COMPSYS 303: Microcomputers and Embedded Systems

Lecture Notes

Nicholas Russell

Saturday, October 5, 2024

Table of contents

Introduction

These notes provide a detailed breakdown of the lectures for COMPSYS 303. Each lecture is represented in its own document, allowing for modular learning and easy reference.

Lectures Included

1. **Lecture 1: Concurrency**
   - Overview of concurrency in embedded systems, challenges like race conditions and priority inversion, and real-world examples.
2. **Lecture 2: SCCharts and Synchronous Concurrency**
   - Detailed discussion on SCCharts, the synchronous approach, and cyber-physical systems.
3. **Lecture 3: Peripherals and Interfacing**
   - Types of I/O, data transfer mechanisms, and interrupt handling in embedded systems.
4. **Lecture 4: Embedded Processors**
   - Overview of processor design in embedded systems, FSMD, types of processors, and synthesis from C code.
5. **Lecture 5: SoC and SOPC Buses**
   - Discussion of bus design issues, synchronous vs. asynchronous buses, and bus arbitration.
6. **Lecture 6: Embedded Control Systems 1**
   - Real-time systems, sampling and quantization, aliasing, and worst-case execution time.
7. **Lecture 7: Embedded Control Systems 2**
   - Interaction with real-world peripherals, including PWM, stepper motors, and ADC.
8. **Lecture 8: Embedded Control Systems 3**

- Control of physical processes, key metrics in control systems, and examples like cruise control.

9. **Lecture 9: Industrial Automation 1**
    - Introduction to industrial automation, PLCs, HMIs, and generations of automation.

10. **Lecture 10: Industrial Automation 2**
    - IEC 61131 standard, ladder logic features, and an example of a pneumatic cylinder control sequence.

Part I

Partha's Lectures

1 Lecture 1: Concurrency

1.1 Introduction to Embedded Systems and Concurrency

Embedded systems are a specialized class of computers designed to continuously interact with their environment in a real-time, often reactive manner. Unlike general-purpose computers, embedded systems are purpose-built to handle specific tasks under stringent conditions, including timing constraints. These systems are embedded as integral parts of larger devices and must operate dependably, often without human intervention. Examples include automotive control systems, medical devices like pacemakers, and consumer electronics such as washing machines.

When deadlines must be met without exception, the system is termed a **hard real-time system**; otherwise, it is a **soft real-time system**. In hard real-time systems, any missed deadline could result in catastrophic failure, such as in medical devices or automotive safety features. In contrast, soft real-time systems can tolerate occasional missed deadlines, although they may affect performance.

The concept of **concurrency** is crucial in embedded systems because these systems often involve multiple processes interacting simultaneously, such as a pacemaker continuously monitoring the heart and delivering pulses when needed. Concurrency enables these systems to effectively manage different tasks that need to be performed either synchronously or asynchronously. For example, while monitoring the heart, a pacemaker must also manage pulse delivery in real-time, demonstrating the need for concurrent tasks.

1.2 Key Concepts in Concurrency

Concurrency refers to multiple components or processes operating at the same time within a system. In embedded systems like pacemakers, various components (controllers, sensors, and timers) work concurrently to ensure precise timing between events. Concurrency becomes especially challenging when different tasks access shared resources simultaneously, potentially leading to issues such as **race conditions**, **deadlocks**, and **livelocks**.

- **Race Conditions:** These occur when two or more processes access shared data concurrently, and the final result depends on the timing of their access. A typical example is the producer-consumer problem, where one process generates data while another consumes it. If these processes do not manage shared data properly, inconsistencies may arise. For instance, if two processes increment a shared counter without proper synchronization, the final value of the counter may be incorrect due to overlapping access. This issue is exacerbated in systems with multiple cores or processors, where different threads can execute simultaneously.

