Embedded Systems

Theory

Christian Rossi

Academic Year 2024-2025

Abstract

The course delves into embedded systems, covering their characteristics, requirements, and constraints. It explores hardware architectures, including various types of software executors, communication methods, interfacing techniques, off-the-shelf components, and architectures suited for both prototyping and large-scale production.

In terms of software architectures, the course examines abstraction levels, real-time operating systems, complex networked systems, and the tools and methodologies used for code analysis, profiling, and optimization.

Students will also learn to analyze and optimize hardware/software architectures for embedded systems, focusing on managing design constraints and selecting appropriate architectures. Key topics include estimating and optimizing performance and power at various abstraction levels, project management, and designing for reuse.

Additionally, the course addresses run-time resource management and includes case studies to illustrate trade-offs based on application fields and system sizes.

Contents

CHAPTER 1

Introduction

1.1 Introduction

Embedded systems are characterized by their ubiquitous presence, low power consumption, high performance, and interconnected nature. These systems are commonly utilized in four main application contexts:

- *Public infrastructures*: safety is a primary concern to prevent potential attacks, and in some cases, latency is also crucial. Examples include highways, bridges, and airports.
- *Industrial systems*: reliability and safety are the predominant concerns. Key industries utilizing embedded systems include automotive, aerospace, and medical.
- *Private spaces*: this includes control systems for houses and offices.
- *Nomadic system*: these systems involve data collection related to the health and positions of animals and people, requiring both security and low latency.

In the future, embedded systems will evolve in several key ways:

- *Networked*: transitioning from isolated operations to interconnected, distributed solutions.
- *Secure*: addressing significant security challenges that impact both technical and economic viability.
- *Complex*: enhanced by advancements in nanotechnology and communication technologies.
- *Low power*: utilizing energy scavenging methods.
- *Thermal and power control*: implementing runtime resource management.

Usually, we process useful data locally and send the relevant data to the cloud less frequently. This approach conserves energy, thereby extending battery life. Due to time constraints, developing a device from scratch is often infeasible, so we typically collaborate with eco-alliance partners. The design process involves a multidisciplinary team, as it integrates multiple domains of expertise. The first prototype resulting from this design phase is called the Minimum Viable Product (MVP), which is the initial sellable version of the product.

1.1.1 Technological problems

Applications are expanding rapidly, pushing the need for mass-market compatibility and integrating into all aspects of life, which in turn drives up volumes. As technology evolves, the scale of integration grows, supported by new materials and programming paradigms. However, finding a balance in this progress is complicated by several factors. CMOS technology is reaching its physical limits, and the cost of developing new foundational technologies can often be unaffordable. Additionally, the power and energy demands create significant barriers, as does the exponential proliferation of data. Transitioning from invention to innovation is proving difficult, and the design methodologies in use today heavily exploit human effort—though this reliance on human ingenuity is, in some ways, a fortunate necessity.

Vertical applications Vertical applications requires various technologies to work ad desired such as sensors, computing units, storage, communication elments, and so on. Energy and power dissipation have become even more problematic with the introduction of the newest technology nodes, exacerbating existing challenges. Dependability, which encompasses security, safety, and privacy, is now a major concern. At the same time, the growing complexity of systems is reaching nearly unmanageable levels. Despite this, complexity continues to rise, driven by applications that build upon increasingly interconnected systems of systems.

1.2 Applications future

Future applications will be highly compute-intensive, requiring efficient hardware and software components across various domains, including embedded systems, mobile devices, and data centers. Key characteristics of these future applications include:

- *Compute-intensiveness*: they will demand significant computational resources, necessitating optimized hardware and software regardless of their application domain.
- *Connectivity*: these applications will be interconnected, either wired or wirelessly, and will often be online—globally interconnected through the Internet.
- *Physical entanglement*: they will be embedded within and capable of interacting with the physical world, not only observing but also controlling their environment. These systems will effectively merge into our everyday surroundings.
- *Intelligence*: these applications will possess the capability to interpret noisy, incomplete, analog, or remote data from the physical world, allowing for smarter interactions and decision-making.

All major future applications will exhibit these traits to varying degrees. The ongoing integration of the digital and physical worlds (manifested through the Internet of Things (IoT) and Cyber-Physical Systems (CPS)) will be driven by advances in cognitive computing, big data analytics, and data mining.

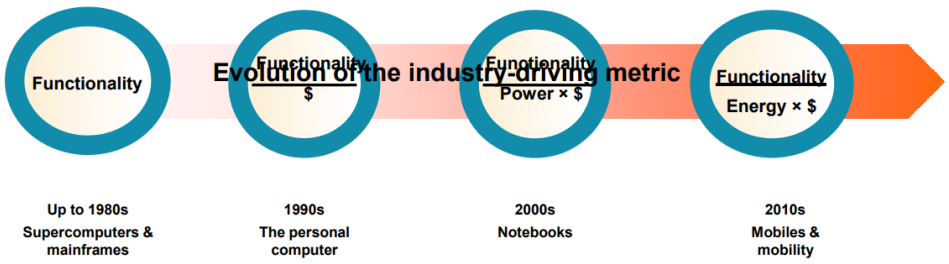


Figure 1.1: Industry requirements

The challenge is that optimizing for functionality, energy, and cost remains difficult:

$$\frac{\text{functionality}}{\text{energy} \times \text{cost}}$$

However, successfully optimizing this metric often leads to improvements across simpler metrics as well.

While the number of transistors in modern systems can continue to increase, a key challenge is that we cannot power all of them simultaneously. This limitation drives the need for innovative approaches to using extra transistors more efficiently. Techniques like multi-core and many-core processors, as well as domain-specific processors, are becoming essential. Additionally, heterogeneous processing combined with aggressive power management is crucial for optimizing performance across various tasks. As data generation continues to accelerate it becomes increasingly important to ensure that computation is carried out in the most efficient location. This efficiency is necessary to cope with the imbalance between the massive growth of data and the slowing progress of Moore's Law.

1.2.1 Transistors and cores

As transistors shrink in size, several challenges emerge. Process variation, physical failures, and aging mechanisms, such as negative bias temperature instability (NBTI), can degrade device performance over time. With very-large-scale integration (VLSI), packing more transistors into smaller spaces increases power density, which in turn creates thermal issues. Furthermore, traditional communication subsystems struggle to provide adequate power-performance trade-offs in these densely packed systems.

For instance, Network-on-Chip (NoC) designs can consume up to 30% of total chip power, and their performance plays a critical role in the efficiency of multicore architectures, which are increasingly necessary for improving system performance across various applications.

1.2.2 Silicon challenges

Despite silicon being a relatively good thermal conductor (about four times worse than copper), large chips can still experience significant temperature gradients, especially in high-performance CPUs. This creates hot spots that can affect chip reliability and performance. One practical solution involves dividing the chip into concentric rings and applying dynamic voltage and frequency scaling (DVFS) to each region. By fine-tuning voltage and frequency for each ring, it becomes possible to optimize performance while maintaining thermal balance, preventing excessive heat buildup in certain areas.

1.2.3 Frequency scaling

Modern systems address power management by partitioning chips into independent islands that operate at different voltage and frequency levels. This approach allows sections of the chip to be dynamically turned off when not in use, a process known as power gating. By adjusting power usage in this way, overall system energy efficiency is greatly improved without compromising performance when it is needed.

1.2.4 Cooling systems

Cooling systems are critical to managing the thermal output of modern computing hardware, and various methods are used depending on the specific requirements. Air cooling, while common, suffers from a low heat transfer coefficient (HTC), poor chip temperature uniformity, and requires large heat sinks and air ducts, particularly in data centers. It is also noisy and expensive to maintain. Water cooling offers an improvement, with better HTC, more uniform chip temperatures, smaller heat sinks, and fewer fans. Water cooling also allows for potential heat recovery, though it requires large pumps to function effectively.

Two-phase cooling systems present an even more efficient solution. They provide higher HTC, better chip temperature uniformity, and smaller pumps, along with isothermal coolant, which helps maintain consistent temperatures across the chip. This method excels at cooling hot spots and also allows for heat recovery. However, two-phase cooling systems suffer from low pump efficiency and reliability issues.

New innovations, such as thermosyphon cooling, are emerging as promising alternatives, offering advanced methods to maintain chip performance while effectively managing heat dissipation.

1.3 Productivity and planning

The revenue model can be visualized simply as a triangle, where the product's life is represented by a span of 2W, peaking at W. The time of market entry defines the triangle, representing market penetration, with the area of the triangle corresponding to total revenue. Any delay in market entry results in a loss, which is the difference between the areas of the on-time and delayed triangles.

The productivity gap, however, reveals a more challenging situation. In theory, increasing the number of designers on a team should reduce project completion time. In practice, though, productivity per designer tends to decrease due to the complexities of team management and communication, a phenomenon famously referred to as the mythical man-month (Brooks 1975). At a certain point, adding more designers can even extend project timelines, a classic case of too many cooks spoiling the broth.

1.3.1 Platform-based design

To enhance productivity, the design methodology must support reuse, particularly at higher abstraction levels, and this should be backed by standardization. As integrated systems increasingly require both digital and non-digital functionalities, this dual trend is captured by the International Technology Roadmap for Semiconductors. It emphasizes the miniaturization of digital functions, often referred to as More Moore, and functional diversification, known as More-than-Moore.

1.3.2 Technology trends

Two significant trends in technology development are System-on-Chip (SoC) and System-in-Package (SiP). SoC focuses on full integration and achieving the lowest cost per transistor, while SiP focuses on lowering the cost per function for the entire system. These architectures are complementary rather than competitive, each requiring distinct industrial approaches and advanced research and design knowledge. SoC emphasizes miniaturization, whereas SiP centers on integrating multiple components, necessitating different manufacturing competencies for each.

MEMS Micro-Electro-Mechanical Systems (MEMS) involve the creation of 3D structures using integrated circuit fabrication technologies and specialized micromachining processes, typically on silicon or glass wafers. MEMS devices include transducers, microsensors, microactuators, and other mechanically functional microstructures. Applications range from microfluidics (valves, pumps, and flow channels) to microengines (gears, turbines, combustion engines). Integrated microsystems combine circuitry and transducers to perform tasks autonomously or with the assistance of a host computer. MEMS components bridge the gap between the electrical and non-electrical world, where sensors receive inputs from non-electronic events and actuators output to them.

Energy scavenging Energy scavenging involves capturing energy from objects with temperature gradients. Another source of scavenging is vibrations, such as self-winding watches, which generate around 5 microwatts on average when worn and up to 1 milliwatt when shaken vigorously.

