internal clock sources

High-speed internal 16 MHz RC oscillator (HSI16)
Multi-speed internal RC oscillators for system clock and for peripherals kernel clock (MSIS and MSIK)
Low-speed internal 32 kHz RC oscillator (LSI)
High-speed internal 48 MHz RC oscillator (HSI48)
Secure high-speed 48 MHz RC oscillator (SHSI) dedicated to SAES

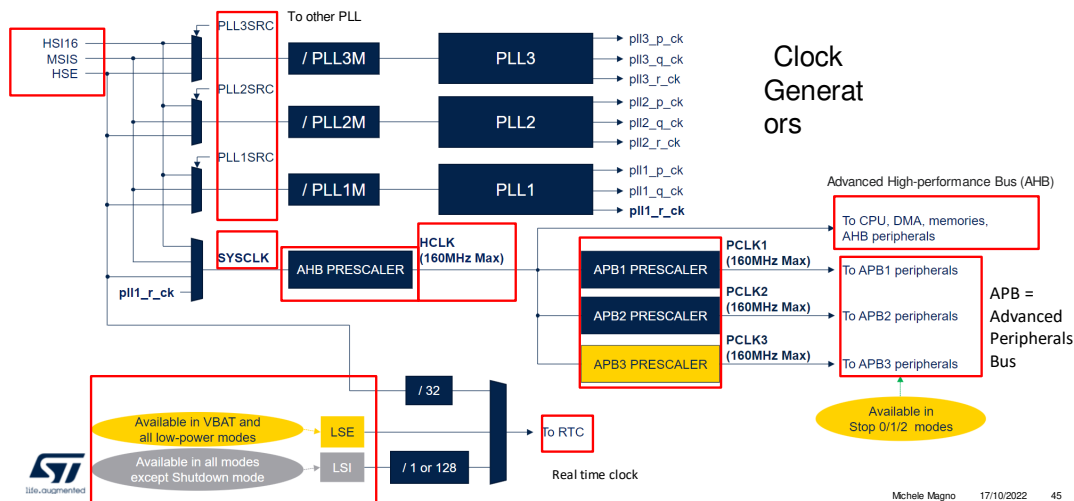
Two external oscillators

High-speed external 4 to 50 MHz oscillator (HSE) with clock security system
Low-speed external 32.768 kHz oscillator (LSE) with clock security system

Three PLL, each with 3 independent outputs

A phase-locked loop or phase lock loop (PLL) is a control system that generates an output signal whose phase is related to the phase of an input signal.

STL32F4: Simplify Clocks Tree



Michele Magno 17/10/2022 45
3 - 45

Comparison MSI vs HSI

	MSI (100 kHz to 48 MHz)		HSI16 (16 MHz)
	MSI mode (w/o LSE)	PLL mode (w/ 32.768 kHz LSE)	
Accuracy (typ.)	Over [0 - 85 °C]: +/- 1.55 %	Average accuracy = LSE accuracy	Over [0-85 °C]: +/- 0.8 %
	Over [1.62 - 3.6 V]: +0.8/-4.5 %	Jitter < 0.25%	Over [1.62 - 3.6 V]: +0.1/-0.2 %
Consumption (typ.)	100 kHz : 0.6 µA 800 kHz : 1.9 µA 1 MHz : 4.7 µA 8 MHz : 18.5 µA 16 MHz : 62 µA 48 MHz : 155 µA		150 µA
Startup time (typ.)	100 kHz : 10 µs 48 MHz : 2.5 µs	5% final frequency : 0.5 ms 1% final frequency : 1.5 ms max	1 µs



Michele Magno 17/10/2022 46
3 - 46

Lets see what is out of the MCU! General purpose In/Out pins GPIOs

General Purpose Input Out (GPIO): Theory

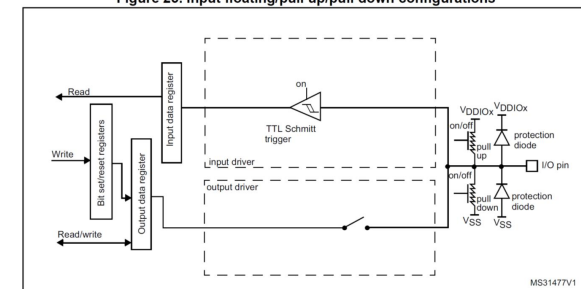
Each MCU pin can be used as a General Purpose digital input or output.

General-purpose I/Os (GPIO)

RM0351

Figure 25 shows the input configuration of the I/O port bit.

Figure 25. Input floating/pull up/pull down configurations



MS31477V1



Michele Magno 17/10/2022 47
3 - 47

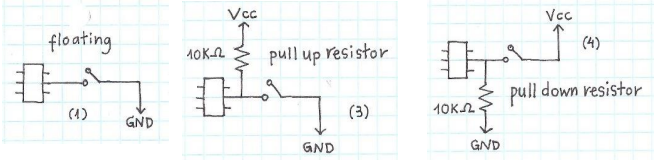
Michele Magno 17/10/2022 48
3 - 48

General Purpose Input Out (GPIO)

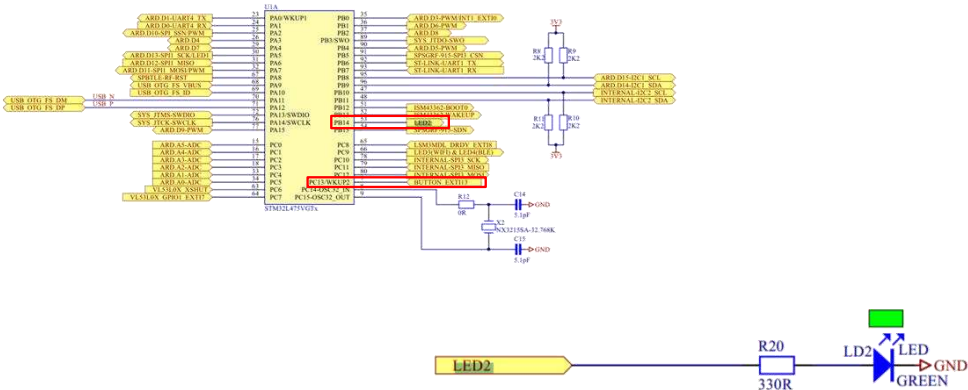
- Are used to control single pin devices (LEDS, Buttons, etc)
- First Hello World
- Are physically connected to other devices.
- Can have several functions

GPIO operating modes

- Input mode
 - Floating
 - Input wit pull-up/down
 - Analog input mode
- Output mode
 - Push-pull, open drain

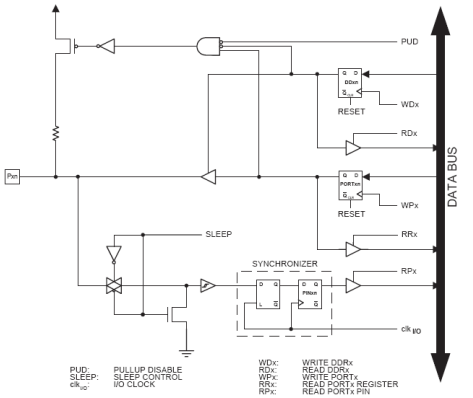


LED2 schematic and MCU connection



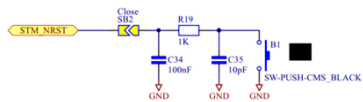
Inside GPIOs

- Each pin is independent
- Ports (out) and Pins (in) are **different!!!**



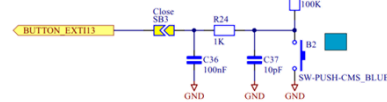
UM2839 User manual (ES-LAB)

RESET BUTTON



100nF should be place close to the MCU
10pF and 1K should be place close to the button

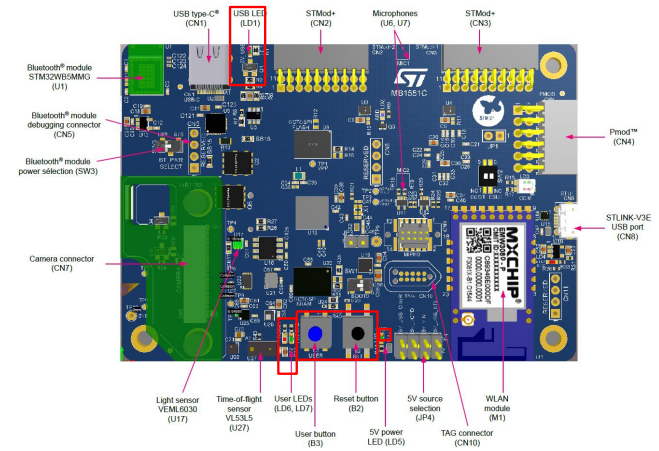
USER & WAKE-UP Button



100nF should be place close to the MCU
10pF and 1K should be place close to the button

Michele Magno 17/10/2022 53
3 - 53

Figure 3. B-US851-HOT02A Discovery kit layout (top view)



Michele Magno 17/10/2022 54
3 - 54

UM2839 section 7.1

I/O	Reference	Color	Name	Comment
NRST	B2	Black	RST	System reset
-	B1	Black	BT_RST	Bluetooth® module reset
PC13	B3	Blue	USER	User button
-	LD3	Bicolor (red and green)	COM	Blinking when flashing or debugging
-	LD4	Red	STLK_OVC	Red when the current is upper than 500 mA
PH7	LD7	Green	LD7	User LED lights up when PH7 is set to 0.
PH6	LD6	Red	LD6	User LED lights up when PH6 is set to 0.
-	LD5	Green	PWR	Available 5 V
PE13	LD2	Blue	ARD	ARDUINO® LED lights up when PE13 is set to 1.
-	LD1	Green	5V_USB	Available VBUS_C

Michele Magno 17/10/2022 55
3 - 55

What did you Learn?

- Compilation and deploying code on an embedded system
- Memory is crucial for MCU
 - What are the registers for data but also for instructions
 - Different type and architecture.
- Many clocks in a small device
 - Frequency, latency, energy tradeoffs
- External world
 - GPIO

3 - 56

Embedded Systems

Lecture 4.

Hardware Software Architecture.

Interrupts and Serial interfaces.

© Lothar Thiele

Computer Engineering and Networks Laboratory

Michele Magno

D-ITET center for project based learning

ETH

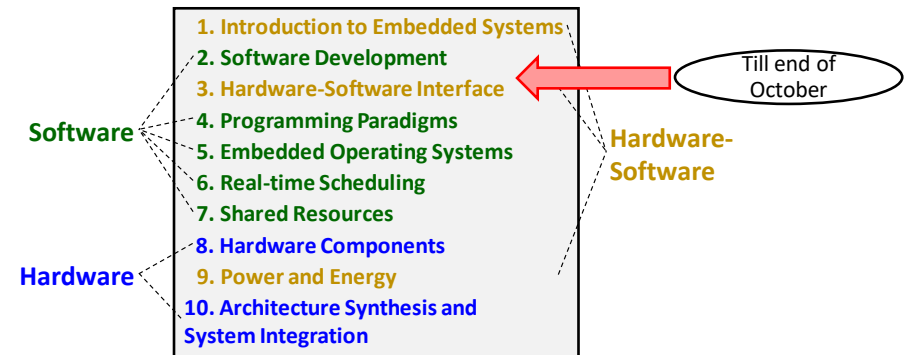
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Credits: Philipp Mayer, Francesco Conti.

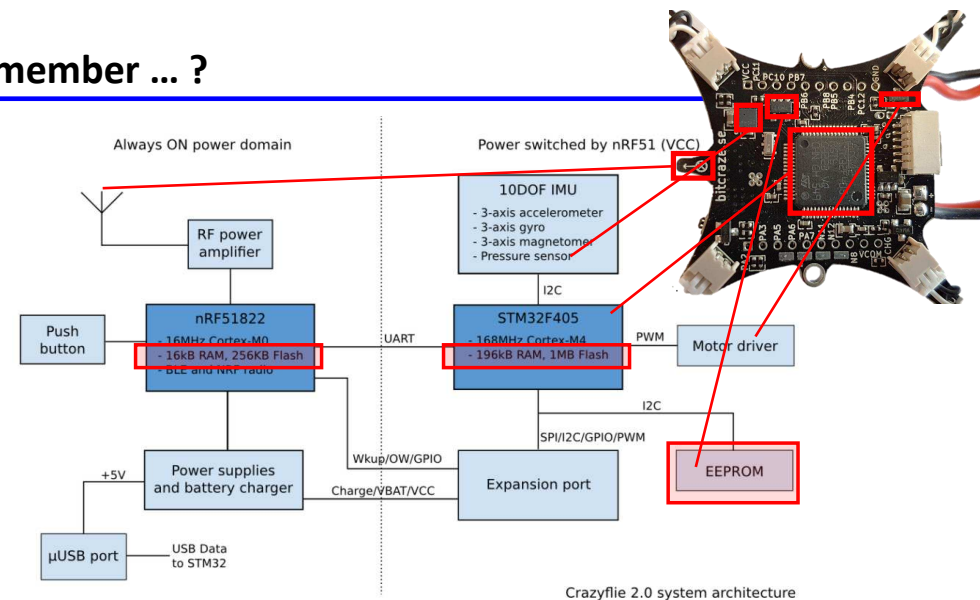
Do you Remember ?

- Memory
 - SRAM, DRAM, Flash
 - Addresses and value
 - Memory map
- Clocks
 - Many clocks
 - Flexibility
 - Energy and latency
- GPIO first peripherals going out.

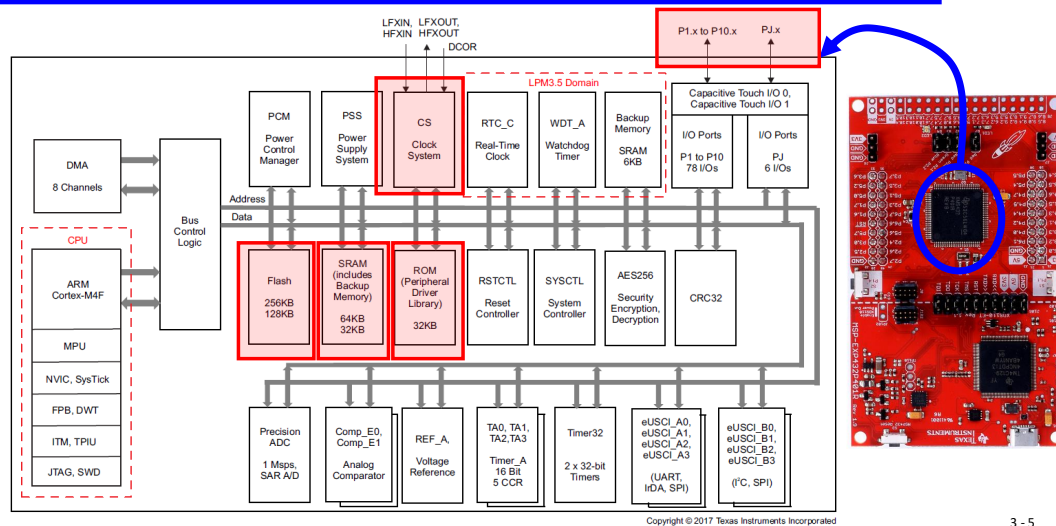
Where we are ...



Remember ... ?



MSP432P401R (Last year lab - ES-Lab)



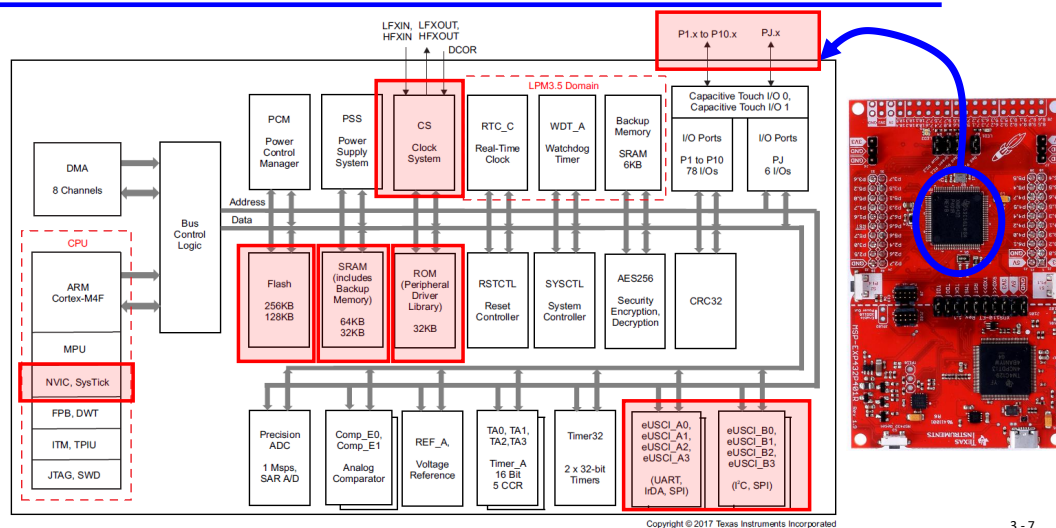
3 - 5

Today class will focus on

- Interrupts
- Input / Output
 - Serial communication
 - SPI
 - I2C
 - UART
- Link with the lab board

3 - 6

MSP432P401R (Last year lab - ES-Lab)

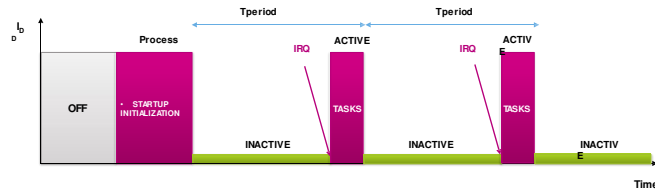


3 - 7

Interrupts

3 - 8

Typical application profile with Embedded systems



•Application phases:

- OFF – power is not applied to MCU
- STARTUP INITIALIZATION – MCU performs configuration (peripherals, clocks, ...)
- Tperiod
 - INACTIVE – MCU is in low power mode to reduce power consumption
 - ACTIVE – MCU is in normal mode and performs tasks

3 - 9

'Interrupt' Defined

From Wikipedia:

A hardware interrupt is an electronic alerting signal sent to the processor from an external device, either a part of the [device, such as an internal peripheral] or an external peripheral.

3 - 10

Waiting for an Event: Family Vacation

Polling

Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?



An engineering example...

Waiting for an Event: Family Vacation

Polling

Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?

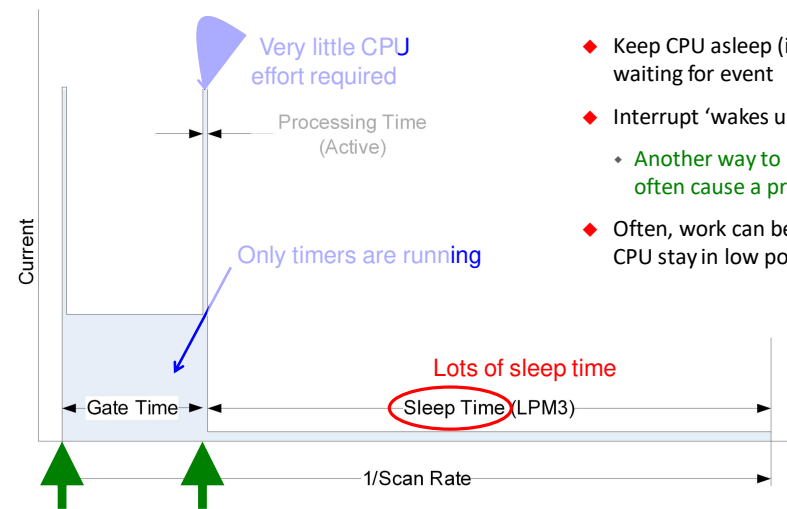


Interrupts

Wake me up when we get there...

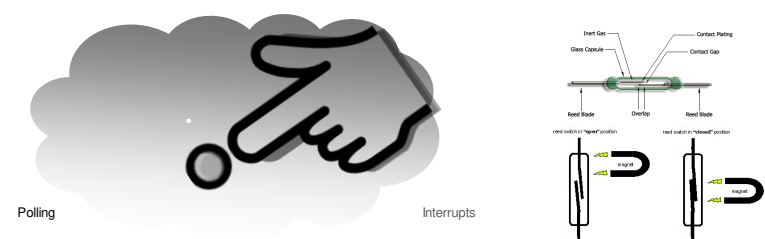
An engineering example...

Interrupts Help Support Ultra Low Power



- ◆ Keep CPU asleep (i.e. in Low Power Mode) while waiting for event
- ◆ Interrupt 'wakes up' CPU when it's required
 - ◆ Another way to look at it is that interrupts often cause a program state change
- ◆ Often, work can be done by peripherals, letting CPU stay in low power mode (e.g. Gate Time)

Waiting for an Event: Button Push



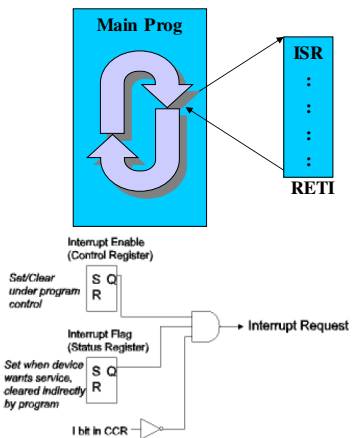
```
while(1) {  
    // Polling GPIO button  
    while (GPIO_getInputPinValue() == 1)  
        GPIO_toggleOutputOnPin();  
}  
  
// GPIO button interrupt  
#Interrupt service routine  
__interrupt void rx (void){  
    GPIO_toggleOutputOnPin();  
}
```

> 0.1% CPU Load
How interrupts can affect system design...

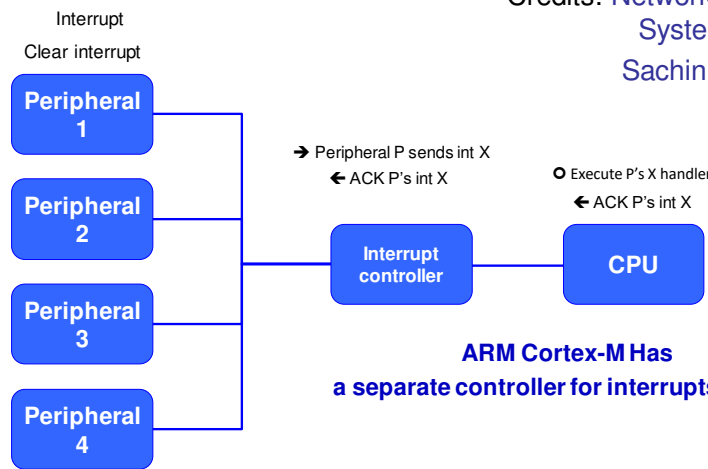
Interrupts

- A way to respond to an external event (i.e., flag being set) without polling
- How it works:
- H/W senses flag being set
 - Automatically transfers control to s/w that "services" the interrupt
 - When done, H/W returns control to wherever it left off

- Advantages:
- Transparent to user
 - cleaner code
 - μ C doesn't waste time polling



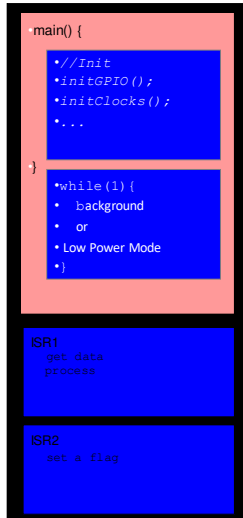
How do interrupts work?



ARM Cortex-M Has a separate controller for interrupts! WHY?

Credits: Networked Embedded Systems
Sachin Katti

Foreground / Background Scheduling



•System Initialization

- ◆ The beginning part of main() is usually dedicated to setting up your system

•Background

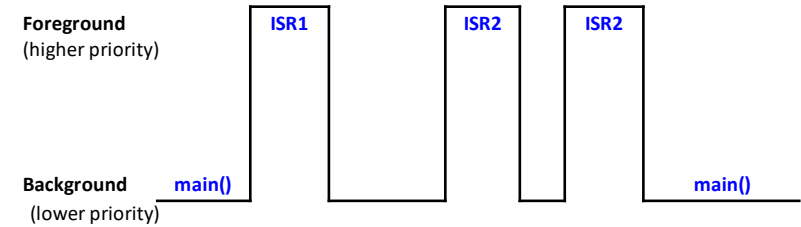
- ◆ Most systems have an endless loop that runs 'forever' in the background
- ◆ In this case, 'Background' implies that it runs at a lower priority than 'Foreground'
- ◆ In microcontrollers, the background loop often contains a **Low Power Mode** command – this sleeps the CPU/System until an interrupt event wakes it up

•Foreground

- ◆ **Interrupt Service Routine (ISR)** runs in response to enabled hardware interrupt
- ◆ These events may change modes in Background – such as waking the CPU out of low-power mode
- ◆ ISR's, by default, are not interruptible
- ◆ Some processing may be done in ISR, but it's usually best to keep them short

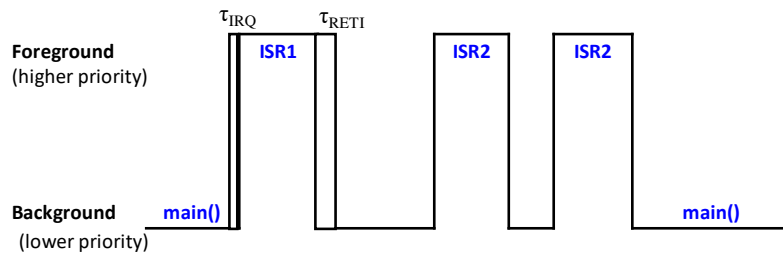
Michele Magno 17/10/2022 17
3 - 17

Foreground / Background (States)



Michele Magno 17/10/2022 18
3 - 18

The interrupt has a latency cost as everything

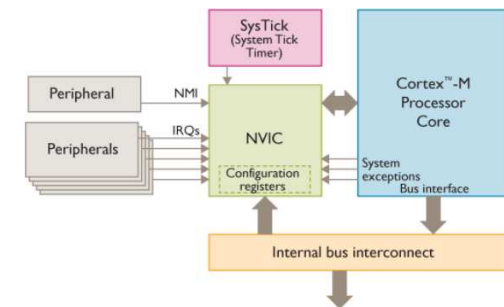


3 - 19

ARM Cortex-Mx / Nested Vector Interrupt Controller

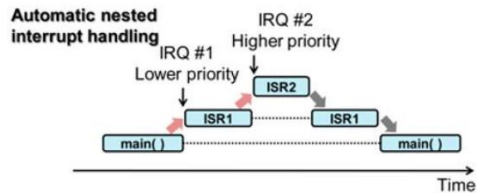
The **NVIC** includes the following features:

- A large number of **maskable interrupt channels**
- Several programmable priority levels (4 bits of interrupt priority are used)
- Pow-latency exception and interrupt handling (**12 Cycles, can change with architecture!!!**)
- Power management control
- Implementation of system control registers



3 - 20

Nested Vector Interrupt Controller

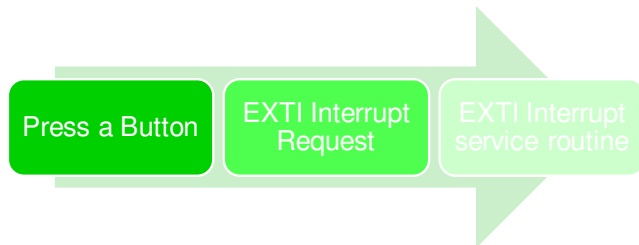


- **Nested Interrupt:** If an interrupt request (IRQ) with higher priority is raised, it is served first

3 - 21

EXTI – Purpose

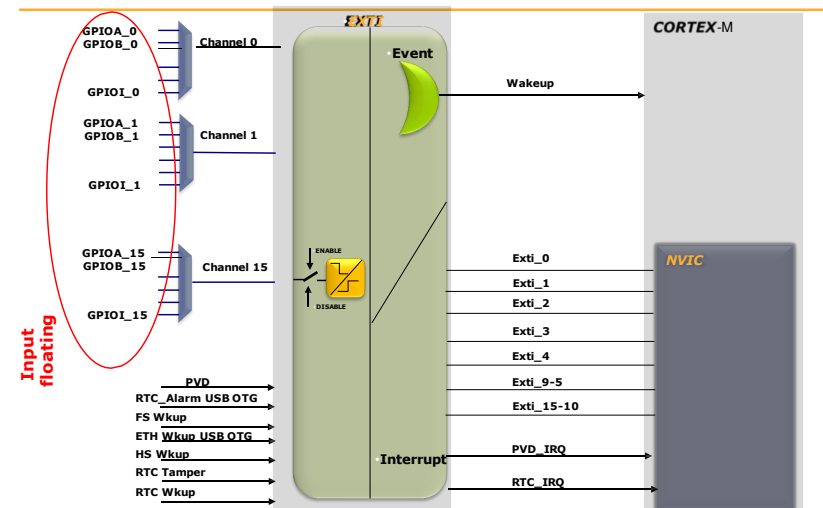
- We want to configure an external interrupt line.
- An EXTI line is configured to generate an interrupt on each falling edge.
- In the interrupt routine a led connected to a specific GPIO pin is toggled.



3 - 23

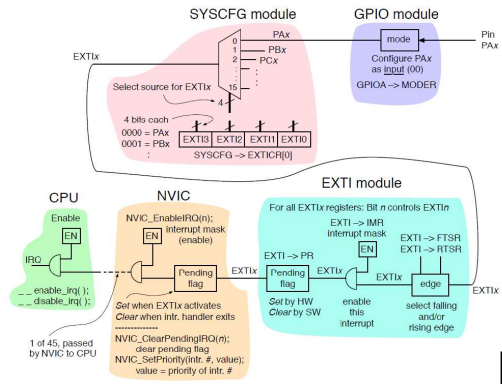
EXTernal Interrupt (EXTI)

EXTI module: from pin to NVIC



Functional Signal Flow

Signal flow/setup for External Interrupt EXT_i, $x = 0 \dots 15$



SYSCFG

remap the memory accessible in the code area
manage the external interrupt line connection to the GPIOs.

EXTI

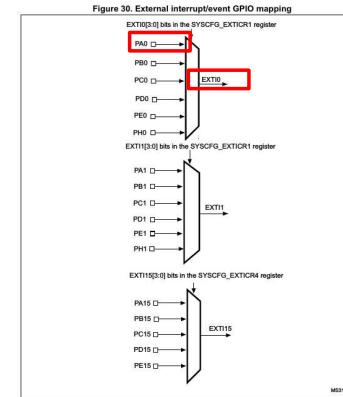
Configure GPIO pin as a digital input
Select the pin as the EXTIsource (in SYSCFG module)
Enable interrupt to be requested when a flag is set by the desired event (rising/falling edge)
Clear the pending flag (to ignore any previous events)

NVIC

Enable interrupt: `NVIC_EnableIRQ(IRQn);`
Set priority: `NVIC_SetPriority(IRQn, priority);`
Clear pending status: `NVIC_ClearPendingIRQ(IRQn);`
Initialize counters, pointers, global variables, etc.
Enable CPU Interrupts:

Many steps clocked means
many clocks to be served

EXTI - Capabilities



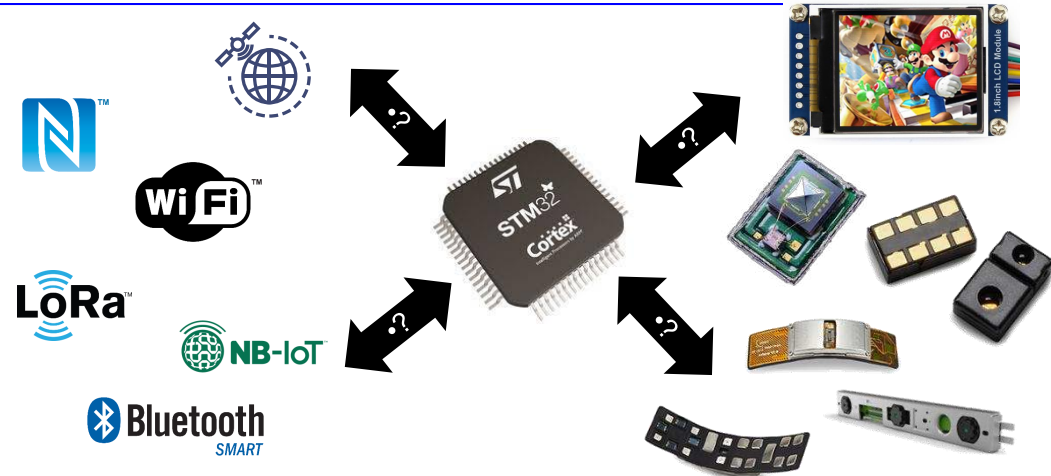
- There are 16 EXTI lines connected to GPIOs
- All pins with the same pin number are connected on the same EXTI line (eg. **Pin_2 Port A** and **Pin_2 Port C** share the same EXTI2)
- EXTI 16 – 22 are reserved for RTC, USB etc...

3 - 25

3 - 26

Input and Output

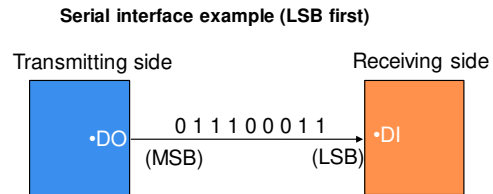
Serial Interfaces: Sensors, Display, Radio etc.



3 - 27

3 - 28

Comparison between Parallel and Serial Communication



Michele Magno 10/17/2022 29
3 - 29

Comparison between Parallel and Serial Communication

Characteristic	Parallel	Serial
Bus line	One line per bit	One line
Sequence	All bits of one word simultaneously	Sequence of bits
Transmission rate	High	Low
Bus length	Short distances	Short and long distances
Cost	High	Low
Critical characteristics	Synchronization between the different bits is demanding	Asynchronous transmission needs start and stop bits Synchronous transmission needs some other synchronization

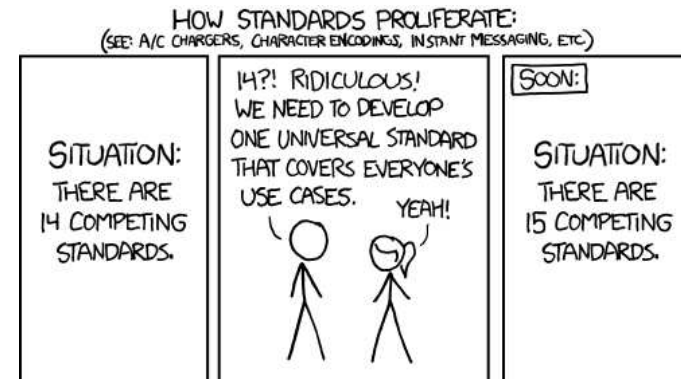
Michele Magno 10/17/2022 30
3 - 30

MCU Interfaces

- Digital, GPIOs
 - Several protocols for inter-chip communication
- Very often, a processor needs to *exchange information with other processors* or devices. To satisfy various needs, there exists many different *communication protocols*, such as
 - **UART** (Universal Asynchronous Receiver-Transmitter)
 - **SPI** (Serial Peripheral Interface Bus)
 - **I2C** (Inter-Integrated Circuit)
 - **USB** (Universal Serial Bus)
- As the principles are similar, we will just explain a representative of an asynchronous protocol (**UART**, no shared clock signal between sender and receiver) and one of a synchronous protocol (**SPI**, shared clock signal).

Michele Magno 10/17/2022 31
3 - 31

Serial Interface Standards



3 - 32

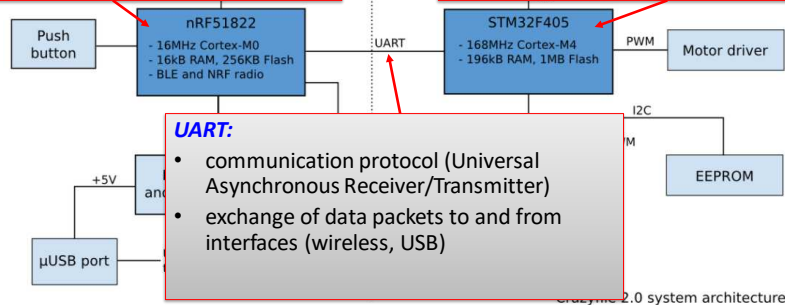
Remember?

low power CPU

- enabling power to the rest of the system
- battery charging and voltage measurement
- wireless radio (boot and operate)
- detect and check expansion boards

higher performance CPU

- sensor reading and motor control
- flight control
- telemetry (including the battery voltage)
- additional user development
- USB connection



I²C: Inter-Integrated Circuit Bus - 1

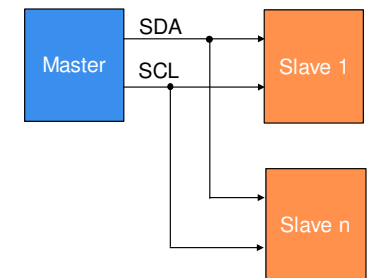
- Usually pronounced “I-Squared-C”
- Introduced by Philips (now NXP Semiconductors) in 1982
- Used for communication with external peripherals, for example:
 - EEPROMs
 - thermal sensors
 - real-time clocks
- Also used as a control interface for signal processing devices with separate data interfaces, for example:
 - radio frequency tuners
 - video decoders and encoders
 - audio processors

I²C – Inter-Integrated Circuit Bus



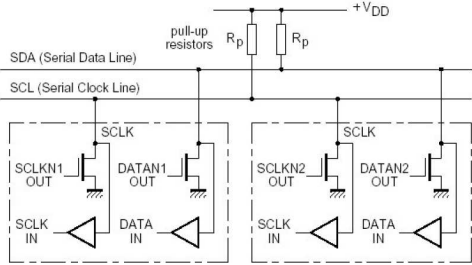
I²C: Inter-Integrated Circuit Bus - 2

- Three supported speed modes:
 - slow (under 100 Kbps)
 - fast (400 Kbps)
 - high-speed (3.4 Mbps) – in I2C v.2.0
- Maximum inter-IC distance of about 3 meters
 - (for moderate speeds, less for high-speed)
- Can support multi-master mode
 - For complex applications
 - Communication is always started by a master, both in single-master and multi-master mode
- Half-duplex synchronous communication scheme
 - the master of the communication generates the clock (SCL) on which data (SDA) is synchronized



I²C: Inter-Integrated Circuit Bus - 3

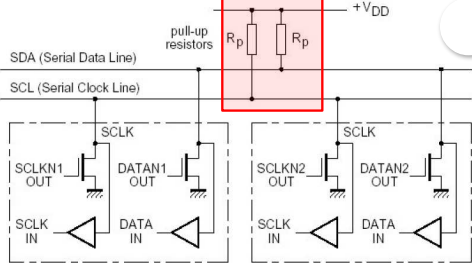
- Based on two lines:
 - SCL (serial clock)
 - SDA (serial data)



Pull-Up resistors, **Pull-Down** by open-drain drivers
Wired-AND: if *any* driver pulls down, the line is low (avoids short circuits)
Any module on the bus can act as **master**, **slave** or both
typical case: MCU is the master, peripherals/sensors are slaves

I²C: Inter-Integrated Circuit Bus - 3

- Based on two lines:
 - SCL (serial clock)
 - SDA (serial data)

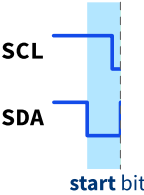


Pull-Up resistors, **Pull-Down** by open-drain drivers
Wired-AND: if *any* driver pulls down, the line is low (avoids short circuits)
Any module on the bus can act as **master**, **slave** or both
typical case: MCU is the master, peripherals/sensors are slaves

I²C: Interface Protocol

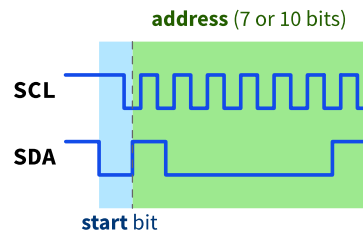
SCL
SDA

- In idle, both **SCL** and **SDA** are pulled-up to **1**



To start the communication, the **master**:
asserts the **start** bit (**SDA** 1→0 transition while **SCL** is still **1**)
then, it starts generating the **SCL** clock
except for the start and stop bits, **SDA** transitions *only* when **SCL** is **0**

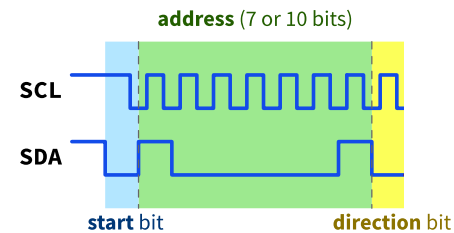
I²C: Interface Protocol



- The **master** transmits the **slave address**:
 - **broadcasted** to all devices on the I²C bus
 - used to select the target **slave**
 - either **7 bits** or **10 bits** (newer devices – 7 bits address space is small!)
 - *in the example, the address is 7'b1000001*

3 - 41

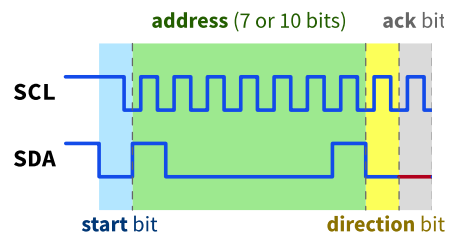
I²C: Interface Protocol



- The **master** transmits a **direction bit**:
 - a **0** for **master → slave** (write) transfer
 - a **1** for **slave → master** (read) transfer
 - *in the example, suppose a write transfer*

3 - 42

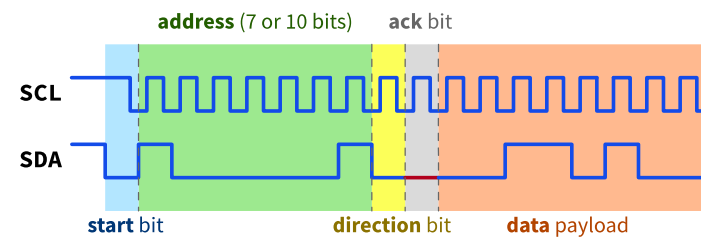
I²C: Interface Protocol



- The **slave** then **acknowledges** reception:
 - by driving **SDA** to **0**
 - if not acknowledged, the transaction must be repeated by the master

3 - 43

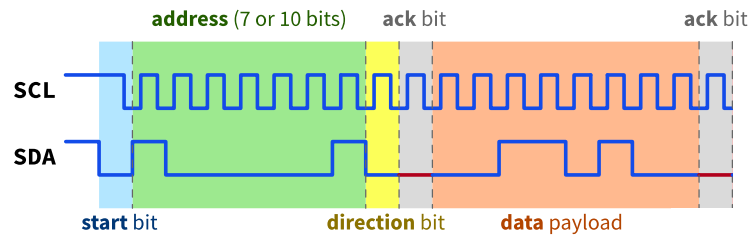
I²C: Interface Protocol



- The **master** transmits its data payload:
 - each payload packet is **8 bits**
 - there might be more than one packet, depending on application
 - *in the example, data payload is 8'b00110100*

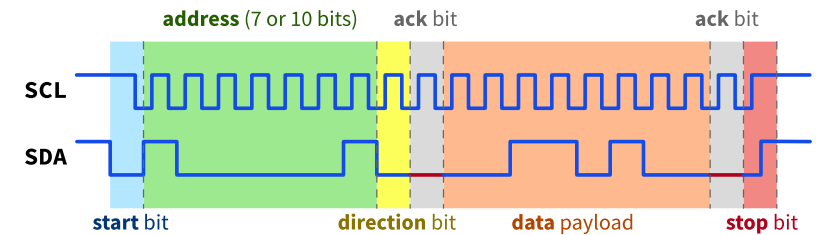
3 - 44

I²C: Interface Protocol



6. The **slave** acknowledges reception of the data packet:
 - 1 **ack bit** every 8 **payload bits**
 - slave must acknowledge each packet

I²C: Interface Protocol

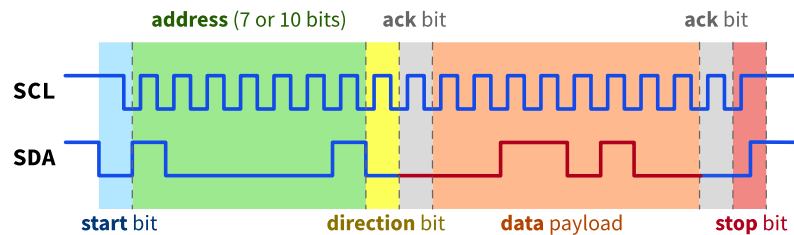


7. At the end of the transfer, the **master** transmits a **stop bit**:
 - first, it sets **SDA** to 0
 - then it releases **SCL** (i.e. it lets it go to 1)
 - finally, it releases **SDA** which also goes to 1

3 - 45

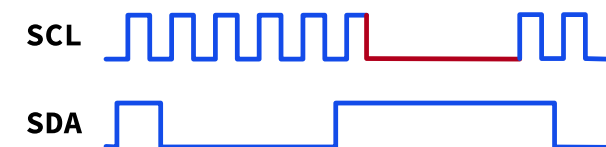
3 - 46

I²C: Interface Protocol



- Reads work similarly, but data transfer – ack roles are reversed:
- the **slave** drives **SDA** when transmitting the data byte
 - the **master** acknowledges the transfer

I²C: Interface Protocol



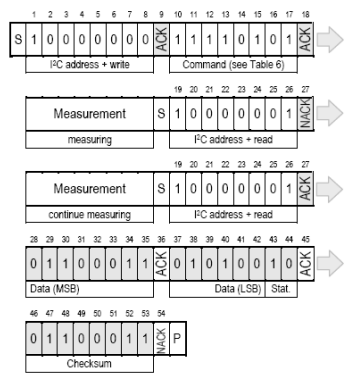
- Slave** can ask for more time to process a bit by **clock stretching**:
- drive **SCL** to 0 if in need of more processing time

3 - 47

3 - 48

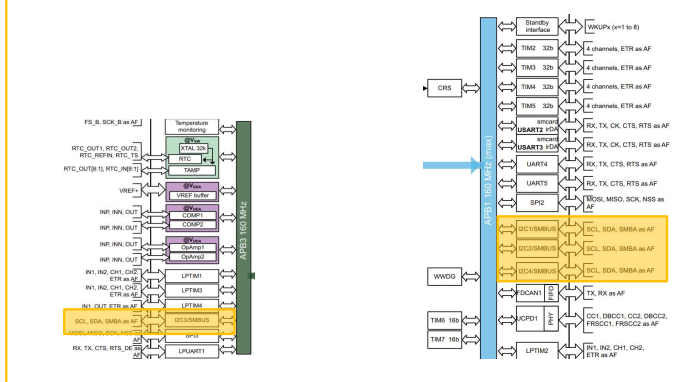
I2C: Interface Protocol

Example of MCU – sensor communication (data acquisition) via I2C bus

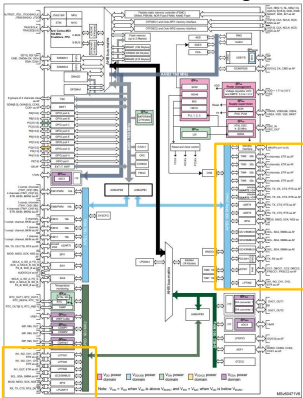


I2C – STM32U585xx

4 x I2C



STM32U585xx block diagram



<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Datasheet, page 21

I2C – Connected Sensors

•3-axis magnetometer
IIS2MDCTR
Read=00111101(3Dh)
Write=00111100(3Ch)

•Humidity and temperature
HTS221
Read=10111111(BFh)
Write=00111110(BEh)

•ToF, gesture-detection
VL53L5CXV0GC/1
Read=01010011(53h)
Write=01010010(52h)

MEMS nano pressure sensor
LPS22HH
Read=10111011(BBh)
Write=10111010(BAh)

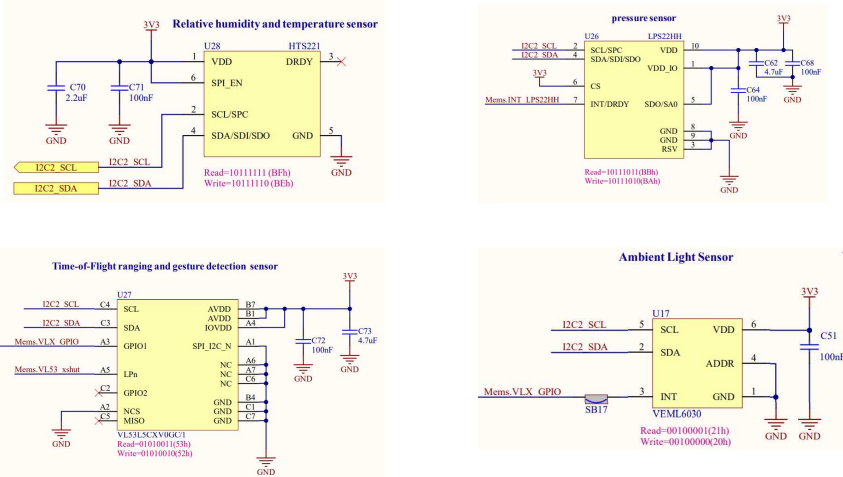
I2C2 @ VMCU
-PH4 / PH5

•Ambient light sensor
•VEML6030
Read=00100001(21h)
Write=00100000(20h)

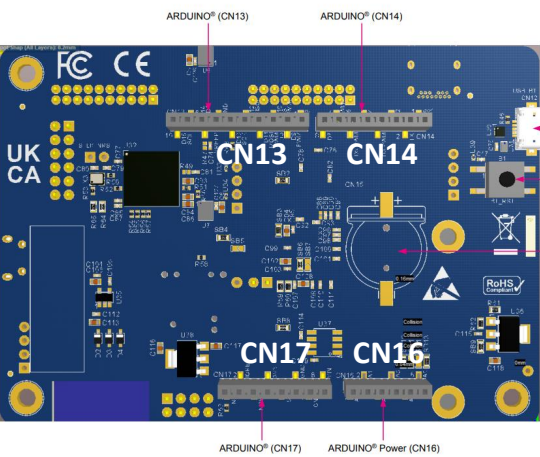
•Authentication and security
•STSAFE-A110
Read=01000000(40h)
Write=01000001(41h)

•3D accelerometer and 3D gyroscope
•ISM330DHCX
Read=11010101(D5h)
Write=11010100(D4h)

I2C – Connected Sensors Schematic



I2C – External Connection



<https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html>
→ Documents → UM2839, page 10

Connector	Pin number	Pin name	Signal name	STM32L4x pin	Function
CN17	1	NC	-	-	-
	2	IOREF	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	3V3	-	-	3.3 V I/O
	5	5V	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
	8	VIN	-	-	Power input
CN16	1	A0	ARD_ADC_A0	PC0	ADC
	2	A1	ARD_ADC_A1	PC2	ADC
	3	A2	ARD_ADC_A2	PC4	ADC
	4	A3	ARD_ADC_A3	PC8	ADC
	5	A4	ARD_ADC_A4	PA7	ADC / I2C1_SDA
	6	A5	ARD_ADC_A5	PB0	ADC / I2C1_SCL
	10	SCL/D15	I2C1_SCL	PE8	I2C1_SCL
	9	SDA/D14	I2C1_SDA	PB9	I2C1_SDA
CN13	8	VREFP	VREFP	-	VREFP
	7	GND	GND	-	GND
	6	SKVD13	ARD_D13_SPI1_SCK	PE13	SPI1_SCK / LD2
	5	MSBOD12	ARD_D12_SPI1_MISO	PE14	SPI1_MISO
	4	PWMASD11	ARD_D11_SPI1_MOSI	PE16	SPI1_MOSI / TIM1_CH4N
	3	PWMAOSD10	ARD_D10_TIM_SPI1_NSS	PE12	SPI1_NSS / TIM1_CH4N
	2	PWMAOSD9	ARD_D9_TIM	PA8	TIM1_CH1
	1	DA	ARD_D8_TM	PE9	TIM16_CH1
CN14	7	PWMAOSD8	ARD_D8_TM	PE9	TIM16_CH1
	6	DA	ARD_D4_TM	PE7	GPIO
	5	PWMAOSD7	ARD_D3_TM	PE6	TIM16_CH4N
	4	D2	ARD_D2_IO	PD15	GPIO
	3	TXD1	ARD_D1_TX	PD8	UART1_TX
	2	RXD0	ARD_D0_RX	PD9	UART1_RX
	1	RXD0	ARD_D0_RX	PD9	UART1_RX

I2C – Typical Datasheet - HTS221

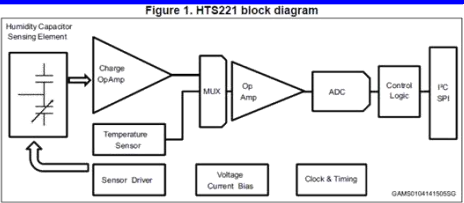


Figure 2. Pin configuration (bottom view)

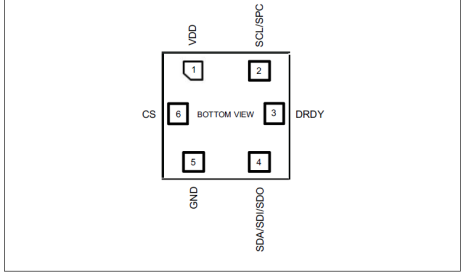


Table 10. SAD + Read/Write patterns

Command	SAD[6:0]	R/W	SAD+R/W
Read	1011111	1	10111111 (BFh)
Write	1011111	0	10111110 (BEh)

Address sometimes hidden in the text.

What happens if I want to use one sensor twice on the bus?

Table 2. Pin description

Pin n°	Name	Function
1	VDD	Power supply
2	SCL/SPC	I2C serial clock (SCL) SPI serial port clock (SPC)
3	DRDY	Data Ready output signal
4	SDA/SDI/SDO	I2C serial data (SDA) 3-wire SPI serial data input / output (SDI/SDO)
5	GND	Ground
6	SPI enable	I2C/SPI mode selection (1: SPI idle mode / I2C communication enabled; 0: SPI communication mode / I2C disabled)

I2C – Typical Datasheet - HTS221

Table 15. Register address map

Name	Type	Register address (hex)	Default (hex)
Reserved		00-0E	Do not modify
WHO_AM_I	R	0F	BC
AV_CONF	R/W	10	1B
Reserved		11-1C	Do not modify
CTRL_REG1	R/W	20	0
CTRL_REG2	R/W	21	0
CTRL_REG3	R/W	22	0
Reserved		23-26	Do not modify
STATUS_REG	R	27	0
HUMIDITY_OUT_L	R	28	Output
HUMIDITY_OUT_H	R	29	Output
TEMP_OUT_L	R	2A	Output
TEMP_OUT_H	R	2B	Output
Reserved		2C-2F	Do not modify
CALIB_0_F	R/W	30-3F	Do not modify

WHO_AM_I (0Fh)

Device identification

7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	0

This read-only register contains the device identifier, set to 0BCh

Configuration

7.9 TEMP_OUT_L (2Ah)

Temperature data (LSB)

7	6	5	4	3	2	1	0
TOUT7	TOUT6	TOUT5	TOUT4	TOUT3	TOUT2	TOUT1	TOUT0
[7:0] TOUT7 - TOUT0: Temperature data LSB (see TEMPERATURE_OUT_H)							

7.10 TEMP_OUT_H (2Bh)

Temperature data (MSB)

15	14	13	12	11	10	9	8
TOUT15	TOUT14	TOUT13	TOUT12	TOUT11	TOUT10	TOUT9	TOUT8
[15:8] TOUT15 - TOUT8: Temperature data MSB							

Temperature data are expressed as TEMP_OUT_H & TEMP_OUT_L as 2's complement numbers.

SPI – Serial Peripherals Interface



SPI: Serial Peripheral Interface - 1

NXP – Qualcomm (almost) - NXP

- Introduced by Motorola (now Freescale Semiconductors) for the MC68HCxx line of microcontrollers
- Use cases are generally similar to I2C
- Generally faster than I2C (up to several Mbit/s)
 - Short-distance (i.e. on printed circuit boards)
- Single-master, multiple slave
 - needs one chip select per slave device (no broadcast addressing)
- Full-duplex synchronous communication scheme
 - master drives the clock (SCLK or SCK)
 - clock polarity (i.e. write/read edges) and phase depend on specific application!