- **Critical Sections and Mutual Exclusion:** A **critical section** is a part of a program that should only be executed by one process at a time to avoid data corruption. Ensuring **mutual exclusion**, where no two processes are in their critical sections simultaneously, is essential for preventing race conditions. Mutual exclusion can be achieved using synchronization mechanisms like **locks**, which ensure that only one process can access the critical section at any given time. Common techniques include **mutexes** and **binary semaphores**, both of which can lock shared resources to ensure orderly access. It is also important to minimize the time a process spends in a critical section to reduce the chances of blocking other processes.
- **Deadlocks and Livelocks: Deadlocks** occur when processes are waiting on each other indefinitely, leading to a situation where no progress can be made. This typically happens in resource allocation scenarios where two or more processes form a circular chain of dependencies. A classic example is two processes holding a resource that the other needs to proceed, resulting in a stalemate. **Livelocks** are similar to deadlocks but differ in that processes keep changing state in response to each other without making any actual progress. Both issues are critical in concurrent systems and need to be addressed through careful design, such as avoiding circular waiting conditions, implementing resource hierarchies, or using **deadlock detection algorithms** that can detect and break deadlocks. Timeout mechanisms can also be used to release resources if a process waits too long.

1.3 Approaches to Manage Concurrency

Concurrency in embedded systems is managed through a combination of hardware and software synchronization mechanisms. Some of the methods discussed include:

- **Busy Waiting and Non-Busy Waiting: Busy waiting** occurs when a process remains in the ready state, actively checking for a condition to be satisfied. This approach is inefficient in terms of CPU utilization, as the CPU is occupied with checking instead of performing useful work. **Non-busy waiting** solutions, such as putting a process in the blocked state while waiting for a resource, are more efficient for embedded systems where CPU resources are limited. Busy waiting is often used in systems where the wait time is expected to be very short, whereas non-busy waiting is preferred for longer wait times to free up CPU resources. In embedded systems, avoiding busy waiting is crucial because the CPU often needs to manage multiple tasks under strict timing constraints.

- **Test and Set Instruction:** The **Test and Set** instruction is an atomic operation used to achieve mutual exclusion by using a global variable to indicate whether a critical section is occupied. By using Test and Set, embedded systems can ensure that processes access shared resources in a predictable and coordinated manner. The atomicity of this instruction prevents race conditions by ensuring that the check-and-set operation is performed without interruption. However, Test and Set can lead to **busy waiting**, which may not be suitable for all embedded applications. In systems with limited processing power or battery life, busy waiting can degrade performance and reduce efficiency.
- **Semaphores and Mutex Locks: Semaphores** are synchronization mechanisms used to manage concurrent access to shared resources. They allow processes to signal and wait for access, ensuring a controlled flow of execution. Semaphores can be **binary** (acting like a mutex) or **counting**, allowing a specified number of processes to access a resource concurrently. For example, counting semaphores are often used in producer-consumer scenarios where multiple producers or consumers need access to a shared buffer. **Mutex locks** are similar but are typically used to protect access to a particular data structure or variable, providing mutual exclusion in critical sections. Mutexes are often simpler to implement but are limited to binary states (locked or unlocked), whereas semaphores offer more flexibility for resource management.
- **Priority Inversion and Solutions: Priority inversion** occurs when a higher-priority task is waiting for a resource held by a lower-priority task, which can lead to missed deadlines in real-time systems. A well-known example is the **Mars Pathfinder mission**, where a low-priority task holding a shared resource caused a higher-priority task to miss its deadline, resulting in system resets. Solutions to priority inversion include **priority inheritance**, where the lower-priority task temporarily inherits the higher priority to complete its task and release the resource. Another approach is **priority ceiling**, where a resource is assigned a priority equal to the highest priority of any task that may lock it, thus preventing priority inversion.

1.4 Examples and Applications

The lecture provided various examples of where concurrency is critical, including the automotive industry (Slide 6) and robotics. For instance, the **2010 Toyota recall** was attributed to software issues in their anti-lock braking systems, highlighting the importance of deterministic behavior in safety-critical systems. This incident shows how improperly managed concurrency can lead to life-threatening failures. Similarly, **Tesla's accidents** were referenced, emphasizing the challenges of building reliable autonomous systems. These examples underscore the need for proper concurrency management to ensure system reliability and safety.

In embedded systems such as a **pacemaker**, concurrency is an intrinsic characteristic. The pacemaker must concurrently monitor the heart and trigger electrical pulses without delay, thus operating as a real-time reactive system. Concurrency, in this context, is about ensuring different components work seamlessly together to meet life-critical deadlines. The pacemaker example illustrates how timing constraints and concurrent interactions must be carefully managed to avoid potentially fatal outcomes. The system must ensure that sensing, decision-making, and actuation are carried out with precise timing to maintain patient safety.

