الجمهورية الجزائرية الديمقراطية الشعبية
People's Democratic Republic of Algeria
وزارة التعليم العالي و البحث العلمي
Ministry of Higher Education and Scientific Research
جـامـعـة الشهيد حمّه لخضر – الوادي
Martyr Hama Lakhdar University in El Oued

Faculty of Exact Sciences
Computer Science Department

كليـة العلوم الدقيقة
قسم الإعلام الآلي

Course Content

Prepared for

EMBEDDED SYSTEM DESIGN

Computer Science
Master 1
Internet of Things and Cyber Security
Semester 2

Prepared by;

Mounir Beggas

Course Content of

EMBEDDED SYSTEM DESIGN

Computer Science
Master 1
Internet of Things and Cyber Security
Semester 2

Prepared by;

Mounir Beggas

Table of Contents

Table of Contents **2**
Introduction **5**
CHAPTER 1 ntroduction to Embedded Systems and IoT **9**
1. What is an Embedded System? 9
1.1. Definition 9
1.2. Key Characteristics 9
1.3. Examples 9
2. Embedded System Architecture 9
2.1. Hardware Components 9
2.2. Software Components 10
3. Introduction to IoT (Internet of Things) 10
3.1. Definition 10
3.2. IoT Architecture Layers 10
4. Key Technologies in Embedded IoT 11
4.1. Microcontrollers for IoT 11
4.2. Communication Protocols 11
4.3. Security Challenges 11
5. Applications of Embedded IoT 11
5.1. Smart Agriculture 11
5.2. Smart Cities 11
5.3. Healthcare 11
5.4. Industrial IoT (IIoT) 11
CHAPTER 2 Embeded System Architecture and MCU Programming **12**
1. Introduction 12
2. AVR microcontroller ATmega328P 12
2.1. ATmega328P Pins 14
2.2. AVR architecture 17
2.3. AVR CPU Architecture 27
2.3.1. AVR CPU core 28
2.3.2. Status and Control 29
2.3.3. AVR CPU General Purpose Working Registers 35
2.3.4. AVR CPU SRAM Data Memory 37

2.3.5. AVR Program Memory....39
2.4. Assembly Programming in AVR CPU....42
2.4.1. Working With Registers R0-R31....42
2.4.2. Logical Instructions....43
2.4.3. Arithmetic Instructions....44
2.4.4. Bit Shifts....45
2.4.5. Subroutines and The Stack Pointer....46
2.4.6. Addressing Modes....51
2.4.7. Accessing I/O registers....52
2.4.8. Accessing SRAM....53
2.4.9. Loading Data from Program Memory....55
CHAPTER 3 Interrupts and Events in Embedded Systems....57
1. Introduction....57
2. Interrupts....57
2.1. Interrupt Vector and ISR....58
2.2. Interrupt Handling....63
2.3. External Interrupts....63
2.3.1. INT0 and INT1....64
2.3.2. Pin Change Interrupt PCINT[23:0]....66
CHAPTER 4 Timers and Event Scheduling in Embedded System....68
1. Introduction....68
2. Timer Interrupt....69
2.1. Timer Interrupt Modes....69
3. Timer/Counter Control Registers....69
4. Timer Prescaler....70
5. Configuring Timer Interrupt....71
5.1. Timer/Counter 1 interrupt overflow....71
5.2. Timer/Counter 1 preloading....77
5.3. Timer Compare Match Registers....79
CHAPTER 5 Communications Protocols....84
1. Introduction....84
2. Basics of Serial Communications....84
3. Transmitting and receiving serial data....85
3.1. Data framing....85
3.2. UART Interface....88
3.3. Steps of UART Transmission....89
4. Serial Communications with the ATmega328P....92

4.1. BAUD Rate and Control Registers....92
4.2. UART Configuration....100
5. Transmission and Recepetion....103
5.1. Polling transmission....103
5.2. Polling Reception....104
5.3. Interrupt Transmission....104
5.4. Interrupt Reception....105
CHAPTER 6 Analog-to-Digital Conversion for Sensor and Actutor Control....107
1. Introduction....107
2. How Does an ADC Work?....107
3. ADC in Microcontroller, Case of ATmega328P....110
3.1. Reference Voltages....110
3.2. Conversion Rate....111
3.3. Registers....111
3.4. Using the ADC....115
3.4.1. ADC Conversions Using Polling....116
3.4.2. ADC Conversions Using Interrupts....117
CHAPTER 7 Finite State Machine for Embedded Systems Design....120
1. Introduction....120
2. What is a state machine(FSM) ?....120
3. Benefits of Using State Machines (FSMs)....121
4. UML state machines....122
4.1. Guidance Example: UML State Machine....122
4.2. State....123
4.2.1. Types of states in UML....124
4.2.2. Simple state....124
4.2.3. Composite State....127
4.3. Transition....130
4.4. Events (Triggers)....134
4.5. Pseudo states....135
4.6. Implementation of state machine....139
4.6.1. Nested Switch Approach....139
4.6.2. State Table Approach....141
4.6.3. State Handler Approach (Function Pointer Approach)....143
CHAPTER 8 Sensor and Actuator Control Strategies in Embedded Systems.... 148
1. Introduction....148
2. What is a Controller?....148

3. Open-Loop Control Systems..... 149
4. Closed-Loop (Feedback) Control Systems..... 149
5. Control Algorithm Types in Closed-Loop Systems..... 150
5.1. Proportional (P) Control..... 150
5.1.1. Embedded System Implementation.....151
5.2. Integral (I) Control..... 153
5.3. Embedded System Example: Integral Control.....155
5.4. Derivative (D) Control..... 157
5.4.1. Embedded Systems Example..... 158
5.5. Proportional-Integral-Derivative (PID) Controller..... 159
5.5.1. Embedded System Example..... 160
References..... 163

Introduction

This course provides a comprehensive introduction to microcontroller architecture and embedded systems design, with a focus on practical applications and hands-on implementation. Embedded systems are specialized computing systems designed to perform dedicated functions within larger mechanical or electrical systems. These systems are ubiquitous in modern technology, found in everything from household appliances and automotive systems to industrial machines and medical devices. At the heart of every embedded system lies a microcontroller—a compact integrated circuit that combines a processor core with memory and programmable input/output peripherals.

The course begins with an exploration of microcontroller architecture, using the popular ATmega328P microcontroller as a primary example. Students will gain an understanding of the internal structure of microcontrollers, including CPU cores, memory organization (Flash, SRAM, EEPROM), and various peripheral interfaces. The ATmega328P, known for its use in Arduino boards, serves as an excellent platform for learning due to its widespread adoption in educational and professional settings. We will examine its key features such as operating voltage range, clock speed, memory capacity, and the array of built-in peripherals including timers, PWM channels, and communication interfaces.

The first chapter of the course is dedicated to low-level programming, particularly assembly language programming for the AVR architecture. Students will learn how to write efficient code that interacts directly with hardware resources, gaining insights into instruction sets, addressing modes, and register operations. This foundational knowledge is crucial for developing optimized embedded systems where resource constraints and real-time performance are critical considerations.

The course then progresses to cover essential topics in embedded systems development, including interrupt handling (Chapter 2), timer/counter operations (Chapter 3), and various communication protocols. Interrupts are presented as a powerful mechanism for handling asynchronous events, allowing microcontrollers to respond immediately to specific conditions without constant polling. Students will learn

about interrupt vectors, priority handling, and the structure of Interrupt Service Routines (ISRs). The timer/counter modules are examined in depth, demonstrating their use in generating accurate time delays, PWM signals, and capturing external events.

Serial communication (Chapter 4) forms another critical component of the curriculum, with particular emphasis on USART (Universal Synchronous/Asynchronous Receiver-Transmitter) implementation. Students will understand the principles of asynchronous communication, baud rate configuration, and data framing, along with both polling and interrupt-driven approaches to data transmission and reception. The course also covers analog-to-digital conversion (ADC) (Chapter 5), explaining the process of converting real-world analog signals into digital values that microcontrollers can process, including sampling, quantization, and encoding techniques.

A unique aspect of this course is its focus on structured design methodologies for embedded systems. Students are introduced to UML state machines -Chapter 6) as a powerful tool for modeling complex system behaviors. Through practical examples like timer control applications, learners will understand how to implement finite state machines using various approaches including nested switch statements, state tables, and state handler functions. This formal design approach helps manage complexity in embedded systems and improves code maintainability.

The course concludes with an examination of control strategies in embedded systems (Chapter 7), comparing open-loop and closed-loop (feedback) control systems. Students will explore fundamental control algorithms including proportional (P), integral (I), and derivative (D) controllers, culminating in the implementation of complete PID (Proportional-Integral-Derivative) controllers. Practical examples, such as temperature control systems, demonstrate how these algorithms are implemented in real-world embedded applications using microcontrollers, sensors, and actuators.

Throughout the course, emphasis is placed on the practical application of concepts through hands-on exercises and projects. Students will work with actual microcontroller hardware (or simulators) to implement the various concepts covered, from basic I/O operations to complex state machines and control systems. By the end of the course, students will have developed a strong foundation in microcontroller architecture and embedded systems design, equipping them with the skills needed to tackle real-world embedded systems challenges across various industries. The knowledge gained will be

applicable not only to the ATmega328P but to microcontroller programming in general, providing a versatile skill set for future embedded systems work.

CHAPTER 1 ntroduction to Embedded Systems and IoT

1. What is an Embedded System?

1.1. Definition

An embedded system is a dedicated computing system designed to perform specific tasks, often with real-time constraints. Unlike general-purpose computers, embedded systems are optimized for efficiency, reliability, and low power consumption.

1.2. Key Characteristics

- Task-Specific: Designed for a dedicated function (e.g., microwave control, automotive ECU).
- Real-Time Operation: Many systems require deterministic response times (e.g., anti-lock braking systems).
- Resource-Constrained: Limited memory, processing power, and energy (battery-powered devices).
- Low-Level Hardware Interaction: Direct control over sensors, actuators, and peripherals.

1.3. Examples

- Consumer Electronics: Smartwatches, washing machines.
- Automotive: Engine control units (ECUs), infotainment systems.
- Industrial: PLCs (Programmable Logic Controllers), robotics.
- Medical: Pacemakers, insulin pumps.

2. Embedded System Architecture

2.1. Hardware Components

1. Microcontroller/Microprocessor (e.g., ARM Cortex-M, AVR, ESP32).
2. Memory: Flash (program storage), RAM (runtime data), EEPROM (persistent data).
3. I/O Peripherals:

- Sensors (temperature, motion, moisture).
- Actuators (motors, valves, LEDs).
- Communication interfaces (UART, SPI, I2C, CAN bus).

2.2. Software Components

- Real-Time Operating System (RTOS) (e.g., FreeRTOS, Zephyr).
- Firmware: Low-level code (C/C++, Rust) interacting directly with hardware.
- Device Drivers: Interface between hardware and OS.

3. Introduction to IoT (Internet of Things)

3.1. Definition

IoT extends embedded systems with connectivity, enabling data exchange over networks (Wi-Fi, Bluetooth, LoRa, cellular).

IoT vs. Traditional Embedded Systems

Feature	**Embedded Systems**	**IoT Devices**
Connectivity	Often standalone	Network-connected
Data Usage	Local processing	Cloud/edge analytics
Power	Optimized for low power	May use higher power (Wi-Fi)
Examples	Microwave controller	Smart thermostat

3.2. IoT Architecture Layers

1. Perception Layer: Sensors/actuators (e.g., soil moisture sensor).
2. Network Layer: Communication protocols (MQTT, HTTP, CoAP).
3. Cloud/Edge Layer: Data storage and processing (AWS IoT, Azure IoT).
4. Application Layer: User interfaces (mobile apps, dashboards).

4. Key Technologies in Embedded IoT

4.1. Microcontrollers for IoT

- Low Power: ESP32 (Wi-Fi/BLE), STM32U5 (ultra-low-power).
- Wireless: Nordic nRF52 (Bluetooth), LoRaWAN nodes.

4.2. Communication Protocols

- Short-Range: Bluetooth Low Energy (BLE), Zigbee.
- Long-Range: LoRa, NB-IoT.
- Internet Protocols: MQTT (lightweight), HTTP/HTTPS.

4.3. Security Challenges

- Device Authentication: Secure boot, TLS certificates.
- Data Encryption: AES-256, RSA.
- OTA Updates: Secure firmware updates.

5. Applications of Embedded IoT

5.1. Smart Agriculture

- Soil monitoring → Automated irrigation (like the UML state machine example).

5.2. Smart Cities

- Traffic light control, waste management.

5.3. Healthcare

- Wearable ECG monitors, remote patient monitoring.

5.4. Industrial IoT (IIoT)

- Predictive maintenance using vibration sensors.

CHAPTER 2 Embeded System Architecture and MCU Programming

1. Introduction

Understanding microcontroller architecture is fundamental to developing efficient and reliable embedded systems. This chapter provides an overview of microcontroller fundamentals, with a focus on the AVR architecture, particularly the ATmega328P. It explores the internal structure of the microcontroller, including the CPU core, general-purpose registers, memory organization (Flash, SRAM, EEPROM), and input/output ports. In addition, the chapter introduces assembly language programming as a low-level approach to directly control hardware resources. It explains the syntax and structure of assembly instructions, addressing modes, and the use of the AVR instruction set for data manipulation, control flow, and hardware interfacing. By mastering the foundational concepts presented in this chapter, readers will gain the essential knowledge required to understand and program AVR-based systems at the hardware level.

2. AVR microcontroller ATmega328P

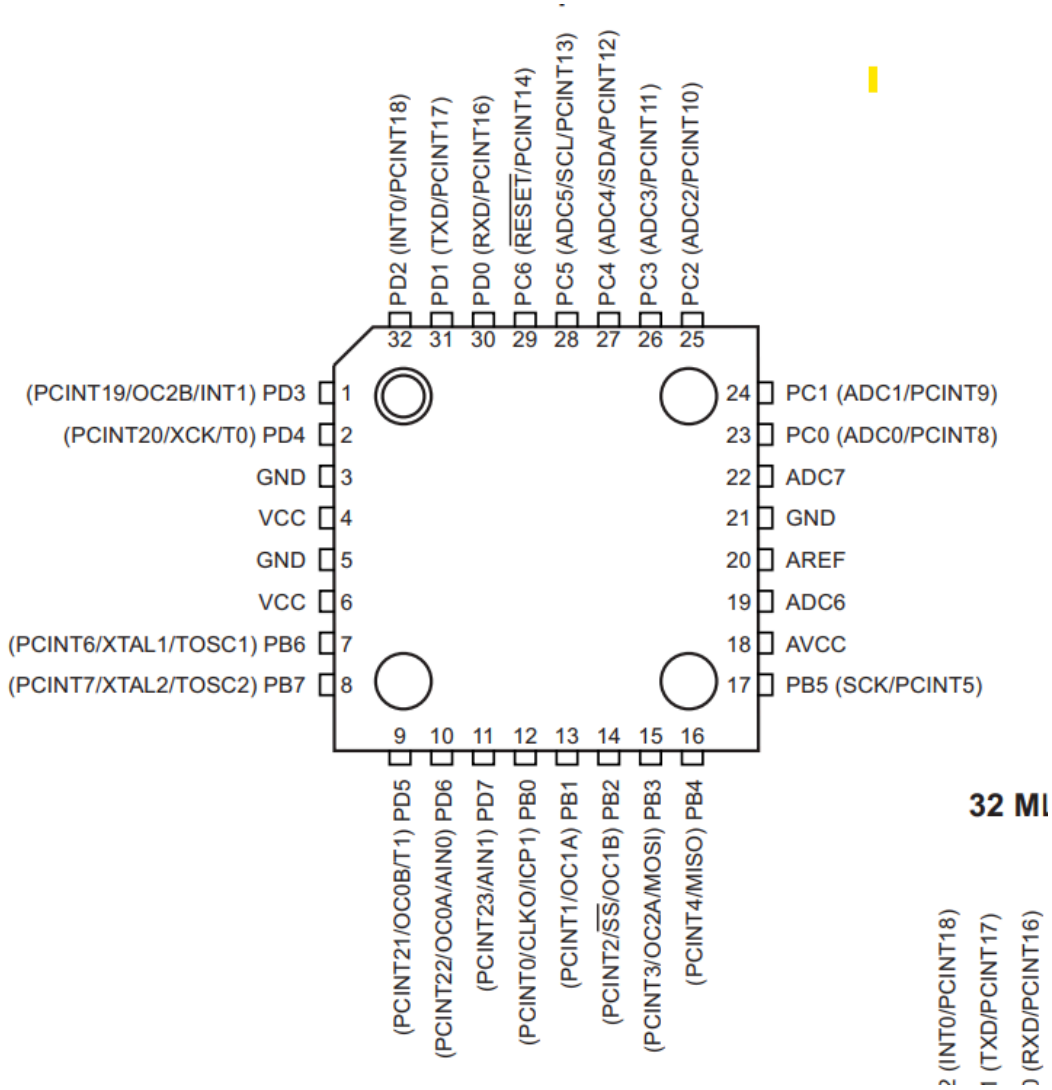
The ATmega328P is an 8-bit AVR RISC-based microcontroller developed by Microchip Technology (formerly Atmel). It's widely known as the core microcontroller of the Arduino Uno, making it popular in embedded systems, DIY electronics, and educational platforms.

Key Features

- **Architecture:** 8-bit AVR
- **Operating Voltage:** 1.8V – 5.5V
- **CPU Speed:** Up to 20 MHz

- **Flash Memory:** 32 KB (of which 0.5 KB is used by the bootloader)
- **SRAM:** 2 KB
- **EEPROM:** 1 KB
- **GPIO Pins:** 23 I/O lines
- **Timers:** 3 (Two 8-bit and one 16-bit)
- **PWM Channels:** 6
- **ADC:** 10-bit, 6-channel ADC (up to 8 channels in TQFP/MLF package)
- **Communication Protocols:**
    - **USART** (serial communication)
    - **I2C (TWI)**
    - **SPI**
- **Watchdog Timer** with separate on-chip oscillator
- **External and Internal Interrupts**
- **Power-saving Modes:** Idle, ADC Noise Reduction, Power-down, Power-save, Standby, and Extended Standby

2.1. ATmega328P Pins



The ATmega328P microcontroller features a 28-pin configuration, each serving specific functions essential for various applications. Here's a concise overview of its pin assignments:

Power Supply Pins:

- **VCC (Pin 7):** Digital supply voltage.
- **GND (Pins 8 and 22):** Ground reference points.
- **AVCC (Pin 20):** Supply voltage for the analog-to-digital converter (ADC).

- **AREF (Pin 21):** Reference voltage for the ADC.

Port B (Pins 9-16): This 8-bit bi-directional I/O port can serve as digital inputs or outputs. Additionally, certain pins offer specialized functions:

- **PB0-PB5 (Pins 12-17):** Digital I/O pins; also function as SPI communication lines (MOSI, MISO, SCK) and PWM outputs.
- **PB6-PB7 (Pins 9-10):** Oscillator pins for external crystal connections.

Port C (Pins 23-28): Another 8-bit bi-directional I/O port, primarily used for analog inputs:

- **PC0-PC5 (Pins 23-28):** Analog input channels (ADC0 to ADC5); can also function as digital I/O.

Port D (Pins 1-8): This port serves multiple roles, including serial communication and external interrupts:

- **PD0-PD1 (Pins 2-3):** UART serial communication pins (RXD and TXD).
- **PD2-PD3 (Pins 4-5):** External interrupt pins (INT0 and INT1).
- **PD4-PD7 (Pins 6-9):** Digital I/O pins; some support PWM output.

Reset Pin:

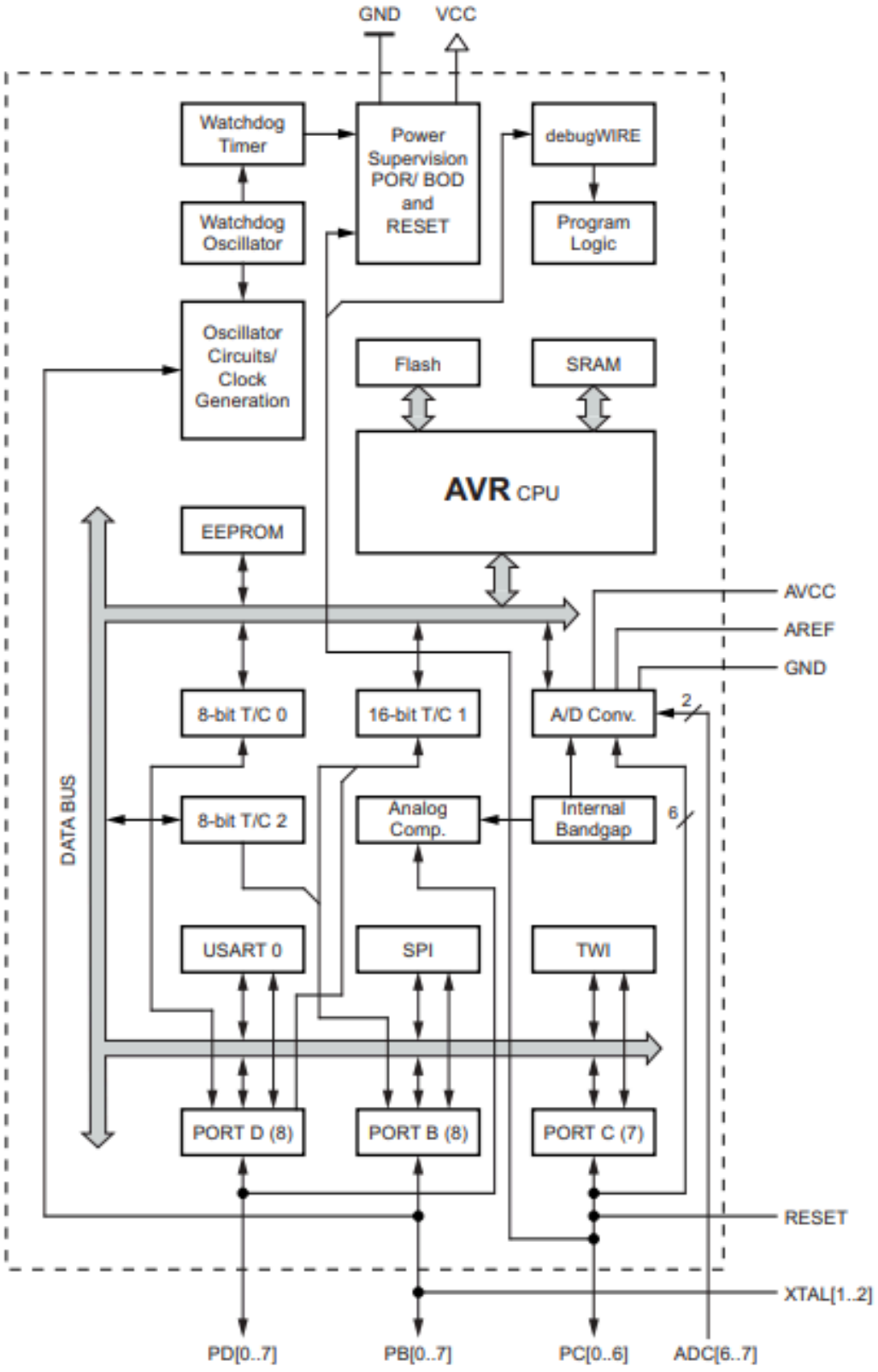
- **RESET (Pin 1):** Active-low input that resets the microcontroller when pulled low.

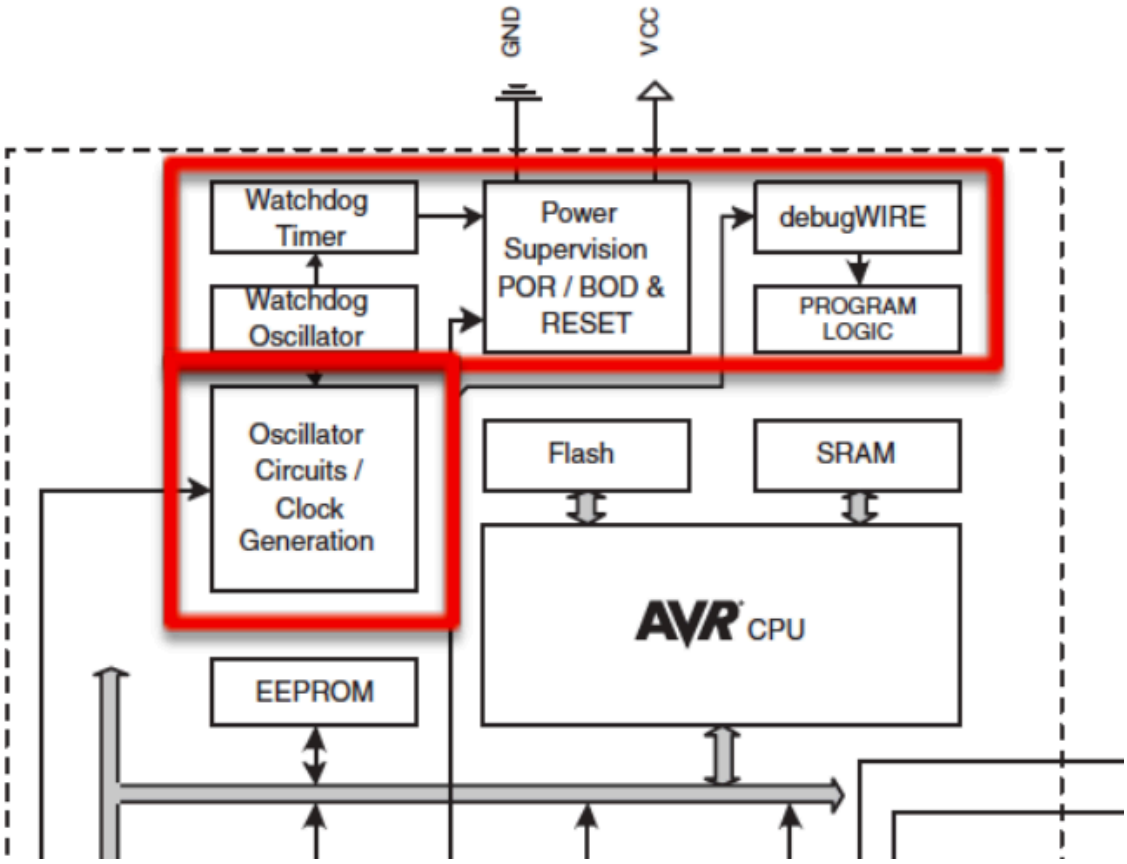
Port Pin Definitions in AVR-C

```
#define PINB _SFR_IO8(0x03)
#define PINB0 0
...
#define PINB7 7

#define DDRB _SFR_IO8(0x04)
#define DDB0 0
...
#define DDB7 7

#define PORTB _SFR_IO8(0x05)
#define PORTB0 0
...
#define PORTB7 7

#define PINC _SFR_IO8(0x06)
#define PINC0 0
...
#define PINC6 6

#define DDRC _SFR_IO8(0x07)
#define DDC0 0
...
#define DDC6 6
```

```
#define PORTC _SFR_IO8(0x08)
#define PORTC0 0
...
#define PORTC6 6

#define PIND _SFR_IO8(0x09)
#define PIND0 0
...
#define PIND7 7

#define DDRD _SFR_IO8(0x0A)
#define DDD0 0
...
#define DDD7 7

#define PORTD _SFR_IO8(0x0B)
#define PORTD0 0
...
#define PORTD7 7
```

2.2. AVR architecture


GND
VCC
Watchdog Timer
Power Supervision POR/ BOD and RESET
debugWIRE
Watchdog Oscillator
Program Logic
Oscillator Circuits/ Clock Generation
Flash
SRAM
AVR CPU
EEPROM
AVCC
AREF
GND
8-bit T/C 0
16-bit T/C 1
A/D Conv.
2
DATA BUS
8-bit T/C 2
Analog Comp.
Internal Bandgap
6
USART 0
SPI
TWI
PORT D (8)
PORT B (8)
PORT C (7)
RESET
XTAL[1..2]
PD[0..7]
PB[0..7]
PC[0..6]
ADC[6..7]



Watchdog Timer (WDT): A safety feature that resets the microcontroller if the program becomes unresponsive.

Watchdog Oscillator: A low-power independent clock source that drives the Watchdog Timer.

Oscillator Circuit: Generates the system clock to control the execution speed of the microcontroller.

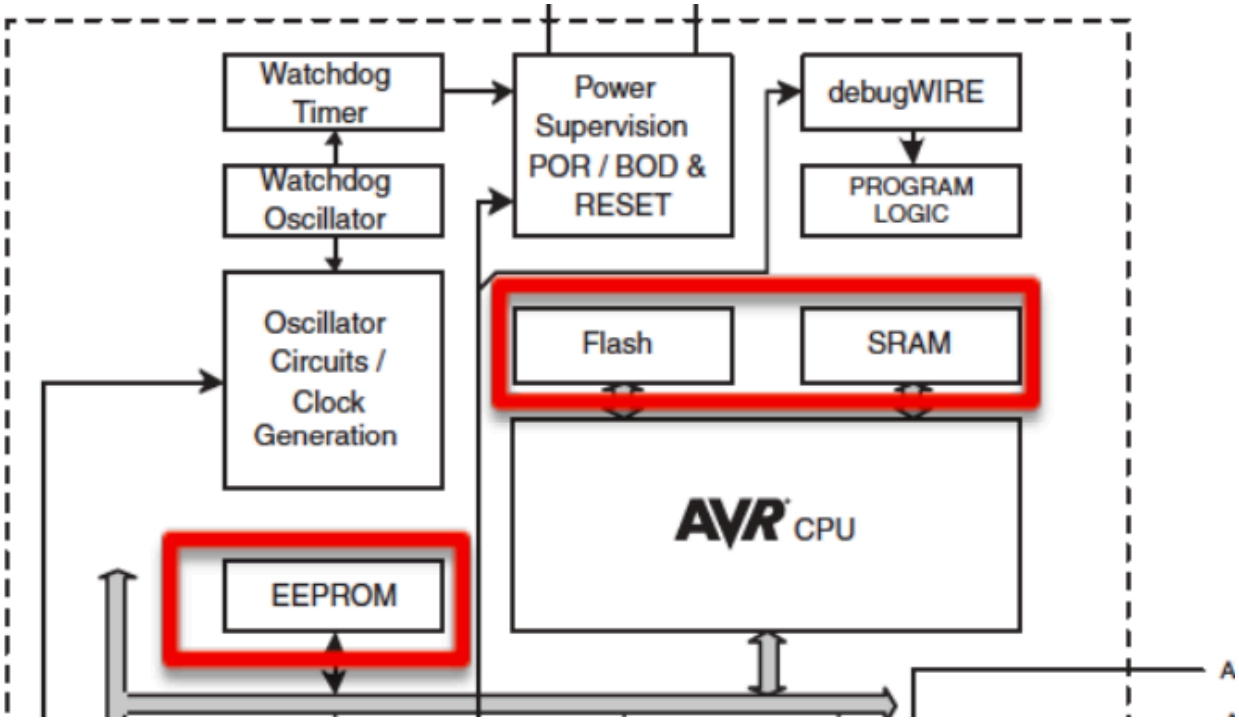
Power Supervision: Ensures stable operation by managing voltage fluctuations through reset mechanisms.