3 - 57

SPI: Serial Peripheral Interface - 2

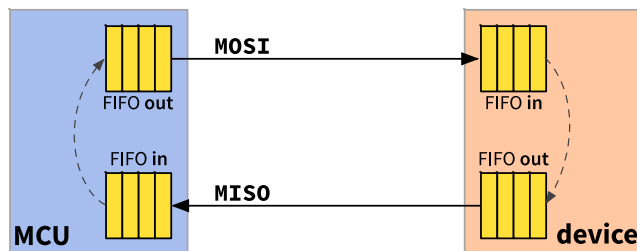
- Based on two data and two control lines:
 - MISO (master-in, slave-out data)
 - MOSI (master-out, slave-in data)
 - SCK (clock)
 - CSN (chip select, one per slave – usually active low)
- Names are not standard, beware! Some possible alternatives:
 - SDI (SPI data in) instead of MISO
 - SDO (SPI data out) instead of MOSI
 - SCLK, CLK, SPC, ... instead of SCK
 - CS, SS (slave select), SSN (slave select, active low) ... instead of CSN



3 - 58

SPI: Serial Peripheral Interface - 3

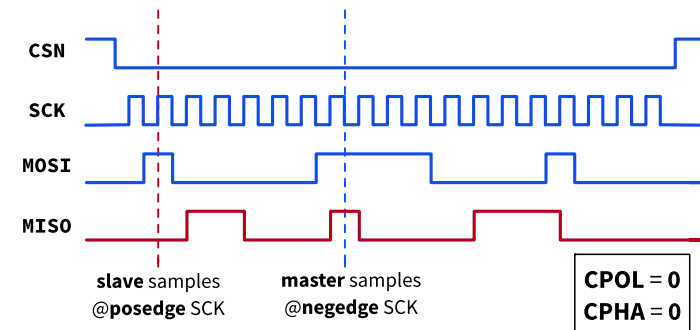
- Full-duplex transfer: data is streamed between master and slave shift-registers / FIFO buffers:
 - the master pushes the content of its buffer to the slave via MOSI
 - the slave pushes the content of its buffer to the master via MISO
- Processing / sensing / ... happens in between (dashed line)



3 - 59

SPI: Interface Protocol - 1

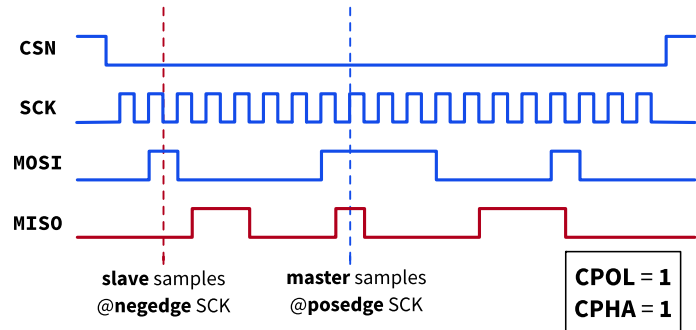
- Four operating modes, varying by clock polarity (CPOL) and phase (CPHA):
 - polarity sets the initial value of the SPI clock signal
 - phase defines the edge at which MOSI is switched and the one at which MISO is sampled



3 - 60

SPI: Interface Protocol - 1

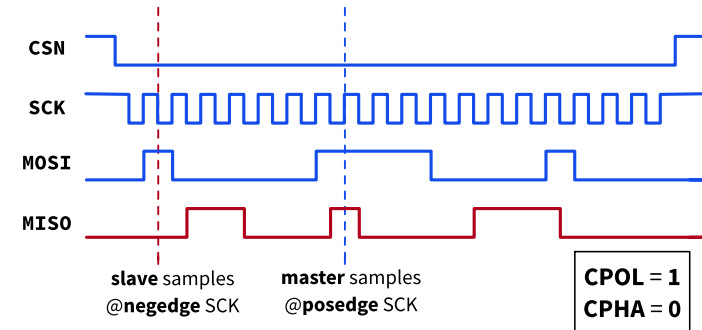
- Four operating modes, varying by clock polarity (CPOL) and phase (CPHA):
 - polarity sets the initial value of the SPI clock signal
 - phase defines the edge at which MOSI is switched and the one at which MISO is sampled



3 - 61

SPI: Interface Protocol - 1

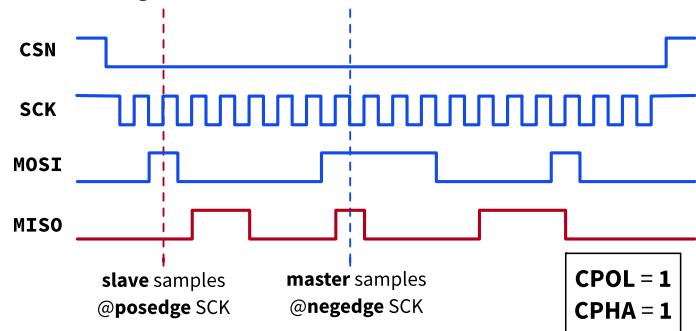
- Four operating modes, varying by clock polarity (CPOL) and phase (CPHA):
 - polarity sets the initial value of the SPI clock signal
 - phase defines the edge at which MOSI is switched and the one at which MISO is sampled



3 - 62

SPI: Interface Protocol - 1

- Four operating modes, varying by clock polarity (CPOL) and phase (CPHA):
 - polarity sets the initial value of the SPI clock signal
 - phase defines the edge at which MOSI is switched and the one at which MISO is sampled

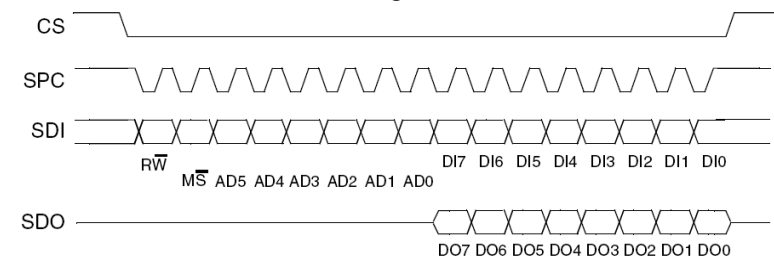


3 - 63

SPI: Interface Protocol - 2

- Master completely in charge of transfer
 - no ack, no clock stretching contrarily to I2C
- More complex behavior than simple data streaming can be mapped on top of SPI protocol
 - e.g. command + address + data streaming

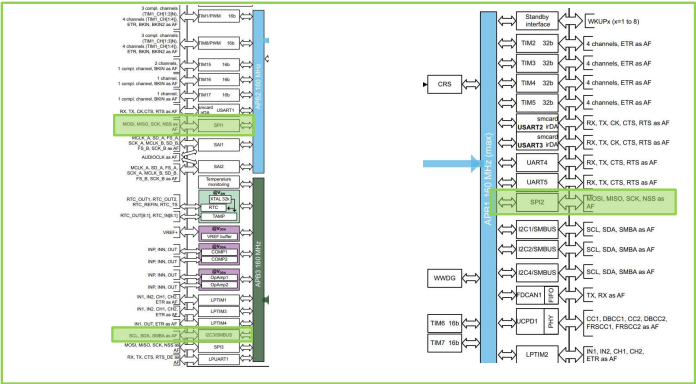
Several names!



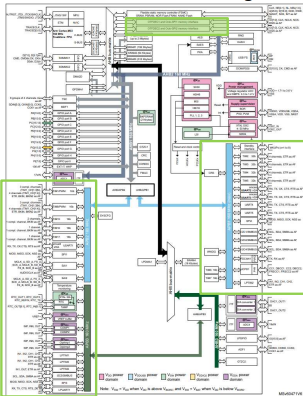
3 - 64

SPI – STM32U585xx

3 x SPI 1 x SPI with dual OCTOSPI



STM32U585xx block diagram

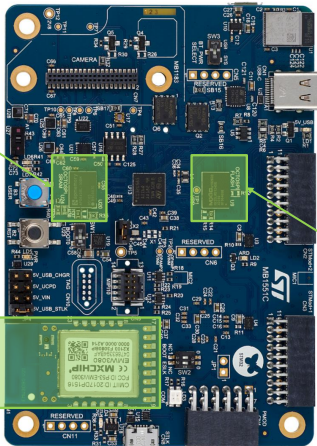


<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Datasheet, page 21

SPI – Connected Sensors

12-Mbit Octo-SPI Flash
MX25LM51245GXD1005
OCTOSPI2

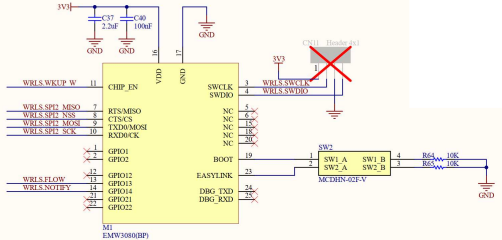
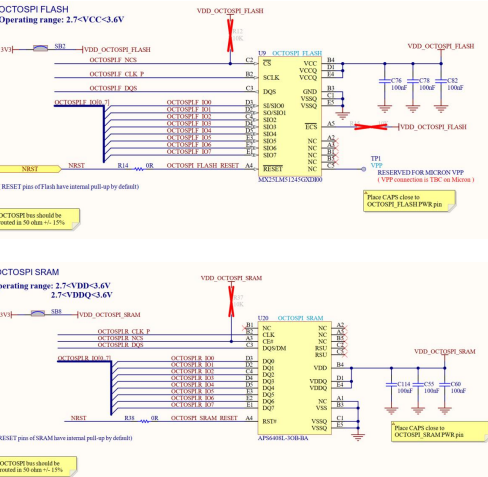
Wifi Module
• EMW3080
• SPI2



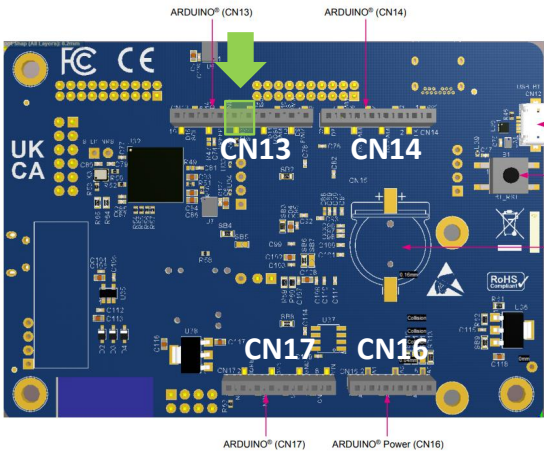
SPI2 @ 3V3
OCTOSPI1 @ 3V3
OCTOSPI2 @ 3V3

12-Mbit Octo-SPI RAM
• APS6408L-30B-BA
• Octo-SPI

SPI – Connected Modules and Memory



SPI – External Connection



<https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html>
→ Documents → UM2839, page 10

Connector	Pin number	Pin name	Signal name	STM32L pin	Function
CN17	1	NC	-	-	-
	2	IREF	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	3V3	-	-	3.3 V I/O
	5	5V	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
	8	VIN	-	-	Power input
CN16	1	A0	ARD_ADC_A0	PC0	ADC
	2	A1	ARD_ADC_A1	PC2	ADC
	3	A2	ARD_ADC_A2	PC4	ADC
	4	A3	ARD_ADC_A3	PC5	ADC
	5	A4	ARD_ADC_A4	PA7	ADC / IC1_SDA
	6	A5	ARD_ADC_A5	PB0	ADC / IC1_SCL
	10	IC1_SDA	IC1_SCL	PB8	IC1_SCL
	8	SDA/I2C	IC1_SDA	PB9	IC1_SDA
	9	VREFP	VREFP	-	-
	7	GND	-	-	GND
CN13	6	SKD13	ARD_D13_SPI1_SCK	PE13	SPI1_SCK / LD2
	5	MISO12	ARD_D12_SPI1_MISO	PE14	SPI1_MISO
CN14	4	PWMMSIO10	ARD_D11_SPI1_MOSI	PE15	SPI1_MOSI / TMR1_CHN
	3	PWMMSIO10	ARD_D10_SPI1_NSS	PE12	SPI1_NSS
	2	PWMMSIO9	ARD_D9_TM	PA8	TMR1_CH1
	1	D8	ARD_D8_JD	PC1	GPIO
	8	D7	ARD_D7_JD	PF13	GPIO
	7	PWMMSIO6	ARD_D6_TM	PB6	TMR1_CH2
	6	PWMMSIO5	ARD_D5_TM	PC3	TMR1_CH1
	5	D4	ARD_D4_INT	PE7	GPIO
	4	PWMMSIO3	ARD_D3_TM	PB2	TMR1_CHN
	3	D2	ARD_D2_JD	PD15	GPIO
CN17	2	TXD1	ARD_D1_TX	PD6	USART3_TX
	1	RXD0	ARD_D0_RX	PD5	USART3_RX

SPI vs I²C

For **point-to-point**, SPI is simple and efficient

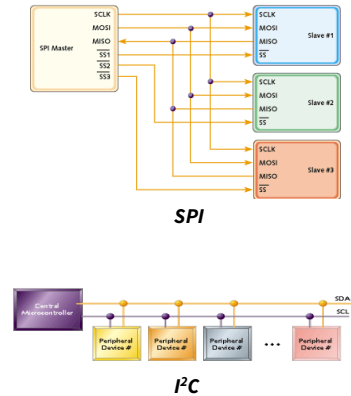
Less overhead than I²C due to **lack of addressing**, plus SPI is **full-duplex**.

For multiple slaves, each slave needs **separate slave select** signal

SPI requires more effort and more hardware than I²C

Quad-SPI also exists

4x the bandwidth, often used by Flash drives



UART - 1

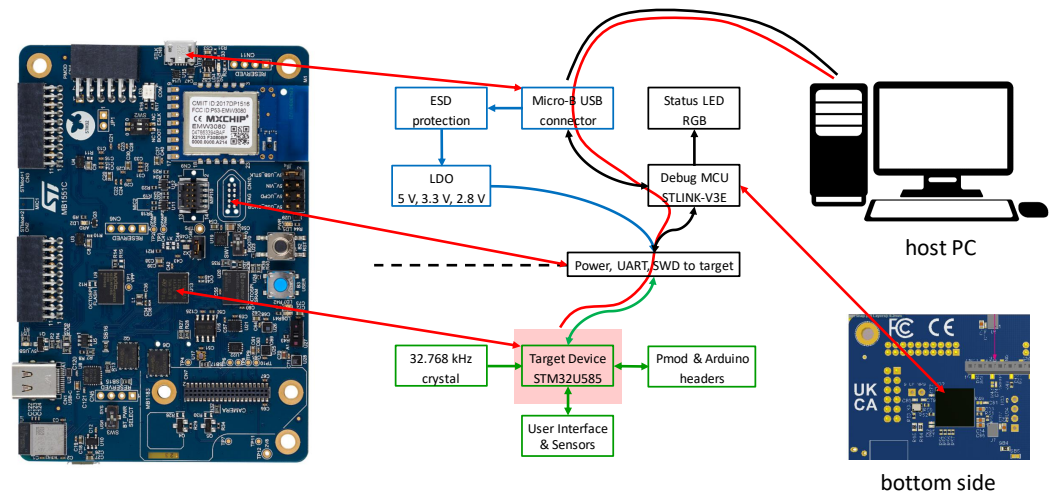
- Stands for Universal Asynchronous Receiver-Transmitter
 - sometimes also found as USART (Universal Synchronous-Asynchronous Receiver Transmitter)
- Used to interface MCUs with other computing devices:
 - Communication with other processors, a PC (e.g. a serial terminal)
 - Used to interface the microcontroller with others transmission bus as: RS232, RS485, USB, CAN BUS, KNX, LonWorks ecc.
 - Used to connect MCUs with modems and transceivers as telephone modems, Bluetooth, Wi-Fi, GSM/GPRS/HDPSA

UART - Universal Asynchronous Receiver-Transmitter



3 - 69

Software Development with STM32U585xx (ES-Lab)

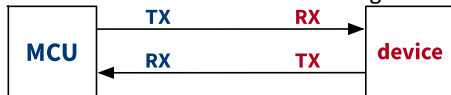


3 - 71

3 - 72

UART - 2

- Essentially a parallel2serial (TX), serial2parallel (RX) converter couple
 - e.g. using shift registers for P2S conversion
- Asynchronous: no common clock shared
 - Each device has its own local clock, typically running faster than the bit rate (e.g. 8x faster)
 - The phase of the receiver clock is locked onto the edge of the transmitted data



- Highly configurable
 - parity / no parity
 - data framing (e.g number of stop bits, number of payload bits)
 - simplex, full-duplex or half-duplex

3 - 73

UART: “baud rate” vs “bit rate”

- UART communication speed is defined by its symbol rate measured in baud:
 - 1 baud = 1 **symbol per second**
 - in UART, a **symbol** has two values (0/1) -> **1 bit**
 - this number includes both data payload and protocol bits (e.g. parity, framing) – this number is also called “physical” or “gross” bit rate
- This can cause some confusion
 - Some people use “bit rate” for UART when referring only to payload bits
 - In some devices (e.g. modems) one symbol might correspond to more bits -> baud rate is not the same as gross bit rate
 - Bottom line: to be 100% clear, **always talk of baud rate when referring to UART**, and remember that in UART **1 symbol = 1 bit**

Gross bit rate [edit]

In digital communication systems, the **physical layer gross bitrate**,^[5] *raw bitrate*,^[6] *data signaling rate*,^[7] *gross data transfer rate*^[8] or *uncoded transmission rate*^[9] (sometimes written as a variable R_b ^{[5][6]} or f_b ^[9]) is the total number of physically transferred bits per second over a communication link, including useful data as well as protocol overhead.

In case of *serial communications*, the gross bit rate is related to the bit transmission time T_b , as:

$$R_b = \frac{1}{T_b},$$

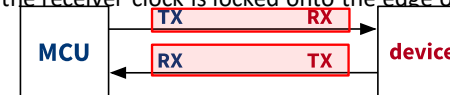
The gross bit rate is related to the **symbol rate** or modulation rate, which is expressed in **bauds** or symbols per second. However, the gross bit rate and the baud value are equal *only* when there are only two levels per symbol, representing 0 and 1, meaning that each symbol of a **data transmission system** carries exactly one bit of data; for example, this is not the case for modern modulation systems used in **modems** and **LAN equipment**.^[10]

Ref. Wikipedia “Bit rate” page

3 - 75

UART - 2

- Essentially a parallel2serial (TX), serial2parallel (RX) converter couple
 - e.g. using shift registers for P2S conversion
- Asynchronous: no common clock shared
 - Each device has its own local clock, typically running faster than the bit rate (e.g. 8x faster)
 - The phase of the receiver clock is locked onto the edge of the transmitted data

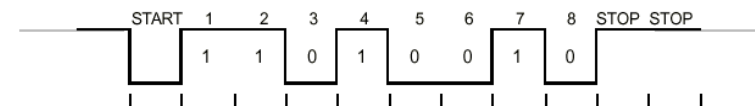


- Highly configurable
 - parity / no parity
 - data framing (e.g number of stop bits, number of payload bits)
 - simplex, full-duplex or half-duplex

3 - 74

UART: Interface Protocol

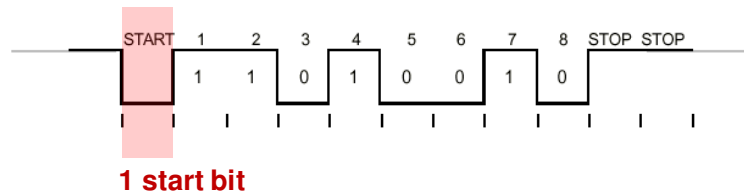
- In idle, the transmission line is driven to 1



3 - 76

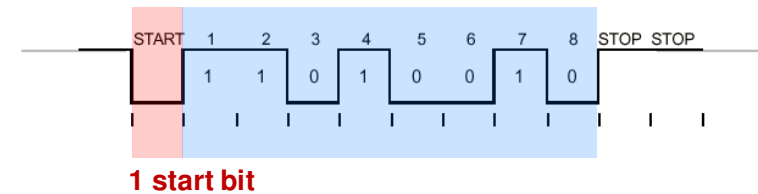
UART: Interface Protocol

- The transfer begins with a start bit:
 - the transmission line is driven to 0



UART: Interface Protocol

- Then, a symbol of 5 to 9 bits is transmitted:
 - most often, 8 bits (1 ASCII character)
 - the symbol size is defined by the application and known a-priori with respect to the communication

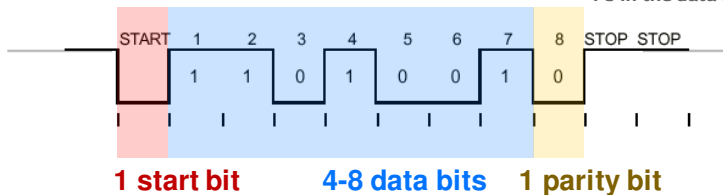


UART: Interface Protocol

- One of the data bits can be used for parity:

- odd parity
- even parity

- in this case, 4-8 bits can be used for data



$$p_{\text{odd}} = \bigoplus_{i=1}^N b_i$$

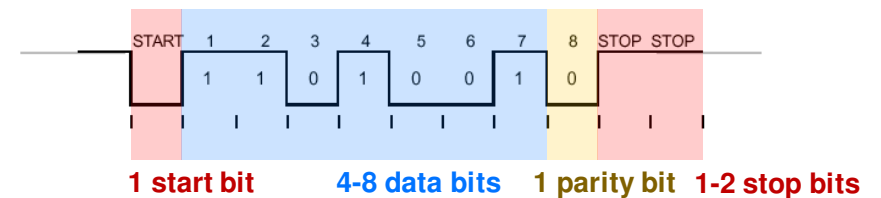
$$p_{\text{even}} = \mathbf{XNOR}_{i=1}^N b_i$$

Even parity where a 1-parity-bit is sent if there are an even number of 1-bits

Symbol	Truth Table																																				
 3-input Ex-OR Gate	<table><tr><th>C</th><th>B</th><th>A</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	C	B	A	Q	0	0	0	0	0	0	1	1	0	1	0	1	0	1	1	0	1	0	0	1	1	0	1	0	1	1	0	0	1	1	1	1
C	B	A	Q																																		
0	0	0	0																																		
0	0	1	1																																		
0	1	0	1																																		
0	1	1	0																																		
1	0	0	1																																		
1	0	1	0																																		
1	1	0	0																																		
1	1	1	1																																		
Boolean Expression $Q = A \oplus B \oplus C$	"Any ODD Number of Inputs" gives Q																																				

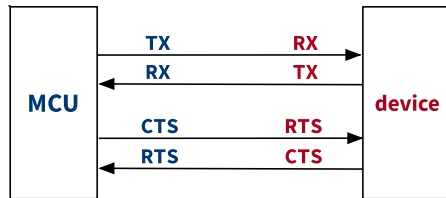
UART: Interface Protocol

- Finally, 1-2 stop bits:
 - Transmission line brought back to 1
 - 1 or 2 stop bits depending on application



UART: Handshake

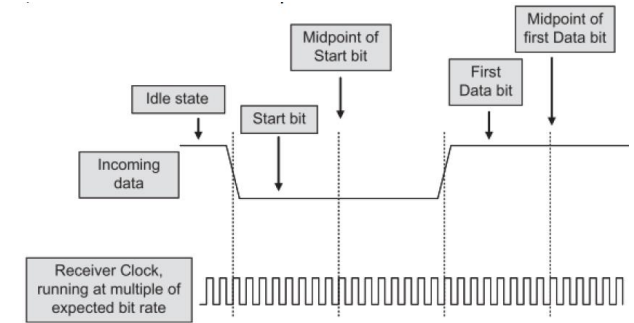
- The UART protocol can also include a handshake:
- request-to-send (RTS) signal from the MCU to the device means that the MCU can accept new data
- clear-to-send (CTS) signal from the device to the MCU means that the device can send new data
- signals have dual meaning if seen from the other point of view
- exchange happens when CTS and RTS are both asserted



3 - 81

UART

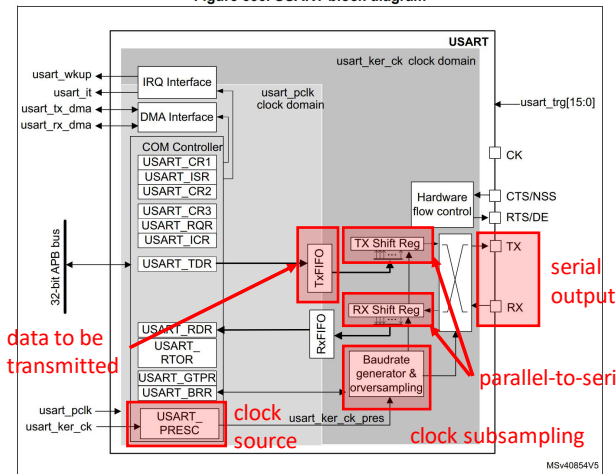
- The receiver runs an *internal clock* whose frequency is an exact multiple of the expected bit rate.
- When a *Start bit* is detected, a counter begins to count clock cycles e.g. 8 cycles until the midpoint of the anticipated Start bit is reached.
- The clock counter counts a further 16 cycles, to the middle of the first *Data bit*, and so on until the *Stop bit*.



3 - 82

Transmission Rate

Figure 656. USART block diagram



Clock subsampling:

- The clock subsampling block is complex, as one tries to match a large set of transmission rates with a fixed input frequency.

Clock Source:

- USART_PCLK in the lab is then divided to have the baud rate generator

Example:

- Will be done in the LAB!

<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Reference Manual RM0456, page 2258

3 - 83

Memory-Mapped Device Access

Bus	Secure boundary address	Non-secure boundary address	Size (bytes)	Peripheral	Peripheral register map
	0x5000 5000 - 0x5000 53FF	0x4000 5000 - 0x4000 53FF	1 K	UART5	USART register map
	0x5000 4C00 - 0x5000 4FFF	0x4000 4C00 - 0x4000 4FFF	1 K	UART4	
	0x5000 4800 - 0x5000 4BFF	0x4000 4800 - 0x4000 4BFF	1 K	USART3	
	0x5000 4400 - 0x5000 47FF	0x4000 4400 - 0x4000 47FF	1 K	USART2	

57.8.12 USART receive data register (USART_RDR)
Address offset: 0x24
Reset value: 0x0000 0000



Bits 31:9: Reserved, must be kept at reset value.
Bits 8:0: RDR[8:0]: Receive data value

57.8.13 USART transmit data register (USART_TDR)
Address offset: 0x28
Reset value: 0x0000 0000



Bits 31:9: Reserved, must be kept at reset value.
Bits 8:0: TDR[8:0]: Transmit data value

- Configuration of Transmitter and Receiver must match; otherwise, they can not communicate.
- Examples of configuration parameters:
 - transmission rate (baud rate, i.e., symbols/s)
 - LSB or MSB first
 - number of bits per packet
 - parity bit
 - number of stop bits
 - interrupt-based communication
 - clock source

buffer for received bits and bits that should be transmitted

<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Reference Manual RM0456, page 2337

3 - 84

Software Interface

Part of C program that *prints one byte to a UART* terminal on the host PC:

```
void uart_init()
{
    USART2_CLK_ENABLE(); // Enable clocksource for USART2

    /* Configure USART2 */
    huart2.Instance = USART2;
    huart2.Init.BaudRate = 115200;
    huart2.Init.WordLength = UART_WORDLENGTH_8B;
    huart2.Init.StopBits = UART_STOPBITS_1;
    huart2.Init.Parity = UART_PARITY_NONE;
    huart2.Init.Mode = UART_MODE_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;

    HAL_UART_Init(&huart2); // Init USART2 peripheral
}

int main(void)
{
    HAL_Init(); // Initialize HAL
    uart_gpio_init(); // Init UART GPIO (RX and TX pins)
    uart_init(); // Init UART peripheral

    /* Transmit one byte with 100 ms timeout */
    HAL_UART_Transmit(&huart2, &byte, 1, 100)
    /* Code */
}
```

data structure `uartConfig` contains the configuration of the UART

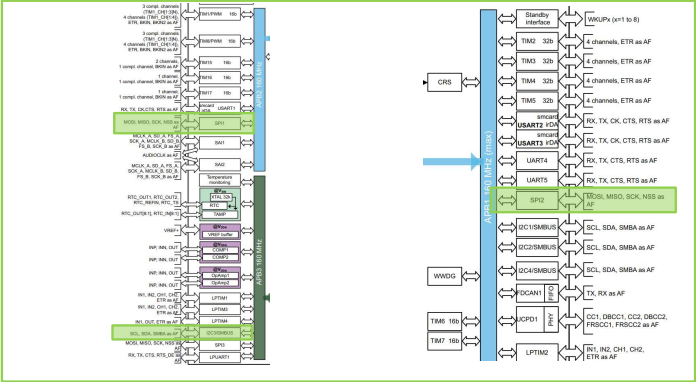
USART2 defines the start address for the USART2 memory block: 0x4000_4400 (for non-secure)

Use `HAL_UART_Init` to write the configuration registers of the instance USART2

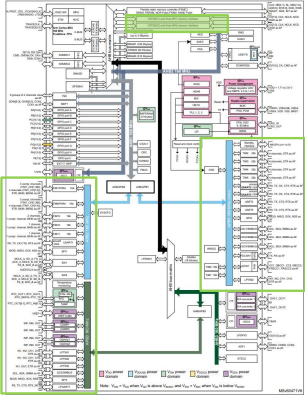
start UART

SPI – STM32U585xx

3 x SPI 1 x SPI with dual OCTOSPI



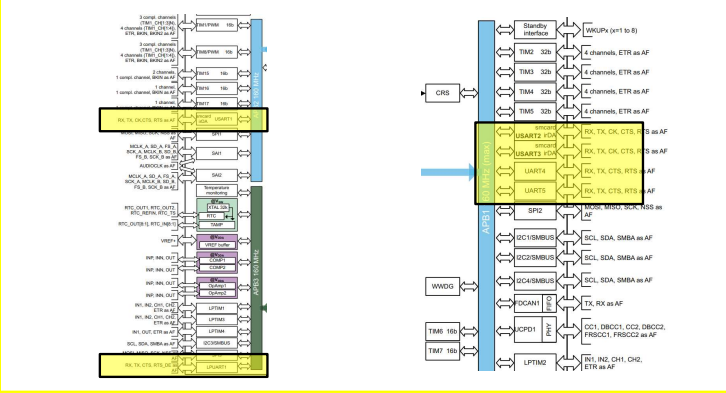
STM32U585xx block diagram



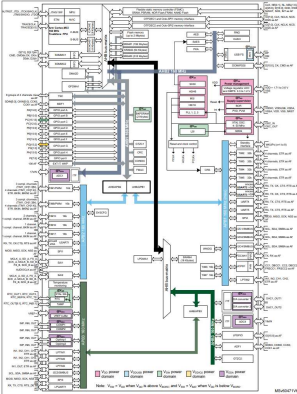
<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Datasheet, page 21

UART – STM32U5

3x USART 2x UART and 1x LPUART



STM32U585xx block diagram



<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Datasheet, page 21

UART – STM32U585xx

Table 22. USART, UART and LPUART features

USART modes/features ⁽¹⁾	USART1/2/3	UART4/5	LPUART1
Hardware flow control for modem	X	X	X
Continuous communication using DMA	X	X	X
Multiprocessor communication	X	X	X
Synchronous mode (master/slave)	X	-	-
Smartcard mode	X	-	-
Single-wire half-duplex communication	X	X	X
IrDA SIR ENDEC block	X	X	-
LIN mode	X	X	-
Dual-clock domain and wakeup from Stop mode	X ⁽²⁾	X ⁽²⁾	X ⁽³⁾
Receiver timeout interrupt	X	X	-
Modbus communication	X	X	-
Auto-baud rate detection	X	X	-
Driver enable	X	X	X
USART data length	7, 8 and 9 bits		
Tx/Rx FIFO	X	X	X
Tx/Rx FIFO size	8 bytes		
Autonomous mode	X	X	X

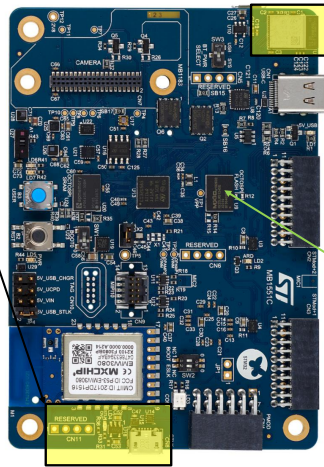
- 1. X = supported.
- 2. Wakeup supported from Stop 0 and Stop 1 modes.
- 3. Wakeup supported from Stop 0, Stop 1 and Stop 2 modes.

Table 72. Typical dynamic current consumption of peripherals (continued)

Bus	Peripheral	LDO				SMPS				Unit
		Range1	Range2	Range3	Stop 2 Stop 1	Range1	Range2	Range3	Stop 1 Stop 2	
USART	USART1	2.38	2.16	1.96	1.76	-	1.14	0.97	0.81	0.62
	USART1 indep ⁽¹⁾	4.48	4.09	3.71	3.35	-	2.17	1.84	1.54	1.19
	LPUART1	1.10	1.07	0.97	0.88	0.97	0.48	0.41	0.31	0.31
	LPUART1 indep ⁽¹⁾	1.98	1.78	1.62	1.45	1.46	0.89	0.80	0.67	0.52
	UART4	1.86	1.70	1.54	1.39	-	0.90	0.76	0.64	0.50
	UART4 indep ⁽¹⁾	3.47	3.17	2.87	2.60	-	1.68	1.42	1.19	0.93
	UART5	1.93	1.76	1.60	1.44	-	0.94	0.79	0.66	0.51
	UART5 indep ⁽¹⁾	3.57	3.25	2.95	2.67	-	1.72	1.46	1.23	0.95
	UCPD1	1.60	1.46	1.33	1.17	-	0.78	0.66	0.55	0.43
	USART2	5.35	5.04	4.57	4.12	-	2.67	2.26	1.91	1.65
USART	USART2 indep ⁽¹⁾	3.57	3.24	2.95	2.69	-	1.72	1.46	1.22	0.94
	USART3	2.10	1.91	1.73	1.57	-	1.02	0.86	0.72	0.56
	USART3 indep ⁽¹⁾	4.24	3.88	3.5	3.17	-	2.05	1.73	1.45	1.12

<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Datasheet, page 84

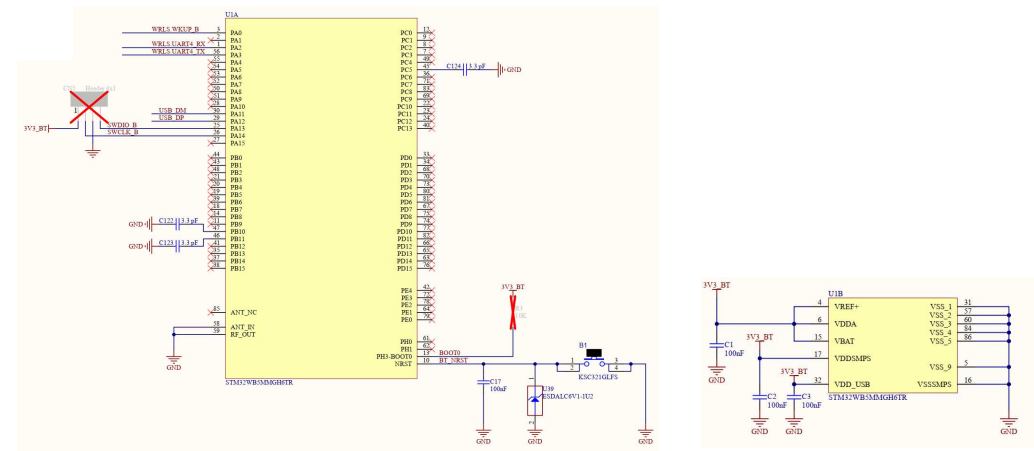
UART – Connected Modules



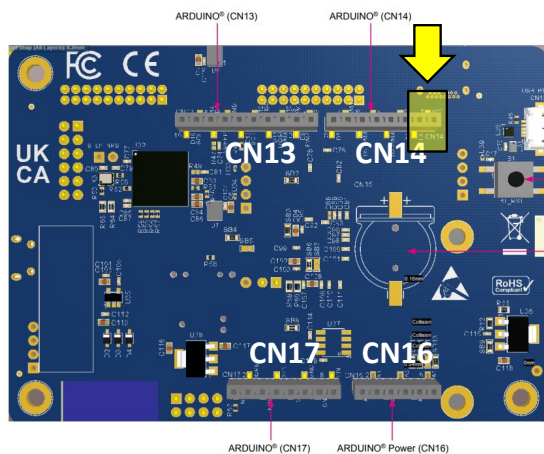
- DEBUGGING CONNECTION
- ST-LINK
- UART1

- Bluetooth Module
- STM32WB5MMGH6TR
- UART4

UART – Connected Module



UART – External Connection



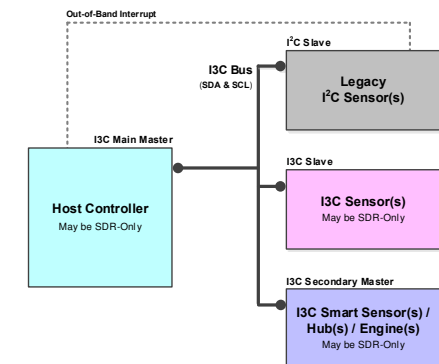
<https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html>

→ Documents → UM2839, page 10

Connector	Pin number	Pin name	Signal name	474032, 474033	Function
CN17	1	NC	-	-	-
	2	NRST	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	DV3	-	-	3.3 V I/O
	5	SV3	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
CN18	8	VIN	-	-	Power input
	1	A2	ARD_ADC_A2	PC2	ADC
	2	A1	ARD_ADC_A1	PC2	ADC
	3	A2	ARD_ADC_A3	PC4	ADC
	4	A3	ARD_ADC_A3	PC5	ADC
	5	A4	ARD_ADC_A4	PA7	ADC / I2C1_SDA
	6	A5	ARD_ADC_A5	PB0	ADC / I2C1_SCL
CN19	10	SCKD15	I2C1_SCL	PB8	I2C1_SCL
	9	SCKD14	I2C1_SDA	PB9	I2C1_SDA
	8	VREFP	VREFP	-	VREFP
	7	GND	GND	-	GND
	6	SCKD13	ARD_D13_SPH1_SCK	PE13	SPH1_SCK / L2D
	5	MISOID12	ARD_D12_SPH1_MISO	PE14	SPH1_MISO
	4	PWMCMOS11	ARD_D11_SPH1_MOSI	PE15	SPH1_MOSI / TM2_CHAN
CN20	3	PWMCMOS10	ARD_D10_TM_SPH1_NSS	PE12	SPH1_NSS / TM2_CHAN
	1	DM8	ARD_D8_TM	PA8	TM2_CHAN
	8	D7	ARD_D7_IO	PC1	GPIO
	7	DM7	ARD_D7_TM	PF13	GPIO
	7	PWMCM6	ARD_D6_TM	PE6	TM2_CHAN
	6	PWMCM5	ARD_D5_TM	PE0	TM2_CHAN
	5	DM4	ARD_D4_TM	PE7	GPIO
CN21	4	PWMCM3	ARD_D3_TM	PE7	TM2_CHAN
	2	TX01	ARD_D1_TX	PC9	UART3_TX
CN22	1	RXD0	ARD_D0_RX	PC8	UART3_RX

The next-generation: I3C example

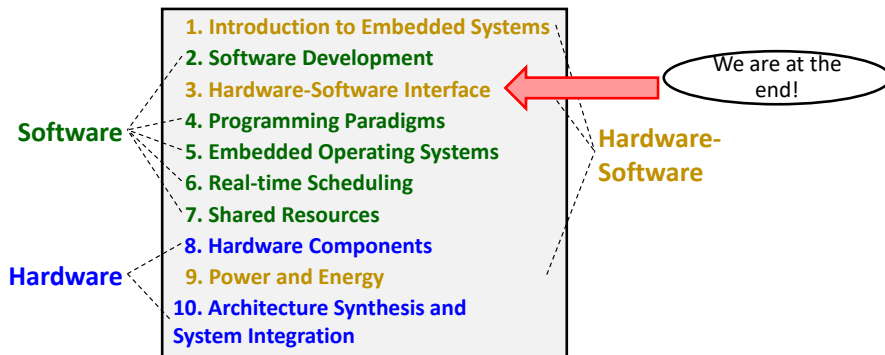
- An evolution of I2C proposed by the MIPI alliance (2016/7)
- Designed to fit applications currently using I2C, SPI, UART
- Many operating modes, I2C backward-compatibility also supported
 - without some of the most «exotic» features such as SCL stretching
 - supporting also (mainly) push-pull drivers
- Targets high data rate and energy efficiency



What did you Learn?

- Interrupts are crucial for
 - Work in parallel
 - Duty cycling
 - Energy Efficiency
- External world Interface
 - Serial ports
 - SPI
 - I2C
 - UART
- This is today used in real embedded systems

Where we are ...



Embedded Systems

Lecture 5.

Hardware Software Architecture.

Analog Digital Converter and Digital Analog Converters PWM

© Lothar Thiele

Computer Engineering and Networks Laboratory

Michele Magno

D-ITET center for project based learning

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3 - 93

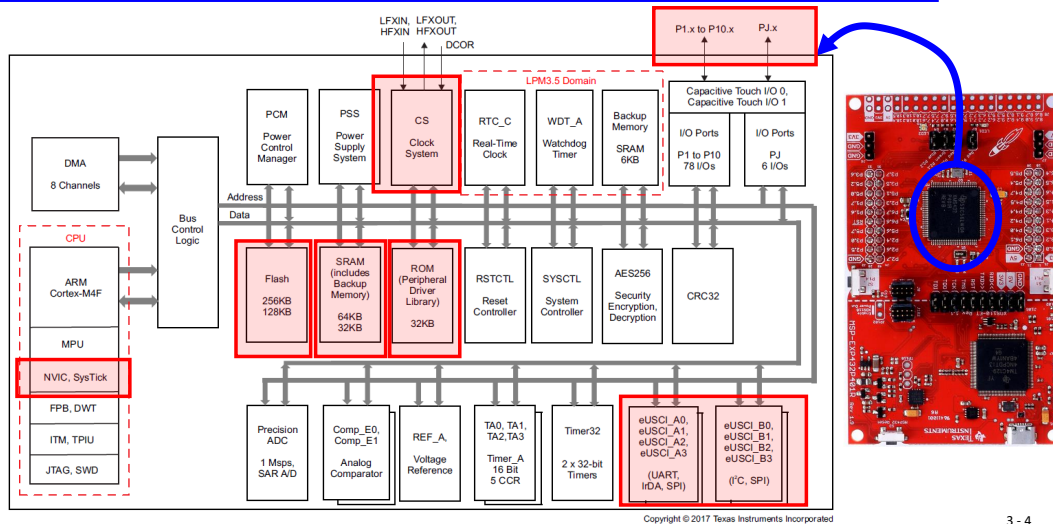
Do you Remember ?

- Interrupt
 - Hardware Software Mechanism
 - Used for wake up the systems from sleep state
 - IRQ, Interrupt Request
 - ISR, Interrupt Service Routine
- Serial Interface.
 - I2C
 - SPI
 - UART
 - USB (Only mentioned)

3 - 2

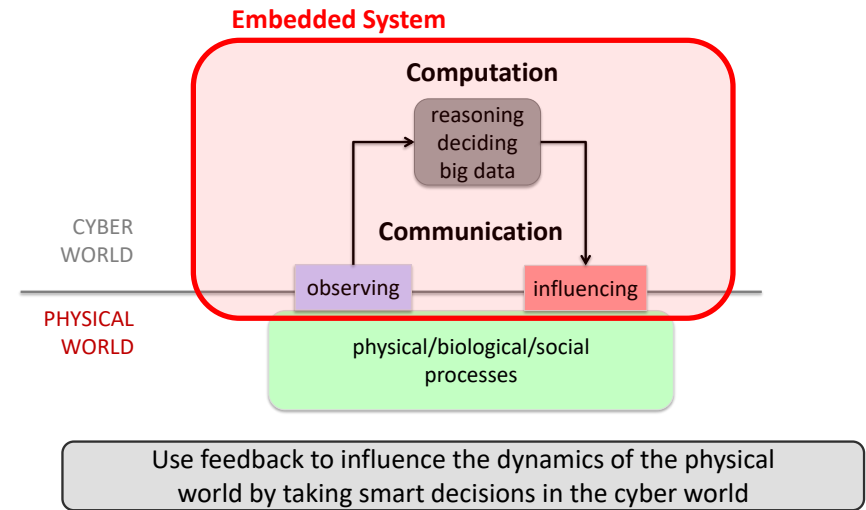
3 - 3

MSP432P401R (Last year lab - ES-Lab)



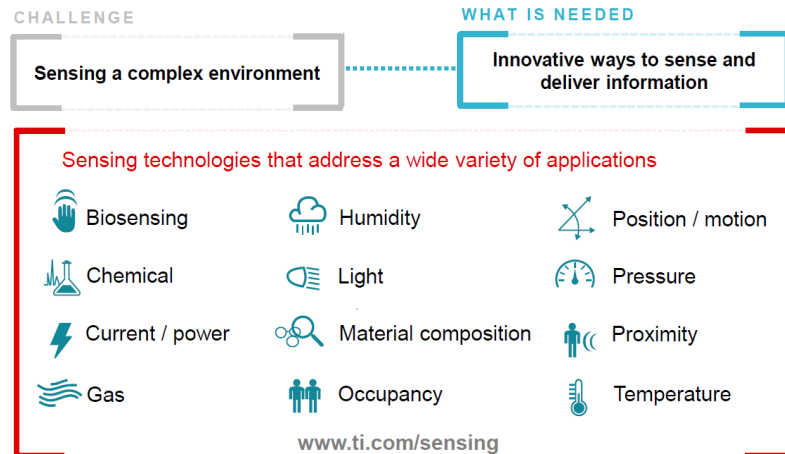
3 - 4

Today class will focus on Observing and Influencing.



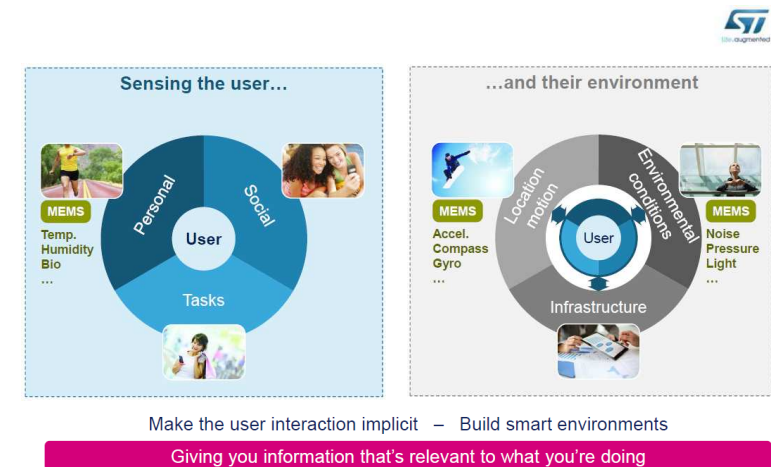
3 - 5

Sensing is a Must



3 - 6

What can they do?

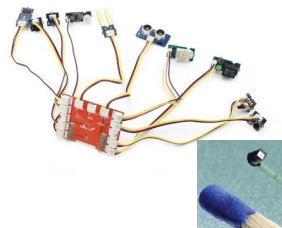


3 - 7

Sensors

- Integrated 9DoF IMU (Invensense)
- Low-power cameras (Centeye, Awaiba, SoftKinetic)
- IR sensors (LEAP, PMD)
- EF sensors (Microchip GestIC, Plessey sensors)
- ECG contactless (EPIC from Plessey) and bio sensors
- Multi-Gas sensors
- Contactless Voltage + Current
- Acoustic sensors (MEMS, Analog, Digital)

Grove Starter Kit for LaunchPad

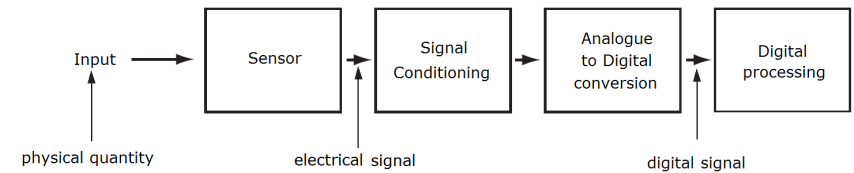


8

3 - 8

Sensors data acquisition

Data acquisition system components:



Sensors:

Convert analogue measurements of physical quantities (e.g. temperature, pressure, humidity, velocity, flow-rate, linear motion, position) into electrical signals (voltage or current).

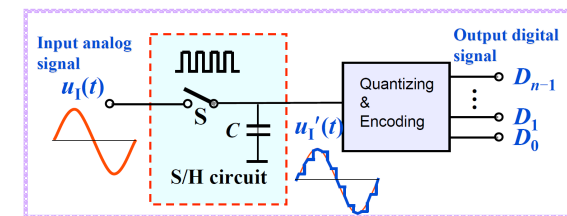
3 - 9

Data acquisition system components

- **Signal conditioning (filtering and amplification):**
 - The operations required to convert the measured analogue signal to the electrical signal range of the analogue-to-digital converter (ADC) may involve filtering, amplification, attenuation or impedance transformation.
- **Analogue-to-Digital Converter (ADC):**
 - Input: Signal to be measured;
 - Output: A digital code compatible with the digital processing system;
 - Requires:
 - Sample-and-hold: Used to take a snapshot of the continuously changing input signal and maintain the value over the sample interval set by a clock system;
 - A sampling frequency based on the Nyquist theorem.

3 - 10

ADC process



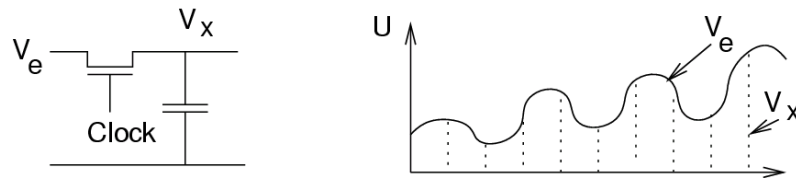
2 steps

- Sampling and Holding (S/H)
- Quantizing and Encoding (Q/E)

3 - 11

ADC conversion Sample and Hold

Convert signal function in a series of value



Restriction to digital information processing.

Known digital computers can only process discrete time series

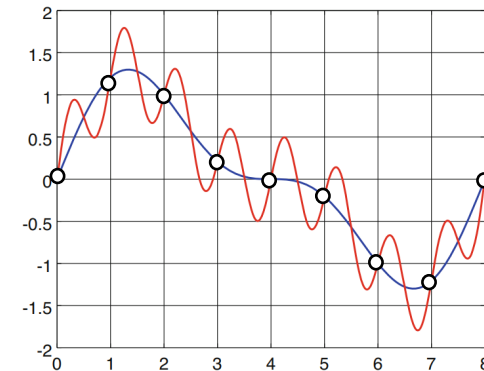
Sample and hold-devices.

Ideally: width of clock pulse $\rightarrow 0$

- *Sample and Hold circuit does the signal time quantization*

Importance of sampling Frequency

Visualization of functions $e_3(t)$ (blue) and $e_4(t)$ (red)



$$e_3(t) = \sin\left(\frac{2\pi t}{8}\right) + 0.5 \sin\left(\frac{2\pi t}{4}\right)$$

$$e_4(t) = \sin\left(\frac{2\pi t}{8}\right) + 0.5 \sin\left(\frac{2\pi t}{4}\right) + 0.5 \sin\left(\frac{2\pi t}{1}\right)$$

The fact that two or more unsampled signals can have the same sampled representation is called **aliasing**.

Sampling the signals at integer times

3 - 12

3 - 13

How can we avoid aliasing?

- Let us assume that we are sampling the input signal at constant time intervals, such that T_s is the sampling period

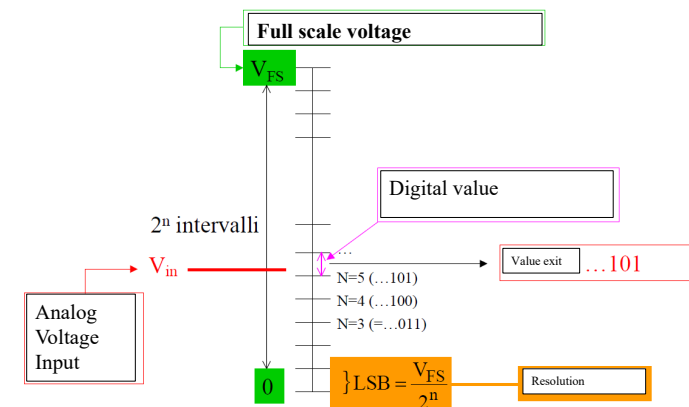
- The sampling frequency is
$$f_s = \frac{1}{T_s}$$

- Sampling Theorem: aliasing is avoided if we restrict the frequencies of the incoming signal to less than half of the sampling frequency f_s .