Wireless sensor nodes Wireless sensor nodes are small, battery-powered devices that monitor local conditions. These devices typically have limited resources and form nodes within a wireless network that covers a region or object of interest. Wireless sensor nodes enable new applications by collecting, fusing, reasoning, and responding to sensor data. These applications can lead to smarter systems in fields ranging from environmental monitoring to industrial automation.

CHAPTER 2

Microprocessors

2.1 Introduction

Microprocessors play a crucial role in embedded systems by implementing software algorithms that govern their functionality. The key features of microprocessor-based applications include:

- *Flexibility*: this encompasses maintainability and the ability to evolve the application. Software development becomes less complex and faster, verification processes are less critical, and a larger pool of software designers is available.
- *Time To Market* (TTM): this refers to the duration required to bring a product to the customer, emphasizing the importance of efficient development cycles.
- *Easy upgrade*: software solutions can be updated and improved with minimal disruption.
- *Cost*: the overall cost is volume-dependent. Design costs are incurred only once, while production expenses are primarily linked to the cost of silicon.

This approach is cost-effective since embedded systems often do not require the full processing power of a microprocessor; they typically utilize only a fraction of it. Consequently, microprocessors retain only the necessary functions and capabilities, resulting in reduced production costs and material consumption.

While an equivalent hardware solution may generally provide better performance, microprocessor architectures are optimized for flexibility, making them more generic and less specialized. Performance encompasses various factors beyond computing speed, such as energy consumption, power efficiency, memory footprint, and chip area. While software implementations are typically easier to develop than hardware solutions, modifying hardware to meet new requirements is often simpler, with performance advantages remaining on the hardware side.

Designing a software solution necessitates a comprehensive understanding of the microprocessor architectures available in the market. The characteristics of the problem (algorithm) should guide the designer, alongside non-functional constraints such as performance, cost, development time, and power consumption. The initial step before selecting a specific processor is determining the appropriate class of processor and the form in which it will be acquired.

2.2 Processors taxonomy

Processors can be categorized into two main types:

- *Application Specific Processors* (ASP): these processors are tailored for specific classes of applications that require high performance, handling numerous operations per second.
- *General Purpose Processors* (GPP): known as versatile four-season processors, GPPs are not optimized for any particular application, making them suitable for various types of tasks.

In typical embedded systems, a multi-processor solution often employs GPPs for supervising and controlling the activities of one or more ASPs.

Processors availability Processors can further be classified as follows:

- *Components Off The Shelf* (COTS): these are standard chips purchased off-the-shelf and mounted onto a printed circuit board (PCB) along with the necessary interfaces to integrate with the rest of the system.
- *Intellectual Property* (IP): this involves purchasing the design specifications of a microprocessor. There are several abstraction levels in IP:
  - *Soft-macro*: the microprocessor is described using Hardware Description Language (HDL) at the register-transfer (RT) level.
  - *Hard-macro*: the description includes details down to the layout level.

With the increasing prevalence of Programmable Logic Devices (PLDs), the use of IP is becoming more popular. Suppliers of Complex Programmable Logic Devices (CPLDs) and/or Field Programmable Gate Arrays (FPGAs) often provide one or more cores, sometimes even for free.

2.2.1 Selection process

The selection process is based on:

- *Class*: the nature of the algorithm and the operations and data to be processed are the primary drivers for selecting a processor class.
- *Form*: the target architecture of the system plays a crucial role.
- *Performance*: a key metric for performance is the average number of instructions per clock cycle (IPC/CPI), which is relative to the clock frequency and should be scaled for comparison across different architectures. MIPS (Million Instructions Per Second) serves as an absolute measure of throughput but can be misleading when comparing processors with different Instruction Sets (ISAs). For floating-point operations, MFLOPS is commonly used, while specialized architectures like Digital Signal Processors (DSPs) often use MMACS (Million Multiply-Accumulate Operations Per Second), and Network Processors (NPs) typically measure the average number of processed packets per time unit.

- *Power*: power efficiency is perhaps the most critical driver for embedded systems. Both average power and peak power are important metrics for estimating the maximum or average power consumption of the overall system. Initially, a combined measure of power and speed (related to performance) is often used.

Other important considerations include:

- *Memory*: the bandwidth and size requirements of the application can impose hard constraints. In some cases, only internal memory may be available, making external memory utilization impractical. Achieving sufficient bandwidth may require integrated memory solutions, and for complex systems, addressing space can be a critical factor.

- *Peripherals*: embedded systems often process external signals and control physical devices. Designers can either choose an architecture focused on computation and design the rest of the system accordingly or opt for a single-chip solution that integrates computing, interfacing, and peripherals. Selecting the appropriate microprocessor can necessitate using a PCB or a System on Chip (SoC) that consolidates all peripherals. Using a microcontroller simplifies integration challenges, albeit with fewer alternatives available.

- *Software*: many embedded applications have limited legacy code and primarily utilize standard functions and libraries. The availability of libraries can simplify both the design and validation processes, making the development of software solutions feasible that might otherwise be impractical. Access to an operating system and SDK specific to the processor is also vital, serving as the foundation for software development. Differences among SDKs are significant, and factors such as the software compilation flow, code quality, and flexibility offered to the designer are important. The availability and reliability of analysis tools, including debuggers, as well as documentation and reference designs, contribute to the ecosystem surrounding the microprocessor.

- *Packaging*: for COTS processors, various packages are available, differing in size, pinout, and materials.

- *Certifications*: different certification standards exist, including consumer, industrial, aerospace, automotive, and military certifications. Some certifications are tailored for specific fields, such as MISRA rules for automotive code. The availability of a component with the appropriate certification can be a critical constraint for its adoption.

2.3 General Pupropose Processors

General Purpose Processors (GPPs) are characterized by a generic architecture that is applicable across a wide range of fields. They are commonly found in PCs and Personal Digital Assistants (PDAs), and for embedded systems, they are suitable for low-end applications where energy efficiency and performance are not critical, and the application nature is quite heterogeneous. GPPs handle tasks such as controlling and managing slow interfaces with sensors and enabling interactivity through graphical user interfaces (GUIs). GPPs can be categorized into two main architectural styles:

- *Complex Instruction Set Computer* (CISC): features a large number of complex instructions.

- *Reduced Instruction Set Computer* (RISC): utilizes a smaller set of simple instructions.

2.3.1 Complex Instruction Set Computer

CISC architectures aim to allow each arithmetic or logic operation to access data and write results using any available addressing mode. They feature a fully orthogonal architecture and instruction set. However, in practice, the number of combinations of operations and addressing modes is often limited, typically with one operand referencing memory and the other being a register that acts as both source and destination. Despite the complexity, CISC instructions are encoded in variable-length formats, which necessitates more complex fetch and decode units and a higher speed for memory access.

The Arithmetic Logic Unit (ALU) implements various basic operations such as addition, subtraction, multiplication, division, logic operations, shifting, and comparisons. Some architectures even support vector instructions, enabling operations on multiple registers simultaneously, indicative of a Single Instruction Multiple Data (SIMD) architecture. The diversity in addressing modes and arithmetic/logical operations requires a complex data path that is challenging to optimize for speed. To enhance throughput, CISC processors often employ intricate pipelining techniques (e.g., up to 23 stages), which can be challenging to manage. Additionally, they may alter the order of execution for assembly instructions without changing the program's semantics. This complexity makes it difficult for compilers to predict instruction execution status, potentially leading to suboptimal instruction decoding. Modern CISC processors integrate hardware units that dynamically reschedule instructions, enabling out-of-order execution.

Although CISC architectures provide impressive performance, the focus on energy efficiency and cost has gradually shifted toward other architectures.

2.3.2 Reduced Instruction Set Computer

RISC architectures simplify instruction sets, utilizing a limited number of straightforward instructions. By the late 1980s, the declining prices of DRAMs and the increasing scale of integration allowed for more complex architectures on a single chip. CISC operations could be decomposed into multiple RISC instructions, streamlining architecture.

The primary advantages of RISC architectures include a simple instruction set and straightforward architecture, which simplify instruction decoding. RISC instructions typically have fixed lengths, facilitating balanced pipeline stages and higher clock speeds. Execution units, such as the ALU and Branch Processing Unit (BPU), benefit from simpler instructions, allowing RISC architectures to operate predominantly on registers, thus improving speed and predictability.

RISC employs a load/store architecture for input/output operations, which involves preloading data and writing results back to memory. While the limited number of addressing modes can aid performance, effective use of memory hierarchy is crucial for minimizing data access time.

When comparing RISC and CISC using the same high-level source code, RISC generally requires a higher number of instructions. RISC architectures do not have hardware mechanisms to resolve pipeline conflicts, but their predictable execution allows compilers to generate efficient code. Furthermore, RISC architectures typically exhibit lower power consumption due to their inherent simplicity compared to CISC architectures.

2.3.3 Superscalar

Superscalar architectures feature multiple execution units, enabling the simultaneous execution of more than one instruction per clock cycle. This parallelism enhances throughput but also

increases the complexity of control logic, impacting clock speed and power consumption. In these architectures, the microprocessor dynamically schedules instructions, relieving the compiler from managing code organization for specific hardware structures. However, a significant portion of the processor's resources is dedicated to implementing complex scheduling mechanisms and maintaining consistency.

2.3.4 Complex Instruction Set Computer and Reduced Instruction Set Computer

Some architectures combine out-of-order execution with superscalarity while maintaining compatibility with legacy code. The x86 instruction set serves as the de facto standard for CISC, where each instruction is decomposed into a series of simpler instructions that can be executed by an efficient core. This approach incurs additional hardware costs but merges the advantages of backward compatibility with the high performance achievable through a RISC core.

2.3.5 Very Long Instruction Word

To address the challenges associated with hardware scheduling complexity, Intel developed a class of processors known as EPIC (Explicitly Parallel Instruction Computing) or VLIW (Very Long Instruction Word). VLIW supports explicit parallelism, allowing multiple instructions to be executed simultaneously under program control. A VLIW instruction consolidates multiple elementary instructions, typically of RISC nature, into a single wide word with a fixed structure.