Brown-Out Detection (BOD): Resets the microcontroller if the supply voltage drops below a safe threshold.

Power-On Reset (POR): Ensures the microcontroller starts in a known state when

power is applied.

DebugWIRE: A one-wire interface for debugging and real-time monitoring via the RESET pin.



The **Harvard architecture** is a computer architecture that uses **separate memory and buses** for **program instructions** and **data**. This allows the CPU to **fetch instructions and access data simultaneously**, improving performance and efficiency.

The **Flash memory** stores the program (instructions).32k

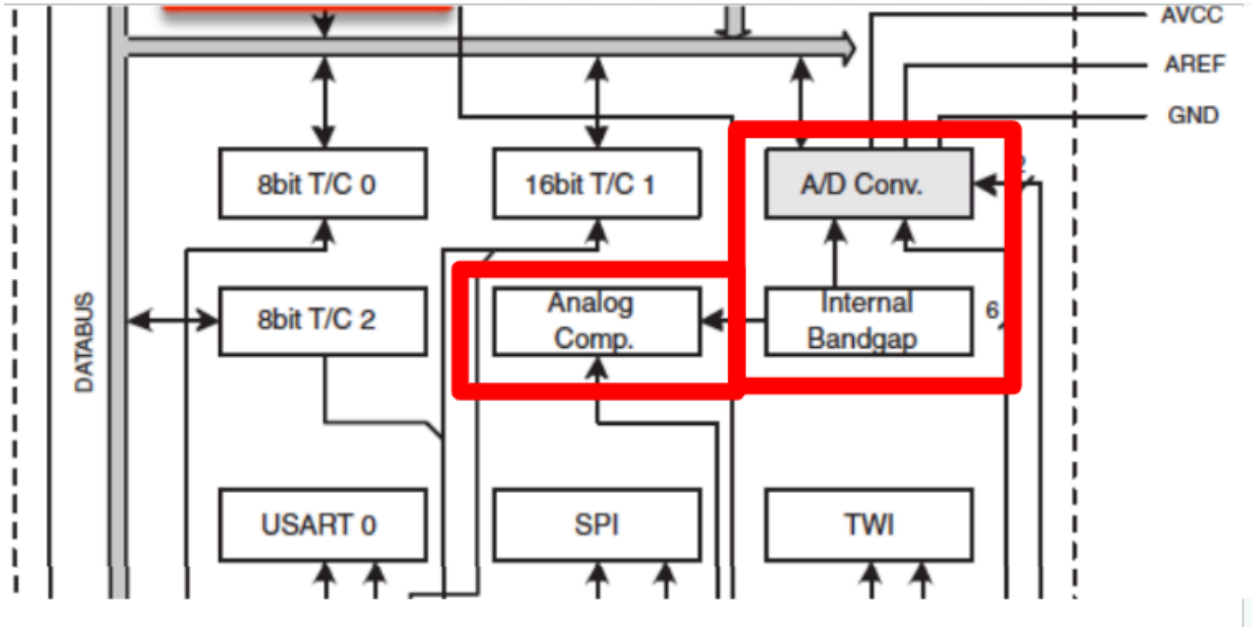
Non-volatile

The **SRAM stores data and variables** used during execution 2k.

Volatile

The **EEPROM stores non-volatile data** that remains after power-off. 1k

Long-term data



A/D Converter (ADC - Analog to Digital Converter)

- The **ADC** converts an **analog voltage** (from an external sensor or input) into a **digital value** that the microcontroller can process.

Internal Bandgap Reference

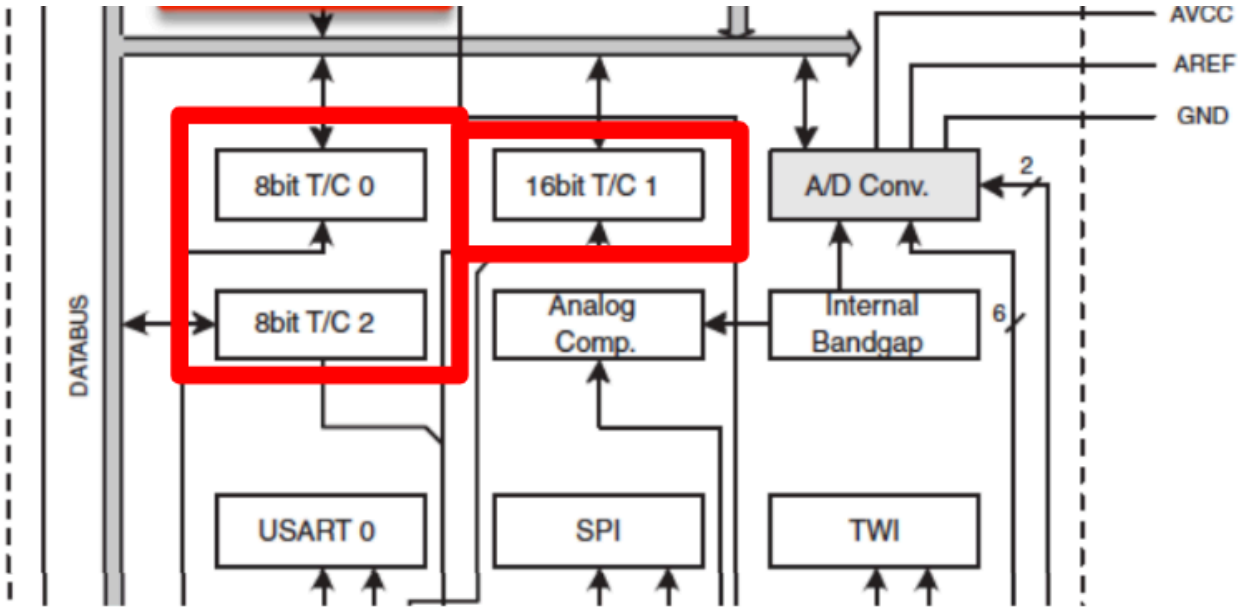
- The **internal bandgap** is a **fixed, stable voltage source (1.1V)** inside the

microcontroller.

- It can be used as a **reference voltage** for the ADC to improve accuracy.

Analog Comparator (Analog Comp)

- The **Analog Comparator** compares two **analog voltage inputs** and outputs a digital HIGH or LOW signal based on which voltage is higher.



8-bit Timer/Counter 0 (T/C 0)

- **Resolution:** 8-bit (counts from 0 to 255).
- **Usage:** Used for **PWM generation, timing events, and simple delays**.

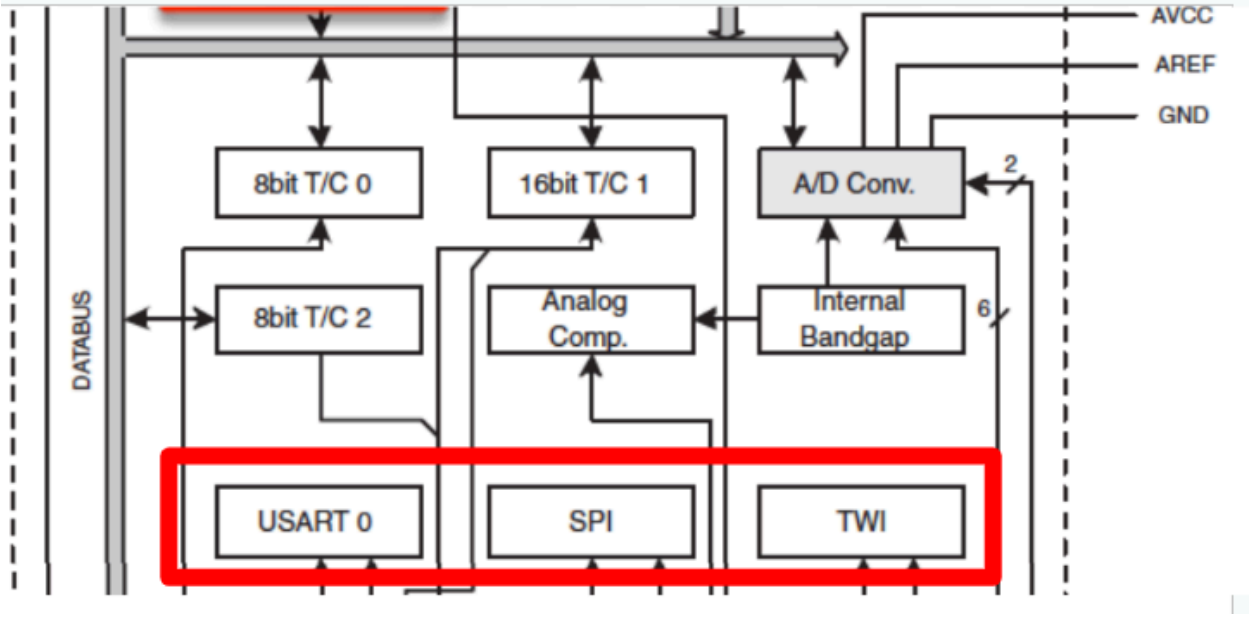
16-bit Timer/Counter 1 (T/C 1)

- **Resolution:** 16-bit (counts from 0 to 65,535).

8-bit Timer/Counter 2 (T/C 2)

- **Resolution:** 8-bit (counts from 0 to 255).
- **Usage:** Similar to Timer 0, but can also operate with an **asynchronous clock**

(external 32.768 kHz crystal for real-time clock applications).



USART (Universal Synchronous and Asynchronous Receiver-Transmitter)

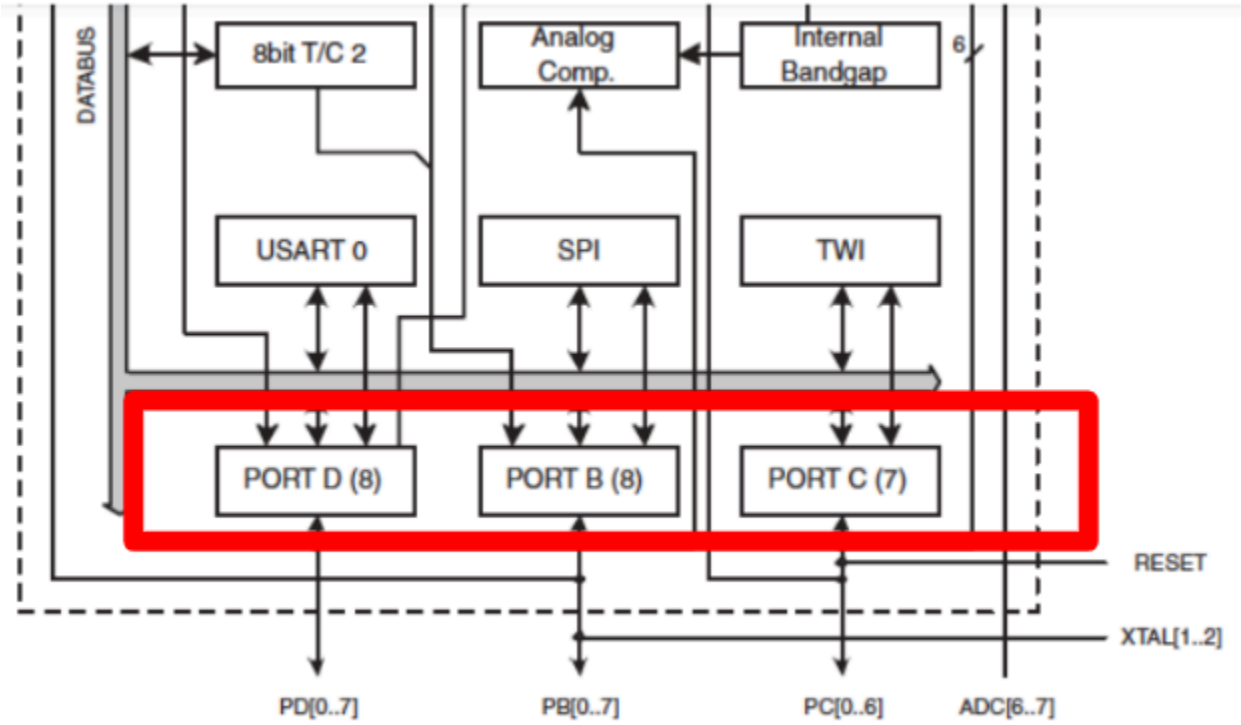
- **Type:** Serial communication (UART/USART)
- **Mode:** Can work in both **synchronous** (clocked) and **asynchronous** (no clock) modes.
- **Pins:** TX (Transmit) and RX (Receive) on **PD0 (RX) and PD1 (TX)**.

SPI (Serial Peripheral Interface)

- **Type:** Synchronous serial communication (Master-Slave).

TWI (Two-Wire Interface) / I²C (Inter-Integrated Circuit)

- **Type:** Synchronous serial communication (Multi-Master, Multi-Slave).



PORT D, PORT B, and PORT C in ATmega328P

In the **ATmega328P** microcontroller, the **GPIO (General-Purpose Input/Output) ports** are grouped into three main **8-bit** ports: **PORT D, PORT B, and PORT C**. Each port consists of **8 pins** that can function as **digital input or output** and sometimes have **special functions**.

PORT D (PD0 – PD7)

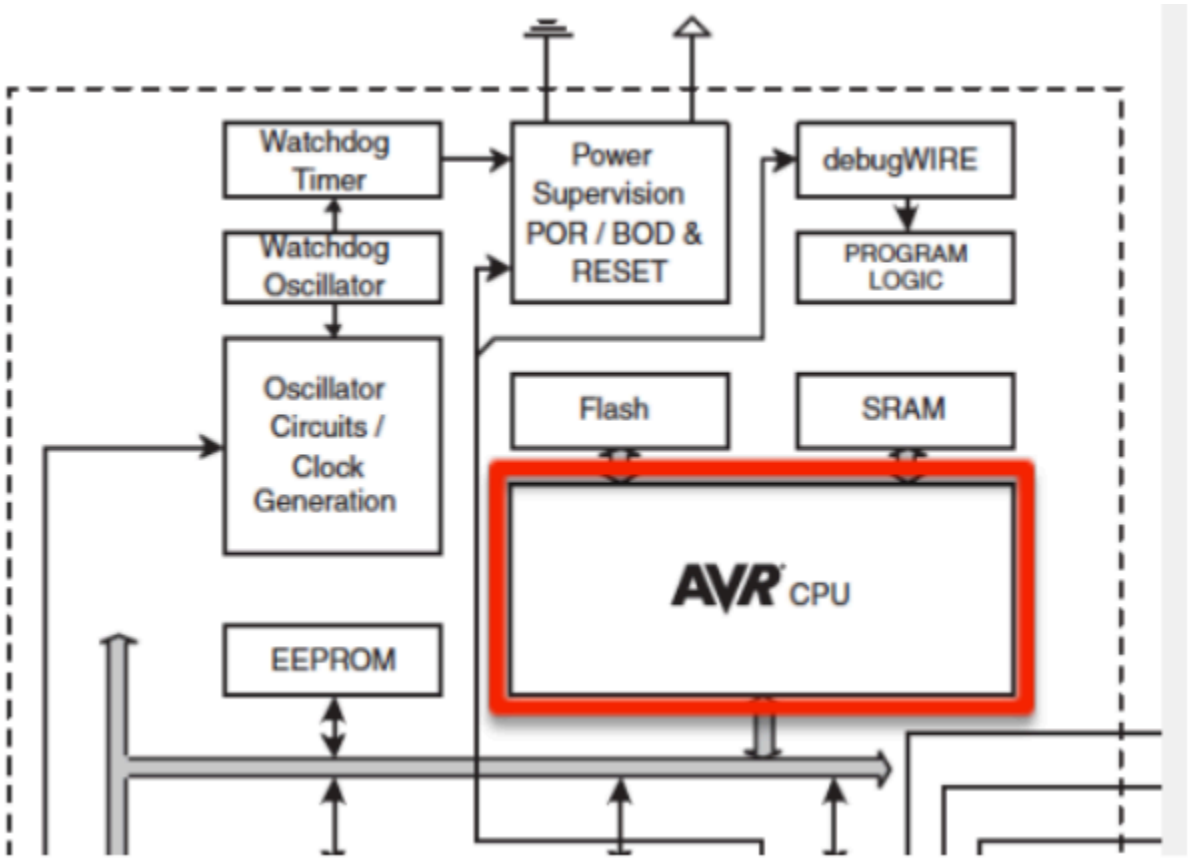
- **Pins: PD0 to PD7**
- **Digital I/O** (Input/Output).
    - **USART (Serial Communication)**:**PD0 (RX)** – Receive data.**PD1 (TX)** – Transmit data.
    - **External Interrupts:PD2 (INT0)** and **PD3 (INT1)** – Used for external interrupts.
    - **PWM (Pulse Width Modulation): PD5, PD6** – Used in PWM mode, ex, for motor control.

PORT B (PB0 – PB7)

- **Pins: PB0 to PB7**
- **SPI Communication (Serial Peripheral Interface):**
    - **PB3 (MOSI)** – Master Out, Slave In.
    - **PB4 (MISO)** – Master In, Slave Out.
    - **PB5 (SCK)** – Serial Clock.
    - **PB2 (SS)** – Slave Select.
    - **PWM Outputs:**
        - **PB1, PB2, PB3** – Used in PWM mode.
    - **Oscillator (External Crystal Connection):**
        - **PB6 (XTAL1)** and **PB7 (XTAL2)** – Used for connecting an external crystal oscillator.

PORT C (PC0 – PC7)

- **Pins: PC0 to PC6** (*PC7 is not available in ATmega328P DIP package*).
- **Digital I/O**.
- **Analog Inputs (ADC - Analog-to-Digital Converter):**
    - **PC0 to PC5 (ADC0 – ADC5)** – Used as analog input channels for sensors.
- **TWI (I²C) Communication:**
    - **PC4 (SDA)** – Data line for I²C.
    - **PC5 (SCL)** – Clock line for I²C.



AVR CPU (Central Processing Unit)

The **AVR CPU** is the core of the **ATmega328P** microcontroller, responsible for executing instructions and handling operations. It follows the **Harvard architecture**, meaning it has **separate memory spaces** for **program instructions (Flash memory)** and **data (SRAM, EEPROM)**.

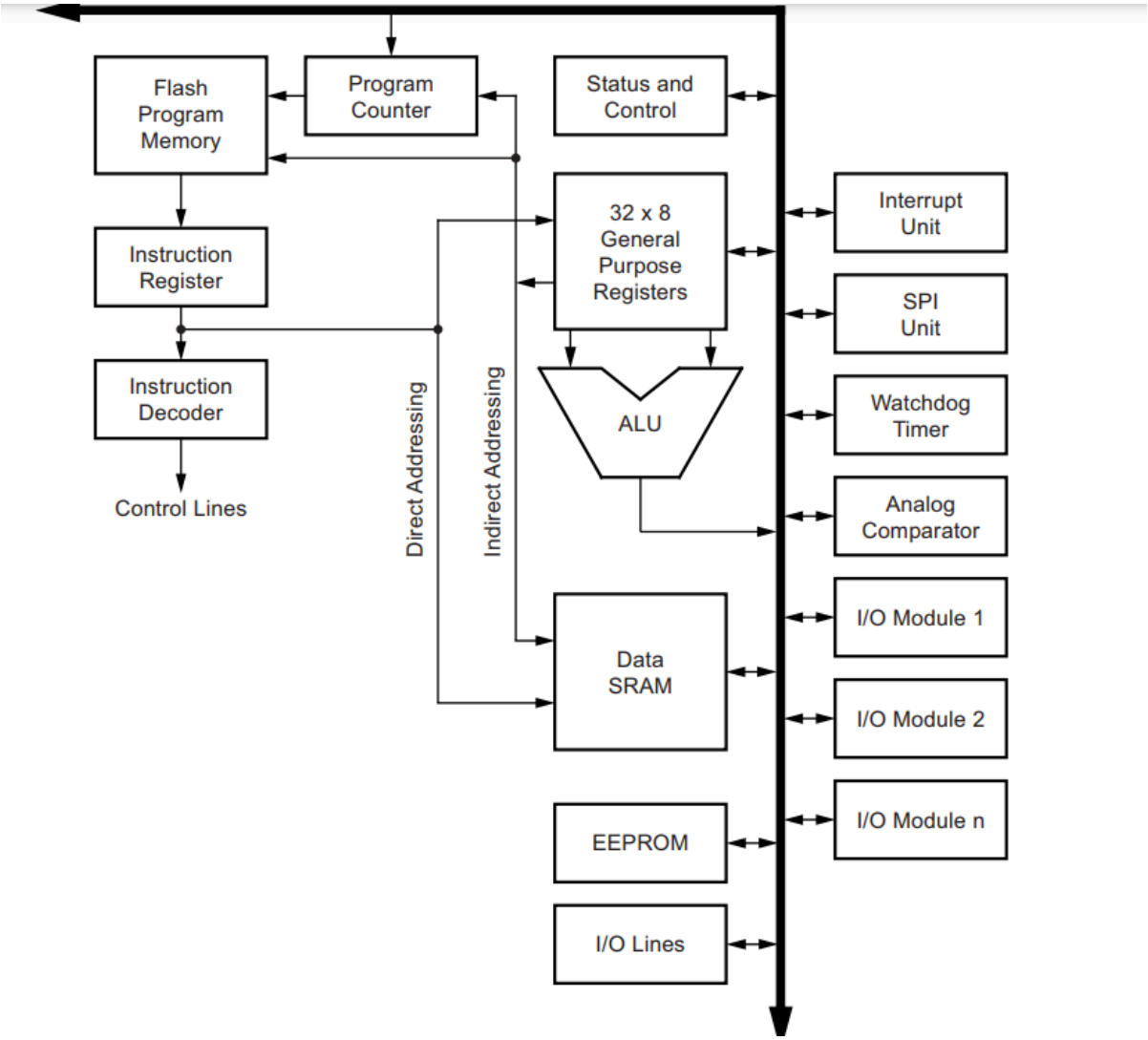
RISC Architecture (Reduced Instruction Set Computing)

- Executes most instructions in **1 clock cycle**, making it fast and efficient.
- Supports **131 powerful instructions**, including arithmetic, logic, and branching operations.

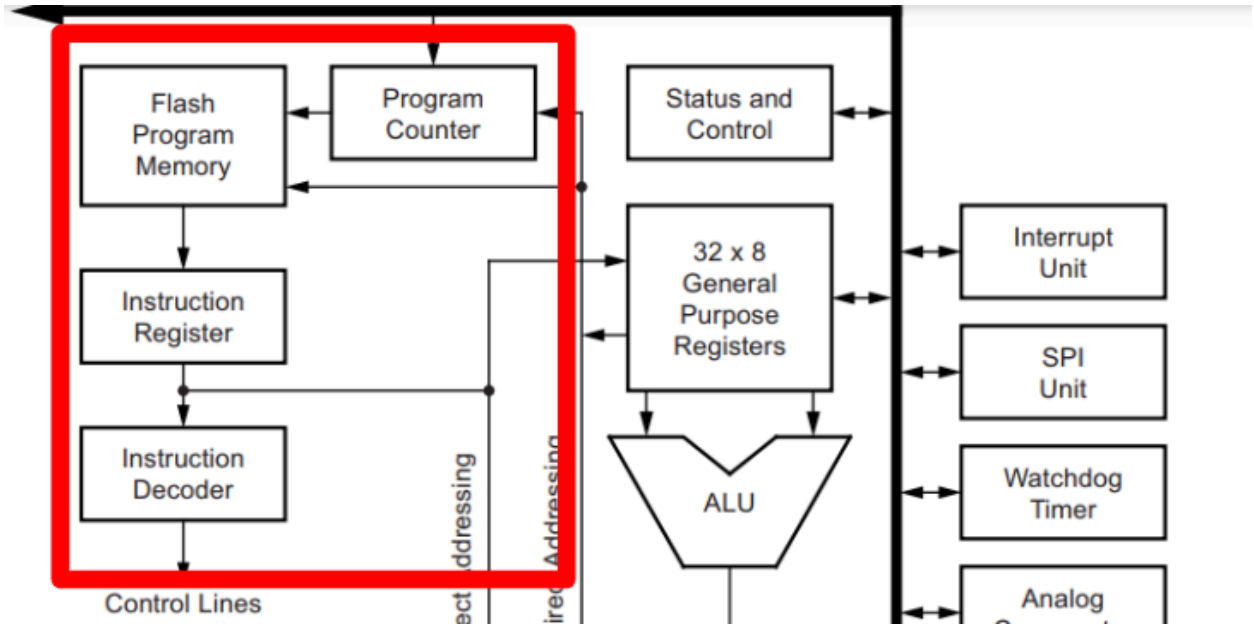
8-bit Processor

- Handles **8-bit data operations**, meaning it processes data in chunks of **8 bits (1 byte)** at a time.

2.3. AVR CPU Architecture



2.3.1. AVR CPU core



Program Counter (PC)

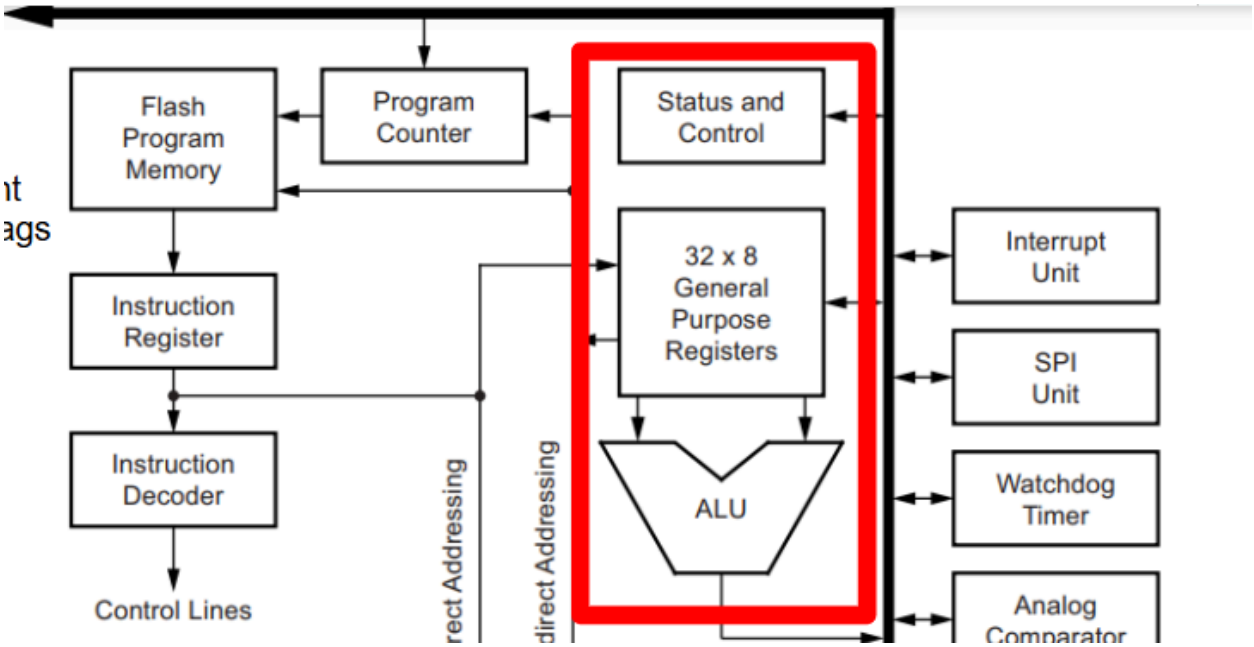
- A **16-bit register** that stores the address of the next instruction to be executed.
- Since **ATmega328P** has **Flash memory of up to 32 KB**, the **PC** can address the entire program memory space.

Instruction Register (IR)

- The **Instruction Register (IR)** temporarily holds the **current instruction** being executed.
- It receives the instruction from **Flash memory** during the **fetch cycle** and passes it to the **Instruction Decoder** for interpretation.

Instruction Decoder

- The **Instruction Decoder** interprets the binary instruction stored in the **Instruction Register** and generates control signals for execution.
- It **decodes the opcode** (operation code) and determines which operation needs to be performed (e.g., ADD, SUB, LOAD, STORE).



2.3.2. Status and Control

The **Status and Control** section of the AVR CPU manages execution flow, flags, and system behavior. The most important part is the **Status Register (SREG)**, which stores important flags that indicate the result of an operation.

32 General-Purpose Registers (R0 – R31) 8 bits

- **32 fast-access registers** in the CPU.
- The first **16 registers (R0 – R15)** support general operations, while the last **16 (R16 – R31)** can be used with immediate values.

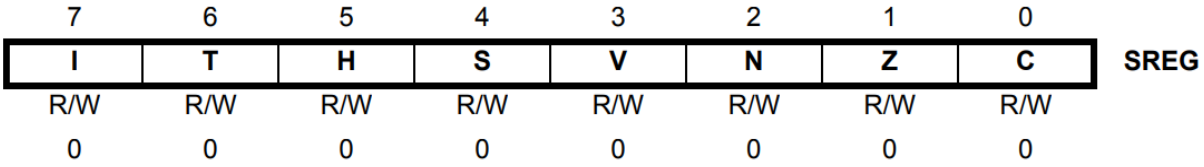
Arithmetic Logic Unit (ALU)

The **ALU (Arithmetic Logic Unit)** performs arithmetic and logical operations on data. It is a key part of the AVR CPU and works closely with the **instruction decoder** and **registers**.

Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed.

- **Arithmetic Operations**: ADD, SUB, MUL, DIV

- **Logical Operations**: AND, OR, XOR, NOT
- **Bit Manipulation**: SHIFT, ROTATE
- **Comparison**: CMP, BRANCHING based on flags



The status register contains information about the result of the most recently executed arithmetic instruction. It is updated after all ALU operations.

Bit 7 – I: **Global Interrupt Enable,** It controls whether the AVR microcontroller can process **interrupts**.

```
CLI (Clear Interrupts),
SEI (Set Interrupts)
```

Bit 6 – T: Acts as a **temporary storage bit** for bit manipulation

```
BST R16, 2 ; Store bit 2 of R16 into T bit
BLD R17, 5  ; Load T bit into bit 5 of R17
```

Bit 5 – H Half Carry Flag: Indicates a half-carry condition between two nibbles in BCD operations (rarely used directly).

Bit 4 – S Sign Flag: Set if signed overflow or negative result is detected (computed as $S = N \oplus V$). It Occurs in an addition where overflow happens, S = 1.

Bit 3 – V: **Two's Complement Overflow Flag**: Set if overflow occurs in signed arithmetic operations. It indicates an **overflow** condition that occurs in **signed arithmetic operations**

The **V bit** is set (V = 1) when an **overflow** occurs in a signed arithmetic operation (addition or subtraction),

```
LDI R16, 127    ; Load 127 into R16 (maximum positive value)
LDI R17, 1      ; Load 1 into R17
ADD R16, R17    ; Add R16 and R17: 127 + 1 = 128
```

In this case, the result **overflows** into the negative range (in two's complement), so the **V flag** will be set.

```
LDI R16, -128   ; Load -128 into R16 (minimum negative value)
LDI R17, -1     ; Load -1 into R17
SUB R16, R17    ; Subtract R17 from R16: -128 - (-1) = -129
```

In this case, the result **overflows** into the positive range, so the **V flag** will be set.