- Assuming a sin wave
$$T_s < \frac{T_N}{2} \text{ where } T_N \text{ is the period of the "fastest" sine wave, or}$$

$$f_s > 2f_N \text{ where } f_N \text{ is the frequency of the "fastest" sine wave}$$

Analog to Digital Conversion

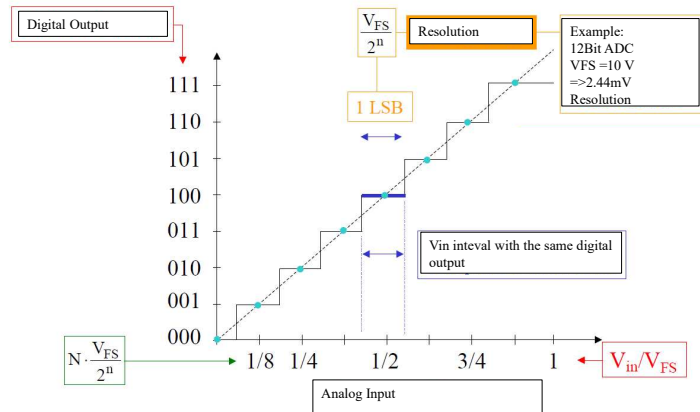


Quantizing and Encoding

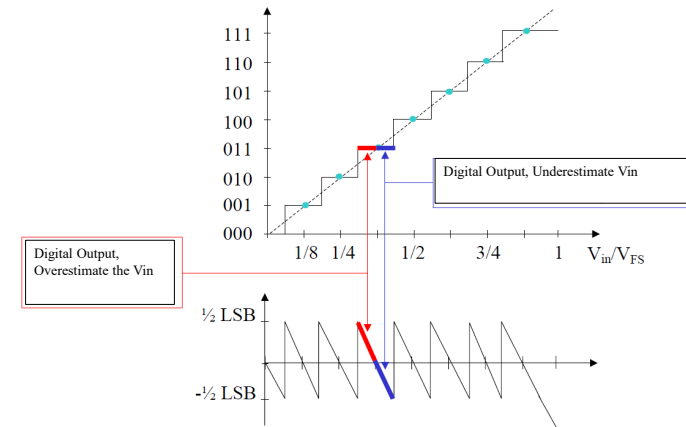
3 - 14

3 - 15

Input/Output plot – ideal ADC



Quantization error



3 - 16

3 - 17

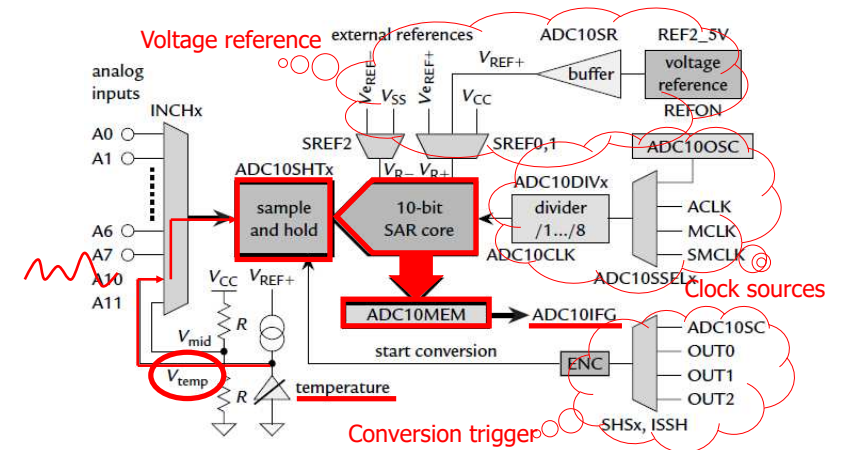
ADC performance

- Resolution, R :
 - The resolution specifies the width of the digital output word;
 - 10, 12, 16 Bit ADC
 - The width of the word implies the smallest change to the analogue voltage that can be converted into a digital code;
 - The Least Significant Bit (LSB):

$$V_{LSB} = \frac{V_{ref}}{2^n}$$

- Accuracy:
 - Degree of conformity of a digital code representing the analogue voltage.
- Speed:
 - Maximum output data rate expressed in sample per second (sps)

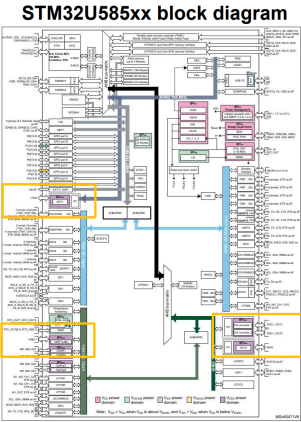
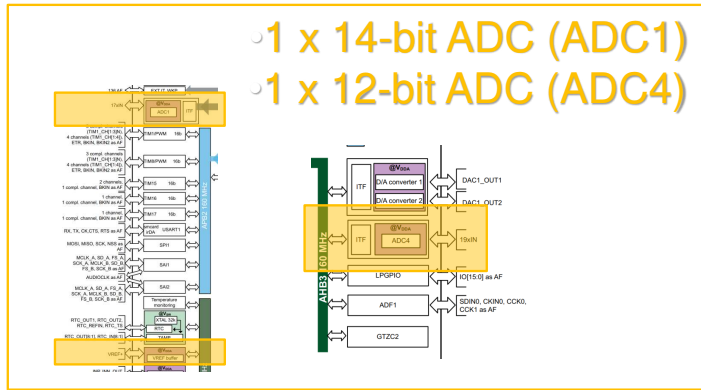
ADC Block Examples in MCUs



3 - 18

3 - 19

ADC – STM32U585xx (1)

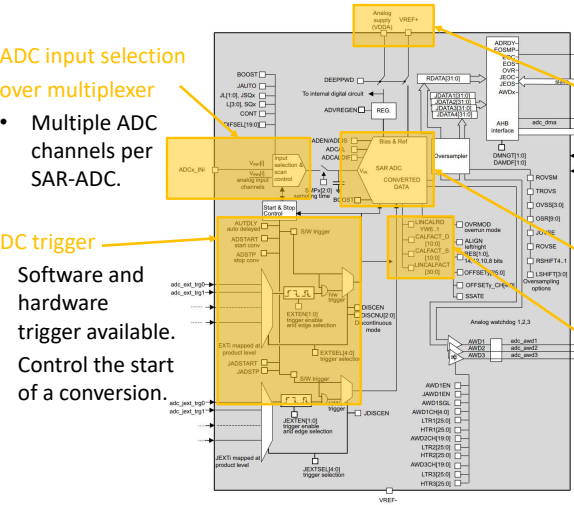


<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Datasheet, page 21

ADC – STM32U585xx (2)

	ADC1	ADC4
Resolution	14 bits	12 bits
Data register	32 bits	16 bits
Maximum sampling speed	2.5MSPS	2.5MSPS
Hardware linearity calibration	Yes	No
Differential Inputs	Yes	No
Injected channel conversion	Yes	No
Autonomous mode	No	Yes
Concurrent mode protection	-	Yes
Flexible sampling time	Yes	No
Oversampling	Up to x1024	Up to x256
Offset compensation	Yes	No
Gain compensation	Yes	No

ADC – Architecture



Analog voltage domain and reference voltage

- VDDA powers the analog domain. The voltage needs to stable for precise conversions.
- Vref builds the reference voltage for the ADC conversion.

Successive approximation ADC (SAR-ADC)

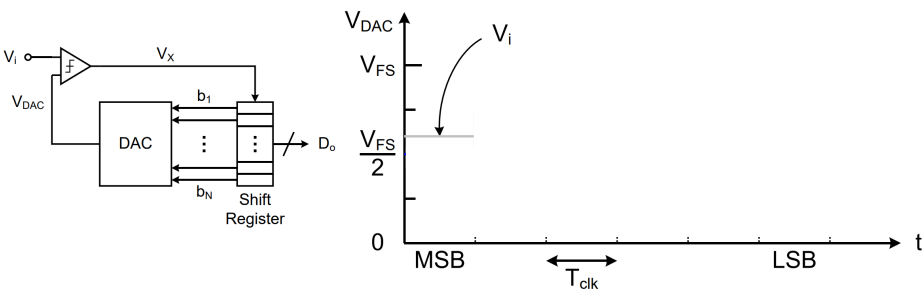
- Converts analog voltage into a digital representation (Vref dependent).

Calibration registers

- ADC needs to be calibrated for precise measurements.

<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Documentation → RM0456, page 1002

SAR-ADC – Functionality (1)

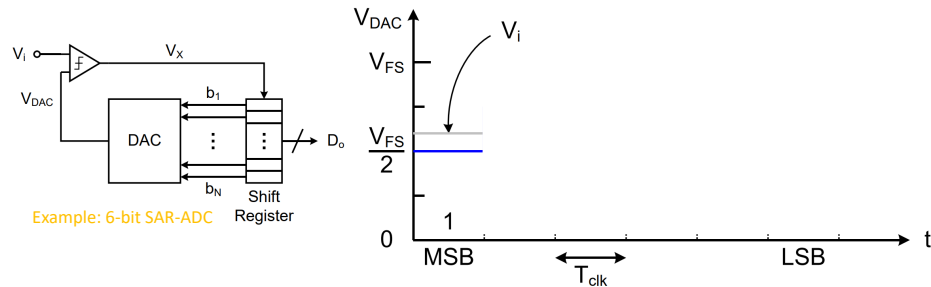


Successive approximation ADC implements the binary search algorithm

- Is built out of a comparator, a DAC and a shift register to store the conversion progress.
- Analog voltage Vi is successively approximated; conversion needs several comparisons.
- SAR-ADC needs a very high clock frequency for fast conversions (oversampling)

<https://lumerink.com/courses/ece517/s17/Lecture%20Notes/SAR%20ADC%202014.pdf>

SAR-ADC – Functionality (2)

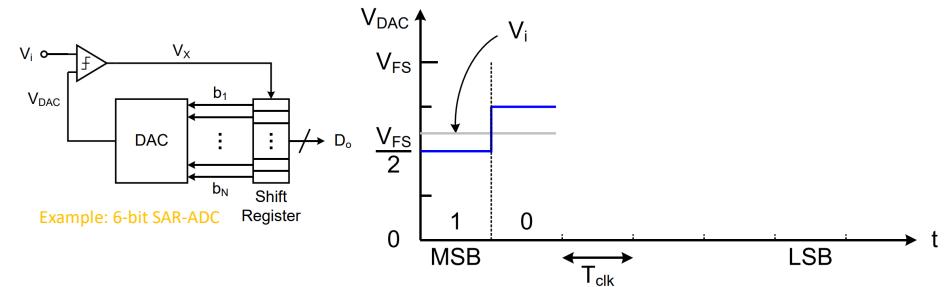


- At the beginning, V_{DAC} is set to $V_{FS}/2$
- The comparator compares V_{DAC} with the analog input: V_i is larger than V_{DAC} and V_x is high
- This leads to logic 1 in the shift register: the MSB bit is thus “1”

<https://lumerink.com/courses/ece517/s17/Lecture%20Notes/SAR%20ADC%202014.pdf>

3 - 24

SAR-ADC – Functionality (3)

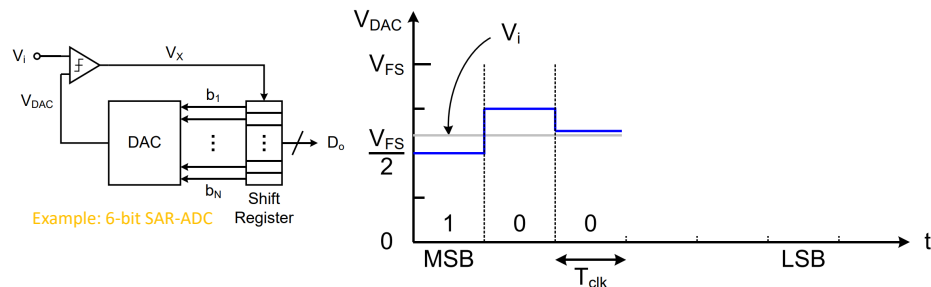


- Since the last conversion yielded logic “1”
 - V_{DAC} is set to $V_{FS}/2 + V_{FS}/4 = 3V_{FS}/4$
- The comparator compares V_{DAC} with the analog input: V_i is smaller than V_{DAC} and V_x is low
- This leads to logic 0 in the shift register: the next bit is thus “0”

<https://lumerink.com/courses/ece517/s17/Lecture%20Notes/SAR%20ADC%202014.pdf>

3 - 25

SAR-ADC – Functionality (4)

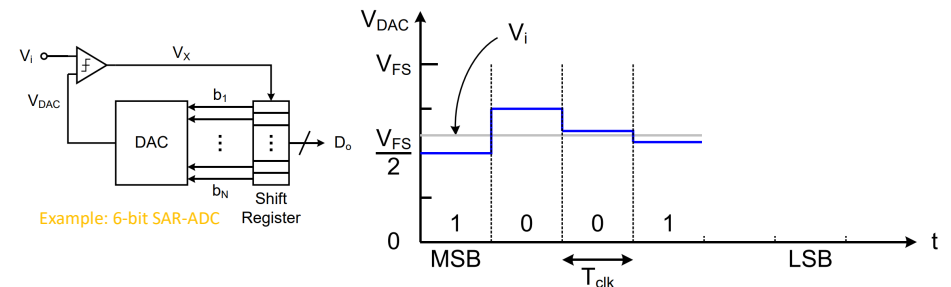


- Since the last conversion yielded logic “0”
 - V_{DAC} is set to $V_{FS}/2 + V_{FS}/4 - V_{FS}/8 = 5V_{FS}/8$
- The comparator compares V_{DAC} with the analog input: V_i is smaller than V_{DAC} and V_x is low
- This leads to logic 0 in the shift register: the next bit is thus “0”

<https://lumerink.com/courses/ece517/s17/Lecture%20Notes/SAR%20ADC%202014.pdf>

3 - 26

SAR-ADC – Functionality (5)

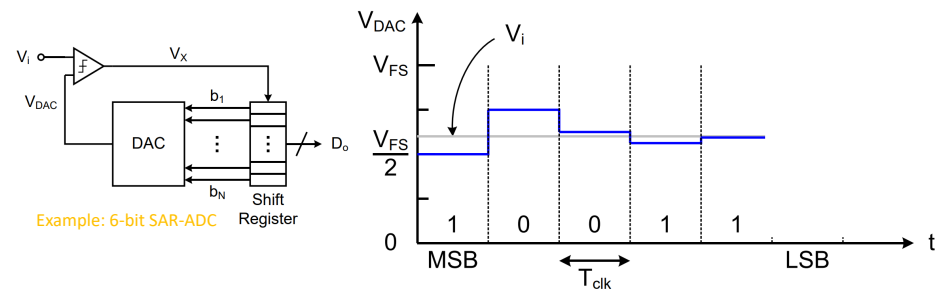


- Since the last conversion yielded logic “0”
 - V_{DAC} is set to $V_{FS}/2 + V_{FS}/4 - V_{FS}/8 + V_{FS}/16 = 9V_{FS}/16$
- The comparator compares V_{DAC} with the analog input: V_i is larger than V_{DAC} and V_x is high
- This leads to logic 1 in the shift register: the next bit is thus “1”

<https://lumerink.com/courses/ece517/s17/Lecture%20Notes/SAR%20ADC%202014.pdf>

3 - 27

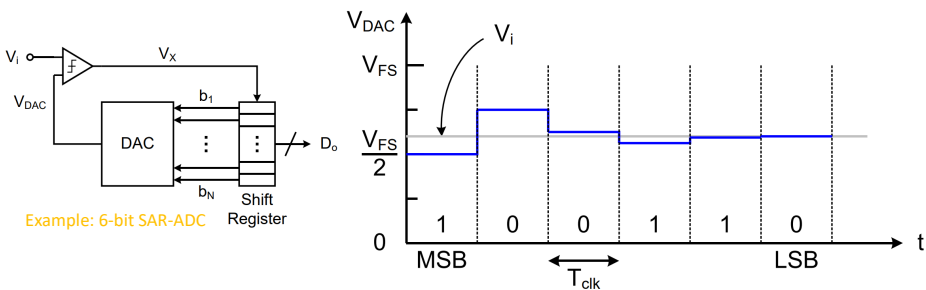
SAR-ADC – Functionality (6)



- Since the last conversion yielded logic “1”
 - V_{DAC} is set to $V_{FS}/2 + V_{FS}/4 - V_{FS}/8 - V_{FS}/16 + V_{FS}/32 = 19V_{FS}/32$
- The comparator compares V_{DAC} with the analog input: V_i is slightly larger than V_{DAC} and V_X is high
- This leads to logic 1 in the shift register: the next bit is thus “1”

<https://lumerink.com/courses/ece517/s17/Lecture%20Notes/SAR%20ADC%202014.pdf>

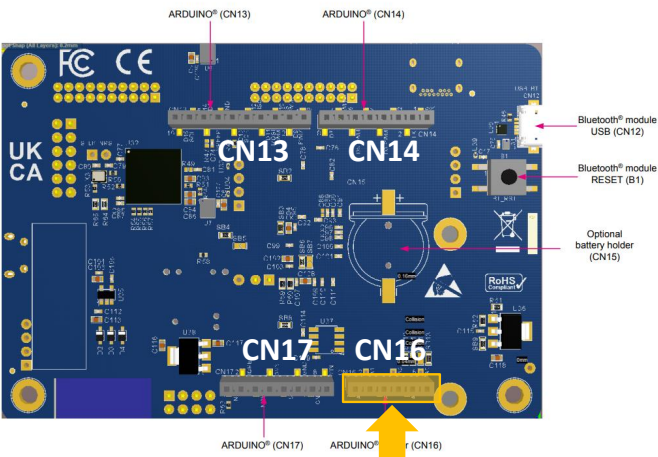
SAR-ADC – Functionality (7)



- Since the last conversion yielded logic “1”,
 - V_{DAC} is set to $V_{FS}/2 + V_{FS}/4 - V_{FS}/8 - V_{FS}/16 + V_{FS}/32 + V_{FS}/64 = 39V_{FS}/64$
- The conversion ends when the shift register is full.
 - Higher order SAR-ADCs have better precision but they need more conversion steps

<https://lumerink.com/courses/ece517/s17/Lecture%20Notes/SAR%20ADC%202014.pdf>

ADC – External Connection

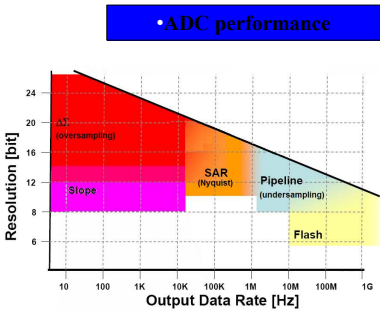


<https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html>
→ Documents → UM2839, page 34

Connector	Pin number	Pin name	Signal name	STM32L4 pin	Function
CN17	1	NC	-	-	-
	2	IOREF	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	VDD	-	-	3.3 V I/O
	5	SV	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
	8	VIN	-	-	Power input
CN13	1	A0	ARD_ADC_A0	PC0	ADC
	2	A1	ARD_ADC_A1	PC2	ADC
	3	A2	ARD_ADC_A2	PC4	ADC
	4	A3	ARD_ADC_A3	PC5	ADC
	5	A4	ARD_ADC_A4	PA7	ADC / I2C1_SDA
	6	A5	ARD_ADC_A5	PA6	ADC / I2C1_SCL
CN14	10	SCLD15	I2C1_SCL	PA6	I2C1_SCL
	9	SQAD14	I2C1_SDA	PA7	I2C1_SDA
	8	VREFP	VREFP	-	VREFP
	7	GND	GND	-	GND
	6	SDKD13	ARD_D13_SPI1_SCK	PE13	SPI1_SCK / LD2
	5	MISO12	ARD_D12_SPI1_MISO	PE14	SPI1_MISO
CN16	4	PWMMSD10	ARD_D11_SPI1_MOSI	PE15	SPI1_MOSI / TIM1_CH4
	3	PWMMSD10	ARD_D10_TIM1_SPI1_NSS	PE12	SPI1_NSS / TIM1_CH3
	2	PWMDS9	ARD_D9_TIM	PA8	TIM1_CH1
	1	D8	ARD_D8_JO	PC1	GPIO
	6	D7	ARD_D7_JO	PF13	GPIO
	7	PWMDS6	ARD_D6_TIM	PA6	TIM4_CH1
CN14	6	PWMDS5	ARD_D5_TIM	PE3	TIM4_CH1
	5	D4	ARD_D4_INT	GPIO	GPIO
	4	PWMDS3	ARD_D3_TIM	PE2	TIM8_CHN
	3	D2	ARD_D2_JO	PD15	GPIO
	2	TXD1	ARD_D1_TX	PD6	UART1_TX
	1	RXD0	ARD_D0_RX	PD5	UART1_RX

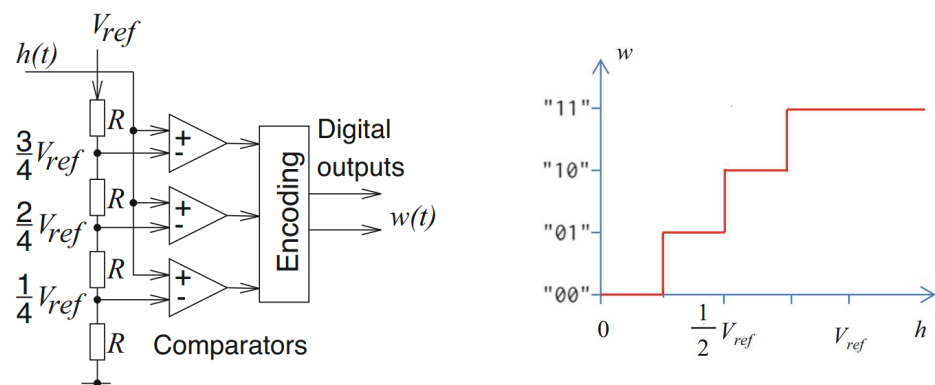
Analogue-to-Digital Converter (ADC)

- The ADC takes the voltage from the acquisition system (after signal conditioning and sampling and hold) and converts it to an equivalent digital code;



- ADC architectures:
 - Successive Approximation (SAR)
 - Sigma Delta (SD or $\Delta\Sigma$)
 - Slope or Dual Slope
 - Pipeline
 - Flash

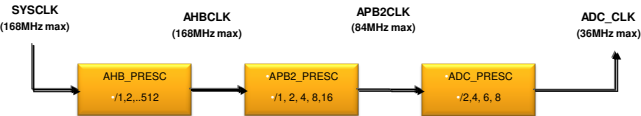
Fast ADC



ADC speed performances

AHBCLK	APB2CLK	ADC_CLK	ADC speed (15 cycles)
168MHz	(a) 84MHz	(2) 21MHz	0.714µs 1.4 Msample/s
144MHz	(a) 72MHz	(1) 36MHz	0.416µs 2.4 Msample/s
120MHz	(a) 60MHz	(1) 30MHz	0.5µs 2 Msample/s
96MHz	(a) 48MHz	(1) 24MHz	0.625µs 1.6 Msample/s
72MHz	(b) 72MHz	(1) 36MHz	0.416µs 2.4 Msample/s

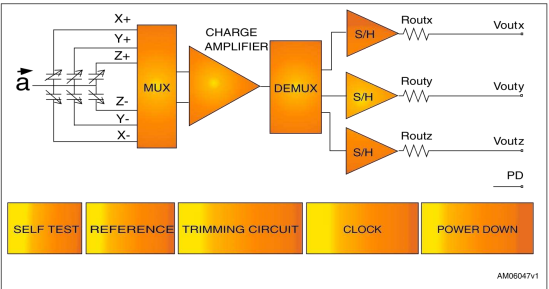
(1). ADC_PRESC = /2 (2).
ADC_PRESC = /4
APB_PRESC = /2
APB_PRESC = /1



3 - 32

3 - 33

Analog MEMS block schematic



Output voltage is proportional to measured physical quantity

Advantages:
Low cost per unit
Low power consumption
Very small footprint

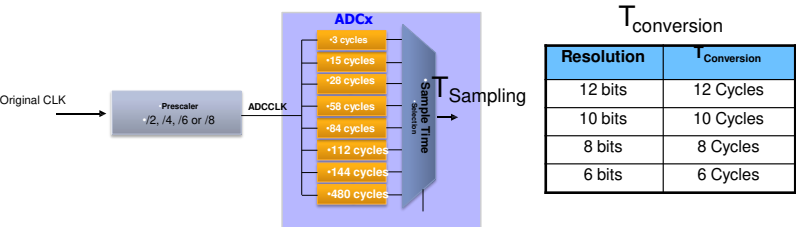
Disadvantages:
Output not always linear
Signal is not filtered
ADC must be close to sensor
Range not selectable

3 - 34

3 - 35

Total ADC Conversion Time and Max Sampling Rate

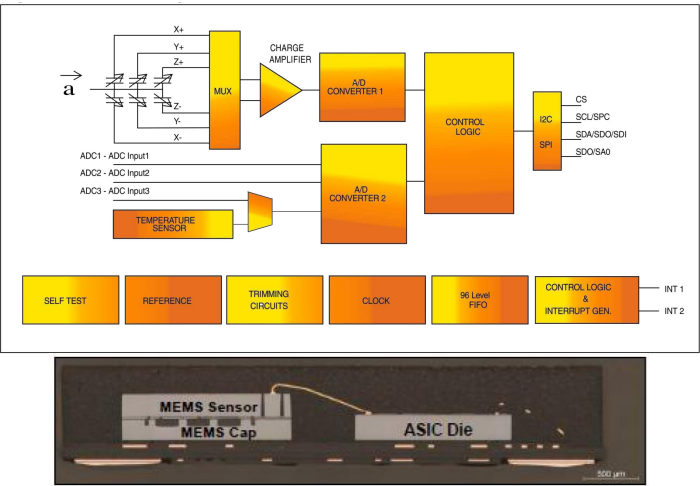
- ADCCLK, up to 36MHz, taken from PCLK through a prescaler (Div2, Div4, Div6 and Div8).
- Programmable sample time for each channel (from 4 to 480 clock cycles)
- Total conversion Time = $T_{\text{Sampling}} + T_{\text{conversion}}$



ASSUMING Sample time= 3 cycles @ ADC_CLK = 36MHz → total conversion time is equal to:

resolution	Total conversion Time	
12 bits	12 + 3 = 15cycles	0.416 us → 2.4 Msps
10 bits	10 + 3 = 13 cycles	0.361 us → 2.71 Msps
8 bits	8 + 3 = 11 cycles	0.305 us → 3.27 Msps
6 bits	6 + 3 = 9 cycles	0.25 us → 4 Msps

Digital MEMS block schematic



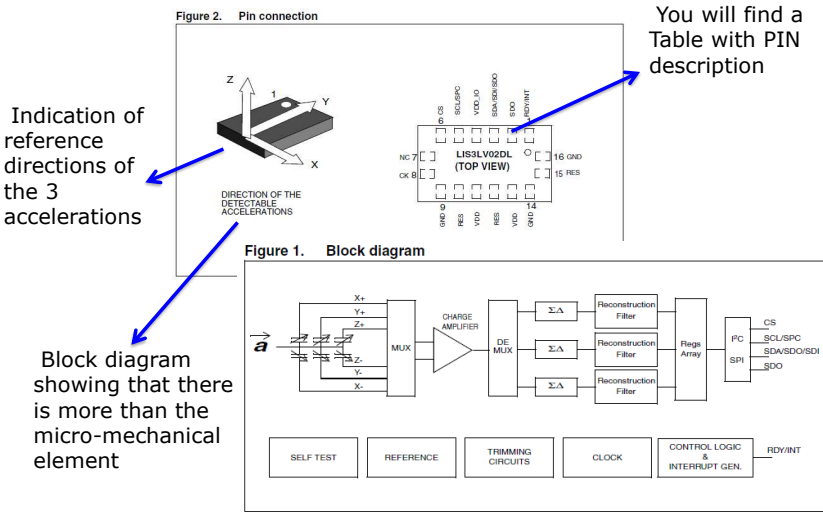
Check the electrical characteristics

Table 5. Electrical characteristics @ Vdd=3.3 V, T=25 °C unless otherwise noted (1)

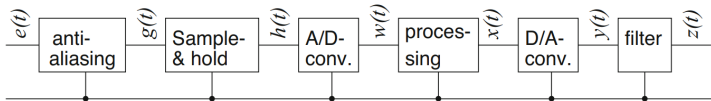
Symbol	Parameter	Test conditions	Min.	Typ.(2)	Max.	Unit
Vdd	Supply voltage		2.16	3.3	3.6	V
Vdd_IO	I/O pads supply voltage		1.71		Vdd	V
Idd	Supply current	Vdd = 3.3 V		0.65	0.80	mA
		Vdd = 2.5 V		0.60	0.75	
IddPdn	Current consumption in Power-down mode			1	10	µA
VIH	Digital High level Input voltage		0.8*Vdd_IO			V
VIL	Digital Low level Input voltage				0.2*Vdd_IO	V
VOH	High level output voltage		0.9*Vdd_IO			V
VOL	Low level output voltage				0.1*Vdd_IO	V
ODR1	Output Data Rate 1	Dec factor = 512		40		Hz
ODR2	Output Data Rate 2	Dec factor = 128		160		
ODR3	Output Data Rate 3	Dec factor = 32		640		
ODR4	Output Data Rate 4	Dec factor = 8		2560		
BW	System bandwidth(3)			ODRx/4		Hz
Ton	Turn-on time (4)			5/ODRx		s
Top	Operating temperature range		-40		+85	°C

1. The product is factory calibrated at 25 °C. The device may be used from 0 °C to 85 °C.

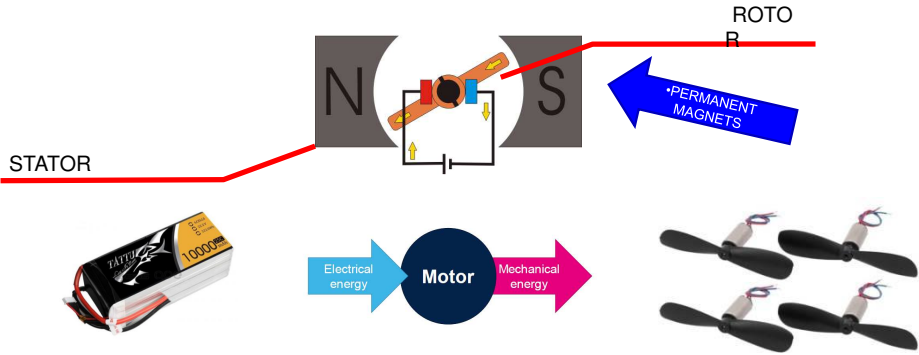
...open the [datasheet](#) 😊



The full Chain! Digital to Analog Converters



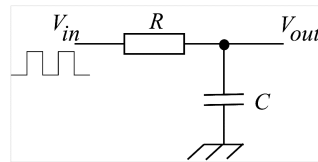
An electric motor is a device converting electrical energy into mechanical energy (generally a torque). This conversion is usually obtained through the generation of a magnetic field by means of a current flowing into one or more coils.



Pulse Width Modulation (PWM)

- The average value can be extracted from the PWM stream with a low-pass filter
- In this case, and as long as PWM frequency and values of R and C are appropriately chosen, V_{out} becomes an analog output

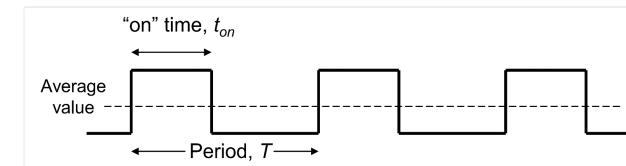
- In practice, this sort of filtering is not always required; many physical systems have response characteristics which, in reality, act like low pass filters



Pulse Width Modulation (PWM)

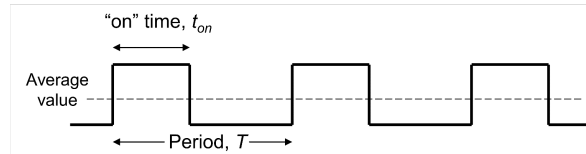
- The period is normally kept constant, and the pulse width, or “on” time is varied
- The duty cycle is the proportion of time that the pulse is ‘on’ or ‘high’, and is expressed as a percentage:

$$\text{duty cycle} = 100\% * (\text{pulse on time}) / (\text{pulse period})$$



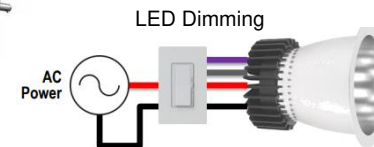
Pulse Width Modulation (PWM)

- Whatever duty cycle a PWM stream has, there is an average value, as indicated by the dotted line
- If the on time is small, the average value is low; if the on time is large, the average value is high
- By controlling the duty cycle, we control this average value

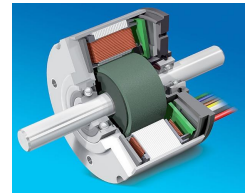


Pulse Width Modulation (PWM)

- The voltage supplied to a DC motor is proportional to the duty cycle
- Both brushed and brushless motors can be used with PWM
- Both analog and digital control techniques and components are available

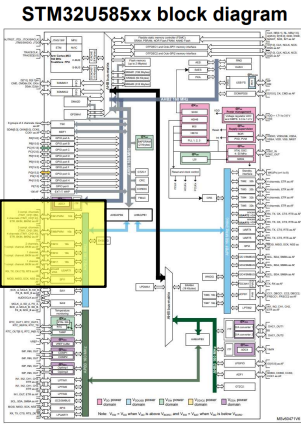
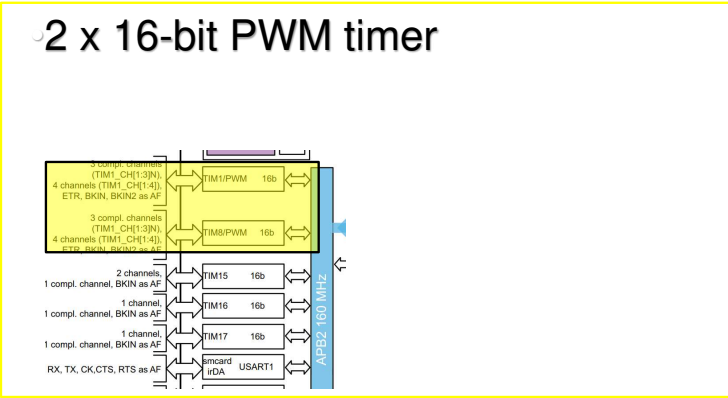


Pulse Width Modulation - Val Tocitu,



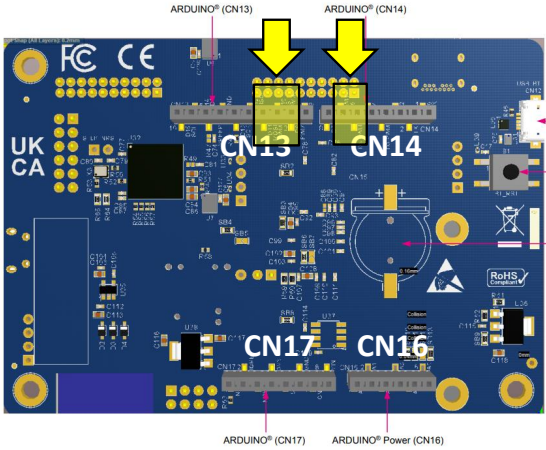
Jason Kulpe, Alex Mariuzza

PWM – STM32U585xx



<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Datasheet, page 21

PWM – External Connection



<https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html>
→ Documents → UM2839, page 34

Connector	Pin number	Pin name	Signal name	STM32L4 pin	Function
CN17	1	NC	-	-	-
	2	IOREF	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	3V3	-	-	3.3 V I/O
	5	5V	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
	8	VIN	-	-	Power input
CN16	1	A0	ARD_ADC_A0	PC0	ADC
	2	A1	ARD_ADC_A1	PC2	ADC
	3	A2	ARD_ADC_A2	PC4	ADC
	4	A3	ARD_ADC_A3	PC5	ADC
	5	A4	ARD_ADC_A4	PA7	ADC / I2C1_SDA
	6	A5	ARD_ADC_A5	PB0	ADC / I2C1_SCL
	10	SCL/I2S	I2C1_SCL	PB6	I2C1_SCL
	9	SDA/I2S	I2C1_SDA	PB6	I2C1_SDA
	8	VREFP	VREFP	-	VREFP
	7	GND	GND	-	GND
CN13	6	SDIO13	ARD_D13_SPL1_SCK	PE13	SPI1_SCK / LD2
	5	MISO/I2S	ARD_D13_SPL1_MISO	PE14	SPI1_MISO
	4	PWMACSD11	ARD_D11_SPL1_MOSI	PE15	SPI1_MOSI / TIM1_CHN1
	3	PWMACSD10	ARD_D10_TIM_SPL1_NSS	PE12	SPI1_NSS / TIM1_CHN1
	2	PWMACSD9	ARD_D9_TIM	PA5	TIM1_CH1
	1	5V	ARD_D9_5V	PA5	5V
	7	PWMACD8	ARD_D8_TIM	PE8	TIM1_CH1
	6	PWMACD5	ARD_D5_TIM	PE5	TIM1_CH1
	5	5V	ARD_D4_5V	PE7	GPI0
	4	PWMACD3	ARD_D3_TIM	PE2	TIM1_CHN1
CN14	3	D2	ARD_D2_IO	PD15	GPI0
	2	TXD0	ARD_D1_TX	PD8	UART3_TX
	1	RXD0	ARD_D0_RX	PD8	UART3_RX

What did you Learn?

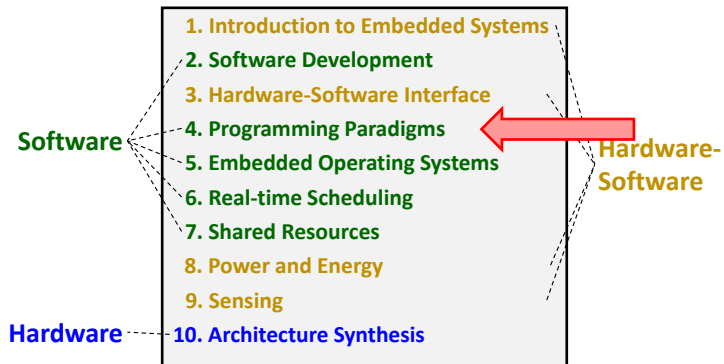
- Physical world is ANALOG
- ADC
 - Sampling and hold
 - Quantization
 - Different ADC
 - Latency and sample rate can be affected
- DAC to control other device
 - PWM
 - Example of motors

Embedded Systems

4. Programming Paradigms

© Lothar Thiele
Computer Engineering and Networks Laboratory

Where we are ...



4 - 2

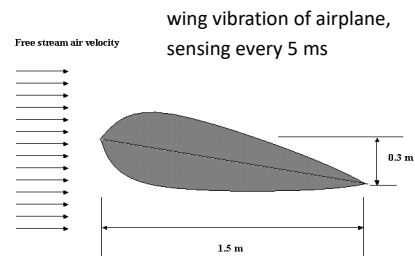
Reactive Systems and Timing

4 - 3

Timing Guarantees

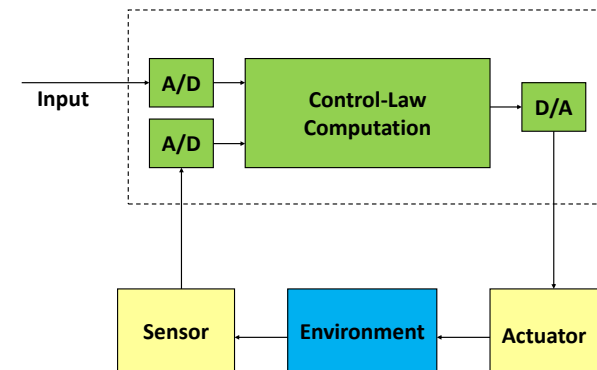
- *Hard real-time systems* can be often found in *safety-critical applications*. They need to provide the result of a computation within a fixed time bound.
- *Typical application domains:*
 - avionics, automotive, train systems, automatic control including robotics, manufacturing, media content production

sideairbag in car,
reaction after event in <10 mSec



4 - 4

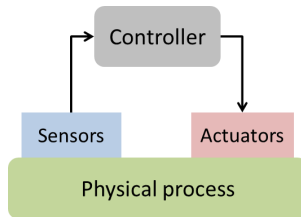
Simple Real-Time Control System



4 - 5

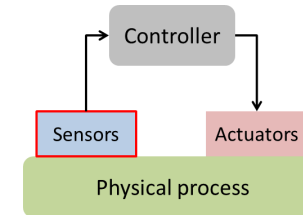
Real-Time Systems

In many *cyber-physical systems (CPSs)*, correct timing is a matter of *correctness*, not performance: *an answer arriving too late is consider to be an error.*



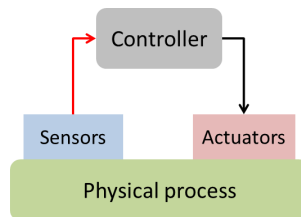
4 - 6

Real-Time Systems



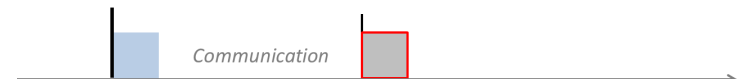
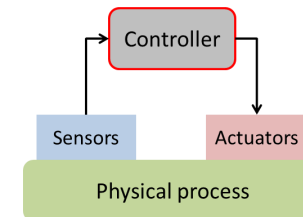
4 - 7

Real-Time Systems



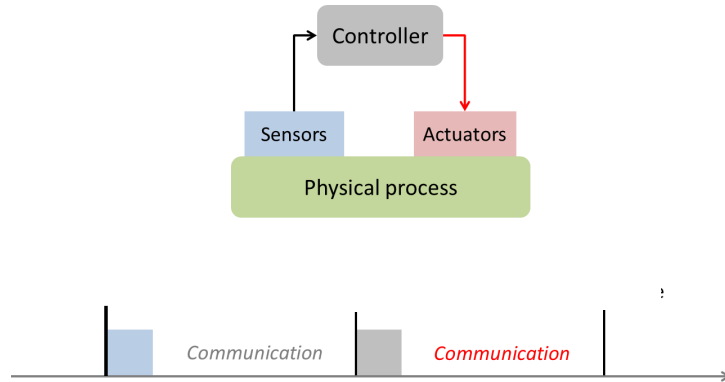
4 - 8

Real-Time Systems



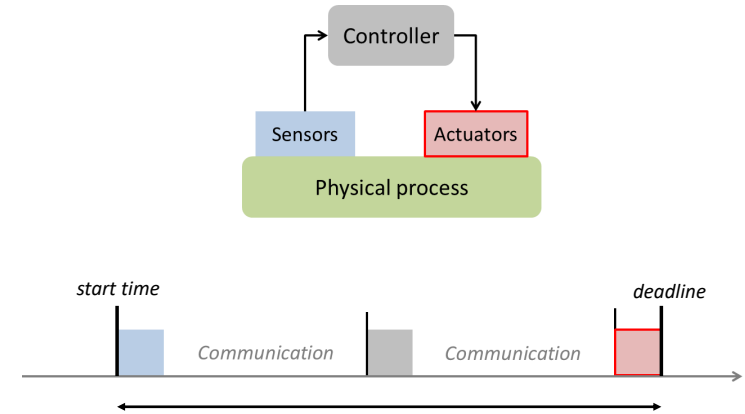
4 - 9

Real-Time Systems



4 - 10

Real-Time Systems



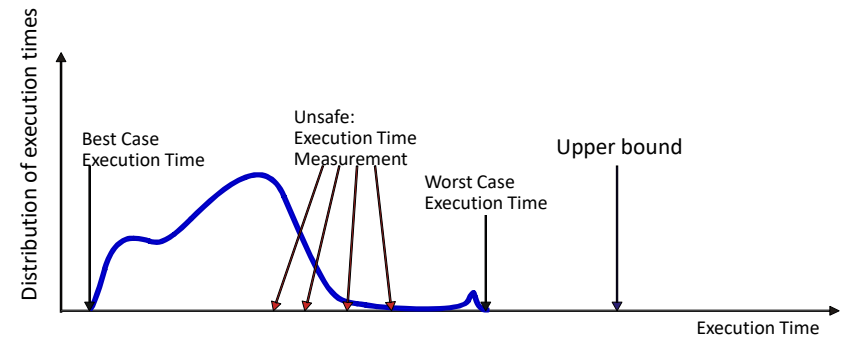
4 - 11

Real-Time Systems

- *Embedded controllers* are often expected to *finish the processing* of data and events reliably *within defined time bounds*. Such a processing may involve sequences of computations and communications.
- Essential for the analysis and design of a real-time system: *Upper bounds on the execution times* of all tasks are statically known. This also includes the communication of information via a wired or wireless connection.
 - This value is commonly called the *Worst-Case Execution Time* (WCET).
 - Analogously, one can define the lower bound on the execution time, the *Best-Case Execution Time* (BCET).

4 - 12

Distribution of Execution Times



4 - 13

Modern Hardware Features

- Modern processors *increase the average performance* (execution of tasks) by using *caches*, *pipelines*, *branch prediction*, and *speculation* techniques, for example.
- *These features make the computation of the WCET very difficult*: The execution times of single instructions vary widely.
- The microarchitecture has a large *time-varying internal state* that is changed by the execution of instructions and that influences the execution times of instructions.
 - *Best case* - everything goes smoothly: no cache miss, operands ready, needed resources free, branch correctly predicted.
 - *Worst case* - everything goes wrong: all loads miss the cache, resources needed are occupied, operands are not ready.
 - *The span between the best case and worst case may be several hundred cycles.*

4 - 14

Different Programming Paradigms

Determine the WCET

Complexity of determining the WCET of tasks:

- In the general case, it is even *undecidable* whether a finite bound exists.
- For *restricted classes of programs* it is possible, in principle. Computing accurate bounds is *simple for „old“ architectures*, but very *complex for new architectures* with pipelines, caches, interrupts, and virtual memory, for example.

Analytic (formal) approaches exist for hardware and software.

- In case of software, it requires the *analysis of the program flow* and the *analysis of the hardware* (microarchitecture). Both are combined in a complex analysis flow, see for example www.absint.de.
- *For the rest of the lecture, we assume that reliable bounds on the WCET are available*, for example by means of exhaustive measurements or simulations, or by analytic formal analysis.

4 - 15

Why Multiple Tasks on one Embedded Device?

- The concept of *concurrent tasks* reflects our intuition about the *functionality of embedded systems*.
- Tasks help us *manage the complexity of concurrent activities* as happening in the system environment:
 - *Input data* arrive from various *sensors* and input devices.
 - These input streams may have different data rates like in multimedia processing, systems with multiple sensors, automatic control of robots
 - The system may also receive *asynchronous (sporadic) input events*.
 - These input event may arrive from user interfaces, from sensors, or from communication interfaces, for example.

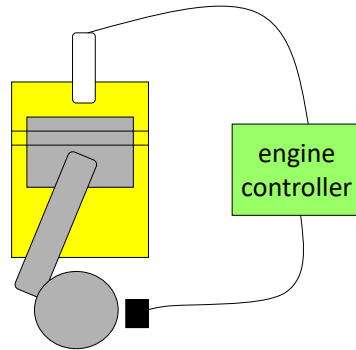
4 - 16

4 - 17

Example: Engine Control

Typical Tasks:

- spark control (actuation)
- crankshaft sensing
- fuel/air mixture (actuation)
- oxygen sensor
- control algorithm (computation)



Overview

- There are many *structured ways of programming an embedded system*.
- In this lecture, only the main principles will be covered:
 - *time triggered approaches*
 - periodic
 - cyclic executive
 - generic time-triggered scheduler
 - *event triggered approaches*
 - non-preemptive
 - preemptive – stack policy
 - preemptive – cooperative scheduling
 - preemptive - multitasking

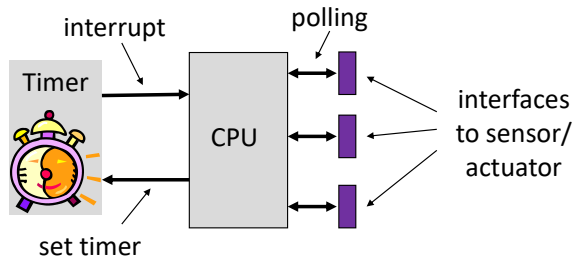
4 - 18

4 - 19

Time-Triggered Systems

Pure time-triggered model:

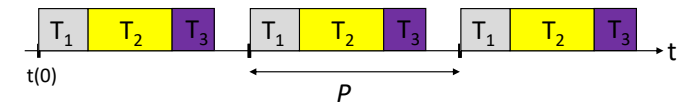
- *no interrupts* are allowed, except by timers
- the *schedule* of tasks is *computed off-line* and therefore, complex sophisticated algorithms can be used
- the scheduling at run-time is fixed and therefore, it is *deterministic*
- the interaction with environment happens through *polling*



4 - 20

Simple Periodic TT Scheduler

- A *timer interrupts regularly* with period P .
- All tasks have *same period* P .

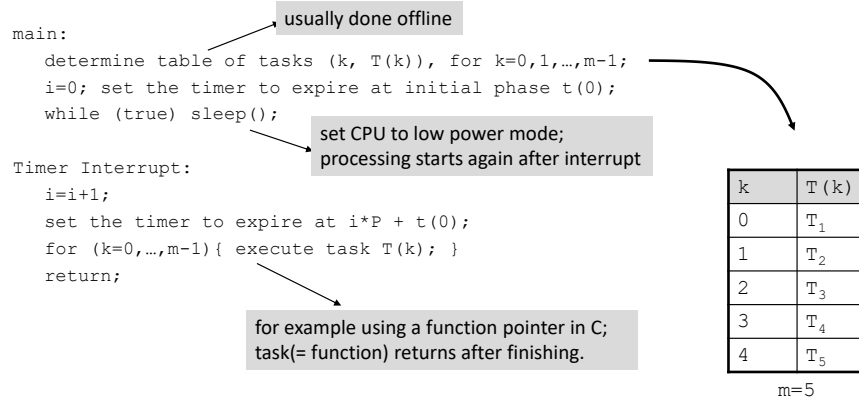


- *Properties:*
 - later tasks, for example T_2 and T_3 , have unpredictable starting times
 - the communication between tasks or the use of common resources is safe, as there is a static ordering of tasks, for example T_2 starts after finishing T_1
 - as a necessary precondition, the sum of WCETs of all tasks within a period is bounded by the period P :

$$\sum_{(k)} WCET(T_k) < P$$

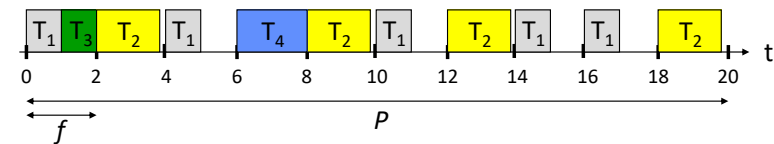
4 - 21

Simple Periodic Time-Triggered Scheduler



Time-Triggered Cyclic Executive Scheduler

- Suppose now, that *tasks may have different periods*.
- To accommodate this situation, the *period P is partitioned into frames of length f*.



- We have a *problem* to determine a feasible schedule, if there are *tasks with a long execution time*.
 - long tasks could be partitioned into a sequence of short sub-tasks
 - but this is tedious and error-prone process, as the local state of the task must be extracted and stored globally

4 - 22

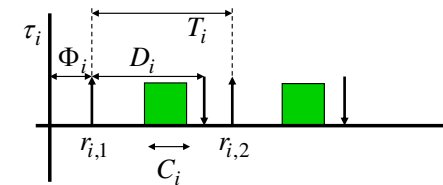
4 - 23

Time-Triggered Cyclic Executive Scheduling

- Examples for periodic tasks*: sensory data acquisition, control loops, action planning and system monitoring.
- When a control application consists of several concurrent periodic tasks with individual timing constraints, *the schedule has to guarantee* that each periodic instance is *regularly activated* at its proper rate and is *completed within its deadline*.
- Definitions*:
 - Γ : denotes the set of all periodic tasks
 - τ_i : denotes a periodic task
 - $\tau_{i,j}$: denotes the j th instance of task i
 - $r_{i,j}, d_{i,j}$: denote the release time and absolute deadline of the j th instance of task i
 - Φ_i : phase of task i (release time of its first instance)
 - D_i : relative deadline of task i

Time-Triggered Cyclic Executive Scheduling

- Example* of a single periodic task τ_i :



- A set of periodic tasks* Γ :
 - task instances should execute in these intervals

4 - 24

4 - 25

Time-Triggered Cyclic Executive Scheduling

- The following *hypotheses* are assumed on the tasks:

- The instances of a periodic task are regularly activated at a constant rate.* The interval T_i between two consecutive activations is called period. The release times satisfy

$$r_{i,j} = \Phi_i + (j-1)T_i$$

- All instances have the same worst case execution time C_i .* The worst case execution time is also denoted as $WCET(i)$.
- All instances of a periodic task have the same relative deadline D_i .* Therefore, the absolute deadlines satisfy

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

Time-Triggered Cyclic Executive Scheduling

Checking for correctness of schedule:

- f_{ij} denotes the number of the frame in which that instance j of task τ_i executes.
- Is P a common multiple of all periods T_i ?
- Is P a multiple of f ?
- Is the frame sufficiently long?

$$\sum_{\{i \mid f_{ij}=k\}} C_i \leq f \quad \forall 1 \leq k \leq \frac{P}{f}$$

- Determine offsets such that instances of tasks start after their release time:

$$\Phi_i = \min_{1 \leq j \leq P/T_i} \{(f_{ij} - 1)f - (j-1)T_i\} \quad \forall \text{ tasks } \tau_i$$

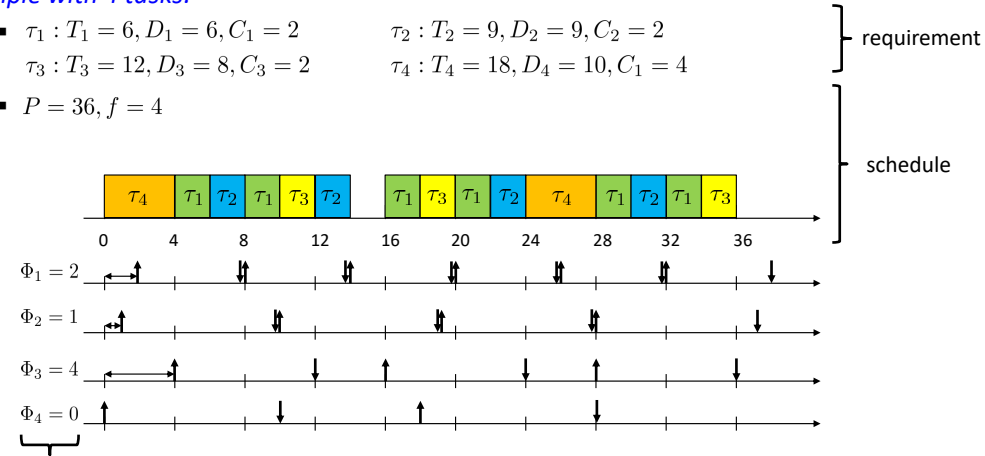
- Are deadlines respected?

$$(j-1)T_i + \Phi_i + D_i \geq f_{ij}f \quad \forall \text{ tasks } \tau_i, 1 \leq j \leq P/T_i$$

Time-Triggered Cyclic Executive Scheduling

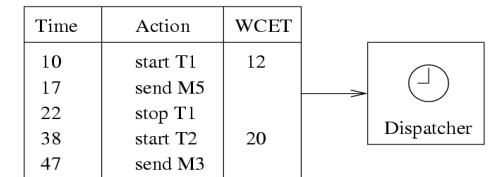
Example with 4 tasks:

- $\tau_1 : T_1 = 6, D_1 = 6, C_1 = 2$ $\tau_2 : T_2 = 9, D_2 = 9, C_2 = 2$
- $\tau_3 : T_3 = 12, D_3 = 8, C_3 = 2$ $\tau_4 : T_4 = 18, D_4 = 10, C_1 = 4$
- $P = 36, f = 4$



Generic Time-Triggered Scheduler

- In an *entirely time-triggered system*, the temporal control structure of all tasks is established a priori by off-line support-tools.
- This *temporal control structure is encoded in a Task-Descriptor List (TDL)* that contains the cyclic schedule for all activities of the node.
- This *schedule* considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary.
- The dispatcher is activated by a synchronized clock tick.* It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].



Simplified Time-Triggered Scheduler

```

main:
    determine static schedule (t(k), T(k)), for k=0,1,...,n-1;
    determine period of the schedule P;
    set i=k=0 initially; set the timer to expire at t(0);
    while (true) sleep();
  
```

usually done offline

Timer Interrupt:

```

k_old := k;
i := i+1; k := i mod n;
set the timer to expire at [i/n] * P + t(k);
execute task T(k_old);
return;
  
```

set CPU to low power mode;
processing continues after interrupt

for example using a function pointer in C;
task returns after finishing.

k	t(k)	T(k)
0	0	T ₁
1	3	T ₂
2	7	T ₁
3	8	T ₃
4	12	T ₂

n=5, P = 16

4 - 30

Summary Time-Triggered Scheduler

Properties:

- **deterministic schedule**; conceptually simple (static table); relatively easy to validate, test and certify
- **no problems** in using **shared resources**
- external communication only via **polling**
- **inflexible** as no adaptation to the environment
- serious **problems** if there are **long tasks**

Extensions:

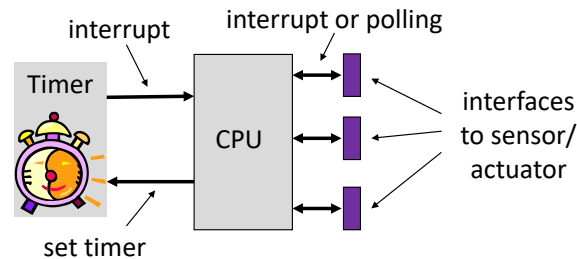
- **allow interrupts** → be careful with shared resources and the WCET of tasks!!
- **allow preemptable** background tasks
- **check for task overruns** (execution time longer than WCET)

4 - 31

Event Triggered Systems

The schedule of tasks is determined by the occurrence of external or internal events:

- **dynamic and adaptive**: there are possible problems with respect to timing, the use of shared resources and buffer over- or underflow
- **guarantees** can be given either off-line (if bounds on the behavior of the environment are known) or during run-time



4 - 32

Non-Preemptive Event-Triggered Scheduling

Principle:

- To each event, there is associated a corresponding task that will be executed.
- Events are emitted by (a) external interrupts or (b) by tasks themselves.
- All events are collected in a single queue; depending on the queuing discipline, an event is chosen for execution, i.e., the corresponding task is executed.
- Tasks can not be preempted.

Extensions:

- A **background task** can run if the event queue is empty. It will be preempted by any event processing.
- **Timed events** are ready for execution only after a time interval elapsed. This enables periodic instantiations, for example.