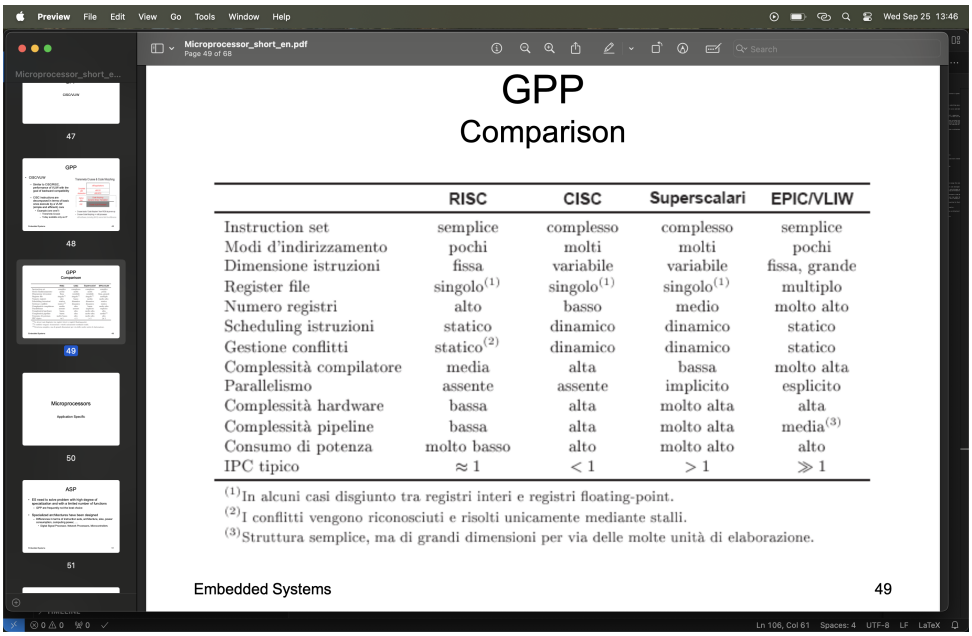
Once fetched from memory, the VLIW word is decoded in parallel, with individual RISC instructions dispatched to various execution units. This fixed structure shifts the complexity of scheduling onto the compiler, which must optimize algorithms to fit the specific architecture.

The goal of EPIC/VLIW is to transfer the burden of scheduling and code optimization to the compiler, resulting in simpler architectures, although with large sizes due to the number of execution units. Power consumption for VLIW architectures falls between that of simpler RISC and superscalar designs. Similar strategies have also been implemented in specialized architectures such as digital signal processors (DSPs) to achieve peak performance.

2.3.6 Complex Instruction Set Computer and Very Long Instruction Word

Similar to the CISC/RISC relationship, VLIW architectures aim for high performance while ensuring backward compatibility. CISC instructions are broken down into basic instructions that can be executed by a VLIW core, which is both simple and efficient.

2.3.7 Comparison



	RISC	CISC	Superscalari	EPIC/VLIW
Instruction set	semplice	complesso	complesso	semplice
Modi d'indirizzamento	pochi	molti	molti	pochi
Dimensione istruzioni	fissa	variabile	variabile	fissa, grande
Register file	singolo(1)	singolo(1)	singolo(1)	multiplo
Numero registri	alto	basso	medio	molto alto
Scheduling istruzioni	statico	dinamico	dinamico	statico
Gestione conflitti	statico(2)	dinamico	dinamico	statico
Complessità compilatore	media	alta	bassa	molto alta
Parallelismo	assente	assente	implicito	esplicito
Complessità hardware	bassa	alta	molto alta	alta
Complessità pipeline	bassa	alta	molto alta	media(3)
Consumo di potenza	molto basso	alto	molto alto	alto
IPC tipico	≈ 1	< 1	> 1	$\gg 1$

(1) In alcuni casi disgiunto tra registri interi e registri floating-point.
(2) I conflitti vengono riconosciuti e risolti unicamente mediante stalli.
(3) Struttura semplice, ma di grandi dimensioni per via delle molte unità di elaborazione.

Figure 2.1: Comp

2.4 Application Specific Microprocessors

Embedded systems often require high specialization and a limited set of functions, making GPPs less suitable for certain applications. To address these specific needs, specialized architectures have been developed.

2.4.1 Digital Signal Processors

Digital Signal Processors (DSPs) are among the most widely used application-specific processors, designed specifically for numerical computing tasks. Despite variations among different manufacturers, DSPs share a common architecture optimized for handling operations like multiplication and accumulation, crucial for many algorithms.

A typical operation in DSPs is represented by the equation:

$$z_{t+1} = z_t + x \cdot y$$

This operation, known as Multiply-Accumulate (MAC), is essential for DSP performance, leading to architectures that optimize both addition and multiplication, often through a single three-operand instruction.

To achieve high efficiency, DSPs are designed to optimize the execution of loops, which are prevalent in many numerical algorithms. This is possible due to certain characteristics of loop structures, such as having small bodies—typically around ten assembly instructions—where the control loop variable remains unchanged. The update of this variable is straightforward, and the access patterns for vectors are generally regular. As a result, DSPs incorporate specific hardware enhancements to facilitate loop optimization, including circular buffers for storing loop bodies and dedicated registers for control loop variables, which allow for increments and decrements without burdening the ALU.

Another challenge for DSPs is memory access speed, which can become a bottleneck. To combat this, DSP architectures often include high-bandwidth interfaces and sophisticated memory hierarchies. This might consist of high-speed buses capable of transferring wide data words, unified cache systems for both data and instructions, and specialized caches, such as Harvard architecture, which maintains separate caches for instructions and data, or SHARC architecture, which can incorporate multiple Level 0 caches for different types of data.

In addition to their flexibility and computational power, many modern DSPs adopt a VLIW architecture. This design is particularly suited for numerical applications, where control is limited, and significant parallelism can be leveraged.

2.4.2 Network Processors

Network Processors (NPs) represent another class of specialized microprocessors tailored for processing network packets. These complex SoC architectures utilize high levels of parallelism to handle the demands of network applications, such as routers. Given the ever-evolving functionalities and protocols within the networking domain, designing an NP can be quite complex. Typical operations performed by NPs include buffering packets, modifying data-link headers, searching for specific fields in IP headers, and computing CRC codes. To maximize performance, NPs incorporate dedicated hardware for high-speed I/O interfaces, queue management, cryptographic cores, and multiple RISC cores.

As network applications frequently handle multiple independent data channels, dedicated hardware units, often called Packet Processors (PPs) or Channel Processors (CPs), are utilized to manage these flows. In cases where data streams are interdependent, RISC cores may be employed for supervision and high-level management. The nature of packet processing typically involves executing a series of straightforward routines on a large volume of data, often requiring programs to be written in assembly for specific PPs, although higher-level languages like C are becoming increasingly common.

2.4.3 Summary

Microcontroller Units (MCUs) constitute another category of application-specific microprocessors. These devices integrate peripherals and interfaces onto a single chip, making them ideal for applications that do not demand high computational power but do require careful management of hardware resources and development time. MCUs are often designed without interfaces to external memory, which results in smaller programs. The lack of memory interfaces can increase pin counts, prompting some architectures to employ multiplexing techniques to limit this increase.

MCUs typically provide a variety of peripheral interfaces, including I2C, SPI, CAN, JTAG, PWM, UART/USART, watchdog timers, and analog I/O capabilities. While programming for MCUs is primarily conducted in C, assembly language is sometimes utilized for specific tasks. The SDKs available for microcontrollers can vary significantly, ranging from basic C compilers to comprehensive frameworks that assist in performance analysis and configuration management.

2.5 Microcontrollers

Microprocessors Microprocessors serve as the backbone of general-purpose computers and are utilized in a wide range of applications, from desktops to servers and supercomputers.

Designed to deliver high to very high computational power, they require significant external memory resources to operate effectively. Characterized by complex multi-core architectures, microprocessors typically boast clock speeds ranging from 2 to 5 GHz. However, they come with high costs, often falling between €100 and €5000, and exhibit substantial power consumption, ranging from 50 to 1500 watts. Additionally, microprocessors tend to have large form factors and limited interaction capabilities with their external environments.

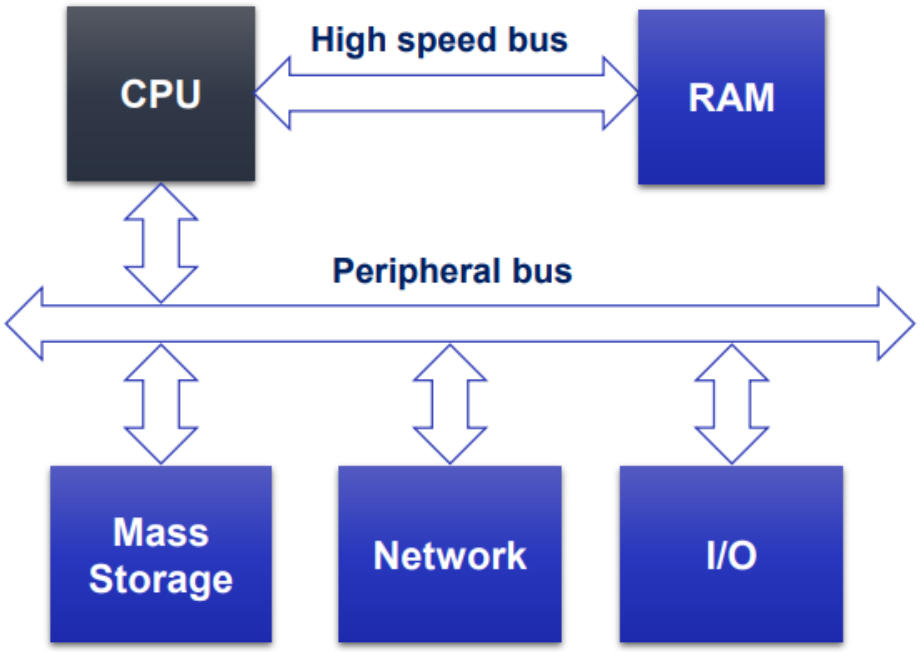


Figure 2.2: Microprocessors structure

Microcontrollers In contrast, microcontrollers are specifically designed for dedicated or embedded applications, where they perform targeted computational tasks. These devices incorporate small, integrated memory solutions, making them ideal for various applications, including home appliances and automotive systems. Microcontrollers feature simpler architectures, typically employing 8, 16, or 32-bit designs with small pipelines. Their clock speeds generally range from 8 to 160 MHz, allowing for efficient processing of specific tasks. Priced affordably between €0.5 and €10, microcontrollers are designed with low power consumption in mind, often consuming between 0.1 and 300 milliwatts. Their compact form factor enables easy integration into diverse environments, where they are engineered to interact closely with external elements.