1.5 Problems with Concurrency and Solutions

Concurrency can lead to serious issues if not properly managed.

- **Deadlocks:** Deadlocks occur when concurrent processes are waiting on each other to release resources, resulting in a state where no process can proceed. To prevent deadlocks, proper scheduling mechanisms must be implemented to ensure that resource allocation does not lead to circular waiting. **Deadlock prevention techniques** include resource ordering, where resources are always requested in a specific order, and **deadlock detection algorithms** that periodically check for cycles in the resource allocation graph. Another approach is **deadlock avoidance**, such as using the **Banker's algorithm**, which allocates resources in a way that ensures a safe state is always maintained.
- **Fairness and Scheduling: Fairness** in resource allocation means that every process eventually gets a chance to execute. Embedded systems often use a **scheduler** to manage processes, ensuring tasks are completed in a timely and fair manner. The role of the scheduler is critical to prevent **starvation**, where some tasks never get executed due to others continuously consuming resources. Common scheduling algorithms include **round-robin**, **priority-based**, and **rate-monotonic scheduling**, each with its own advantages and trade-offs depending on the system requirements. **Round-robin scheduling** is simple

and ensures fairness by giving each process a fixed time slice, while **priority-based scheduling** allows more critical tasks to execute first but can lead to starvation without priority aging.

- **Starvation: Starvation** occurs when a process is perpetually denied the resources it needs to execute, often because other higher-priority processes continuously take precedence. To mitigate starvation, **priority aging** can be used, where the priority of a waiting process is gradually increased over time to ensure it eventually gets the necessary resources. This technique ensures that lower-priority processes are not starved indefinitely, especially in systems with many competing processes.

1.6 Conclusion

Concurrency is an essential aspect of embedded systems, particularly for reactive, real-time applications. This lecture emphasized the importance of understanding the mechanisms—such as **mutual exclusion**, **semaphores**, and **scheduling**—that help manage concurrent processes effectively. For embedded systems like automotive controls, pacemakers, or industrial automation, **deterministic behavior** and avoiding race conditions are paramount. Properly handling concurrency ensures that embedded systems can meet their timing constraints, maintain data consistency, and operate safely under all conditions.

Next steps in the lecture will involve exploring **reactive systems** and how **synchronous concurrency** can be implemented to further ensure the reliability of embedded systems. Understanding these concepts is vital for designing embedded systems that are safe, reliable, and efficient. Future topics will also cover specific synchronization protocols and **real-time operating system (RTOS)** features that support concurrency in embedded environments. Additionally, understanding how to leverage hardware features, such as **interrupt controllers** and **timers**, will be crucial for achieving precise control over concurrent tasks in embedded systems.

2 Lecture 2: SCCharts and Synchronous Concurrency

2.1 Introduction to Synchronous Concurrency and SCCharts

Synchronous Concurrency (SC) is a method used to manage the complexity of Cyber-Physical Systems (CPS) by providing deterministic behavior and efficient handling of concurrent processes. SCCharts are a powerful visual formalism used to model concurrent, reactive, and real-time behaviors in embedded systems. They offer a way to design systems that react to changes in the environment while maintaining reliable and predictable execution.

2.2 The Synchronous Approach

The **synchronous approach** is a paradigm in which all concurrent processes are assumed to execute in lockstep relative to a global logical clock. This provides a deterministic and predictable execution model, which is highly beneficial for embedded systems that operate in safety-critical environments.

Key aspects of the synchronous approach include: - **Synchrony Hypothesis:** This hypothesis assumes that the system's reactions occur infinitely faster than the environment, leading to a **zero-delay model**. This means that the reaction to an input is instantaneous from the system's perspective, ensuring a deterministic response. - **Synchronous Threading and Broadcast Communication:** All concurrent threads execute synchronously, meaning that outputs generated by one component are immediately visible to all other components. This approach resembles **synchronous circuits**, where each clock tick coordinates all activities, providing atomic and instantaneous reactions. - **Atomicity and Instantaneity of Reactions:** The synchronous approach treats reactions as atomic, meaning that no intermediate states are visible during execution. This helps prevent issues like race conditions, which are prevalent in asynchronous systems. - **Reaction to Absence:** The system can react not only to events that occur but also to the absence of expected events, allowing for a more comprehensive model of environmental interaction.