4 - 33

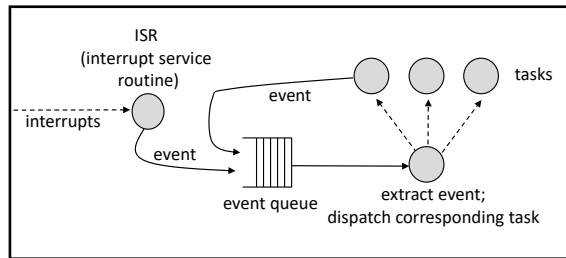
Non-Preemptive Event-Triggered Scheduling

```
main:
  while (true) {
    if (event queue is empty) {
      sleep();
    } else {
      extract event from event queue;
      execute task corresponding to event;
    }
  }
```

set the CPU to low power mode;
continue processing after interrupt

for example using a function pointer in C;
task returns after finishing.

```
Interrupt:
  put event into event queue;
  return;
```

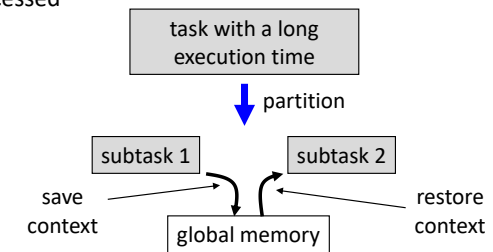


4 - 34

Non-Preemptive Event-Triggered Scheduling

Properties:

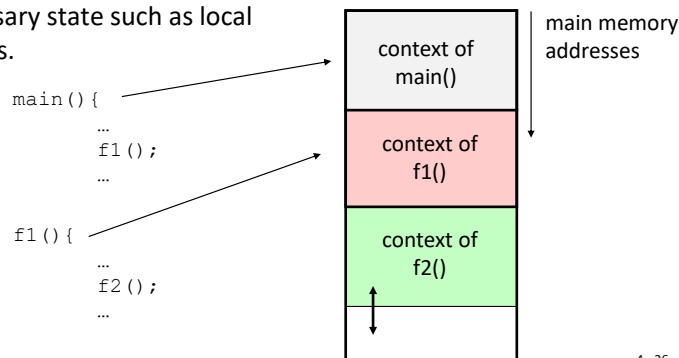
- **communication between tasks** does not lead to a simultaneous access to shared resources, but interrupts may cause problems as they preempt running tasks
- **buffer overflow** may happen if too many events are generated by the environment or by tasks
- **tasks with a long running time** prevent other tasks from running and may cause buffer overflow as no events are being processed during this time
 - partition tasks into smaller ones
 - but the local context must be stored



4 - 35

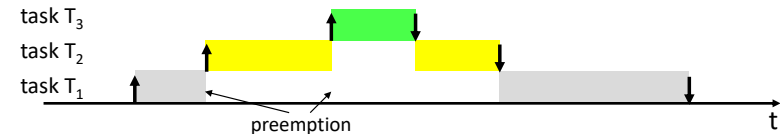
Preemptive Event-Triggered Scheduling – Stack Policy

- This case is similar to non-preemptive case, but **tasks can be preempted by others**; this resolves partly the problem of tasks with a long execution time.
- If **the order of preemption is restricted**, we can use the usual stack-based context mechanism of function calls. The context of a function contains the necessary state such as local variables and saved registers.



4 - 36

Preemptive Event-Triggered Scheduling – Stack Policy



- **Tasks must finish in LIFO (last in first out) order** of their instantiation.
 - this restricts flexibility of the approach
 - it is not useful, if tasks wait some unknown time for external events, i.e., they are blocked
- **Shared resources** (communication between tasks!) **must be protected**, for example by disabling interrupts or by the use of semaphores.

4 - 37

Thread

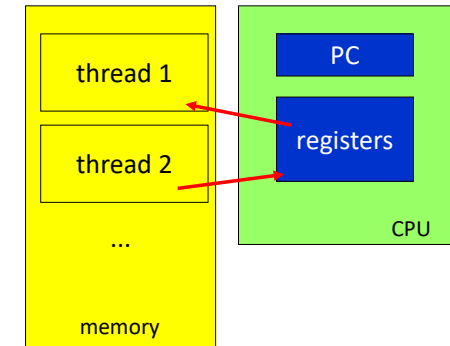
- *A thread is a unique execution of a program.*
 - Several copies of such a “program” may run simultaneously or at different times.
 - Threads share the same processor and its peripherals.
- *A thread has its own local state.* This state consists mainly of:
 - register values;
 - memory stack (local variables);
 - program counter;
- *Several threads may have a shared state* consisting of global variables.

Threads and Memory Organization

- *Activation record* (also denoted as the thread context) contains the thread local state which includes registers and local data structures.

- *Context switch:*

- current CPU context goes out
- new CPU context goes in



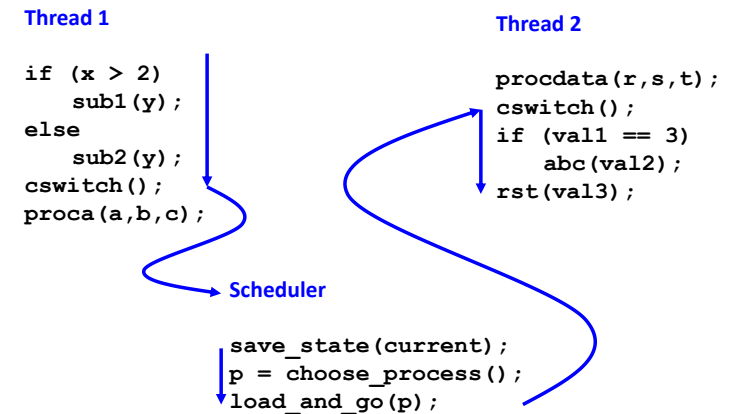
4 - 39

4 - 40

Co-operative Multitasking

- *Each thread allows a context switch to another thread* at a call to the `cswitch()` function.
 - This function is part of the underlying runtime system (operating system).
 - A *scheduler* within this runtime system chooses which thread will run next.
- *Advantages:*
 - predictable, where context switches can occur
 - less errors with use of shared resources if the switch locations are chosen carefully
- *Problems:*
 - programming errors can keep other threads out as a thread may never give up CPU
 - real-time behavior may be at risk if a thread runs too long before the next context switch is allowed

Example: Co-operative Multitasking



4 - 41

4 - 42

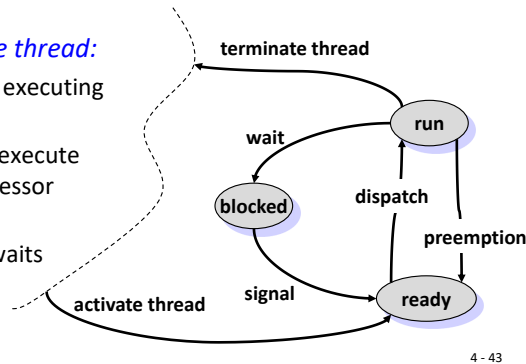
Preemptive Multitasking

- *Most general form of multitasking:*

- The scheduler in the runtime system (operating system) controls when contexts switches take place.
- The scheduler also determines what thread runs next.

- *State diagram corresponding to each single thread:*

- **Run:** A thread enters this state as it starts executing on the processor
- **Ready:** State of threads that are ready to execute but cannot be executed because the processor is assigned to another thread.
- **Blocked:** A task enters this state when it waits for an event.



Embedded Systems

5. Operating Systems

© Lothar Thiele

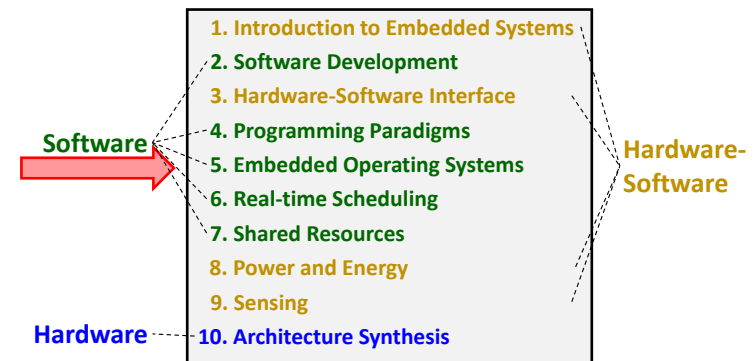
Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Where we are ...

Embedded Operating Systems



Embedded Operating Systems

Features and Architecture

Embedded Operating System

- *Why is a desktop OS not suited?*
 - The monolithic kernel of a desktop OS offers too many features that take space in memory and consume time.
 - Monolithic kernels are often not modular, fault-tolerant, configurable.
 - Requires too much memory space and is often too resource hungry in terms of computation time.
 - Not designed for mission-critical applications.
 - The timing uncertainty may be too large for some applications.

5 - 4

Main Functionality of an Embedded Operating System

Task management:

- *Execution of quasi-parallel tasks* on a processor using processes or threads (lightweight process) by
 - maintaining process states, process queuing,
 - allowing for preemptive tasks (fast context switching) and quick interrupt handling
- *CPU scheduling* (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
- *Inter-task communication* (buffering)
- *Support of real-time clocks*
- *Task synchronization* (critical sections, semaphores, monitors, mutual exclusion)
 - In classical operating systems, synchronization and mutual exclusion is performed via semaphores and monitors.
 - In real-time OS, special semaphores and a deep integration of them into scheduling is necessary (for example priority inheritance protocols as described in a later chapter).

5 - 5

Embedded Operating Systems

One Essential characteristic of an embedded OS: Configurability

- *No single operating system will fit all needs*, but often no overhead for unused functions/data is tolerated. Therefore, configurability is needed.
- For example, there are many embedded systems without external memory, a keyboard, a screen or a mouse.

Configurability examples:

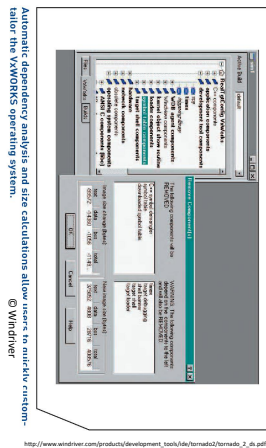
- *Remove unused functions*/libraries (for example by the linker).
- *Use conditional compilation* (using #if and #ifdef commands in C, for example).
- But deriving a consistent configuration is a potential problem of systems with a large number of derived operating systems. There is the danger of missing relevant components.

5 - 6

5 - 7

Embedded Operating Systems Task Management

Example: Configuration of VxWorks



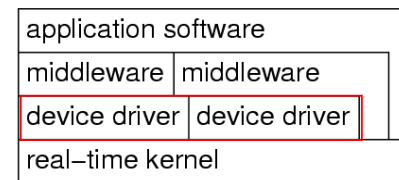
5 - 8

Embedded Operating System vs. Standard OS

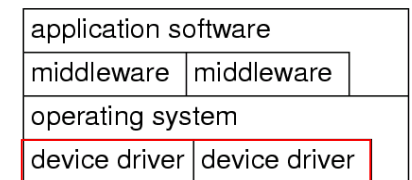
Device drivers are typically handled directly by tasks instead of drivers that are managed by the operating system:

- This architecture *improves timing predictability* as access to devices is also handled by the scheduler
- If several tasks use the same external device and the associated driver, then the access must be carefully managed (shared critical resource, ensure fairness of access)

Embedded OS



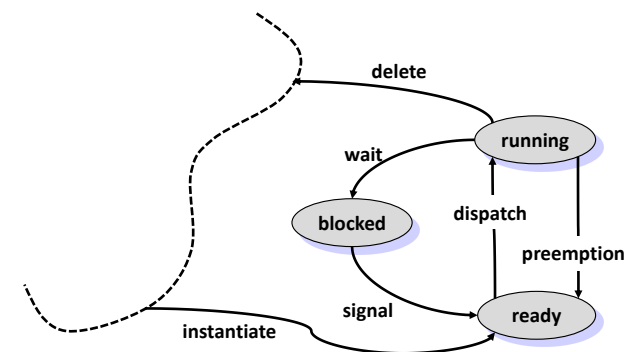
Standard OS



5 - 9

Task States

Minimal Set of Task States:



5 - 10

5 - 11

Task states

Running:

- A task enters this state when it starts executing on the processor. There is at most one task with this state in the system.

Ready:

- State of those tasks that are ready to execute but cannot be run because the processor is assigned to another task, i.e. another task has the state "running".

Blocked:

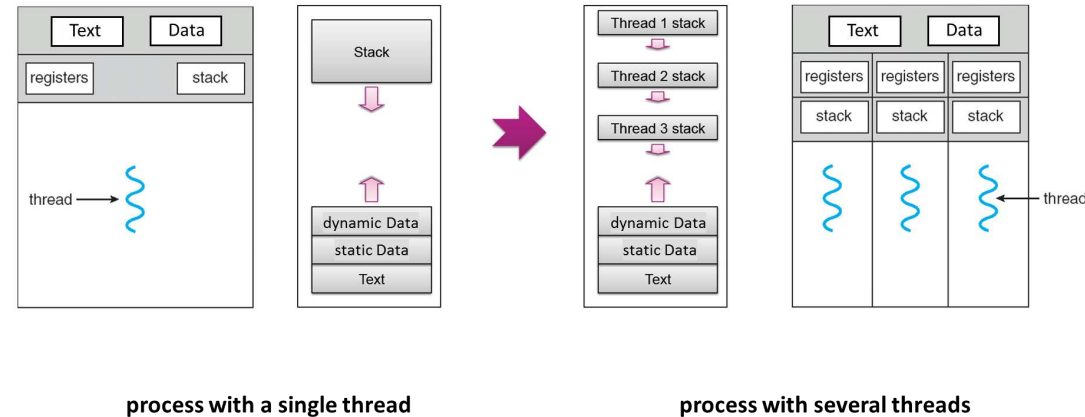
- A task enters the blocked state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore or timer. In this case, the task is inserted in a queue associated with this semaphore. The task at the head is resumed when the semaphore is unlocked by an event.

Threads

A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler; e.g., a thread is a basic unit of CPU utilization.

- Multiple threads can exist within the same process* and share resources such as memory, while different processes do not share these resources:
 - Typically shared by threads: memory.
 - Typically owned by threads: registers, stack.
- Thread advantages and characteristics:*
 - Faster to switch between threads; switching between user-level threads requires no major intervention by the operating system.
 - Typically, an application will have a separate thread for each distinct activity.
 - Thread Control Block (TCB) stores information needed to manage and schedule a thread

Multiple Threads within a Process

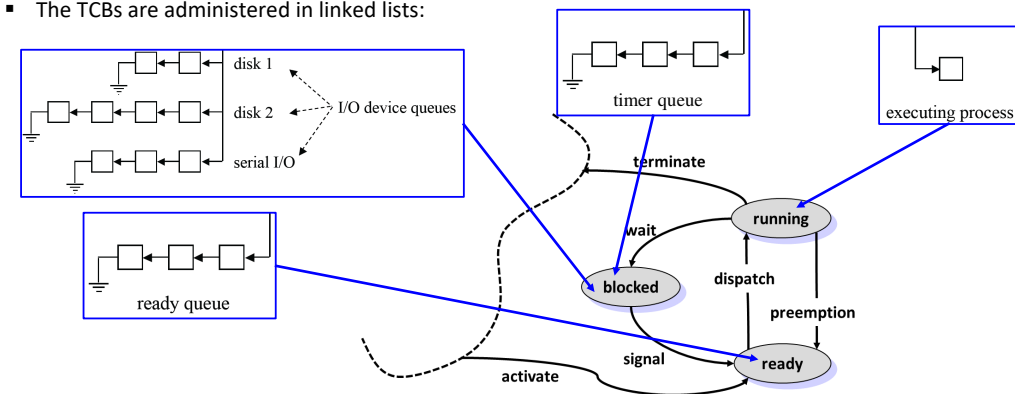


5 - 12

5 - 13

Threads

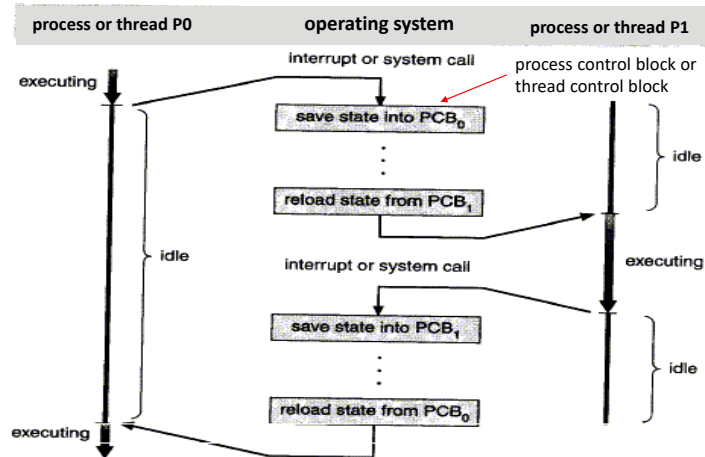
- The *operating system* maintains for each thread a *data structure (TCB – thread control block)* that contains its current status such as program counter, priority, state, scheduling information, thread name.
- The TCBs are administered in linked lists:



5 - 14

5 - 15

Context Switch: Processes or Threads



5 - 16

Embedded Operating Systems Classes of Operating Systems

5 - 17

Class 1: Fast and Efficient Kernels

Fast and efficient kernels

For hard real-time systems, these kernels are questionable, because they are designed to be fast, rather than to be predictable in every respect.

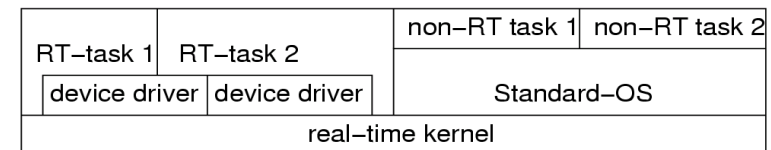
Examples include

ThreadX, FreeRTOS, QNX, eCOS, RT-LINUX, VxWORKS, LynxOS.

Class 2: Extensions to Standard OSs

Real-time extensions to standard OS:

- Attempt to exploit existing and comfortable main stream operating systems.
- A real-time kernel runs all real-time tasks.
- The standard-OS is executed as one task.



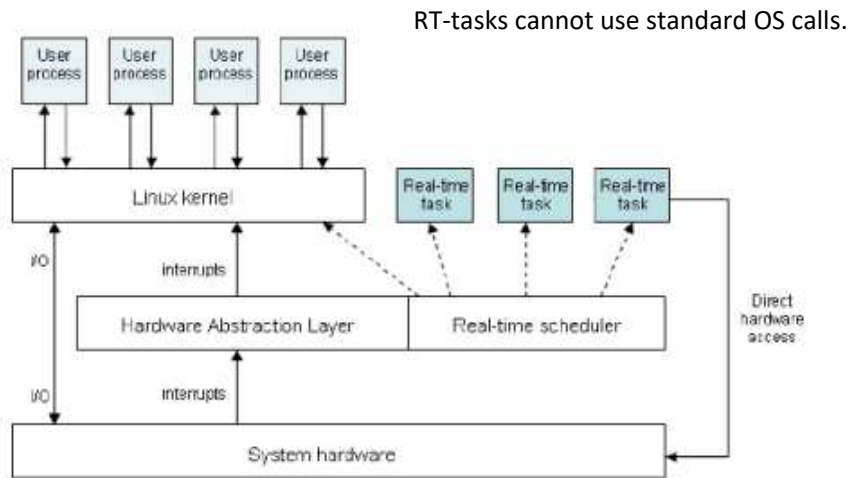
+ Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services;
less comfortable than expected

revival of the concept:
hypervisor

5 - 18

5 - 19

Example: RT Linux



5 - 20

Embedded Operating Systems

ThreadX in the Embedded Systems Lab (ES-Lab)

Class 3: Research Systems

Research systems try to avoid limitations of existing real-time and embedded operating systems.

- Examples include L4, seL4, NICTA, ERIKA, SHARK

Typical Research questions:

- low overhead memory protection,
- temporal protection of computing resources
- RTOS for on-chip multiprocessors
- quality of service (QoS) control (besides real-time constraints)
- formally verified kernel properties

List of current real-time operating systems:

http://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

5 - 21

Example: ThreadX (ES-Lab)

Azure RTOS ThreadX (<https://azure.microsoft.com/en-us/products/rtos>) is an embedded operating system. It is available for many hardware platforms, open source and widely used in industry. It is used in the ES-Lab.

- ThreadX is a **high-performance real-time kernel**.
- Applications are organized as a **collection of independent threads** of execution.
- **Characteristics:** Preemptive and cooperative scheduling, flexible thread priority support, priority inheritance, automatic scaling, deterministic processing, run-time performance monitoring, ...
- ThreadX provides the FreeRTOS/POSIX adaptation layers

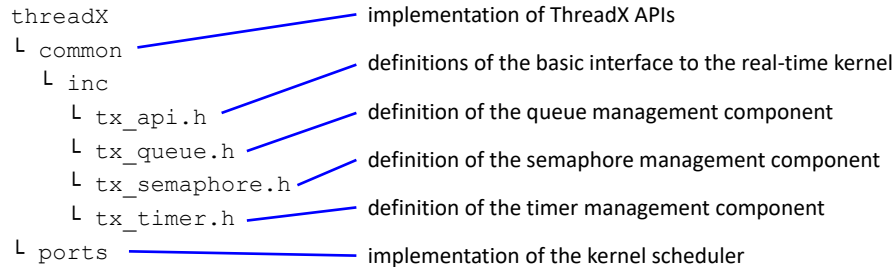


5 - 22

5 - 23

Example: ThreadX (ES-Lab)

Typical directory structure (excerpts):



- *ThreadX can be configured* by a header file called `tx_user.h` that includes almost all configuration options (preemption, priority, stack size, interrupt,...)

5 - 24

Example ThreadX – Task Management

Tasks are implemented as threads.

- The *functionality of a thread* is implemented in form of a *function*:
 - Prototype: `void thread_function(ULONG thread_input);`
 - some name of task function
 - pointer to task arguments
 - Task functions can instantiate other tasks. Each created task is a separate execution instance, with its own stack.
 - *Example*

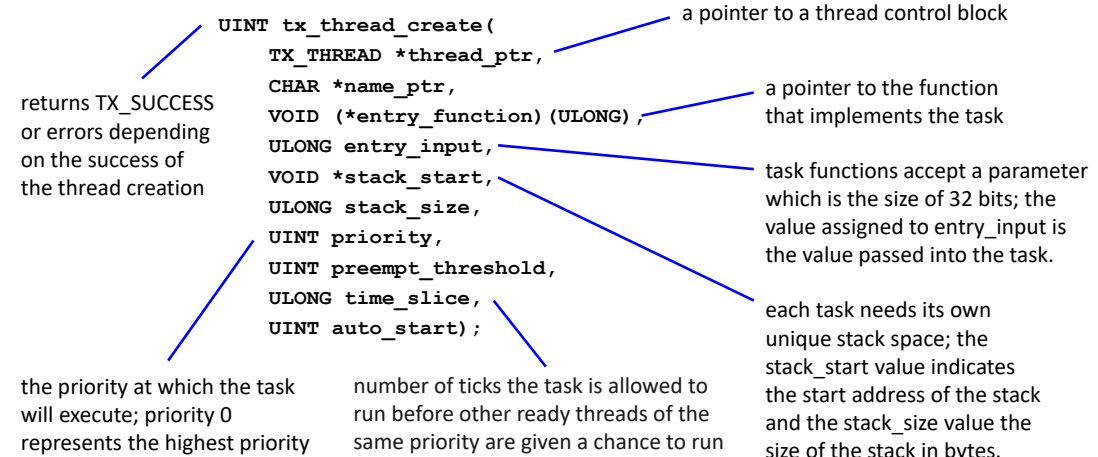
```
void task1_entry( ULONG thread_input ) {
    volatile uint32_t ul; /* volatile to ensure ul is implemented. */
    for( ;; ) {
        ... /* do something repeatedly */
        for( ul = 0; ul < 10000; ul++ ) { /* delay by busy loop */ }
    }
}
```

5 - 26

Embedded Operating Systems ThreadX Task Management

Example ThreadX – Task Management

- *Thread instantiation:*



5 - 25

5 - 27

Example ThreadX – Task Management

Examples for changing properties of tasks:

- Changing the *priority* of a task. In case of preemptive scheduling policy, the ready task with the highest priority is automatically assigned to the “executing” state.

```
UINT tx_thread_priority_change(TX_THREAD *thread_ptr,  
                               UINT new_priority, UINT *old_priority);
```

pointer to a previously created application thread new priority (0 is highest priority) pointer to store the old priority

- A task can *delete* itself or any other task. Deleted tasks no longer exist and cannot enter the “executing” state again.

```
UINT tx_thread_delete(TX_THREAD *thread_ptr);
```

pointer to a previously created application thread

5 - 28

Embedded Operating Systems ThreadX Timers

5 - 29

Example ThreadX– Timers

- The operating system also provides *interfaces to timers* of the processor.
- As an example, we use the ThreadX timer interface to replace the busy loop by a delay. In this case, the task is put into the “suspended” state instead of continuously executing.

```
UINT tx_thread_sleep(ULONG timer_ticks);
```

The macro `TX_TIMER_TICKS_PER_SECOND`, which denotes the number of timer ticks per second, is defined in `tx_api.h` and can be redefined in `tx_user.h`. It is used to convert seconds to “ticks”.

```
void task1_entry( ULONG thread_input ) {  
    for( ;; ) {  
        ... /* do something repeatedly */  
        tx_thread_sleep(2 * TX_TIMER_TICKS_PER_SECOND); /* delay by 2s */  
    }  
}
```

5 - 30

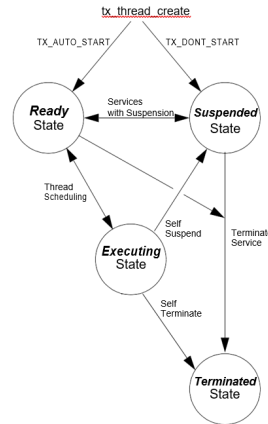
Embedded Operating Systems ThreadX Task States

5 - 31

Example ThreadX – Task States

What are the task states in ThreadX and the corresponding transitions?

- A task that is waiting for an event is said to be in the “suspended” state.
- Tasks can enter the “suspended” state to wait for mainly two different types of event:
 - **Temporal (time-related) events**—the event being either a delay period expiring, or an absolute time being reached.
 - **Synchronization events**—where the events originate from another task or interrupt. For example, queues, semaphores, and mutexes, can be used to create synchronization events.



5 - 32

Example ThreadX – Task States

Example: Two threads with equal priority.

```

void task1_entry( ULONG thread_input ) {
    volatile uint32_t u1;
    for( ;; ) {
        ... /* do something repeatedly */
        for( u1 = 0; u1 < 10000; u1++ ) { }
    }
}
    
```

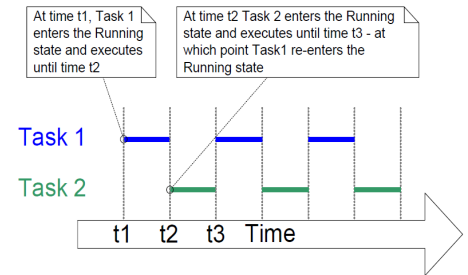
```

void task2_entry( ULONG thread_input ) {
    volatile uint32_t u2;
    for( ;; ) {
        ... /* do something repeatedly */
        for( u2 = 0; u2 < 10000; u2++ ) { }
    }
}
    
```

```

int main( void ) {
    ...
    tx_thread_create(&task1, "Task 1",
        task1_entry, NULL, (VOID*) 0x4000, 0x1000,
        1, 1, 100, TX_AUTO_START);
    tx_thread_create(&task2, "Task 2",
        task2_entry, NULL, (VOID*) 0x5000, 0x1000,
        1, 1, 100, TX_AUTO_START);
    ...
}
    
```

priority time_slice



5 - 33

Embedded Operating Systems

ThreadX Interrupts

Example ThreadX – Interrupts

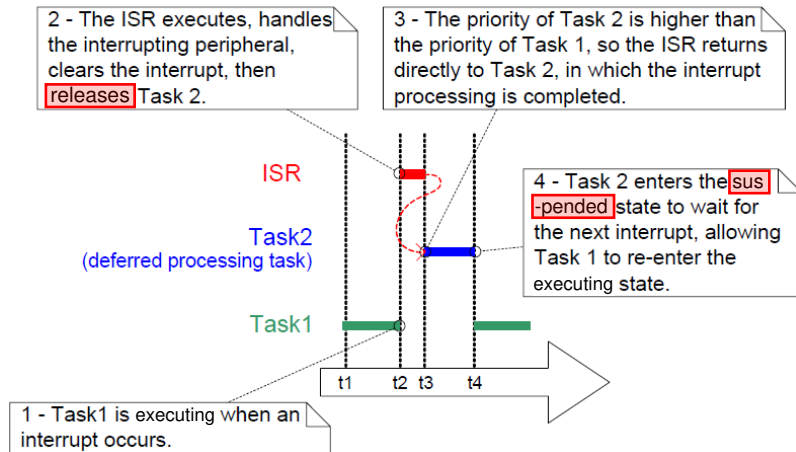
How are tasks (threads) and hardware interrupts scheduled jointly?

- Although written in software, an **interrupt service routine (ISR)** is a hardware/software feature because the hardware controls which interrupt service routine will run, and when it will run.
- **Tasks will only run when there are no ISRs running**, so the lowest priority interrupt will interrupt the highest priority task, and there is no way for a task to pre-empt an ISR. In other words, ISRs have always a higher priority than any other task.
- **Usual pattern:**
 - ISRs are usually very short. They find out the reason for the interrupt, clear the interrupt flag and determine what to do in order to handle the interrupt.
 - Then, they unblock a regular task (thread) that performs the necessary processing related to the interrupt.
 - For blocking and unblocking, usually semaphores are used.

5 - 34

5 - 35

Example ThreadX – Interrupts



Embedded Systems

6. Aperiodic and Periodic Scheduling

© Lothar Thiele

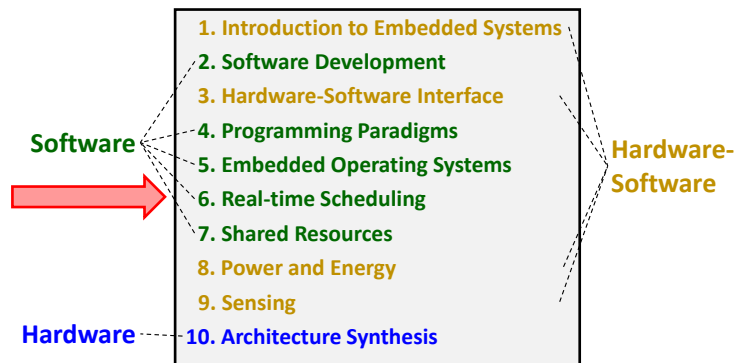
Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

5 - 36

Where we are ...



Basic Terms and Models

6 - 2

6 - 3

Basic Terms

Real-time systems

- **Hard:** A real-time task is said to be hard, if missing its deadline may cause catastrophic consequences on the environment under control. Examples are sensory data acquisition, detection of critical conditions, actuator servoing.
- **Soft:** A real-time task is called soft, if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior. Examples are command interpreter of the user interface, displaying messages on the screen.

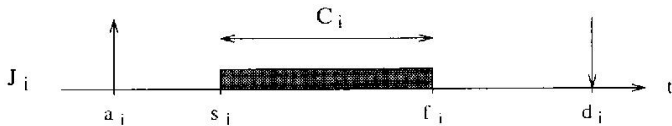
6 - 4

Schedule and Timing

- A schedule is said to be **feasible**, if all task can be completed according to a set of specified constraints.
- A set of tasks is said to be **schedulable**, if there exists at least one algorithm that can produce a feasible schedule.
- **Arrival time** a_i or **release time** r_i is the time at which a task becomes ready for execution.
- **Computation time** C_i is the time necessary to the processor for executing the task without interruption.
- **Deadline** d_i is the time at which a task should be completed.
- **Start time** s_i is the time at which a task starts its execution.
- **Finishing time** f_i is the time at which a task finishes its execution.
- A **preemptive schedule** is a schedule in which the running task can be arbitrarily suspended at any time, to assign the CPU to another task according to a predefined scheduling policy.

6 - 5

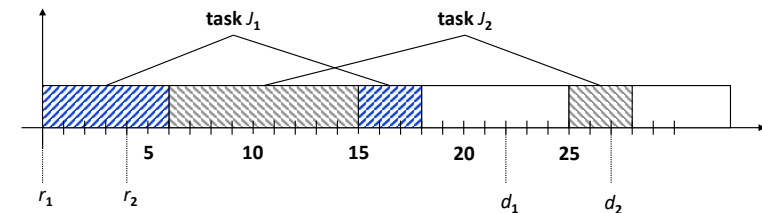
Schedule and Timing



- Using the above definitions, we have $d_i \geq r_i + C_i$
- **Lateness** $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; note that if a task completes before the deadline, its lateness is negative.
- **Tardiness or exceeding time** $E_i = \max(0, L_i)$ is the time a task stays active after its deadline.
- **Laxity or slack time** $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

6 - 6

Example for Real-Time Model



Computation times: $C_1 = 9$, $C_2 = 12$

Start times: $s_1 = 0$, $s_2 = 6$

Finishing times: $f_1 = 18$, $f_2 = 28$

Lateness: $L_1 = -4$, $L_2 = 1$

Tardiness: $E_1 = 0$, $E_2 = 1$

Laxity: $X_1 = 13$, $X_2 = 11$

6 - 7

Metrics to Compare Schedules

- Average response time:

$$\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - r_i)$$

- Total completion time:

$$t_c = \max_i(f_i) - \min_i(r_i)$$

- Weighted sum of response time:

$$t_w = \frac{\sum_{i=1}^n w_i (f_i - r_i)}{\sum_{i=1}^n w_i}$$

- Maximum lateness:

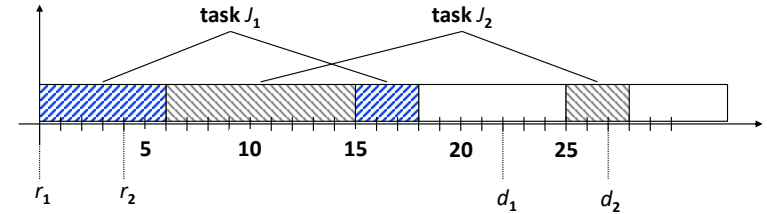
$$L_{\max} = \max_i (f_i - d_i)$$

- Number of late tasks:

$$N_{\text{late}} = \sum_{i=1}^n \text{miss}(f_i)$$

$$\text{miss}(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

Metrics Example



Average response time:

$$\bar{t}_r = \frac{1}{2} (18 + 24) = 21$$

Total completion time:

$$t_c = 28 - 0 = 28$$

Weighted sum of response times: $w_1 = 2, w_2 = 1: t_w = \frac{2 \cdot 18 + 24}{3} = 20$

Number of late tasks:

$$N_{\text{late}} = 1$$

Maximum lateness:

$$L_{\max} = 1$$

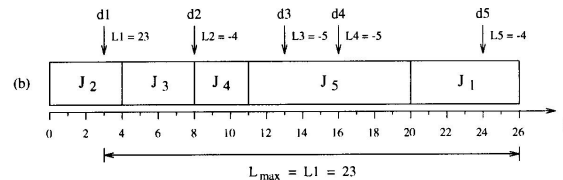
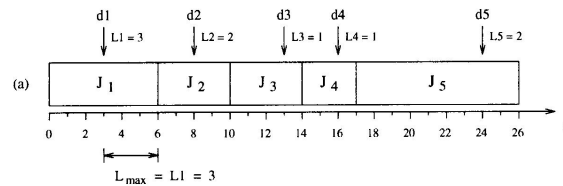
6 - 8

6 - 9

Metrics and Scheduling Example

In schedule (a), the *maximum lateness is minimized*, but all tasks miss their deadlines.

In schedule (b), the maximal lateness is larger, but only one *task misses* its deadline.



6 - 10

6 - 11

Real-Time Scheduling of Aperiodic Tasks

Overview Aperiodic Task Scheduling

Scheduling of *aperiodic tasks* with real-time constraints:

- Table with some known algorithms:

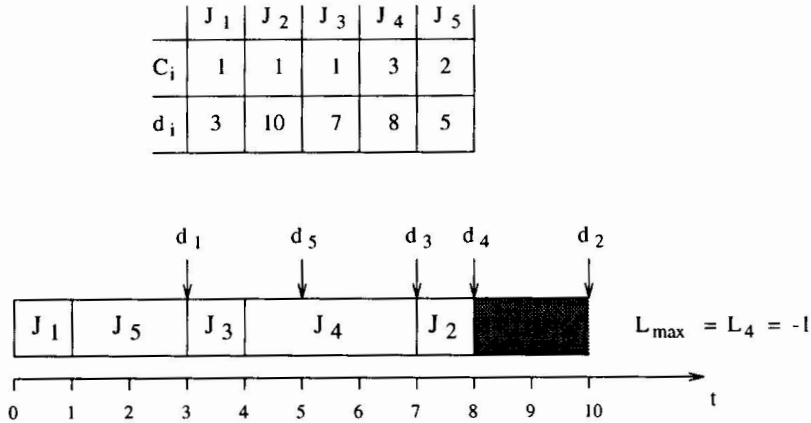
	Equal arrival times non preemptive	Arbitrary arrival times preemptive
Independent tasks	EDD (Jackson)	EDF (Horn)
Dependent tasks	LDF (Lawler)	EDF* (Chetto)

Earliest Deadline Due (EDD)

Jackson's rule: Given a set of n tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.

Earliest Deadline Due (EDD)

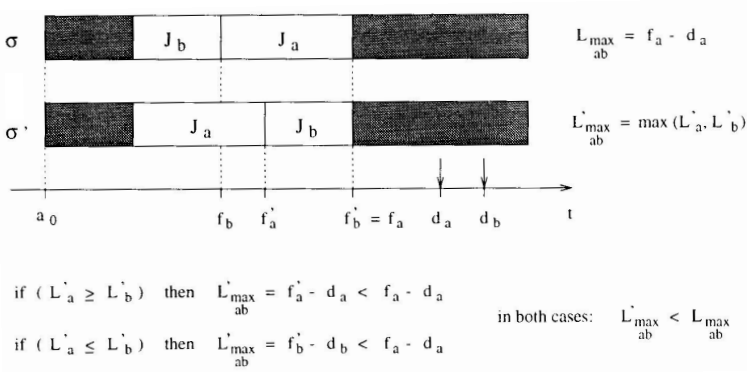
Example 1:



Earliest Deadline Due (EDD)

Jackson's rule: Given a set of n tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.

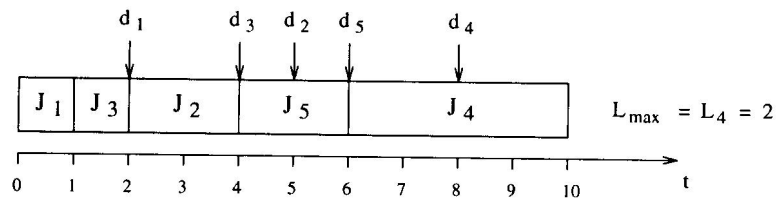
Proof concept:



Earliest Deadline Due (EDD)

Example 2:

	J_1	J_2	J_3	J_4	J_5
C_i	1	2	1	4	2
d_i	2	5	4	8	6



6 - 16

Earliest Deadline First (EDF)

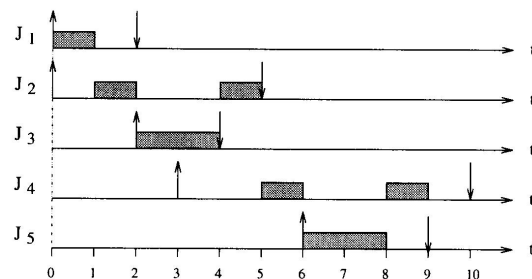
Horn's rule: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes a task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.

6 - 17

Earliest Deadline First (EDF)

Example:

	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9



6 - 18

Earliest Deadline First (EDF)

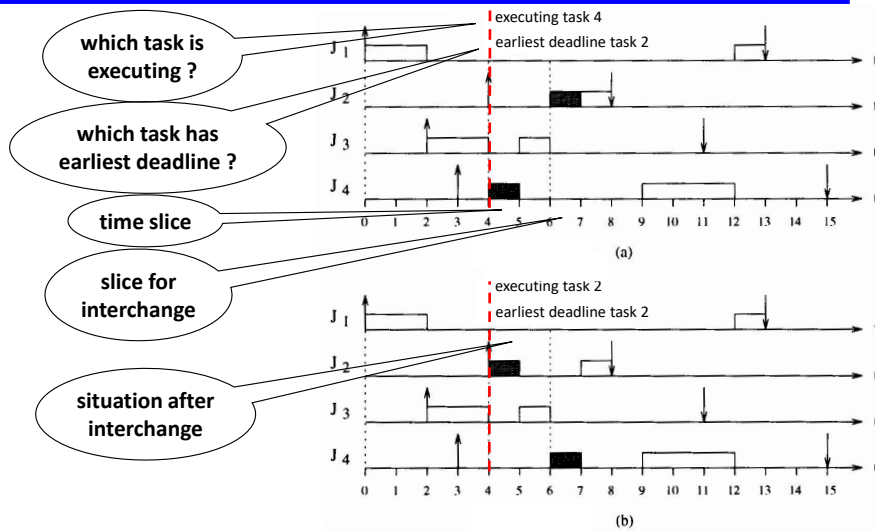
Horn's rule: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.

Concept of proof:

For each time interval $[t, t+1)$ it is verified, whether the actual running task is the one with the earliest absolute deadline. If this is not the case, the task with the earliest absolute deadline is executed in this interval instead. This operation cannot increase the maximum lateness.

6 - 19

Earliest Deadline First (EDF)



6 - 20

Earliest Deadline First (EDF)

Acceptance test:

- worst case finishing time of task i:
- EDF guarantee condition:
- algorithm:

$$f_i = t + \sum_{k=1}^i c_k(t)$$

remaining worst-case execution time of task k

$$\forall i=1, \dots, n \quad t + \sum_{k=1}^i c_k(t) \leq d_i$$

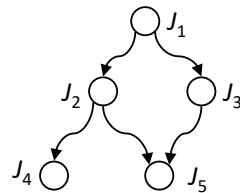
```

Algorithm: EDF_guarantee (J, Jnew)
{
    J' = J ∪ {Jnew}; /* ordered by deadline */
    t = current_time();
    f0 = t;
    for (each Ji ∈ J') {
        fi = fi-1 + ci(t);
        if (fi > di) return (INFEASIBLE);
    }
    return (FEASIBLE);
}
    
```

6 - 21

Precedence Constraints

- Precedence relations** between tasks can be described through an *acyclic directed graph* G where tasks are represented by nodes and precedence relations by arrows. G induces a partial order on the task set.
- There are different *interpretations* possible:
 - All successors of a task are activated (*concurrent task execution*). We will use this interpretation in the lecture.
 - One successor of a task is activated: *non-deterministic choice*.

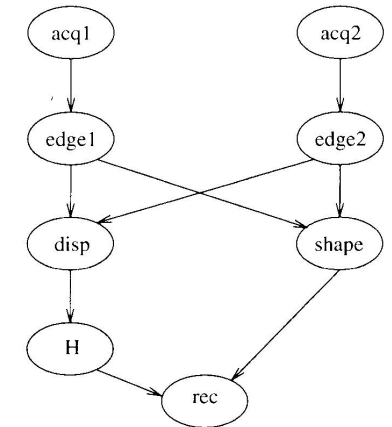
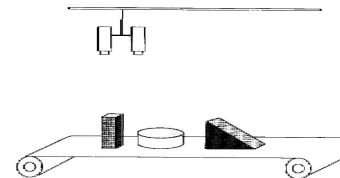


6 - 22

Precedence Constraints

Example for concurrent activation:

- Image acquisition *acq1 acq2*
- Low level image processing *edge1 edge2*
- Feature/contour extraction *shape*
- Pixel disparities *disp*
- Object size *H*
- Object recognition *rec*



6 - 23

Earliest Deadline First (EDF*)

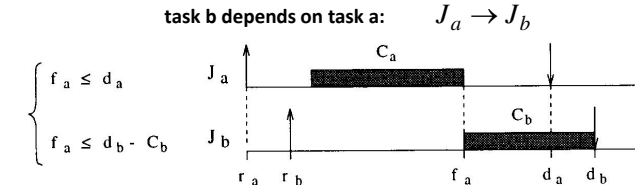
- The problem of *scheduling a set of n tasks with precedence constraints* (concurrent activation) can be solved in polynomial time complexity if tasks are preemptable.
- The **EDF*** algorithm determines a *feasible schedule* in the case of tasks with precedence constraints if there exists one.
- By the modification it is guaranteed that if *there exists a valid schedule* at all then
 - a task starts execution not earlier than its release time and not earlier than the finishing times of its predecessors (a task cannot preempt any predecessor)
 - all tasks finish their execution within their deadlines

6 - 24

Earliest Deadline First (EDF*)

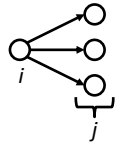
Modification of deadlines:

- Task must finish the execution time within its deadline.
- Task must not finish the execution later than the maximum start time of its successor.



- Solution:**

$$d_i^* = \min(d_i, \min(d_j - C_j : J_i \rightarrow J_j))$$

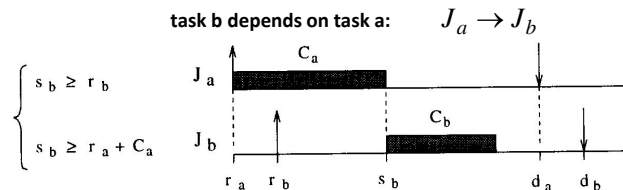


6 - 25

Earliest Deadline First (EDF*)

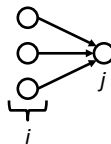
Modification of release times:

- Task must start the execution not earlier than its release time.
- Task must not start the execution earlier than the minimum finishing time of its predecessor.



- Solution:**

$$r_j^* = \max(r_j, \max(r_i + C_i : J_i \rightarrow J_j))$$



6 - 26

Earliest Deadline First (EDF*)

Algorithm for modification of release times:

- For any initial node of the precedence graph set $r_i^* = r_i$
- Select a task j such that its release time has not been modified but the release times of all immediate predecessors i have been modified. If no such task exists, exit.
- Set $r_j^* = \max(r_j, \max(r_i^* + C_i : J_i \rightarrow J_j))$
- Return to step 2

Algorithm for modification of deadlines:

- For any terminal node of the precedence graph set $d_i^* = d_i$
- Select a task i such that its deadline has not been modified but the deadlines of all immediate successors j have been modified. If no such task exists, exit.
- Set $d_i^* = \min(d_i, \min(d_j - C_j : J_i \rightarrow J_j))$
- Return to step 2

6 - 27

Earliest Deadline First (EDF*)

Proof concept:

- Show that if there exists a feasible schedule for the modified task set under EDF then the original task set is also schedulable. To this end, show that the original task set meets the timing constraints also. This can be done by using $r_i^* \geq r_i$, $d_i^* \leq d_i$; we only made the constraints stricter.
- Show that if there exists a schedule for the original task set, then also for the modified one. We can show the following: If there exists no schedule for the modified task set, then there is none for the original task set. This can be done by showing that no feasible schedule was excluded by changing the deadlines and release times.
- In addition, show that the precedence relations in the original task set are not violated. In particular, show that
 - a task cannot start before its predecessor and
 - a task cannot preempt its predecessor.

Overview

Table of some known *preemptive scheduling algorithms for periodic tasks*:

	Deadline equals period	Deadline smaller than period
static priority	RM (rate-monotonic)	DM (deadline-monotonic)
dynamic priority	EDF	EDF*

Real-Time Scheduling of Periodic Tasks

Model of Periodic Tasks

- *Examples*: sensory data acquisition, low-level actuation, control loops, action planning and system monitoring.
- When an *application* consists of several concurrent periodic tasks with individual timing constraints, the OS has to guarantee that each periodic instance is regularly activated at its proper rate and is completed within its deadline.

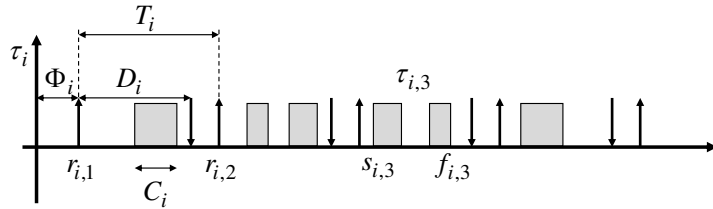
- *Definitions*:

- Γ : denotes a set of periodic tasks
- τ_i : denotes a periodic task
- $\tau_{i,j}$: denotes the jth instance of task i
- $r_{i,j}, s_{i,j}, f_{i,j}, d_{i,j}$: denote the release time, start time, finishing time, absolute deadline of the jth instance of task i
- Φ_i : denotes the phase of task i (release time of its first instance)
- D_i : denotes the relative deadline of task i
- T_i : denotes the period of task i

Model of Periodic Tasks

- The following hypotheses are assumed on the tasks (continued):

- All periodic tasks are *independent*; that is, there are no precedence relations and no resource constraints.
- No task can suspend itself*, for example on I/O operations.
- All tasks are *released as soon as they arrive*.
- All *overheads* in the OS kernel are assumed to be *zero*.
- Example:*



6 - 32

Model of Periodic Tasks

- The following hypotheses are assumed on the tasks:

- The instances of a periodic task are *regularly activated at a constant rate*. The interval T_i between two consecutive activations is called period. The release times satisfy

$$r_{i,j} = \Phi_i + (j-1)T_i$$

- All instances have the *same worst case execution time* C_i
- All instances of a periodic task have the *same relative deadline* D_i . Therefore, the absolute deadlines satisfy

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

- Often, the relative deadline equals the period $D_i = T_i$ (*implicit deadline*), and therefore

$$d_{i,j} = \Phi_i + jT_i$$

6 - 33

Rate Monotonic Scheduling (RM)

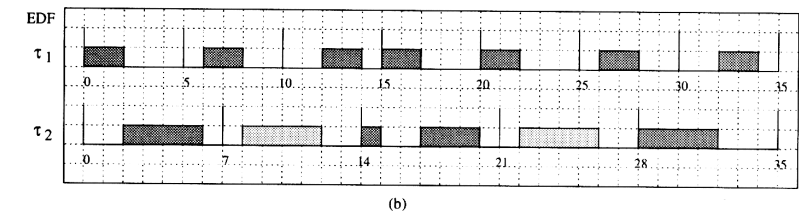
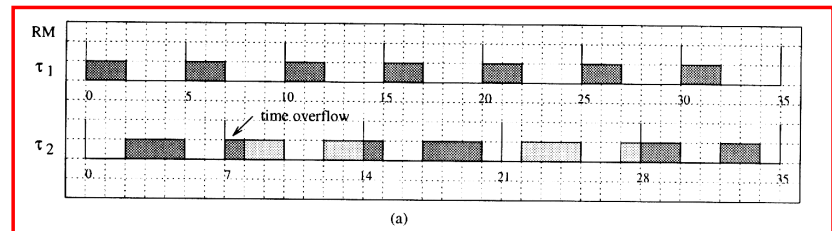
- Assumptions:

- Task priorities are assigned to tasks before execution and do not change over time (static priority assignment).
- RM is intrinsically preemptive: the currently executing job is preempted by a job of a task with higher priority.
- Deadlines equal the periods $D_i = T_i$.

Rate-Monotonic Scheduling Algorithm: Each task is assigned a priority. Tasks with higher request rates (that is with shorter periods) will have higher priorities. Jobs of tasks with higher priority interrupt jobs of tasks with lower priority.

Periodic Tasks

Example: 2 tasks, deadlines = periods, utilization = 97%



6 - 34

6 - 35

Rate Monotonic Scheduling (RM)

Optimality: RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithm can schedule a task set that cannot be scheduled by RM.

- The **proof** is done by considering several cases that may occur, but the main ideas are as follows:
 - A **critical instant** for any task occurs whenever the task is released simultaneously with all higher priority tasks. The tasks schedulability can easily be checked at their critical instants. If all tasks are feasible at their critical instant, then the task set is schedulable in any other condition.
 - Show that, given two periodic tasks, if the schedule is feasible by an arbitrary priority assignment, then it is also feasible by RM.
 - Extend the result to a set of n periodic tasks.

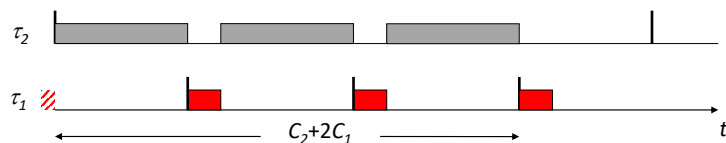
Proof of Critical Instance

Definition: A critical instant of a task is the time at which the release of a job will produce the largest response time.

Lemma: For any task, the critical instant occurs if a job is simultaneously released with all higher priority jobs.

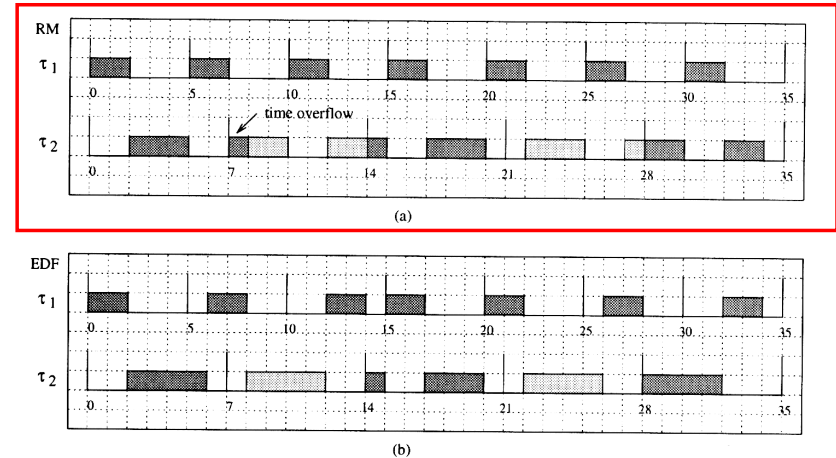
Proof sketch: Start with 2 tasks τ_1 and τ_2 .

Response time of a job of τ_2 is delayed by jobs of τ_1 of higher priority:



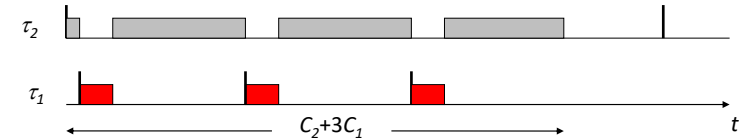
Periodic Tasks

Example: 2 tasks, deadlines = periods, utilization = 97%



Proof of Critical Instance

Delay may increase if τ_1 starts earlier:



Maximum delay achieved if τ_2 and τ_1 start simultaneously.

Repeating the argument for all higher priority tasks of some task τ_2 :

The worst case response time of a job occurs when it is released simultaneously with all higher-priority jobs.

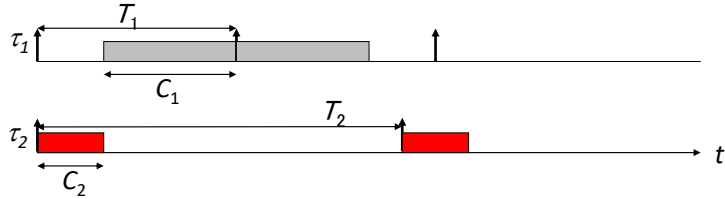
Proof of RM Optimality (2 Tasks)

We have two tasks τ_1, τ_2 with periods $T_1 < T_2$.