Bit 2 – N: Negative Flag: Set if the result of the operation is negative. After subtracting two numbers, if the result is negative, N = 1

Bit 1 – Z: Zero Flag: Set if the result of the operation is zero. CP (Compare) instruction that results in zero will set Z.

Bit 0 – C: Carry Flag: Set if a carry-out or borrow occurs in an arithmetic operation. After ADD for example, if there's a carry, C = 1

Examples of status control operations

Example 8: Carry Flag (C bit)

```
LDI R16, 0xFF ; Load R16 with 255 (max 8-bit value)
LDI R17, 1 ; Load R17 with 1
ADD R16, R17 ; 255 + 1 = 0 (Carry occurs)
BRCS carry_set ; Branch if Carry Set

carry_set:
```

Changed Flags:

- C = 1 (Carry occurred)
- Z = 1 (Result is zero)

- N = 0 (Not negative)
- V = 0 (No overflow)

Example 7: Zero Flag (Z bit)

```
LDI R16, 5        ; Load R16 with 5
LDI R17, 5        ; Load R17 with 5
SUB R16, R17      ; Subtract 5 - 5 = 0
BREQ equal        ; Branch if result is zero
equal:
```

Changed Flags:

- Z = 1 (Result is zero)
- N = 0 (Not negative)
- V = 0 (No overflow)
- C = 0 (No borrow)

Example 6: Negative Flag (N bit)

```
LDI R16, -1       ; Load -1 (11111111)
LDI R17, 1        ; Load 1
ADD R16, R17      ; -1 + 1 = 0
BRPL positive     ; Branch if result is positive
positive:
```

Changed Flags:

- N = 0 (Positive result)
- Z = 1 (Zero result)
- C = 0 (No carry)

Example 5: Two’s Complement Overflow (V bit)

```
LDI R16, 127      ; Load R16 with max positive value (01111111)
LDI R17, 1        ; Load R17 with 1
```

```
ADD R16, R17      ; Add 127 + 1 (Overflow!)
BRVS overflow     ; Branch if Overflow Set
overflow:
```

Changed Flags:

- V = 1 (Overflow occurred)
- C = 0 (No carry)
- N = 1 (Negative result)
- Z = 0 (Result is not zero)
- S = 1 (Negative due to overflow)

Example 4: Sign Flag (S bit)

```
LDI R16, -5       ; Load R16 with -5
LDI R17, -10      ; Load R17 with -10
ADD R16, R17      ; Add (-5) + (-10) = -15
BRMI negative     ; Branch if the result is negative
negative:
```

Changed Flags:

- S = 1 (Result is negative)
- N = 1 (Negative flag set)
- V = 0 (No overflow)
- Z = 0 (Result is not zero)

Example 3: Half Carry Flag (H bit)

```
LDI R16, 0x0F     ; Load R16 with 0x0F (00001111)
LDI R17, 0x01     ; Load R17 with 0x01 (00000001)
ADD R16, R17      ; Add R16 and R17 (0x0F + 0x01 = 0x10)
RHC skip         ; Branch if Half Carry flag is clear
skip:
```

Changed Flags:

- `H = 1` (Half carry occurred)
- `C = 0` (No full carry)
- `Z = 0` (Result is not zero)
- `N = 0` (Positive result)

Example 2: Bit Copy Storage (T bit)

```
LDI R16, 0b00001000 ; Load R16 with binary 00001000
BST R16, 3       ; Store bit 3 of R16 into T
BLD R17, 0      ; Load bit T into bit 0 of R17
```

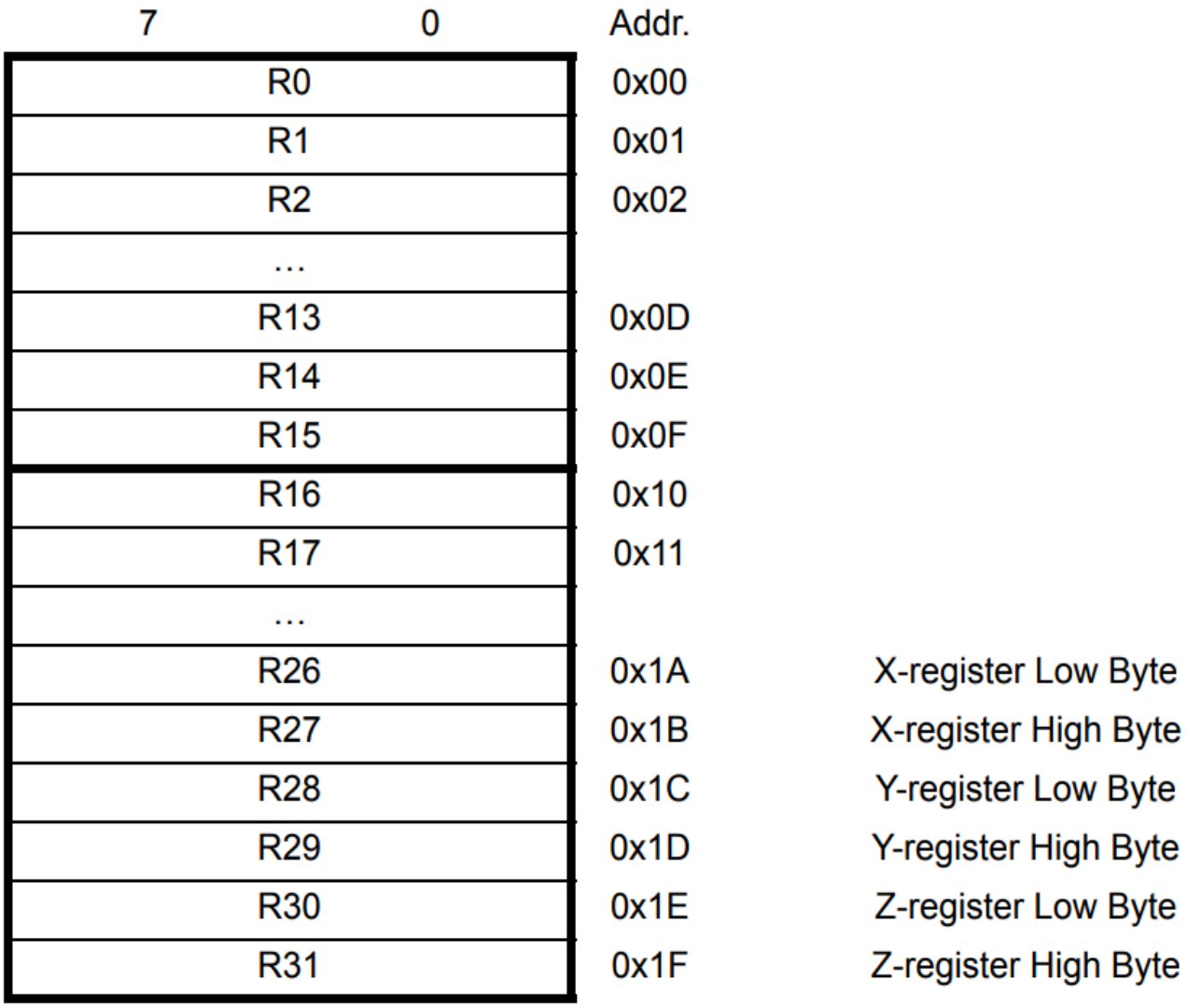
Changed Flag: `T` (Bit Copy Storage)

Example 1: Global Interrupt Enable (I bit)

```
SEI   ; Set Global Interrupt Enable (I = 1)
NOP   ; Do nothing (interrupts are now enabled)
CLI   ; Clear Global Interrupt Enable (I = 0)
```

Changed Flag: `I` (Interrupt Enable)

2.3.3. AVR CPU General Purpose Working Registers



The **AVR CPU architecture**, including the **ATmega328P**, features a fast and efficient set of **32 General Purpose Working Registers** (named **R0 to R31**) at the core of its **8-bit RISC (Reduced Instruction Set Computer)** design.

R0–R31 General-purpose 8-bit registers used for all arithmetic, logic, and data transfer operations.

These registers are directly connected to the **Arithmetic Logic Unit (ALU)**, enabling **single-cycle instruction execution** for most operations.

AVR allows combining some registers into **16-bit register pairs** for operations like pointer handling and indirect addressing:

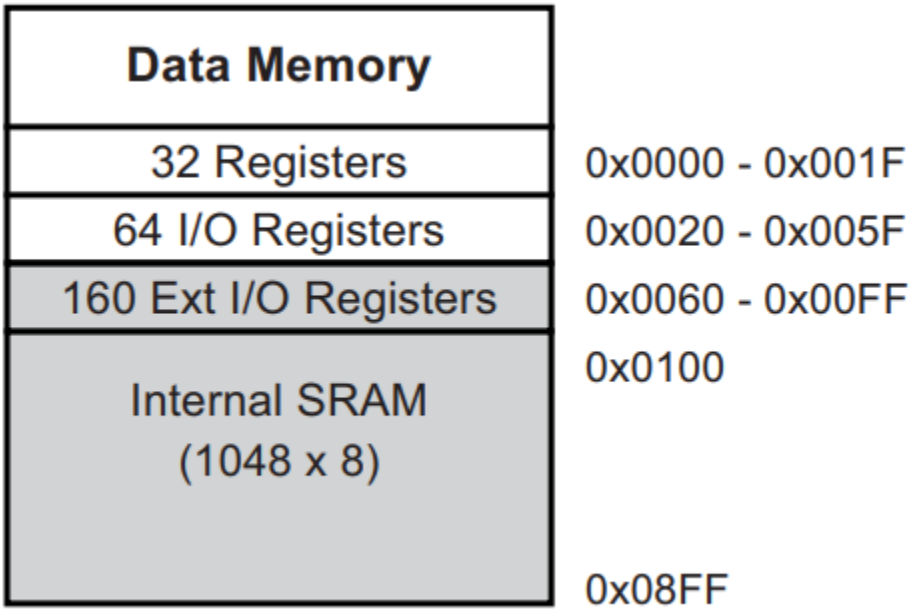
16-bit Name	Register Pair	Usage
X	R27:R26	Indirect addressing (e.g., `LD`, `ST` instructions)
Y	R29:R28	Indirect addressing with optional displacement
Z	R31:R30	Indirect addressing with optional displacement; also used for function calls/returns

Little-endian order: Low byte comes first. For example, `X = R26 (low) + R27 (high)`

Some instructions use or affect specific registers:

- R1 is often cleared to 0 and used as a "zero register."
- R0 and R1 may be clobbered during some interrupt operations or function prologues.
- Register pairs (R26–R31) are often reserved for pointer use in C compilers.

2.3.4. AVR CPU SRAM Data Memory



The **AVR architecture** organizes its **Data Memory** into a **linear address space**, combining several memory regions:

Address Range	Region	Description
`0x0000 – 0x001F`	**Register File**	32 General Purpose Registers (R0–R31)

`0x0020 – 0x005F`	**I/O Registers**	64 memory-mapped I/O registers
`0x0060 – 0x00FF`	**Extended I/O Registers**	160 extended I/O registers
`0x0100 – end`	**SRAM (Data SRAM)**	Actual read/write memory for program data

1. General Purpose Registers (R0–R31)

- Directly connected to the ALU
- Located at the very beginning of data memory (`0x0000 – 0x001F`)

2. I/O Registers

- Control peripherals like timers, ADCs, serial interfaces, etc.
- Accessible using special `IN`/`OUT` instructions or normal data memory access

3. SRAM (Static RAM)

- Used for:
    - SRAM Size; **2 KB (2048 bytes)**
    - **SRAM Address Range 0x0100 – 0x08FF**
    - Used for:
        - Global and local variables
        - Stacks (function calls, return addresses, etc.).

- Dynamic memory (in languages like C)

4. Stack

- Grows **downward** in SRAM (from higher to lower addresses)
- Controlled by the **Stack Pointer (SP)**
- Used for subroutine return addresses, local variables, and context saving
- Usually set to `0x08FF` by default (top of SRAM)

Stack Pointer

The stack is mainly used for storing local variables and for storing return addresses after interrupts and subroutine calls.

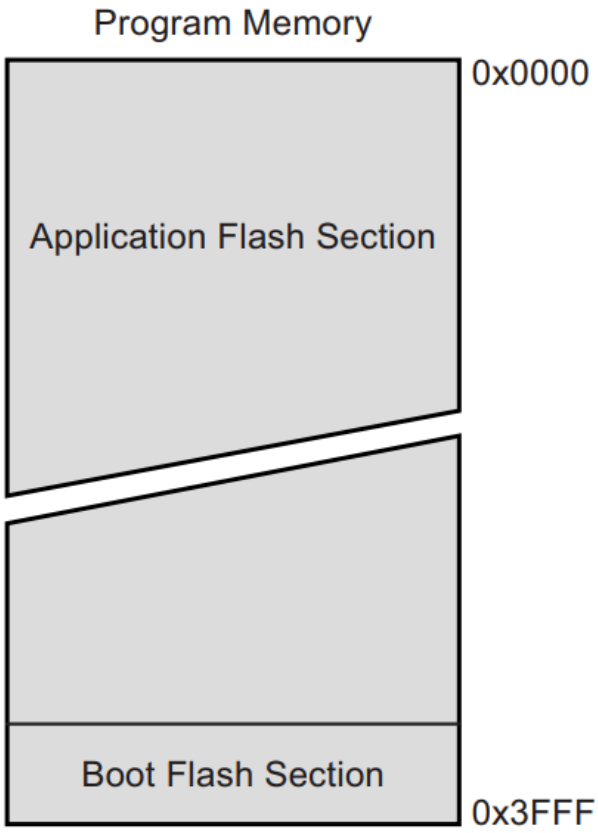
The stack pointer register always points to the top of the stack.

The stack in the data SRAM must be defined by the program before any subroutine calls are executed or interrupts are enabled using SPH and SPL – Stack Pointer High and Stack Pointer Low Register

Instruction	Stack pointer	Description
PUSH	Decremented by 1	Data is pushed onto the stack
CALL ICALL RCALL	Decremented by 2	Return address is pushed onto the stack with a subroutine call or interrupt
POP	Incremented by 1	Data is popped from the stack
RET RETI	Incremented by 2	Return address is popped from the stack with return from subroutine or return from interrupt

2.3.5. AVR Program Memory

The **AVR microcontroller** uses **Flash memory** as its **Program Memory**, where the compiled machine code (instructions) is stored. This memory is **non-volatile**, meaning it retains contents even when power is removed.



Due to the **Harvard architecture**, Flash and SRAM are separate, and data in Flash cannot be accessed like normal RAM. Instead, AVR provides:

- `LPM` (Load Program Memory): Used to read constant data from Flash into registers
- `SPM` (Store Program Memory): Used for self-programming, typically by the bootloader

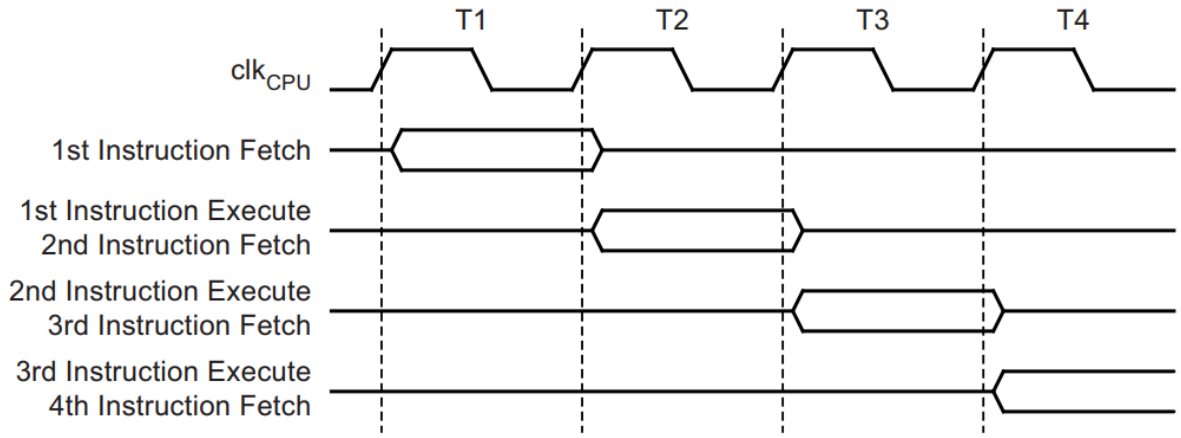
Flash stores:

- Your compiled C/assembly code
- `const` variables (e.g., lookup tables, strings)

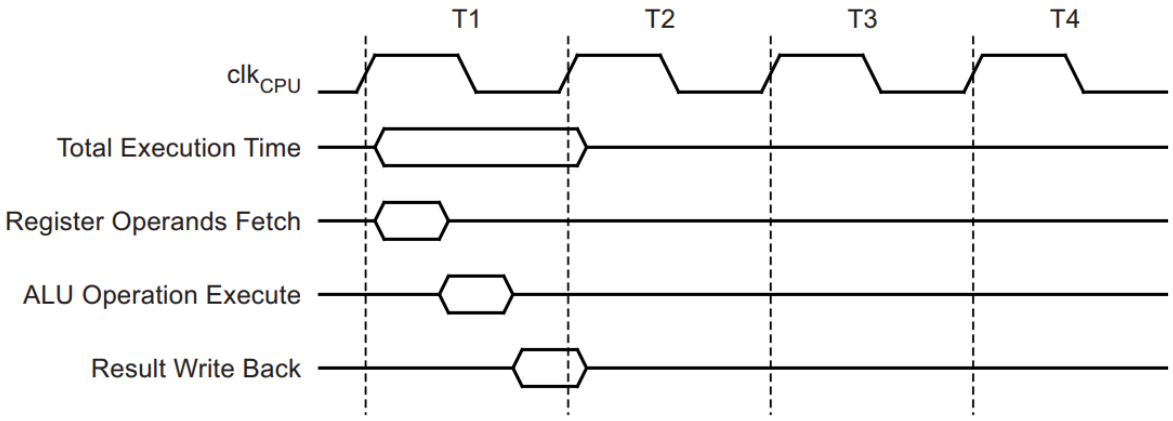
AVR microcontrollers allow a **bootloader** to be programmed into a reserved section of program memory:

- The bootloader can self-program the rest of the Flash.
- Useful for updating firmware without external programmers.
- Configured using **fuse bits**.

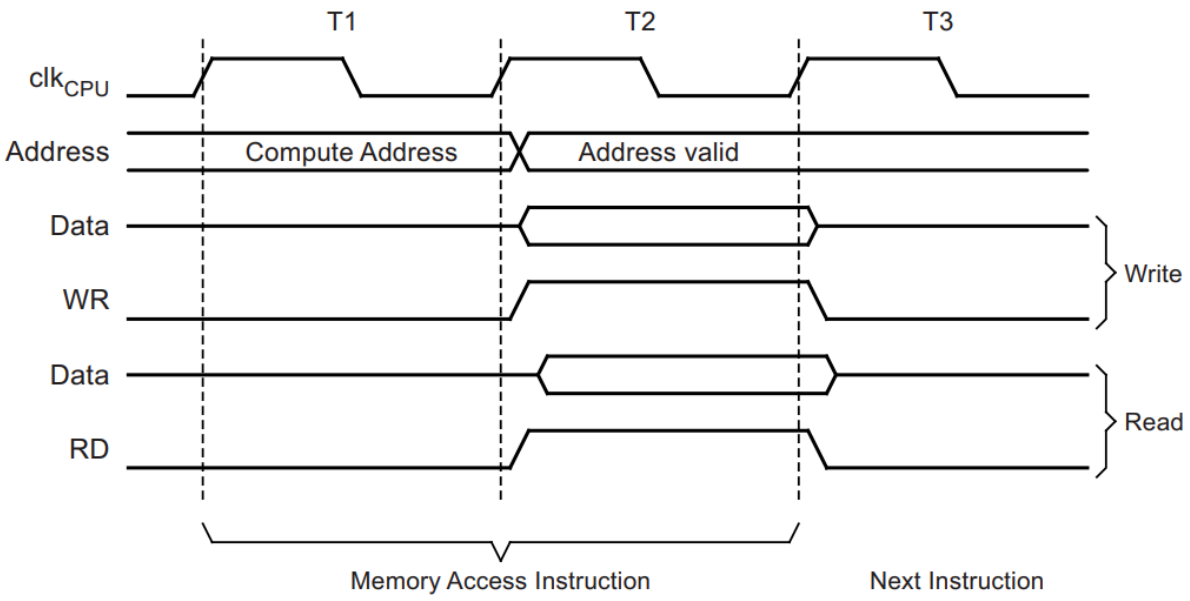
Instruction Execution Timing



Single Cycle ALU Operation



Data Memory Access Time. Ex. On-chip Data SRAM Access Cycles



2.4. Assembly Programming in AVR CPU

2.4.1. Working With Registers R0-R31

Mnemonic	Description
ldi	load immediate
mov	copy register
movw	copy register pair

```
ldi   r16,85      ; load the value 85 into register 16
ldi   r16,0x55    ; load 85 into register 16 (hex)
mov   r0,r16      ; copy the contents of r16 into r0
movw  r1:r0,r17:r16    ; copy the contents of r17:r16 into r1:r0
movw  r0,r16           ; copy the contents of r17:r16 into r1:r0
```

2.4.2. Logical Instructions

Mnemonic	Description
and	logical AND
andi	logical AND with immediate
or	logical OR
ori	logical OR with immediate
eor	exclusive OR
com	one's complement
neg	two's complement

```
ldi r16,0x55; load 0x55 into r16
andir16,0x0F; mask the upper 4 bits of r16 (result
= 0x05)
ldi r17,0x00; load 0x00 into r17
ori r17,0x0F; set lower 4 bits of r17 (result=0x0F)
and r17,r16
ldi r16,0x55; load 0x55 into r16
com r16; one's complement of r16 (result = 0xAA)
```

2.4.3. Arithmetic Instructions

Mnemonic	Description
add	add without carry
adc	add with carry
adiw	add immediate to word
sub	subtract without carry
subi	subtract immediate
sbc	subtract with carry
sbci	subtract immediate with carry
sbiw	subtract immediate from word
inc	increment
dec	decrement
mul	multiply unsigned(1)
muls	multiply signed(1)
mulsu	multiply signed with unsigned(1)
fmul	fractional multiply unsigned(1)
fmuls	fractional multiply signed(1)
fmulsu	fractional multiply signed with unsigned(1)

```
ldi     r16,0x34        ; place lower byte of 0x1234 in r16
ldi     r17,0x12        ; place upper byte of 0x1234 in r17
ldi     r18,0xCD        ; place lower byte of 0xABCD in r18
ldi     r19,0xAB        ; place upper byte of 0xABCD in r19

add     r16,r18         ; compute sum of lower bytes (result = 0x01)
adc     r17,r19         ; compute sum of upper bytes with carry (result = 0xBE)
```

```
ldi     r24,0x00        ; load 0x1000 into
ldi     r25,0x10        ; registers r24:r25

adiw    r24,0x0A        ; add 0x0A to r24:r25 (result = 0x100A)

ldi     XL,0x80         ; load 0x8080 into
ldi     XH,0x80         ; X pointer

adiw    X,1             ; increment X pointer (result = 0x80801)
```

2.4.4. Bit Shifts

Mnemonic	Description
lsl	logical shift left
lsr	logical shift Right
rol	rotate left through carry
ror	rotate right through carry
asr	arithmetic shift right

```
ldi     r16,0b10101010  ; load 1010101010 into r16
lsl     r16             ; shift r16 left (result = 01010100)
lsr     r16             ; shift r16 right (result = 00101010)
```

```
ldi     r16,0b10101010  ; load 1010101010 into r16
ldi     r17,0b10101010  ; load 1010101010 into r17

lsl     r16             ; shift r16 left (result = 01010100)
rol     r17             ; shift r17 left and bring carry (result = 01010101)
```

```
ldi     r16,0b10101010  ; load 1010101010 into r16
ldi     r17,0b10101010  ; load 1010101010 into r17

lsr     r17             ; shift r17 right (result = 01010101)
ror     r16             ; shift r16 right and bring carry (result = 01010101)
```

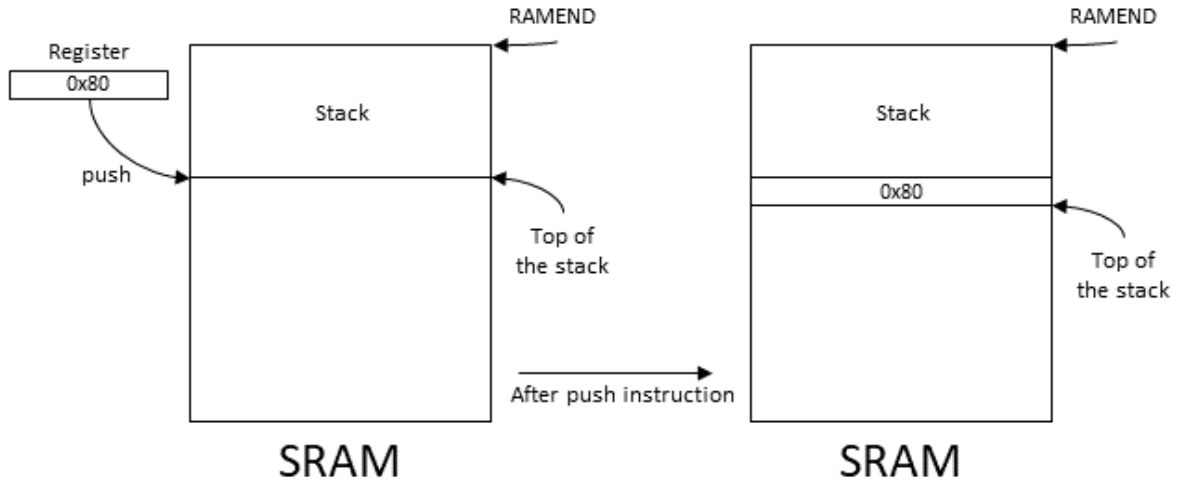
2.4.5. Subroutines and The Stack Pointer

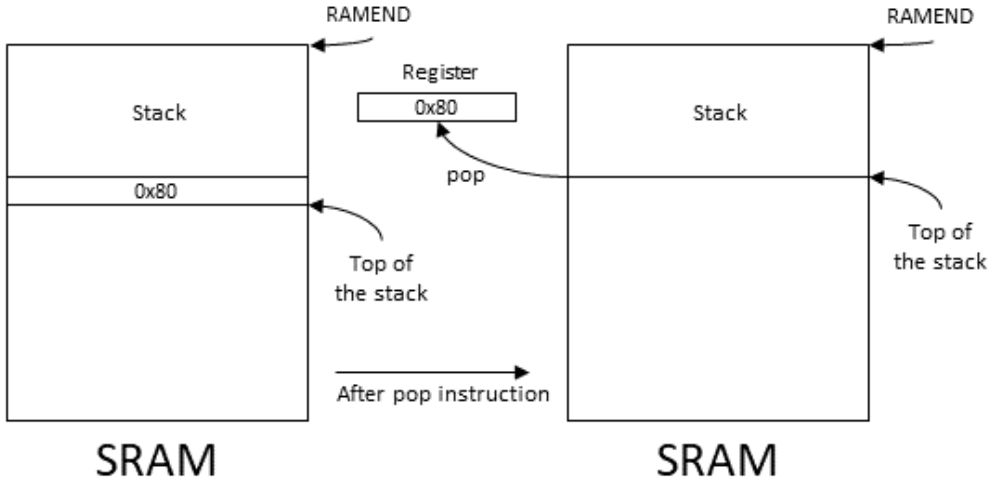
The Stack Pointer is a 16-bit register in I/O Memory defined in include files as SPH and SPL.that points to space allocated in SRAM (The Stack). The Stack is used to temporarily store register values and return addresses when subroutines are called. Typically, the stack begins at the end of SRAM,

In old AVR microcontroller it should be initialized manually (RAMEND is a constant defined in the include file as the last address in SRAM)

```
ldi     r16,LOW(RAMEND)         ; load low byte of RAMEND into r16
out     SPL,r16                 ; store r16 in stack pointer low
ldi     r16,HIGH(RAMEND)        ; load high byte of RAMEND into r16
out     SPH,r16                 ; store r16 in stack pointer high
```

Registers can be saved or loaded to the stack - referred to as pushing or popping.





When pushing multiple registers to the stack, pop instructions must be called in reverse order to restore values to their original registers, i.e.

```
push    r0                  ; push contents of r0 to stack
push    r1                  ; push contents of r1 to stack
push    r2                  ; push contents of r2 to stack

pop     r2                  ; restore contents of r2
pop     r1                  ; restore contents of r1
pop     r0                  ; restore contents of r0
```

Subroutines

Subroutines are sequences of code that can be reused at any point in a program.
When a subroutine is called, the microcontroller will push a return address to the stack. It will then jump to the location of the subroutine and execute the code there.
When a return statement is reached, the return address will be popped from the stack and the microcontroller will jump to the instruction immediately following the subroutine call.

Mnemonic	Description
call	long call to subroutine
icall	indirect call to subroutine
rcall	relative call to subroutine
ret	return from subroutine

```
        ldi     r16,0x01                ; load r16 with 0x01
        ldi     r17,0x02                ; load r17 with 0x02

        call    addReg                  ; call subroutine

loop:   rjmp    loop                    ; infinite loop

addReg:
        add     r16,r17                 ; add r16 and r17

        ret                             ; return from subroutine
```

The instruction call is used with the label of our subroutine addReg. Using the call instruction will force the microcontroller to jump to the label given, execute the code there, and when the instruction ret is reached, jump back to the instruction immidiately following call - in this case the infinite loop we have setup with rjmp.

Note that we cannot explicitly pass parameters to a subroutine like with functions in C. In the example above, the subroutine expects its parameters to already be in registers r16 and r17.

Subroutines can be called from the entire program space using **call**, or relatively over distances of up to 4K words using **rcall**.

Program execution continues at address PC + k + 1. The relative address k is from -2048 to 2047. (K is the relative address)

```
        rcall   doSomething             ; call subroutine doSomething
        ...                             ; other program code

doSomething:
        ...                             ; subroutine code
        ret                             ; return from subroutine
```

icall ijumps to the address of the subroutine loaded in the Z pointer.

Indirect calls are useful when different subroutines must be called depending on runtime parameters.

This would usually be implemented with a lookup table, rather than explicitly loading parameters with **ldi** as shown above.

```
        ldi     ZL,LOW(doSomething)     ; load address of doSomething
        ldi     ZH,HIGH(doSomething)    ; in Z pointer

        icall                           ; indirect call to doSomething

doSomething:
        ...                             ; subroutine code
        ret                             ; return from subroutine
```

Saving Registers

When a subroutine is called, it may modify registers that you need later in the program. To prevent this from happening, registers can be pushed to the stack at the beginning of the subroutine, and popped back again at the end.