2.3 SCCharts Overview

SCCharts are an extension of Statecharts, introduced by David Harel, that add the benefits of the synchronous paradigm to the powerful modeling features of Statecharts. SCCharts are used to describe complex systems involving multiple states and concurrent processes.

2.3.1 Similarities and Differences Between SCCharts and Statecharts

- **Similarities:** Both SCCharts and Harel's Statecharts share common elements such as states, transitions, signals/events, and modularity. They both support **hierarchy**, **parallelism**, and **broadcast communication** between states, which allows for more compact and expressive models.
- **Differences:** SCCharts operate within a **synchronous framework** and guarantee deterministic behavior, which contrasts with the often non-deterministic nature of Statecharts. SCCharts do not interpret events for simulations, which eliminates hidden behaviors and allows for more precise analysis. SCCharts also do not support **inter-level transitions** and provide deterministic handling of concurrent events, which simplifies debugging and validation.

2.4 Boolean Mealy Machine (BMM)

An SCChart can be represented using a **Boolean Mealy Machine (BMM)**. A BMM is defined as a tuple $M = \langle Q, q_0, I, O, T \rangle$, where: - Q: Set of states. - q_0: Initial state. - I: Set of inputs. - O: Set of outputs. - T: Transition relation, involving states and boolean conditions on inputs.

BMMs provide a formal basis for understanding how SCCharts operate, with transitions being defined based on the current state and inputs. The key properties of BMMs are **determinism** and **reactivity**: - **Determinism**: At most one transition is enabled for any valid combination of inputs from the environment. This ensures that the system behaves predictably regardless of concurrent events. - **Reactivity**: At least one transition is enabled for any valid combination of inputs, ensuring the system can always respond appropriately to environmental changes.

2.5 The VABRO Example

The **Valued ABRO (VABRO)** example, which is an adaptation of Berry's **ABRO**, is used to demonstrate the synchronous programming model. In this example, the system generates an output when two specific inputs have occurred. The model incorporates **strong pre-emption**, allowing the behavior to be reset when a particular condition is met. VABRO demonstrates how SCCharts can handle complex state-based logic, including counting events and generating output values based on conditions.

2.6 Constructiveness in Synchronous Systems

Constructiveness is an important concept in synchronous systems, ensuring that concurrent processes do not interfere with each other's operation. **Berry's Constructiveness** requires that all reads happen only after writes to shared entities are completed, and that there is at most one write to any shared entity during a reaction. This ensures consistency and prevents conflicting updates during a single tick of the logical clock.

Sequential Constructiveness extends Berry's constructiveness by allowing sequential updates to shared variables under certain conditions. The **Init-Update-Read (IRU) protocol** is used to manage variable updates in the following order: 1. **Initialization**: Concurrent threads can make confluent initializations to shared variables, provided they assign the same value. 2. **Update**: Threads update shared variables using the same update function, ensuring confluent updates. 3. **Read**: Finally, threads are allowed to read the updated value, guaranteeing consistency.

2.7 Timed SCCharts and Real-Time Systems

Timed SCCharts add temporal constraints to SCCharts, enabling the modeling of **real-time systems** where timing is crucial. These models can compute the **Worst Case Reaction Time (WCRT)** and adjust the tick length accordingly to ensure the system meets its timing requirements. By counting ticks, timed SCCharts allow real-time systems to be modeled effectively, ensuring that actions are taken within the required time frame.

Dynamic Ticks are also introduced to allow the system to adjust tick lengths dynamically, providing more flexibility in real-time operation. This flexibility is essential for systems that need to adapt to changing conditions, such as varying processing loads or external inputs that influence timing requirements.