Define $F = \lfloor T_2/T_1 \rfloor$: the number of periods of τ_1 fully contained in T_2

Consider two cases A and B:

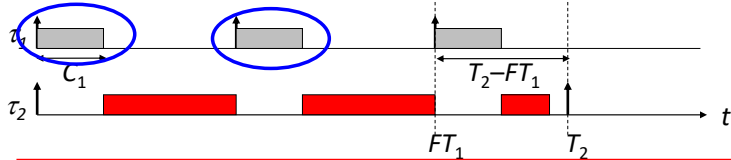
Case A: Assume RM is **not** used $\rightarrow$ prio(τ_2) is highest:



Schedule is feasible if $C_1 + C_2 \leq T_1$ and $C_2 \leq T_2$ (A)

Proof of RM Optimality (2 Tasks)

Case B: Assume RM **is** used $\rightarrow$ prio(τ_1) is highest:



Schedulable is feasible if $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$ and $C_1 \leq T_1$ (B)

We need to show that (A) $\Rightarrow$ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$

$$C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1 \Rightarrow$$

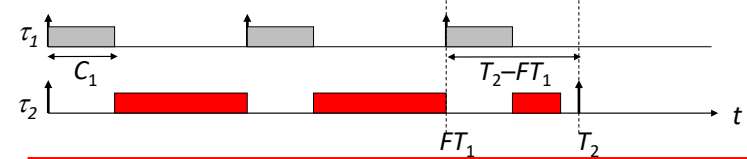
$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq FT_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

6 - 40

Proof of RM Optimality (2 Tasks)

Case B: Assume RM **is** used $\rightarrow$ prio(τ_1) is highest:



Schedulable is feasible if $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$ and $C_1 \leq T_1$ (B)

We need to show that (A) $\Rightarrow$ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$

$$C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1 \Rightarrow$$

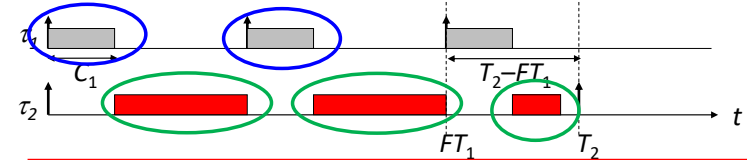
$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq FT_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

6 - 41

Proof of RM Optimality (2 Tasks)

Case B: Assume RM **is** used $\rightarrow$ prio(τ_1) is highest:



Schedulable is feasible if $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$ and $C_1 \leq T_1$ (B)

We need to show that (A) $\Rightarrow$ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$

$$C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1 \Rightarrow$$

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq FT_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$$

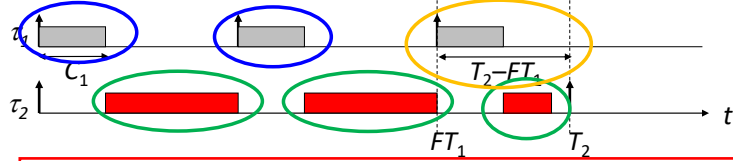
Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

6 - 42

6 - 43

Proof of RM Optimality (2 Tasks)

Case B: Assume RM is used $\rightarrow$ prio(τ_1) is highest:



Schedulable is feasible if $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$ and $C_1 \leq T_1$ (B)

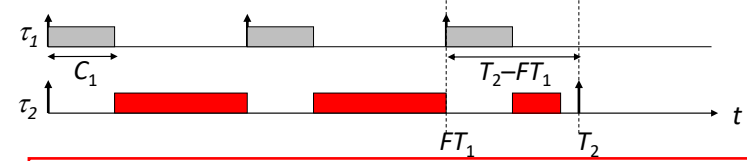
We need to show that (A) $\Rightarrow$ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$
 $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1 \Rightarrow$
 $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq FT_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

6 - 44

Proof of RM Optimality (2 Tasks)

Case B: Assume RM is used $\rightarrow$ prio(τ_1) is highest:



Schedulable is feasible if $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$ and $C_1 \leq T_1$ (B)

We need to show that (A) $\Rightarrow$ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$
 $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1 \Rightarrow$
 $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq FT_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

6 - 45

Rate Monotonic Scheduling (RM)

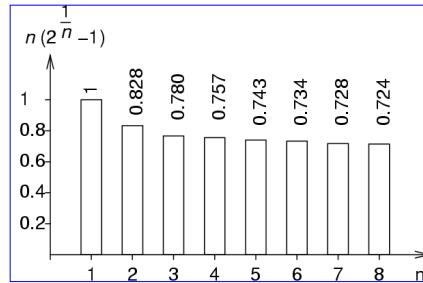
Schedulability analysis: A set of periodic tasks is schedulable with RM if

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

This condition is sufficient but not necessary.

The term $U = \sum_{i=1}^n \frac{C_i}{T_i}$ denotes the **processor utilization factor**

utilization factor U which is the fraction of processor time spent in the execution of the task set.



6 - 46

Proof of Utilization Bound (2 Tasks)

We have two tasks τ_1, τ_2 with periods $T_1 < T_2$.

Define $F = \lfloor T_2 / T_1 \rfloor$: number of periods of τ_1 fully contained in T_2

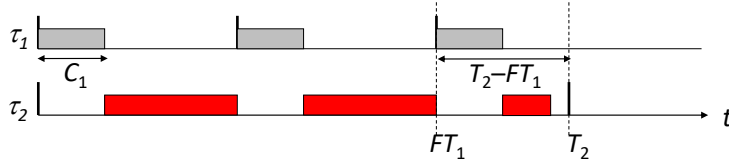
Proof Concept: Compute upper bound on utilization U such that the task set is still schedulable:

- assign priorities according to RM;
- compute upper bound U_{up} by increasing the computation time C_2 to just meet the deadline of τ_2 ; we will determine this limit of C_2 using the results of the RM optimality proof.
- minimize upper bound with respect to other task parameters in order to find the utilization below which the system is definitely schedulable.

6 - 47

Proof of Utilization Bound (2 Tasks)

As before:



Schedulable if $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$ and $C_1 \leq T_1$

Utilization:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - FC_1 - \min\{T_2 - FT_1, C_1\}}{T_2}$$

$$= 1 + \frac{C_1(T_2 - FT_1) - T_1 \min\{T_2 - FT_1, C_1\}}{T_1 T_2}$$

6 - 48

Proof of Utilization Bound (2 Tasks)

Minimize utilization bound w.r.t C_1 :

- If $C_1 \leq T_2 - FT_1$ then U decreases with increasing C_1
- If $T_2 - FT_1 \leq C_1$ then U decreases with decreasing C_1
- Therefore, minimum U is obtained with $C_1 = T_2 - FT_1$:

$$U = 1 + \frac{(T_2 - FT_1)^2 - T_1(T_2 - FT_1)}{T_1 T_2}$$

$$= 1 + \frac{T_1}{T_2} \left(\left(\frac{T_2}{T_1} - F \right)^2 - \left(\frac{T_2}{T_1} - F \right) \right)$$

We now need to minimize w.r.t. $G = T_2/T_1$ where $F = \lfloor T_2/T_1 \rfloor$ and $T_1 < T_2$. As F is integer, we first suppose that it is independent of $G = T_2/T_1$. Then we obtain

$$U = \frac{T_1}{T_2} \left(\left(\frac{T_2}{T_1} - F \right)^2 + F \right) = \frac{(G - F)^2 + F}{G}$$

6 - 50

Proof of Utilization Bound (2 Tasks)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - FC_1 - \min\{T_2 - FT_1, C_1\}}{T_2}$$

$$= 1 + \frac{C_1(T_2 - FT_1) - T_1 \min\{T_2 - FT_1, C_1\}}{T_1 T_2}$$

6 - 49

Proof of Utilization Bound (2 Tasks)

Minimizing U with respect to G yields

$$2G(G - F) - (G - F)^2 - F = G^2 - (F^2 + F) = 0$$

If we set $F = 1$, then we obtain

$$G = \frac{T_2}{T_1} = \sqrt{2} \quad U = 2(\sqrt{2} - 1)$$

It can easily be checked, that all other integer values for F lead to a larger upper bound on the utilization.

6 - 51

Deadline Monotonic Scheduling (DM)

- Assumptions are as in rate monotonic scheduling, but *deadlines may be smaller than the period*, i.e.

$$C_i \leq D_i \leq T_i$$

Algorithm: Each task is assigned a priority. Tasks with smaller relative deadlines will have higher priorities. Jobs with higher priority interrupt jobs with lower priority.

- Schedulability Analysis:** A set of periodic tasks is schedulable with DM if

$$\sum_{i=1}^n \frac{C_i}{D_i} \leq n(2^{1/n} - 1)$$

This condition is sufficient but not necessary (in general).

6 - 52

Deadline Monotonic Scheduling (DM)

There is also a *necessary and sufficient schedulability test* which is computationally more involved. It is based on the following observations:

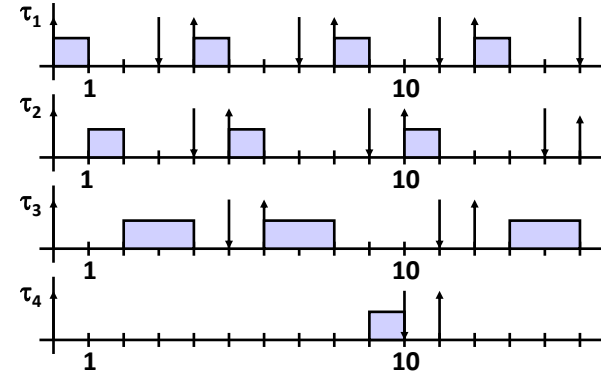
- The *worst-case processor demand* occurs when all tasks are released simultaneously; that is, at their critical instances.
- For each task i , the sum of its processing time and the *interference* imposed by higher priority tasks must be less than or equal to D_i .
- A measure of the *worst case interference* for task i can be computed as the sum of the processing times of all higher priority tasks released before some time t where tasks are ordered according to $m < n \Leftrightarrow D_m < D_n$:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

6 - 54

Deadline Monotonic Scheduling (DM) - Example

$$U = 0.874 \quad \sum_{i=1}^n \frac{C_i}{D_i} = 1.08 > n(2^{1/n} - 1) = 0.757$$



6 - 53

Deadline Monotonic Scheduling (DM)

- The *longest response time* R_i of a job of a periodic task i is computed, at the critical instant, as the sum of its computation time and the interference due to preemption by higher priority tasks:

$$R_i = C_i + I_i$$

- Hence, the schedulability test needs to compute the smallest R_i that satisfies

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

for all tasks i . Then, $R_i \leq D_i$ must hold for all tasks i .

- It can be shown that this *condition is necessary and sufficient*.

6 - 55

Deadline Monotonic Scheduling (DM)

The longest response times R_i of the periodic tasks i can be computed iteratively by the following algorithm:

```

Algorithm: DM_guarantee ( $\Gamma$ )
{
  for (each  $\tau_i \in \Gamma$ ) {
    I = 0;
    do {
      R = I + C_i;
      if (R > D_i) return(UNSCHEDULABLE);
      I =  $\sum_{j=1, \dots, (i-1)} \lceil R/T_j \rceil C_j$ ;
    } while (I + C_i > R);
  }
  return(SCHEDULABLE);
}

```

DM Example

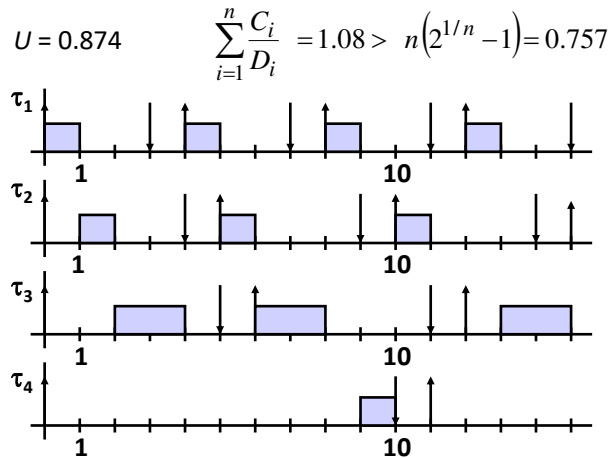
Example:

- Task 1: $C_1 = 1; T_1 = 4; D_1 = 3$
- Task 2: $C_2 = 1; T_2 = 5; D_2 = 4$
- Task 3: $C_3 = 2; T_3 = 6; D_3 = 5$
- Task 4: $C_4 = 1; T_4 = 11; D_4 = 10$
- Algorithm for the schedulability test for task 4:
 - Step 0: $R_4 = 1$
 - Step 1: $R_4 = 5$
 - Step 2: $R_4 = 6$
 - Step 3: $R_4 = 7$
 - Step 4: $R_4 = 9$
 - Step 5: $R_4 = 10$

6 - 56

6 - 57

DM Example



6 - 58

EDF Scheduling (earliest deadline first)

▪ **Assumptions:**

- dynamic priority assignment
- intrinsically preemptive

- **Algorithm:** The currently executing task is preempted whenever another periodic instance with earlier deadline becomes active.

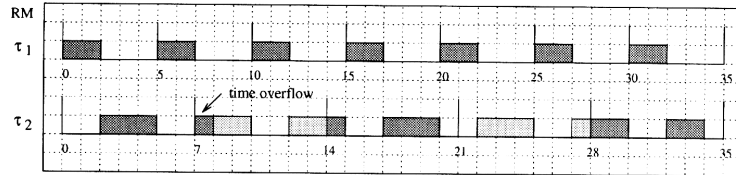
$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

- **Optimality:** No other algorithm can schedule a set of periodic tasks if the set that can not be scheduled by EDF.
- The proof is simple and follows that of the aperiodic case.

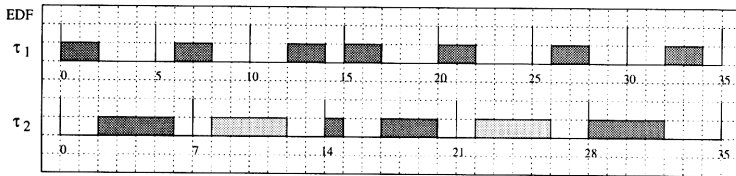
6 - 59

Periodic Tasks

Example: 2 tasks, deadlines = periods, utilization = 97%



(a)



(b)

EDF Scheduling

- If the utilization satisfies $U > 1$, then there is no valid schedule: The total demand of computation time in interval $T = T_1 \cdot T_2 \cdot \dots \cdot T_n$ is

$$\sum_{i=1}^n \frac{C_i}{T_i} T = UT > T$$

and therefore, it exceeds the available processor time in this interval.

- If the utilization satisfies $U \leq 1$, then there is a valid schedule.

We will prove this fact by contradiction: Assume that deadline is missed at some time t_2 . Then we will show that the utilization was larger than 1.

EDF Scheduling

A *necessary and sufficient schedulability test* for $D_i = T_i$:

A set of periodic tasks is schedulable with EDF if and only if $\sum_{i=1}^n \frac{C_i}{T_i} = U \leq 1$

The term $U = \sum_{i=1}^n \frac{C_i}{T_i}$ denotes the *average processor utilization*.

EDF Scheduling

- If the deadline was missed at t_2 then define t_1 as a time before t_2 such that (a) the processor is continuously busy in $[t_1, t_2]$ and (b) the processor only executes tasks that have their arrival time AND their deadline in $[t_1, t_2]$.
- Why does such a time t_1 exist? We find such a t_1 by starting at t_2 and going backwards in time, always ensuring that the processor only executed tasks that have their deadline before or at t_2 :
 - Because of EDF, the processor will be busy shortly before t_2 and it executes on the task that has deadline at t_2 .
 - Suppose that we reach a time such that shortly before the processor works on a task with deadline after t_2 or the processor is idle, then we found t_1 : We know that there is no execution on a task with deadline after t_2 .
 - But it could be in principle, that a task that arrived before t_1 is executing in $[t_1, t_2]$.
 - If the processor is idle before t_1 , then this is clearly not possible due to EDF (the processor is not idle, if there is a ready task).
 - If the processor is not idle before t_1 , this is not possible as well. Due to EDF, the processor will always work on the task with the closest deadline and therefore, once starting with a task with deadline after t_2 all task with deadlines before t_2 are finished.

EDF Scheduling

- Within the interval $[t_1, t_2]$ the total *computation time demanded* by the periodic tasks is bounded by

$$C_p(t_1, t_2) = \sum_{i=1}^n \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^n \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)U$$

number of complete periods
of task i in the interval

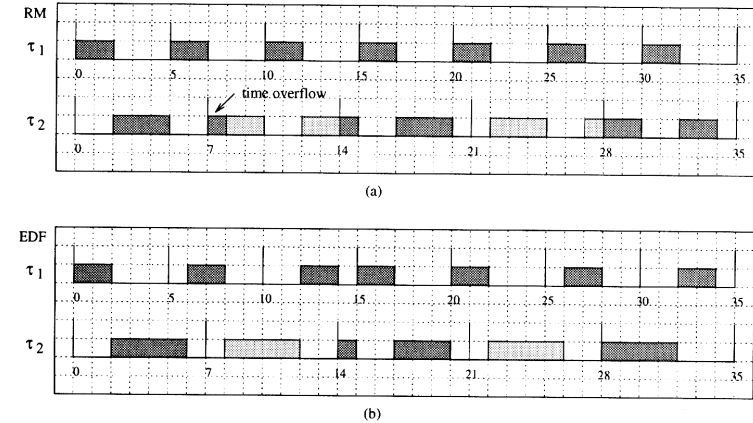
- Since the deadline at time t_2 is missed, we must have:

$$t_2 - t_1 < C_p(t_1, t_2) \leq (t_2 - t_1)U \Rightarrow U > 1$$

Real-Time Scheduling of Mixed Task Sets

Periodic Task Scheduling

Example: 2 tasks, deadlines = periods, utilization = 97%



Problem of Mixed Task Sets

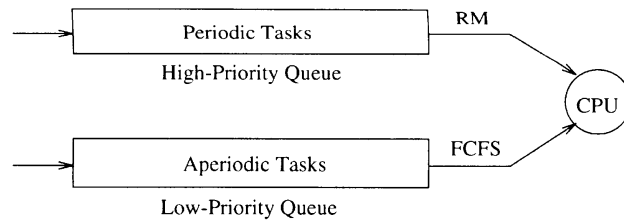
In many applications, there are aperiodic as well as periodic tasks.

- Periodic tasks: time-driven**, execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates.
- Aperiodic tasks: event-driven**, may have hard, soft, non-real-time requirements depending on the specific application.
- Sporadic tasks:** Offline guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment; that is by assuming a *maximum arrival rate* for each critical event. Aperiodic tasks characterized by a minimum interarrival time are called sporadic.

Background Scheduling

Background scheduling is a simple solution for RM and EDF:

- Processing of aperiodic tasks in the background, i.e. execute if there are no pending periodic requests.
- Periodic tasks are not affected.
- Response of aperiodic tasks may be prohibitively long and there is no possibility to assign a higher priority to them.
- Example:



6 - 68

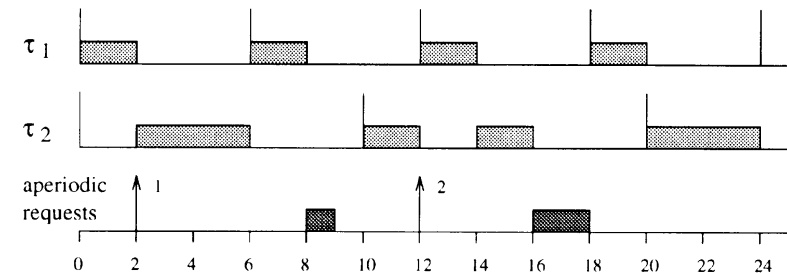
Rate-Monotonic Polling Server

- Idea:** Introduce an artificial periodic task whose purpose is to service aperiodic requests as soon as possible (therefore, “server”).
- Function of *polling server (PS)*
 - At regular intervals equal to T_s , a PS task is instantiated. When it has the highest current priority, it serves any pending aperiodic requests within the limit of its capacity C_s .
 - If no aperiodic requests are pending, PS suspends itself until the beginning of the next period and the time originally allocated for aperiodic service is not preserved for aperiodic execution.
 - Its priority (period!) can be chosen to match the response time requirement for the aperiodic tasks.
- Disadvantage:** If an aperiodic requests arrives just after the server has suspended, it must wait until the beginning of the next polling period.

6 - 70

Background Scheduling

Example (rate monotonic periodic schedule):



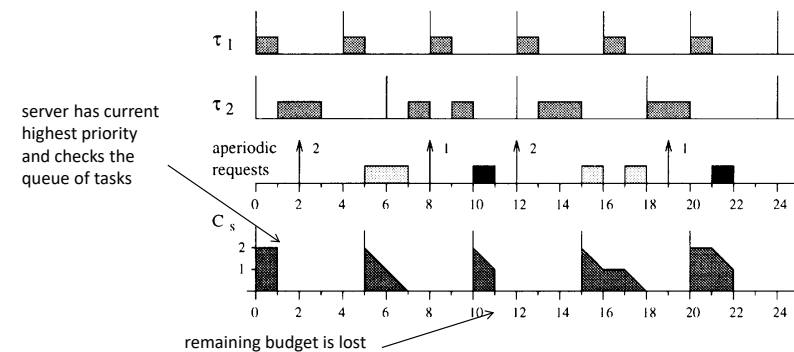
6 - 69

Rate-Monotonic Polling Server

Example:

	C_i	T_i
τ_1	1	4
τ_2	2	6

Server
$C_s = 2$
$T_s = 5$



6 - 71

Rate-Monotonic Polling Server

Schedulability analysis of periodic tasks:

- The interference by a server task is the same as the one introduced by an equivalent periodic task in rate-monotonic fixed-priority scheduling.
- A set of periodic tasks and a server task can be executed within their deadlines if

$$\frac{C_s}{T_s} + \sum_{i=1}^n \frac{C_i}{T_i} \leq (n+1)(2^{1/(n+1)} - 1)$$

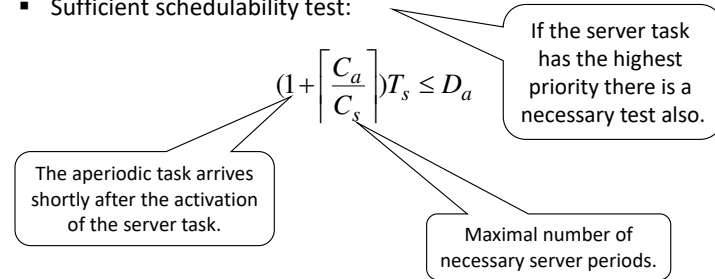
- Again, this test is sufficient but not necessary.

Rate-Monotonic Polling Server

Guarantee the response time of aperiodic requests:

- **Assumption:** An aperiodic task is finished before a new aperiodic request arrives.

- Computation time C_a , deadline D_a
- Sufficient schedulability test:



6 - 72

6 - 73

EDF – Total Bandwidth Server

Total Bandwidth Server:

- When the kth aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

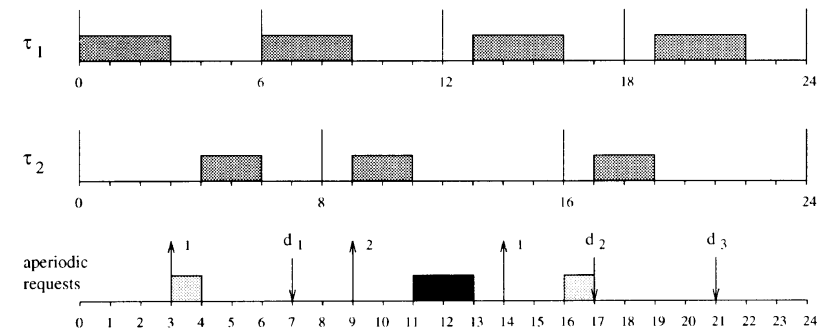
where C_k is the execution time of the request and U_s is the server utilization factor (that is, its bandwidth). By definition, $d_0=0$.

- Once a deadline is assigned, the request is inserted into the ready queue of the system as any other periodic instance.

EDF – Total Bandwidth Server

Example:

$$U_p = 0.75, \quad U_s = 0.25, \quad U_p + U_s = 1$$



6 - 74

6 - 75

EDF – Total Bandwidth Server

Schedulability test:

Given a set of n periodic tasks with processor utilization U_p and a total bandwidth server with utilization U_s , the whole set is schedulable by EDF if and only if

$$U_p + U_s \leq 1$$

Proof:

- In each interval of time $[t_1, t_2]$, if C_{ape} is the total execution time demanded by aperiodic requests arrived at t_1 or later and served with deadlines less or equal to t_2 , then

$$C_{ape} \leq (t_2 - t_1)U_s$$

EDF – Total Bandwidth Server

If this has been proven, the proof of the schedulability test follows closely that of the periodic case.

Proof of lemma:

$$\begin{aligned} C_{ape} &= \sum_{k=k_1}^{k_2} C_k \\ &= U_s \sum_{k=k_1}^{k_2} (d_k - \max(r_k, d_{k-1})) \\ &\leq U_s (d_{k_2} - \max(r_{k_1}, d_{k_1-1})) \\ &\leq U_s (t_2 - t_1) \end{aligned}$$

6 - 76

6 - 77

Embedded Systems

7. Shared Resources

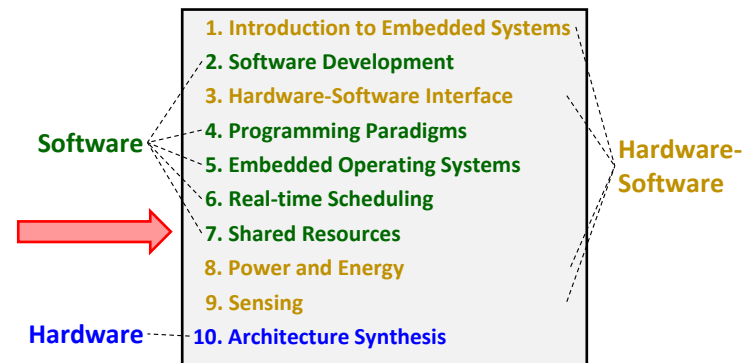
© Lothar Thiele

Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Where we are ...

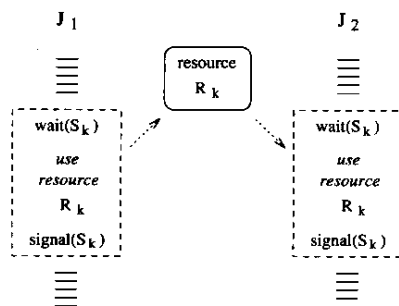


7 - 2

Ressource Sharing

Protecting Exclusive Resources using Semaphores

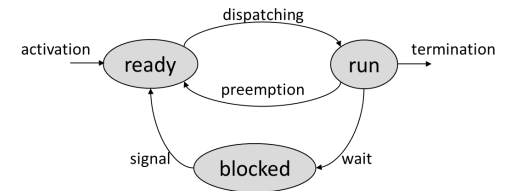
- Each *exclusive resource* R_i must be protected by a different *semaphore* S_i . Each critical section operating on a resource must begin with a $wait(S_i)$ primitive and end with a $signal(S_i)$ primitive.



- All tasks blocked on the same resource are kept in a queue associated with the semaphore. When a running task executes a *wait* on a *locked semaphore*, it enters a *blocked state*, until another task executes a *signal* primitive that *unlocks the semaphore*.

Resource Sharing

- Examples of *shared resources*: data structures, variables, main memory area, file, set of registers, I/O unit, ...
- Many shared resources do not allow simultaneous accesses but require *mutual exclusion*. These resources are called *exclusive resources*. In this case, no two threads are allowed to operate on the resource at the same time.
- There are several methods available to *protect exclusive resources*, for example
 - disabling interrupts* and preemption or
 - using concepts like *semaphores* and *mutex* that put threads into the blocked state if necessary.



Example ThreadX (ES-Lab)

To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a 'mutual exclusion' technique.

One possibility is to disable preemption:

```
...
UINT old_thr;
UINT unused;
tx_thread_preemption_change(thread_ptr, 0, &old_thr);
... /* access to some exclusive resource */
tx_thread_preemption_change(thread_ptr, old_thr, &unused);
...
```

pointer to thread

new preemption-threshold priority level: prevents preemption of the specified thread by threads with equal to or higher priority level (equal or smaller priority) than the preemption-threshold value.

previous preemption-threshold

This kind of critical sections must be kept very short, otherwise they will adversely affect interrupt response times.

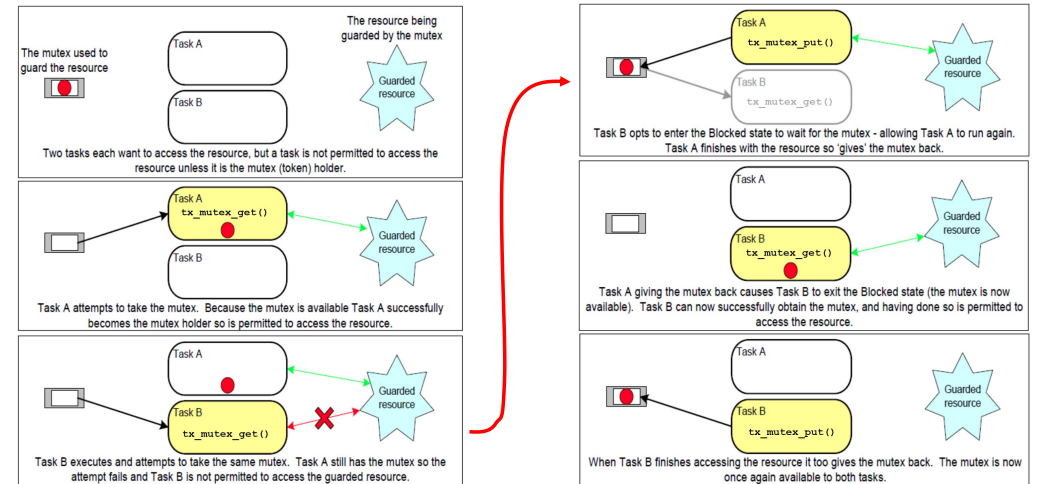
Example ThreadX (ES-Lab)

Another possibility is to use mutual exclusion: In ThreadX, a *mutex* is a special type of *semaphore* that is used to *control access* to a resource that is shared between two or more tasks. *A semaphore that is used for mutual exclusion must always be returned:*

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully 'get' the token (be the token holder). When the token holder has finished with the resource, it must 'put' the token back.
- Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource.

7 - 7

Example ThreadX (ES-Lab)



7 - 8

Example ThreadX (ES-Lab)

Example:

create mutex semaphore

some defined constant for infinite timeout;
otherwise, the function would return if the
mutex was not available for the specified time

```
TX_MUTEX mutex;
TX_THREAD task1;
TX_THREAD task2;

UINT App_ThreadX_Init(VOID *memory_ptr) {
    UINT ret = TX_SUCCESS;
    ret = tx_mutex_create(&mutex, "Mutex", 1);
    if (ret == TX_SUCCESS) {
        tx_thread_create(&task1, "Task 1", t1_fct, ...);
        tx_thread_create(&task2, "Task 2", t2_fct, ...);
    }
    for(;;);
}
```

specifies whether or not this
mutex supports priority inheritance
(see later)

```
void t1_fct( ULONG parameter ) {
    for(;;) {
        ...
        tx_mutex_get(&mutex, TX_WAIT_FOREVER);
        ... /* access to exclusive resource */
        tx_mutex_put(&mutex);
        ...
    }
}
```

```
void t2_fct( ULONG parameter ) {
    for(;;) {
        ...
        tx_mutex_get(&mutex, TX_WAIT_FOREVER);
        ... /* access to exclusive resource */
        tx_mutex_put(&mutex);
        ...
    }
}
```

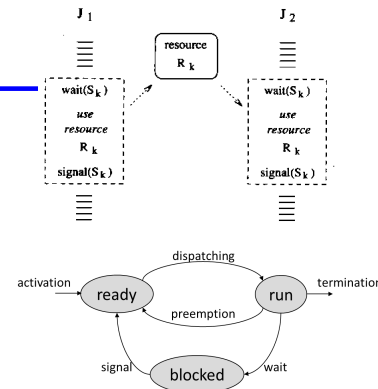
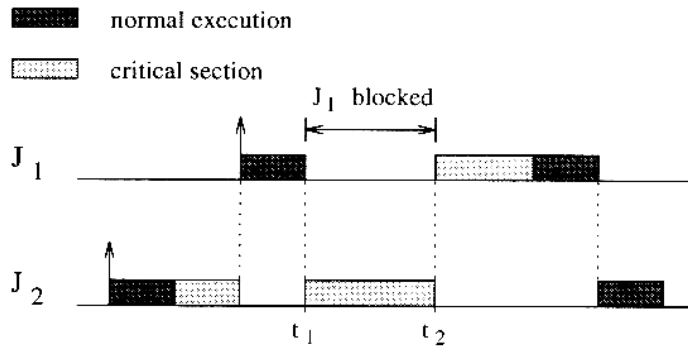
7 - 9

Resource Sharing Priority Inversion

7 - 10

Priority Inversion (1)

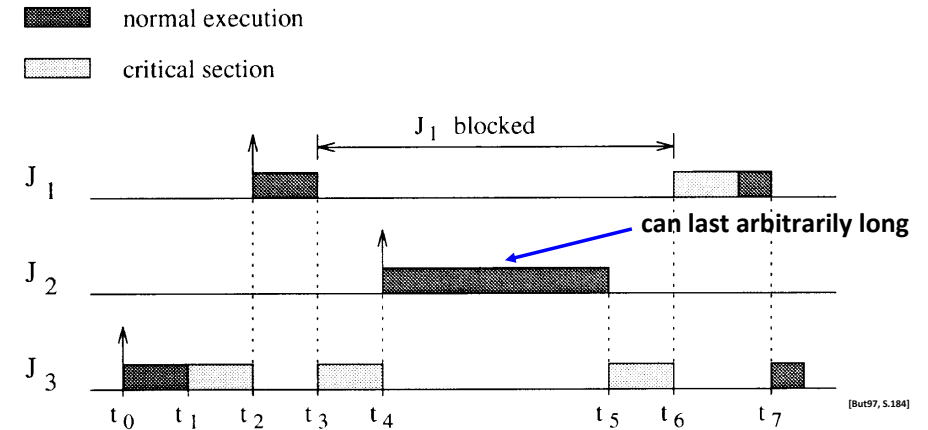
Unavoidable blocking:



7 - 11

Priority Inversion (2)

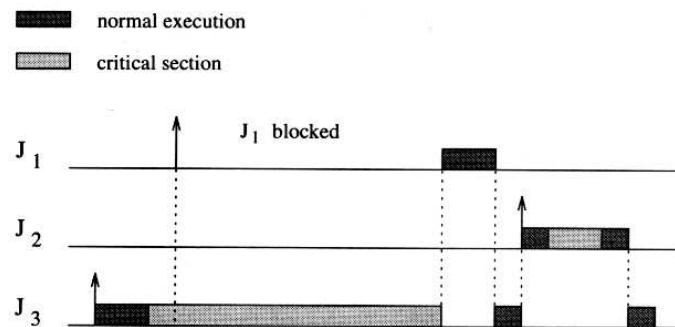
Priority Inversion:



7 - 12

Solutions to Priority Inversion

Disallow preemption during the execution of all critical sections. Simple approach, but it creates unnecessary blocking as unrelated tasks may be blocked.



7 - 13

Resource Access Protocols

Basic idea: Modify the priority of those tasks that cause blocking. When a task J_i blocks one or more higher priority tasks, it temporarily assumes a higher priority.

Specific Methods:

- Priority Inheritance Protocol (PIP), for static priorities
- Priority Ceiling Protocol (PCP), for static priorities
- Stack Resource Policy (SRP), for static and dynamic priorities
- others ...

7 - 14

Priority Inheritance Protocol (PIP)

Assumptions:

n tasks which cooperate through m shared resources; fixed priorities, all critical sections on a resource begin with a $wait(S_i)$ and end with a $signal(S_i)$ operation.

Basic idea:

When a task J_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

Terms:

We distinguish a fixed **nominal priority** P_i and an **active priority** p_i larger or equal to P_i . Jobs $J_1, \dots, J_n$ are ordered with respect to nominal priority where J_1 has **highest priority**. Jobs do not suspend themselves.

Priority Inheritance Protocol (PIP)

Algorithm:

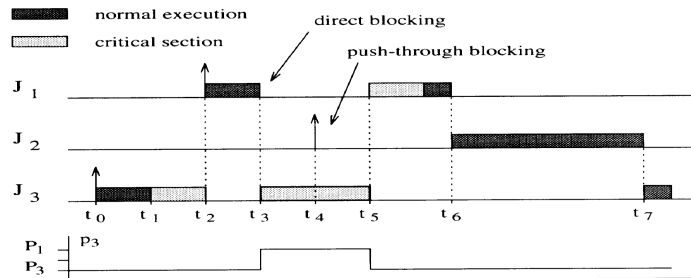
- Jobs are scheduled based on their **active priorities**. Jobs with the same priority are executed in a FCFS discipline.
- When a job J_i tries to **enter a critical section** and the resource is blocked by a lower priority job, the job J_i is blocked. Otherwise it enters the critical section.
- When a job J_i is **blocked**, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k = p_i$ (it **inherits** the priority of the highest priority of the jobs blocked by it).
- When J_k exits a critical section, it **unlocks** the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k , then p_k is set to P_k , otherwise it is set to the highest priority of the jobs still blocked by J_k .
- Priority inheritance is **transitive**, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.

7 - 15

7 - 16

Priority Inheritance Protocol (PIP)

Example:



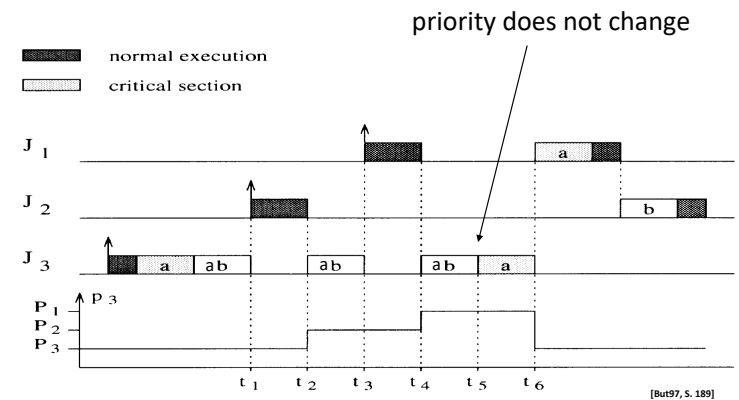
Direct Blocking: higher-priority job tries to acquire a resource held by a lower-priority job

Push-through Blocking: medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks

7 - 17

Priority Inheritance Protocol (PIP)

Example with nested critical sections:

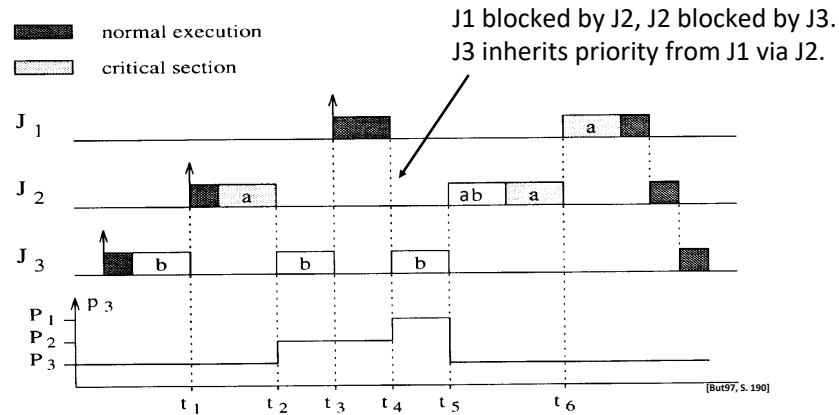


[But97, S. 189]

7 - 18

Priority Inheritance Protocol (PIP)

Example of transitive priority inheritance:

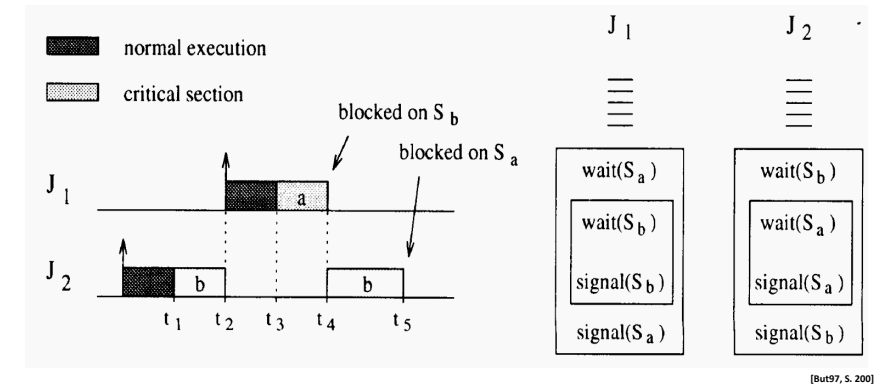


7 - 19

Priority Inheritance Protocol (PIP)

Still a Problem: Deadlock

.... but there are other protocols like the Priority Ceiling Protocol ...



7 - 20

The MARS Pathfinder Problem (1)

"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data."



7 - 21

The MARS Pathfinder Problem (2)

"VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks."

"Pathfinder contained an 'information bus', which you can think of as a shared memory area used for passing information between different components of the spacecraft."

- A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes)."

7 - 22

The MARS Pathfinder Problem (3)

- The meteorological data gathering task ran as an infrequent, low priority thread. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
- The spacecraft also contained a communications task that ran with medium priority.

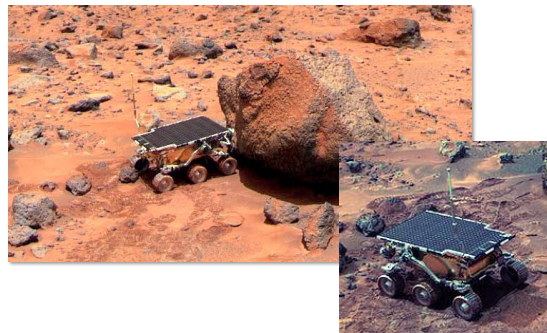
High priority:	retrieval of data from shared memory
Medium priority:	communications task
Low priority:	thread collecting meteorological data

7 - 23

Priority Inversion on Mars

Priority inheritance also solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to “on”. When the software was shipped, it was set to “off”.

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to “on”, while the Pathfinder was already on the Mars [Jones, 1997].



7 - 25

The MARS Pathfinder Problem (4)

“Most of the time this combination worked fine.

However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running.

After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset. This scenario is a classic case of priority inversion.”

7 - 24

Timing Anomalies

7 - 26

Timing Anomaly

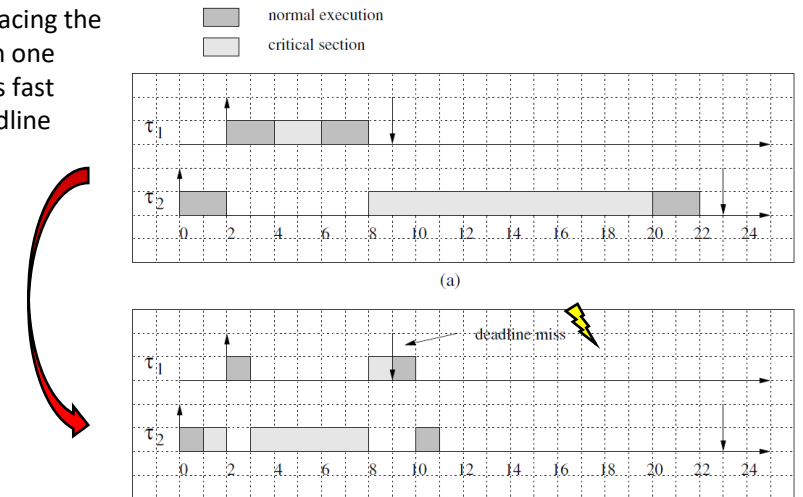
Suppose, a real-time system works correctly with a given processor architecture. Now, you replace the processor with a faster one. Are real-time constraints still satisfied?

Unfortunately, this is not true in general. *Monotonicity* does not hold in general, i.e., making a part of the system operate faster does not lead to a faster system execution. In other words, *many software and systems architectures are fragile*.

There are usually many timing anomalies in a system, starting from the microarchitecture (caches, pipelines, speculation) via single processor scheduling to multiprocessor scheduling.

Single Processor with Critical Sections

Example: Replacing the processor with one that is twice as fast leads to a deadline miss.

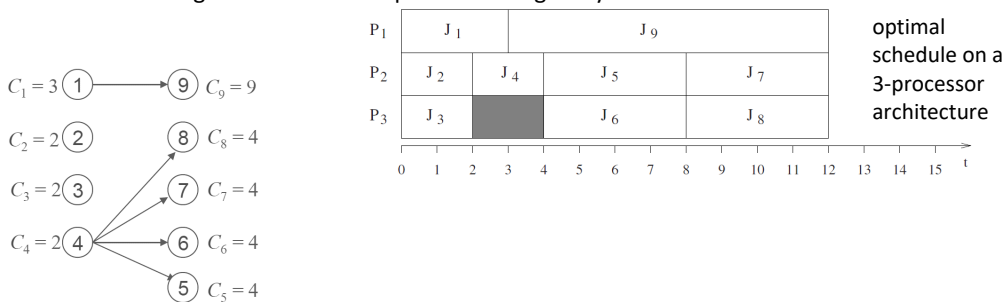


7 - 27

7 - 28

Multiprocessor Example (Richard's Anomalies)

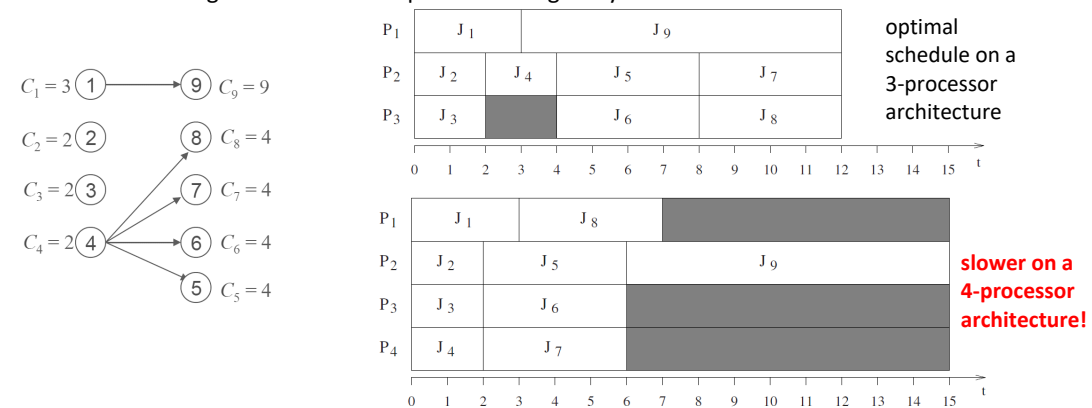
Example: 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.



7 - 29

Multiprocessor Example (Richard's Anomalies)

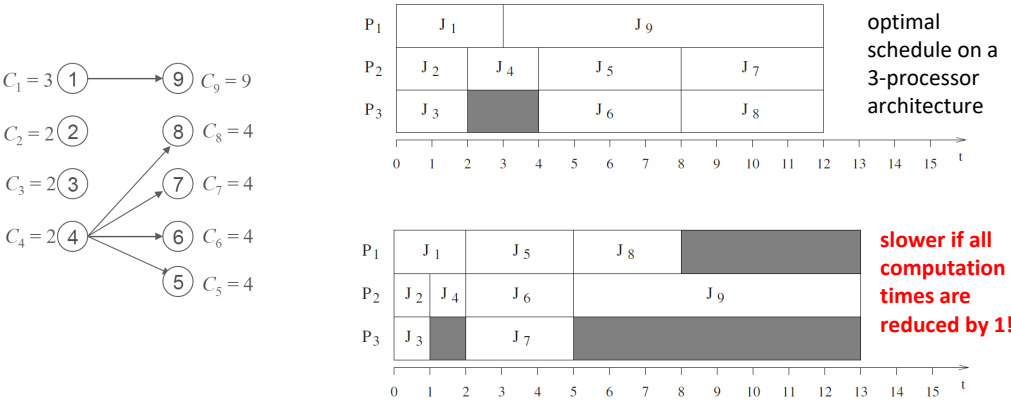
Example: 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.



7 - 30

Multiprocessor Example (Richard's Anomalies)

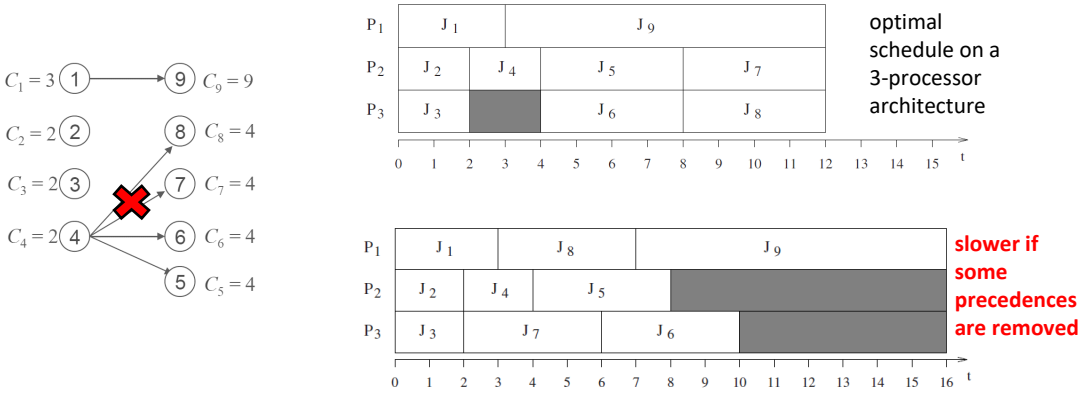
Example: 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.



7 - 31

Multiprocessor Example (Richard's Anomalies)

Example: 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.



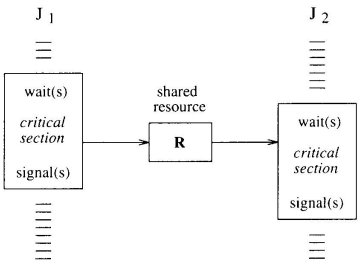
7 - 32

Communication and Synchronization

Communication Between Tasks

Problem: the use of shared memory for implementing communication between tasks may cause priority inversion and blocking.

Therefore, either the implementation of the shared medium is “thread safe” or the data exchange must be *protected by critical sections*.



7 - 33

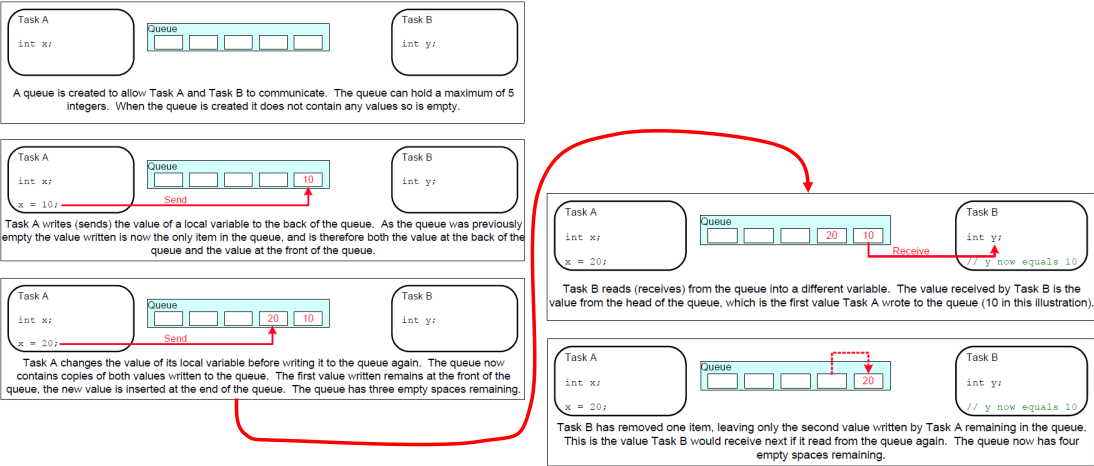
7 - 34

Communication Mechanisms

Synchronous communication:

- Whenever two tasks want to communicate they must be synchronized for a message transfer to take place (rendez-vous).
- They have to wait for each other, i.e. both must be at the same time ready to do the data exchange.
- Problem:**
 - In case of dynamic real-time systems, estimating the maximum blocking time for a process rendez-vous is difficult.
 - Communication always needs synchronization. Therefore, the timing of the communication partners is closely linked.

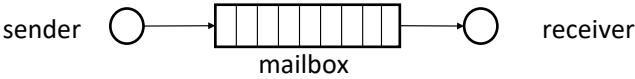
Example: ThreadX (ES-Lab)



Communication Mechanisms

Asynchronous communication:

- Tasks do not necessarily have to wait for each other.
- The sender just deposits its message into a channel and continues its execution; similarly the receiver can directly access the message if at least a message has been deposited into the channel.
- More suited for real-time systems than synchronous communication.
- Mailbox:** Shared memory buffer, FIFO-queue, basic operations are send and receive, usually has a fixed capacity.
- Problem:** Blocking behavior if the channel is full or empty; alternative approach is provided by cyclical asynchronous buffers or double buffering.

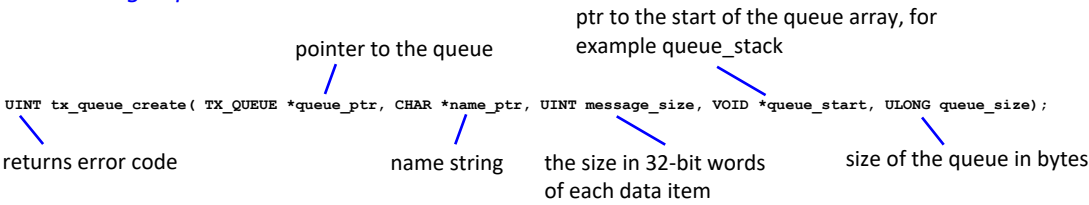


Example: ThreadX (ES-Lab)

Allocating (static) memory for the queue:

```
uint8_t queue_stack[QUEUE_STACK_SIZE];
```

Creating a queue:



Example: ThreadX (ES-Lab)

Sending item to a queue:

```
UINT tx_queue_send( TX_QUEUE *queue_ptr, VOID *source_ptr, ULONG wait_option);
```

pointer to the queue

a pointer to the data to be copied into the queue

returns TX_SUCCESS if item was successfully added to queue

the maximum amount of time (number of timer ticks) the task should remain in the blocked state to wait for space to become available on the queue

Example: ThreadX (ES-Lab)

Receiving item from a queue:

```
UINT tx_queue_receive( TX_QUEUE *queue_ptr, VOID *destination_ptr, ULONG wait_option);
```

pointer to the queue

a pointer to the storage for the data copied from the queue

returns TX_SUCCESS if item was successfully removed to queue

the maximum amount of time (number of timer ticks) the task should remain in the Blocked state to wait for item to become available on the queue

Example:

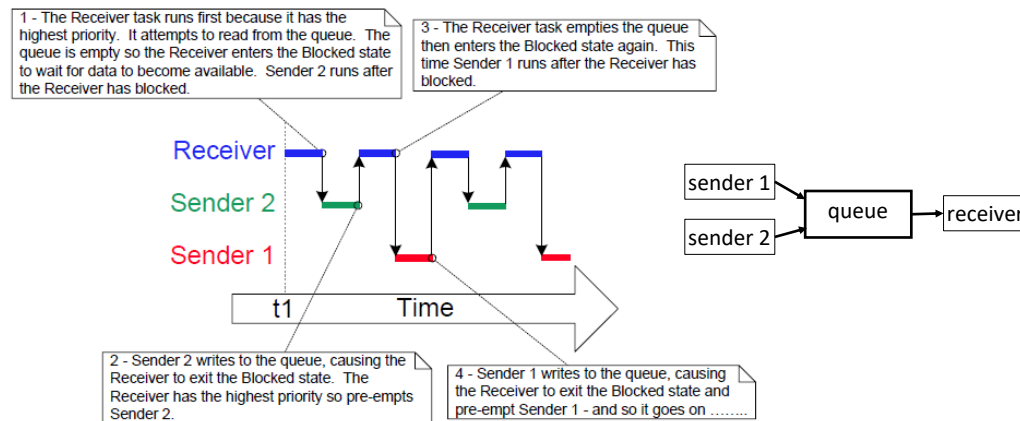
- Two sending tasks with equal priority 1 and one receiving task with priority 2.
- ThreadX schedules tasks with equal priority in a round-robin manner: A blocked or preempted task is put to the end of the ready queue for its priority. The same holds for the currently running task at the expiration of the time slice.

7 - 39

7 - 40

Example: ThreadX (ES-Lab)

Example cont.:



7 - 41

Embedded Systems

Lecture 10.

Hardware Software Architecture.

Sensors and Actuators - Digital Analog Converters PWM

© Lothar Thiele

Computer Engineering and Networks Laboratory

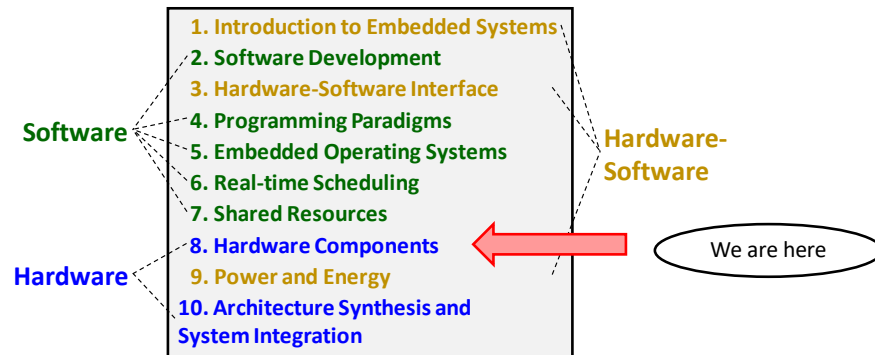
Michele Magno

D-ITET center for project based learning

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Where we are ...