Passing arguments into subroutine

Passing arguments through registers

In the following program the BIGGER function gets two values through R21 and R22. After comparing R21 and R22, it returns the bigger value through R21.

parameters: R21 and R22: the values to be compared

returns: ; R21: containing the bigger value

```
        LDI     R21,5   ; R21 = 5
        LDI     R22,7   ; R22 = 7
        CALL    BIGGER
HERE:
    RJMP HERE


BIGGER:
        CP      R21, R22
        BRSH    L1
        MOV     R21, R22
L1:     RET
```

Passing through memory using references

We can store the data in memory and pass its address through a register. In the following program, the address of a string is passed to the function through the Z register. The string ends with 0. The function puts the contents of the string on PORTB until it reaches 0.

```
MYDATA: .DB "Hello World",0         ;a zero ended string

        LDI     ZL,LOW(MYDATA<<1)
        LDI     ZH,HIGH(MYDATA<<1)
        CALL    MY_FUNC
HERE: RJMP HERE
```

```
MY_FUNC:
        LPM     R20,Z+
        CPI     R20,0           ;is 0 ?
        BREQ    L_END           ;return if it is 0
        OUT     PORTB,R20
        RJMP    MY_FUNC
L_END:  RET             ;return
```

Passing arguments through stack

Passing through the stack is a flexible way of passing arguments. To do so, the arguments are pushed onto the stack just before calling the function and popped off after returning.

This method of passing arguments is used in x86 computers because they have very few general purpose registers. In AVR CPU,

the arguments are passed in registers. If the arguments are more than registers, the rest are passed on the stack. It is important to remember that after returning from the call, the caller must clear the arguments on the stack.

2.4.6. Addressing Modes

Addressing modes determine how the CPU accesses data in memory or registers when executing instructions. The AVR instruction set supports several addressing modes to provide flexibility and efficiency.

Mode	Description	Example
Immediate	Operand is part of the instruction	`LDI R16, 0x20` (Load 0x20 into R16)

Direct	Access I/O or SRAM using a fixed address	`LDS R20, 0x0060` (Load from address 0x60)
Indirect	Address is held in a register pair (X, Y, Z)	`LD R18, X` (Load from address in X = R27:R26)
Indirect with Displacement	Access memory at base register + offset (only with Y or Z)	`LD R18, Y+5`
Indirect with Pre/Post-Increment/Decrement	Automatically modify pointer before or after access	`LD R18, X+` or `ST -Z, R19`
Register Direct	Operands are in general-purpose registers	`ADD R16, R17`
I/O Direct	Access special I/O registers using addresses 0–63	`OUT 0x1B, R16` (Write to PORTA)
Relative	Used in branching instructions (PC-relative addressing)	`RJMP +4` (Relative jump forward 4 instructions)

2.4.7. Accessing I/O registers

These instructions are used to access the I/O space, I/O Registers, but not Extended I/O Registers.

The I/O registers can be accessed using

```
in Rd, PORTADDRESS
```

```
out PORTADDRESS, Rs
Rd, Rs: any register from register file.
```

PORTADDRESS : any register from the entire range of 0x00 to 0x3F (a total of 64 I/O registers).

Example

Ex

```
IN R10, PINB
```

if the input to PORTB is 0x03 and all the pins are configured as an inputs, R10 becomes 0x03

```
OUT PORTC, R1
```

The contents of R1 are used to set the output register of PORTC

2.4.8. Accessing SRAM

Loading and Storing Direct

lds - load direct from data space
sts - store direct to data space.

Example:

```
        .dseg
        .org    SRAM_START
var:    .byte   2

        ...

        ldi     r16,0xAA                ; load r16 with 0xAA
        ldi     r17,0x55                ; load r17 with 0x55

        sts     var,r16                 ; store 0x55AA in
        sts     var+1,r17               ; var

        lds     r0,var                  ; load var into
        lds     r1,var+1                ; r1:r0
```

.dseg: data segment

.byte 2 allocates 2 bytes for the address var

Loading and Storing Indirect

We can access Data Memory by loading an address in one of X; Y and Z registers that will be used like pointers to access data memory

ld - load indirect from data space
st - store indirect to data space.

Example

```
        .dseg
        .org    SRAM_START
var:    .byte   1

        ...

        ldi     r16,0xFF                ; load 0xFF into r16

        ldi     XL,LOW(var)             ; initialize X pointer
        ldi     XH,HIGH(var)            ; to var address

        st      X,r16                   ; store r16 to var

        ld      r0,X                    ; load r0 with var
```

.dseg data segment

.byte 2 allocates 2 bytes for the address var

2.4.9. Loading Data from Program Memory

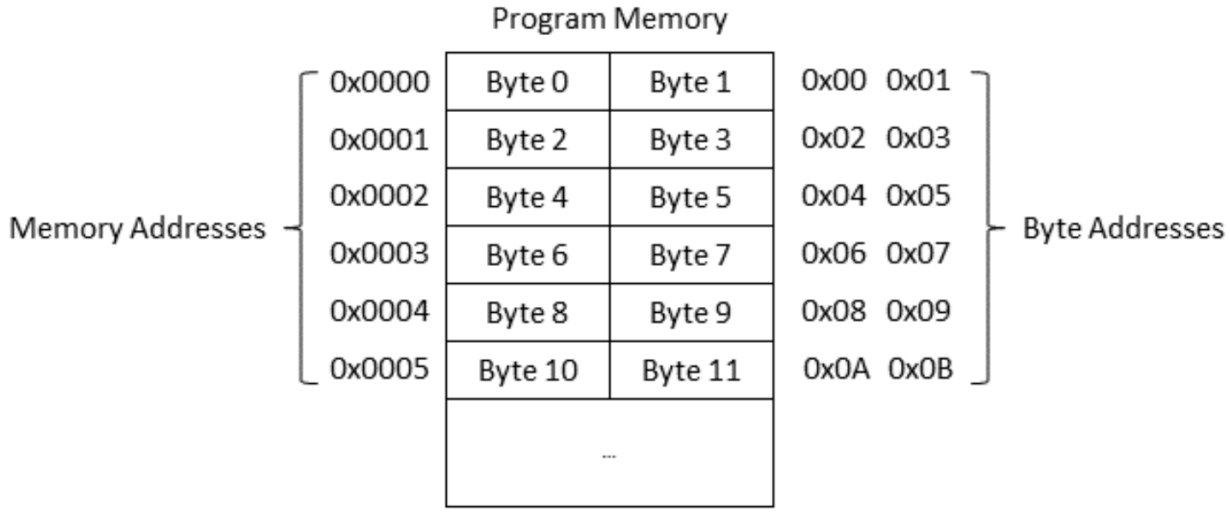
Loading and Storing Indirect

Loading data from Program Memory can only be done indirect using the Z pointer. This is shown in the example below

```
        ldi     ZL,LOW(2*var)           ; load 2*var
        ldi     ZH,HIGH(2*var)          ; into Z pointer

        lpm     r4,Z                    ; load pmem var into r4

var:    .db     3
```

.db 2 put the constant value 3 in 1 the program address var

The used address is **2*var** because the program memory is a 16-bit addressed memory as depicted in the figure bellow



CHAPTER 3 Interrupts and Events in Embedded Systems

1. Introduction

Interrupts provide a powerful mechanism for handling asynchronous events in embedded systems. Rather than continuously polling for changes, interrupts allow the AVR microcontroller to respond immediately to specific conditions, improving both efficiency and responsiveness. This chapter introduces the concept of interrupts and their implementation in the AVR architecture. It covers the types of interrupts available—external, timer, and peripheral—as well as the structure of interrupt vectors, priority handling, and enabling/disabling mechanisms. Key registers involved in interrupt control and execution are discussed, alongside the use of Interrupt Service Routines (ISRs). Practical examples demonstrate how to configure and use interrupts effectively in real-time applications. By the end of this chapter, readers will understand how to design interrupt-driven systems for enhanced performance and reliability.

2. Interrupts

Interrupts are hardware- or software-triggered signals that pause the current program to execute a special routine called an Interrupt Service Routine (ISR). Once the ISR finishes, the microcontroller resumes normal execution.

This allows the AVR microcontroller to respond quickly to events like a timer overflow, external pin change, or received serial data.

- Allow the program to respond to events when they occur
- Allow the program to ignore events until they occur
- Events could be external or internal events

Event sources of interrupts could be:

- External events:
    - Signal change on pin: when the signal on a specific pin changes state (HIGH to LOW or LOW to HIGH).
    - Action depends on context: Sleep mode can be woken up by external pin interrupts
    - UART ready with/for next character: This refers to interrupt-based UART

communication,

- Internal events
    - Power failure: The power supply voltage drops below a safe threshold, triggering a Brown-Out Reset (hardware event)
    - Timer “tick”: a periodic interrupt generated by a hardware timer

Types of Interrupts in ATmega328P are:

Source	Example Trigger
External Interrupts	Change on INT0 or INT1 pin
Pin Change	Any change on PCINT0–23 (digital pins)
Timer/Counter	Overflow, compare match, capture
USART	Receive complete, transmit complete
ADC	ADC conversion complete
TWI (I2C)	TWI events (start, stop, transmit)
SPI	SPI transfer complete
Watchdog Timer	Watchdog timeout

2.1. Interrupt Vector and ISR

In ATmega328P, the Interrupt Vector Table (IVT) starts at address 0x0000 in program memory. Each interrupt has a fixed memory address where the microcontroller jumps when an interrupt occurs.

When an interrupt event occurs:

- Processor does an automatic procedure call
- CALL automatically done to address for that interrupt
    - Push current PC, Jump to interrupt address
- Each event has its own interrupt address
- The global interrupt enable bit (in SREG) is automatically cleared

    - i.e. nested interrupts are disabled
    - SREG bit can be set to enable nested interrupts if desired

Interrupt procedure ISR (interrupt handler)

- Does whatever it needs to, then returns via RETI
- The global interrupt enable bit is automatically set on RETI
- One program instruction is always executed after RETI: if another interrupt occurs immediately after RETI, the next instruction after RETI will always execute before jumping to the new ISR.

Global interrupt enable should be set to allow the CPU to respond the interrupt:

- Bit in SREG
- Allows all interrupts to be enabled/disabled with one bit
- SEI in asm or sei() in C – set the bit to enable global interrupts
- CLI in asm cli() – clear the bit to disable global interrupts

Interrupt priority is determined by order in table where Lower addresses have higher priority

Interrupt routine

An Interrupt Service Routine (ISR) is a special function that gets automatically executed when a specific interrupt event occurs.

In AVR C programming using AVR-GCC, an ISR is defined like this:

```
ISR(<interrupt_vector_name>) {
    //Code to run when the interrupt occurs
}
```

reti() – return from interrupt isautomatically generated for ISR

`ISR` is a macro defined in `<avr/interrupt.h>`

Interrupt vector

Table in memory containing the first instruction of each interrupt handler. Typically, it starts at program address 0

Interrupt Source	Vector Name	Description
Reset	`RESET`	Power-on or external reset
INT0	`INT0_vect`	External Interrupt Request 0
INT1	`INT1_vect`	External Interrupt Request 1
PCINT0	`PCINT0_vect`	Pin Change Interrupt 0
TIMER0 OVF	`TIMER0_OVF_vect`	Timer/Counter0 Overflow
ADC	`ADC_vect`	ADC Conversion Complete
USART RX	`USART_RX_vect`	USART Receive Complete
SPI	`SPI_STC_vect`	SPI Serial Transfer Complete
...	...	...

Full list available in the ATmega328P datasheet.

```
Address  Labels Code                 Comments
0x0000          jmp    RESET        ; Reset Handler
0x0002          jmp    EXT_INT0     ; IRQ0 Handler
0x0004          jmp    EXT_INT1     ; IRQ1 Handler
0x0006          jmp    PCINT0       ; PCINT0 Handler
0x0008          jmp    PCINT1       ; PCINT1 Handler
0x000A          jmp    PCINT2       ; PCINT2 Handler
0x000C          jmp    WDT          ; Watchdog Timer Handler
0x000E          jmp    TIM2_COMPA   ; Timer2 Compare A Handler
0x0010          jmp    TIM2_COMPB   ; Timer2 Compare B Handler
0x0012          jmp    TIM2_OVF     ; Timer2 Overflow Handler
0x0014          jmp    TIM1_CAPT    ; Timer1 Capture Handler
0x0016          jmp    TIM1_COMPA   ; Timer1 Compare A Handler
0x0018          jmp    TIM1_COMPB   ; Timer1 Compare B Handler
0x001A          jmp    TIM1_OVF     ; Timer1 Overflow Handler
0x001C          jmp    TIM0_COMPA   ; Timer0 Compare A Handler
0x001E          jmp    TIM0_COMPB   ; Timer0 Compare B Handler
```

If interrupts are not used, this memory can be used as part of the program, i.e. nothing special about this part of memory.

In AVR-C interrupt vectors are defined as macro

```
#define INT0_vect          _VECTOR(1)    /* External Interrupt Request 0  */
#define INT1_vect          _VECTOR(2)    /* External Interrupt Request 1  */
#define PCINT0_vect        _VECTOR(3)    /* Pin Change Interrupt Request  0 *
#define PCINT1_vect        _VECTOR(4)    /* Pin Change Interrupt Request  0 *
#define PCINT2_vect        _VECTOR(5)    /* Pin Change Interrupt Request  1 *
#define WDT_vect           _VECTOR(6)    /* Watchdog Time-out Interrupt */
#define TIMER2_COMPA_vect  _VECTOR(7)    /* Timer/Counter2 Compare Match A */
#define TIMER2_COMPB_vect  _VECTOR(8)    /* Timer/Counter2 Compare Match A */
#define TIMER2_OVF_vect    _VECTOR(9)    /* Timer/Counter2 Overflow */
#define TIMER1_CAPT_vect   _VECTOR(10)   /* Timer/Counter1 Capture Event */
#define TIMER1_COMPA_vect  _VECTOR(11)   /* Timer/Counter1 Compare Match A */
#define TIMER1_COMPB_vect  _VECTOR(12)   /* Timer/Counter1 Compare Match B */
#define TIMER1_OVF_vect    _VECTOR(13)   /* Timer/Counter1 Overflow */
#define TIMER0_COMPA_vect  _VECTOR(14)   /* TimerCounter0 Compare Match A */
#define TIMER0_COMPB_vect  _VECTOR(15)   /* TimerCounter0 Compare Match B */
#define TIMER0_OVF_vect    _VECTOR(16)   /* Timer/Couner0 Overflow */
#define SPI_STC_vect       _VECTOR(17)   /* SPI Serial Transfer Complete */
#define USART_RX_vect      _VECTOR(18)   /* USART Rx Complete */
#define USART_UDRE_vect    _VECTOR(19)   /* USART, Data Register Empty */
#define USART_TX_vect      _VECTOR(20)   /* USART Tx Complete */
#define ADC_vect           _VECTOR(21)   /* ADC Conversion Complete */
#define EE_READY_vect      _VECTOR(22)   /* EEPROM Ready */
#define ANALOG_COMP_vect   _VECTOR(23)   /* Analog Comparator */
#define TWI_vect           _VECTOR(24)   /* Two-wire Serial Interface */
#define SPM_READY_vect     _VECTOR(25)   /* Store Program Memory Read */
```

Event types

In AVR microcontrollers, some events are latched (remembered) even if interrupts are disabled, while others are lost.

Type 1 – Event is remembered when interrupt is disabled

- If interrupt is not enabled, flag is set, When interrupt is enabled again, interrupt takes place, and flag is reset
- Ex. External Interrupts (INT0, INT1): If a signal change (e.g., falling edge) occurs on INT0, but the interrupt is disabled, the event is latched. When the interrupt is enabled again, the ISR triggers immediately.

Type 2 – Event is not remembered when interrupt is disabled

- If the interrupt is disabled, the event is not recorded
- Ex. Pin Change Interrupt (PCINTx): If a pin change occurs while its interrupt is

disabled, the event is lost. Once the interrupt is re-enabled, the ISR will not trigger for the missed change

2.2. Interrupt Handling

The basic interrupt flow is:

1. Interrupt occurs
2. AVR saves program counter (PC) on the stack
3. AVR jumps to the ISR
4. ISR runs
5. AVR restores PC from the stack and resumes

Interrupt hander is invisible to program When an interrupt occurs, the CPU automatically saves the program state and jumps to the Interrupt Service Routine (ISR). The main program does not explicitly call the ISR—it is invoked by the hardware.

Since the main program doesn't directly call the ISR, the only way to detect an interrupt's execution is through side-effects ,ex. Setting flags (volatile variables)

When an interrupt occurs, it pauses the main program, executes the ISR, then resumes execution. This introduces timing variations in the main program, because

Without interrupts, you could precisely calculate execution time. With interrupts, execution time becomes unpredictable, because an ISR might execute at any time,

Save and restore any registers used

If an ISR modifies a register used by the main program and does not restore it, the main program will see incorrect values, leading to unexpected behavior

Save and restore any registers usedin the ISR including SREG

2.3. External Interrupts

External interrupts are hardware-triggered signals that originate from outside the microcontroller and cause it to temporarily pause its main execution to respond to an event on a specific input pin.

In the ATmega328P, external interrupts are typically triggered by:

- INT0 and INT1 - triggered by changes in voltage level or edge (rising/falling) on

INT0 (PD2) and INT1 (PD3) pins.

- PCINT[23:0] – triggered by any signal change (toggle): In the ATmega328P, 23 pins can trigger a Pin Change Interrupt (PCINT). These pins are divided into three groups (PCINT0-7, PCINT8-14, PCINT16-23), each associated with a specific interrupt vector. Where:
    - PCINT[7:0] – PORT B [7:0]
    - PCINT[14:8] – PORT C [6:0]
    - PCINT[23:16] – PORT D [7:0]

Pulses on inputs must be slower than I/O clock rate: This means that the input signal must remain stable (HIGH or LOW) for at least one I/O clock cycle to be reliably detected by the microcontroller

2.3.1. INT0 and INT1

INT0 and INT1 are triggered by changes in voltage level or edge (rising/falling) on INT0 (PD2) and INT1 (PD3) pins. They are controlled by External Interrupt Control Register: EICRA, where:

- ISC01, IS00 controls INT0
- ISC11, IS10 controls INT1

External Interrupt Control Register

Bit	7	6	5	4	3	2	1	0	
(0x69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 12-1. Interrupt 1 Sense Control

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Interrupt configured with Pin is LOW (Continuous Until Pin Goes HIGH). This means that an interrupt is triggered continuously as long as the pin remains in a LOW (0V) state. The interrupt stops when the pin goes HIGH (1). If the pin stays LOW, the ISR will keep executing (or be re-triggered immediately after returning). Once the pin goes

HIGH, the interrupt will stop triggering.

Interrupt on Falling Edge: The interrupt is triggered when the pin changes from HIGH (1) to LOW (0)

Interrupt on Rising Edge: The interrupt is triggered when the pin changes from LOW (0) to HIGH (1).

External Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

If INT0/1 bit is set (and the SREG I-bit is set), then interrupts are enabled on pin INT0/1

External Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	–	–	–	–	–	–	INTF1	INTF0	EIFR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Stores flags that indicate if an external interrupt request (INT0 or INT1) has occurred.

- INTF1 (Bit 1) – Set to 1 when an interrupt on INT1 (PD3) occurs.
- INTF0 (Bit 0) – Set to 1 when an interrupt on INT0 (PD2) occurs.
- Bits 2–7 are unused (reserved).

Flag is cleared automatically when interrupt routine is executed

Example of INT0 configuration on AVR-C

```
// Set External Interrupt Control Register A
// to 10: The low level of INT0 generates an interrupt request.
EICRA = (EICRA & ~(1 << ISC00))| (1 << ISC01);

// Set External Interrupt Mask Register
// to INT0 to activate interrupt INT0
EIMSK |= (1 << INT0);
```

```
// Enable global interrupts
sei();
```

2.3.2. Pin Change Interrupt PCINT[23:0]

PCINT[23:0] – triggered by any signal change (toggle): In the ATmega328P, 23 pins can trigger a Pin Change Interrupt (PCINT). These pins are divided into three groups (PCINT0-7, PCINT8-14, PCINT16-23), each associated with a specific interrupt vector. Where:

- PCINT[7:0] – PORT B [7:0]
- PCINT[14:8] – PORT C [6:0]
- PCINT[23:16] – PORT D [7:0]

Pin Change Interrupt Control Register

Bit	7	6	5	4	3	2	1	0	
(0x68)	–	–	–	–	–	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- PCIE2 enables interrupts for PCINT[23:16] PD0–PD7 (PORTD)
- PCIE1 enables interrupts for PCINT[14:8] PC0–PC6 (PORTC)
- PCIE0 enables interrupts for PCINT[7:0] PC0–PC6 (PORTC)

Pin Change Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1B (0x3B)	–	–	–	–	–	PCIF2	PCIF1	PCIF0	PCIFR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PCIF# is set if corresponding pins generate an interrupt request and cleared automatically when the interrupt routine is executed.

Pin Change Mask Register 2

Bit	7	6	5	4	3	2	1	0	
(0x6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	PCMSK2
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Each bit controls whether interrupts are enabled for the corresponding pin. Change on any enabled pin causes an interrupt.

Mask registers 1 and 0 are similar to mask register 2.

Example of PCINT configuration on AVR-C

```
// Enable PCIE2 Bit3 = 1 (Port D)
PCICR |= B00000100;
// Select PCINT23 Bit7 = 1 (Pin D7)
PCMSK2 |= B10000000;
```

CHAPTER 4 Timers and Event Scheduling in Embedded System

1. Introduction

Timers are internal hardware modules in the ATmega328P that count clock cycles and can be used to generate accurate time delays, trigger interrupts, or create PWM signals.

Timers are essential peripherals in AVR microcontrollers that enable precise time-based operations without burdening the CPU. This chapter introduces the timer modules available in the AVR family, particularly the ATmega328P, which includes Timer0, Timer1, and Timer2. It explains the core concepts of timer functionality, including counting, prescaling, overflow, compare match, and the generation of Pulse Width Modulation (PWM) signals. Various timer modes such as Normal mode, CTC (Clear Timer on Compare Match), and PWM modes are discussed in detail. The chapter also describes how to configure and use timers through associated control and status registers, as well as how to handle timer-related interrupts. By mastering these concepts, readers will be able to implement accurate delays, periodic tasks, and signal generation in embedded applications.

Timer counts from 0 to maximum (255 or 65535), then overflows and starts again. Useful for simple delays and overflow interrupts.

Types of Timers in ATmega328P

Timer	Bits	Features
Timer0	8-bit	Delays, PWM, Interrupts
Timer1	16-bit	More precise timing, PWM
Timer2	8-bit	Asynchronous capability, PWM

2. Timer Interrupt

The ATmega328P microcontroller has a built-in timer/counter module that can be used to generate timer interrupts.

Timer interrupts are useful for executing code at regular intervals without the need for constant polling, controlling PWM outputs, or measuring time intervals.

The ATmega328P has three timers:

- Timer 0: 8-bit timer.
- Timer 1: 16-bit timer (more precise timing and PWM generation).
- Timer 2: 8-bit timer (similar to Timer 0 but with some additional features).

Each timer can generate interrupts based on specific events, such as overflow, compare match, or input capture.

2.1. Timer Interrupt Modes

Overflow Interrupt: Occurs when the timer counter reaches its maximum value and rolls over to 0.

For example, in an 8-bit timer, the overflow occurs at 255 (0xFF).

Compare Match Interrupt: Occurs when the timer counter matches a value stored in a compare register (e.g., OCRxA or OCRxB).

This is often used for precise timing or PWM generation.

Input Capture Interrupt: Occurs when an external event is detected on the input capture pin (ICPx). Used for measuring pulse width or frequency

3. Timer/Counter Control Registers

We can initialize, configure, and control Timers & Timer Interrupts using the associated registers as stated in the datasheet of the microcontroller. The Timer-associated registers are as follows:

- TCCRxA: Timer/Counter Control Register A. Configures the waveform generation mode and output behavior

- TCCRxB: Timer/Counter Control Register B. Configures the timer's clock source, prescaler, and additional settings
- TCNTx: Timer/Counter Registers. Stores the current count value of Timerx
- OCRxA: Output Compare A Register. Holds the compare match value for Output Compare Unit A
- OCRxB: Output Compare B Register. Holds the compare match value for Output Compare Unit B
- TMISKx: Timer Interrupts Mask Register, enable/disable timer interrupts.
- TIFRx: Timer interrupts Flag Bits Register, read/clear timer interrupt flag bits.

Where x can be 0, 1, or 2 for timers (Timer/Counter 0, Timer/Counter 1, and Timer/Counter 2) respectively.

In time mode, the timer module will keep counting (0 to 255 or 65535 depending on resolution). At overflow, a timer overflow interrupt is fired and the timer rolls over back to zero and starts counting again.

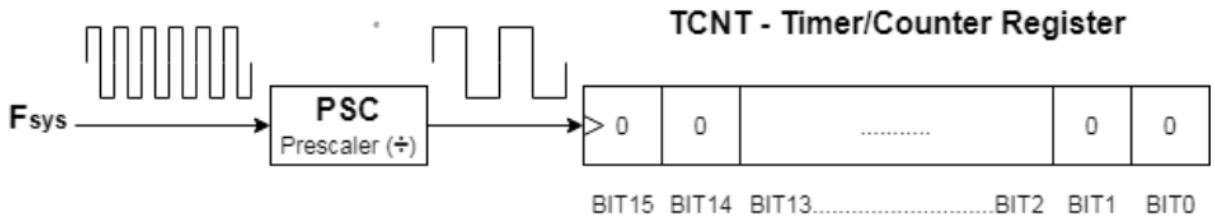
4. Timer Prescaler

A prescaler in a hardware timer module is a digital circuit that is used to divide the clock signal's frequency by a configurable number to bring down the timer clock rate so it takes longer to reach the overflow (maximum count number).

This is really useful to control the maximum time interval that can be generated using the timer module, the PWM output frequency, or the range of time that can be measured using the timer module.

Running the timer module at the system frequency is good for resolution but will generate so many timer overflow interrupts that needs extra care in your code. Hence, using a prescaler can be useful to avoid this situation in the first place if needed.

In timer mode, the timer module will have the internal clock of the system as a clock source and it passes through the prescaler as shown below. You can programmatically control the division ratio of the prescaler to control the timer input clock frequency as you need.



5. Configuring Timer Interrupt

5.1. Timer/Counter 1 interrupt overflow

To configure the Timer/Counter 1 interrupt overflow, we follow the following steps;

STEP 1. Setting Timer/Counter 1 in normal mode

The Timer/Counter 1 mode is set up by TTCR1A and TTCR1B Timer/Counter 1 control register

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

For normal mode set both TTCR1A and TTCR1B to 0

Table 15-4. Waveform Generation Mode Bit Description[1]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

In the simple mode the Timer/Counter 1 Register (TCNT1H:TCNT1L) counts up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 16-bit value 0xFFFF and then restarts 0x0000.

There are no special cases to consider in the Normal mode, a new counter value can be written anytime.

In normal operation the Timer/Counter Overflow Flag (TOV1) bit located in the Timer/Counter1 Interrupt Flag Register (T1FR1) will be set in the same timer clock cycle as the Timer/Counter 1 Register (TCNT1H:TCNT1L) becomes zero.

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

STEP 2. Setting Timer/Counter 1 prescaler

The clock input to Timer/Counter 1 (TCNT1) can be pre-scaled (divided down) by 5

preset values (1, 8, 64, 256, and 1024).

Table 13-5. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Clock Select for Timer/Counter 1 (CS1) bits 2:0 are located in Timer/Counter Control Registers B [yellow].

Normal Mode (WGM 1 bits 3:0 = 0000_2)

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

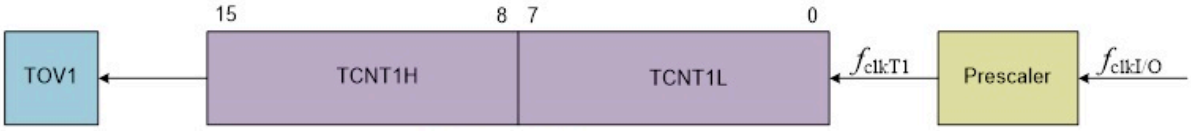
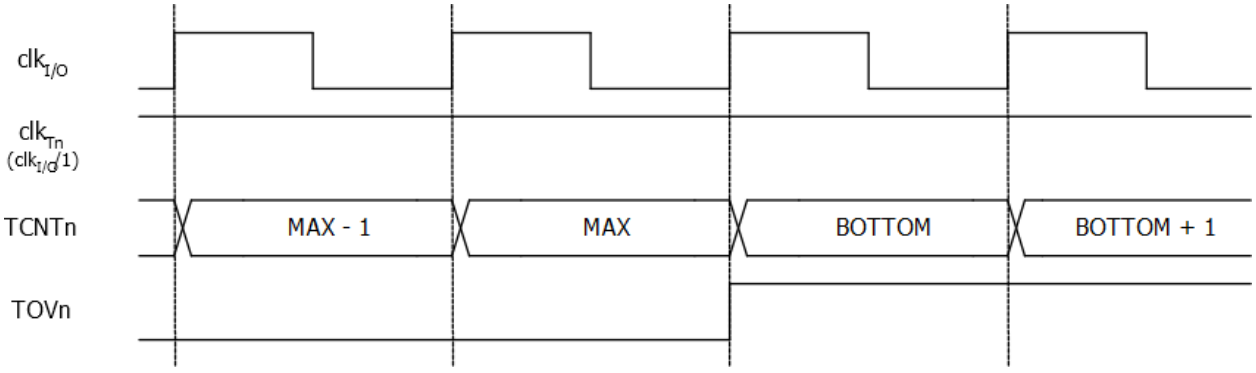


Table 13-5. Clock Select Bit Description

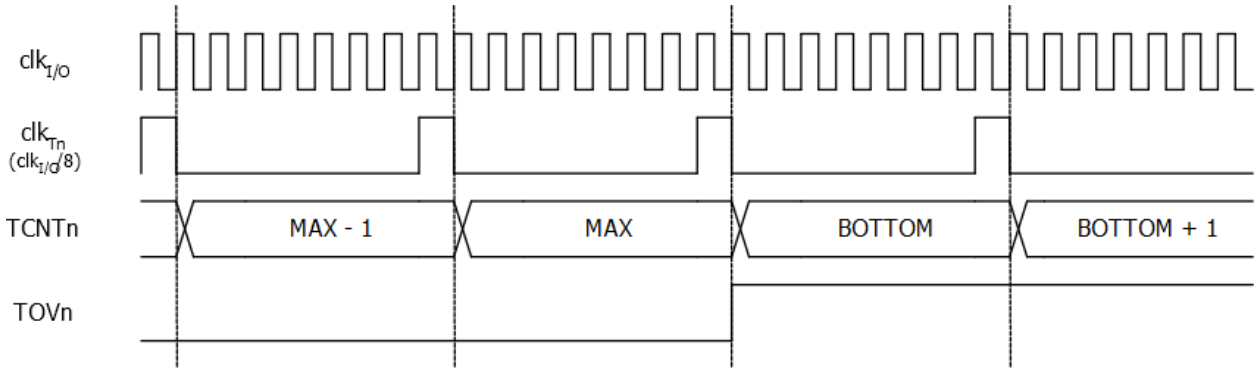
CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

The figures include information on when interrupt flags are set. The first figure below illustrates timing data for basic Timer/Counter operation close to the MAX value in all modes other than Phase Correct PWM mode.