2.8 Designing Real-Time Systems with SCCharts

The synchronous approach is particularly suitable for **real-time systems**, where deterministic behavior is essential for safety and reliability. By using SCCharts: - **Concurrency is managed in a deterministic manner**, ensuring that all processes execute predictably without interference or unintended interactions. - **Pre-emption and Suspension** are modeled explicitly, allowing designers to represent high-priority tasks interrupting lower-priority tasks, or suspending ongoing activities until a condition is met. - The **generated code from SCCharts** compiles the concurrent specifications into sequential code, which is efficient and suitable for **bare metal implementations** without requiring an operating system. This makes SCCharts an excellent choice for systems with constrained resources, such as microcontrollers in embedded environments.

2.9 Conclusion

Synchronous concurrency, as implemented using SCCharts, provides a robust framework for designing **deterministic, reactive, and real-time embedded systems**. By combining the power of Statecharts with the guarantees of the synchronous model, SCCharts offer a visual and formal way to specify complex concurrent behavior while ensuring predictable execution. The **synchronous approach** is well-suited for **safety-critical CPS** applications, such as medical devices, automotive systems, and robotics, where the cost of failure is high.

Future discussions will delve deeper into **synchronous pre-emption**, **timed automata**, and how SCCharts can be extended to model more complex temporal behaviors. These tools and concepts are crucial for ensuring that CPS can meet their real-time requirements in a safe and reliable manner.

Part II

Avinash's Lectures

3 Lecture 3: Peripherals and Interfacing

3.1 Overview

- **Input and Output (IO)**: Embedded systems interface with external peripherals to gather input (e.g., sensors) and send output (e.g., motor control).
- **Types of IO**: Can be categorized as *polling*, *interrupt-driven*, or using *Direct Memory Access (DMA)* for efficiency.
  - **Polling**: The processor actively waits until a peripheral is ready. Inefficient as it wastes processing power.
  - **Interrupts**: The peripheral signals the processor when it has data, allowing efficient use of CPU resources.
  - **DMA**: Offloads data transfer to a dedicated controller, enabling parallel processing.

3.2 Memory-Mapped vs Direct IO

- **Memory-Mapped IO**: IO devices are treated like memory locations, accessed using memory read/write instructions.
- **Direct IO**: Uses a separate address space for IO operations, typically with specialized instructions (e.g., IN and OUT instructions in the x86 architecture).

3.3 Data Transfer Mechanisms

- **Programmed IO**: The processor executes specific instructions for data transfer. This is inherently slower and is suitable for simple, low-bandwidth peripherals.
- **DMA Mode**: Allows for efficient data transfer, especially when large data volumes are involved, by using an additional master controller to handle the transfers.

3.4 Interrupts

- **Fixed Interrupt**: The ISR (Interrupt Service Routine) address is fixed and cannot be changed.
- **Vectored Interrupt**: The peripheral provides the ISR address, allowing for more flexible interrupt handling.

- **Interrupt Address Table**: A hybrid approach where an index is provided into a memory table to determine the ISR address.

3.5 Examples of Peripheral Handling

- **Interrupt-Driven IO**: Discussed various types of interrupts, including *maskable* (which can be ignored during critical operations) and *non-maskable* (used for urgent events like power failure).
- **NIOS II Development Board**: The board supports non-vectored interrupts, relying on the HAL system library to determine the interrupt source.

4 Lecture 4: Embedded Processors

4.1 Overview

- **Custom Processor Design**: Embedded systems often require specialized processors designed specifically for the application to ensure optimal performance and efficiency.
- **FSMD (Finite State Machine with Datapath)**: A model that integrates a finite state machine (FSM) with a datapath to execute instructions in an embedded system.
- **Datapath Design**: Consists of elements like registers, ALUs, and multiplexers that perform arithmetic operations and manage data flow.
- **Controller Design**: Controls the sequence of operations performed by the datapath, ensuring synchronization between different components.

4.2 Types of Processors

- **General-Purpose Processors**: Flexible and programmable, used for a wide variety of applications. Example: Pentium processors.
- **Single-Purpose Processors**: Designed to execute one specific program efficiently. Example: Accelerators or coprocessors for specialized tasks.
- **Application-Specific Processors**: A compromise between general-purpose and single-purpose processors, optimized for a particular class of applications to balance flexibility, performance, and power consumption.

4.3 C to Hardware Synthesis

- **Hardware Synthesis from C Code**: Converting high-level C code to hardware involves inferring control and data paths from program logic. This approach allows designers to achieve maximum performance for specific applications without manually coding in HDL.