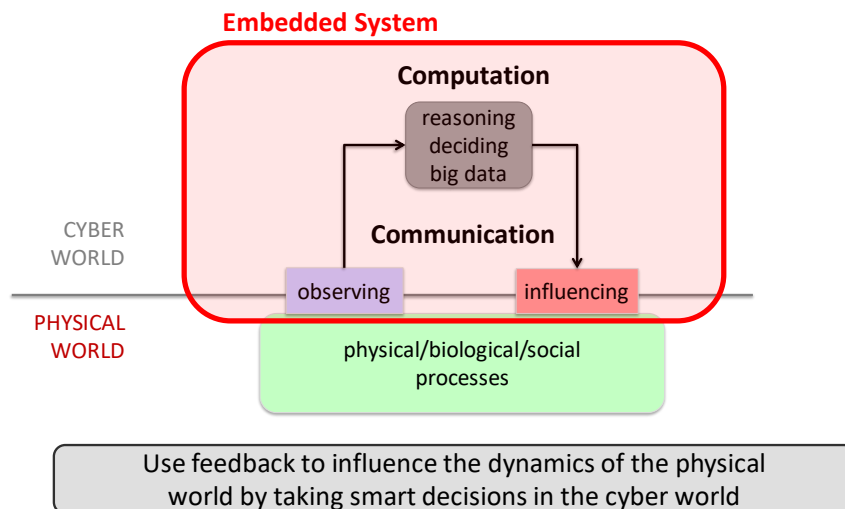
Last 3 weeks!

05.12 / Thiele-Magno	Lecture 10: Hardware-Components, Sensors and actuators, System Integration-PWM
12.12 / Magno	Lecture 11: Chapter 9 Low Power and Energy
19.12 / Magno	Lecture 12: Chapter 10 Architecture Synthesis

3 - 2

3 - 3

Today class will focus on Observing and Influencing.



Do you Remember ?

3 - 4

3 - 5

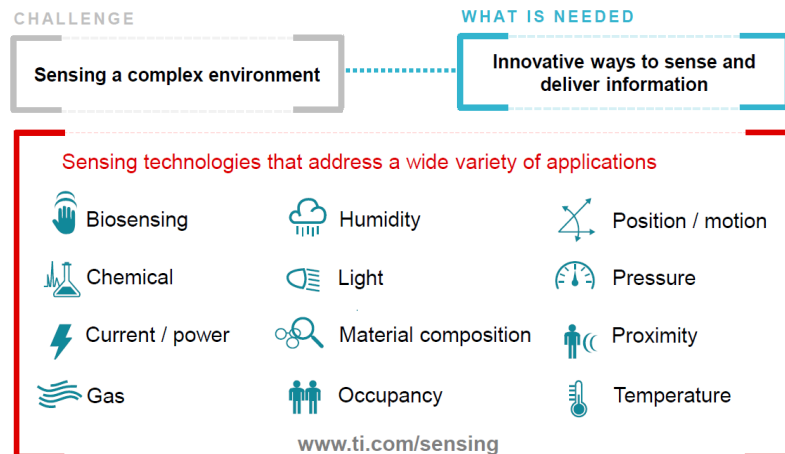
Brief recap of the Embedded Systems Overview

- an embedded system consists of
 - an actuator
 - a sensor
 - an embedded processor
 - and often a communication network
- embedded systems are
 - reliable
 - consider many demands (energy, code size, ...)
 - designed for a specific application
 - often real-time
 - often reactive



3 - 6

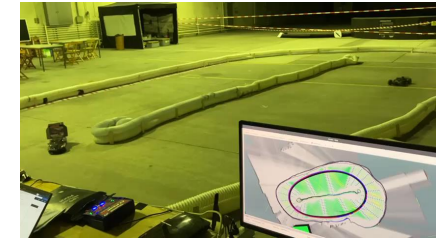
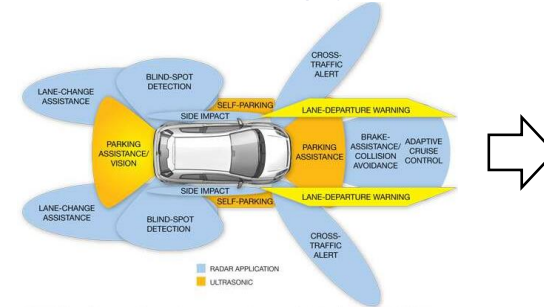
Sensing is a Must



3 - 8

Sensors and actuators

- A sensor is a device that **measures** a physical quantity
- Input / "Read from physical world"

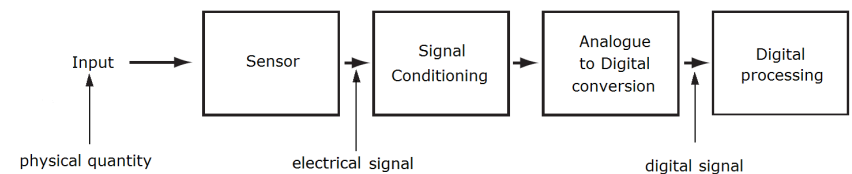


- An actuator is a device that **modifies** a physical quantity
- Output / "Write to physical world"

3 - 7

Sensors data acquisition

Data acquisition system components:

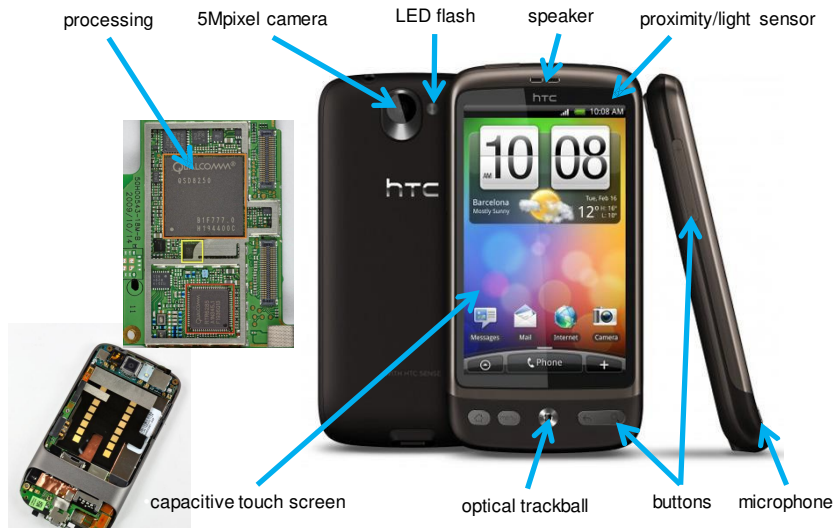


Sensors:

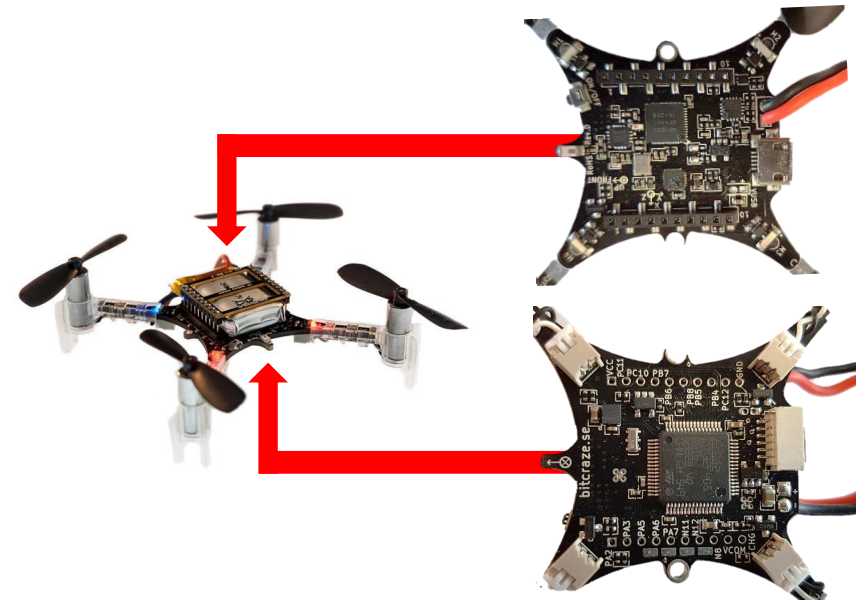
Convert analogue measurements of physical quantities (e.g. temperature, pressure, humidity, velocity, flow-rate, linear motion, position) into electrical signals (voltage or current).

3 - 9

Example of Embedded system

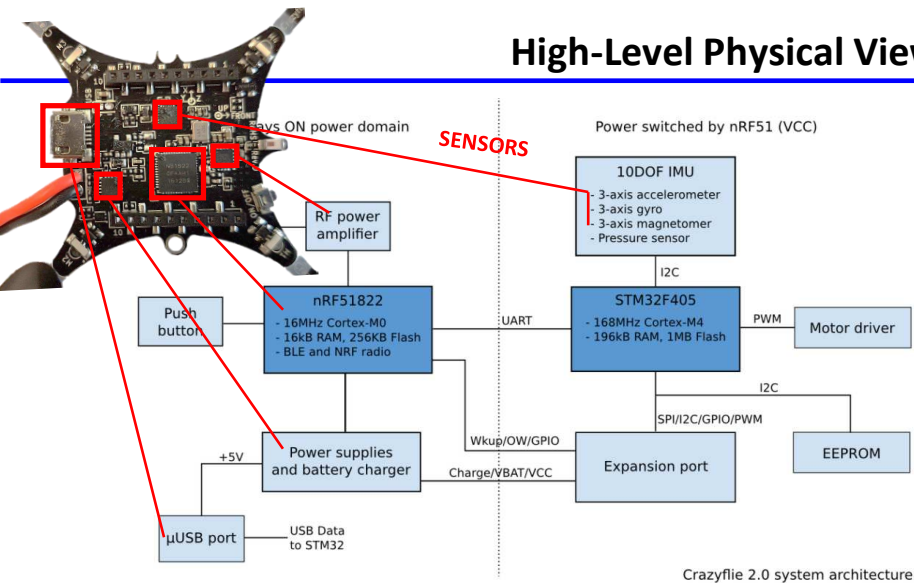


3 - 10



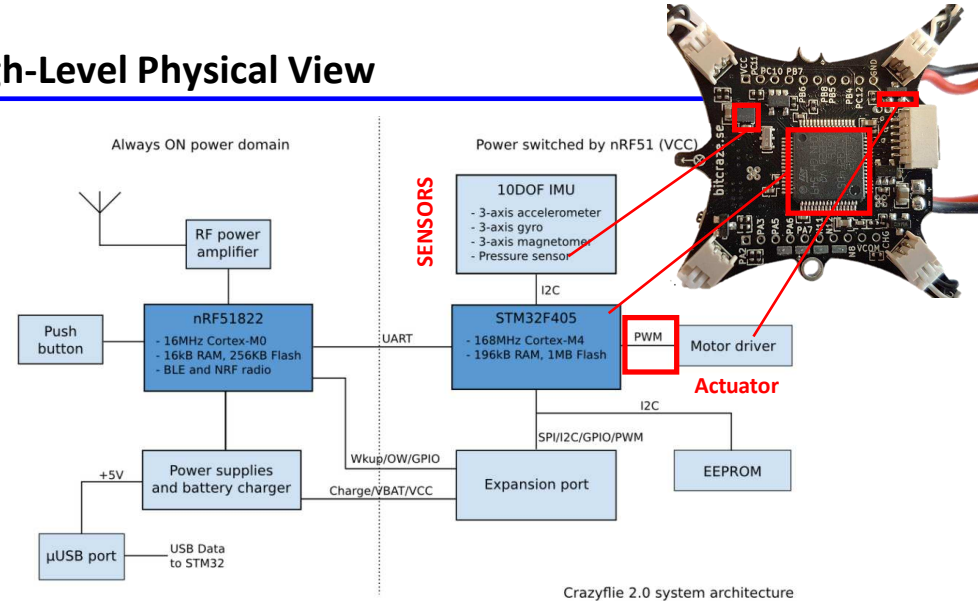
3 - 11

High-Level Physical View



3 - 12

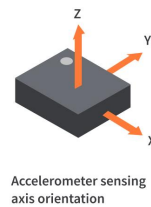
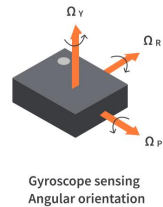
High-Level Physical View



3 - 13

Accelerometer, Gyroscope (ES-LAB) and Magnetometer

- Accelerometer senses acceleration measurements
 - Usually in g or $\frac{m}{s^2}$
- Gyroscope senses angular rotation relative to the gravitational acceleration vector
 - Usually in degrees per second



3 - 14

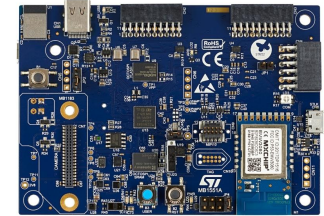
ODR vs Throughput – Bandwidth limits

- ODR is also the sampling rate.
 - 1 sample is 16 bit
 - Max 6.667KHz - Throughput $6667\text{sps} \times 16\text{bit} = 106.67 \text{ kilobit/s}$
 - Max possible I2C speed is 400kHz.
 - Overhead of I2C for accessing the data need to be considered.
- SPI has a high bandwidth of over 3Mbps and lower overhead.

3 - 16

ST ISM330DHCX – 6axes Inertial Measurement Unit

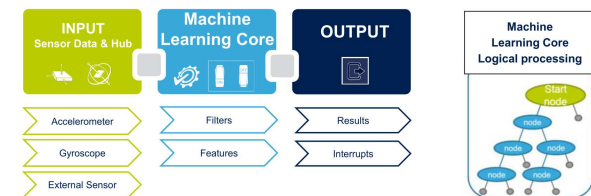
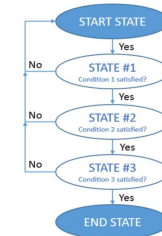
- I2C or SPI
- Inertial Measurement Unit
 - 3D digital accelerometer:
 - Full-scale from $\pm 2g$ to $\pm 16g$
 - 16-bit resolution
 - Output Data Rate from 1.6Hz to 6.66kHz
 - 3D digital gyroscope
 - Full-scale from $\pm 125\text{dps}$ to $\pm 4000\text{dps}$
 - 16-bit resolution
 - ODR from 1.6Hz to 6.66kHz
- Low current consumption (*down to μA*)



3 - 15

ST ISM330DHCX

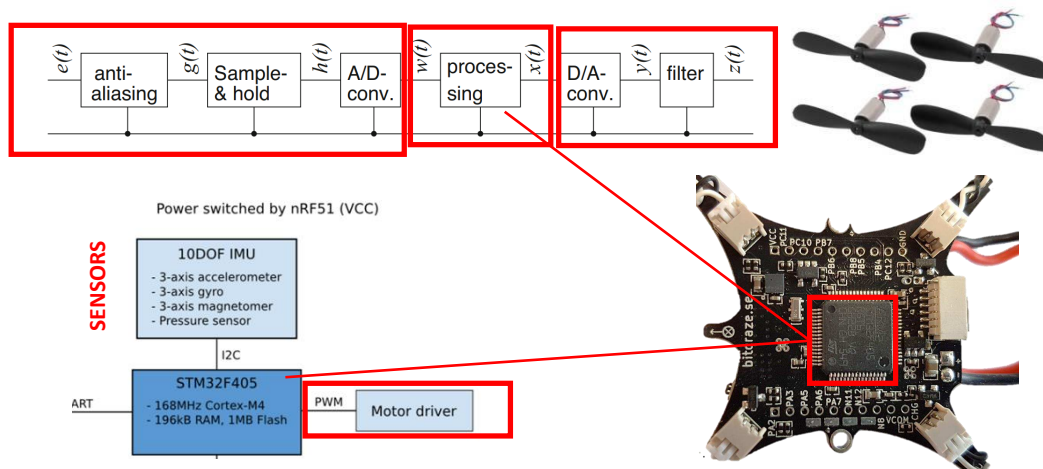
- Additional features:
 - Finite State Machine
 - Machine Learning Core
 - Event-detection
 - Free-fall
 - Wakeup
 - Click and double-click
 - Activity recognition



In Sensor Embedded System!

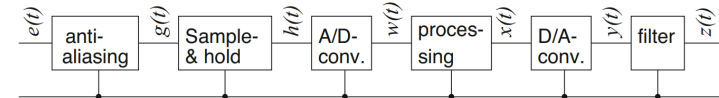
3 - 17

From Sensors to Actuators

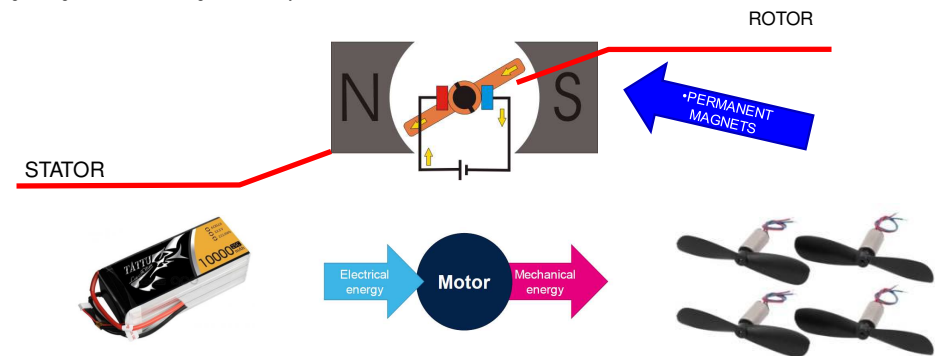


3 - 18

The full Chain! Digital to Analog Converters



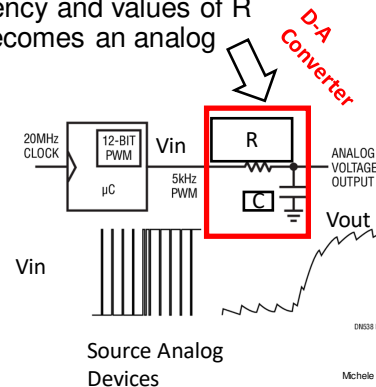
An electric motor is a device converting electrical energy into mechanical energy (generally a torque). This conversion is usually obtained through the generation of a magnetic field by means of a current flowing into one or more coils.



3 - 19

Pulse Width Modulation (PWM)

- The average value can be extracted from the PWM stream with a low-pass filter
- In this case, and as long as PWM frequency and values of R and C are appropriately chosen, V_{out} becomes an analog output



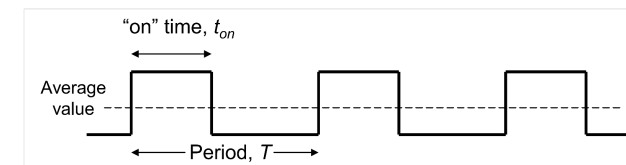
- In practice, this sort of filtering is not always required; many physical systems have response characteristics which, in reality, act like low pass filters

Michele Magno 05.12.2022 20
3 - 20

Pulse Width Modulation (PWM)

- The period is normally kept constant, and the pulse width, or "on" time is varied
- The duty cycle is the proportion of time that the pulse is 'on' or 'high', and is expressed as a percentage:

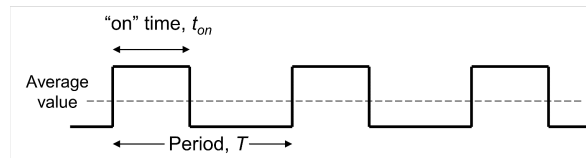
$$\text{duty cycle} = 100\% * (\text{pulse on time}) / (\text{pulse period})$$



Michele Magno 05.12.2022 21
3 - 21

Pulse Width Modulation (PWM)

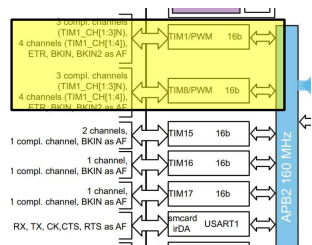
- Whatever duty cycle a PWM stream has, there is an average value, as indicated by the dotted line
- If the on time is small, the average value is low; if the on time is large, the average value is high
- By controlling the duty cycle, we control this average value



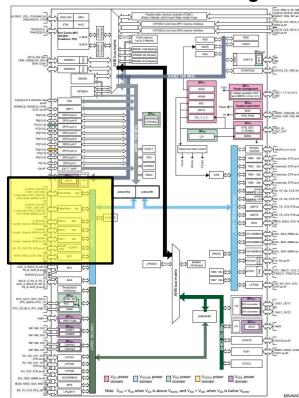
Michele Magno 05.12.2022 22
3 - 22

PWM – STM32U585xx

2 x 16-bit PWM timer



STM32U585xx block diagram

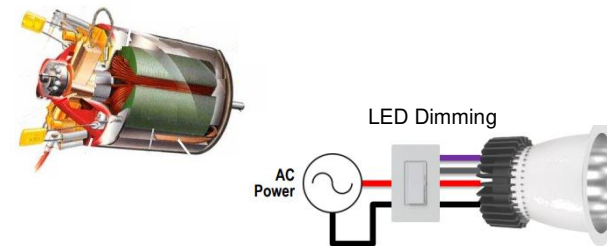


<https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html>
→ Datasheet, page 21

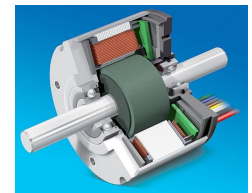
3 - 24

Pulse Width Modulation (PWM)

- The voltage supplied to a DC motor is proportional to the duty cycle
- Both brushed and brushless motors can be used with PWM
- Both analog and digital control techniques and components are available



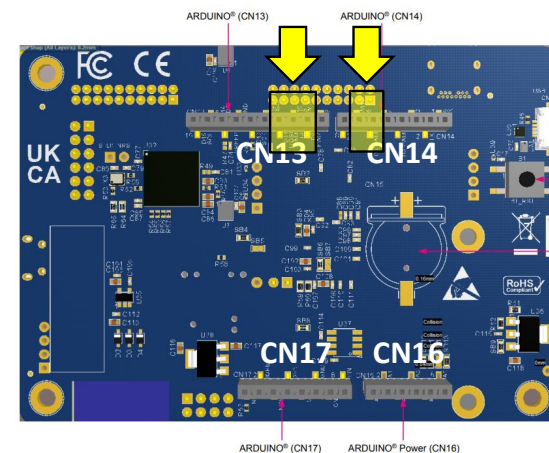
Pulse Width Modulation - Val Tocitu,



Jason Kulpe, Alex Mariuzza

Michele Magno 05.12.2022 23
3 - 23

PWM – External Connection



<https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html>
→ Documents → UM2839, page 34

Connector	Pin number	Pin name	Signal name	STM32L4 pin	Function
CN17	1	NC	-	-	-
	2	IREF	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	SV3	-	-	3.3 V I/O
	5	SV1	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
	8	VIN	-	-	Power input
CN16	1	A0	ARD_ADC_A0	PC0	ADC
	2	A1	ARD_ADC_A1	PC2	ADC
	3	A2	ARD_ADC_A2	PC4	ADC
	4	A3	ARD_ADC_A3	PC5	ADC
	5	A4	ARD_ADC_A4	PA7	ADC / I2C1_SDA
	6	A5	ARD_ADC_A5	PA6	ADC / I2C1_SCL
	10	SCL015	I2C1_SCL	PB8	I2C1_SCL
	9	SDA014	I2C1_SDA	PB9	I2C1_SDA
	8	VREFP	VREFP	-	VREFP
	7	GND	-	-	GND
CN13	6	SDIO_D13	ARD_D13_SPL_SCK	PE13	SPI1_SCK / LD2
	5	SDIO_D12	ARD_D12_SPL_MISO	PE14	SPI1_MISO
	4	PWMMSDIO11	ARD_D11_SPL_MOSI	PE15	SPI1_MOSI / TIM1_CH4N
	3	PWMMSDIO10	ARD_D10_TM_SPL_NSS	PE12	SPI1_NSS / TIM1_CH1
	2	PWMMSDIO9	ARD_D9_TM	PA6	TM1_CH1
	1	SV3	ARD_D8_TM	PA7	SV3
	7	PWMMSDIO8	ARD_D8_TM	PA6	TM1_CH1
	6	PWMMSDIO7	ARD_D7_TM	PA5	TM1_CH1
	5	PWMMSDIO6	ARD_D6_TM	PA4	TM1_CH1
	4	PWMMSDIO5	ARD_D5_TM	PA3	TM1_CH1
CN14	4	PWMMSDIO3	ARD_D3_TM	PA2	TM1_CH1
	3	SD2	ARD_D2_JD	PD15	GPIOS
	2	TXD1	ARD_D1_TX	PD6	UART3_TX
	1	RXD0	ARD_D0_RX	PD5	UART3_RX

3 - 25

What did you Learn?

- Introduction of Sensors and Actuators in Embedded Systems
 - Sensors
 - ODR vs Bandwidth
- DAC to control other device
 - PWM
 - Example of motors

3 - 26

Projects at PBL

3 - 27

Project-based Learning Centre @ D-ITET



Future Learning Initiative (ETH+ Project)

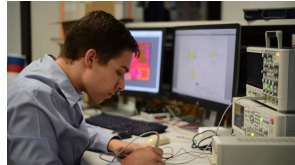
The Future Learning Initiative (FLI) leverages and builds upon our existing strengths in secondary and higher education to develop the **world's leading centre for research in the science of learning** and advance ETH's reputation as the globally leading learning institution.

Placeholder for organisational unit name / logo
(edit in slide master via "View" -> "Slide Master")

Main Goals of the Center

To build project-based learning activities at the department level with a centralized structure.

- In view of the upcoming bachelor projects, the center can help to alleviate the institutes' project supervision load
 - P&S Courses
 - Bachelor group projects and Bachelor Thesis
 - Semester-Master projects.
- Flagship project for a large number of students
 - Interdisciplinary at department level
 - Involving a high number of bachelor students.
 - Open also so semester and master students in collaboration with ITET professors
- Industrial involvement in projects and sponsorship
- Research on Embedded Systems and Smart Sensors**



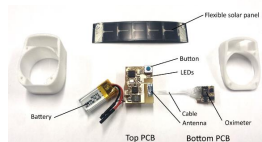
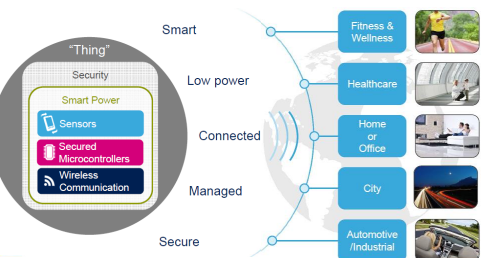
Michele Magno | 05/12/2022 | 30

DITET

Embedded Systems and Smart IoT – Why?

Almost every Electronic Gadget around us is an Embedded System!

- We focus on
 - Wireless Sensor Devices/Networks:
 - Building Systems to Wireless Smart Sensors
 - Long-term monitoring/Communication/energy harvesting
 - Ultra Wide band, Localization, Bluetooth low energy
 - Energy Harvesting for low power sensing
 - Smart sensing (Machine learning on the edge)
 - Make low power sensing smart
 - Drones, Industrial Applications, Wearable devices etc
 - Novel Sensors collaboration with Google Soli, STMicroelectronics, Sony, etc.
 - Camera, DVS Camera, Thermal Camera, 8x8 time of Flight Sensors.



Placeholder for organisational unit name / logo
(edit in slide master via "View" > "Slide Master")

Identity of PBL: Embedded Systems and Wireless Smart Sensors

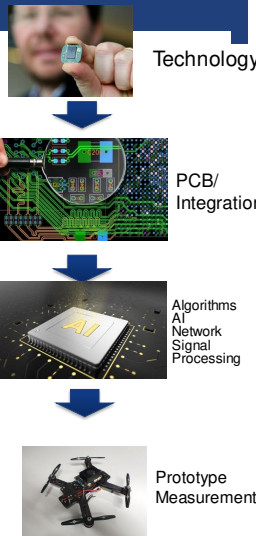
- An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is **designed for a specific function** or for specific functions within a larger system.
- Tiny Machine Learning on Embedded systems (especially MCU)
- Wireless Sensor Nodes and Networks
- Smart Sensors And Autonomous Energy efficient devices (including drones)
- Energy Harvesting for miniaturized devices.
- Systems for application specific!**
 - Biomedical, Smart Farm, Home Automation, Industrial Monitoring, Drones, Surveillance, etc.

DITET

Michele Magno | 05/12/2022 | 31

Embedded Systems – What will you learn?

- Requirements (depending on project)
 - Interest in Sensors and Sensor Network, energy supplies and harvesting
 - C, Python/Matlab, Machine-Learning Algorithms
 - System integration (better with some analog electronics at system level)
- What is the focus of the projects
 - System design and implementation (including PCB Design)
 - Energy Harvesting and power management
 - Embedded programming and application view
 - Measurements in laboratory and field
 - Machine Learning and microcontroller classifiers (or pattern recognition)
 - Wireless communication and High-level algorithms (Python/Matlab)
 - GPS, UWB, Drones, indoor localization with RF etc



Placeholder for organisational unit name / logo
(edit in slide master via "View" > "Slide Master")

Embedded Systems – Previous Highlights

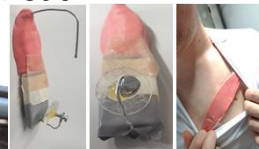
- Smart wireless system and IoT Applications
 - Self-sustaining **Wearable devices** and Sensor nodes
 - Long terms monitoring with BTLE and other radios
 - Energy Harvesting and **Machine Learning for Embedded Systems**
 - WSN for **indoor monitoring UWB** (Light system and Home Automation)
 - High-Precision geo-localization with LoRa and RTK GPS
 - Radar, not commercial sensors** for wearable and **drones**.
 - RF power transmission** for perpetual WSN (patent Proposal)
 - Acoustic detectors and zero-power touch receiver/sensors (PATENT)
 - Underwater** communication (patent proposal)
- 15+ awards won in from 2018-2022 in high level international competitions!**



Placeholder for organisational unit name / logo
(edit in slide master via "View" > "Slide Master")

Flagship Project Bachelor Thesis Formula works Great!!

- 40+ Bachelor Students** involved in different projects in 2022.
- Health and Smart Patch City of Zurich**
 - Involved with EWZ and other industry for smart cities
 - ITET News, Zurich News papers,
- Autonomous Drones: First Slam system with ToF**
 - Journal papers
- F1Tenth**
 - ICRA 2022 – 4th in the world
 - 1st in F1Tenth Lausitzting 2022!**
 - Was missing only the winner of ICRA



<https://pbl.ee.ethz.ch/flagship-p-projects.html>

Michele Magno | 05/12/2022 | 36

DITET

Flagship Projects

Flagship projects

Flagship projects are overarching interdisciplinary projects that allow for a multitude of student projects and are based on a real world application of embedded systems. Flagship projects involve 5 to 6 bachelor's and masters students that work in a self-organized group to achieve a larger task within the general flagship project.

If you are interested in flagship projects **please apply before the 31st of January 2022.**

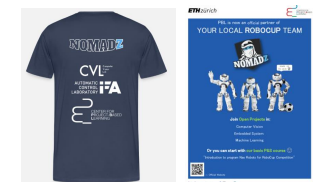


Placeholder for organisational unit name / logo
(edit in slide master via "View" > "Slide Master")



PBL Support to External «Flagship» projects (PBL Official partner)

- 2° to Not-a-boring-competition from Elon Musk
 - 1 semester projects
 - 2 bachelor thesis
 - PBL Official spot for embedded systems and electronics
- Aris projects – Educational Rockets
 - PBL Embedded systems / Sensors / Electronics
- Robocup D-ITET
 - PBL became an official partner on Embedded systems
 - P&S
 - Student projects activated. **Our Student and PBL staff did the Referee Visual challenge and 2nd place!**
- AMZ autonomous eclectic car
 - Working to be an official partner**



Michele Magno | 05/12/2022 | 37

DITET

2 New Flagship projects from 2022/2023

PBL Flagship Project 2022-2023

Truly Autonomous Legged Robots for Assisted Living and more
With a Bachelor/Semester/Master Thesis/ Freelancer and Staff at PBL

Would you like to design, develop and test an Autonomous Legged robot for real-world applications such as blind person assistance and other applications?

Would you like to work with embedded systems, sensors including camera and Lidar, Embedded Control, hardware and software design, Controlling Robotic Arms, and with state of art legged Robots from Unitree?

We are looking students to work in the following areas:

- Embedded Control
- Manipulation of robotic Arm
- AI-Perception for autonomous navigation with sensors
- Computer vision for perception and planning
- And many other topics
- Participation on reviews, competitions

Project in collaboration with:

Find this and more projects at PBL

For info: michele.magno@pbl.ee.ethz.ch




PBL Flagship Project 2022-2023

Open Research-Platform: Low Power Smart Glasses
With a Bachelor/Semester/Master Thesis/ Freelancer and Staff at PBL

Would you like to design, develop the novel, multi-sensors and open research platform for intelligent smart glasses?

Would you like to work with embedded systems, hardware and software design under space and low-power and wearable constraints?

We are looking students to re-design the platform to low power in mind and student interested in the following areas:

- PCB Design and System Integration
- Smart Sensors Interface and Firmware
- Android driver and apps
- Hardware-Software Co-design
- Wireless communication
- Systems and applications on Wearable

Project in collaboration with:

Find this and more projects at PBL

For info: michele.magno@pbl.ee.ethz.ch



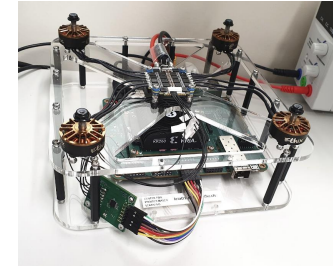
DITET

Supported by 2
P&S

Michele Magno | 05/12/2022 | 38

Projects at PBL – And industry competitions

<https://pbl.ee.ethz.ch/project/currentprojects.html>



Tomorrow we will present demo-flyers at the
AMIV Student meets lab!!

Placeholder for organisational unit name / logo
(edit in slide master via "View" > "Slide Master")

Embedded Systems

Lecture 11 . Power and Energy

© Lothar Thiele

Computer Engineering and Networks Laboratory

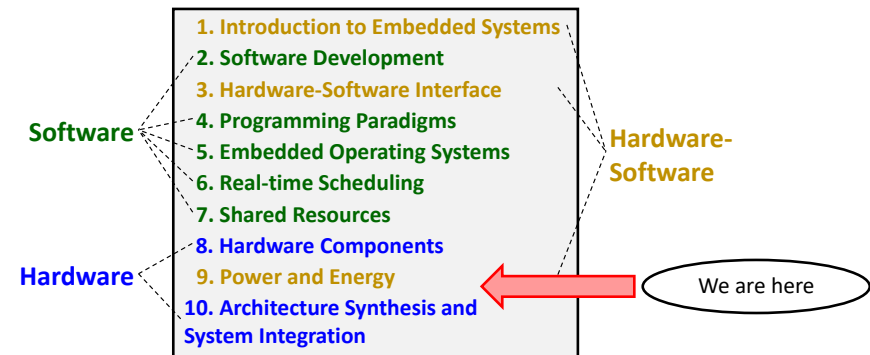
Michele Magno

D-ITET center for project based learning

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Where we are ...



Last 2 weeks!

12.12 / Magno	Lecture 11: Chapter 9 Low Power and Energy
19.12 / Magno	Lecture 12: Chapter 10 Architecture Synthesis

General Remarks

9 - 3

Power and Energy Consumption

- Statements that are true since a decade or longer:

„Power is considered as the most important constraint in embedded systems.“ [in: L. Eggermont (ed): Embedded Systems Roadmap 2002, STW]

“Power demands are increasing rapidly, yet battery capacity cannot keep up.” [in Diztel et al.: Power-Aware Architecting for data-dominated applications, 2007, Springer]

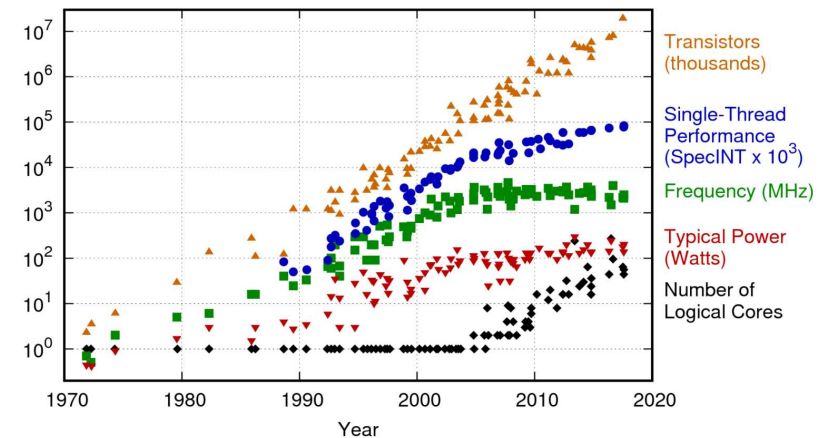
- Main **reasons** are:

- power provisioning is expensive
- battery capacity is growing only slowly
- devices may overheat
- energy harvesting (e.g. from solar cells) is limited due to the relatively low energy available density



9 - 5

Some Trends

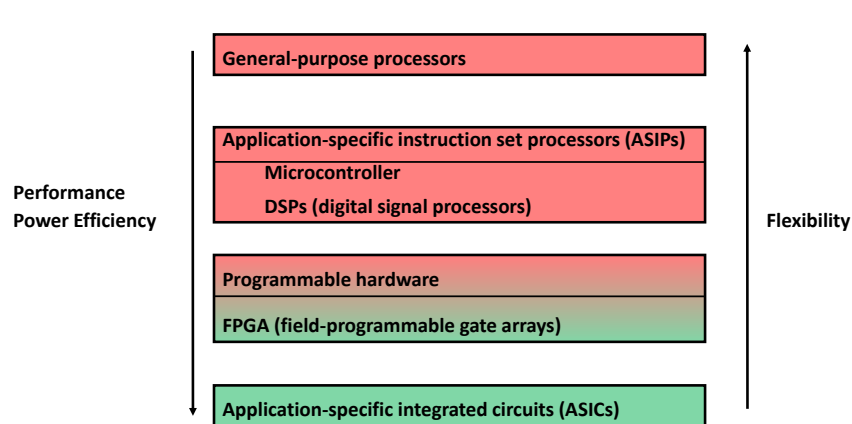


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

9 - 4

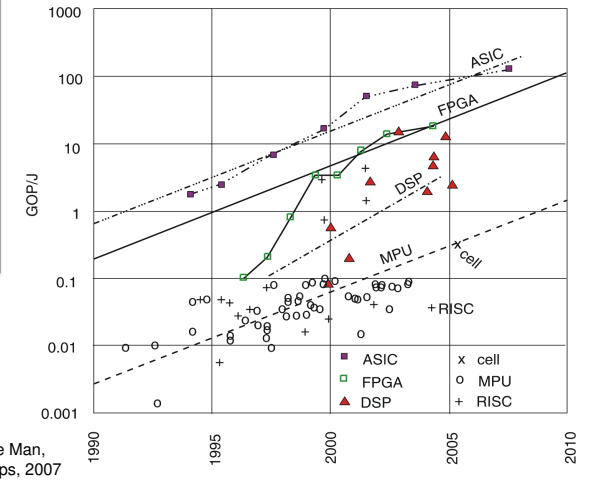
9 - 6

Implementation Alternatives



Energy Efficiency

- It is necessary to **optimize HW and SW**.
- Use **heterogeneous architectures** in order to adapt to required performance and to class of application.
- Apply **specialization techniques**.

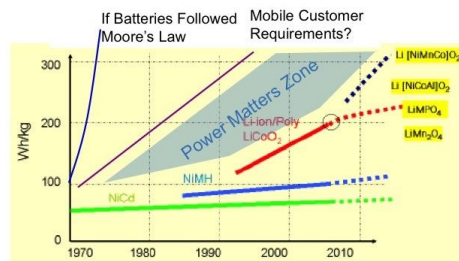


9 - 7

9 - 8

Embedded systems nightmare: Power availability

Ubiquitous computing dream of embedded systems everywhere is accompanied by the nightmare of battery replacement and disposal.



Source: Avicenne

Battery Technology is Stuck!

No Moore's Law in batteries:
2-3%/year growth

Embedded systems' lifetime depends on battery life!!

Solution

Design ultra low power smart systems and harvest energy from ambient (heat, light, radio, or vibrations...)



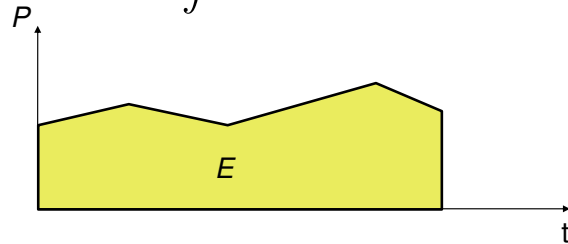
Power and Energy

9 - 9

9 - 10

Power and Energy

$$E = \int P(t)dt$$

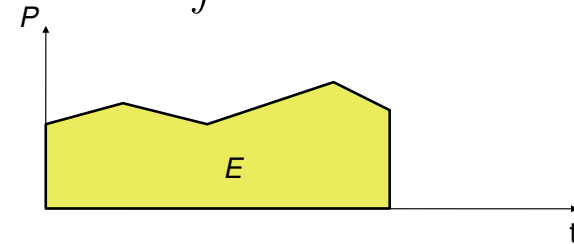


In some cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow for a faster execution.

9 - 11

Power and Energy

$$E = \int P(t)dt$$

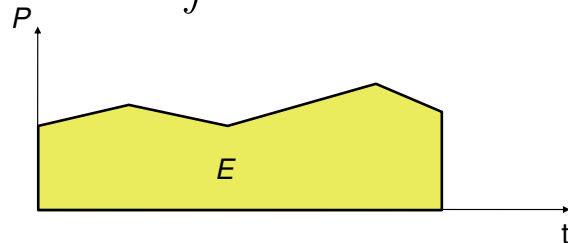


In some cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow for a faster execution.

9 - 12

Power and Energy

$$E = \int P(t)dt$$

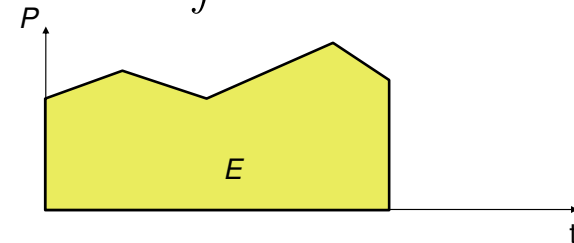


In some cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow for a faster execution.

9 - 13

Power and Energy

$$E = \int P(t)dt$$



In some cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow for a faster execution.

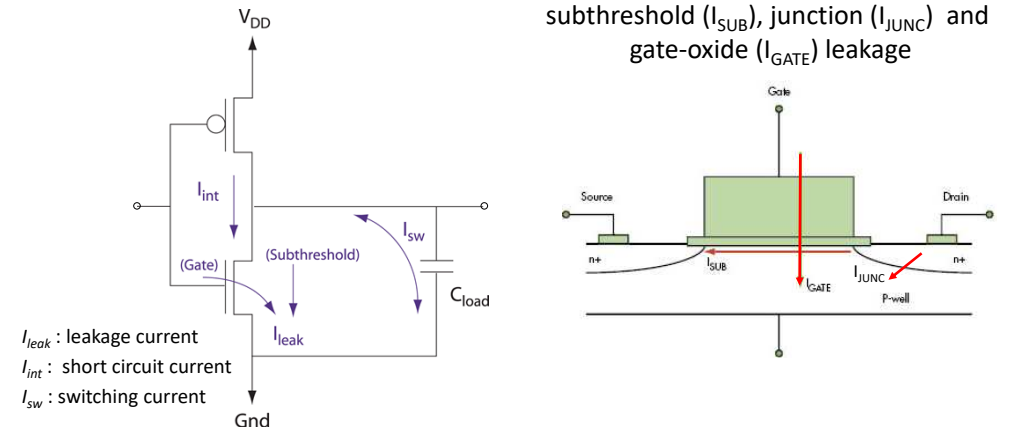
9 - 14

Low Power vs. Low Energy

- Minimizing the **power consumption** (voltage * current) is important for
 - the design of the power supply and voltage regulators
 - the dimensioning of interconnect between power supply and components
 - cooling (short term cooling)
 - high cost
 - limited space
- Minimizing the **energy consumption** is important due to
 - restricted availability of energy (mobile systems)
 - limited battery capacities (only slowly improving)
 - very high costs of energy (energy harvesting, solar panels, maintenance/batteries)
 - long lifetimes, low temperatures

9 - 15

Power Consumption of a CMOS Gate

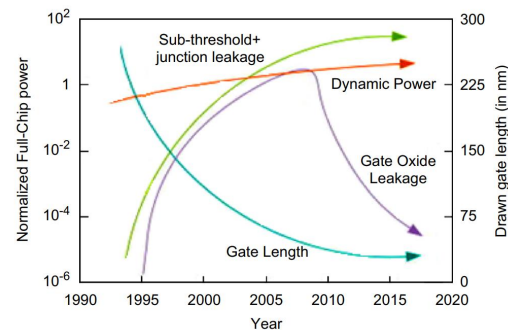


9 - 16

Power Consumption of a CMOS Processors

Main sources:

- Dynamic power consumption
 - charging and discharging capacitors
 - Short circuit power consumption: short circuit path between supply rails during switching
- Leakage and static power
 - gate-oxide/subthreshold/junction leakage
 - becomes one of the major factors due to shrinking feature sizes in semiconductor technology



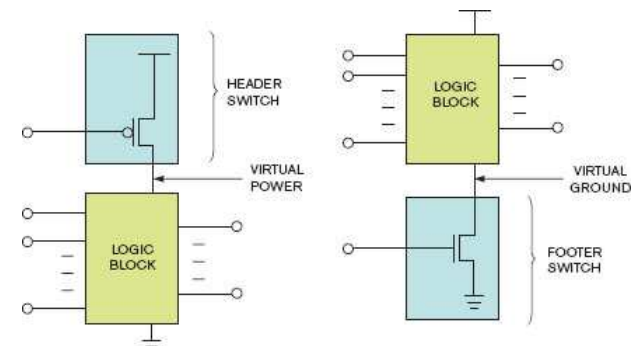
[J. Xue, T. Li, Y. Deng, Z. Yu, Full-chip leakage analysis for 65 nm CMOS technology and beyond, Integration VLSI J. 43 (4) (2010) 353–364]

9 - 17

Reducing Static Power - Power Supply Gating

Power gating is one of the most effective ways of minimizing static power consumption (leakage)

- Cut-off power supply to inactive units/components



9 - 18

Dynamic Voltage Scaling (DVS)

Average power consumption of CMOS circuits (ignoring leakage):

$$P \sim \alpha C_L V_{dd}^2 f$$

V_{dd} : supply voltage
 α : switching activity
 C_L : load capacity
 f : clock frequency

Delay of CMOS circuits:

$$\tau \sim C_L \frac{V_{dd}}{(V_{dd} - V_T)^2}$$

V_{dd} : supply voltage
 V_T : threshold voltage
 $V_T \ll V_{dd}$

Decreasing V_{dd} reduces P quadratically (f constant).

The gate delay increases reciprocally with decreasing V_{dd} .

Maximal frequency $f_{\max}$ decreases linearly with decreasing V_{dd} .

Dynamic Voltage Scaling (DVS)

$$P \sim \alpha C_L V_{dd}^2 f$$

$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\#cycles)$$

Saving energy for a given task:

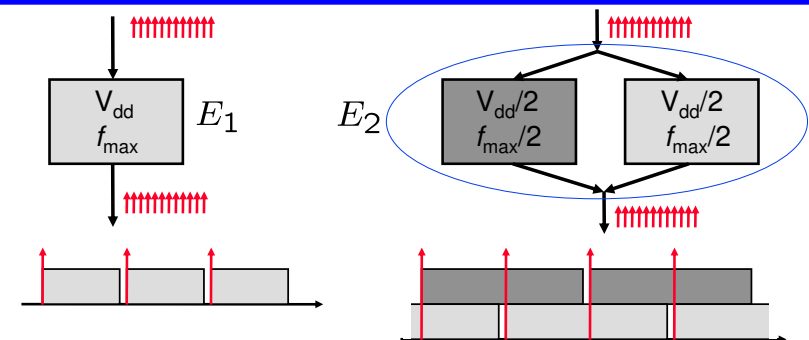
- reduce the supply voltage V_{dd}
- reduce switching activity α
- reduce the load capacitance C_L
- reduce the number of cycles $\#cycles$

9 - 19

9 - 20

Techniques to Reduce Dynamic Power

Parallelism



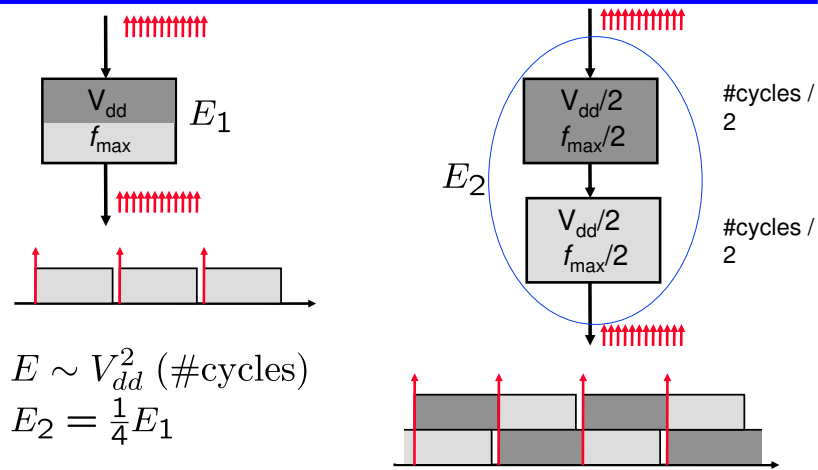
$$E \sim V_{dd}^2 (\#cycles)$$

$$E_2 = \frac{1}{4} E_1$$

9 - 21

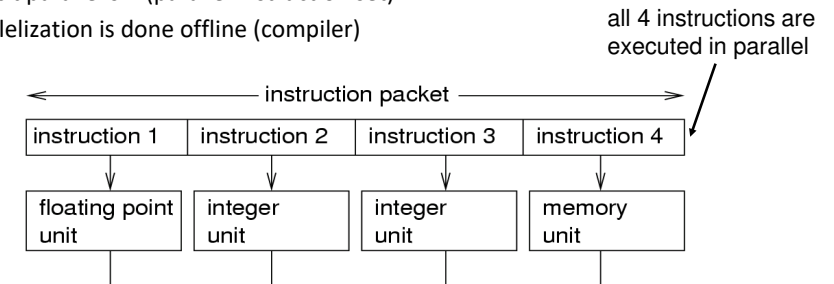
9 - 22

Pipelining



VLIW (Very Long Instruction Word) Architectures

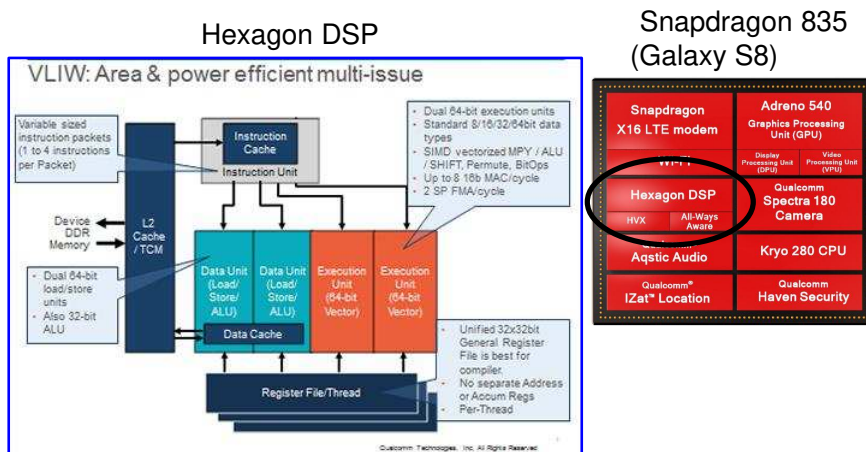
- **Large degree of parallelism**
 - many parallel computational units, (deeply) pipelined
- **Simple hardware architecture**
 - explicit parallelism (parallel instruction set)
 - parallelization is done offline (compiler)



9 - 23

9 - 24

Example: Qualcomm Hexagon



Dynamic Voltage and Frequency Scaling - Optimization

9 - 25

9 - 26

Dynamic Voltage and Frequency Scaling (DVFS)

$$P \sim \alpha C_L V_{dd}^2 f$$

energy per cycle reduce voltage → reduce energy per task

$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\# \text{cycles})$$

maximum frequency of operation gate delay reduce voltage → reduce clock frequency

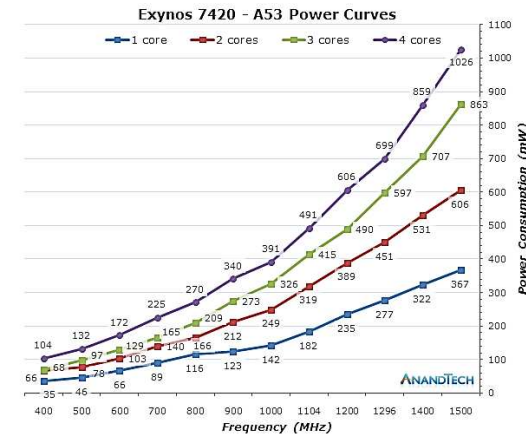
$$f \sim \frac{1}{\tau} \sim V_{dd}$$

Saving energy for a given task:

- reduce the supply voltage V_{dd}
- reduce switching activity α
- reduce the load capacitance C_L
- reduce the number of cycles $\# \text{cycles}$

Example DVFS: Samsung Exynos (ARM processor)

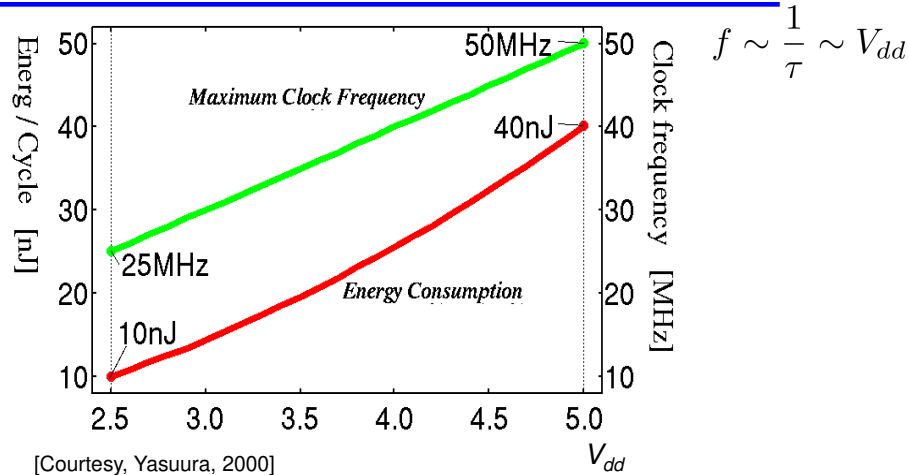
ARM processor core A53 on the Samsung Exynos 7420 (used in mobile phones, e.g. Galaxy S6)



9 - 27

9 - 28

Example: Dynamic Voltage and Frequency Scaling

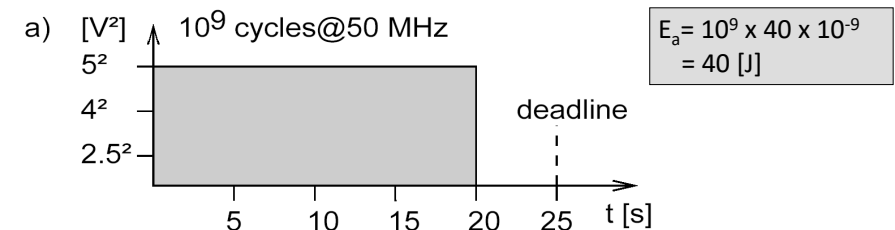


9 - 29

Example: DVFS – Complete Task as Early as Possible

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

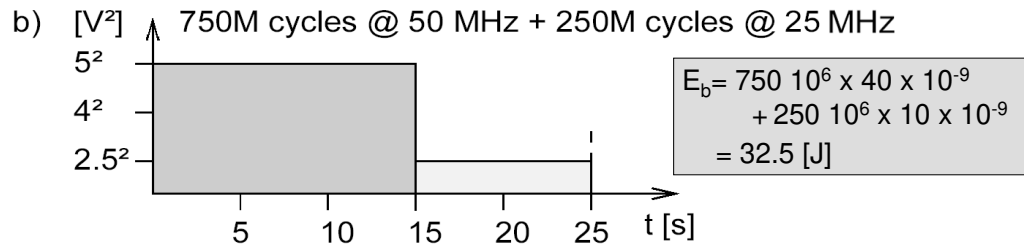
We suppose a task that needs 10^9 cycles to execute within 25 seconds.



9 - 30

Example: DVFS – Use Two Voltages

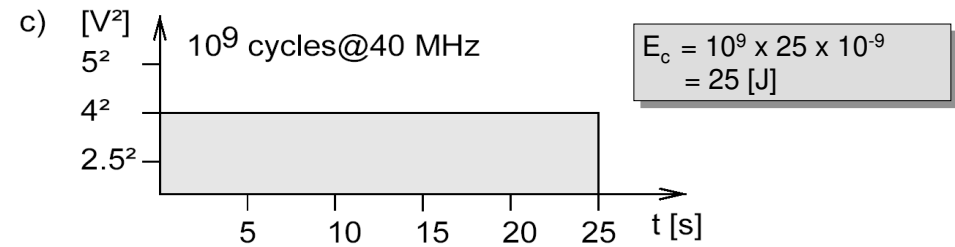
V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40



9 - 31

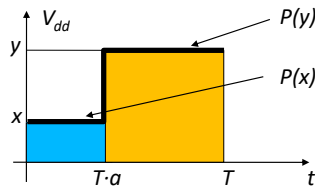
Example: DVFS – Use One Voltage

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40



9 - 32

DVFS: Optimal Strategy



Execute task in fixed time T with variable voltage $V_{dd}(t)$:

gate delay: $\tau \sim \frac{1}{V_{dd}}$

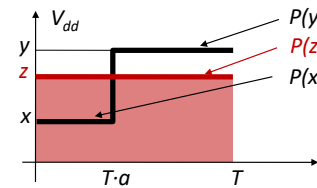
execution rate: $f(t) \sim V_{dd}(t)$

invariant: $\int V_{dd}(t) dt = \text{const.}$

- **case A:** execute at voltage x for $T \cdot a$ time units and at voltage y for $(1-a) \cdot T$ time units;
energy consumption: $T \cdot (P(x) \cdot a + P(y) \cdot (1-a))$

9 - 33

DVFS: Optimal Strategy



Execute task in fixed time T with variable voltage $V_{dd}(t)$:

gate delay: $\tau \sim \frac{1}{V_{dd}}$

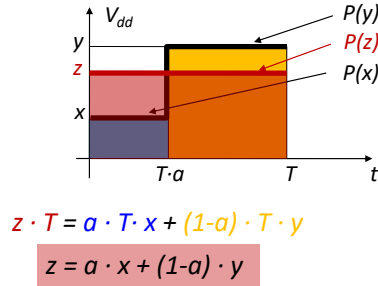
execution rate: $f(t) \sim V_{dd}(t)$

invariant: $\int V_{dd}(t) dt = \text{const.}$

- **case A:** execute at voltage x for $T \cdot a$ time units and at voltage y for $(1-a) \cdot T$ time units;
energy consumption: $T \cdot (P(x) \cdot a + P(y) \cdot (1-a))$
- **case B:** execute at voltage $z = a \cdot x + (1-a) \cdot y$ for T time units;
energy consumption: $T \cdot P(z)$

9 - 34

DVFS: Optimal Strategy



- **case A:** execute at voltage x for $T \cdot a$ time units and at voltage y for $(1-a) \cdot T$ time units;
energy consumption: $T \cdot (P(x) \cdot a + P(y) \cdot (1-a))$

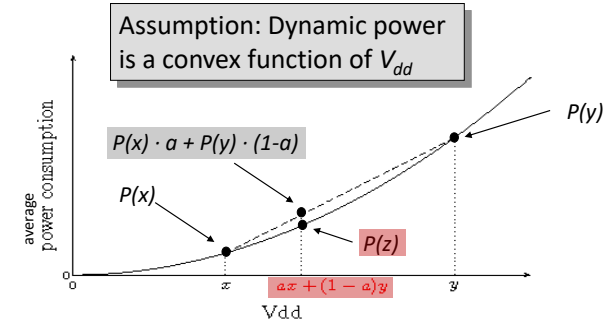
- **case B:** execute at voltage $z = a \cdot x + (1-a) \cdot y$ for T time units;
energy consumption: $T \cdot P(z)$

DVFS: Real-Time Offline Scheduling on One Processor (Optional for FS 2022)

- Let us model a set of independent tasks as follows:
 - We suppose that a task $v_i \in V$
 - requires c_i computation time at normalized processor frequency 1
 - arrives at time a_i
 - has (absolute) deadline constraint d_i
- How do we schedule these tasks such that all these tasks can be finished **no later than their deadlines** and the energy consumption is **minimized**?
 - YDS Algorithm from “A Scheduling Model for Reduce CPU Energy”, Frances Yao, Alan Demers, and Scott Shenker, FOCS 1995.”

If possible, running at a constant frequency (voltage) minimizes the energy consumption for dynamic voltage scaling.

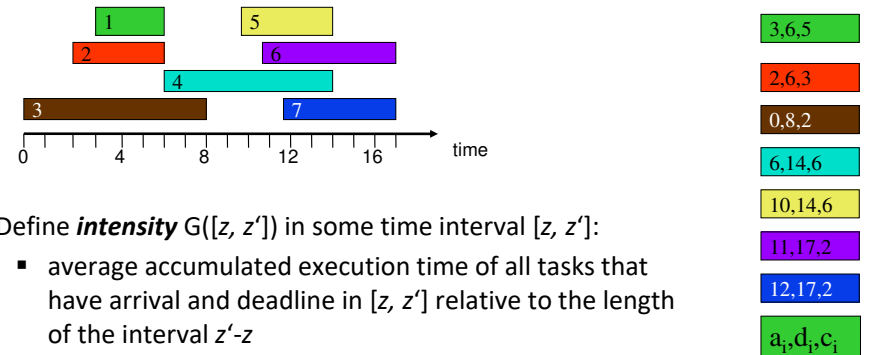
DVFS: Optimal Strategy



If possible, running at a constant frequency (voltage) minimizes the energy consumption for dynamic voltage scaling:

case A is always worse if the power consumption is a convex function of the supply voltage

YDS Optimal DVFS Algorithm for Offline Scheduling (Optional for FS 2022)



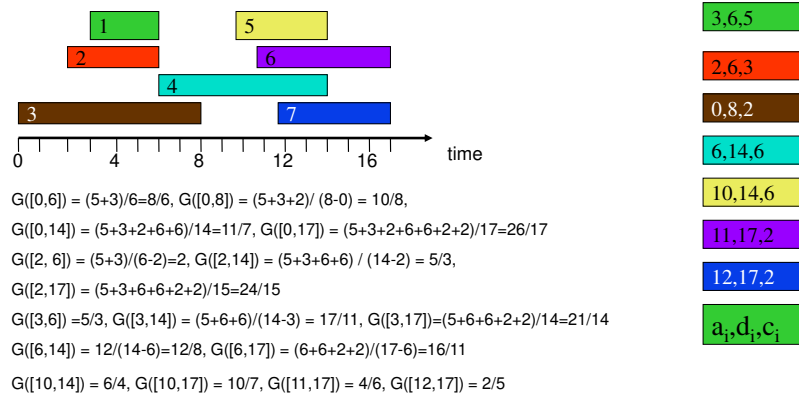
- Define **intensity** $G([z, z'])$ in some time interval $[z, z']$:
 - average accumulated execution time of all tasks that have arrival and deadline in $[z, z']$ relative to the length of the interval $z'-z$

$$V'([z, z']) = \{v_i \in V : z \leq a_i < d_i \leq z'\}$$

$$G([z, z']) = \sum_{v_i \in V'([z, z'])} c_i / (z' - z)$$

YDS Optimal DVFS Algorithm for Offline Scheduling (Optional for FS 2022)

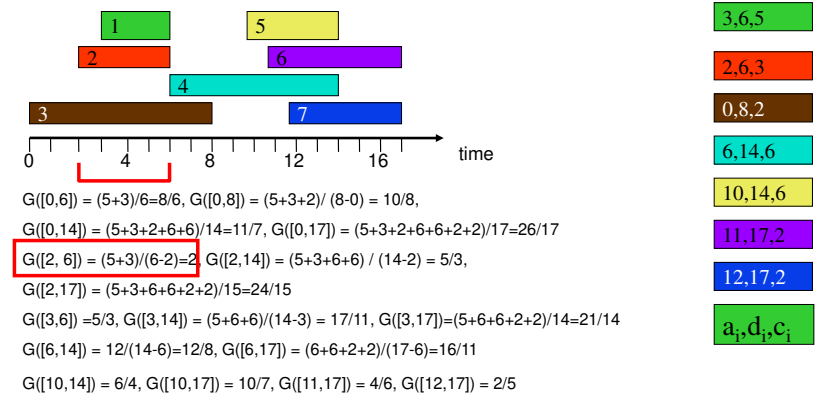
Step 1: Execute jobs in the interval with the highest intensity by using the earliest-deadline first schedule and running at the intensity as the frequency.



9 - 39

YDS Optimal DVFS Algorithm for Offline Scheduling (Optional for FS 2022)

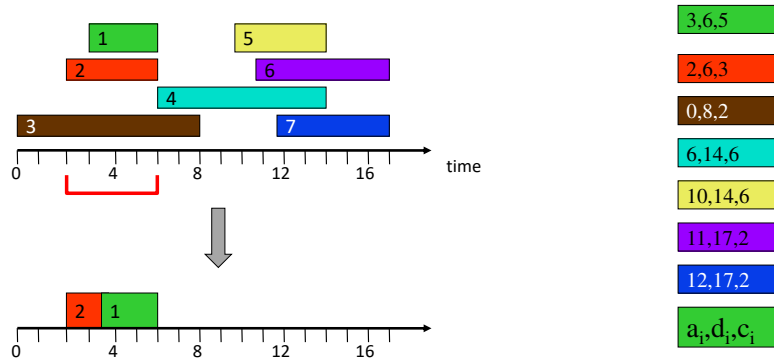
Step 1: Execute jobs in the interval with the highest intensity by using the earliest-deadline first schedule and running at the intensity as the frequency.



9 - 40

YDS Optimal DVFS Algorithm for Offline Scheduling (Optional for FS 2022)

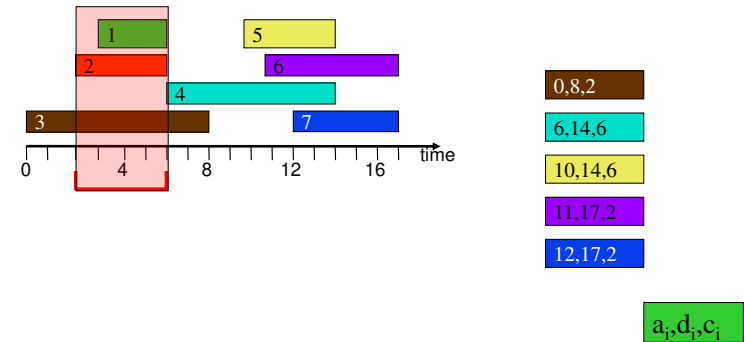
Step 1: Execute jobs in the interval with the highest intensity by using the earliest-deadline first schedule and running at the intensity as the frequency.



9 - 41

YDS Optimal DVFS Algorithm for Offline Scheduling (Optional for FS 2022)

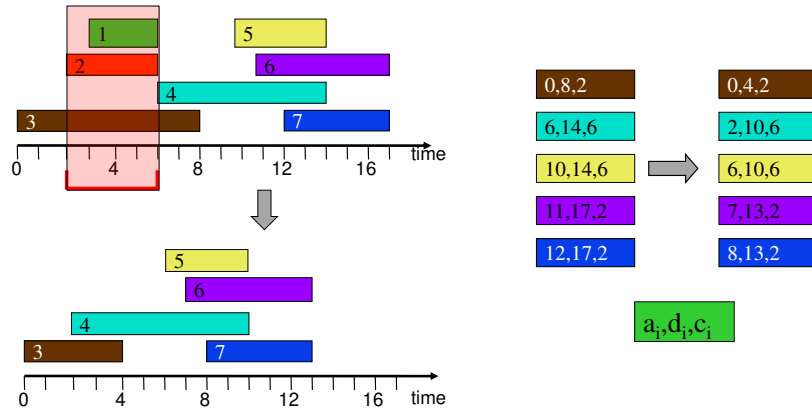
Step 2: Adjust the arrival times and deadlines by excluding the possibility to execute at the previous critical intervals.



9 - 42

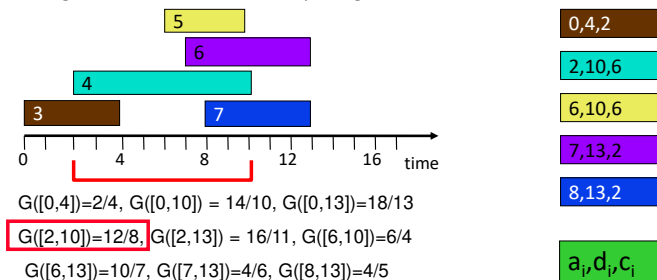
YDS Optimal DVFS Algorithm for Offline Scheduling

Step 2: Adjust the arrival times and deadlines by excluding the possibility to execute at the previous critical intervals.



YDS Optimal DVFS Algorithm for Offline Scheduling (Optional for FS 2022)

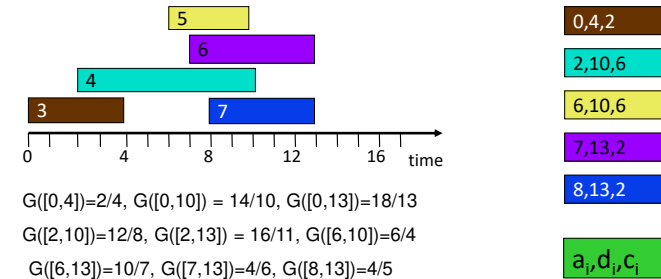
Step 3: Run the algorithm for the revised input again



9 - 43

YDS Optimal DVFS Algorithm for Offline Scheduling (Optional for FS 2022)

Step 3: Run the algorithm for the revised input again



$$G([0,4])=2/4, G([0,10]) = 14/10, G([0,13])=18/13$$

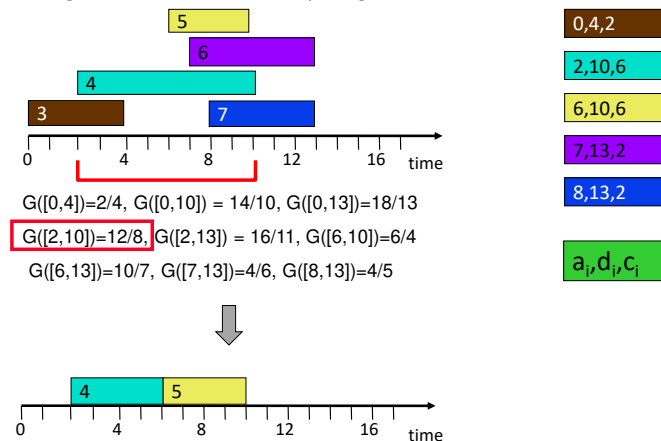
$$G([2,10])=12/8, G([2,13]) = 16/11, G([6,10])=6/4$$

$$G([6,13])=10/7, G([7,13])=4/6, G([8,13])=4/5$$

9 - 44

YDS Optimal DVFS Algorithm for Offline Scheduling

Step 3: Run the algorithm for the revised input again



$$G([0,4])=2/4, G([0,10]) = 14/10, G([0,13])=18/13$$

$$G([2,10])=12/8, G([2,13]) = 16/11, G([6,10])=6/4$$

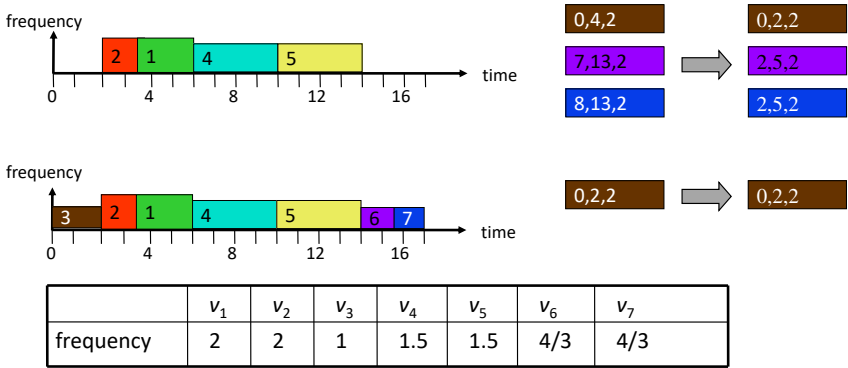
$$G([6,13])=10/7, G([7,13])=4/6, G([8,13])=4/5$$

9 - 45

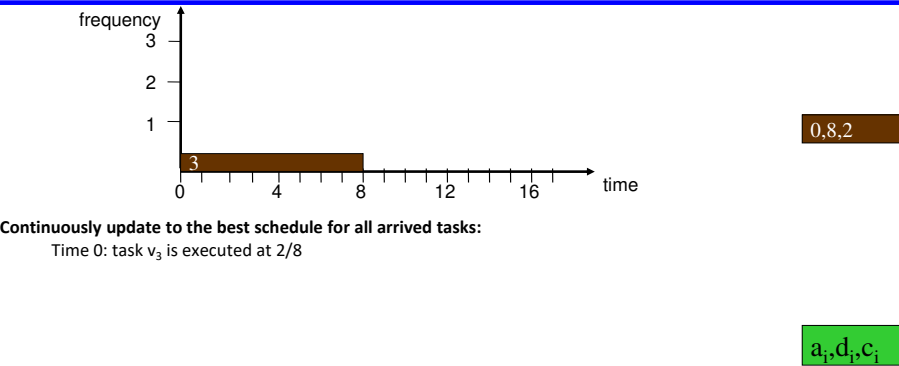
9 - 46

YDS Optimal DVFS Algorithm for Offline Scheduling (Optional for FS 2022)

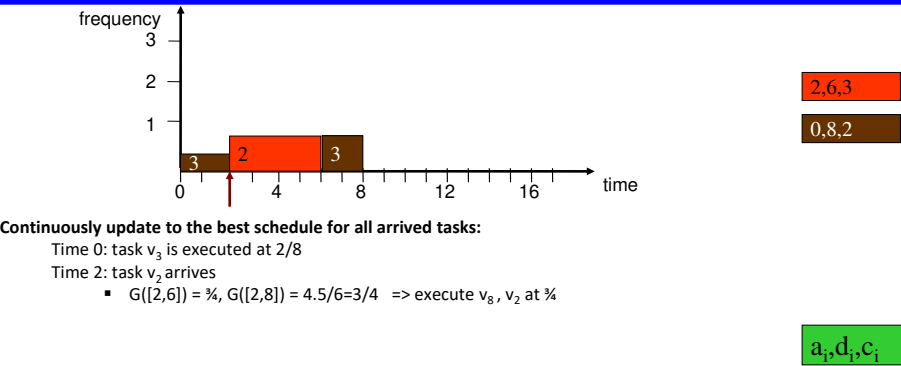
Step 3: Run the algorithm for the revised input again
 Step 4: Put pieces together



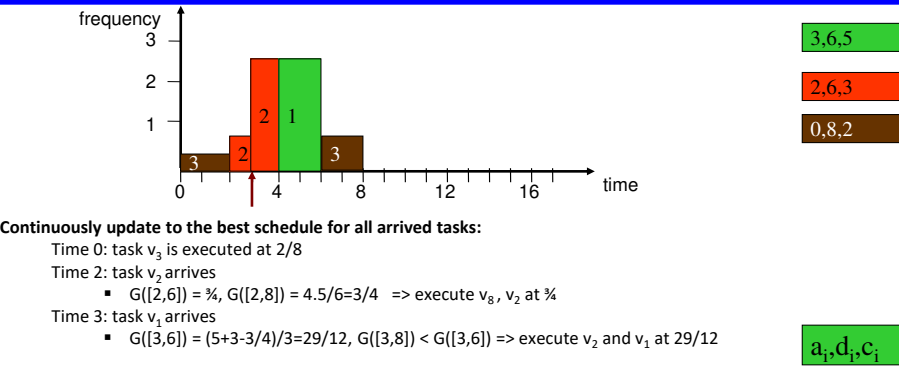
YDS Optimal DVFS Algorithm for Online Scheduling (Optional for FS 2022)



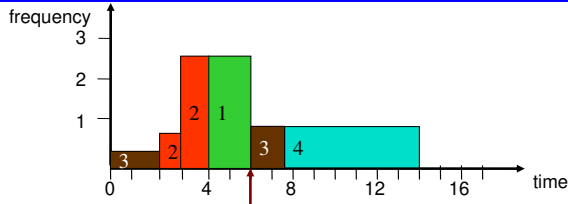
YDS Optimal DVFS Algorithm for Online Scheduling (Optional for FS 2022)



YDS Optimal DVFS Algorithm for Online Scheduling (Optional for FS 2022)



YDS Optimal DVFS Algorithm for Online Scheduling (Optional for FS 2022)



3,6,5
2,6,3
0,8,2
6,14,6

Continuously update to the best schedule for all arrived tasks:

Time 0: task v_3 is executed at 2/8

Time 2: task v_2 arrives

- $G([2,6]) = \frac{1}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v_8, v_2 at $\frac{1}{4}$

Time 3: task v_1 arrives

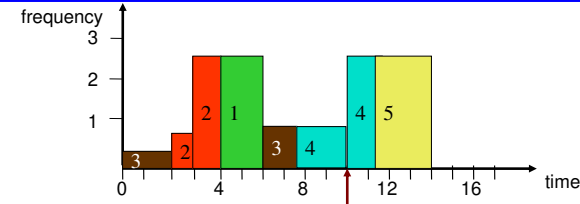
- $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v_2 and v_1 at 29/12

Time 6: task v_4 arrives

- $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v_3 and v_4 at 15/16

a_i, d_i, c_i

YDS Optimal DVFS Algorithm for Online Scheduling (Optional for FS 2022)



3,6,5
2,6,3
0,8,2
6,14,6
10,14,6

Continuously update to the best schedule for all arrived tasks:

Time 0: task v_3 is executed at 2/8

Time 2: task v_2 arrives

- $G([2,6]) = \frac{1}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v_8, v_2 at $\frac{1}{4}$

Time 3: task v_1 arrives

- $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v_2 and v_1 at 29/12

Time 6: task v_4 arrives

- $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v_3 and v_4 at 15/16

Time 10: task v_5 arrives

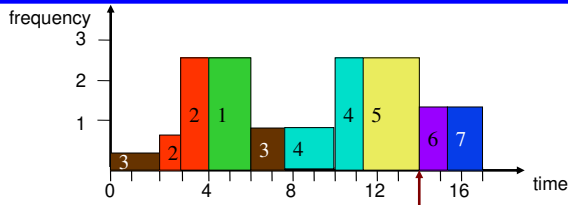
- $G([10,14]) = 39/16 \Rightarrow$ execute v_4 and v_5 at 39/16

a_i, d_i, c_i

9 - 51

9 - 52

YDS Optimal DVFS Algorithm for Online Scheduling



3,6,5
2,6,3
0,8,2
6,14,6
10,14,6
11,17,2
12,17,2
a_i, d_i, c_i

Continuously update to the best schedule for all arrived tasks:

Time 0: task v_3 is executed at 2/8

Time 2: task v_2 arrives

- $G([2,6]) = \frac{1}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v_8, v_2 at $\frac{1}{4}$

Time 3: task v_1 arrives

- $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v_2 and v_1 at 29/12

Time 6: task v_4 arrives

- $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v_3 and v_4 at 15/16

Time 10: task v_5 arrives

- $G([10,14]) = 39/16 \Rightarrow$ execute v_4 and v_5 at 39/16

Time 11 and Time 12

- The arrival of v_6 and v_7 does not change the critical interval

Time 14:

- $G([14,17]) = 4/3 \Rightarrow$ execute v_6 and v_7 at 4/3

Remarks on the YDS Algorithm (Optional for FS 2022)

Offline

- The algorithm guarantees the minimal energy consumption while satisfying the timing constraints
- The time complexity is $O(N^3)$, where N is the number of tasks in V
 - Finding the critical interval can be done in $O(N^2)$
 - The number of iterations is at most N
- Exercise:
 - For periodic real-time tasks with deadline=period, running at **constant speed with 100% utilization** under EDF has minimum energy consumption while satisfying the timing constraints.

Online

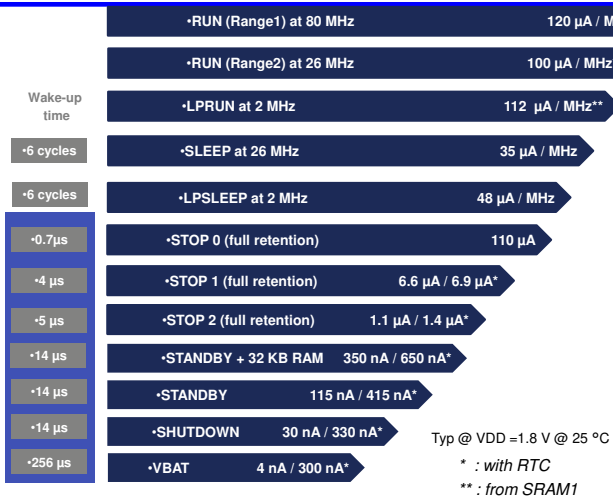
- Compared to the optimal offline solution, the on-line schedule uses at most 27 times of the minimal energy consumption.

9 - 53

9 - 54

Dynamic Power Management

Power Modes STM32L4 (similar to the lab board)



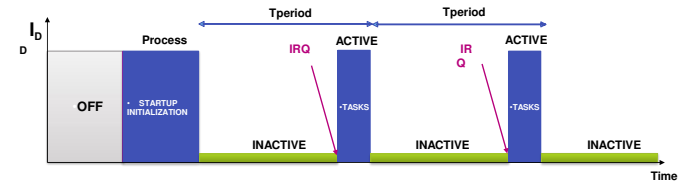
FlexPowerControl

- Efficient running
- 8 low-power modes, several sub-modes
- High flexibility

Application benefits

- High performance
 - CoreMark score = 273
- Outstanding power efficiency
 - ULPBench score = 150

Typical Application Profile with MCU



Application phases:

- OFF – power is not applied to MCU
- STARTUP INITIALIZATION – MCU performs configuration (peripherals, clocks, ...)

$$\text{Duty Cycle [\%]} = \frac{T_{\text{active}}}{T_{\text{period}}} * 100$$

T_{period}

- INACTIVE – MCU is in low power mode to reduce power consumption
- ACTIVE – MCU is in normal mode and performs tasks

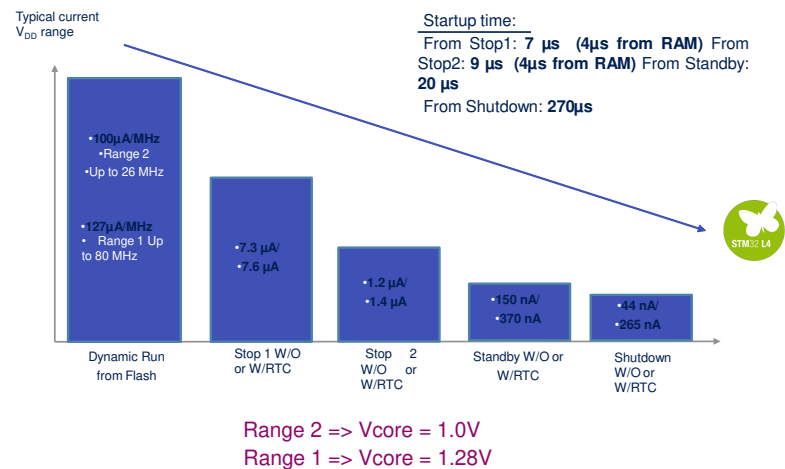
Key features of STM32L4 (similar to the lab board)

- 7 low-power modes with fast wakeup
 - Down to 30 nA with I/O wake-up
 - Down to 350 nA with 32 KB RAM retained
 - Wake-up from high number of peripherals
- Down to 100 µA / MHz in Run mode
- Battery backup mode with RTC and backup registers
- Independent power supplies

Application benefits

- High flexibility to lower power consumption depending on active peripherals, required performance and needed wakeup sources
- Increase battery life
- BOM cost saving by removing external shifters

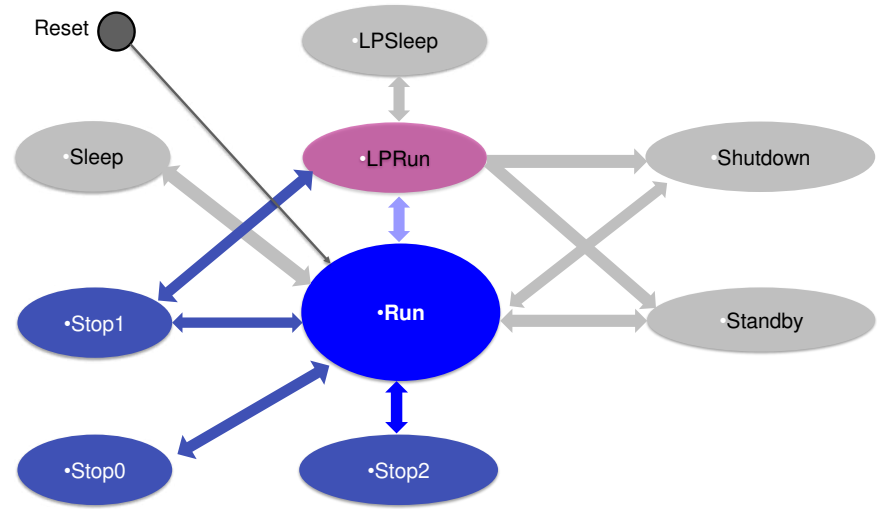
STM32L4 Power Consumption



STM32L4 Power Modes Summary

Mode	Regulator	CPU	Flash	SRAM	Clocks	Peripherals	Consumption @ 1.8V	Wakeup time
Run	R1	Yes	ON ⁽¹⁾	ON	Any	All	127 µA/MHz	N/A
	R2					All except OTG, RNG	111 µA/MHz	
LPRun	LPR	Yes	ON ⁽¹⁾	ON	Any except PLL	All except OTG, RNG	136 µA/MHz	
Sleep	R1	No	ON ⁽¹⁾	ON ⁽²⁾	Any	All	37 µA/MHz	6 cycles
	R2					Any IT or event	35 µA/MHz	
LPSleep	LPR	No	ON ⁽¹⁾	ON ⁽²⁾	Any except PLL	All except OTG, RNG Any IT or event	40 µA/MHz	6 cycles
Stop 0	MR	No	OFF	ON	LSE/LSI	Reset pin, all I/Os BOR,PVD,PVM,RTC,LCD,IWDG, COMPx,DACx,OPAMPx,USARTx, LPUART,I2Cx,LPTIMx,OTG_FS,SWPMI	110µA	0.7 µA RAM 5µA Flash memory
Stop 1	LPR						6.6 µA w/o RTC 6.9 µA w/RTC	4 µA RAM 6 µA Flash memory
Stop 2	LPR	No	OFF	ON	LSE/LSI	Reset pin, all I/Os BOR,PVD,PVM,RTC,LCD,IWDG, COMPx,LPUART,I2C3,LPTIM1	1.1 µA w/o RTC 1.4 µA w/RTC	5 µA RAM 7 µA Flash memory
Standby	LPR	DOWN	OFF	SRAM2 ON	LSE/LSI	Reset pin, 5 WKUPx pins BOR, RTC, IWDG	+ 235 nA	14 µs
	OFF			DOWN			115 nA w/o RTC 415 nA w/RTC	
Shutdown	OFF	DOWN	OFF	DOWN	LSE	Reset pin, 5 WKUPx pins RTC	30 nA w/o RTC 330 nA w/RTC	250 µs

Power Modes Transitions

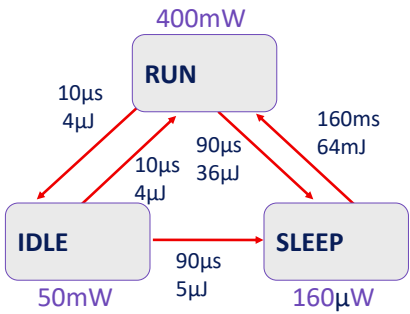


Dynamic Power Management (DPM)

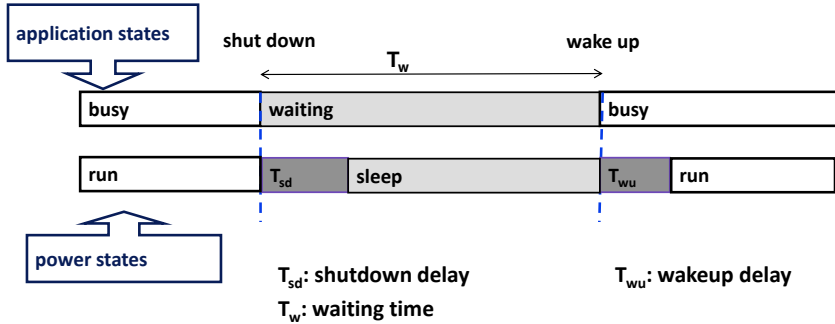
- Dynamic power management tries to assign optimal **power saving states during program execution**
- DPM requires hardware and software support

Example: StrongARM SA1100

RUN: operational
IDLE: a SW routine may stop the CPU when not in use, while monitoring interrupts
SLEEP: Shutdown of on-chip activity



Dynamic Power Management (DPM)

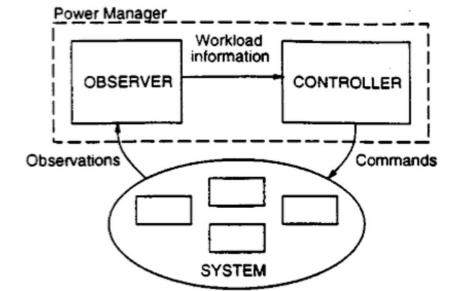


Desired: Shutdown only during long waiting times. This leads to a tradeoff between energy saving and overhead.

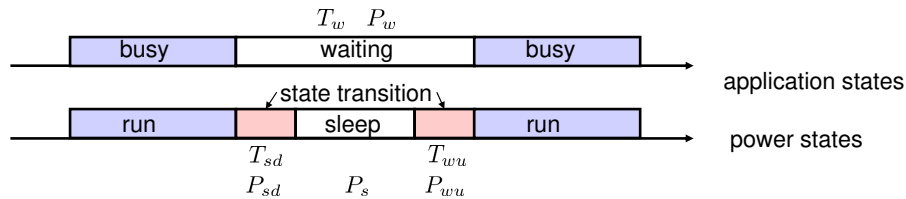
Break-Even Time

Definition: The minimum waiting time required to compensate the cost of entering an inactive (sleep) state.

- Enter an inactive state is beneficial only if the waiting time is longer than the break-even time
- Assumptions for the calculation:
 - No performance penalty is tolerated.
 - An ideal power manager that has the *full* knowledge of the future workload trace. On the previous slide, we supposed that the power manager has *no* knowledge about the future.



Break-Even Time



Scenario 1 (no transition): $E_1 = T_w \cdot P_w$

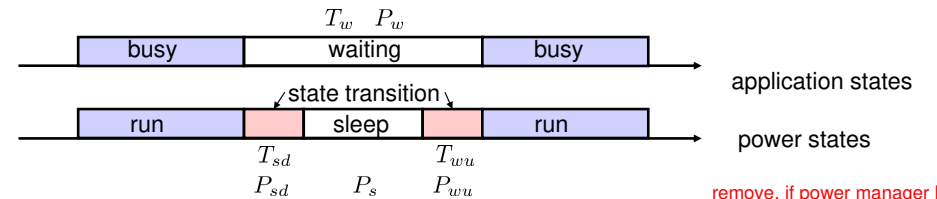
Scenario 2 (state transition): $E_2 = T_{sd} \cdot P_{sd} + T_{wu} \cdot P_{wu} + (T_w - T_{sd} - T_{wu}) \cdot P_s$

Break-even time: Limit for T_w such that $E_2 \leq E_1$

Break-even constraint: $T_w \geq \frac{T_{sd} \cdot (P_{sd} - P_s) + T_{wu} \cdot (P_{wu} - P_s)}{P_w - P_s}$ ← break-even time

Time constraint: $T_w \geq T_{sd} + T_{wu}$

Break-Even Time



Scenario 1 (no transition): $E_1 = T_w \cdot P_w$

Scenario 2 (state transition): $E_2 = T_{sd} \cdot P_{sd} + T_{wu} \cdot P_{wu} + (T_w - T_{sd} - T_{wu}) \cdot P_s$

Break-even time: Limit for T_w such that $E_2 \leq E_1$

Break-even constraint: $T_w \geq \frac{T_{sd} \cdot (P_{sd} - P_s) + T_{wu} \cdot (P_{wu} - P_s)}{P_w - P_s}$ ← break-even time

Time constraint: $T_w \geq T_{sd} + T_{wu}$

remove, if power manager has no knowledge about future

Battery-Operated Systems and Energy Harvesting

9 - 67

Smart Sensors and IoT are getting Momentum

Recent Trends:

- Greater HW/SW complexity
- More functionality per area
- Shorter lifetime

Biggest Challenges:

- Energy storage/consumption/sources



9 - 68

Smart Sensors and IoT are getting Momentum

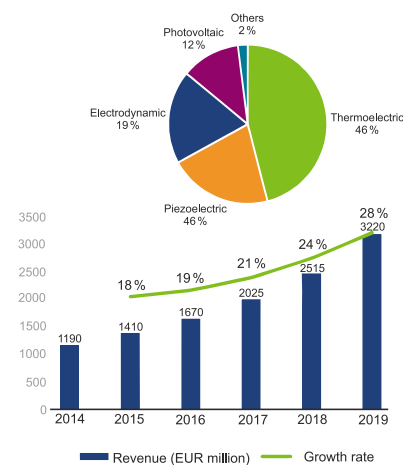
Recent Trends:

- Greater HW/SW complexity
- More functionality per area
- Shorter lifetime

Biggest Challenges:

- Energy storage/consumption/sources

Energy Harvesting as possible solution?



[1] L. Probst, B. Pedersen and L. Dalkak-Armoux, "Energy harvesting to power the rise of the Internet of Things in Digital Transformation Monitoring", Jul. 2017

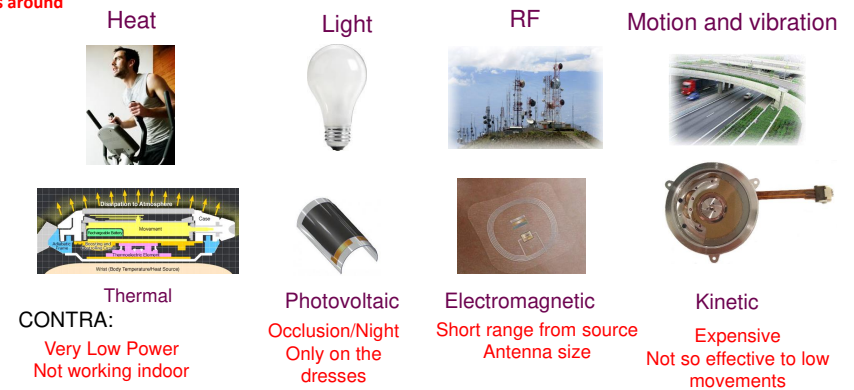
9 - 69

Energy Harvesting

Energy harvesting is the process by which energy is captured and stored.

This term often refers to small autonomous devices – micro energy harvesting

All Energy is around

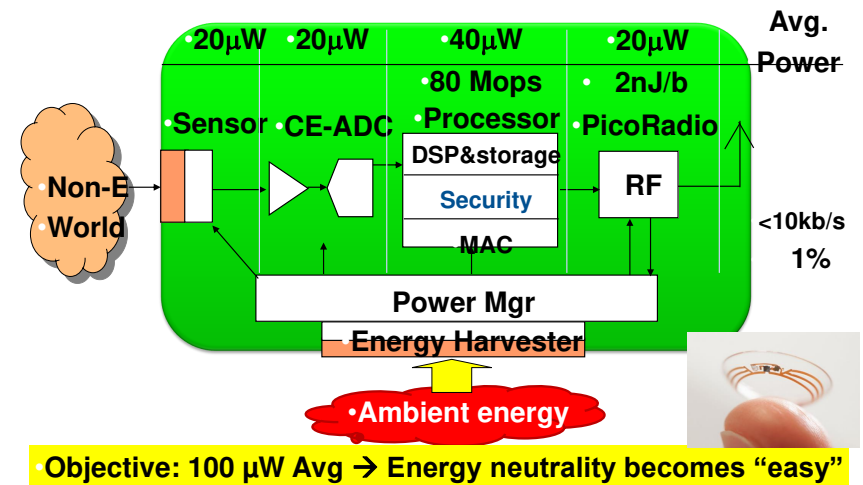


9 - 70

Which source from Embedded Systems?

Energy Source	Source Polarity	Efficiency	Harvested Power	Characteristics
Light	DC	10-24%	100 mW/cm ² (Outdoor)	Operating conditions vary widely with environment light level. MPPT algorithms needed to achieve maximum power transfer
			100 μ W/cm ² (illuminated office)	
Thermal	DC	~0.1%	60 μ W/cm ² (Human)	Low output voltage. Step-up circuit needed. Impedance matching to achieve maximum power transfer
		~3%	~1-10 mW/cm ² (Industrial)	
Vibration	AC	25-50%	~4 μ W/cm ³ (Human motion - Hz)	High AC output voltage with positive and negative fluctuations (spikes). Rectifier & Step-down circuits are needed.
			~800 μ W/cm ³ (Machines - KHz)	
Ambient Air flow	AC	~39% (Dynamic)	35 μ W/cm ² (@ <1 m/s)	Dual or 3-phase output. Rectifier is needed. MPP varies slightly with wind speed. Impedance matching is sufficient to achieve maximum power transfer in many applications
		~41% (Generator)	3.5 mW/cm ² (@ 8.4 m/s)	
RF	AC	~50%	0.1 μ W/cm ² (GSM 900 MHz)	Impedance matching to achieve maximum power transfer
			0.001 mW/cm ² (WiFi)	

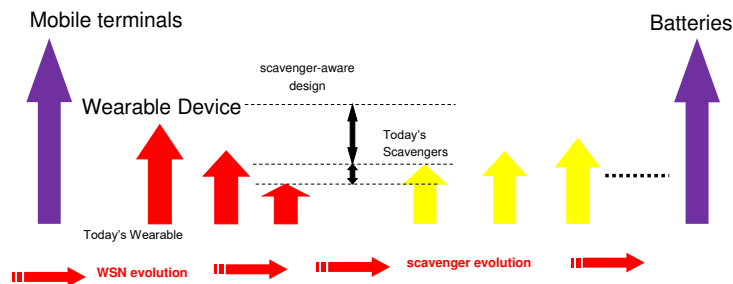
A dream? Achieving Autonomous “IoT”



9 - 71

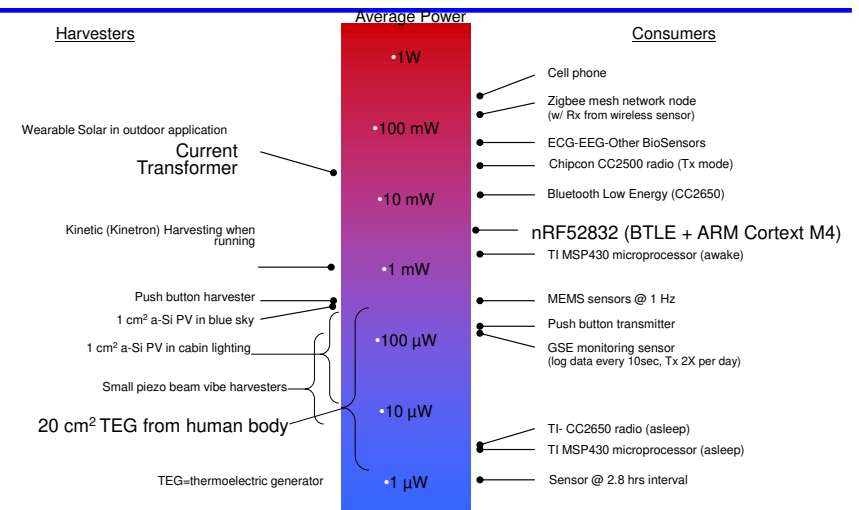
9 - 72

The Good News



- The gap between scavengers energy and requirements of digital systems is shrinking
- Exploit energy management strategies and improvements in scavenger technology
 - Overcome traditional energy management strategies (battery-driven)
- An new **unified design methodology** is required
 - Smart adaptation
 - Design for unreliability
 - Exploit unpredictable power sources

System Design Consideration



9 - 73

Next Generation of IoT devices: **Always-on Embedded System**

1.) Edge Signal Processing and AI

3.) Low power system design

2.) Energy harvesting

4.) Low Power and long-range communication

Smart devices
for perpetual operation



Embedded Systems in the Extreme - Permasense



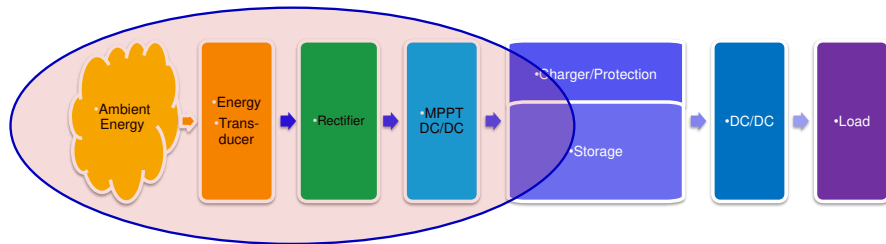
Energy Harvesting Design: Generic Approach

Dedicated blocks, depending on energy source, ambient conditions and application

Not all are required in any application and with any source

Rectifier, DC-DC converter and MPPT are the most **challenging** and require a very **accurate design process**

Charger/limiter/protection consumes additional power and are often to some extent redundant.



D-ITET center for Project-based Learning

79 - 79

Solar Panel Characteristics

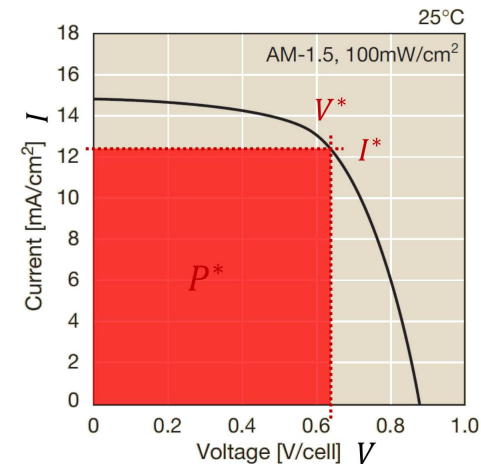


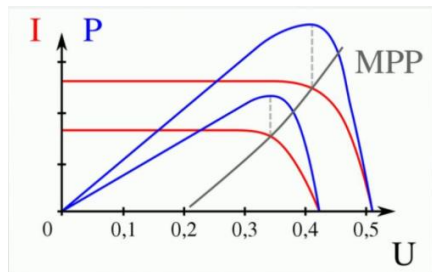
Diagram: Amorton Amorphous Silicon Solar Cells Datasheet, © Panasonic

- Variable output power
 - Illuminance level
 - Electrical operation point
 - (Temperature, age, ...)
- I-V-Characteristics
 - Non-linear
 - Dependent on ambient
- Maximum Power Point Tracking
 - Dynamic algorithm to find P^*

9 - 80

Typical Power Circuitry – Maximum Power Point Tracking

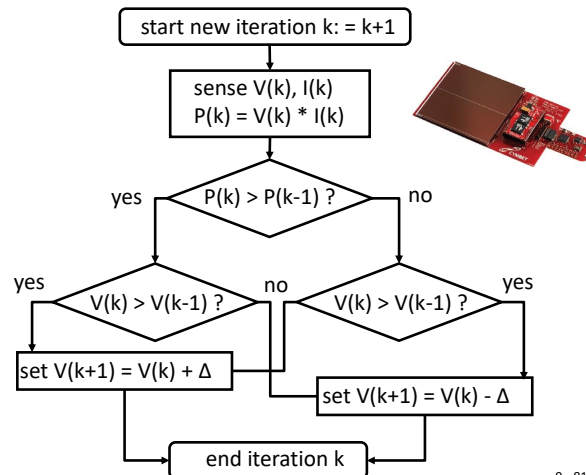
U/I curves of a typical solar cell:



red: current for different light intensities
blue: power for different light intensities
grey: maximal power

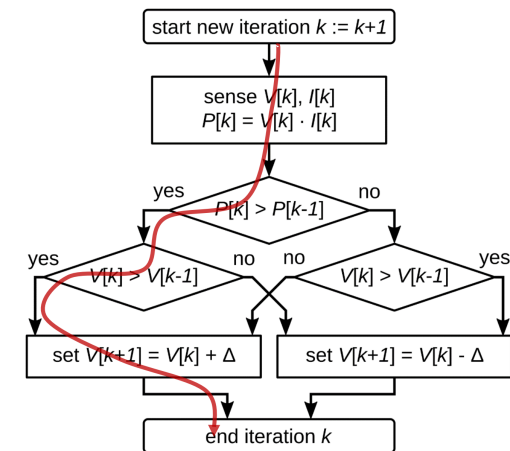
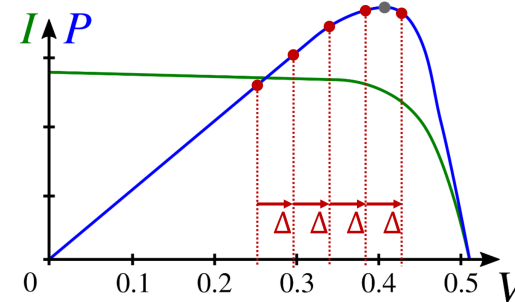
tracking: determine optimal impedance seen by the solar panel

simple tracking algorithm (assume constant illumination) :



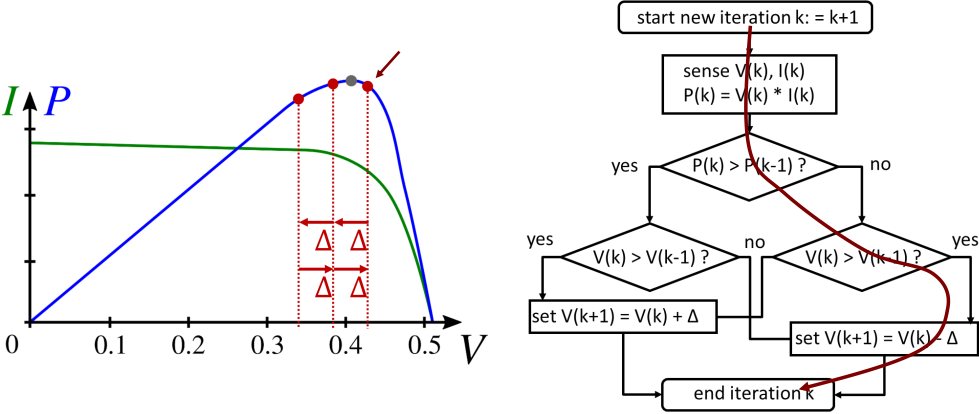
9 - 81

Maximal Power Point Tracking



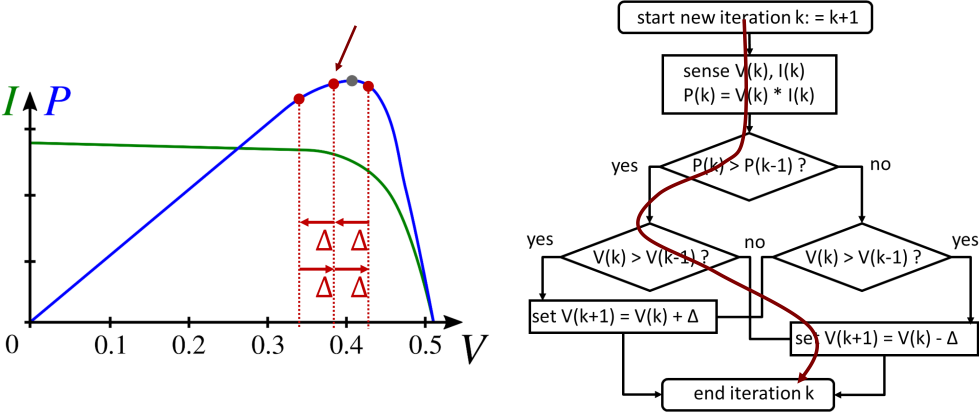
9 - 82

Maximal Power Point Tracking



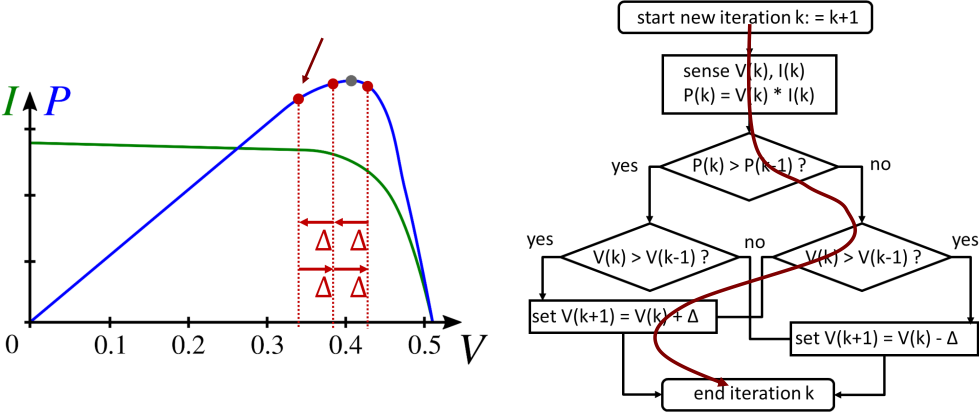
9 - 83

Maximal Power Point Tracking



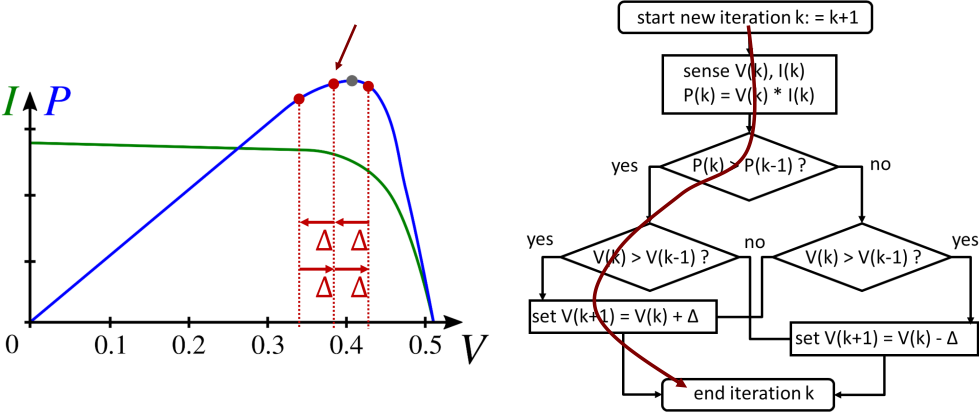
9 - 84

Maximal Power Point Tracking



9 - 85

Maximal Power Point Tracking



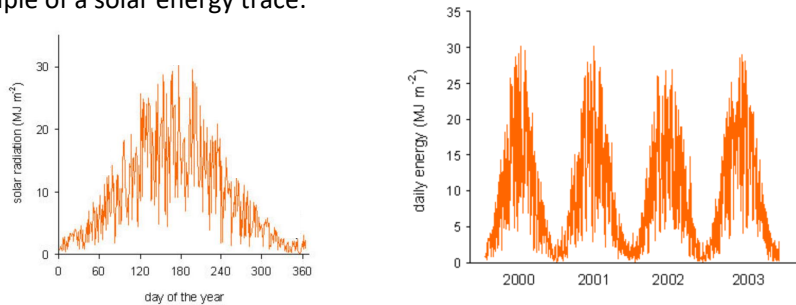
9 - 86

Typical Challenge in (Solar) Harvesting Systems

Challenges:

- What is the optimal maximum capacity of the battery?
- What is the optimal area of the solar cell?
- How can we control the application such that a continuous system operation is possible, even under a varying input energy (summer, winter, clouds)?

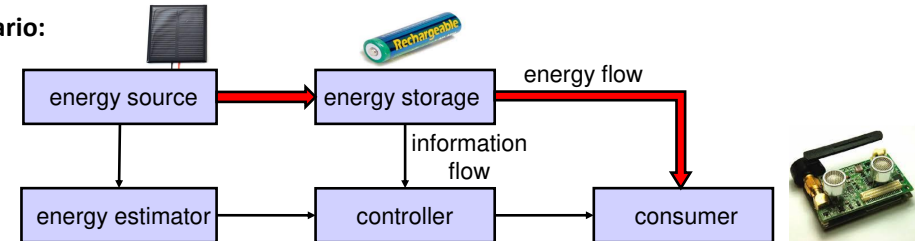
Example of a solar energy trace:



9 - 87

Example: Application Control

Scenario:

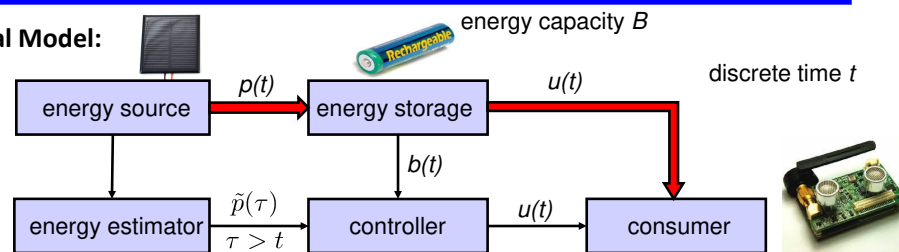


- The controller can adapt the service of the consumer device, for example the sampling rate for its sensors or the transmission rate of information. As a result, the power consumption changes proportionally.
- Precondition for correctness** of application control: Never run out of energy.
- Example for optimality criterion:** Maximize the lowest service of (or equivalently, the lowest energy flow to) the consumer.

9 - 88

Application Control

Formal Model:



- harvested and used energy in $[t, t+1]$: $p(t), u(t)$
- battery model: $b(t+1) = \min\{b(t) + p(t) - u(t), B\}$
- failure state: $b(t) + p(t) - u(t) < 0$
- utility:

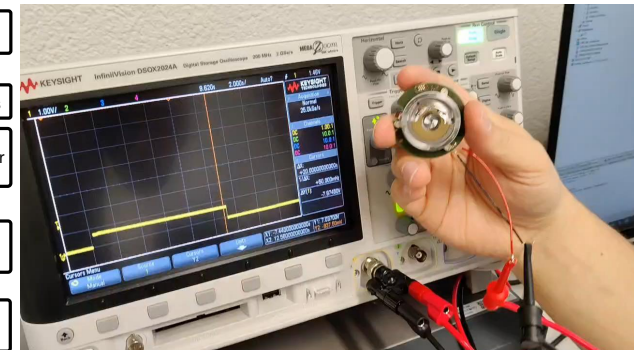
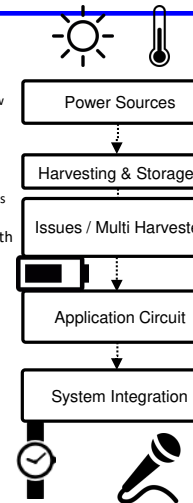
$$U(t_1, t_2) = \sum_{t_1 \leq \tau < t_2} \mu(u(\tau))$$

μ is a strictly concave function;
higher used energy gives a reduced
reward for the overall utility.

9 - 89

Always have in mind the full picture!

- Electrical parameters of the transducers
 - Voltage, Current, Power, How will be converted
- Harvesting Circuits
 - Maximize energy conversion but consider your input
 - Adaptive and multi harvesters is a must
- Design your system according with the energy budget
 - Minimize power, maximize your performance
- Make it smart in power management
 - Non Volatile Processing
 - On-board processing
 - Edge Processing and TinyML



9 - 90

Application Control (Optional in FS2023)

- **What do we want?** We would like to determine an optimal control $u^*(t)$ for time interval $[t, t+1)$ for all t in $[0, T)$ with the following properties:
 - $\forall 0 \leq t < T : b^*(t) + p(t) - u^*(t) \geq 0$
 - There is no feasible use function $u(t)$ with a larger minimal energy:

$$\forall u : \min_{0 \leq t < T} \{u(t)\} \leq \min_{0 \leq t < T} \{u^*(t)\}$$
 - The use function maximizes the utility $U(0, T)$.
 - We suppose that the battery has the same or better state at the end than at the start of the time interval, i.e., $b^*(T) \geq b^*(0)$.
- We would like to answer two questions:
 - Can we say something about the characteristics of $u^*(t)$?
 - How does an algorithm look like that efficiently computes $u^*(t)$?

9 - 91

9 - 92

Application Control (Optional in FS2023)

Theorem: Given a use function $u^*(t)$, $t \in [0, T)$ such that the system never enters a failure state. If $u^*(t)$ is optimal with respect to maximizing the minimal used energy among all use functions and maximizes the utility $U(t, T)$, then the following relations hold for all $\tau \in (0, T)$:

$$u^*(\tau - 1) < u^*(\tau) \implies b^*(\tau) = 0 \quad \leftarrow \text{empty battery}$$

$$u^*(\tau - 1) > u^*(\tau) \implies b^*(\tau) = B \quad \leftarrow \text{full battery}$$

Sketch of a proof: First, let us show that a consequence of the above theorem is true (just reverting the relations):

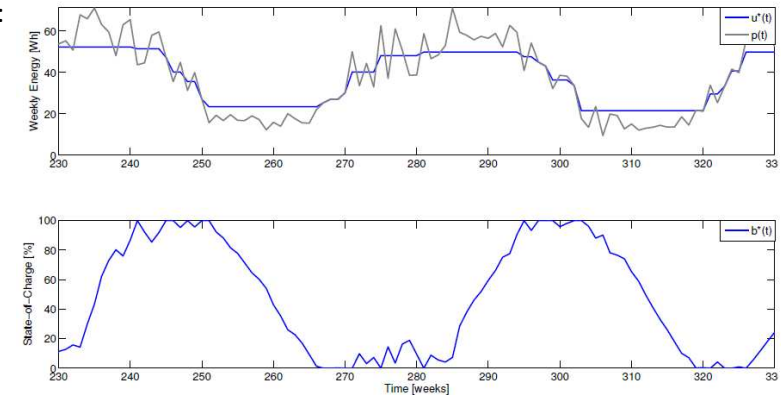
$$\forall \tau \in (s, t] : 0 < b^*(\tau) < B \implies \forall \tau \in [s, t] : u^*(\tau) = u^*(t)$$

In other words, as long as the battery is neither full nor empty, the optimal use function does not change.

9 - 93

Application Control (Optional in FS2023)

- Proof sketch cont.:



(top) Example of an optimal use function $u^*(t)$ for a given harvest function $p(t)$ and (bottom) the corresponding stored energy $b^*(t)$.

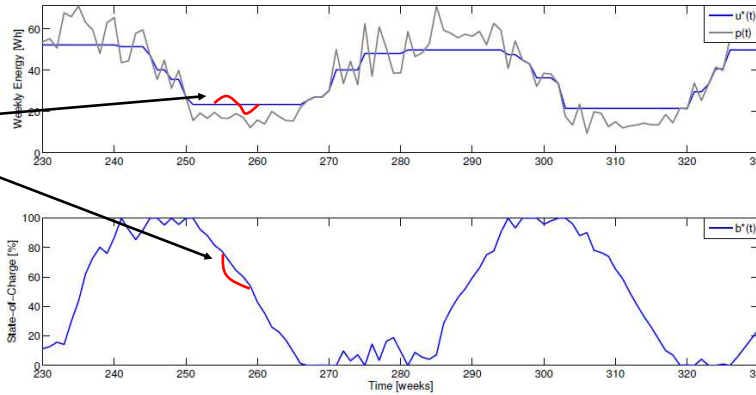
9 - 94

Application Control

Proof sketch cont.:

suppose we change the use function locally from being constant such that the overall battery state does not change

then the utility is worse due to the concave function μ : diminishing reward for higher use function values; and the minimal use function is potentially smaller



(top) Example of an optimal use function $u^*(t)$ for a given harvest function $p(t)$ and (bottom) the corresponding stored energy $b^*(t)$.

9 - 95

Application Control

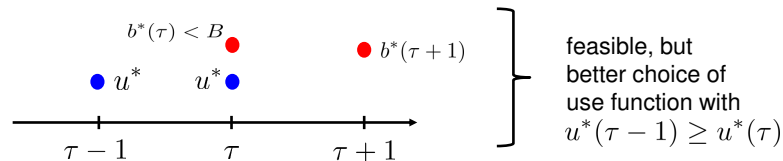
Proof sketch cont.: Now we show that for all $\tau \in (t, T)$

$$u^*(\tau - 1) < u^*(\tau) \implies b^*(\tau) = 0$$

or equivalently

$$b^*(\tau) > 0 \implies u^*(\tau - 1) \geq u^*(\tau)$$

We already have shown this for $0 < b^*(\tau) < B$. Therefore, we only need to show that $b^*(\tau) = B \implies u^*(\tau - 1) \geq u^*(\tau)$. Suppose now that we have $u^*(\tau - 1) < u^*(\tau)$ if the battery is full at τ . Then we can increase the use at time $\tau - 1$ and decrease it at time τ by the same amount without changing the battery level at time $\tau + 1$. This again would increase the overall utility and potentially increase the minimal use function.



9 - 97

Application Control (Optional in FS2023)

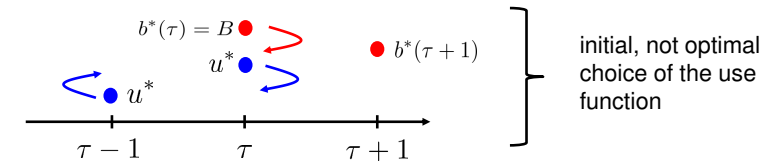
Proof sketch cont.: Now we show that for all $\tau \in (t, T)$

$$u^*(\tau - 1) < u^*(\tau) \implies b^*(\tau) = 0$$

or equivalently

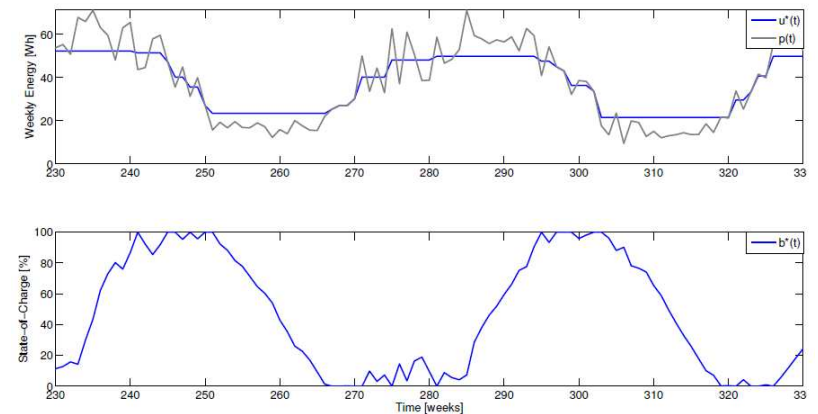
$$b^*(\tau) > 0 \implies u^*(\tau - 1) \geq u^*(\tau)$$

We already have shown this for $0 < b^*(\tau) < B$. Therefore, we only need to show that $b^*(\tau) = B \implies u^*(\tau - 1) \geq u^*(\tau)$. Suppose now that we have $u^*(\tau - 1) < u^*(\tau)$ if the battery is full at τ . Then we can increase the use at time $\tau - 1$ and decrease it at time τ by the same amount without changing the battery level at time $\tau + 1$. This again would increase the overall utility and potentially increase the minimal use function.



9 - 96

Application Control (Optional in FS2023)



(top) Example of an optimal use function $u^*(t)$ for a given harvest function $p(t)$ and (bottom) the corresponding stored energy $b^*(t)$.

9 - 98

Application Control (Optional in FS2023)

- How can we efficiently compute an optimal use function?
 - There are several options available as we just need to solve a convex optimization problem.
 - A simple but inefficient possibility is to convert the problem into a linear program. At first suppose that the utility is simply

$$U(0, T) = \sum_{0 \leq \tau < T} u(\tau)$$

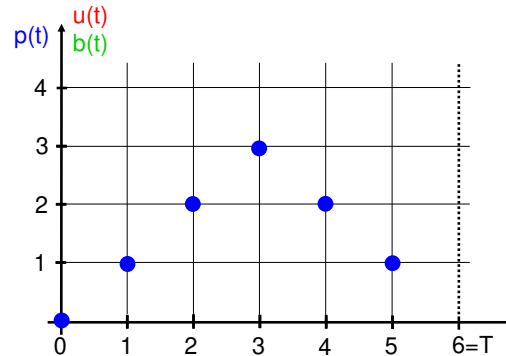
Then the linear program has the form:

[Concave functions μ could be piecewise linearly approximated. This is not shown here.]

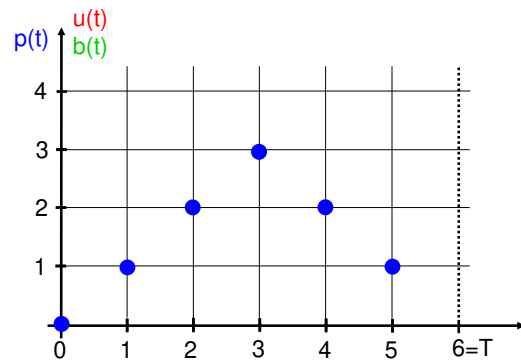
$$\begin{aligned} & \text{maximize} \quad \sum_{0 \leq \tau < T} u(\tau) \\ & \forall \tau \in [0, T) : b(\tau + 1) = b(\tau) - u(\tau) + p(\tau) \\ & \forall \tau \in [0, T) : 0 \leq b(\tau + 1) \leq B \\ & \forall \tau \in [0, T) : u(\tau) \geq 0 \\ & b(T) = b(0) = b_0 \end{aligned}$$

9 - 99

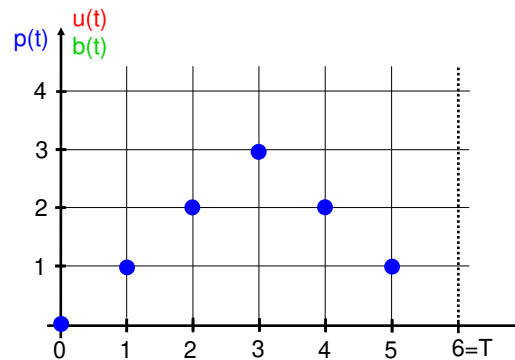
(Optional in FS2023)



9 - 100



(Optional in FS2023)



9 - 101

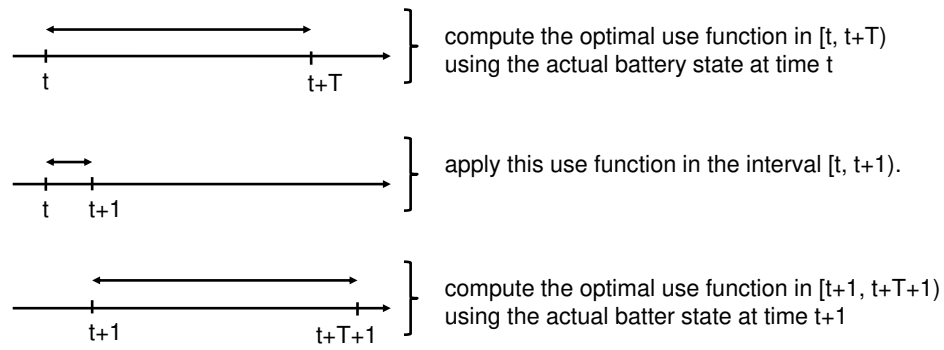
Application Control (Optional in FS2023)

- But what happens if the estimation of the future incoming energy is not correct?
 - If it would be correct, then we would just compute the whole future application control now and would not change anything anymore.
 - This will not work as errors will accumulate and we will end up with many infeasible situations, i.e., the battery is completely empty and we are forced to stop the application.
 - Possibility:** Finite horizon control
 - At time t , we compute the optimal control (see previous slides) using the currently available battery state $b(t)$ with predictions $\tilde{p}(\tau)$ for all $t \leq \tau < t + T$ and $b(t + T) = b(t)$.
 - From the computed optimal use function $u(\tau)$ for all $t \leq \tau < t + T$ we just take the first use value $u(t)$ in order to control the application.
 - At the next time step, we take as initial battery state the actual state; therefore, we take mispredictions into account. For the estimated future energy, we also take the new estimations.

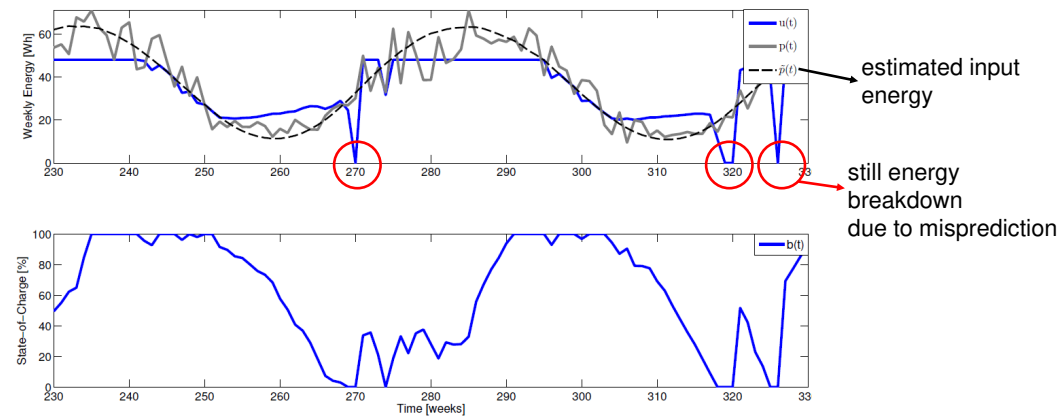
9 - 102

Application Control (Optional in FS2023)

■ Finite horizon control:



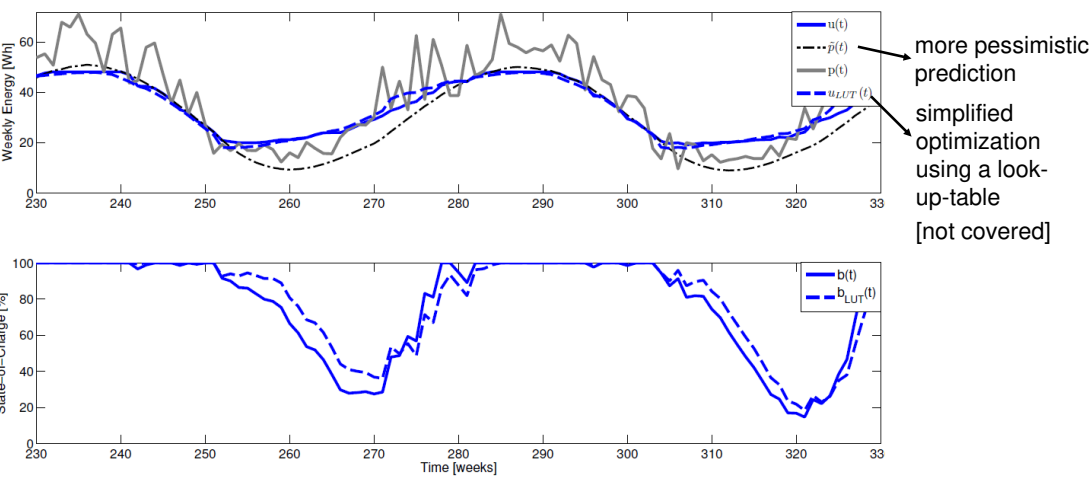
Application Control using Finite Horizon (Optional in FS2023)



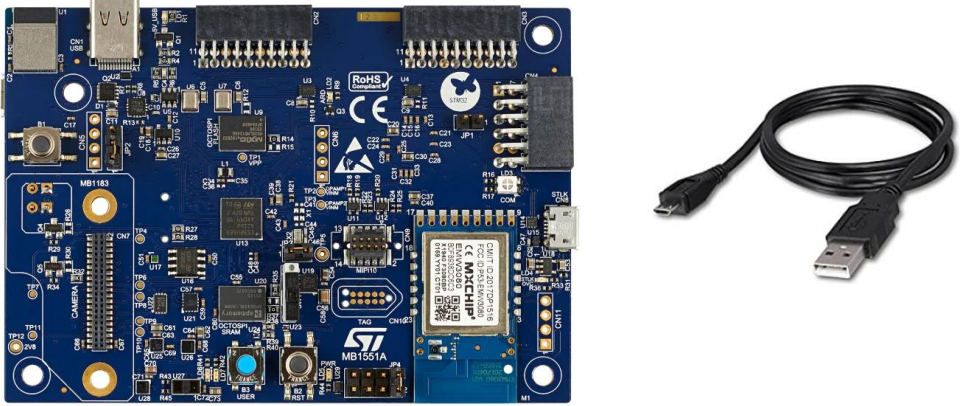
9 - 103

9 - 104

Application Control using Finite Horizon (Optional in FS2023)



Remember: What you got some time ago ...



9 - 105

10 - 1

What we told you: Be careful and please do not ...



**Return the boards at the
embedded systems exam!**

10 - 2

10 - 3

Embedded Systems

Lecture 12. Architecture Synthesis

© Lothar Thiele

Computer Engineering and Networks Laboratory

Michele Magno

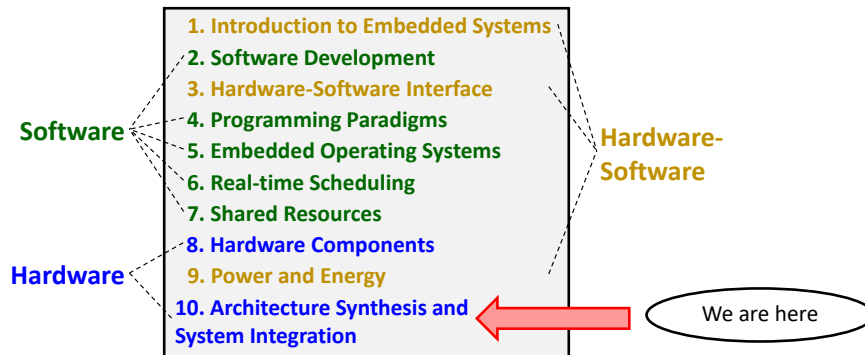
D-ITET center for project based learning

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

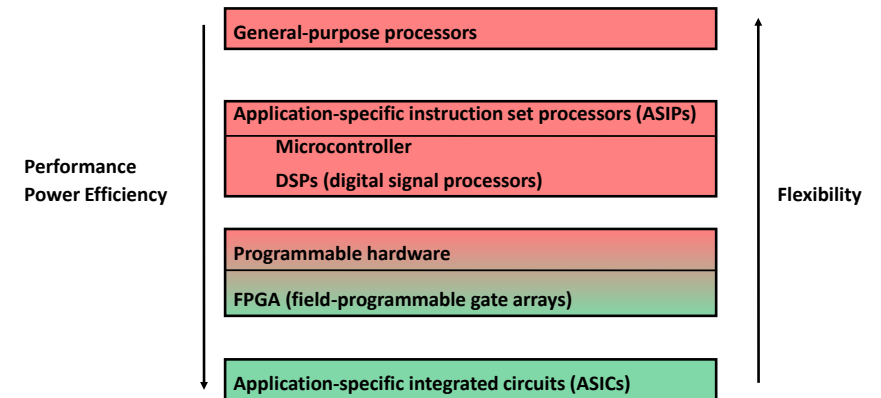
10 - 5

Where we are ...



10 - 6

Implementation Alternatives



10 - 7

Architecture Synthesis

Determine a hardware architecture that efficiently executes a given algorithm.

- Major tasks of architecture synthesis:
 - allocation* (determine the necessary hardware resources)
 - scheduling* (determine the timing of individual operations)
 - binding* (determine relation between individual operations of the algorithm and hardware resources)
- Classification of synthesis algorithms:
 - heuristics or exact methods
- Synthesis methods can often be applied independently of granularity of algorithms, e.g. whether operation is a whole complex task or a single operation.

10 - 8

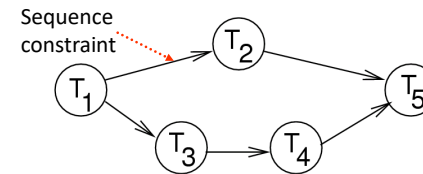
Specification Models

10 - 9

Specification

- *Formal specification* of the desired *functionality and the structure* (architecture) of an embedded systems is a necessary step for using computer aided design methods.
- There exist *many different formalisms* and models of computation, see also the models used for real-time software and general specification models for the whole system.
- Now, we will introduce some relevant models for architecture level (hardware) synthesis.

Task Graph or Dependence Graph (DG)



Nodes are assumed to be a „program“ described in some programming language, e.g. C or Java; or just a single operation.

A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a partial order.

If $(v1, v2) \in E$, then $v1$ is called an **immediate predecessor** of $v2$ and $v2$ is called an **immediate successor** of $v1$.

Suppose E^* is the transitive closure of E . If $(v1, v2) \in E^*$, then $v1$ is called a **predecessor** of $v2$ and $v2$ is called a **successor** of $v1$.

10 - 10

10 - 11

Dependence Graph

- A *dependence graph* describes order relations for the execution of single operations or tasks. Nodes correspond to tasks or operations, edges correspond to relations („executed after“).
- Usually, a dependence graph describes a *partial order between operations* and therefore, leaves freedom for scheduling (parallel or sequential). It represents parallelism in a program but no branches in control flow.
- A *dependence graph is acyclic*.
- Often, there are additional quantities associated to edges or nodes such as
 - execution times, deadlines, arrival times
 - communication demand

Dependence Graph and Single Assignment Form

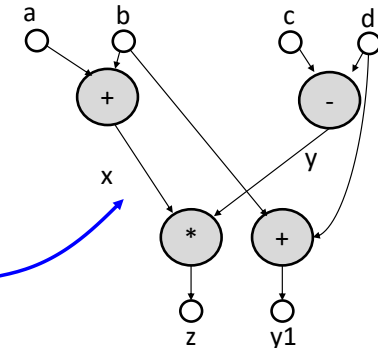
given basic block:

```
x = a + b;
y = c - d;
z = x * y;
y = b + d;
```

single assignment form:

```
x = a + b;
y = c - d;
z = x * y;
y1 = b + d;
```

dependence graph



Limit with reassignment, i.e. $x = a - x$, iteration, loops.

10 - 12

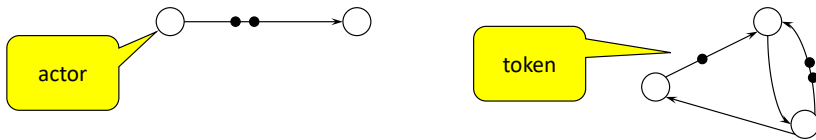
10 - 13

Marked Graph (MG)

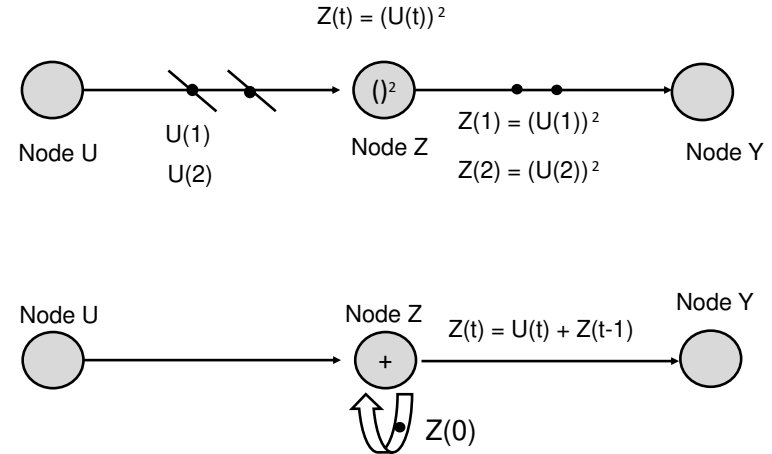
- A **marked graph** $G = (V, A, del)$ consists of

- nodes (actors) $v \in V$
- edges $a = (v_i, v_j) \in A, A \subseteq V \times V$
- number of initial tokens (or marking) on edges $del : A \rightarrow \mathbb{Z}^{\geq 0}$

- The **marking** is often represented in form of a vector: $del = \begin{pmatrix} del_1 \\ \vdots \\ del_i \\ \vdots \\ del_{|A|} \end{pmatrix}$



Starting with some simple example



10 - 14

10 - 15

Marked Graph

- The **token** on the edges correspond to data that are stored in FIFO queues.
- A **node (actor)** is called **activated** if on every input edge there is at least one token.
- A node (actor) can **fire** if it is activated.
- The **firing of a node** v_i (actor operates on the first tokens in the input queues) removes from each input edge a token and adds a token to each output edge. The output token correspond to the processed data.
- Marked graphs are mainly used for modeling regular computations, for example signal flow graphs.

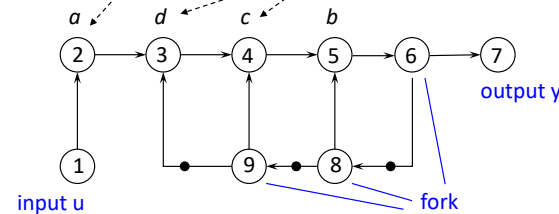
Marked Graph

Example (model of a digital filter with infinite impulse response IIR)

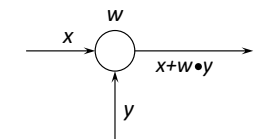
- Filter equation:

$$y(l) = a \cdot u(l) + b \cdot y(l-1) + c \cdot y(l-2) + d \cdot y(l-3)$$

- Possible model as a **marked graph**:



nodes 3-5:



node 2: $x=0$

10 - 16

10 - 17

Models for Architecture Synthesis

- **A sequence graph** $G_S = (V_S, E_S)$ is a dependence graph with a single start node (no incoming edges) and a single end node (no outgoing edges).
 V_S denotes the operations of the algorithm and E_S denotes the dependence relations.
- **A resource graph** $G_R = (V_R, E_R)$, $V_R = V_S \cup V_T$ models resources and bindings.
 V_T denote the resource types of the architecture and G_R is a bipartite graph. An edge $(v_s, v_t) \in E_R$ represents the availability of a resource type v_t for an operation v_s .
- **Cost function** $c : V_T \rightarrow \mathbb{Z}$
- **Execution times** $w : E_R \rightarrow \mathbb{Z}^{\geq 0}$ are assigned to each edge $(v_s, v_t) \in E_R$ and denote the execution time of operation $v_s \in V_S$ on resource type $v_t \in V_T$.

10-18

Models for Architecture Synthesis - Example

Example sequence graph:

- Algorithm (differential equation):

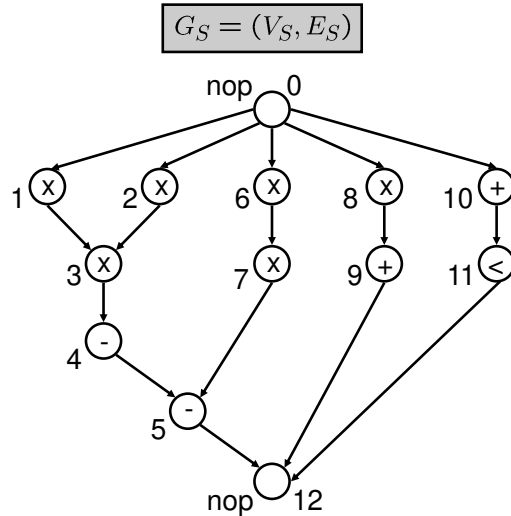
```
int diffeq(int x, int y, int u, int dx, int a) {
    int x1, u1, y1;
    while ( x < a ) {
        x1 = x + dx;
        u1 = u - (3 * x * u * dx) - (3 * y * dx);
        y1 = y + u * dx;
        x = x1;
        u = u1;
        y = y1;
    }
    return y;
}
```

10-19

Models for Architecture Synthesis - Example

- **Corresponding sequence graph:**

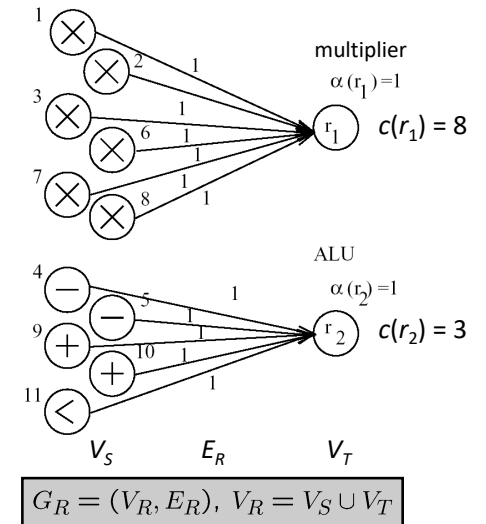
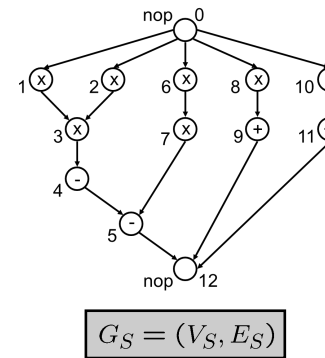
```
int diffeq(int x, int y, int u, int dx, int a) {
    int x1, u1, y1;
    while ( x < a ) {
        x1 = x + dx;
        u1 = u - (3 * x * u * dx) - (3 * y * dx);
        y1 = y + u * dx;
        x = x1;
        u = u1;
        y = y1;
    }
    return y;
}
```



10-20

Models for Architecture Synthesis - Example

- **Corresponding resource graph** with one instance of a multiplier (cost 8) and one instance of an ALU (cost 3):



10-21

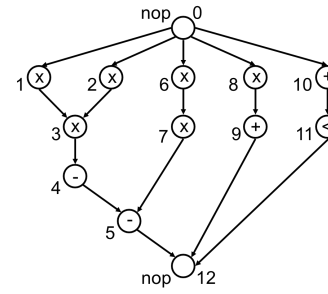
Allocation and Binding

An allocation is a function $\alpha : V_T \rightarrow \mathbb{Z}^{\geq 0}$ that assigns to each resource type $v_t \in V_T$ the number $\alpha(v_t)$ of available instances.

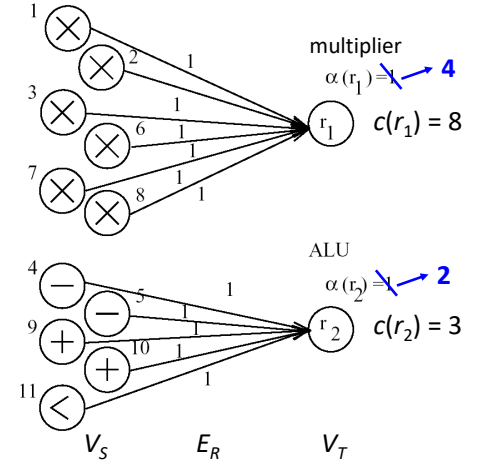
A binding is defined by functions $\beta : V_S \rightarrow V_T$ and $\gamma : V_S \rightarrow \mathbb{Z}^{>0}$. Here, $\beta(v_s) = v_t$ and $\gamma(v_s) = r$ denote that operation $v_s \in V_S$ is implemented on the r th instance of resource type $v_t \in V_T$.

Models for Architecture Synthesis - Example

- Corresponding resource graph with 4 instances of a multiplier (cost 8) and two instance of an ALU (cost 3):



$$G_S = (V_S, E_S)$$



$$G_R = (V_R, E_R), V_R = V_S \cup V_T$$

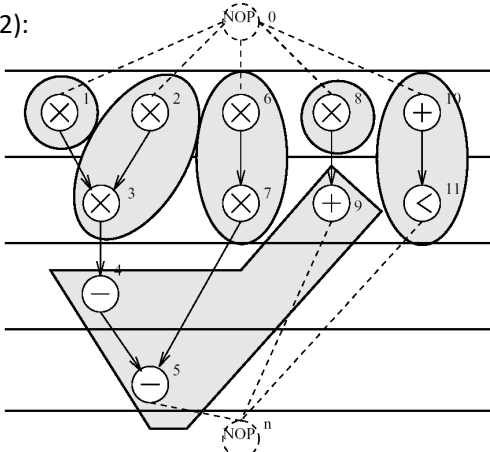
10 - 22

10 - 23

Models for Architecture Synthesis - Example

- Example binding ($\alpha(r_1) = 4, \alpha(r_2) = 2$):

$\beta(v_1) = r_1, \gamma(v_1) = 1,$
 $\beta(v_2) = r_1, \gamma(v_2) = 2,$
 $\beta(v_3) = r_1, \gamma(v_3) = 2,$
 $\beta(v_4) = r_2, \gamma(v_4) = 1,$
 $\beta(v_5) = r_2, \gamma(v_5) = 1,$
 $\beta(v_6) = r_1, \gamma(v_6) = 3,$
 $\beta(v_7) = r_1, \gamma(v_7) = 3,$
 $\beta(v_8) = r_1, \gamma(v_8) = 4,$
 $\beta(v_9) = r_2, \gamma(v_9) = 1,$
 $\beta(v_{10}) = r_2, \gamma(v_{10}) = 2,$
 $\beta(v_{11}) = r_2, \gamma(v_{11}) = 2$



10 - 24

Scheduling

A schedule is a function $\tau : V_S \rightarrow \mathbb{Z}^{>0}$ that determines the starting times of operations. A schedule is feasible if the conditions

$$\tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S$$

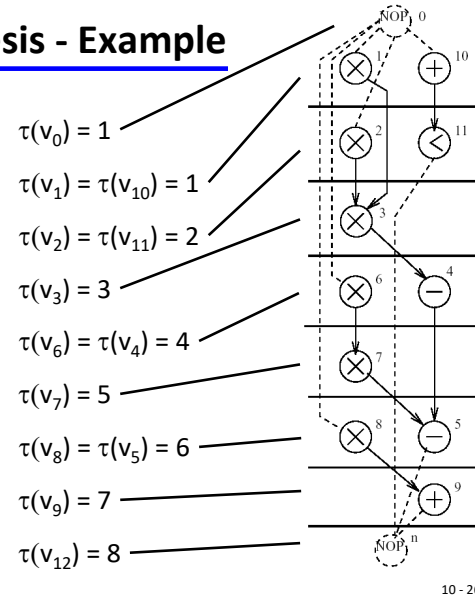
are satisfied. $w(v_i) = w(v_i, \beta(v_i))$ denotes the execution time of operation v_i .

The latency L of a schedule is the time difference between start node v_0 and end node v_n :
 $L = \tau(v_n) - \tau(v_0)$.

10 - 25

Models for Architecture Synthesis - Example

Example: $L = \tau(v_{12}) - \tau(v_0) = 7$



Architecture Synthesis without Resource Constraints

Synthesis Algorithms

Classification

- *unlimited resources:*
 - no constraints in terms of the available resources are defined.
- *limited resources:*
 - constrains are given in terms of the number and type of available resources.

Classes of synthesis algorithms

- *iterative algorithms:*
 - an initial solution to the architecture synthesis is improved step by step.
- *constructive algorithms:*
 - the synthesis problem is solved in one step.
- *transformative algorithms:*
 - the initial problem formulation is converted into a (classical) optimization problem.

Synthesis/Scheduling Without Resource Constraints

The corresponding scheduling method can be used

- as a *preparatory step* for the general synthesis problem
- to determine *bounds on feasible schedules* in the general case
- if there is a *dedicated resource* for each operation.

Given is a sequence graph $G_S(V_S, E_S)$ and a resource graph $G_R(V_R, E_R)$. Then the latency minimization without resource constraints with $\alpha(v_i) \rightarrow \infty$ for all $v_i \in V_T$ is defined as

$$L = \min\{\tau(v_n) - \tau(v_0) : \tau(v_j) - \tau(v_i) \geq w(v_i, \beta(v_i)) \forall (v_i, v_j) \in E_S\}$$

ASAP Algorithm

ASAP = As Soon As Possible

```

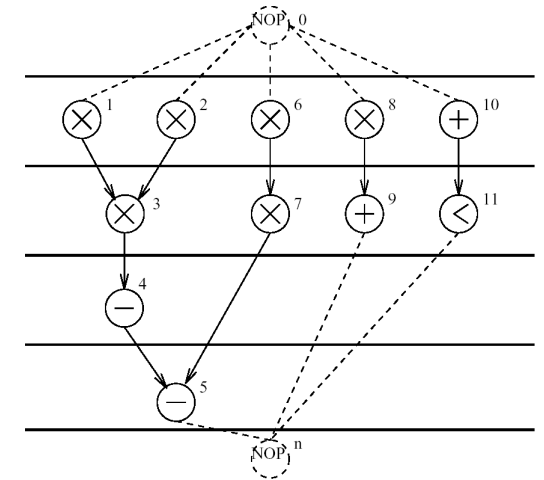
ASAP( $G_S(V_S, E_S), w$ ) {
   $\tau(v_0) = 1$ ;
  REPEAT {
    Determine  $v_i$  whose predec. are planed;
     $\tau(v_i) = \max\{\tau(v_j) + w(v_j) \mid (v_j, v_i) \in E_S\}$ 
  } UNTIL ( $v_n$  is planned);
  RETURN ( $\tau$ );
}

```

The ASAP Algorithm - Example

Example:

$$w(v_i) = 1$$



10 - 30

10 - 31

ALAP Algorithm

ALAP = As Late As Possible

```

ALAP( $G_S(V_S, E_S), w, L_{max}$ ) {
   $\tau(v_n) = L_{max} + 1$ ;
  REPEAT {
    Determine  $v_i$  whose succ. are planed;
     $\tau(v_i) = \min\{\tau(v_j) \mid (v_i, v_j) \in E_S\} - w(v_i)$ 
  } UNTIL ( $v_0$  is planned);
  RETURN ( $\tau$ );
}

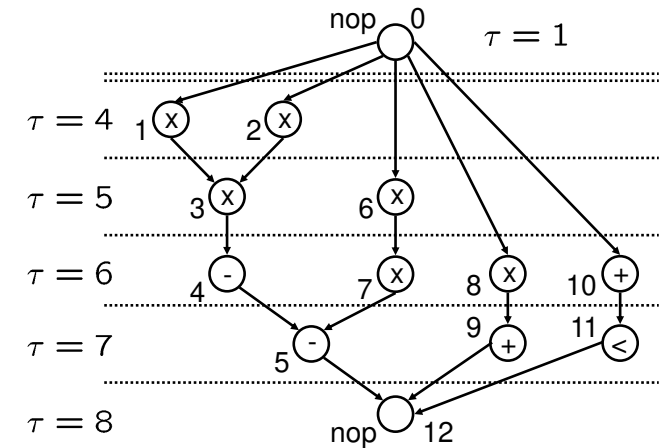
```

ALAP Algorithm - Example

Example:

$$L_{max} = 7$$

$$w(v_i) = 1$$



10 - 32

10 - 33

Scheduling with Timing Constraints

There are different *classes of timing constraints*:

- *deadline* (latest finishing times of operations), for example

$$\tau(v_2) + w(v_2) \leq 5$$

- *release times* (earliest starting times of operations), for example

$$\tau(v_3) \geq 4$$

- *relative constraints* (differences between starting times of a pair of operations), for example

$$\tau(v_6) - \tau(v_7) \geq 4$$

$$\tau(v_4) - \tau(v_1) \leq 2$$

10 - 34

Architecture Synthesis with Resource Constraints

Scheduling with Timing Constraints

We will model all timing constraints using relative constraints. Deadlines and release times are defined relative to the start node v_0 .

Minimum, maximum and equality constraints can be converted into each other:

- *Minimum constraint*:

$$\tau(v_j) \geq \tau(v_i) + l_{ij} \longrightarrow \tau(v_j) - \tau(v_i) \geq l_{ij}$$

- *Maximum constraint*:

$$\tau(v_j) \leq \tau(v_i) + l_{ij} \longrightarrow \tau(v_i) - \tau(v_j) \geq -l_{ij}$$

- *Equality constraint*:

$$\tau(v_j) = \tau(v_i) + l_{ij} \longrightarrow \tau(v_j) - \tau(v_i) \leq l_{ij} \wedge \tau(v_j) - \tau(v_i) \geq l_{ij}$$

10 - 35

Scheduling With Resource Constraints

Given is a sequence graph $G_S = (V_S, E_S)$, a resource graph $G_R = (V_R, E_R)$ and an associated allocation α and binding β .

Then the minimal latency is defined as

dependencies are respected
there are **not more than the available**
resources in use at any moment in
time and for any resource type

$$L = \min\{\tau(v_n) : (\tau(v_j) - \tau(v_i) \geq w(v_i, \beta(v_i))) \forall (v_i, v_j) \in E_S \wedge (|\{v_s : \beta(v_s) = v_t \wedge \tau(v_s) \leq t < \tau(v_s) + w(v_s, v_t)\}| \leq \alpha(v_t)) \forall v_t \in V_T, \forall 1 \leq t \leq L_{max}\}$$

where L_{max} denotes an upper bound on the latency.

10 - 36

10 - 37

List Scheduling

List scheduling is one of the most widely used algorithms for scheduling under resource constraints.

Principles:

- To each operation there is a *priority* assigned which denotes the urgency of being scheduled. This *priority is static*, i.e. determined before the List Scheduling.
- The algorithm schedules one time step after the other.
- U_k denotes the set of operations that (a) are mapped onto resource v_k and (b) whose predecessors finished.
- T_k denotes the currently running operations mapped to resource v_k .

10 - 38

List Scheduling

```

LIST( $G_S(V_S, E_S), G_R(V_R, E_R), \alpha, \beta, priorities$ ){
   $t = 1$ ;
  REPEAT {
     $\swarrow$  resource types
    FORALL  $v_k \in V_T$  {
       $\swarrow v \in V_S$  with  $\beta(v) = v_k$ 
      determine candidates to be scheduled  $U_k$ ;
      determine running operations  $T_k$ ;
      choose  $S_k \subseteq U_k$  with maximal priority
        and  $|S_k| + |T_k| \leq \alpha(v_k)$ ;
       $\tau(v_i) = t \ \forall v_i \in S_k$ ;
    }
     $t = t + 1$ ;
  } UNTIL ( $v_n$  planned)
  RETURN ( $\tau$ );
}

```

10 - 39

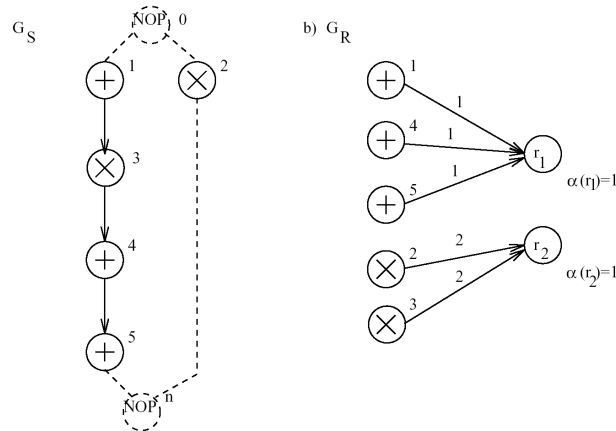
List Scheduling - Example

Example:

```

LIST( $G_S(V_S, E_S), G_R(V_R, E_R), \alpha, \beta, priorities$ ){
   $t = 1$ ;
  REPEAT {
    FORALL  $v_k \in V_T$  {
      determine candidates to be scheduled  $U_k$ ;
      determine running operations  $T_k$ ;
      choose  $S_k \subseteq U_k$  with maximal priority
        and  $|S_k| + |T_k| \leq \alpha(v_k)$ ;
       $\tau(v_i) = t \ \forall v_i \in S_k$ ;
    }
     $t = t + 1$ ;
  } UNTIL ( $v_n$  planned)
  RETURN ( $\tau$ );
}

```



10 - 40

Architecture Synthesis for Iterative Algorithms and Marked Graphs

10 - 41

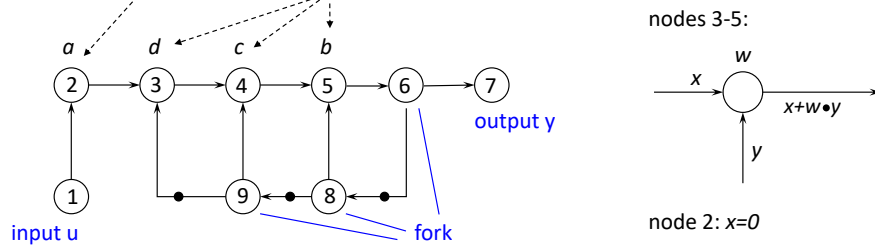
Remember ... : Marked Graph

Example (model of a digital filter with infinite impulse response IIR)

- Filter equation:

$$y(l) = a \cdot u(l) + b \cdot y(l-1) + c \cdot y(l-2) + d \cdot y(l-3)$$

- Possible model as a **marked graph**:



10 - 42

Iterative Algorithms

- Iterative algorithms** consist of a set of **indexed equations** that are evaluated for all values of an index variable l :

$$x_i[l] = F_i[\dots, x_j[l - d_{ji}], \dots] \quad \forall l \quad \forall i \in I$$

Here, x_i denote a set of indexed variables, F_i denote arbitrary functions and d_{ji} are constant index displacements.

- Examples of well known representations are **signal flow graphs** (as used in signal and image processing and automatic control), **marked graphs** and special forms of loops.

10 - 43

Iterative Algorithms

Several **representations** of the same iterative algorithm:

- One indexed equation with constant index dependencies:

$$y[l] = au[l] + by[l-1] + cy[l-2] + dy[l-3] \quad \forall l$$

- Equivalent set of indexed equations:

$$x_1[l] = au[l] \quad \forall l$$

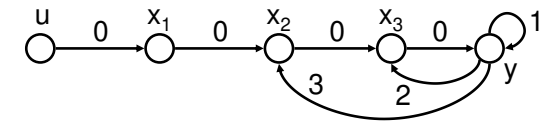
$$x_2[l] = x_1[l] + dy[l-3] \quad \forall l$$

$$x_3[l] = x_2[l] + cy[l-2] \quad \forall l$$

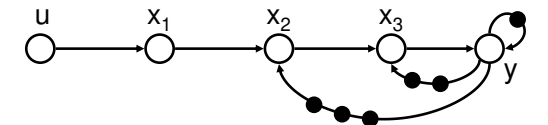
$$y[l] = x_3[l] + by[l-1] \quad \forall l$$

Iterative Algorithms

Extended sequence graph $G_S = (V_S, E_S, d)$: To each edge $(v_i, v_j) \in E_S$ there is associated the index displacement d_{ij} . An edge $(v_i, v_j) \in E_S$ denotes that the variable corresponding to v_j depends on variable corresponding to v_i with displacement d_{ij} .



Equivalent **marked graph**:

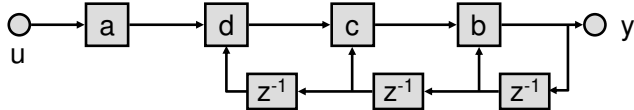


10 - 44

10 - 45

Iterative Algorithms

- *Equivalent signal flow graph:*



- *Equivalent loop program:*

```
while(true) {
    t1 = read(u);
    t5 = a*t1 + d*t2 + c*t3 + b*t4;
    t2 = t3;
    t3 = t4;
    t4 = t5;
    write(y, t5);
}
```

10 - 46

Iterative Algorithms

- An *iteration* is the set of all operations necessary to compute all variables $x_i[l]$ for a fixed index l .
- The *iteration interval* P is the time distance between two successive iterations of an iterative algorithm. $1/P$ denotes the *throughput* of the implementation.
- The *latency* L is the maximal time distance between the starting and the finishing times of operations belonging to one iteration.
- In a pipelined implementation (*functional pipelining*), there exist time instances where the operations of different iterations l are executed simultaneously.

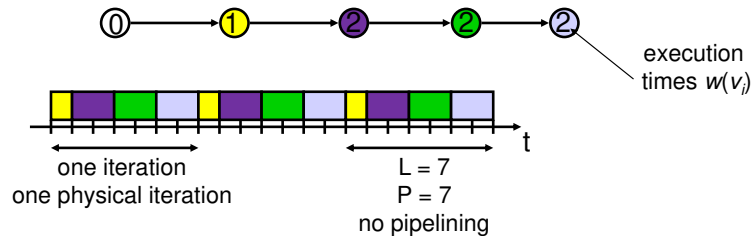
10 - 47

Iterative Algorithms

- *Implementation principles*

- A *simple possibility*, the edges with $d_{ij} > 0$ are removed from the extended sequence graph. The resulting simple sequence graph is implemented using *standard methods*.

Example with unlimited resources:



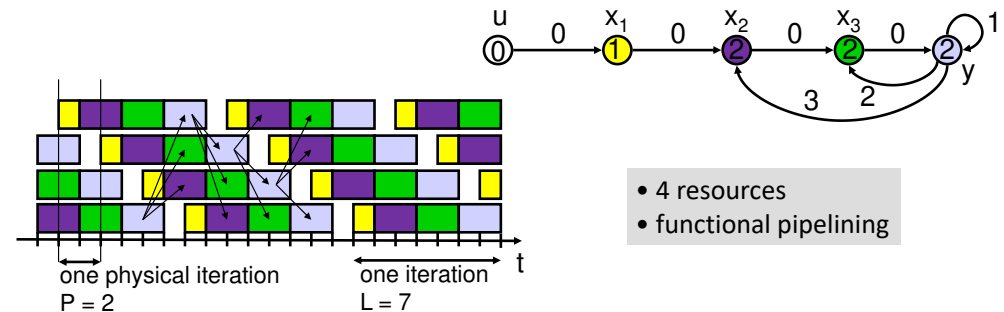
10 - 48

Iterative Algorithms

- *Implementation principles*

- Using *functional pipelining*: Successive iterations overlap and a higher throughput ($1/P$) is obtained.

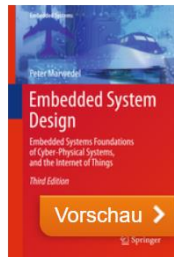
Example with unlimited resources (note data dependencies across iterations!)



10 - 49

Chapter 8

- Not covered this semester.
- Not covered in exam.
- If interested: Read



© 2018

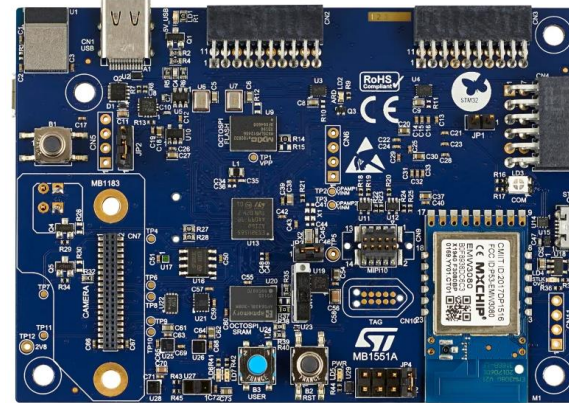
Embedded System Design

Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things

Autoren: Marwedel, Peter

» Zeige nächste Auflage

Remember: What you got some time ago ...



10 - 50

10 - 51

What we told you: Be careful and please do not ...



**Return the boards at the
embedded systems exam!**

10 - 52

10 - 53