The next figure shows the same timing data, but with the prescaler enabled. with

Prescaler /8 (fclk_I/O/8)



Example:

For Timer/Counter 1 TCNT1 is 16 bits.

Bit	7	6	5	4	3	2	1	0	
(0x85)	TCNT1[15:8]								TCNT1H
(0x84)	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

If we prescale for 256 how to calculate the time TOV1 interrupt occurred

Given the Atmegra328P clock is 16 MH = $16\ 10^{6}\ Hz\ =\ 16\ 10^{6}\ s^{-1}$

Each cycle takes $1/(16\ 10^{6})\ =\ 6.25\ 10^{-8}\ s$

If we prescale by 256, each Timer/Counter 1 cycle would be
$6.28\ 10^{-8}\ .\ 256\ =\ 1.6\ 10^{-5}$

To trigger the TOV1 (Timer/Counter 1 overflow interrupt) it takes from 0x0000 to 0xffff increments, each increment is done in one timer cycle with total 2^{16} cycles, ie, 65536 cycles

So the TOV1 is triggered after $1.6\ 10^{-5}\ 65539\ =\ 1.048576\ s$

Equation for calculating TOV1 time

$$T_{MAX} = 2^n N / f_{clk}$$

- T_{MAX} maximum time
- N prescaler (divider)
- n size in bit of the Timer/Counter register and 2^n represents the number of Timer/Counter to reach the overflow

For the previous example,

$$T_{MAX} = 2^{16} 256 / (16\,10^6) = 1.048576\,s$$

STEP 3. Timer/Counter 1 Interrupt Enabling

The TIMSK1 register iTimer/Counter 1 Interrupt Mask Register Timer/Counter 1 mask register

Bit	7	6	5	4	3	2	1	0
			ICIE			OCIEB	OCIEA	TOIE
Access			R/W			R/W	R/W	R/W
Reset			0			0	0	0

Bit 5 – ICIE1: Timer/Counter1, Input Capture Interrupt Enable When this bit is set (1), and the I-flag in the status register is set (interrupts globally enabled), the Timer/Counter1 input capture interrupt is enabled. The corresponding interrupt vector ISR is executed when the ICF1 flag, located in TIFR1, is set.

Bit 2 – OCIE1B: Timer/Counter1, Output Compare B Match Interrupt Enable When this bit is set, and the I-flag in the status register is set (interrupts globally enabled), the Timer/Counter1 output compare B match interrupt is enabled. The corresponding interrupt vector is executed when the OCF1B flag, located in TIFR1, is set.

Bit 1 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable When this bit is set, and the I-flag in the status register is set (interrupts globally enabled), the Timer/Counter1 output compare A match interrupt is enabled. The corresponding

interrupt vector is executed when the OCF1A flag, located in TIFR1, is set.

Bit 0 – TOIE: Overflow Interrupt Enable When this bit is set '1', and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter 1 Overflow interrupt is enabled. The corresponding Interrupt Vector is executed when the TOV Flag, located in TIFR1, is set.

The avr-C code to configure the timer:,

```
#ifndef F_CPU
#define F_CPU 16000000
#endif

#include <avr/io.h>
#include<util/delay.h>
#include <avr/interrupt.h>
#include <stdbool.h>
#include <stdio.h>


// Handle interrupt TIMER1_OVF_vect
ISR (TIMER1_OVF_vect) {
       // Toggle LED
       PORTB = PORTB ^ (1<<PINB5); //XOR to toggle the bit 5
}

int main(void)
{
  DDRB = DDRB | (1<<PINB5);  //0x20   PINB5 output

  TCCR1A = 0;           // Init Timer1A
  TCCR1B = 0;           // Init  Timer1B
  TCCR1B |= 0b00000100;  // Prescaler = 256
  TIMSK1 |= 0b0000001;  // Enable Timer Overflow Interrupt

  sei();

  while (1);
}
```

5.2. Timer/Counter 1 preloading

Let's say you'd like to generate a periodic timer interrupt every 500ms, here we use

Timer/Counter 1 preloading.

By default, the counter register TCNT1 counts 65536 timer ticks (cycles)

In the above equation, ($T_{MAX} = 2^n N / f_{clk}$) 2^n represents the number of ticks.

We can write $T_{out} = nticks\ x\ N / f_{clk}$

By setting T_{MAX} =500ms=0.5 s and replacing other parameters, we find:

0.5=nticks×256/16106

So, nticks=0.5×16 $/(16\ 10^6)$=31250 ticks

To trigger a time of 900 ms before TOV1 interrupt, we should preload TCNT1 = 65536 - 31250 = 34286

The code would be

```
#ifndef F_CPU
#define F_CPU 16000000
#endif

#include <avr/io.h>
#include<util/delay.h>
#include <avr/interrupt.h>
#include <stdbool.h>
#include <stdio.h>


// Handle interrupt TIMER1_OVF_vect
ISR (TIMER1_OVF_vect) {
```

```
    TCNT1 = 34286; // Timer Preloading
  // Handle The 500ms Timer Interrupt
  //...

}

int main(void)
{
  DDRB = DDRB | (1<<PINB5);  //0x20   PINB5 output
  TCCR1A = 0;           // Init Timer/Counter 1
  TCCR1B = 0;           // Init Timer/Counter 1
  TCCR1B |= B00000100;  // Prescalar = 256
  TCNT1 = 34286;        // Timer Preloading
  TIMSK1 |= B00000001;  // Enable Timer Overflow Interrupt

  sei();
  while (1);
}
```

5.3. Timer Compare Match Registers

An easier way to achieve the same goal without disrupting the timer's TCNTx register's value would be to use the compare match interrupt events. This is probably the best way to implement timer-based interrupt events. Because it gives you two (COMPA and COMPB) registers to generate two independent timer interrupt events using the same

timer module.

Considering the previous 500ms time interval interrupt example, we need the timer to tick 31250 ticks to get the 500ms periodic interrupt. So, we'll use COMPA compare register, enable its interrupt, and set its value to 31250. When a compare match occurs (when TCNT1 = OCR1A), the interrupt TIMER1_COMPA_vect is fired. And that's the 500ms periodic interrupt we want.

To keep it running at the same 500ms rate, we need to update the COMPA value because the timer's count has now reached 31250. Therefore, we need to add 31250 ticks to the COMPA value. Don't worry about overflow, at 65535, the register will roll over back to zero exactly like the timer's TCNT1 register does. So it's going to work flawlessly all the time.`

```
#ifndef F_CPU
#define F_CPU 16000000
#endif

#include <avr/io.h>
#include<util/delay.h>
#include <avr/interrupt.h>
#include <stdbool.h>
#include <stdio.h>


// Handle interrupt TIMER1_COMPA_vect
ISR(TIMER1_COMPA_vect)

{

   OCR1A += 31250; // Advance The COMPA Register

   // handle the interrupt..

}
```

```
int main(void)
{
  TCCR1A = 0;           // Init Timer1

  TCCR1B = 0;           // Init Timer1

  TCCR1B |= B00000100;  // Prescalar = 256

  OCR1A = 31250;        // Timer CompareA Register

  TIMSK1 |= B00000010;  // Enable Timer COMPA Interrupt



  sei();

  while (1);
}
```

Example of OVT1 and COMPA interrupts together

Here we set OCR1A to 16384 = 65536/4

So COMPA will be triggered 4 times where OVT1 1 time.

TCCR1B |= B00000101; ie Prescaled to (divided by 1024

```
#ifndef F_CPU
#define F_CPU 16000000
#endif
```

```
#include <avr/io.h>
#include<util/delay.h>
#include <avr/interrupt.h>
#include <stdbool.h>
#include <stdio.h>


// Handle interrupt TIMER1_OVF_vect
ISR (TIMER1_OVF_vect) {
      TCNT1 = 34286; // Timer Preloading

  // Handle The 500ms Timer Interrupt

  //...


}


ISR(TIMER1_COMPA_vect)

{

  OCR1A += 16384; // Advance The COMPA Register

  // Handle The 500ms Timer Interrupt

  //...

}


int main(void)
{
```

```
  TCCR1A = 0;             // Init Timer1
  TCCR1B = 0;             // Init Timer1
  TCCR1B |= B00000101;  // Prescalar = 1024
  OCR1A = 16384;          // Timer CompareA Register
  TIMSK1 |= B00000011;  // Enable Timer COMPA and OVT1 Interrupts

  sei();
  while (1);
}
```

CHAPTER 5 **Communications Protocols**

1. Introduction

Serial communication is a fundamental aspect of microcontroller applications, enabling data exchange between the AVR microcontroller and external devices such as computers, sensors, or other microcontrollers. This chapter provides an in-depth overview of the serial communication mechanisms supported by the AVR architecture, with a primary focus on the Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) module. It covers the principles of asynchronous and synchronous transmission, configuration of communication parameters such as baud rate, frame format, and parity, and the use of relevant control and status registers. Additionally, practical examples demonstrate how to transmit and receive data efficiently, both through polling and interrupt-driven methods. By the end of this chapter, readers will be equipped to implement robust serial interfaces in AVR-based systems.

2. Basics of Serial Communications

Most microcontrollers include serial communications capability. On the ATmega328P it's called the Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART).

Embedded systems, microcontrollers, and computers mostly use UART as a form of device-to-device hardware communication protocol. Among the available communication protocols, UART uses only two wires for its transmitting and receiving ends.

A universal asynchronous receiver-transmitter (UART) is a peripheral device for asynchronous serial communication in which the data format and transmission speeds are configurable. It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits so that precise timing is handled by the communication channel.

3. Transmitting and receiving serial data

A UART contains those following components:

- a clock generator, usually a multiple of the bit rate to allow sampling in the middle of a bit period
- input and output shift registers, along with the transmit/receive or FIFO buffers
- transmit/receive control
- read/write control logic

For UART to work the following settings need to be the same on both the transmitting and receiving side:

- Voltage level:For the voltage level, 2 UART modules work well when they both have the same voltage level, e.g 3V-3V between the 2 UART modules. To use 2 UART modules at different voltage levels, a level switch circuit needs to be added externally.
- Baud Rate: In telecommunication and electronics, baud is a common unit of measurement of symbol rate, which is one of the components that determine the speed of communication over a data channel, in this case if defines the bit rate
- Parity bit: A parity bit, or check bit, is a bit added to a string of binary code. Parity bits are a simple form of error detecting code. Parity bits are generally applied to the smallest units of a communication protocol, typically 8-bit octets (bytes)
- Data bits size
- Stop bits size
- Flow Control: In data communications, flow control is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver.

3.1. Data framing

A UART frame consists of 5 elements:

- Idle (logic high (1)) //no data transfer is taking place,
- Start bit (logic low (0))
- Data bits

- Parity bit
- Stop (logic high (1))

In the most common settings of 8 data bits, no parity and 1 stop bit (aka 8N1), the protocol efficiency is 80%. 8-N-1 is a common shorthand notation for a serial port parameter setting or configuration in asynchronous mode, in which there is one start bit, eight (8) data bits, no (N) parity bit, and one (1) stop bit.

Each character is framed as a logic low start bit, data bits, possibly a parity bit and one or more stop bits.

1	**5-9**	**0-1**	**1-2**
Start bit	Data frame	Parity bits	Stop bits

Start bit

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one (1) clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

Data bit

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one (1) clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

Parity bit

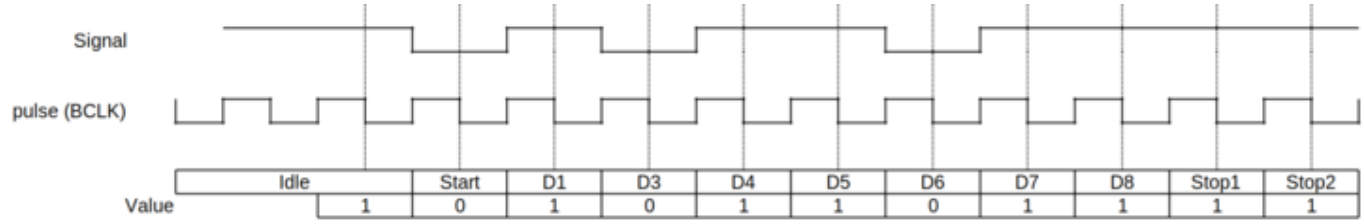
Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers.

After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 or logic-high bit in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bit or logic highs in the data frame should total to an odd number.

When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd, or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

Stop bit

The next one or two bits are always in the mark (logic high, i.e., '1') condition and called the stop bit(s). They signal to the receiver that the character is complete. Since the start bit is logic low (0) and the stop bit is logic high (1) there are always at least two guaranteed signal changes between characters.

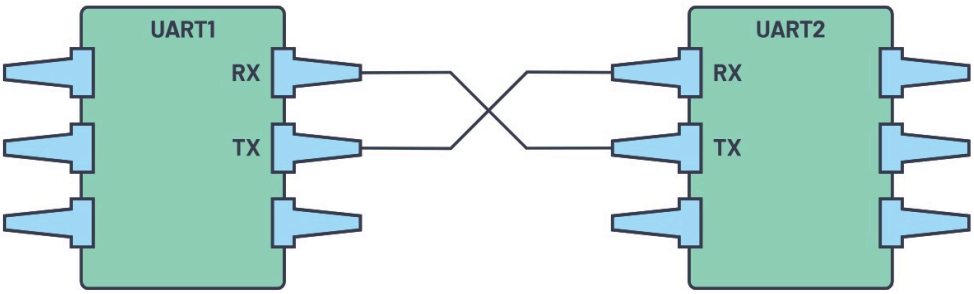


Example of a UART frame. In this diagram, one byte is sent, consisting of a start bit, followed by eight data bits (D1-8), and two stop bits, for a 11-bit UART frame. The number of data and cannotformatting bits, the presence or absence of a parity bit, the form of parity (even or odd) and the transmission speed must be pre-agreed by the communicating parties. The "stop bit" is actually a "stop period"; the stop period of the transmitter may be arbitrarily long. It cannot be shorter than a specified amount, usually 1 to 2 bit times. The receiver requires a shorter stop period than the transmitter. At the end of each data frame, the receiver stops briefly to wait for the next start bit. It is this difference which keeps the transmitter and receiver synchronized. BCLK = Base Clock

Embedded systems, microcontrollers, and computers mostly use UART as a form of device-to-device hardware communication protocol. Among the available communication protocols, UART uses only two wires for its transmitting and receiving ends.

3.2. UART Interface

By definition, UART is a hardware communication protocol that uses asynchronous serial communication with configurable speed. Asynchronous means there is no clock signal to synchronize the output bits from the transmitting device going to the receiving end.



The two signals of each UART device are named:

- Transmitter (Tx)
- Receiver (Rx)

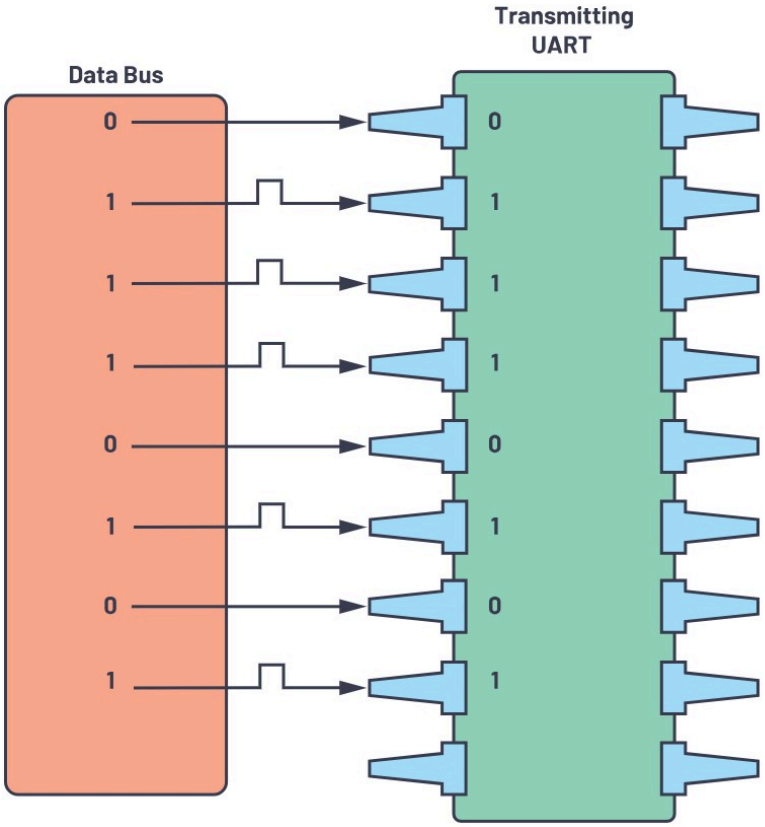
The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication.

The transmitting UART is connected to a controlling data bus that sends data in a parallel form. From this, the data will now be transmitted on the transmission line (wire) serially, bit by bit, to the receiving UART. This, in turn, will convert the serial data into parallel for the receiving device.

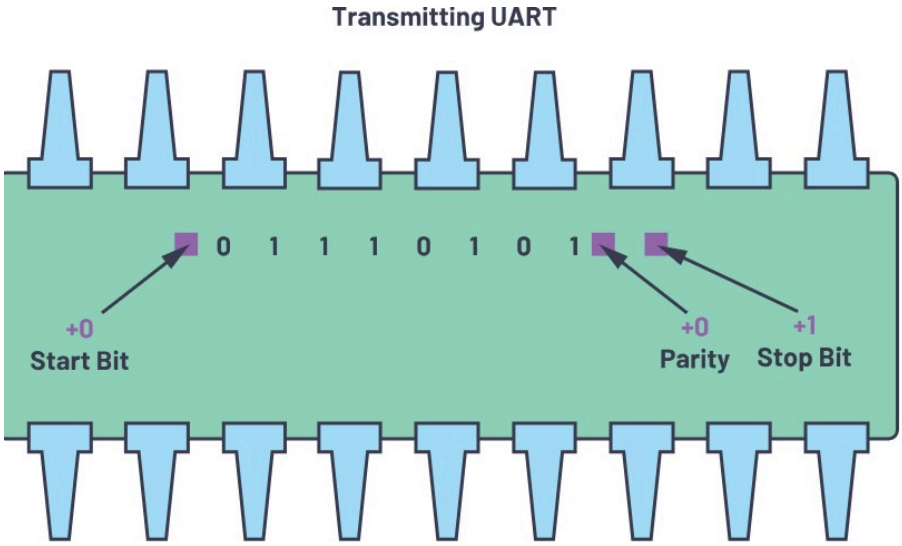
For UART and most serial communications, the baud rate needs to be set the same on both the transmitting and receiving device. The baud rate is the rate at which information is transferred to a communication channel. In the serial port context, the set baud rate will serve as the maximum number of bits per second to be transferred.

3.3. Steps of UART Transmission

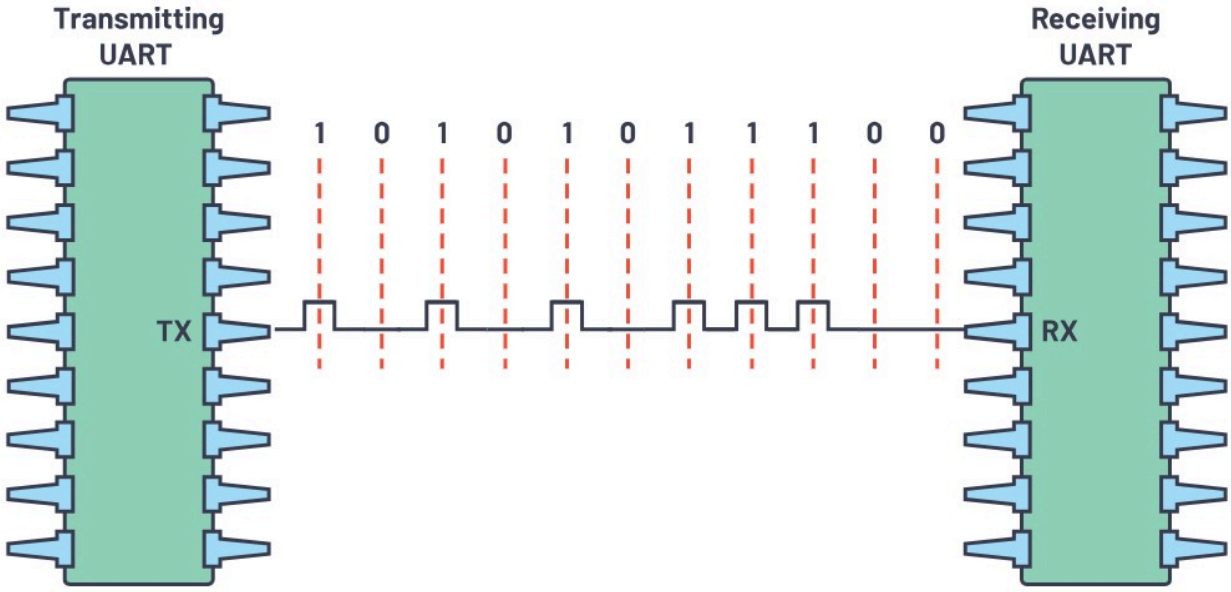
First: The transmitting UART receives data in parallel from the data bus.



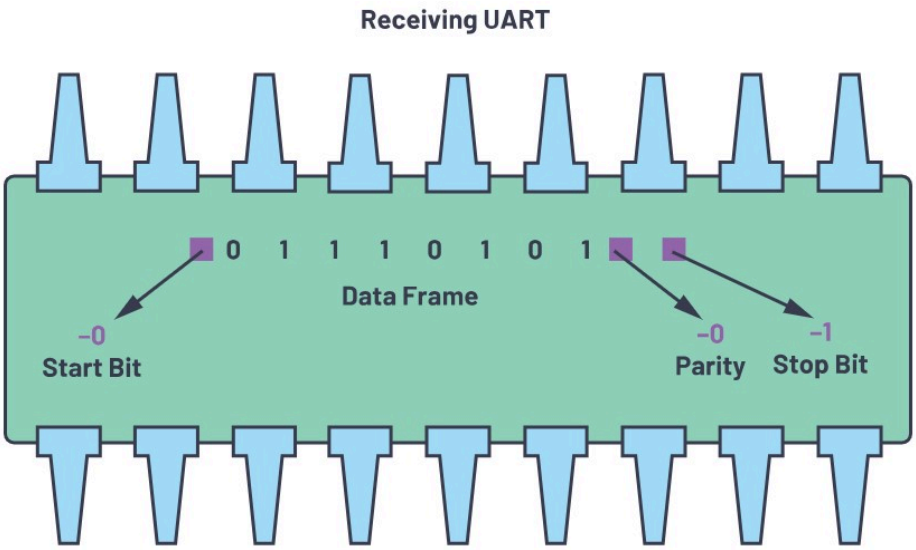
Second: The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame.



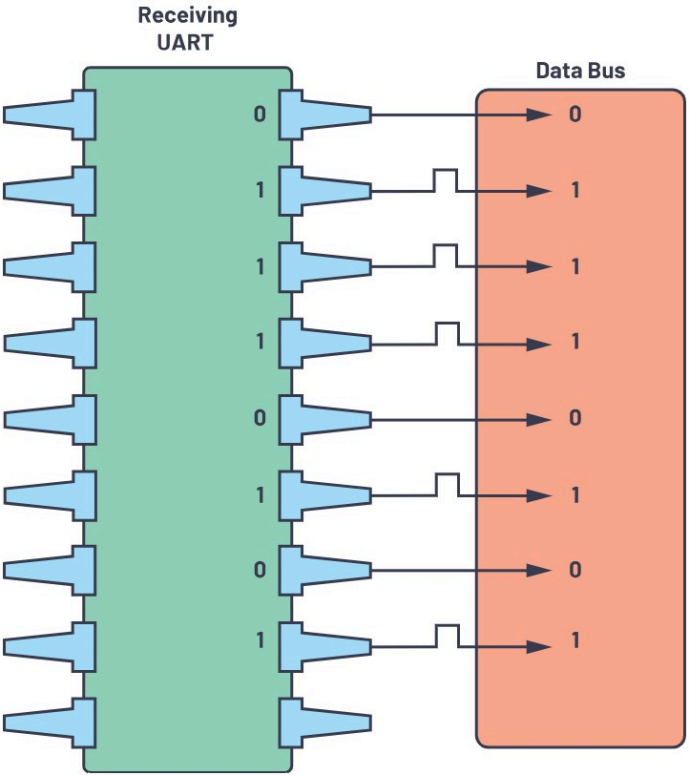
Third: The entire packet is sent serially starting from start bit to stop bit from the transmitting UART to the receiving UART. The receiving UART samples the data line at the preconfigured baud rate.



Fourth: The receiving UART discards the start bit, parity bit, and stop bit from the data frame



Fifth: The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end.



4. Serial Communications with the ATmega328P

The ATmega328P contains a Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) that can be used do serial communications. The data is transmitted from the ATmega328P on the TxD pin, and data is received on the RxD pin. The USART0 transmitter (TxD) and receiver (RxD) use the same pins on the chip as I/O ports PD1 and PD0. This means that applications that use the USART0 can not also use these two bits in Port D. It is not necessary to set any bits in the DDRD register in order to use the USART0

4.1. BAUD Rate and Control Registers

The baud rate is generated from the system clock with the help of Prescaler and clock circuit. The USART Baud Rate Register (UBRR0) controls the programmable down counter / Prescaler to generate a particular clock signal. The down-counter, running at system clock (fclk), is loaded with the UBRR0 value each time the counter has counted down to zero and generates a clock pulse.

asynchronous normal mode (Bit U2Xn=0)

$$BAUD = \frac{f_{clk}}{16 \times (UBRRn + 1)}$$

$$UBRRn = \frac{f_{clk}}{16 \times BAUD} - 1$$

The above formula is used to calculate the right value of UBBR0. For Atmega328p (Arduino UNO) the system clock is running at 16Mhz. If we intend to communicate at a speed of 9600bps:

The value of UBBR0 should be:

$$UBBR0 = \frac{16 \times 10^{6}}{16 \times 9600} - 1 = 103$$

The next step is to set the Frame Format using UCSR0C register. The USART accepts all 30 combinations of the following as valid frame formats:

- 1 start bit
- 5, 6, 7, 8, or 9 data bits
- no, even or odd parity bit
- 1 or 2 stop bits

The next step is to enable the Transmitter and Receiver to use UCSR0B register and load the UBR0 register with the data to transmit. In the case of reception, the UBR0 is read by the application.

UBRR0L and UBRR0H – USART Baud Rate Registers

(0xC5)	–	–	–	–	11	10	9	8
(0xC4)	7	6	5	7	3	2	1	0

• Bit 15:12 – Reserved These bits are reserved for future use. For compatibility with future devices, these bits must be written to zero when UBRRH is written.

• Bit 11:0 – UBRR[11:0]: USART Baud Rate Register This is a 12-bit register that contains the USART baud rate. The UBRRH contains the four most significant bits, and the UBRRL contains the eight least significant bits of the USART baud rate. Ongoing transmissions by the transmitter and Receiver will be corrupted if the baud rate is changed. Writing UBRRL will trigger an immediate update of the baud rate Prescaler.

Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRRn Value
Asynchronous Normal mode (U2Xn = 0)	$BAUD = \frac{f_{OSC}}{16(UBRRn+1)}$	$UBRRn = \frac{f_{OSC}}{16BAUD} - 1$
Asynchronous Double Speed mode (U2Xn = 1)	$BAUD = \frac{f_{OSC}}{8(UBRRn+1)}$	$UBRRn = \frac{f_{OSC}}{8BAUD} - 1$
Synchronous Master mode	$BAUD = \frac{f_{OSC}}{2(UBRRn+1)}$	$UBRRn = \frac{f_{OSC}}{2BAUD} - 1$

UCSR0C – USART Control and Status Register C

Bit	7	6	5	4	3	2	1	0
(0xC2)	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ0	UCPOL0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	1	1	0

Bits 7:6 – UMSEL0 1:0 USART Mode Select

These bits select the mode of operation of the USART

UMSEL1	UMSEL0	Mode
0	0	Asynchronous USART / UART
0	1	Synchronous USART
1	0	(Reserved)
1	1	Master SPI (MSPIM)

Bits 5:4 – UPM0 1:0: Parity Mode

These bits enable and set the type of parity generation and check. If enabled, the Transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The Receiver will generate a parity value for the incoming data and compare it to the UPM setting. If a mismatch is detected, the UPE Flag in UCSRA will be set.

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

Bit 3 – USBS0: Stop Bit Select

This bit selects the number of stop bits to be inserted by the Transmitter. The Receiver ignores this setting.

USBS	Stop Bit(s)
0	1-bit
1	2-bit

Bit 2:1 – UCSZ0 1:0: Character Size

The UCSZ 1:0 bits combined with the UCSZ 2 bit in UCSRB sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use.

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Bit 0 – UCPOL0: Clock Polarity. This bit is used for synchronous mode only. Write this bit to zero when asynchronous mode is used. The UCPOL bit sets the relationship between data output change and data input sample, and the synchronous clock (XCK).

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

UCSR0B – USART Control and Status Register B

Bit	7	6	5	4	3	2	1	0
(0xC1)	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Initial Value	0	0	0	0	0	0	0	0

• Bit 7 – RXCIE0: RX Complete Interrupt Enable Writing this bit to one enables interrupt on the RXC Flag. A USART Receive Complete interrupt will be generated only if the RXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one, and the RXC bit in UCSRA is set.

• Bit 6 – TXCIE0: TX Complete Interrupt Enable Writing this bit to one enables interrupt on the TXC Flag. A USART Transmit Complete interrupt will be generated only if the TXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXC bit in UCSRA is set.

• Bit 5 – UDRIE0: USART Data Register Empty Interrupt Enable Writing this bit to one enables interrupt on the UDRE Flag. A Data Register Empty interrupt will be generated only if the UDRIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDRE bit in UCSRA is set.

• Bit 4 – RXEN0: Receiver Enable Writing this bit to one enables the USART Receiver. The Receiver will override normal port operation for the RxD pin when enabled. Disabling the Receiver will flush the receive buffer invalidating the FE, DOR, and UPE Flags.

• Bit 3 – TXEN0: Transmitter Enable Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxD pin when enabled. The disabling of the Transmitter (writing TXEN to zero) will not become effective until ongoing and pending transmissions are completed, i.e., when the Transmit Shift Register and Transmit Buffer Register do not contain data to be transmitted. When disabled, the Transmitter will no longer override the TxD port.

• Bit 2 – UCSZ02: Character Size The UCSZ2 bits combined with the UCSZ 1:0 bit in UCSRC sets the number of data bits (Character Size) in a frame the Receiver and Transmitter use.

• Bit 1 – RXB80: Receive Data Bit 8 RXB8 is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from UDR.