4.4 FSMD Design

- **State Diagram Templates**: Used to model control flow in the FSMD, involving components like loops, branches, and assignments.

- **Datapath Creation**: Involves creating registers for variables, functional units for operations, and interconnecting them with control signals to create a functional system.
- **Controller FSM**: Works with the datapath to manage control flow and determine which operations are executed at each step.

4.5 Example: Greatest Common Divisor (GCD)

- **GCD Implementation**: Demonstrates how an algorithm (finding the GCD of two numbers) can be converted into an FSMD with a state diagram representing the control flow and a datapath for arithmetic operations.

5 Lecture 5: SoC and SOPC Buses

5.1 Overview

- **System on Chip (SoC) and System on Programmable Chip (SOPC)**: SoCs integrate all components of a computer or other systems onto a single chip, while SOPCs use programmable logic to create flexible embedded solutions.
- **Bus Design**: The bus is the communication system that transfers data between components inside a system, and its design impacts the overall system performance.

5.2 Bus Design Issues

- **Bus Width**: Determines the amount of data that can be transferred in a single cycle. Wider buses improve performance but increase cost.
- **Bus Type**: Can be *dedicated* (separate buses for different types of data) or *multiplexed* (shared bus for multiple purposes), impacting cost and efficiency.
- **Bus Operations**: Include basic operations like read, write, block transfer, and interrupt. Block transfers are particularly useful for filling cache lines.

5.3 Synchronous vs Asynchronous Buses

- **Synchronous Bus**: Operates with a clock signal, making implementation simpler but requiring careful timing of actions to avoid delays.
- **Asynchronous Bus**: Uses handshaking between master and slave components, allowing for flexibility in timing but adding complexity.

5.4 Bus Arbitration

- **Bus Arbitration Mechanisms**: Required when multiple bus masters need to access the bus. Can be *static* (predetermined priority) or *dynamic* (priority assigned based on demand).
  - **Fixed Priority**: Assigns a fixed priority to each master, which can lead to bus hogging by high-priority devices.
  - **Rotating Priority**: Adjusts priorities based on waiting time, ensuring fair access.

  - **Hybrid Policies**: Combine aspects of both fixed and rotating priorities to achieve a balance between fairness and efficiency.

5.5 Examples of Bus Usage

- **Synchronous Bus Operations**: Memory read and write operations are synchronized with the bus clock, with or without wait states to manage slower memory.
- **Dynamic Bus Arbitration**: Allows multiple masters to request access to the bus and ensures that access is granted based on current system needs.

Part III

Nathan's Lectures

6 Lecture 6: Embedded Control Systems 1

6.1 Embedded Control Systems

- **Real-Time Systems**: Industrial embedded systems are real-time in nature and often safety-critical, requiring both functional correctness and timing correctness.
  - **Functional Correctness**: Ensures that the system performs the desired function (e.g., a robotic arm stopping on intrusion).
  - **Timing Correctness**: Ensures that the system operates within specified deadlines (e.g., stopping within 10 ms of intrusion).

6.2 Sampling and Quantization

- **Sampling**: The process of taking snapshots of a signal over time (e.g., video, sound, voltage).
- **Quantization**: Replacing real values with an approximation from a finite set of discrete values. For an n-bit ADC, there are 2^n quantization levels (e.g., an 8-bit ADC has 256 levels).

6.3 Aliasing

- **Aliasing**: Occurs due to insufficient sampling rates, resulting in different signals appearing indistinguishable. To prevent aliasing, signals must be sampled above the Nyquist frequency (at least twice the highest frequency component).

6.4 Computation Delays

- **Sources of Delays**: Include unbounded software (e.g., unpredictable runtime branches, unbounded loops), floating-point operations, search algorithms, and cache misses.
- **Worst-Case Execution Time (WCET)**: The longest time it takes to complete execution. Can be determined through measurement or static analysis, but speculative hardware makes static analysis difficult.

6.5 Static Timing Analysis

- **Control Flow Graph (CFG)**: Used to determine the longest path of execution, crucial for determining WCET. Compilation transforms code into machine-level instructions, enabling timing analysis of each operation.