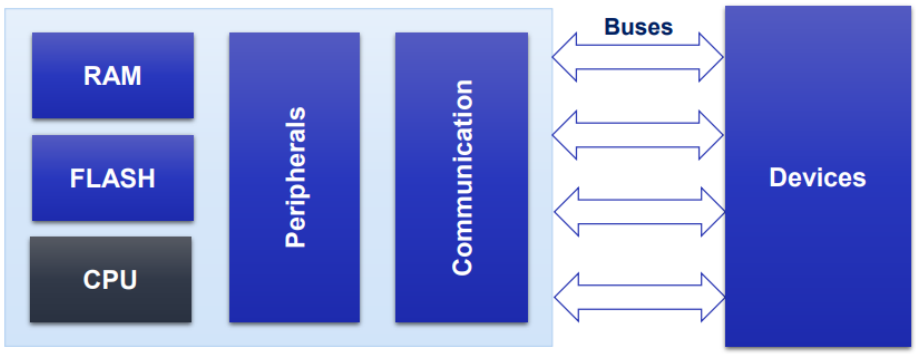


Figure 2.3: Microcontrollers structure

2.5.1 Clock

The clock is a fundamental component that synchronizes all subsystems, including the core, memories, and peripherals. In typical systems, two primary types of clocks are utilized: the main clock and the real-time clock (RTC). The main clock is responsible for driving the core, memories, and peripherals, operating within a frequency range of 2 MHz to 144 MHz. On the other hand, the RTC is primarily used for timekeeping and is often employed in low-power modes, typically operating at a frequency of 32,768 Hz.

Clock signals can originate from various sources. Internal oscillators, while convenient, tend to be imprecise due to drift caused by temperature fluctuations and significant process variability. To mitigate these issues, the frequency generated by an internal oscillator is adjusted through a Digital Phase-Locked Loop (DPLL), and it is often paired with an external crystal to enhance stability. In contrast, external oscillators provide higher accuracy as they operate independently of the microcontroller. Similar to internal oscillators, their frequencies can also be adjusted with a DPLL and are frequently used in conjunction with an external crystal.

Moreover, several internal secondary sources contribute to clock management. The main clock can be multiplied or divided by dedicated clock generation and distribution logic, which enables various clock feeds to supply different domains. Each clock domain can be individually controlled, allowing for regulation and gating tailored to specific needs.

A clock domain is defined as a group of elements that are powered by the same clock. The core domain includes components such as the CPU, RAM, FLASH memory, and timers, while the analog domain encompasses analog-to-digital converters (ADCs), digital-to-analog converters (DACs), and comparators. The bus domain incorporates interfaces like SPI, UART, and I2C, and specific domains are dedicated to protocols such as Ethernet and USB. The ability to configure interconnects among these domains ensures flexible clock distribution, which is crucial for adapting to the varying requirements of different applications.

Key characteristics of effective clock management include maintaining low drift over time and temperature, particularly for specific peripherals like watchdog timers. Typically, drift values are around 1%. Accuracy is another critical factor, especially for precise time measurement, which should fall within a range of 5 to 50 parts per million (ppm). Higher accuracy can be achieved through specialized devices or periodic resynchronization. Lastly, power consumption is an important consideration; techniques such as frequency scaling and clock gating are employed to save power during idle periods, contributing to the overall energy efficiency of the system.

Real-Time Clock The real-time clock (RTC) serves two primary functions: calendar timekeeping and generating periodic alarms. It is designed for high accuracy over extended periods, making it essential for applications that require precise time tracking. The RTC can generate alarms at regular intervals, such as every second, minute, or day, as well as aperiodic events set for specific moments in the future. One of the most critical aspects of the RTC is its drift, which arises from clock inaccuracies. To achieve more accurate timing, periodic resynchronization with external sources, such as GPS time, is necessary.

Timer Timers are versatile tools widely used in embedded applications, operating primarily in two modes: as a counter or as a timer. They offer various functionalities, including free-running timers for generating periodic interrupts, one-shot timers that generate delayed interrupts, compare functionalities for generating Pulse Width Modulation (PWM) signals, and capture modes to measure time intervals. Additionally, timers can count external events, making them integral to numerous embedded system applications.

Watchdog The watchdog is a specialized periodic timer designed to monitor the execution of applications. Configured once during the boot process, it is set to trigger an interrupt or reset at a fixed time interval. To prevent unintentional triggering, the watchdog must be cleared before it expires. There are two primary modes of operation for watchdog timers: non-windowed and windowed. In the non-windowed mode, the watchdog must be cleared before a specified time to avoid triggering an interrupt or reset. In contrast, the windowed mode requires that the watchdog be cleared within a specific time window; it must be cleared before a designated time and after another fixed time interval. If the watchdog is cleared too soon, it will trigger the interrupt or reset, ensuring that the application is functioning as intended.

2.5.2 Memory

The memory architecture in most systems is relatively straightforward, characterized by a single addressing space. Typically, this addressing space employs either 16-bit or 32-bit addressing. Various types of memory, such as RAM and Flash, are mapped to different regions within this same addressing space, eliminating the need for a cache in many cases. However, some systems may incorporate a small unified cache or utilize a split-cache architecture to enhance performance.

In scenarios where the processor architecture imposes limitations on the addressing space, banked memory becomes essential. This is particularly relevant when the instruction set operates on 8- or 16-bit operands. In such cases, there is a clear distinction between local and global memory addresses. Local addresses target a small subset of the available memory, whereas global addresses are designed to access the entire memory space.

However, global addresses often necessitate a wider address bit-width than what the core registers can accommodate. To address this limitation, specialized segment registers are employed to extend the address bit-width, facilitating access to a larger memory range.

2.5.3 General Purpose Input Output

General Purpose Input/Output (GPIO) pins are versatile elements in embedded systems, each capable of serving multiple functions beyond basic input and output. Typically, GPIOs are organized into groups known as ports, allowing all pins within a port to be sampled simultaneously, thereby improving efficiency in data handling. GPIO pins primarily serve four main

functions: digital input, digital output, analog input, and analog output. To manage these functionalities effectively, several registers are utilized:

- *Data Direction Register* (DDR): this register defines the direction of each pin, determining whether it functions as an input or an output.
- *Port Data Register* (PDR): this register holds the data for the pin when it is used in digital mode, enabling the system to read or write values as needed.
- *Port Pin State Register* (PPSR): reflecting the current state of the port pins, this register allows for real-time monitoring of pin statuses.
- *Edge Port Control Registers* (EPCR): these registers are essential for configuring pins for various detection modes, including level detection and detection of rising or falling edges. They also enable or disable interrupts related to specific pins and manage internal pull-up or pull-down resistors.
- *Edge Port Status Register* (EPSR): this register maintains status flags associated with pins that have interrupt capabilities, providing important information about the state of these pins.

By utilizing these registers, GPIO pins can be effectively managed and configured, enabling a wide range of applications in embedded systems.

2.5.4 Analog comparators

Analog comparators are specialized differential amplifiers designed to compare two input voltages, saturating the output to either the supply voltage (Vdd) or ground (Vss). They feature two analog inputs, designated as A and B, and provide a single digital output, Y.

The behavior of the analog comparator is straightforward: when the voltage at input A exceeds the voltage at input B, the output Y generates a logic 1. Conversely, if the voltage at A is less than or equal to that at B, the output results in a logic 0. The polarity of this behavior can be inverted through polarity selection, allowing for flexible application in various scenarios.

Analog comparators can deliver output in several ways:

- *Polling output*: in this mode, the software periodically reads the result of the comparison. Based on the comparison value, the software can perform slow operations, making it suitable for applications that do not require immediate response.
- *Interrupt output*: here, the comparator's output is connected to an interrupt controller. When a comparison event occurs, an interrupt service routine (ISR) is executed, allowing the system to react swiftly to rare events. This configuration is advantageous for time-sensitive applications.
- *Hardware output*: in this setup, the output of the comparator is fed directly to an output pin, bypassing software intervention entirely. This approach effectively squares the input waveform, ensuring rapid response without the latency of software processing.
- *Hardware count or capture output*: the output can also be connected to a control input of a timer. In count mode, every change in the output generates a front that is counted, while in capture mode, the system measures the time interval between two fronts. This functionality is beneficial for precise timing applications.

Through these various configurations, analog comparators provide versatile solutions for comparing voltages and responding to different input conditions in embedded systems.

2.5.5 Analog to digital converter

An Analog to Digital Converter (ADC) is a critical component that encodes an input voltage signal into a numeric value, enabling digital processing of analog signals. The operation of an ADC involves several key processes: sampling, quantization, accurate timing, stable power supply considerations, and input multiplexing.

Sampling is the first step, where the input signal, which varies continuously over time, is observed at specific moments. This observation occurs based on a periodic time base, allowing the ADC to capture representative values of the input signal at defined intervals.

Once the signal is sampled, quantization takes place. Here, the ADC rounds the real-valued physical quantity of the input signal to fixed discrete intervals, which are determined by the power supply voltage. This process converts the continuous signal into a finite set of values, facilitating digital representation.

Accurate timing is essential for the proper functioning of an ADC. This is typically achieved through the use of an external crystal oscillator connected to the main clock source, ensuring that sampling occurs at precisely defined intervals.

The ADC's performance can be affected by variations in the stable power supply. If the power supply of the microcontroller—and consequently that of the ADC—changes while the input voltage remains stable, it can manifest as an apparent change in the input voltage. To mitigate this issue, ratiometric measurement techniques are employed, which require a fixed absolute voltage reference, either internal or external, to maintain accurate readings.

Input multiplexing is another important feature of ADCs, allowing multiple analog inputs to be connected and converted in a cyclic manner. Through time multiplexing, the ADC can sample and convert several inputs sequentially, increasing flexibility and efficiency in systems that require monitoring of multiple signals.

In summary, ADCs play a vital role in bridging the gap between the analog and digital worlds, enabling accurate and reliable measurement of real-world signals for various applications.

CHAPTER 3

Timers

3.1 Phase-Locked Loop

A Phase-Locked Loop (PLL) is a fundamental electronic circuit used in various communication and signal processing applications. It operates by synchronizing the frequency of a Voltage-Controlled Oscillator (VCO) with that of an input signal. The key components of a PLL include a phase detector, a low-pass filter, and the VCO, all connected in a feedback loop. The primary function of the PLL is to keep the VCO frequency "locked" to the frequency of the input signal, ensuring they remain in sync.

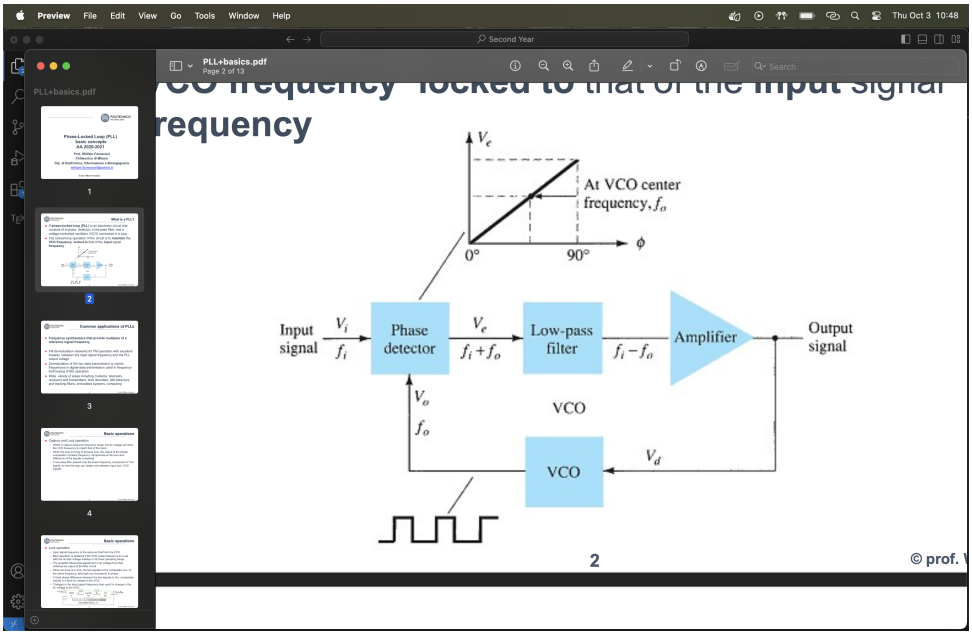


Figure 3.1: Phase-Locked Lopp

3.1.1 Locking mechanism

The fundamental goal of the PLL is to align the VCO's output frequency with the input signal frequency. The best performance is achieved when the VCO's center frequency, f_o, is set within

the middle of its linear range using a DC bias voltage. This allows the circuit to maintain control over the VCO's frequency with maximum efficiency.

In normal operation, when the loop is locked, the input and VCO output frequencies are identical. However, they may not necessarily be in phase. The phase detector generates a fixed phase difference between these signals, which is converted into a DC control voltage applied to the VCO. As the input signal frequency fluctuates, the phase detector adjusts this DC voltage to maintain synchronization.

Due to the limited operating range of the VCO and the feedback nature of the PLL, two key frequency ranges are critical for proper operation:

- *Capture range*: the frequency range around the VCO's free-running frequency, f_o, within which the PLL can initially acquire lock with the input signal.
- *Lock range*: once the PLL is locked, it can maintain synchronization with the input signal over a wider frequency range than the capture range. This lock range ensures the PLL can handle variations in the input signal while maintaining phase coherence.

Phase-Locked Loops (PLLs) are essential components in modern electronics due to their versatile functionality. Here are some of the key applications:

3.1.2 Applications

The main applications of a PLL include:

- *Clock multiplier or clock generator*: generates a clock signal that is a multiple of the input frequency or provides multiple clock outputs.
- *Frequency synthesizer (fractional-N, integer-N)*: produces a clock signal with an arbitrary frequency by dividing or multiplying a reference clock.
- *Clock and data recovery*: recovers digital data and a synchronized clock signal from a serial data stream, using a specialized phase detector.
- *FM demodulation*: demodulates a radio signal by tracking the frequency modulation and converting it into a usable signal.

Frequency syntesis In frequency synthesis applications, a frequency divider is placed between the VCO output and the phase comparator. The output frequency, f_o, is divided down before being fed back to the phase detector. As long as the loop remains locked, the VCO output will be an integer multiple of the input frequency. This principle is widely used in communication systems to generate precise clock signals.

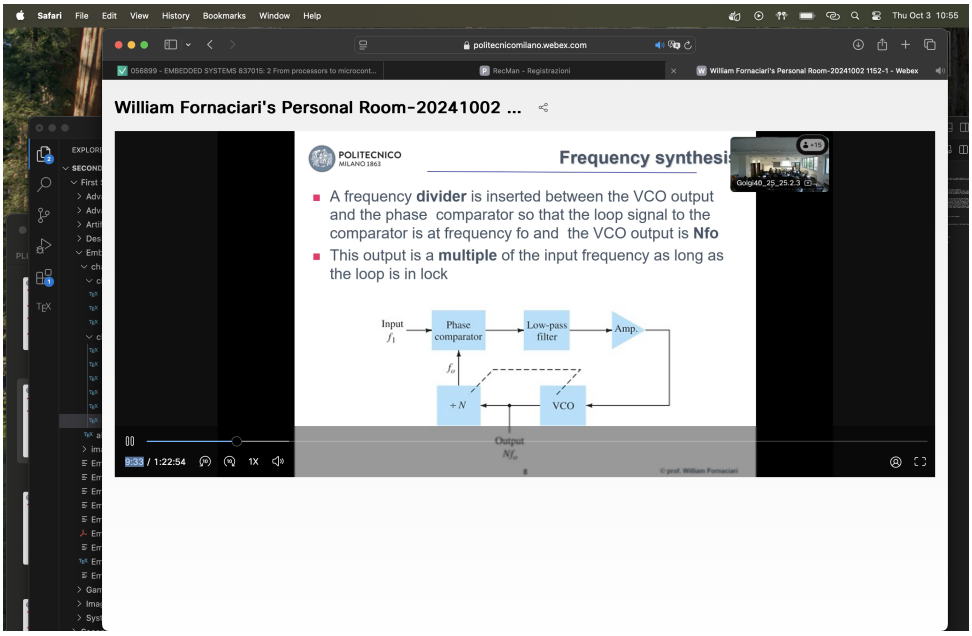


Figure 3.2: Frequency synthesis

Integer-N synthesizer The resolution of the output frequency in an Integer-N synthesizer is determined by the reference frequency applied to the phase detector. The step size or frequency resolution represents the smallest frequency increment the system can generate. This approach simplifies creating low-frequency sources, as producing a stable frequency in the kilohertz (kHz) range directly from a crystal oscillator can be impractical due to its large size.

A more practical solution is to use a stable, high-frequency crystal oscillator and divide its output using an Integer-N synthesizer. The output frequency f_o can be expressed as:

$$f_o = f_{\text{ref}} \frac{n}{r}$$

Here, f_{ref} is the reference frequency (often derived from a crystal oscillator), n and r are programmable integer values that can be selected based on system requirements.

By adjusting N and R, the synthesizer can generate multiple clock frequencies from a single crystal oscillator, allowing for efficient clock domain creation and variable-speed operation in computing systems.

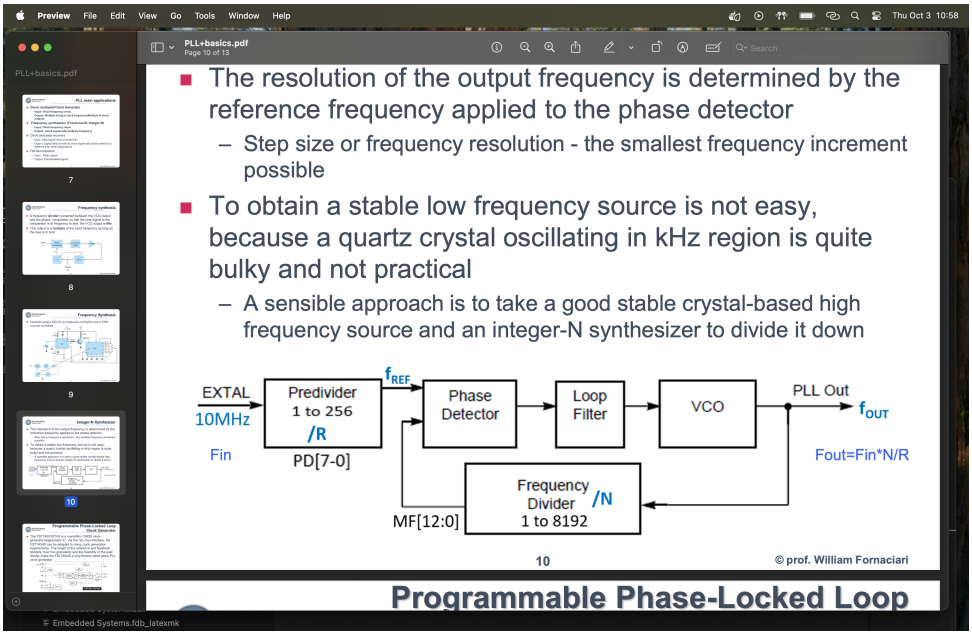


Figure 3.3: Integer-N synthesis

Integer-N synthesizer with prescalers To extend the frequency range of a synthesizer while still allowing low-frequency synthesis, prescalers are introduced. Prescalers are frequency dividers that reduce the frequency of the VCO before it reaches the phase detector. This technique is essential in systems, such as microcontrollers, where the clock frequency is fixed at design time, but dynamic frequency scaling is needed to save power or manage thermal performance.

A four-modulus prescaler is a logical extension of the traditional dual-modulus prescaler, offering more flexibility. It provides four scaling factors and uses two control signals to select one of the available factors. This allows the synthesizer to operate over a wider range of frequencies, covering both high and low-frequency requirements.

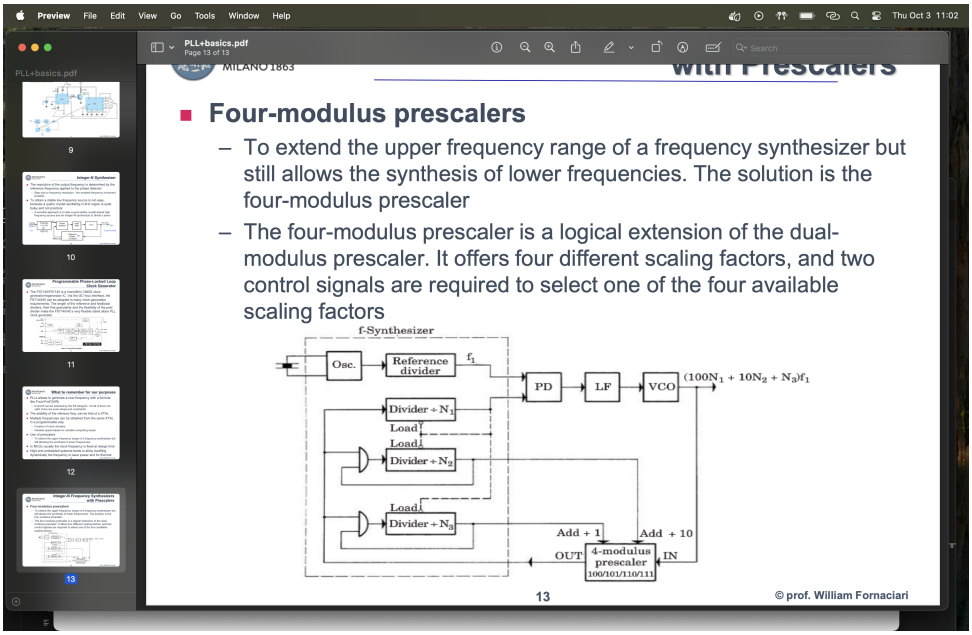
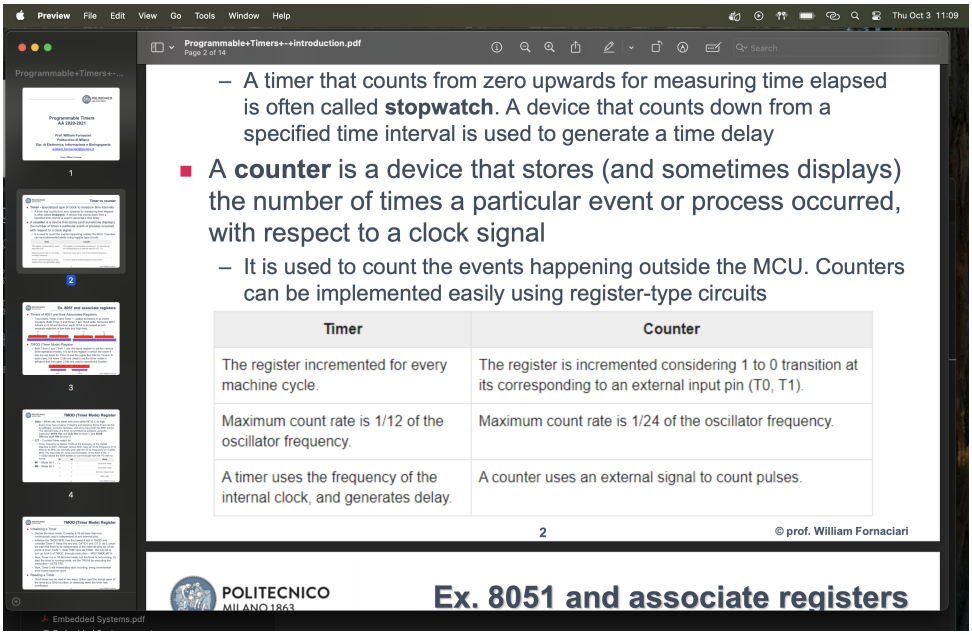


Figure 3.4: Integer-N synthesis with prescalers

In high-end embedded systems, such a design allows dynamic adjustment of clock speeds to optimize performance and power consumption. By combining prescalers with PLLs, systems can adapt to variable computational loads while maintaining efficient power use.

3.2 Programmable timers

A programmable timer is a specialized clock used to measure time intervals or count events, and it is a fundamental component in many embedded systems. Timers can operate in different modes, either counting upwards to measure elapsed time (like a stopwatch) or counting downwards to generate a delay for timing purposes. Additionally, timers often incorporate counters, which track and record the number of events triggered by external signals.



Counters Counters are similar to timers but are designed to count external events (e.g., signal pulses) instead of clock cycles. They are useful in tracking occurrences of specific events such as input signals, sensor triggers, or external interrupts.

In microcontroller-based systems, timers and counters are widely used for generating time delays, event counting, and more. Counters are implemented using register-based circuits, which can store and display the number of events or clock cycles that have occurred.

3.2.1 Operations and settings

Timer Mode registers Timers are controlled through Special Function Registers (SFRs) that define how the timer operates. The configuration options typically include:

- *Gate*: when set, the timer only runs while an external interrupt is active (high). Otherwise, the timer operates continuously.
- *Start and stop control*: timers can be started and stopped via software or hardware control.

- *Counter or timer select bit* (C/T): this bit determines whether the module operates as a timer (incrementing with the internal clock) or a counter (incrementing based on external events).

Initialization To initialize a timer, the configuration process typically involves setting the appropriate bits in the timer's control register. Below is an example for initializing a 16-bit timer that operates continuously without external pin dependencies:

1. Select timer mode: we will configure Timer 0 to operate in 16-bit mode, which allows the timer to count from 0 to 65535 before overflowing.
2. Initialize the TMOD register: set the timer mode by modifying the lower 4 bits of the TMOD register.
3. Execute the instruction `MOV TMOD, #01h` to configure Timer 0.
4. Start the Timer: after configuring the timer, start it by setting the `TR0` bit, which starts the timer incrementing with every machine cycle. Execute `SETB TR0` to start Timer 0.

Now, Timer 0 will begin counting, incrementing once every machine cycle, and will continue until it overflows or is manually stopped.

Reading There are two primary ways to read the value of a 16-bit timer:

- *Reading the timer value*: you can read the actual count value stored in the timer registers. This provides the current count, allowing you to measure elapsed time or events.
- *Detecting timer overflow*: alternatively, you can monitor the timer overflow flag, which is set when the timer reaches its maximum value (e.g., 65535 in a 16-bit timer). Once overflow occurs, the timer resets and starts counting from zero again. This is useful for generating interrupts or periodic signals.

By configuring the timer to trigger an interrupt on overflow, you can create precise, repeatable time delays or event-driven processes in your system.

Settings Timers can be operated either by polling or by using interrupts:

- *Polling*: this involves continuously reading the status registers to check for timer events or the current counter value. However, polling can consume significant processing time and may introduce variability in response times, especially in complex programs.
- *Interrupts*: a more efficient approach is to configure the timer to generate an interrupt when a specific event occurs. When the event occurs, the timer triggers a hardware signal that is sent to the microcontroller's interrupt controller, which suspends the main program and jumps to an Interrupt Service Routine (ISR). After executing the ISR, the program returns to its main loop. This method ensures fast and predictable responses to timer events, without requiring the main program to check for them continuously.

3.2.2 Structure

The basic structure of a timer in an microcontroller consists of several essential components:

- Clock source: every timer requires a clock to measure time intervals. Multiple clock sources may be available, with the possibility of selecting an external clock if needed.
- Prescaler: to extend the count range, the clock signal is passed through a prescaler, which divides the input clock by a factor (often a power of 2). Some prescalers allow division by factors as large as 2^{16} (up to 65536).
- Main counter: after the prescaler, the clock feeds into a main counter. Typically 16 bits wide, this counter can count up to 65,535. It increments or decrements depending on the timer mode.
- Modulus Value (M): the main counter's range is controlled by a modulus value, which can be programmed into a register. When the counter reaches zero, it reloads this value and continues counting. This configuration allows for generating periodic signals or interrupts.
- Control logic: the control logic defines the operational mode of the timer. This includes configuring the clock source, prescaler, modulus value, and other control bits. The control registers vary across different microcontrollers, providing flexibility to adapt the timer to different applications.

Typical initial configurations for an microcontroller timer include:

- Selecting the appropriate clock source.
- Setting a prescaler value.
- Programming the modulus value.
- Configuring control bits to define the timer mode.
- Enabling interrupts (if required).
- Setting up peripheral triggers, such as Direct Memory Access (DMA) or other microcontroller peripherals.
- Configuring input/output pin connections if the timer is used for signal generation or external event detection.

3.2.3 Periodic timers

Periodic timers are used to generate repetitive events or ticks with a fixed period. The key parameter in such applications is the period, which is determined by the modulus value programmed into the timer. Periodic timers are commonly used for:

- Generating the baud rate clock for serial communication.
- Polling digital inputs, such as pushbuttons, at regular intervals.
- Scheduling tasks in real-time operating systems (RTOS) based on precise time intervals.
- Pacing DMA transfers to peripherals such as Digital-to-Analog Converters (DACs).

- Triggering Analog-to-Digital Converters (ADCs) to ensure accurate sample rates.

Periodic timers provide a highly reliable mechanism for scheduling and synchronization in embedded systems, ensuring that events occur at precisely controlled intervals.

3.2.4 Delay functions

Delay timers are used to execute actions after a specific time interval has passed. This function works by resetting the timer to zero (event A) and waiting for a defined number of ticks before triggering the desired action (event B). Delays are essential for:

- Implementing debounce mechanisms for pushbuttons.
- Waiting for peripherals to complete their operations.
- Introducing pauses in the operation of mechanical systems or communication protocols.

By using programmable timers in microcontroller design, developers can achieve highly precise control over time-sensitive operations, ensuring that embedded systems can meet real-time constraints and efficiently manage resources.

3.2.5 Microcontrollers design

Timers and counters are some of the most critical peripherals in microcontroller designs, enabling a wide variety of functions that improve performance, reduce power consumption, and simplify designs. They can offload repetitive tasks from the CPU by handling timing-based operations through interrupts, freeing the processor to focus on more complex tasks.

- Timers and counters can be used in virtually any application to enhance performance, reduce power usage, or streamline design by replacing CPU-based operations with interrupt-driven tasks.
- Many Commercial Off-The-Shelf (COTS) microcontroller devices include built-in support for programmable timers, enabling easy integration into designs.
- Some of the most sophisticated uses of timers are in Pulse Width Modulation (PWM) applications, such as motor control. Here, hardware handles most of the PWM functions, minimizing processor involvement in low-level operations.
- Manufacturers provide development kits and reference designs tailored to specific applications that rely heavily on programmable timers.

3.3 Watchdogs

A watchdog timer is basically an internal or external device (timer) that is installed in a system in order to detect any software anomalies that may arise in embedded systems. It has the responsibility to reset and restart the processor when needed in the case of a software glitch. It is also called Computer Operating Properly (COP).

With billions of IoT devices being deployed in the field, it would be impossible for a technician to service them in a timely manner if something goes wrong – IoT systems must be able to detect and recover from faults on their own without any human intervention Watchdogs

come in many different shapes and sizes, but can generally be categorized as simple timers, windowed timers and smart watchdogs – Watchdogs may exist internally to microcontrollers as hardware and software, externally as hardware, and even as separate microcontrollers with both hardware and software components – No matter which watchdog solution is used, the sole purpose is to monitor and recover the system. To this end, each watchdog has its own unique characteristics and design challenges that developers need to consider for a robust IoT system

3.3.1 Workflow

Normally the computer regularly resets the watchdog timer to prevent it from elapsing, or "timing out" In the case of hw fault or program error, the computer fails to reset the watchdog, the timer will elapse and generate a timeout signal The timeout signal initiates corrective actions (not always RESET!!!) The corrective actions typically include placing the computer system in a safe state and restoring normal system operation

3.3.2 Architecture

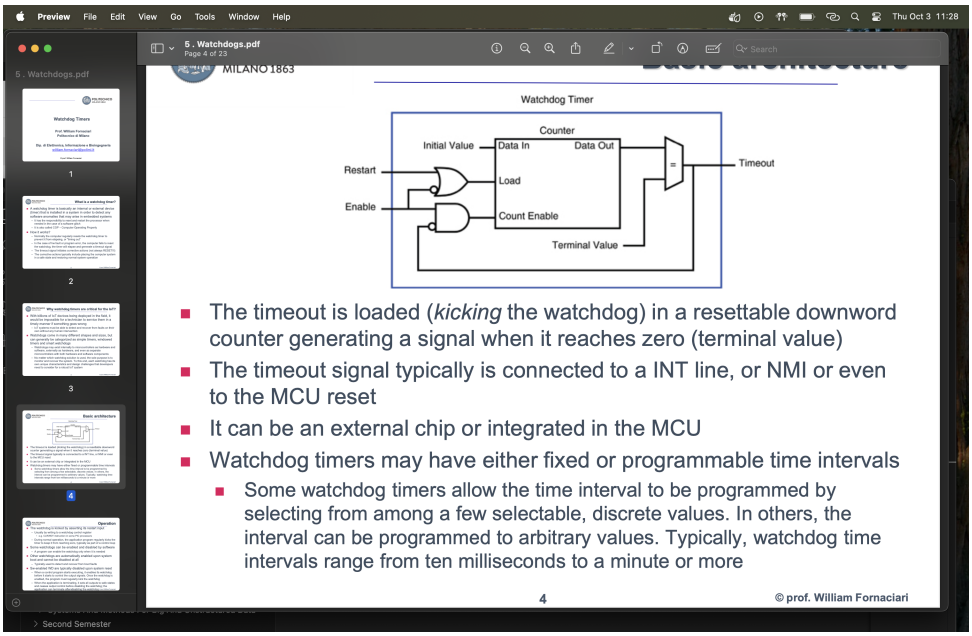


Figure 3.5: Watchdogs architecture

The timeout is loaded (kicking the watchdog) in a resettable downword counter generating a signal when it reaches zero (terminal value) The timeout signal typically is connected to a INT line, or NMI or even to the MCU reset It can be an external chip or integrated in the MCU Watchdog timers may have either fixed or programmable time intervals Some watchdog timers allow the time interval to be programmed by selecting from among a few selectable, discrete values. In others, the interval can be programmed to arbitrary values. Typically, watchdog time intervals range from ten milliseconds to a minute or more

The watchdog is kicked by asserting its restart input – Usually by writing to a watchdog control register • e.g. CLRWDT instruction in some PIC processors – During normal operation, the application program regularly kicks the timer to keep it from reaching zero, typically as part of a control loop Some watchdogs can be enabled and disabled by software – A program

can enable the watchdog only when it is needed Other watchdogs are automatically enabled upon system boot and cannot be disabled at all – Typically used to detect and recover from boot faults Sw-enabled WD are typically disabled upon system reset – When a control program starts executing, it enables its watchdog before it starts to control the output signals. Once the watchdog is enabled, the program must regularly kick the watchdog – When the application is terminating, it sets all outputs to safe states and ceases output control before disabling the watchdog; the application can terminate after disabling the watchdog

In case of problems Two corrective actions exists – (1) Set of the MCU control outputs to safe levels so that potentially dangerous devices such as motors and heaters will not pose threats to people or equipment. High priority action that must occur as soon as a fault is detected – (2) After setting the outputs to safe levels restore normal system operation • This can be as simple as restarting the computer, as if a human operator has pressed the computer's reset pushbutton, or it may involve a sequence of actions that ultimately ends with a computer restart • A watchdog timer can respond to faults more quickly than a human operator, making it invaluable in cases where a human operator would be too slow to react to a fault condition • Watchdog timers may also be used when running untrusted code in a sandbox, to limit the CPU time available to the code and thus prevent some types of denial-of-service (DoS) attacks

3.3.3 Single stage watchdog

Single timer that invokes an immediate restart upon timeout – The timeout signal is connected to the computer's system reset input, either directly or through a conditioning circuit, so that a computer restart will occur when the watchdog times out – This architecture depends on the system reset to force control outputs to their safe states • Some computers will power-down if a continuous timeout signal is applied to the system reset input. In such cases, a pulse may be required to initiate a system restart. A pulse generator can often be used to satisfy this requirement • MCU and watchdog may share the same clock signal

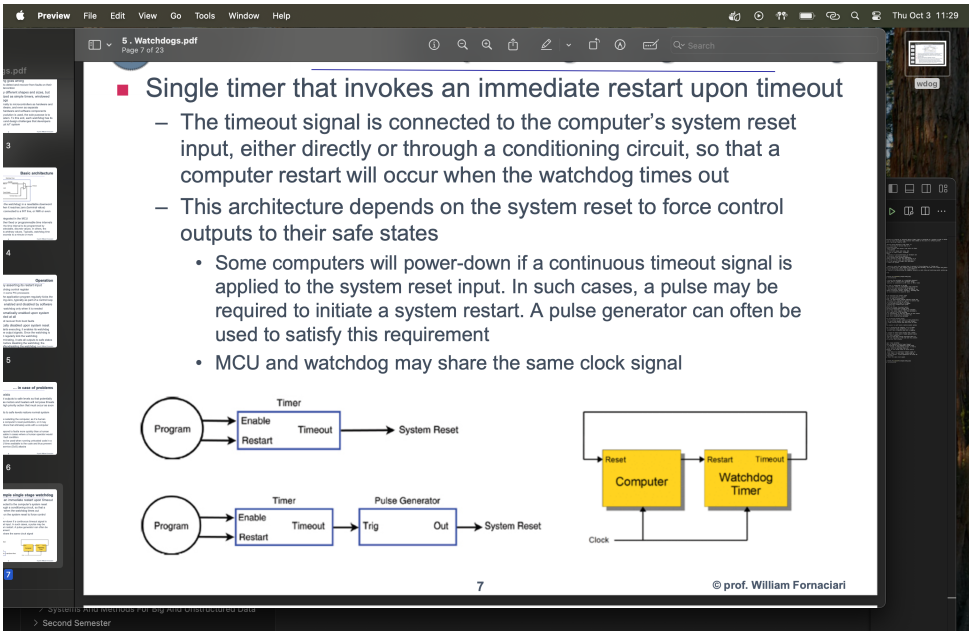


Figure 3.6: Single stage watchdog

3.3.4 Multiple stage watchdog

Two or more timers cascaded to form a multistage watchdog timer. Only the first stage is kicked by the processor – Upon first stage timeout, a corrective action is initiated and the next stage in the cascade is started. As each subsequent stage times out, it triggers a corrective action and starts the next stage – Upon final stage timeout, a corrective action is initiated, but no other stage is started because the end of the cascade has been reached – Typically, single-stage watchdog timers are used to simply restart the computer, whereas multistage watchdog timers will sequentially trigger a series of corrective actions, with the final stage triggering a computer restart

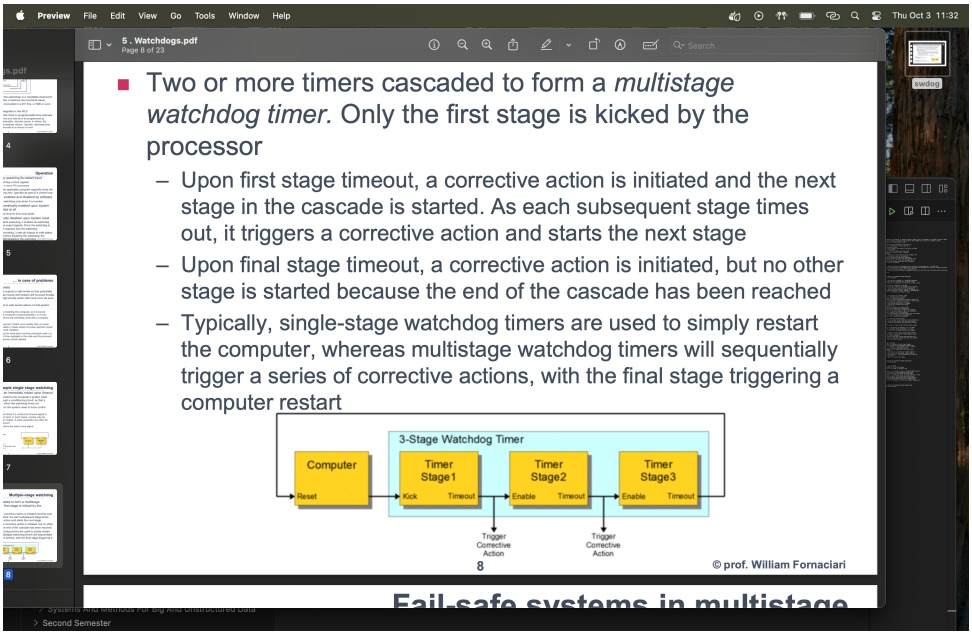


Figure 3.7: Multiple stage watchdog

Multistage watchdog timeout doesn't immediately restart MCU; it merely schedules a restart to occur at a future time A multistage watchdog must work in concert with special circuitry that will switch outputs to safe states upon timeout, prior to the computer/MCU restart One way to do this is to employ a dedicated control reset signal that resets the control circuitry (but not the computer) upon watchdog timeout. This is easily implemented but it presents some complications and shortcomings: Also, a reset can cause the loss of important state information needed for fault recovery, and it may interfere with the operation of interfaces that could otherwise continue to function normally during a fault condition

Run mode and Safe mode The program can modify Runmode states at any time, but it can change Safemode states only when permitted by a special write- protect mechanism. Typically, the program will begin to control the Runmode states after it establishes Safemode states, which comprise a complete, customized set of safe states for all outputs – During normal operation, the Runmode states are routed through the data selector to the outputs. Upon watchdog timeout, the data selector switches input sets so that Safemode states are applied to the outputs in place of Runmode states. Since the control circuitry has not been reset, it will continue to function normally (to the extent possible) and it will retain interface state information (e.g., incremental encoder counts, captured events, hardware configuration) that may be needed for fault recovery

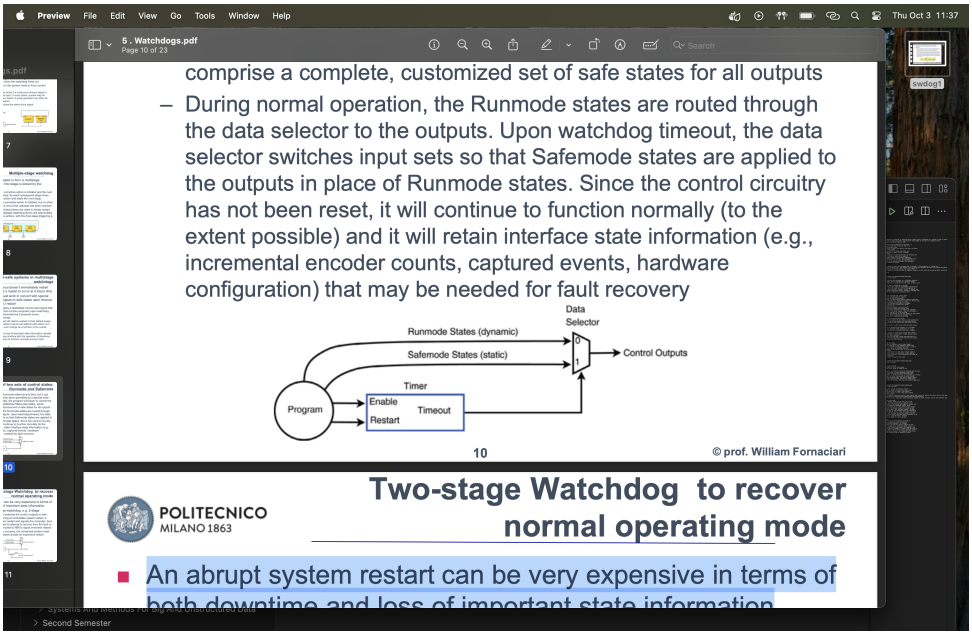


Figure 3.8: Watchdog mode

Recovering An abrupt system restart can be very expensive in terms of both downtime and loss of important state information Mitigated with a multistage watchdog, e.g. 2-stage – The watchdog immediately switches the control outputs to safe states, but instead of triggering an immediate system restart, it schedules a deferred system restart and signals the computer, thus allowing time for the program to attempt to recover from the fault or log state information (IRQ routed to NMI to signal imminent restart) – If the program successfully recovers, the scheduled system reset will be canceled and the system avoids an expensive restart

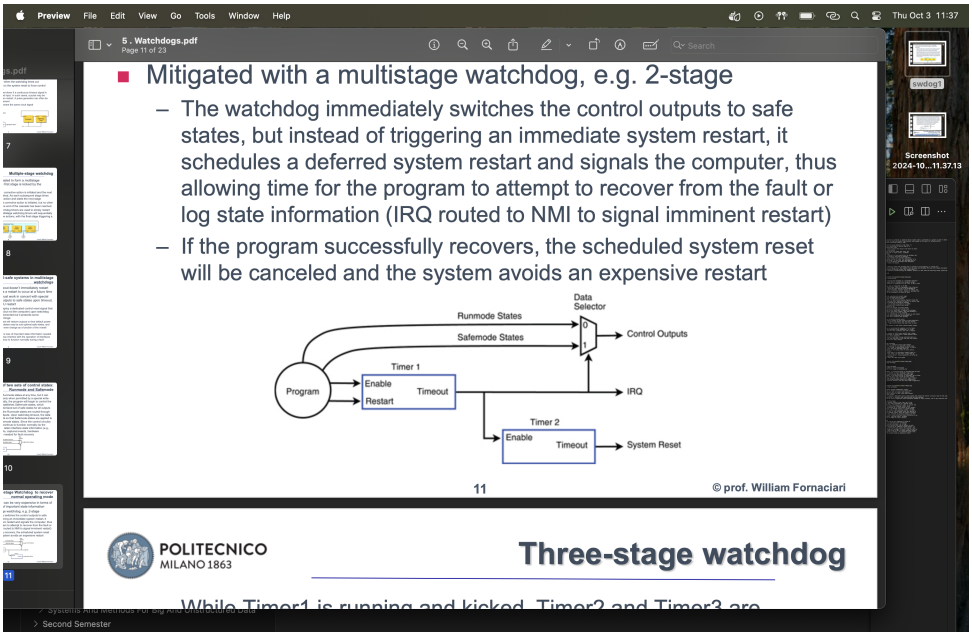


Figure 3.9: Watchdog recovery

3.3.5 Three-stage watchdog

While Timer1 is running and kicked, Timer2 and Timer3 are disabled and held at their initial values and the control outputs are allowed to change under program control – Timer1 timeout will switch the outputs to safe states, start Timer2, and requests interrupt service. If the computer is able to respond to the IRQ, the program will attempt to recover from the fault condition and, if successful, the program will disable Timer1 (and by extension, Timer2), thus canceling further corrective actions – If the computer cannot respond to the IRQ • Timer2 will timeout, start Timer3 and assert a NMI to indicate imminent system restart • Make possible logging important fault information (e.g., crash dump) before restart

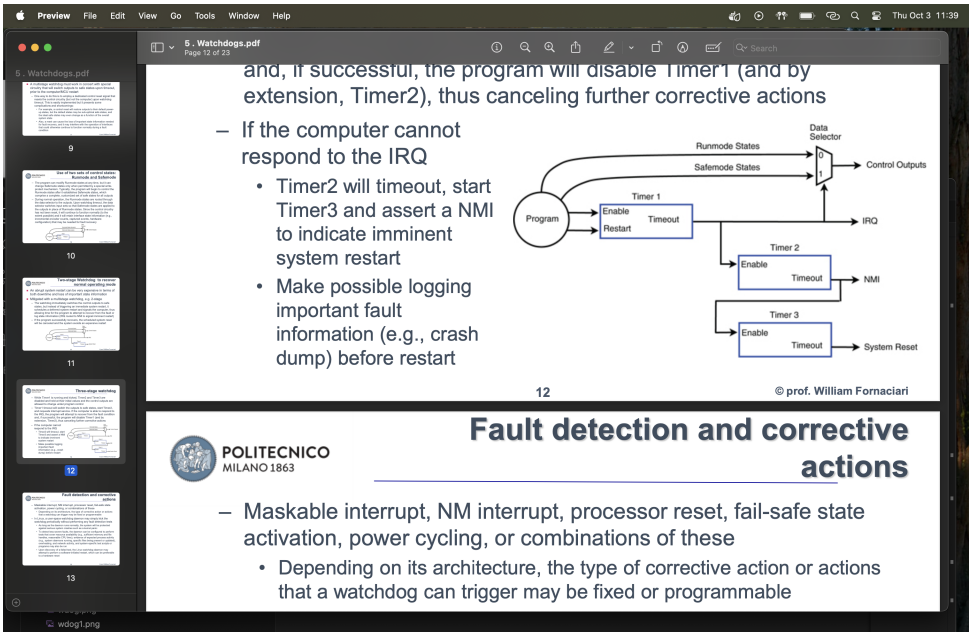


Figure 3.10: Three-stage watchdog

Fault detection Maskable interrupt, NM interrupt, processor reset, fail-safe state activation, power cycling, or combinations of these • Depending on its architecture, the type of corrective action or actions that a watchdog can trigger may be fixed or programmable – In Linux, a user-space watchdog daemon may simply kick the watchdog periodically without performing any fault detection tests • As long as the daemon runs normally, the system will be protected against serious system crashes such as a kernel panic • To detect less severe faults, the daemon can be configured to perform tests that cover resource availability (e.g., sufficient memory and file handles, reasonable CPU time), evidence of expected process activity (e.g., system daemons running, specific files being present or updated), overheating, and network activity, and system-specific test scripts or programs may also be run • Upon discovery of a failed test, the Linux watchdog daemon may attempt to perform a software-initiated restart, which can be preferable to a hardware reset

3.3.6 AVR watchdog

AVR® devices have an Enhanced Watchdog Timer (WDT) that runs on a separate oscillator from the main clock – WDT is essentially a counter that increments based on the clock cycles of an on-chip 128kHz oscillator – The WDT forces an interrupt or a system reset when the counter reaches a given time-out value – Selectable Time-out period from 16 milliseconds to

8 seconds In normal operation mode, the application code needs to issue a Watchdog Timer Reset (WDR) instruction to restart the counter before the time-out value is reached. If the system doesn't restart the counter, an interrupt or system reset will be issued Three operating modes – Interrupt Mode • WDT forces an interrupt when the timer expires. This interrupt can be used to wake the device from any of the sleep-modes, and also as a general system timer. One example is to limit the maximum time allowed for certain operations, forcing an interrupt when the operation has run longer than expected. This is enabled by setting the Interrupt mode bit (WDIE) in the Watchdog Timer Control Register (WDTCSR) – System Reset • WDT forces a reset when the timer expires. This is typically used to prevent system hang-up in the case of runaway code. This is enabled by setting the System Reset mode bit (WDE) in the Watchdog Timer Control Register (WDTCSR) – Interrupt and System Reset Mode • Interrupt and System Reset mode combines the other two modes by first forcing an interrupt and then switching to the System Reset mode. This mode will offer a safe shutdown by allowing time to save critical parameters before a system reset. This is enabled when both the WDTIE and WDTE are set

3.3.7 Watchdogs usage

– Never disable the watchdog for any reason – Never clear the watchdog in a periodic interrupt independent from software functionality checks – Verify that the watchdog timer is an independent watchdog. Independent watchdogs have a separate clock that allows them to detect if the system clock has halted – Use a watchdog that has a windowed watchdog feature • These watchdogs require a minimum time before the watchdog can be cleared. If an attempt is made prior to the start of the window, the watchdog will reset the system. This prevents runaway software from overriding the watchdog timer – Internal WDT are a good step towards building a robust embedded system, but on their own they don't provide a very robust solution • In order to really up the ante with respect to robustness, developers need to consider external watchdogs

3.3.8 External watchdogs

Many internal implementations have flaws – Examples are sharing the system clock, and having a disable option External watchdog has many advantages, such as – Performing a hard system reset that ensures the microcontroller is power cycled, which in turn power cycles the internal peripherals – Separating the watchdog from the microcontrollers oscillator circuit – Providing a completely independent process for monitoring the system All of these contribute to system robustness, although there are also a few disadvantages to using an external WDT – Increase in hardware costs due to the addition of an IC as well as an increase in system complexity

3.3.9 Smart watchdogs

A smart watchdog is a supervisory microcontroller that, in addition to performing basic heartbeat monitoring, can also monitor system communications – There can be instances where the microcontroller stops responding to the Internet, but is still successfully clearing the external watchdog – When this happens, a command could be sent over the Internet to reset the microcontroller – The smart watchdog can monitor the communication lines, such as UART transmit and receive lines, for a special command that tells it to restart the system – There are several mic