• Bit 0 – TXB80: Transmit Data Bit 8 TXB8 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to UDR.

UCSR0A – USART Control and Status Register A

Bit	7	6	5	4	3	2	1	0
(0xC0)	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Read/Write	R	R/W	R	R	R	R	R	R
Initial Value	0	0	1	0	0	0	0	0

• Bit 7 – RXC0: USART Receive Complete This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty. The RXC flag can be used to generate a receive complete interrupt.

• Bit 6 – TXC0: USART Transmit Complete This flag bit is set when the entire frame in the transmit shift register has been shifted out and there are no new data currently present in the transmit buffer (UDR). The TXC Flag can generate a transmit complete interrupt.

• Bit 5 – UDRE0: USART Data Register Empty The UDRE Flag indicates if the transmit buffer (UDR) is ready to receive new data. If UDRE is one, the buffer is empty, and therefore ready to be written. The UDRE Flag can generate a Data Register Empty interrupt (see description of the UDRIE bit). UDRE is set after a reset to indicate that the Transmitter is ready.

• Bit 4 – FE0: Frame Error This bit is set if the next character in the receive buffer had a Frame Error when received. I.e., when the first stop bit of the next character in the receive buffer is zero. This bit is valid until the receive buffer (UDR) is read. The FE bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRA.

• Bit 3 – DOR0: Data OverRun This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the Receive Shift Register, and a new start bit is detected. This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

• Bit 2 – UPE0: USART Parity Error This bit is set if the next character in the receive buffer had a Parity Error when received and the Parity Checking was enabled at that point (UPM01 = 1). This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

• Bit 1 – U2X0: Double the USART Transmission Speed This bit only has an effect on the asynchronous operation. Write this bit to zero when using synchronous operation. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

• Bit 0 – MPCM0: Multi-processor Communication Mode This bit enables the Multi-processor Communication mode. When the MPCMn bit is written to one, all the incoming frames received by the USART Receiver that do not contain address information will be ignored. The Transmitter is unaffected by the MPCM setting.

UDR0 – USART I/O Data Register

Bit (0xC6)	7	6	5	4	3	2	1	0
UDR (Read)	RXB7	RXB6	RXB5	RXB4	RXB3	RXB2	RXB1	RXB0
UDR (Write)	TXB7	TXB6	TXB5	TXB4	TXB3	TXB2	TXB1	TXB0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDRn. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDRn Register location. Reading the UDR Register location will return the contents of the Receive Data Buffer Register (RXB).

When data is written to the transmit buffer, and the Transmitter is enabled, the Transmitter will load the data into the Transmit Shift Register when the Shift Register is empty. Then the data will be serially transmitted on the TxD pin.

4.2. UART Configuration

The function: USART_Init() initializes the USART module to enable serial communication between the AVR microcontroller and external devices like PCs, sensors, or other microcontrollers.

```
#include <avr/io.h>      // Contains all the I/O Register Macros
```

```c
#include <util/delay.h>  // Generates a Blocking Delay
#include <avr/interrupt.h> // Contains all interrupt vectors

#define USART_BAUDRATE 9600 // Desired Baud Rate
#define BAUD_PRESCALER (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)

#define ASYNCHRONOUS (0<<UMSEL00) // USART Mode Selection

#define DISABLED    (0<<UPM00)
#define EVEN_PARITY (2<<UPM00)
#define ODD_PARITY  (3<<UPM00)
#define PARITY_MODE  DISABLED // USART Parity Bit Selection

#define ONE_BIT (0<<USBS0)
#define TWO_BIT (1<<USBS0)
#define STOP_BIT ONE_BIT      // USART Stop Bit Selection

#define FIVE_BIT  (0<<UCSZ00)
#define SIX_BIT   (1<<UCSZ00)
#define SEVEN_BIT (2<<UCSZ00)
#define EIGHT_BIT (3<<UCSZ00)
#define DATA_BIT   EIGHT_BIT  // USART Data Bit Selection

#define RX_COMPLETE_INTERRUPT         (1<<RXCIE0)
#define DATA_REGISTER_EMPTY_INTERRUPT (1<<UDRIE0)

volatile uint8_t USART_TransmitBuffer; // Global Buffer

void USART_Init()
{
      // Set Baud Rate
      UBRR0H = BAUD_PRESCALER >> 8;
      UBRR0L = BAUD_PRESCALER;

      // Set Frame Format
      UCSR0C = ASYNCHRONOUS | PARITY_MODE | STOP_BIT | DATA_BIT;

      // Enable Receiver and Transmitter
```

```
    UCSR0B = (1<<RXEN0) | (1<<TXEN0);

    //Enable Global Interrupts
    sei();
}
```

Code Breakdown:

```
// Set Baud Rate
UBRR0H = BAUD_PRESCALER >> 8;
UBRR0L = BAUD_PRESCALER;
```

Purpose: Configures the baud rate for UART communication.

Explanation: The value of BAUD_PRESCALER is split into two 8-bit registers:

- UBRR0H (high byte)
- UBRR0L (low byte)

This sets the speed at which data is transmitted and received.

```
// Set Frame Format
UCSR0C = ASYNCHRONOUS | PARITY_MODE | STOP_BIT | DATA_BIT;
```

Purpose: Sets the USART frame format.

Explanation: The frame format includes:

- Mode (asynchronous/synchronous)
- Parity (none, even, or odd)
- Stop bits (1 or 2)
- Data bits (usually 8 bits)

These options are combined into UCSR0C using predefined macros/constants.

```
// Enable Receiver and Transmitter
```

```
UCSR0B = (1<<RXEN0) | (1<<TXEN0);
```

Purpose: Enables the USART receiver and transmitter.

Explanation:

- RXEN0 enables reception (receiving data)
- TXEN0 enables transmission (sending data)

5. Transmission and Recepetion

5.1. Polling transmission

Polling transmission is the simplest method of transmission where the application software keeps monitoring the status of the USART transmitter hardware and loads a byte of data into the USART buffer UDR0 only when the hardware is ready for transmission. This wastes CPU time in constantly monitoring the status of UDRE0 bit of UCSR0A register.

To configure the transmission

```
#define F_CPU 16000000UL // Defining the CPU Frequency
#define USART_BAUDRATE 9600 // Desired Baud Rate
#define BAUD_PRESCALER (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)

#define ASYNCHRONOUS (0<<UMSEL00) // USART Mode Selection

void USART_Init()
{
	// Set Baud Rate
	UBRR0H = BAUD_PRESCALER >> 8;
	UBRR0L = BAUD_PRESCALER;

	// Set Frame Format
	UCSR0C = ASYNCHRONOUS | PARITY_MODE | STOP_BIT | DATA_BIT;

	// Enable Receiver and Transmitter
	UCSR0B = (1<<RXEN0) | (1<<TXEN0);
}
```

To send data

```
void USART_TransmitPolling(uint8_t DataByte)
{
    // Do nothing until UDR is ready
      while (( UCSR0A & (1<<UDRE0)) == 0) {};     UDR0 = DataByte;
}
```

5.2. Polling Reception

Polling reception is the simplest method of reception where the application software keeps monitoring the status of the USART receiver hardware and reads the data from the USART buffer UDR0 only when the hardware has received a byte of data. This wastes CPU time in constantly monitoring the status of RXC0 bit of UCSR0A register.

```
uint8_t USART_ReceivePolling()
{
      uint8_t DataByte;

     // Do nothing until data have been received
      while (( UCSR0A & (1<<RXC0)) == 0) {};
      DataByte = UDR0 ;
      return DataByte;
}
```

5.3. Interrupt Transmission

In polling the CPU waste valuable time monitoring the USART registers. This valuable time could be used in the execution of other instructions. This problem is solved by interrupt based transmission. Below code transmits using interrupts. The code transmits

the character ‘a’ endlessly while the D13 LED on Arduino UNO keeps blinking. The CPU keeps performing the LED blinking in an infinite loop and every time the transmission finishes an interrupt is generated to state that the UDR0 buffer is ready to receive new data. The CPU pauses the LED blinking and serves the ISR.

```
void USART_TransmitInterrupt(uint8_t Buffer)
{
      USART_TransmitBuffer = Buffer;
      UCSR0B |= DATA_REGISTER_EMPTY_INTERRUPT; // Enables the
Interrupt
}

ISR(USART_UDRE_vect)
{
      UDR0 = USART_TransmitBuffer;
      //UCSR0B &= ~DATA_REGISTER_EMPTY_INTERRUPT; // Disables the
Interrupt, uncomment for one time transmission of data
}
```

5.4. Interrupt Reception

Interrupt reception behaves exactly the same as polling reception but in the case of interrupt reception. The CPU is busy looping an infinite loop and whenever data is received in USART Buffer an interrupt is thrown and the CPU serves it and toggles the LED accordingly. The CPU doesn’t have to monitor the USART register bits to check the status of the reception.

```
ISR(USART_RX_vect)
{
      USART_ReceiveBuffer = UDR0;
      if (USART_ReceiveBuffer == 'a')
      {
            PORTB |= 1<<5;    // Writing HIGH to glow LED
      }
      else
      {
            PORTB &= ~(1<<5); // Writing LOW
```

```
        }

    }
```

CHAPTER 6 Analog-to-Digital Conversion for Sensor and Actutor Control

1. Introduction

Many real-world signals, like temperature, light, and sound, are **analog** — they vary continuously over time. However, **microcontrollers** work with **digital data** (discrete binary numbers: 0s and 1s).

An **Analog-to-Digital Converter (ADC)** is a key component in digital systems that converts real-world, **continuous analog signals** (like temperature, voltage, sound, or light) into **discrete digital numbers**. This process is essential because **digital devices like microcontrollers** can only process numbers—not continuous voltages.

An **Analog-to-Digital Converter (ADC)** is a **hardware feature** that converts these analog signals into a corresponding digital value that can be processed by a microcontroller.

2. How Does an ADC Work?

An Analog-to-Digital Converter (ADC) works by taking a continuous analog input signal—such as voltage from a sensor—and converting it into a digital number that a microcontroller or digital system can process. This conversion involves measuring the analog signal at specific moments in time, assigning each measurement to the nearest available digital level, and encoding it as a binary value. This process allows digital systems to interpret and respond to real-world, analog phenomena like temperature, light, or sound.

The 3 Key Steps in ADC

1. **Sampling (Time Quantization):**

The analog signal is measured at regular time intervals. This process captures the signal's value at specific moments, allowing the system to track how it changes over time.

- **Sampling** is the process of measuring the analog signal at **regular intervals**.
- Each measurement is a **snapshot** of the signal's amplitude at that time.
- The **sampling rate** (e.g., 1 kHz = 1000 samples/sec) must follow **Nyquist's Theorem**:

  To accurately capture a signal, the sampling rate should be **at least twice** the highest frequency in the signal.

2. **Quantization (Amplitude Discretization):**

Each sampled value is rounded to the nearest level within a fixed range. The number of available levels depends on the ADC's resolution (e.g., 8-bit = 256 levels).

The **LSB (Least Significant Bit)** is the **smallest change in voltage** the ADC can detect. For example, in a 10-bit ADC with a 5V reference:

$$\text{LSB} = \frac{5V}{2^{10} - 1} = \frac{5V}{1023} \approx 4.88\,\text{mV}$$

When the ADC measures a voltage, the real input voltage might fall **between two levels**. The ADC will round it to the **nearest level**, causing a small error.

This **maximum possible error** due to rounding is **half of one LSB**, This means the true analog value may be up to **half an LSB higher or lower** than the converted digital value represents. So,

- After sampling, the signal is **rounded off** to the nearest digital level.
- The number of levels depends on the **resolution** of the ADC:
  - A **4-bit ADC** has $2^4 = 16$ levels
  - A **10-bit ADC** has $2^{10} = 1024$ levels

- This rounding introduces **quantization error** or **noise** — the difference between the actual analog value and the digital approximation.

Quantization Noise

  - Maximum error = **±½ LSB (Least Significant Bit)**
  - Increasing the number of bits reduces this error and **improves accuracy**

3. **Encoding:** The quantized level is then converted into a binary number, which the microcontroller can read and process.
   - Each quantized level is then **encoded into a binary number** (e.g., 6 → `0110`)
   - This binary output is what your microcontroller reads and processes.

This process allows continuous real-world signals to be represented in a form usable by digital systems.

Important Concepts

Term	Meaning
Resolution (n bits)	Determines how many discrete levels the ADC can represent: 2n2^n
Reference Voltage (Vref)	The max input voltage that corresponds to the highest digital output
Step Size (LSB size)	Smallest voltage difference the ADC can distinguish: $$\text{Step Size} = \frac{V_{\text{ref}}}{2^n - 1}$$
Threshold Voltage	Voltage range between levels; each digital output corresponds to a voltage band

Calculation Example

Given:

- ADC resolution: **4-bit**
- Reference voltage: **8V**
- Input voltage: **2.85V**

Step size (LSB):

$$\text{Step Size} = \frac{8V}{2^4 - 1} = \frac{8V}{15} \approx 0.533V$$

Quantized Level:

$$\frac{2.85V}{0.533V} \approx 5.34 \rightarrow \text{Round to nearest level} = 5$$

Binary Output:

Level 5 = 0101 → This is what the microcontroller reads.

3. ADC in Microcontroller, Case of ATmega328P

The Atmel ATmega328P microcontroller used on the Arduino Uno has an analog-to-digital conversion (ADC) module capable of converting an analog voltage into a 10-bit number from 0 to 1023 or an 8-bit number from 0 to 255. The input to the module can be selected to come from any one of six inputs on the chip. While this gives the microcontroller the ability to convert the signals from six different analog sources into 10-bit values only one channel can be converted at a time. The ADC can convert signals at a rate of about 15 kSPS (samples per second). The inputs to the ADC module appear on the Arduino board as connections A0 through A5.

3.1. Reference Voltages

In order to convert an analog voltage to a digital value on any ADC, the converter has to also be provided with the range of voltages it is expected to deal with. Otherwise it’s like

asking “How high is high?”. The range is specified to the converter by supplying a low and a high reference voltage. The converter will then assign digital values linearly between the minimum and maximum digital value as the input signal changes from the low reference to the high reference. If the input is equal to the low reference, the output is all zeros. If it’s equal to the high reference it outputs all ones, and if the input is midway between the low and high references the output would be the digital value in the middle of the numerical range.

On the ATmega328P the low reference is fixed at ground but it has the ability to select one of three sources to serve as the high reference.

AREF-Thisis a separate pin on the microcontroller that can be used to provide any high reference voltage you wish to use as long as it’s in the range of 1.0V to VCC. On the Uno it’s wired to one of the black headers.

AVCC- This pin on the microcontroller provides power to the ADC circuitry on the chip. On the Uno it is connected to VCC. This is the simplest option to use provided AVCC is connected to VCC.

1.1V- The microcontroller has an internal 1.1V reference voltage that can be used.

3.2. Conversion Rate

The ADC circuitry needs to have a clock signal provided to it in the range of 50kHz to 200kHz. There is no external ADC clock input on the chip but rather the clock is generated internally from same clock used to run the microcontroller. This CPU clock is too fast (16MHz on the Uno) so the chip includes an adjustable “prescaler” to divide the CPU clock down to something acceptable. The prescaler can be set to divide by a choice of divisors (2, 4, 8, 16, 32, 64, or 128) and it’s up to the programmer to select one that results in a ADC clock within the acceptable range.

3.3. Registers

The interface to the ADC module from the software is through a group of registers. The descriptions below are very brief and only cover some of the functions of the registers. For full description see the ATmega328P datasheet. In the description below notation with brackets and a colon like “XYZ[2:0]” means the combination of the bits XYZ2,

XYZ1 and XYZ0. For example if we say “XYZ[2:0] = 5”, this means the binary value of 5 (101) has been assigned to the three XYZ bits (XYZ2 = 1, XYZ1 = 0, XYZ0 = 1).

ADMUX- Multiplexer Selection Register

The ADMUX register contains three fields of bits for controlling various aspects of the data conversion.

ADMUX	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR		MUX3	MUX2	MUX1	MUX0

Bits 7:6- Bits REFS1 and REFS0 control the selection of the high reference source.

REFS1	REFS0	High voltage selection
0	0	AREF
0	1	AVCC
1	1	Internal 1.1V

Bit 5- The ADLAR bit determines whether to left adjust the result or right adjust it.

ADLAR	Conversion result
0	Right adjusted for 10 bit results
1	Left adjusted for 8 bit results

Bits 3:0- The MUX3 through MUX0 bits control the selection of which of the six input lines to connect to the digitizing circuit.

MUX3	MUX2	MUX1	MUX0	Input to ADC module
0	0	0	0	Arduino A0
0	0	0	1	Arduino A1
0	0	1	0	Arduino A2
0	0	1	1	Arduino A3
0	1	0	0	Arduino A4
0	1	0	1	Arduino A5

ADCSRA- Control and Status Register A

The ADC module of the ATmega328P has two control and status registers, ADCSRA and ADCSRB. For basic ADC operations only the bits in the ADCSRA register have to be modified. For information on the ADCSRB register, see the ATmega328P data sheet.

ADCSRA	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

This register has bits for initiating actions and reporting various conditions in the ADC module. The ones needed for this lab are these.

Bit 7- The ADEN bit enables the ADC module. Must be set to 1 to do any ADC operations.

Bit 6- Setting the ADSC bit to a 1 initiates a single conversion. This bit will remain a 1 until the conversion is complete. If your program using the polling method to determine when the conversion is compete, it can test the state of this bit to determine when it can read the result of the conversion from the data registers. As long as this bit is a one, the data registers do not yet contain a valid result.

Bit 3- Setting the ADIE bit to a 1 enables interrupts. An interrupt will be generated on the completion of a conversion. The interrupt vector name is “ADC_vect”.

Bits 2:0- The ADPS2, ADPS1 and ADPS0 bits selects the prescaler divisor value.

ADPS2	ADPS1	ADPS0	Prescaler value
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADCH and ADCL- Data Register (high and low bytes)

The results of a conversion are stored in the two bytes of the Data Register. The most significant bits of the result are stored in ADCH and the least significant bits are in ADCL. If you wish to use the full 10-bit conversion results, the ADLAR bit in the ADMUX register should be cleared to a zero. This causes the 10-bit conversion result to be right justified in the 16-bit combination of ADCH and ADCL (referred to in a program as ADC). The two most significant bits are in ADCH(bits 1 and 0) and the lower eight bits are in ADCL.

ADCH	7	6	5	4	3	2	1 ADC9	0 ADC8
ADCL	7 ADC7	6 ADC6	5 ADC5	4 ADC4	3 ADC3	2 ADC2	1 ADC1	0 ADC0

When using the 10-bit results (ADLAR=0), the results can be read in a program with code like this.

```
uint16_t  x;
x = ADC;
```

This will result in the high order bits from ADCH being stored in the upper byte of the 16-bit variable, and the lower eight bits from ADCL being stored in the lower byte of the variable.

In many cases only an eight-bit conversion value is needed. For this situation, set the ADLAR bit in the ADMUXregister to a one. The conversion results will be left justified with the eight most significant bits in ADCH. The 8-bit result can then be obtained by just reading the ADCH register and ignoring the contents of ADCL.

ADCH	7 ADC9	6 ADC8	5 ADC7	4 ADC6	3 ADC5	2 ADC4	1 ADC3	0 ADC2
ADCL	7 ADC1	6 ADC0	5	4	3	2	1	0

When using 8-bit results (ADLAR=1), the results can be read in a program with code like this.

```
uint8_t x;
x = ADCH;
```

3.4. Using the ADC

ADC Setup

The basic steps in setting up the ADC module to do conversions are these:

1. Configure the REFS[1:0] bits to select the high reference voltage to use. Using AVCC is recommended.
2. Set or clear the ADLAR bit depending on whether you want to use 10- or 8-bit conversion results.
3. Configure the MUX[3:0] bits to select the input channel to be used.
4. Configure the ADPS[2:0] bits to select the clock prescaler value.
5. Set the ADEN bit in ADCSRA to a one. This enables the ADC and you're now ready to initiate a conversion.

If using interrupts, also do these steps:

6. Write an interrupt service routine (ISR) for the ADC_vect interrupt.
7. Set the ADIE bit in ADCSRA to a one to enable the module to interrupt.
8. Enable global interrupts with the sei() function call.

3.4.1. ADC Conversions Using Polling

To do conversions using polling, follow these steps:

1. Set the ADSC bit in ADCSRA to one. This starts a conversion.
2. Enter a loop that checks the state of the ADSC bit. As long as it remains one, the conversion is in progress. Once it becomes zero, the conversion is complete.
3. Read the result from ADCH (8-bit) or ADC (10-bit).

```
#include <avr/io.h>

void adc_init() {
    // Select AVCC as reference voltage, and set ADC channel to ADC0
(A0)
    ADMUX = (1 << REFS0);

    // Set ADC prescaler to 128 for 16 MHz clock => 125 kHz ADC clock
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 <<
ADPS0);
}

uint16_t adc_read() {
```

```
    // Start conversion by setting ADSC bit
    ADCSRA |= (1 << ADSC);

    // Wait for conversion to complete (ADSC becomes '0' again)
    while (ADCSRA & (1 << ADSC));

    // Read and return the 10-bit result from ADC
    return ADC;  // ADC is a 16-bit register (ADCL + ADCH)
}

int main(void) {
    uint16_t result;

    adc_init();  // Initialize the ADC

    while (1) {
        result = adc_read();  // Read analog value from A0
        // You can now use 'result' as needed
    }
}
```

- This example reads from analog pin A0 (ADC0).
- The result is a 10-bit value (0–1023).
- The ADC runs in polling mode — the CPU waits in a loop until the conversion completes.

3.4.2. ADC Conversions Using Interrupts

To do conversions using interrupts, follow these steps:

1. Set the ADSC bit in ADCSRA to one to start the first conversion.
2. Enter a loop where you either wait idly or perform other tasks until the interrupt occurs.
3. In the ISR, read the result from ADCH (8-bit) or ADC (10-bit). If you want to start another conversion immediately, you can do so within the ISR by setting the ADSC bit again.

```
#include <avr/io.h>
#include <avr/interrupt.h>

volatile uint16_t adc_result = 0;  // Variable to store the ADC
result

void adc_init() {
    // Select AVCC as reference voltage, and ADC0 (A0) as input
    ADMUX = (1 << REFS0);

    // Enable ADC, ADC interrupt, and set prescaler to 128 (for 125
kHz ADC clock with 16 MHz system clock)
    ADCSRA = (1 << ADEN) | (1 << ADIE) |
             (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

    sei();                // Enable global interrupts
    ADCSRA |= (1 << ADSC);  // Start first conversion
}

ISR(ADC_vect) {
    adc_result = ADC;        // Read the 10-bit result
     // You can now use 'result' as needed
    ADCSRA |= (1 << ADSC);   // Start next conversion
}

int main(void) {
    adc_init();  // Initialize the ADC with interrupts

    while (1) {
    // Main loop can perform other tasks while ADC runs in background
    // You can use adc_result variable wherever needed
    }
}
```

Explanation:

- The ADC is initialized to use AVCC as the reference and ADC0 (A0) as the input.

- The ADC interrupt (ADIE) is enabled, and global interrupts are enabled using `sei()`.
- The ISR(ADC_vect) is triggered automatically when a conversion is complete.
- Inside the ISR:
    - The result is read from the `ADC` register (10-bit).
    - A new conversion is started by setting the `ADSC` bit again.

This setup allows the CPU to perform other operations without being blocked by the ADC conversion.

CHAPTER 7 Finite State Machine for Embedded Systems Design

1. Introduction

Embedded systems are often tasked with managing complex behaviors, reacting to events in real time, and transitioning between multiple modes of operation. Traditional programming approaches can lead to tangled code when handling such logic, especially as system complexity grows. To address this challenge, UML (Unified Modeling Language) state machines offer a structured, graphical way to model and implement system behavior.

UML state machines are particularly effective for designing reactive embedded systems—systems that must respond predictably to external and internal events. They allow designers to define system states, transitions triggered by events, and actions executed in response. This formalism helps ensure correctness, improves maintainability, and bridges the gap between system modeling and code implementation.

This chapter introduces the principles of UML state machines, focusing on their application in embedded systems design. We explore the core concepts such as states, transitions, events, and actions, and demonstrate how to model real-time behaviors using flat and hierarchical state machines.

2. What is a state machine(FSM) ?

A **state machine**, also known as a **Finite State Machine (FSM)**, is a fundamental software model of computation used widely in embedded systems and control design. It consists of a **finite number of well-defined states**, which the system can occupy at any given moment. The finite nature of these states is what gives FSM its name.

In a state machine, **transitions** occur between states based on specific **input events**. These transitions are pre-defined and limited, ensuring the system behaves in a predictable and controlled manner. Because the system changes its state in response to external or internal events, an FSM is classified as an **event-driven system**.

In addition to changing states, a state machine can also **produce outputs**. These outputs are determined by both the **current state** and the **input events** received. This behavior allows FSMs to model complex logic with clarity and precision, making them particularly valuable in embedded system applications where deterministic and responsive behavior is essential.

3. Benefits of Using State Machines (FSMs)

Finite State Machines (FSMs) offer a powerful way to model and manage the behavior of reactive systems. One of their key benefits is the ability to **clearly describe the lifecycle or behavior of an application** by defining its various states and the transitions between them. This modeling approach allows developers to represent scenarios where a system must respond to different events in different ways, depending on its current state.

FSMs are especially useful in **complex applications involving multiple decision points**, varied outputs (actions), and numerous events to handle. They help break down this complexity into manageable components, where each state represents a specific condition or mode of operation, and transitions define how and when the system shifts between these modes.

Another advantage of FSMs is their **visual representation** through state diagrams or state charts. These visual tools not only aid developers in understanding the flow of control but also make it easier to **communicate system behavior with non-developers**, such as stakeholders or designers.

State machines also make it easier to **modify and implement behavioral changes**. Since the system is organized into distinct states, updates or extensions to behavior can often be made by modifying only specific states or transitions, rather than refactoring the entire codebase.

A complex system modeled with FSMs can be **decomposed into loosely coupled units**, each responsible for a specific behavior. These units (individual state machines) can be **developed, tested, and reused independently**, improving code reusability and maintainability.

Moreover, FSMs enhance **debugging and maintenance**, since developers can isolate issues to specific states or transitions. The structured nature of state machines also supports **scalability**, as new states and transitions can be added without disrupting the existing architecture.

Ultimately, FSMs help **reduce overall system complexity** by narrowing the focus to state-level behavior. This makes the design more analyzable, testable, and easier to implement in embedded systems and other event-driven applications.

4. UML state machines

UML (Unified Modeling Language) State Machines are a visual modeling tool used to describe the dynamic behavior of a system. They are particularly useful in modeling **reactive systems**—systems that respond to sequences of external or internal events. UML state machines are an extension of classical Finite State Machines (FSMs) and offer enhanced features such as **hierarchical states**, **composite states**, and **entry/exit actions**, making them highly expressive and suitable for complex embedded systems.

4.1. Guidance Example: UML State Machine

To support the understanding of UML state machine concepts in the context of embedded systems design, this guidance example will be used throughout the chapter. It provides a practical scenario involving a timer control application with multiple input events and behavioral transitions. Each key concept of UML state machines—such as states, transitions, events, actions, entry/exit behaviors, and guard conditions—will be illustrated using this example. This approach aims to bridge theoretical principles with real-world application, allowing for clearer visualization and better comprehension of how state machines are applied in embedded system development.

- We are designing an embedded timer application with the following user interactions:
- + Button: Increments the countdown time (minutes).
- – Button: Decrements the countdown time (minutes).
- S/P Button: Starts or pauses the countdown; displays status (STAT).
- + and – Together: Aborts the countdown and returns to IDLE mode.

- When the countdown is paused, time can be modified.
- When entering IDLE mode, the system should beep 20 times.
- While in IDLE, pressing S/P should show STAT for 1 second, then auto-return to IDLE.

4.2. State

A state represents a distinct stage in the lifecycle of an object. In the context of a StateMachine Behavior, a state models a situation during execution in which a specific invariant condition holds. Often, this condition is not explicitly defined but rather implied by the name given to the state, as described in the OMG® UML 2.5.1 specification. To define or fix a state, one typically maps the various scenarios that an object encounters throughout its lifecycle into a corresponding number of discrete states.

How do you arrive at fixing a state?

Map different scenarios through which an object lifecycle passes into number of states

Example,

For the guidance example, ehe Protimer application goes through several distinct scenarios during its lifecycle. These include the I**DLE, TIME-SET, COUNTDOWN, PAUSE,** and **STAT** states. Each of these represents a specific situation or phase in the application's behavior, outlining how the object transitions through its various stages over time.

- **IDLE**: The default or resting state when the timer is not active and awaiting user interaction.
- **TIME-SET**: The state in which the user sets the desired countdown time.
- **COUNTDOWN**: The active state where the timer is running and counting down from the set time.
- **PAUSE**: A temporary halt in the countdown, allowing the user to resume or reset later.

- **STAT**: The state used to display statistics or results after the countdown has ended.

These states collectively model the lifecycle of the Protimer application object.

To create a state in a state machine diagram, you begin by drawing a round-cornered rectangle, which visually represents the state. Inside this rectangle, you create a horizontal compartment for the state's name. It is important to assign a name that is unique within the context of the state machine diagram to clearly distinguish it from other states.

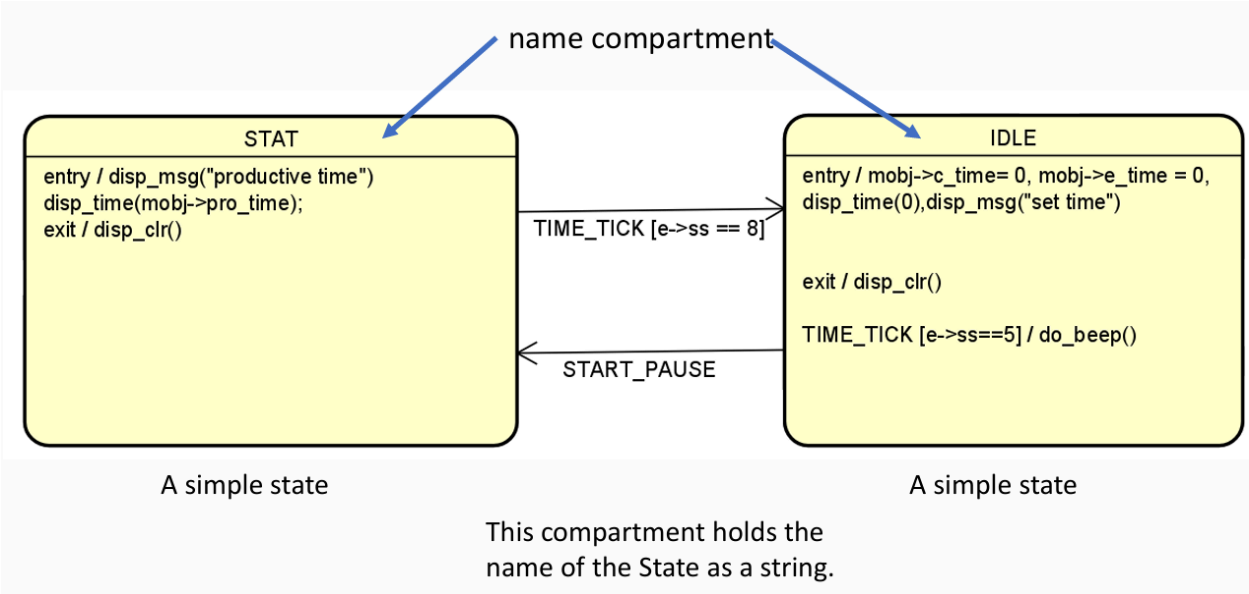
4.2.1. Types of states in UML

In UML, there are several types of states used to model the behavior of objects within a state machine. These include:

- **Simple State**: A basic state with no internal structure; represents a single condition or situation in an object's lifecycle
- **Composite State**: A state that contains nested states, allowing for more complex behavior and transitions within it.
- **Submachine State**: A reference to another state machine used as a state, enabling reuse and modular design within state machine diagrams.