7 Lecture 7: Embedded Control Systems 2

7.1 Peripherals: PWM, Stepper Motors, A2D Conversion

- **Interaction with Real World**: Embedded systems use sensors (e.g., ADC for temperature, sound, light) and actuators (e.g., motors) to interact with the environment.
- **Pulse Width Modulation (PWM)**: A simple type of Digital to Analog Converter (DAC) used to control devices like DC motors by adjusting the duty cycle.
- **Stepper Motors**: Rotates a fixed number of degrees when given a step signal, making them more precise compared to DC motors. Stepper motor controllers simplify the control by providing specific voltage sequences to coils.

7.2 Analog to Digital Conversion (ADC)

- **Resolution**: Defined by the number of bits in the ADC. Higher resolution provides greater accuracy but requires more bits.
- **Quantization Levels**: The range of an ADC is determined by the reference voltage, divided by the number of levels (e.g., an 8-level ADC with a range of 3.5V has a resolution of 0.4375V).

8 Lecture 8: Embedded Control Systems 3

8.1 Control of Physical Processes

- **Control Systems**: Aim to make the physical system's output track a desired reference input by setting the system's inputs.
  - **Open Loop**: No feedback from the output; relies on an accurate model of the plant.
  - **Closed Loop**: Utilizes feedback to adjust the output to the desired reference, improving accuracy.

8.2 Key Metrics in Control Systems

- **Performance Metrics**: Rise time, peak time, overshoot, and settling time are critical in evaluating control systems.
- **Disturbance Rejection**: A control system must minimize the impact of external disturbances to maintain stability and performance.
- **Stability**: Ensuring that all control variables remain bounded is a primary objective. A stable system should have minimal oscillations and rapid convergence to the desired state.

8.3 Example: Cruise Control

- **Open Loop Control**: A cruise control system adjusts throttle to maintain speed. The model describes how speed changes based on throttle position and current speed, with no feedback mechanism.
- **Closed Loop Control**: Adding sensors to detect actual speed and adjust throttle accordingly helps maintain speed despite disturbances like wind or road conditions.

9 Lecture 9: Industrial Automation 1

9.1 Introduction to Industrial Automation

- **Definition**: Industrial Automation refers to using computers to control machinery and processes in industries, aiming to reduce human intervention and increase efficiency.
- **Advantages**: Improved productivity, consistent quality, reduced waste, increased flexibility, accurate information, and enhanced safety.
- **Drawbacks**: High initial cost and maintenance requirements.

9.2 Programmable Logic Controllers (PLCs)

- **PLCs**: Specialized, hardened computers that are used to control industrial processes. They provide fast response, durability, maintainability, and reliability in tough environments.
- **Human-Machine Interfaces (HMIs)**: Interfaces that allow human operators to interact with PLCs, such as entering parameters or monitoring system status.

9.3 Generations of Industrial Automation

- **Generation 1**: Hard-wired relay logic, replaced by more flexible PLCs.
- **Generation 2**: Introduction of PLCs, significantly improving flexibility and automation capabilities.
- **Generation 3**: Multifunctional PLCs supporting multiple programming languages.
- **Generation 4**: Distributed automation solutions using networking for improved flexibility and modularity.

10 Lecture 10: Industrial Automation 2

10.1 IEC 61131 Standard

- **PLC Programming Languages**: The IEC 61131 standard defines programming languages for PLCs, including Ladder Logic, Instruction List, Structured Text, Function Block Diagrams, and Function Charts.
- **Ladder Logic**: An old but still widely used language, using graphical symbols to represent control actions, making it easy to understand and maintain.

10.2 Ladder Logic Features

- **Basic Elements**: Includes contacts, coils, and connectors, with vertical power rails representing power flow through the rungs of the diagram.
- **Timers and Counters**: Used for controlling time-based operations, with variations like On-Delay, Off-Delay, and Pulse timers. Counters are used to count occurrences and trigger actions after a specific count.

10.3 Example: Pneumatic Cylinder

- **Control Sequence**: A pneumatic cylinder controlled by ladder logic, moving forward when a START button is pressed, and returning when fully extended. This sequence can be represented as a Moore Machine for precise state transitions.
- **Concurrency in Ladder Logic**: Used to control multiple devices simultaneously, ensuring that state transitions occur correctly based on sensor input and predefined logic.