4.2.2. Simple state

Simple State: A state is considered a simple state if it does not contain any substates, transitions, regions, or submachines. It represents a basic, atomic condition in the state machine.



Internal Activities Compartment

The **Internal Activities Compartment** of a state holds a list of internal behaviors associated with that state. Each behavior is described using a specific format:

- **<behavior-type-label> [‘/’ <behavior-expression>]**

UML defines standard behavior-type labels such as:

- **entry** – an action executed upon entering the state
- **exit** – an action executed when exiting the state

- **do** – an ongoing activity performed while the system remains in the state

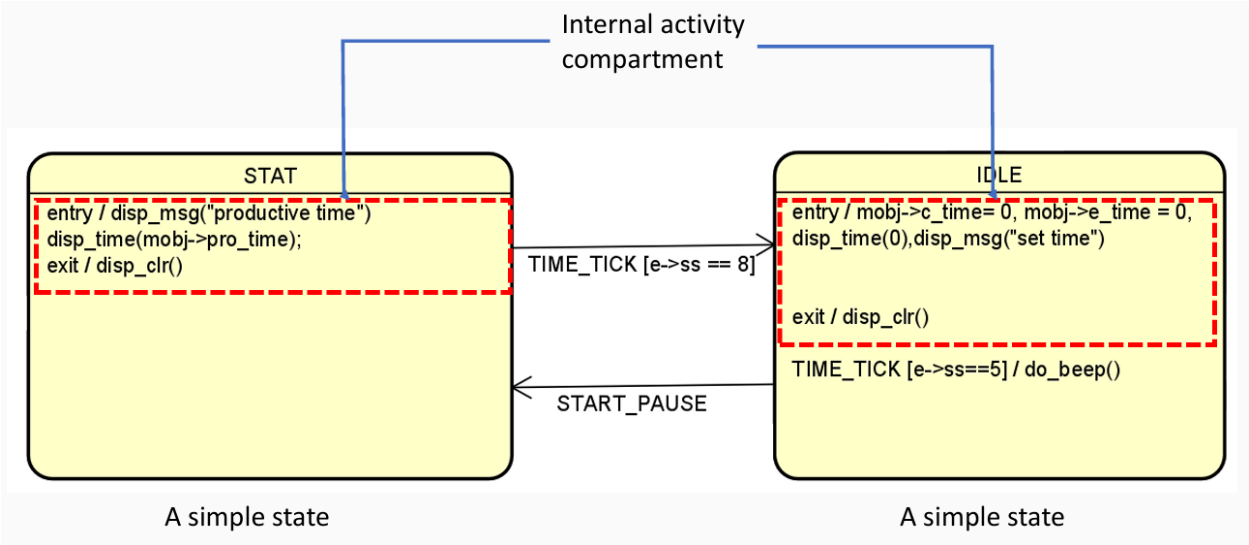
It is important to note that these keywords (**entry**, **exit**, and **do**) are reserved for internal activities and should not be used to represent events in the state machine diagram.

Internal activity labels

Internal activity labels are used to identify the conditions under which specific behaviors, defined by a *behavior-expression*, are executed within a state. These labels are:

- **entry**: The behavior specified by the expression is executed when the system enters the state. Use the `entry` label if the state includes an entry action.
- **exit**: The behavior specified by the expression is executed when the system exits the state. Use the `exit` label if the state includes an exit action.
- **do**: The behavior specified by the expression is executed continuously while the object remains in the state or until the specified computation is completed. This represents an ongoing activity. Use the `do` label only if the state involves a `do` action.

These labels help structure and define the internal behavior of states in a UML state machine.



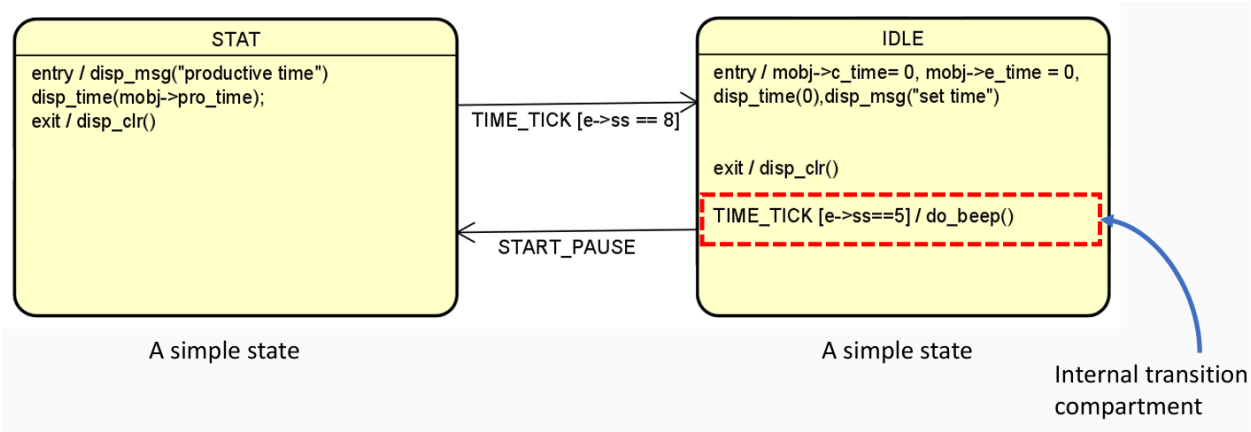
Internal transition

An **internal transition** defines behavior that is executed in response to an event without causing the state to exit or re-enter. Each internal transition follows this syntax:

{<trigger>} ['[' <guard> ']'] [/ <behavior-expression>]*

If an event occurrence matches the specified **trigger**, and the **guard condition** evaluates to **TRUE**, then the behavior defined by the **behavior-expression** is executed.

This allows the state to handle specific events internally, maintaining its current status without any transition to another state.



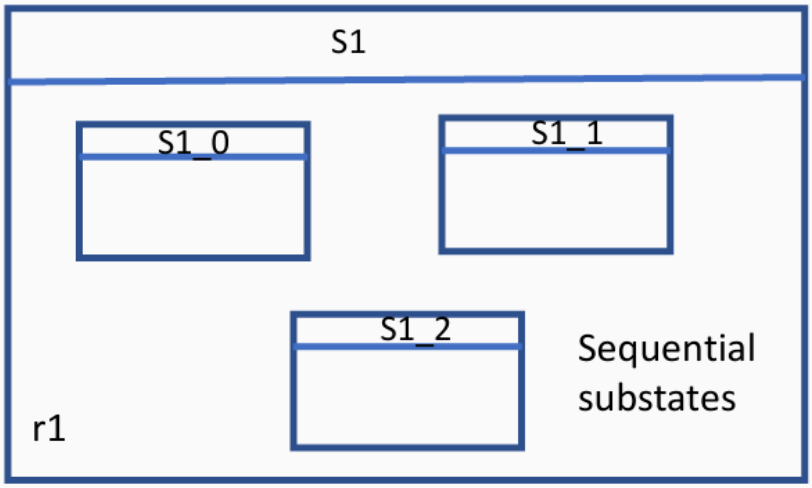
4.2.3. Composite State

A **composite state** is a state that contains substates, with at least one region. There are two types of composite states:

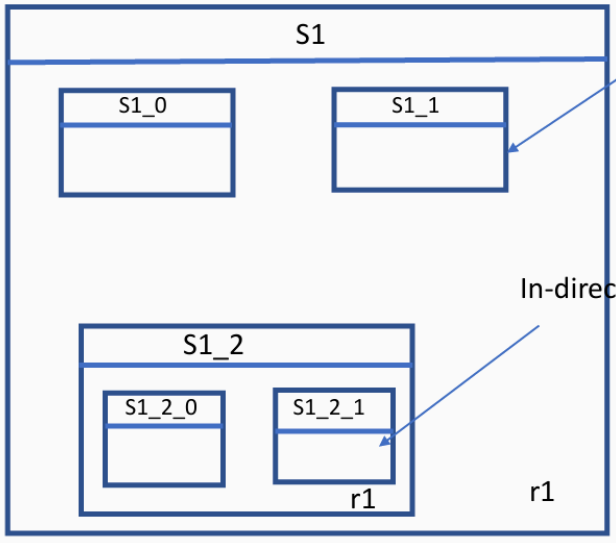
1. **Simple Composite State**: A composite state that has exactly one region.
2. **Orthogonal State**: A composite state that contains multiple regions, where each region operates independently.

Composite states are useful for expressing state hierarchies, enabling a more structured and organized representation of complex behavior. They also help make statecharts more comprehensible by reducing the number of transitions between states.

Any state enclosed within a region of a composite state is called a substate of that composite state [OMG® UML 2.5.1]



S1 is a composite state
S1 has 1 region
S1_x are sub states of S1
S1 is a superstate of S1_x
S1_x are simple states



Direct substate of S1

In-direct substate of S1

A simple composite state

S1 is a composite state
S1 has 1 region
S1_x are sub states of S1
S1 is a superstate of S1_x
S1_0 ,S1_1,S1_2_0,S1_2_1 are simple states
S1_2 is a composite state
S1_2 is superstate of S1_2_0 and S1_2_1
S1_2 has 1 region

Orthogonal State

An **orthogonal state** is a special case of a composite state that contains multiple regions. Each region within an orthogonal state can operate independently, allowing for concurrent behavior in different parts of the state. This structure enhances the flexibility and complexity of the state machine by supporting parallel execution within the same state.

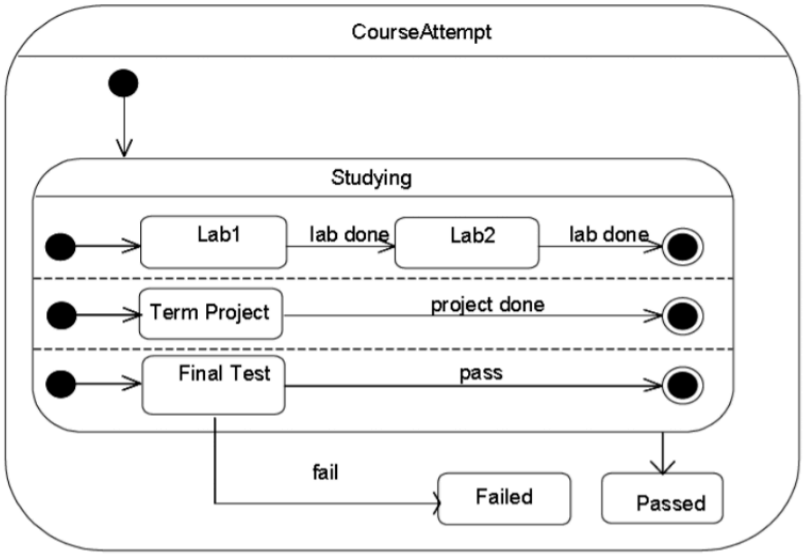
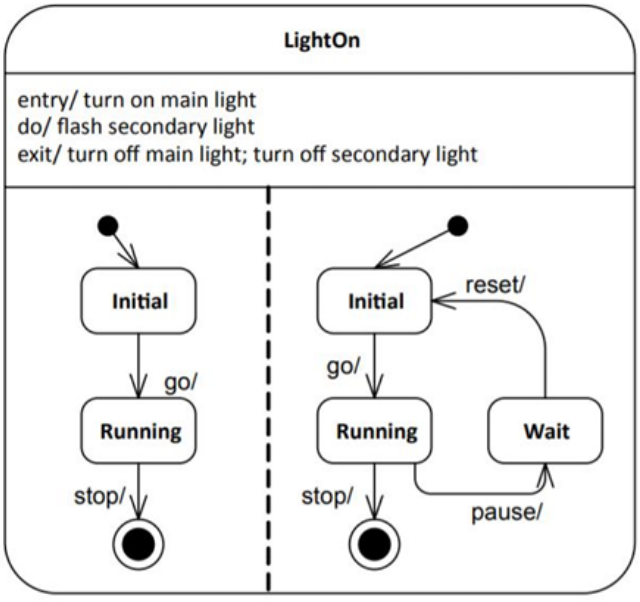


Figure 14.9 Composite State with Regions

Studying is a orthogonal state (composite state having 3 regions) having concurrent substates(not sequential)



Composite State with two regions and entry, exit and do behaviors

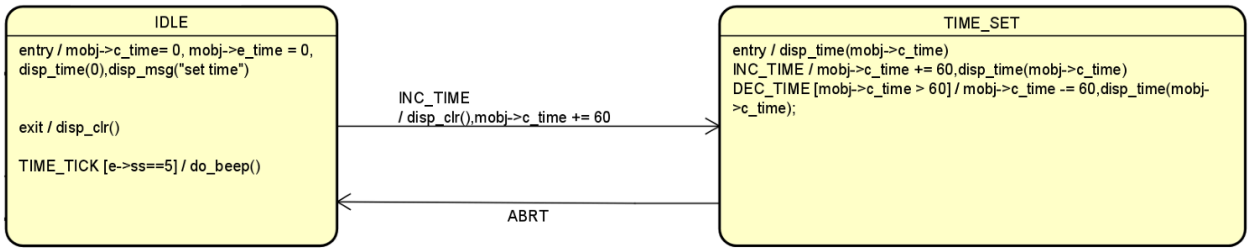
4.3. Transition

There are three types of transitions in state machine diagrams:

- **External Transition:** A transition that causes the state machine to exit the current state and enter a different state, typically triggered by an event.
- **Local Transition:** A transition that occurs within the same state, allowing for a change in the state's behavior without exiting or entering a new state.
- **Internal Transition:** A transition that triggers behavior within the current state, executing a specified action without causing the state to exit or re-enter.

External transition

An **external transition** occurs when the source state is exited due to the occurrence of a trigger. In this case, the optional action associated with the transition is executed first, followed by the execution of the exit action of the source state. This type of transition results in the system leaving the current state and moving to a new state.



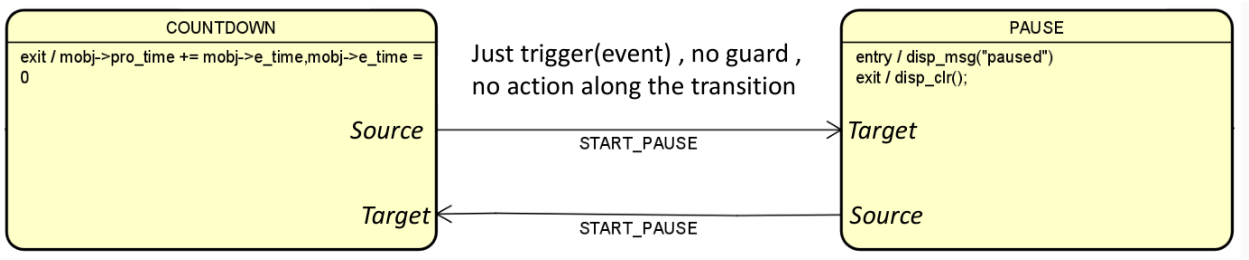
An **external transition** occurs when the source state is exited due to the occurrence of a trigger. In this process, the optional action associated with the transition is executed, followed by the execution of the exit action of the state. An external transition signifies a change in the state or situation of an object within its lifecycle. Once the state is changed, the object is ready to process a new set of events and execute a new set of actions. Transitions in state machine diagrams are denoted by lines with arrowheads, indicating the flow from the source state to the target state.

A **transition** in state machine diagrams follows a specific syntax to define its behavior:

- *{<trigger>} ['[' <guard> ']'] [/* <behavior-*expression*> *]**

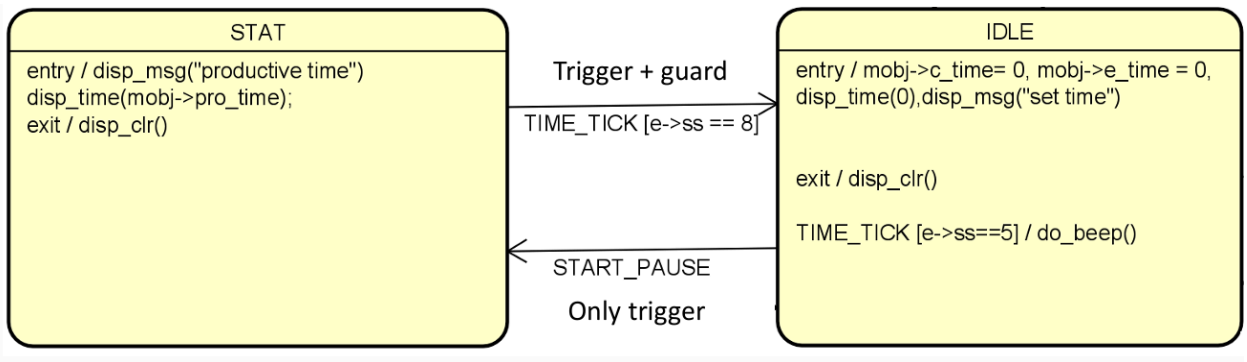
- **{<trigger>} [guard] / action**
- **event [guard] / action**

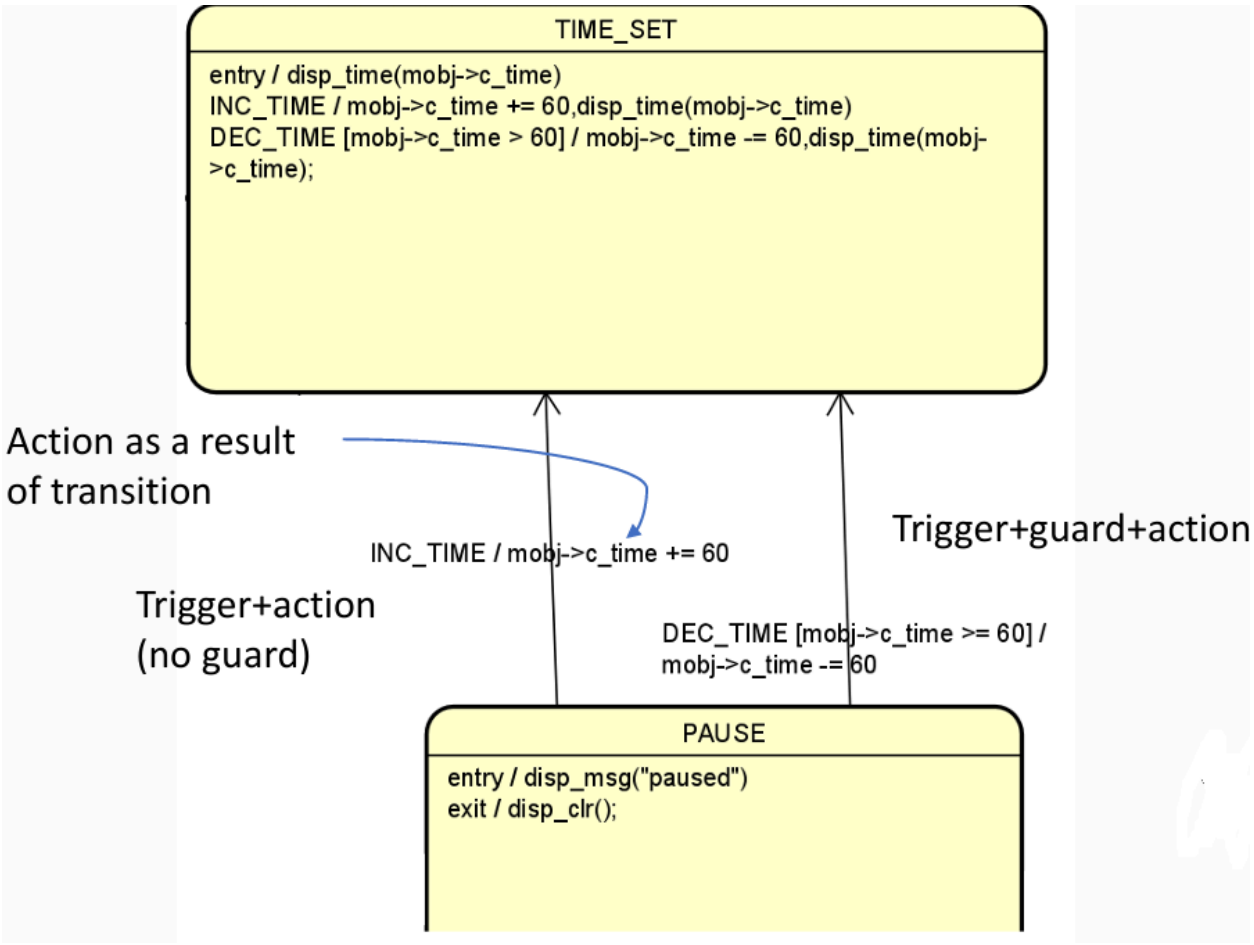
In this syntax, the **trigger** specifies the event that causes the transition, the **guard** is an optional condition that must be true for the transition to occur, and the **action** represents the behavior executed when the transition is triggered. The **behavior-expression** can be used to define additional actions associated with the transition.



When the current state of the object is COUNTDOWN and if the event START_PAUSE is received then object transitions to state PAUSE (updates its state to PAUSE)

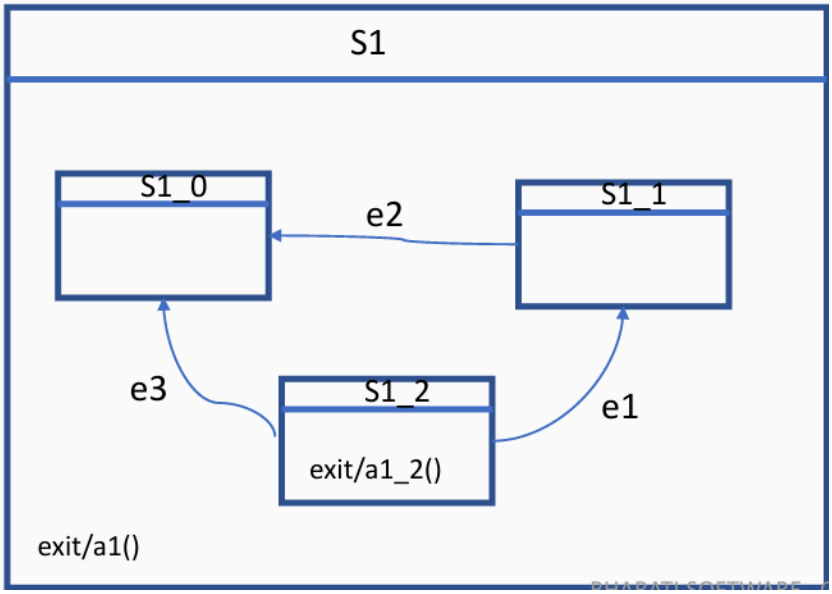
Guard is a boolean expression and must evaluate to TRUE for transition to fire





Local transition

Local transition: local is the opposite of external, meaning that the Transition does not exit its containing state (and, hence, the exit Behavior of the containing State will not be executed). However, for local Transitions, the target Vertex must be different from its source Vertex. A local Transition can only exist within a composite State. [OMG® UML 2.5.1]



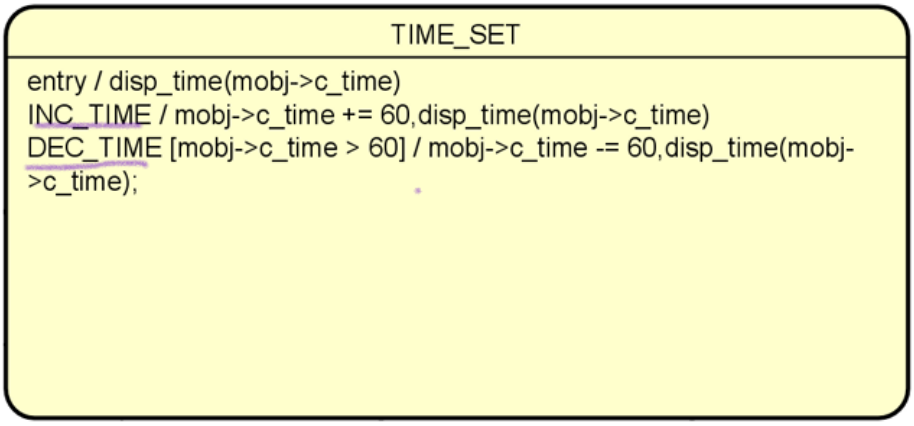
Set of Triggers

A Transition may owen a set of triggers each of which specifies an Event whose occurrence, when dispatched, may trigger traversal of the Transition. A Transition trigger is said to be enabled if the dispatched Event occurrence matches its Event type. When mutile tiggers are defined for a Transition, they are logically disjunctive, tht is, if any of them are enabled, the Transition will be triggered. [OMG® UML 2.5.1].

e1,e2,e3[a > 0] / action_1()

Internal Transition

Internal : internal is a special case of a local Transition that is a self-transition (i.e., with the same source and target States), such that the State is never exited (and, thus, not re-entered), which means that no exit or entry Behaviors are executed when this Transition is executed. This kind of Transition can only be defined if the source Vertex is a State [OMG® UML 2.5.1]



4.4. Events (Triggers)

An **event** is an incident or stimulus that triggers a state machine to transition. These incidents are abstracted as events in the state machine diagram, and they can cause both external and internal transitions. In the operation of a microwave oven, some examples of events include:

1. **Opening the door**: The heater turns off, and the lights turn on.
2. **Closing the door**: The lights turn off.
3. **Setting the timer**: Manages the time for cooking.
4. **Starting**: The heater turns on.

These events lead to specific actions or state changes in the system, such as turning on or off components.

An **event** typically consists of two components:

1. **Signal**: The primary stimulus or incident that triggers the event.
2. **Associated values or parameters (optional)**: Additional information or parameters that can be included with the signal to provide more context or specify particular details about the event.

These components work together to define the event and its associated actions or transitions in a state machine.

User activity	Event generated : SIGNAL	Parameters	note
Press '+' button	INC_TIME	none	This event gets posted to the state machine whenever the user presses the + button
Press - button	DEC_TIME	none	This event gets posted to the state machine whenever the user presses the - button
Press S/P button	START_PAUSE		This event gets posted to the state machine whenever the user presses the S/P button
Press + and – button together	ABRT		This event gets posted to the state machine whenever the user presses the + and – buttons together
	TIME_TICK	ss (sub second)	This event is system generated for every 100ms ss parameter value can vary between 1 to 10 1 → 100ms

4.5. Pseudo states

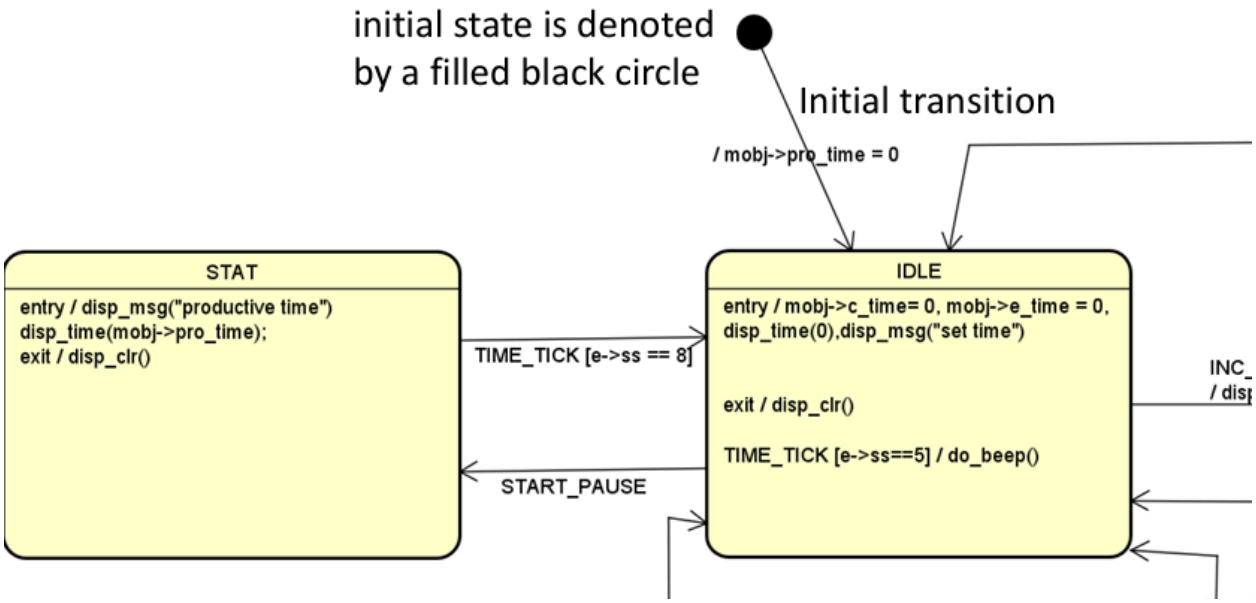
In UML state machines, there are several types of pseudo-states that serve specific functions:

- **Initial**: Marks the starting point of the state machine, where execution begins.
- **Choice**: Allows for branching in the state machine, enabling the transition to different states based on conditions.
- **Join**: Combines multiple parallel transitions into a single state, synchronizing the behavior.
- **Deep History**: Preserves the last active substate in a composite state, restoring the state machine to that point when re-entering the composite state.
- **Shallow History**: Records the last active state in the composite state but does not track the entire substate hierarchy.

For a complete list of pseudo-states, refer to **OMG® UML 2.5.1**

Initial Pseudostate

Initial Pseudostate: Represents the starting point for a **Region** in a state machine. It marks where execution of the contained behavior begins when the Region is entered through default activation. It serves as the source for at most one transition, which may have an associated effect (behavior), but no trigger or guard. A Region can have only one initial vertex.



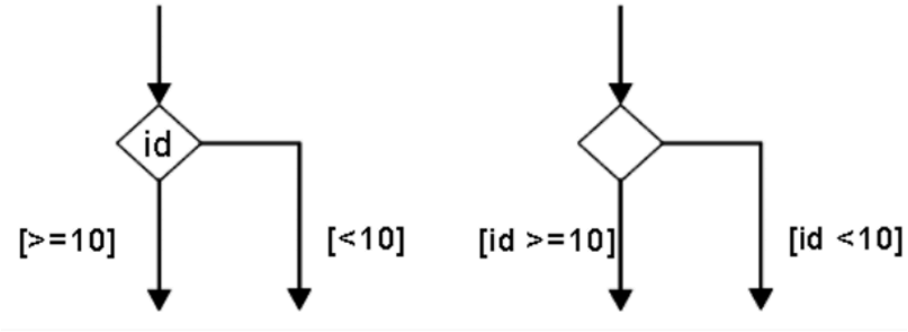
In our example, the **initial transition** leads to the **IDLE** state.

- When the application object is created, we must **manually call a function** that:
  - Executes the action associated with the initial transition.
  - Sets the initial state to **IDLE**.
- This setup must be done **before processing any events**, to ensure the state machine starts in a well-defined state.

Choice Pseudostate

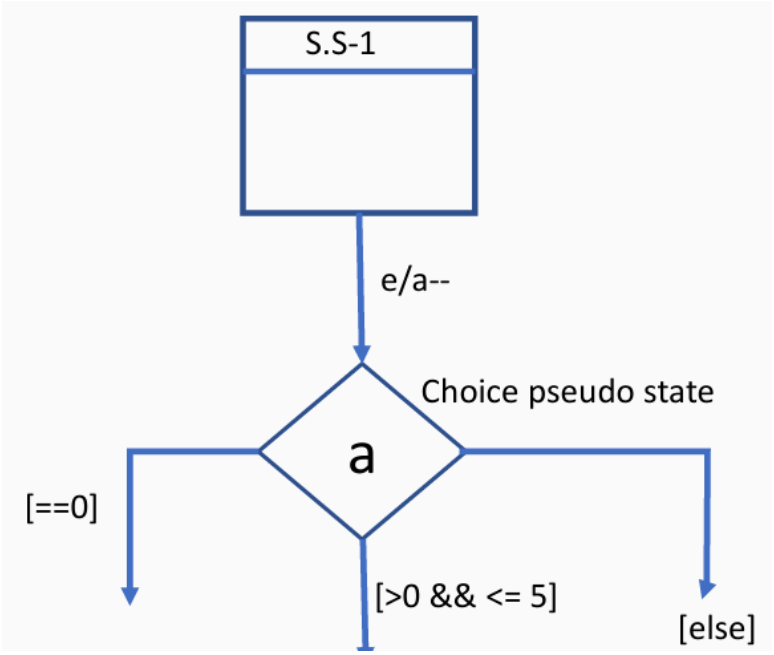
A **Choice Pseudostate** is represented by a **diamond-shaped symbol** in UML state machine diagrams.

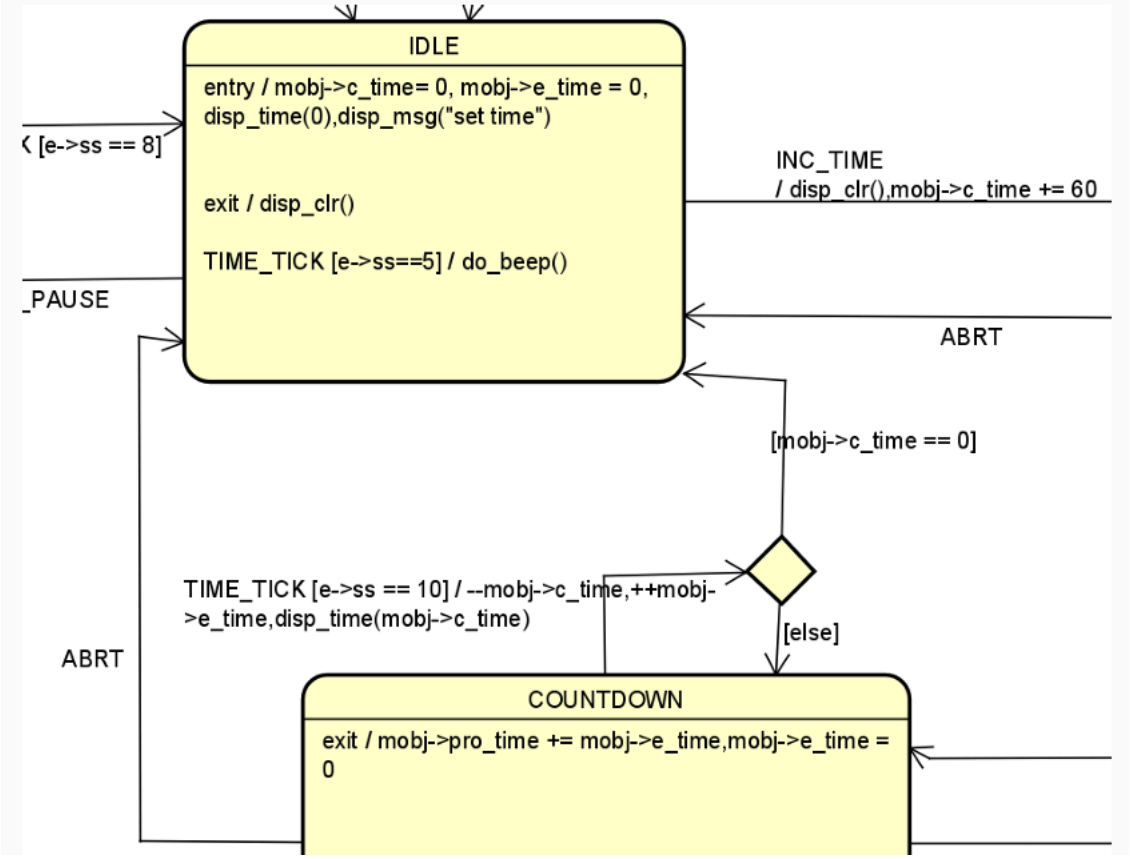
- It has **one incoming transition**.
- It has **two or more outgoing transitions**, each guarded by a condition.
- At runtime, the guard conditions are evaluated, and the transition whose guard evaluates to **true** is taken.
- It is used to model **branching logic** where the next state depends on runtime conditions.



A **Choice Pseudostate** is used to implement **dynamic conditional branching** in a UML state machine.

- It allows **splitting a compound transition** into multiple alternative paths.
- The decision on which path to take depends on the outcome of **Behavior executions** performed earlier in the same compound transition.
- If **none of the guard conditions** evaluates to `true`, the model is considered **ill-formed**.
- If **multiple guards** evaluate to `true`, **one** of the corresponding transitions is selected (non-deterministically).
- It is **recommended** to include one outgoing transition with the predefined **`[else]` guard** to handle any case where all other guards are false.


S.S-1
e/a--
Choice pseudo state
a
[==0]
[>0 && <= 5]
[else]


IDLE
entry / mobj->c_time= 0, mobj->e_time = 0,
disp_time(0),disp_msg("set time")
exit / disp_clr()
TIME_TICK [e->ss==5] / do_beep()
[e->ss == 8]
_PAUSE
INC_TIME
/ disp_clr(),mobj->c_time += 60
ABRT
[mobj->c_time == 0]
TIME_TICK [e->ss == 10] / --mobj->c_time,++mobj-
>e_time,disp_time(mobj->c_time)
[else]
ABRT
COUNTDOWN
exit / mobj->pro_time += mobj->e_time,mobj->e_time =
0

4.6. Implementation of state machine

There are multiple ways to implement state machines in code, each with its strengths and trade-offs. The three common approaches are:

- Nested switch approach
- State table approach
- State handler approach

4.6.1. Nested Switch Approach

- Implements the state machine using **nested `switch-case` or `if-else` statements**.
- The **outer switch** handles the current **state**, and the **inner switch** handles the **event or trigger**.
- Enums are typically used to define all possible states and events, making the code more readable, type-safe, and maintainable.

Structure:

Define Enums for States and Events

```
// Define all possible states
typedef enum {
    STATE_IDLE,
    STATE_TIME_SET,
    STATE_COUNTDOWN,
    STATE_PAUSE,
    STATE_STAT
} State;

// Define all possible events
typedef enum {
    EVENT_START,
    EVENT_PAUSE,
    EVENT_RESET,
    EVENT_TIMEOUT
```

```
} Event;
```

Using enums avoids the use of magic numbers or strings, enhances code readability, and allows compile-time checking

Simple implementation

```
void processEvent(State currentState, Event event) {
    switch (currentState) {
        case STATE_IDLE:
            switch (event) {
                case EVENT_START:
                    // Perform action
                    startCountdown();
                    currentState = STATE_COUNTDOWN;
                    break;
                case EVENT_RESET:
                    resetTimer();
                    currentState = STATE_IDLE;
                    break;
            }
            break;

        case STATE_COUNTDOWN:
            switch (event) {
                case EVENT_PAUSE:
                    pauseCountdown();
                    currentState = STATE_PAUSE;
                    break;
                case EVENT_TIMEOUT:
                    notifyUser();
                    currentState = STATE_STAT;
                    break;
            }
            break;

        // Add cases for other states similarly
```

```
    }
}
```

Advantages:

- Easy to implement and understand.
- Enums improve readability, avoid errors, and make debugging easier.
- Good for quick prototypes or small applications.

Disadvantages:

- Hard to scale for complex systems.
- Violates the open-closed principle (difficult to extend without modifying code).
- Can become hard to maintain as the number of states/events grows.

4.6.2. State Table Approach

The **State Table Approach** models the state machine behavior as a **lookup table** (often a 2D array or list of records). Each entry in the table defines:

- Current State
- Event (Trigger)
- Next State
- Action to Perform

When an event occurs, the program looks up the current state and event in the table, retrieves the corresponding next state and action, performs the action, and updates the state.

Example Structure (C-pseudocode)

```
typedef struct {
    State currentState;
    Event event;
    State nextState;
    void (*action)(void);  // Function pointer to the action
```

```
} StateTableEntry;
```

A sample table might look like:

Current State	Event	Next State	Action
IDLE	START	COUNTDOWN	startTimer
COUNTDOWN	PAUSE	PAUSE	pauseTimer
PAUSE	START	COUNTDOWN	resumeTimer
COUNTDOWN	TIMEOUT	STAT	displayStats
STAT	RESET	IDLE	resetApp

This would be represented in code as an array of `StateTableEntry`.

```
StateTableEntry stateTable[] = {
    {STATE_IDLE, EVENT_START, STATE_COUNTDOWN, startTimer},
    {STATE_COUNTDOWN, EVENT_PAUSE, STATE_PAUSE, pauseTimer},
    {STATE_PAUSE, EVENT_START, STATE_COUNTDOWN, resumeTimer},
    {STATE_COUNTDOWN, EVENT_TIMEOUT, STATE_STAT, displayStats},
    {STATE_STAT, EVENT_RESET, STATE_IDLE, resetApp},
    // Add more entries as needed
};

void handleEvent(State *currentState, Event event) {
    for (int i = 0; i < sizeof(stateTable)/sizeof(StateTableEntry);
i++) {
        if (stateTable[i].currentState == *currentState &&
            stateTable[i].event == event) {

            // Perform the action
            if (stateTable[i].action != NULL) {
                stateTable[i].action();
            }
```

```
            // Update the current state
            *currentState = stateTable[i].nextState;
            return;
        }
    }

    // No matching transition found
    // Do something
}
```

Advantages:

- **Separation of logic and structure**: All transitions and actions are defined in one place.
- **Easy to modify**: You can add new transitions without modifying core logic.
- **Scalable**: Suitable for complex systems with many states/events.
- **Readable and maintainable**: Especially with large numbers of states/events.

Disadvantages:

- Slightly more complex to implement than the nested switch approach.
- May be less intuitive for beginners compared to `switch-case`.

4.6.3. State Handler Approach (Function Pointer Approach)

In the **State Handler Approach**, each state is implemented as a separate **function** (called a **state handler**). The state machine maintains a pointer to the **current state handler function**. On receiving an event, the state handler processes it, performs any actions, and optionally transitions to a new state by updating the pointer to a different state function.

Core Elements

- **State handler functions**: Implemented as function pointers; each one represents a unique state and handles all events relevant to that state.
- **Event dispatcher**: Calls the current state handler with the incoming event.

- **State transitions**: Achieved by changing the function pointer to point to a different state function.

Example of Implementation

Define Types:

```
typedef enum {
    EVENT_START,
    EVENT_PAUSE,
    EVENT_TIMEOUT,
    EVENT_RESET
} Event;

typedef void (*StateHandler)(Event);  // Function pointer type

StateHandler currentStateHandler;
```

State Functions:

```
void stateIdle(Event event) {
    switch (event) {
        case EVENT_START:
            printf("Transition: IDLE -> COUNTDOWN\n");
            startTimer();
            currentStateHandler = stateCountdown;
            break;
        default:
            printf("Invalid event in IDLE state.\n");
    }
}

void stateCountdown(Event event) {
    switch (event) {
        case EVENT_PAUSE:
            printf("Transition: COUNTDOWN -> PAUSE\n");
            pauseTimer();
```

```
            currentStateHandler = statePause;
            break;
        case EVENT_TIMEOUT:
            printf("Transition: COUNTDOWN -> STAT\n");
            displayStats();
            currentStateHandler = stateStat;
            break;
        default:
            printf("Invalid event in COUNTDOWN state.\n");
    }
}

void statePause(Event event) {
    switch (event) {
        case EVENT_START:
            printf("Transition: PAUSE -> COUNTDOWN\n");
            resumeTimer();
            currentStateHandler = stateCountdown;
            break;
        case EVENT_RESET:
            printf("Transition: PAUSE -> IDLE\n");
            resetApp();
            currentStateHandler = stateIdle;
            break;
        default:
            printf("Invalid event in PAUSE state.\n");
    }
}

void stateStat(Event event) {
    switch (event) {
        case EVENT_RESET:
            printf("Transition: STAT -> IDLE\n");
            resetApp();
            currentStateHandler = stateIdle;
            break;
        default:
            printf("Invalid event in STAT state.\n");
```

```
    }
}
```

Event Dispatcher:

```
void dispatchEvent(Event event) {
    if (currentStateHandler != NULL) {
        currentStateHandler(event);
    }
}
```

Main Function Simulation:

```
int main() {
    currentStateHandler = stateIdle;  // Set the initial state

    while (1) {
       // Simulate or wait for the next event
        Event event = getNextEvent();
        // Handle the event in the current state
        dispatchEvent(event);
    }

    return 0;
}
```

Summary

Element	Role
`StateHandler`	Function pointer type for state logic
`currentStateHandler`	Pointer to current state logic
`dispatchEvent()`	Calls the current state function with the event

`getNextEvent()`	Reactively waits for and returns the next event
`main()`	Runs a loop, fetches events, and dispatches them to the state logic

Advantages:

- **Encapsulation**: Each state's logic is isolated in its own function, improving readability and maintainability.
- **Polymorphic behavior**: States act like polymorphic objects—useful in C and embedded systems without classes.
- **Dynamic state transitions**: Easily change state at runtime by reassigning function pointers.

Disadvantages:

- Slightly more complex to understand for beginners.
- Debugging may be harder if function pointers are misused.
- Harder to visualize compared to state tables.

When to Use:

- In embedded systems or low-level C programming where you want fast, efficient, and modular state logic.
- In systems with frequent state transitions and state-specific event handling logic.

CHAPTER 8 Sensor and Actuator Control Strategies in Embedded Systems

1. Introduction

In embedded systems, **control strategies** define how a system behaves in response to inputs and changes in the environment. At the heart of any control system is the goal of regulating a physical process or variable, such as temperature, speed, or pressure. The two most fundamental approaches are **open-loop control** and **closed-loop (feedback) control**.

Understanding the difference between these strategies is crucial for designing reliable and efficient embedded systems.

2. What is a Controller?

In the context of embedded systems, a controller refers to a piece of software that continuously monitors and influences the operational conditions of a dynamic system. This system typically includes both the hardware of a device and the surrounding environment in which it operates. The controller's role is to ensure the system behaves as desired, often through a feedback mechanism.

A common example of a controller in action is the heating system of a room, which functions as a closed-loop control system. In this scenario, a temperature sensor is used to monitor the current temperature of the room. The controller—implemented as an algorithm running on a programmable thermostat, such as one based on a microcontroller—receives this temperature data, compares it to the desired value, and makes decisions on whether to activate or deactivate the heater. The heater then acts upon these decisions to adjust the room temperature. The complete dynamical system consists of the room itself, the temperature sensor, the thermostat running the control algorithm, and the heater.

3. Open-Loop Control Systems

An **open-loop control system** is one in which the control action is **independent** of the output. The system does not measure or respond to the actual result of its actions. Instead, it performs its operation based solely on predefined instructions.

Characteristics

- **No feedback** is used to monitor output.
- **Simple and low-cost** implementation.
- **Faster response**, since no measurement or correction is required.
- **Less accurate**, especially when disturbances or environmental changes occur.

Example: Washing Machine

A washing machine with a fixed cycle time is an open-loop system. Once started, it runs through all stages (wash, rinse, spin) regardless of whether the clothes are clean or how much load there is.

Applications

- Motor timers
- LED blinking patterns
- Alarm buzzers
- Heating systems without thermostats

4. Closed-Loop (Feedback) Control Systems

A **closed-loop control system**, also known as a **feedback control system**, monitors the output and compares it with the desired value. The system then **adjusts its input** based on the difference (error) between the actual output and the setpoint.

Characteristics

- **Feedback** is essential for self-correction.
- **More accurate** and robust in the presence of disturbances.

- **Complex and costlier** to implement.
- May require **sensor calibration** and controller tuning.

Example: Room Thermostat

A room heating system with a thermostat is a classic closed-loop control system. The thermostat senses room temperature and turns the heater on or off to maintain the target temperature.

Feedback Loop Components

1. **Sensor** – Measures the current output (e.g., temperature).
2. **Controller** – Compares the measured output to the desired value.
3. **Actuator** – Adjusts the process to reduce the error (e.g., turns heater on/off).

Applications

- Cruise control in cars
- Air conditioning systems
- Robot arms with position correction
- Voltage regulators

5. Control Algorithm Types in Closed-Loop Systems

Closed-loop systems often use algorithms to minimize the error over time. Common control algorithms include:

- **Proportional (P)**: Correction is proportional to the error.
- **Integral (I)**: Correction accounts for accumulated past error.
- **Derivative (D)**: Correction based on rate of change of error.
- **PID Controller**: Combines P, I, and D for precise control.

5.1. Proportional (P) Control

In Proportional (P) control, the corrective action applied by the controller is directly proportional to the magnitude of the error, which is the difference between the desired

setpoint and the measured process variable. Mathematically, the control signal is given by:

u(t) = Kp × e(t)

where **u(t)** is the control signal, **Kp** is the proportional gain, and **e(t)** is the error at time *t*. This method ensures that the larger the error, the stronger the correction, enabling faster response. However, P control alone may not eliminate the steady-state error completely, especially in systems requiring high accuracy.

Example: Temperature Control Using Proportional (P) Controller

An embedded system is used to maintain the temperature of a room using a heater. The desired temperature (setpoint) is **25°C**, and the current measured temperature is **20°C**.

Controller Parameters

- Proportional gain (**Kp**) = **2**
- Error (**e**) = Setpoint – Measured = 25°C – 20°C = **5°C**
- Control signal (**u**) = **Kp × e = 2 × 5 = 10 units**

The output control signal is **10 units**, which may correspond, for instance, to a **10% increase in heater power** (depending on the system calibration). The larger the error, the stronger the correction. If the temperature were only 1°C below the setpoint, the correction would be smaller:

e = 1°C, then **u = 2 × 1 = 2 units** → only **2% power increase**

5.1.1. Embedded System Implementation

- **Microcontroller**: ATmega328P
- **Sensor**: LM35 (10 mV/°C output, connected to ADC channel 0)
- **Actuator**: PWM output (e.g., OC0A pin) to control heater
- **Sampling Rate**: 1 Hz (1 sample/second)
- **Setpoint**: 25°C
- **Kp**: 10 (tunable)

```c
#include <avr/io.h>
#include <util/delay.h>

#define SETPOINT 25         // Desired temperature in °C
#define KP 10               // Proportional gain

void ADC_Init() {
    ADMUX = (1<<REFS0);                 // AVcc reference, ADC0 input
    ADCSRA = (1<<ADEN) | (1<<ADPS2);    // Enable ADC, prescaler = 16
}

uint16_t ADC_Read() {
    ADCSRA |= (1<<ADSC);                // Start conversion
    while (ADCSRA & (1<<ADSC));         // Wait for conversion
    return ADC;
}

void PWM_Init() {
    DDRD |= (1<<PD6);                   // OC0A (PD6) as output
    TCCR0A = (1<<COM0A1) | (1<<WGM00); // Fast PWM, non-inverting
    TCCR0B = (1<<CS01);                 // Prescaler = 8
}

void set_heater_power(uint8_t duty) {
    OCR0A = duty;  // Set PWM duty cycle (0-255)
}

int main(void) {
    uint16_t adc_val;
    float temperature;
    int error;
    int control_signal;

    ADC_Init();
    PWM_Init();

    while (1) {
        adc_val = ADC_Read();           // Read ADC value (0-1023)
```

```
        temperature = (adc_val * 5.0/1024.0) * 100.0;//Convert to °C

        error = SETPOINT - (int)temperature;
        control_signal = KP * error;

        // Clamp the control signal to 0-255 range
        if (control_signal > 255) control_signal = 255;
        if (control_signal < 0) control_signal = 0;

        set_heater_power((uint8_t)control_signal);

        _delay_ms(1000); // Sample every 1 second
    }
}
```

Explanation

- The LM35 outputs **10 mV/°C** → 25°C = 250 mV → ADC ≈ 51
- ADC value is scaled to get temperature in °C
- Proportional control output is mapped directly to **PWM duty cycle**
- The heater is controlled by setting PWM output proportional to error

5.2. Integral (I) Control

Integral control is one of the three main components of the PID (Proportional-Integral-Derivative) control strategy. It addresses a limitation of pure **Proportional (P)** control: the **steady-state error** — the difference that remains between the desired output (setpoint) and the actual output even after a long time.

The **Integral term** adds up (integrates) the **past errors over time** and uses this sum to adjust the control output. This means:

> If a system has a consistent small error, the integral term will **accumulate** this error and **gradually increase the control effort** until the error is corrected.

Mathematical Expression

If e(t) is the error at time t (difference between setpoint and process value), the integral control output I(t) is:

$$I(t) = K_i \cdot \int_0^t e(\tau)\, d\tau$$

Where:

- k_i: Integral gain
- $\int_0^\tau e(\tau)\, d\tau$: Cumulative (historical) error

In **discrete time** (e.g., with a microcontroller sampling once per second):

$$I[k] = I[k-1] + K_i \cdot e[k] \cdot T$$

- **T**: Sampling interval
- **e[k]**: Error at time step kkk

Example

- **Setpoint** = 50°C
- **Current temperature** = 48°C
- So, **Error (e)** = 2°C
- **Ki** = 5
- **Sampling time (T)** = 1 second

Integral Calculation (after 3 samples):

$$I = 5 \cdot (2 + 2 + 2) \cdot 1 = 30$$

This means after 3 seconds, the integral term has increased the control signal by **30 units** to compensate for the persistent 2°C error.

5.3. Embedded System Example: Integral Control

Maintain a target temperature (e.g., 50°C) using only **integral control**. The heater output is adjusted based on the **accumulated error** over time.

System Components

- Microcontroller: ATmega328P
- Sensor: LM35 temperature sensor (10 mV/°C)
- Actuator: Heater controlled via PWM (Timer0 on ATmega328P)
- Sampling Time: 1 second
- Setpoint: 50°C

Metrics Example

Time (s)	Temp (°C)	Error (°C)	Integral Output (PWM Duty)
0	46.0	4.0	20
1	46.5	3.5	37.5
2	47.5	2.5	50
3	48.5	1.5	57.5
4	49.2	0.8	61.5
5	49.8	0.2	62.5
6	50.0	0.0	62.5

- The heater PWM output increases **cumulatively** even if the error is small.
- As the temperature approaches the setpoint, the error diminishes, and the integral stops growing.
- **No steady-state error** remains — a core strength of integral control.
- However, if the temperature overshoots, **integral windup** could occur.

Integral Control Code Snippet in C

```c
#include <avr/io.h>
#include <util/delay.h>

#define SETPOINT 50.0     // Desired temperature in °C
#define Ki 5.0            // Integral gain
#define SAMPLING_TIME 1.0 // seconds

float integral = 0.0;

// Initialize ADC for LM35
void ADC_Init() {
    ADMUX = (1<<REFS0); // AVcc as reference
    ADCSRA = (1<<ADEN) | (1<<ADPS2); // Enable ADC, prescaler 16
}

// Read temperature from LM35
float Read_Temperature() {
    ADMUX &= 0xF0; // Select ADC0
    ADCSRA |= (1<<ADSC); // Start conversion
    while (ADCSRA & (1<<ADSC));
    uint16_t adc = ADC;
    return (adc * 5.0 / 1024.0) * 100.0; // Convert to °C
}

// Initialize PWM for heater (Timer0)
void PWM_Init() {
    DDRD |= (1 << PD6); // PWM output pin
    TCCR0A = (1<<COM0A1) | (1<<WGM01) | (1<<WGM00); // Fast PWM
    TCCR0B = (1<<CS01); // Prescaler 8
}

// Set PWM duty cycle (0 to 255)
void Set_Heater(uint8_t duty) {
    OCR0A = duty;
}
```

```
int main(void) {
    ADC_Init();
    PWM_Init();

    while (1) {
        float current_temp = Read_Temperature();
        float error = SETPOINT - current_temp;

        // Integrate the error
        integral += error * Ki * SAMPLING_TIME;

        // Clamp output to PWM limits
        if (integral > 255) integral = 255;
        if (integral < 0) integral = 0;

        Set_Heater((uint8_t)integral);

        _delay_ms(1000); // 1-second sampling rate
    }
}
```

5.4. Derivative (D) Control

The **derivative controller** provides **correction based on how fast the error is changing**, not just the size of the error itself.

It **predicts future error** by observing the rate of change of the current error. If the error is changing rapidly, the derivative term acts **proactively** to prevent overshooting.

$$D_{\text{output}} = K_d \cdot \frac{de(t)}{dt}$$

Where:

k_d = Derivative gain

$\frac{de(t)}{dt}$ = Rate of change of error

The derivative control helps **dampen oscillations, reduces overshoot, and** stabilizes systems where rapid changes in error could lead to instability

5.4.1. Embedded Systems Example

We are designing a temperature controller with:

- ATmega328P
- LM35 temperature sensor
- PWM-controlled heater

You've already implemented P or PI control but now you want to add D to reduce overshoot during fast changes.

Example Metrics

Time (s)	Temp (°C)	Error	d(Error)/dt	Derivative Output
0	45.0	5.0	N/A	0
1	46.0	4.0	-1.0	-10.0
2	47.5	2.5	-1.5	-15.0
3	48.8	1.2	-1.3	-13.0
4	49.6	0.4	-0.8	-8.0

> You can see how the derivative term **slows down** the actuator even if the error is still positive, avoiding overshoot.

Implementation Snippet in C

```
#define SETPOINT 50.0       // °C
#define Kd 10.0             // Derivative gain
#define SAMPLING_TIME 1.0   // in seconds
```

```
float previous_error = 0.0;

float compute_derivative(float current_error) {
    float derivative = (current_error - previous_error) /
SAMPLING_TIME;
    previous_error = current_error;
    return Kd * derivative;
}
```

And then in your main loop:

```
float error = SETPOINT - current_temp;
float d_output = compute_derivative(error);
float output = base_output + d_output;  // Combine with P or PI
```

Clamp `output` before applying it to the PWM to prevent overflow.

5.5. Proportional-Integral-Derivative (PID) Controller

A **PID controller** is a **feedback-based control algorithm** that continuously calculates an **error value** as the difference between a desired setpoint and a measured process variable (e.g., temperature, speed, position). It then applies a correction based on three components:

1) **Proportional (P)**: Responds to the current error. The bigger the error, the stronger the correction.

$$P_{\text{output}} = K_p \cdot e(t)$$

2) **Integral (I)**: Responds to the accumulated error over time, helping eliminate long-term steady-state error.

$$I_{\text{output}} = K_i \cdot \int_0^t e(\tau)\, d\tau$$

3) **Derivative (D)**: Responds to the rate of change of the error. It helps predict future error and reduce overshoot.

$$D_{\text{output}} = K_d \cdot \frac{de(t)}{dt}$$

The Total **PID** Output

$$\text{Output}(t) = P + I + D = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}$$

Why PID controller is used?

- **Smooth, precise control** even in nonlinear or time-varying systems
- Compensates for both fast error changes and long-term drift
- Most widely used control algorithm in **embedded systems**, **robotics**, **industrial automation**, and **mechatronics**

5.5.1. Embedded System Example

You're using:

- **Microcontroller**: ATmega328P
- **Sensor**: LM35 to measure temperature
- **Actuator**: PWM-controlled heating element
- **Goal**: Maintain 50°C with minimal oscillation and quick stabilization

Example Metrics

Time (s)	Temp (°C)	Error	P	I	D	PID Output
0	45.0	5.0	10	2.5	5.0	17.5
1	46.5	3.5	7	4.25	-1.5	9.75

2	48.0	2.0	4	5.25	-1.5	7.75
3	49.0	1.0	2	5.75	-1.0	6.75
4	50.0	0.0	0	5.75	-1.0	4.75

Notice how:

- **P** shrinks as error decreases
- **I** grows slowly to eliminate steady-state error
- **D** reacts to the rate of temperature increase, slowing it down

Simplified PID in C

```c
#define SETPOINT 50.0
#define KP 2.0
#define KI 0.5
#define KD 1.0
#define SAMPLE_TIME 1.0  // 1 second

float previous_error = 0.0;
float integral = 0.0;

float compute_PID(float current_temp) {
    float error = SETPOINT - current_temp;

    // Integral term
    integral += error * SAMPLE_TIME;

    // Derivative term
    float derivative = (error - previous_error) / SAMPLE_TIME;

    // PID output
    float output = KP * error + KI * integral + KD * derivative;

    previous_error = error;

    return output;
```

```
}
```

You would then:

- Convert `output` to a PWM duty cycle
- Apply it to your heating element

References

Valvano, J. W. (2012). *Embedded systems: Introduction to ARM Cortex-M microcontrollers* (Vol. 1). CreateSpace Independent Publishing.

White, E. (2011). *Making embedded systems: Design patterns for great software*. O'Reilly Media.

Mazidi, M. A., Naimi, S., & Naimi, S. (2011). *The AVR microcontroller and embedded systems: Using assembly and C for ATMega*. Pearson Education.

Barnett, R. H., Cox, S., & O'Cull, L. (2007). *Embedded C programming and the Atmel AVR*. Thomson Delmar Learning.

Wilmshurst, T. (2010). *Designing embedded systems with PIC microcontrollers: Principles and applications* (2nd ed.). Newnes.

Ganssle, J. (2008). *The art of designing embedded systems* (2nd ed.). Newnes.

Pont, M. J. (2001). *Embedded C*. Addison-Wesley.

Yalamanchili, S. (2010). *Fundamentals of embedded systems design*. CreateSpace.

Microchip Technology Inc. (2016). *ATmega328P datasheet*. Retrieved from https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf