

### 1 A brief history of the AVR microcontroller

The basic architecture of AVR was designed by two students of Norwegian Institute of Technology (NTH), Alf-Egil Bogen and Vegard Wollan, and then was bought and developed by Atmel in 1996. There are many kinds of AVR microcontroller with different properties. Except for AVR32, which is a 32-bit microcontroller, AVRs are all 8-bit microprocessors, meaning that the CPU can work on only 8 bits of data at a time. Data larger than 8 bits has to be broken into 8-bit pieces to be processed by the CPU. One of the problems with the AVR microcontrollers is that they are not all 100% compatible in terms of software when going from one family to another family. AVRs are generally classified into four broad groups: Mega, Tiny, Special purpose, and Classic. Here, we will focus on ATmega32 since it is powerful, widely available, and comes in DIP packages, which makes it ideal for educational purposes.

#### 1.1 AVR Features

The AVR microcontroller family, is widely regarded for its flexibility, efficiency, and ease of use, making it a popular choice for a variety of applications. Some of the major features that make AVR microcontrollers ideal for different applications are:

##### 1. High-Performance Architecture

AVR microcontrollers are based on RISC architecture, enabling faster execution with single-cycle instructions and high clock speeds for real-time applications.

##### 2. Wide Range of Models

Available in 8-bit, 16-bit, and 32-bit variants, AVR microcontrollers cater to diverse applications and come in multiple package types like DIP, QFN, and TQFP.

##### 3. Efficient Power Management

These microcontrollers have low power consumption and support multiple sleep modes, making them suitable for battery-powered devices.

##### 4. Integrated Peripherals

Built-in peripherals like ADCs, PWM, timers, and communication protocols (USART, SPI, I2C) allow easy integration into various systems.

##### 5. Flash Memory

In-system programmable flash memory enables reprogramming without removing the chip and provides ample storage for application code.

##### 6. EEPROM and SRAM

Non-volatile EEPROM retains data after power-off, while built-in SRAM supports runtime data processing efficiently.

### 7. Interrupt Handling

Hardware interrupts with priority mechanisms ensure responsive and efficient real-time processing.

### 8. Development Tools

AVR microcontrollers are supported by Atmel Studio and third-party IDEs, offering robust development and debugging environments.

### 9. Ease of Programming

These microcontrollers support popular languages like C and Assembly, and bootloader functionality allows for easy firmware updates.

### 10. Community Support

A strong community and extensive resources, including tutorials and forums, help in learning and troubleshooting.

## 1.2 General Architecture of AVR Microcontrollers

AVR microcontrollers are a family of microcontrollers based on a high-performance **RISC (Reduced Instruction Set Computing)** architecture. These controllers are categorized into four major groups: **Classic**, **Mega**, **Tiny**, and **Special Purpose**. While they differ in terms of features, peripherals, and resources, the majority of them share the same core architecture, with more advanced versions offering additional functionalities. Some of the common features found in these microcontrollers are:

### 1. Core Architecture

All AVR microcontrollers are based on a **Harvard architecture**, which separates program memory (Flash) and data memory (SRAM), enabling simultaneous access for faster execution. The key components of the core architecture include:

- **RISC-Based Design:**
  - Executes most instructions in a single clock cycle.
  - Features a compact and efficient instruction set, typically with about 130 instructions.
- **General Purpose Registers:**

# Programming for Embedded Systems

## Unit 2

- Includes 32 working registers that directly interface with the Arithmetic Logic Unit (ALU), ensuring fast computations.

### 2. Program and Data Memory

- **Flash Memory:**
  - Non-volatile memory used for storing program code.
  - Supports in-system programming and self-programming through bootloaders.
- **SRAM:**
  - Provides fast-access volatile memory for temporary data storage during program execution.
- **EEPROM:**
  - Retains data after power-off, making it suitable for storing user preferences or configuration data.

### 3. Interrupt System

AVR microcontrollers feature a robust interrupt system that includes:

- Support for multiple interrupt sources.
- A **Global Interrupt Enable (GIE)** mechanism to prioritize and manage interrupt handling efficiently.

### 4. Peripherals and Features

- **Timers/Counters:**
  - Available in all AVR families, enabling precise time-keeping and control operations.
- **Analog-to-Digital Converters (ADC):**
  - Converts analog sensor inputs into digital data for processing.
- **PWM Modules:**
  - Essential for motor control, LED dimming, and signal generation.
- **Communication Interfaces:**
  - **USART:** For serial communication.
  - **SPI and I2C:** For interfacing with external devices like sensors and memory modules.
- **GPIO Pins:**
  - Configurable as input or output for controlling or sensing devices.

### 5. Clock System

- Supports **internal oscillators** for cost-effective designs.
- External crystal oscillators provide precise clock signals for applications requiring high accuracy.

### 6. Power Management

- AVR microcontrollers include multiple **power-saving modes**, such as Idle and Sleep, making them ideal for battery-powered applications.

The AVR microcontroller family offers a scalable and versatile platform with a shared core architecture, catering to a wide range of applications. Among the different AVR controllers, the **ATmega32**, part of the Mega series, is particularly popular for its balance of performance, features, and ease of use. In the subsequent sections, we will discuss the ATmega32 in detail, exploring its architecture, features, and applications.

### 1.3 Introduction to ATmega32

The **ATmega32 microcontroller** stands out as a versatile and efficient solution for embedded system applications, offering an impressive blend of memory capacity, advanced peripherals, and power management features. Designed with a **RISC-based architecture**, it delivers high performance while maintaining low power consumption, making it ideal for tasks ranging from motor control to temperature monitoring and beyond.

### 1.4 AVR Microcontroller (ATmega32) Pin Diagram

The **ATmega32 microcontroller** comes with a versatile pin configuration that includes four primary ports: **Port-A**, **Port-B**, **Port-C**, and **Port-D**. These ports consist of 8 pins each, providing a total of 32 general-purpose **I/O** lines, along with several specialized pins for analog input, clock, reset, and reference purposes. Below is a detailed explanation of the pin assignments for each port and special functions.

# Programming for Embedded Systems

## Unit 2

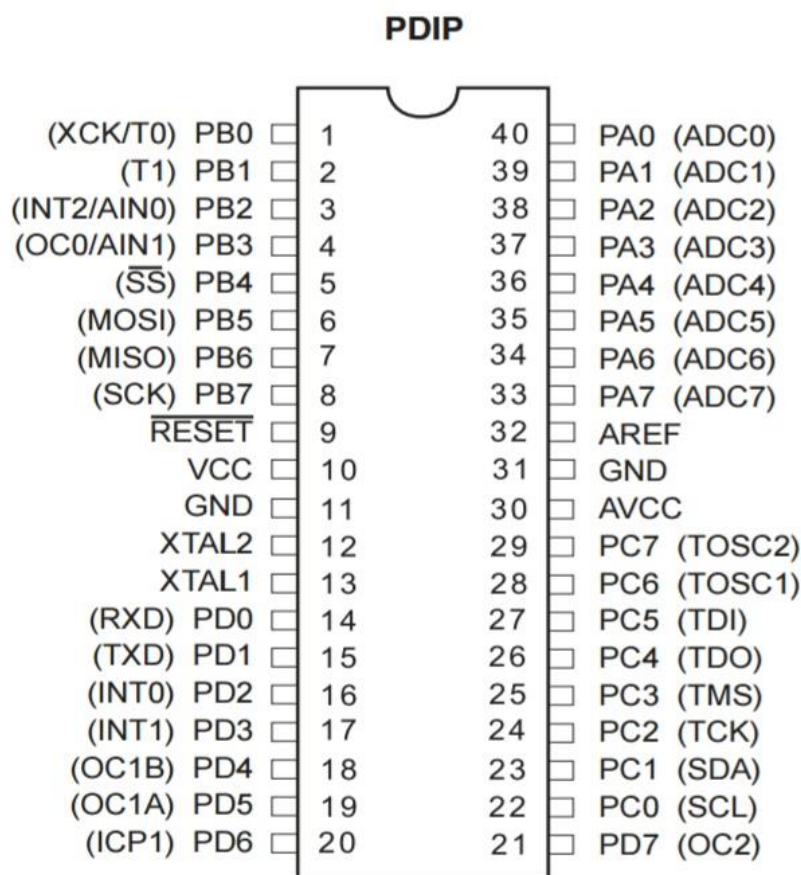


Figure 1-1: Pin Diagram of ATMEGA 32

### 1. Port A (PA0 - PA7) - Pins 33 to 40

- **Major Function:** Analog input for the ADC (Analog-to-Digital Converter) channels (ADC0-ADC7).
- **Alternative Functions:**
  - General Purpose I/O (GPIO).
  - Can serve as digital inputs or outputs when the ADC is not used.

### 2. Port B (PB0 - PB7) - Pins 1 to 8

- **Major Function:** Digital I/O.
- **Alternative Functions:**
  - **SPI Interface:**
    - PB4 (MISO): Master in Slave Out.
    - PB5 (MOSI): Master Out Slave In.
    - PB6 (SCK): Serial Clock.
    - PB7 ( $\overline{SS}$ ): Slave Select.

# Programming for Embedded Systems

## Unit 2

- **Timer/Counter:**
  - PB3 (OC0): Output Compare Match for Timer/Counter0.
  - PB0 and PB1 (ICP1 and T1): Input Capture for Timer/Counter1.
- **External interrupts:**
  - PB2 (/INT2): External Interrupt Request 2.

### 3. Port C (PC0 - PC7) - Pins 22 to 29

- **Major Function:** Digital I/O.
- **Alternative Functions:**
  - JTAG(Joint Test Action Group) Interface (used for debugging and programming):
    - PC2 (TCK): JTAG Test Clock.
    - PC3 (TMS): JTAG Test Mode Select.
    - PC4 (TDO): JTAG Test Data Out.
    - PC5 (TDI): JTAG Test Data In.
  - ADC Reference Voltage:
    - PC0 (AREF): Reference voltage input for the ADC.
  - Reset:
    - PC6 (/RESET): Active low reset pin.

### 4. Port D (PD0 - PD7) - Pins 14 to 21

- **Major Function:** Digital I/O.
- **Alternative Functions:**
  - **USART Communication:**
    - PD0 (RXD): Receive Data.
    - PD1 (TXD): Transmit Data.
  - **External interrupts:**
    - PD2 (/INT0): External Interrupt Request 0.
    - PD3 (/INT1): External Interrupt Request 1.
  - **Timer/Counter:**
    - PD4 (OC1B): Output Compare Match for Timer/Counter1 Channel B.
    - PD5 (OC1A): Output Compare Match for Timer/Counter1 Channel A.
    - PD6 (ICP1): Input Capture for Timer/Counter1.

# Programming for Embedded Systems

## Unit 2

- PD7 (OC2): Output Compare Match for Timer/Counter2.

### 5. VCC - Pin 10

- **Major Function:** Supply voltage (+5V for operation).

### 6. GND (Ground) - Pins 11 and 31

- **Major Function:** Ground reference for the circuit.

### 7. AVCC - Pin 30

- **Major Function:** Power supply for the ADC and analog comparator.
- **Note:** Must be connected to VCC for ADC operation.

### 8. AREF - Pin 32

- **Major Function:** Reference voltage for the ADC.

### 9. XTAL1 and XTAL2 - Pins 12 and 13

- **Major Function:**
  - XTAL1: Input to the inverting oscillator amplifier.
  - XTAL2: Output from the inverting oscillator amplifier.
- **Alternative Function:** These pins are used to connect an external crystal oscillator or clock source.

## Key Notes

1. **GPIO Configuration:** All I/O pins can be configured as input or output using **DDRx** registers (where x is the port letter: A, B, C, or D).
2. **Pull-up Resistors:** Internal pull-up resistors can be enabled for input pins by setting the corresponding **PORTx** register bit when **DDRx** is configured as input.
3. **Alternate Functions:** The alternate functions of pins are prioritized, which means you cannot use a pin for multiple functions simultaneously.
4. **Reset Pin:** PC6 can be disabled as a reset pin to be used as a general-purpose I/O but at the cost of losing external reset functionality.
5. **AVR Programming:** SPI pins (PB4, PB5, PB6, PB7) are also used for in-system programming (ISP).

## 1.5 Architecture of AVR (ATmega32) Microcontroller

The AVR microcontroller, including the ATmega32, is designed with an **Advanced RISC (Reduced Instruction Set Computing)** architecture. It is optimized for efficient

# Programming for Embedded Systems

## Unit 2

performance and provides an excellent balance between computational power and ease of use.

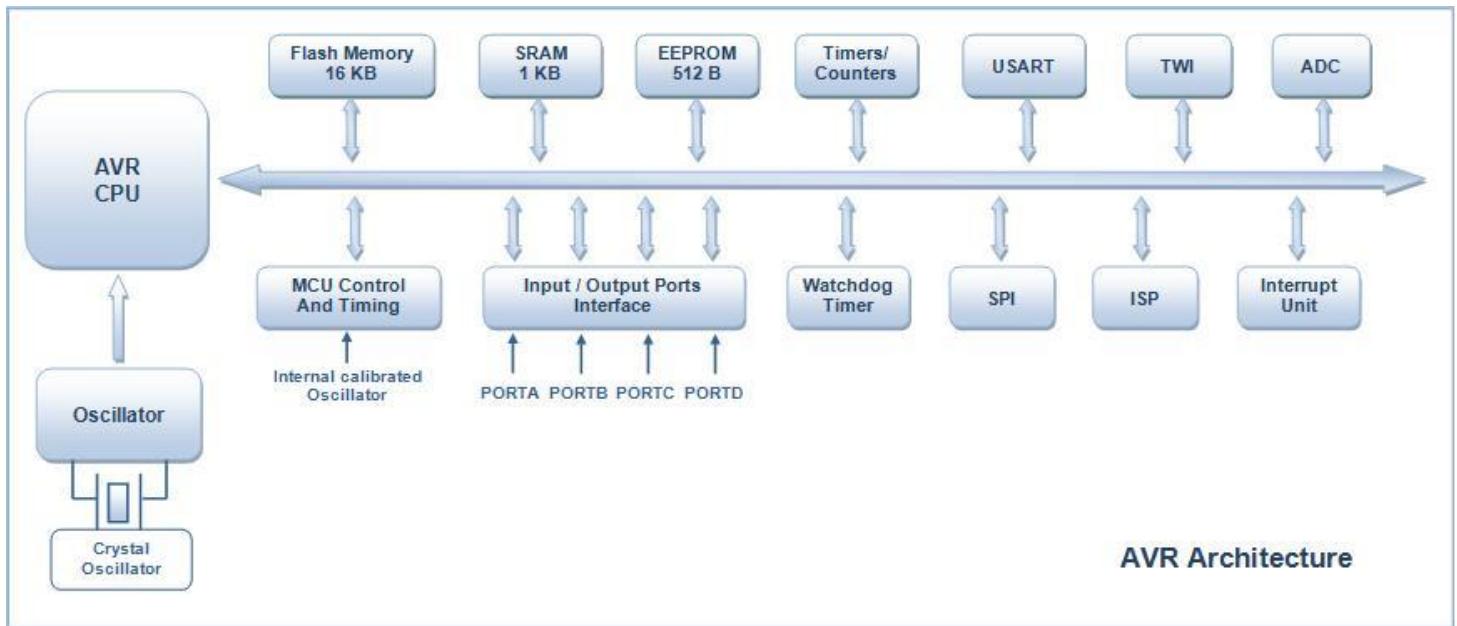


Figure 1-2: Simplified Architecture of AVR Microcontroller



# Programming for Embedded Systems

## Unit 2

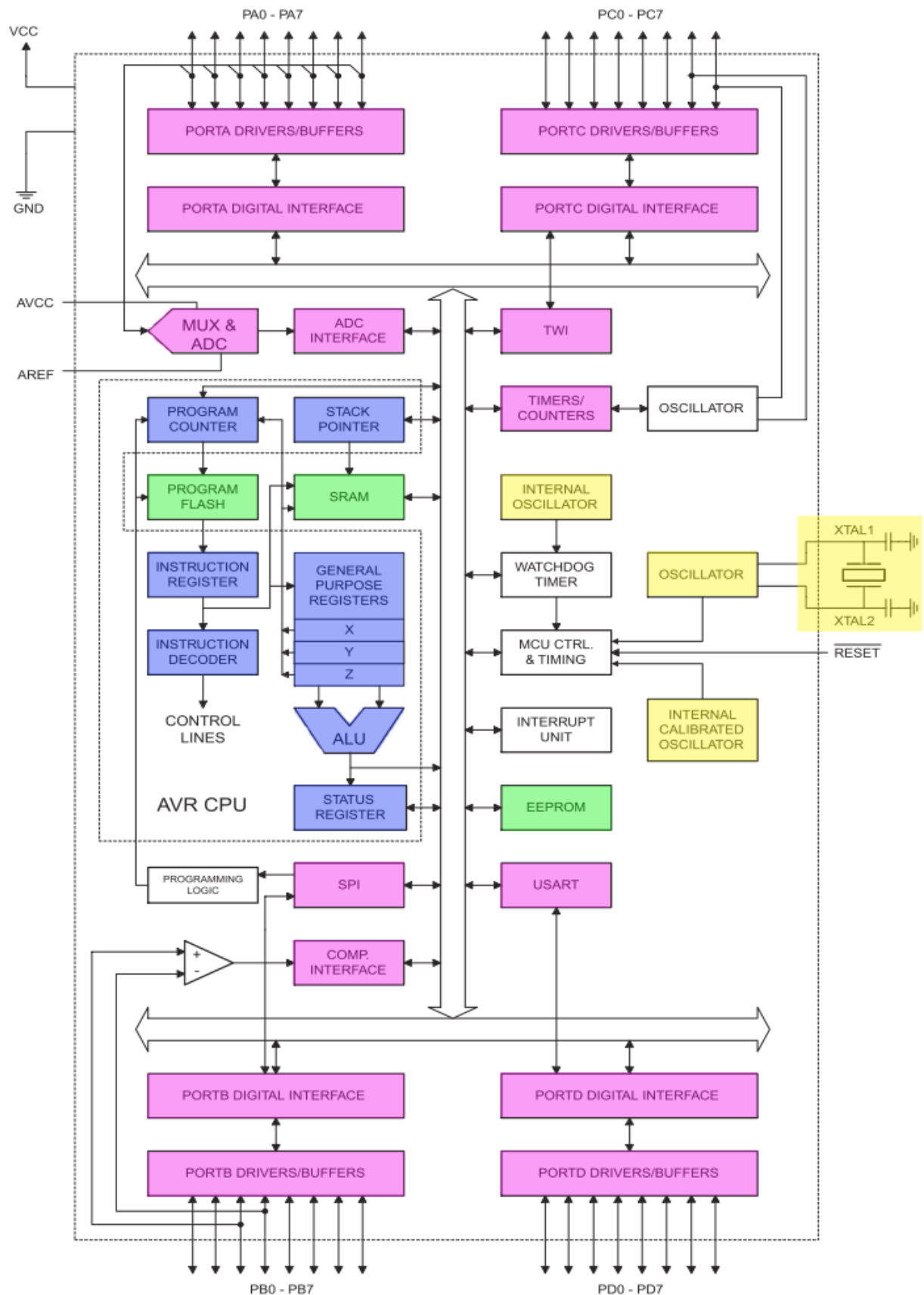


Figure 1-3: Detail Architecture of ATMEGA 32 Architecture

### 1. Core Features of the AVR Architecture

#### 1. RISC-Based Design

- AVR microcontrollers are based on a 32 x 8-bit general-purpose register set.
- These registers allow fast access to data, with most instructions executed in a single clock cycle.
- The design enables high throughput, allowing the microcontroller to perform up to **1 million instructions per second (MIPS)** at a clock speed of 1 MHz.

#### 2. Arithmetic Logic Unit (ALU)

- The ALU performs **arithmetic and logical operations** on data fetched from the registers.
- It connects directly to the general-purpose registers for faster data processing.
- Results of operations are stored back in the registers for efficient execution.

#### 3. Single-Cycle Execution

- Most instructions execute in a single clock cycle, contributing to high performance.
- Higher clock frequencies lead to faster operation but also increase power consumption, making frequency optimization important for energy-efficient designs.

### 2. Key Building Blocks of AVR Microcontroller Architecture

#### 1. I/O Ports

- The ATmega32 has **four 8-bit input/output ports**: PORTA, PORTB, PORTC, and PORTD.
- Each port can be configured as either input or output, providing flexibility in hardware interfacing.

#### 2. Internal Calibrated Oscillator

- The microcontroller features an internal oscillator to generate its clock signal.
- It operates at a default frequency of **1 MHz**, adjustable up to **8 MHz**.

#### 3. Analog-to-Digital Converter (ADC)

- An 8-channel, 10-bit ADC is integrated into the microcontroller.
- It converts analog signals into digital form, making it suitable for interfacing with sensors.

#### 4. Timers/Counters

- The ATmega32 includes:

- **Two 8-bit timers/counters.**
- **One 16-bit timer/counter.**
- These components generate precision delays, measure intervals, or produce PWM signals.

### 5. Watchdog Timer

- This safety feature resets the microcontroller if it hangs or gets stuck during execution.
- It operates independently using an internal oscillator.

### 6. Interrupt System

- The microcontroller supports **21 interrupts**, categorized as:
  - **16 internal interrupts** for peripherals like ADC, USART, timers, etc.
  - external interrupts **for responding to external events.**

## 2 USART (Universal Synchronous and Asynchronous Receiver Transmitter)

- Used for serial communication.
- Supports both synchronous and asynchronous modes for interfacing with devices like PCs or sensors.

## 3 General Purpose Registers

- The microcontroller has **32 general-purpose registers** connected directly to the ALU.
- These registers enhance processing speed by eliminating the need to fetch data from memory for every operation.

## 3. Memory Structure of AVR Microcontroller

### 1. Flash EEPROM

- **Size:** 16 KB.
- Non-volatile memory used to store the program code.
- Supports in-system programming (ISP), allowing updates without removing the microcontroller from the circuit.

### 2. Byte-Addressable EEPROM

- **Size:** 512 bytes.

- Non-volatile memory ideal for storing small amounts of data, such as lock codes or settings.

### 3. SRAM (Static Random-Access Memory)

- **Size:** 1 KB.
- Volatile memory used for temporary data storage during program execution.
- A portion is reserved for registers and peripheral subsystems.

## 4. Communication Interfaces

### 1. SPI (Serial Peripheral Interface)

- Enables high-speed serial communication with devices sharing a common clock.
- Suitable for interfacing with sensors, memory devices, or other microcontrollers.

### 2. TWI (Two-Wire Interface)

- Also known as I2C, it connects multiple devices over two lines: data (SDA) and clock (SCL).
- Each device has a unique address, supporting multi-device communication.

## 5. Additional Features

### 1. In-System Programming (ISP)

- The ATmega32 supports ISP via SPI pins, enabling the microcontroller to be reprogrammed without removal from its application circuit.

### 2. Power Efficiency

- The microcontroller includes multiple power-saving modes to optimize energy consumption based on application needs.

### 3. Oscillator Options

- Besides the internal oscillator, external crystals or clock sources can be used for precise frequency control.

## 3.1 Embedded C Programming for Microcontrollers

// can be updated later on

### 4 Memory Management in Embedded Systems

Memory management is crucial in embedded systems due to their limited resources and the need for reliable operation. Embedded devices often work with constrained memory, and improper memory handling can lead to system crashes, unpredictable behavior, or inefficient performance. Effective memory management ensures optimal use of available memory, avoids wastage through techniques like reducing fragmentation and preventing memory leaks, and enables the system to execute tasks predictably and efficiently. This is especially important in real-time embedded systems where timing and stability are critical for proper functioning.

In Embedded C, memory management is achieved primary through three techniques: static memory allocation, dynamic memory allocation, and memory pool allocation. Each technique serves specific needs and comes with its advantages and limitations.

#### 1. Static Memory Allocation

Static memory allocation reserves memory at compile-time, meaning the memory size and location are fixed throughout the program's execution. This technique is predictable and eliminates runtime allocation overhead.

##### Features:

- Memory is allocated before the program starts.
- No runtime overhead for allocation or deallocation.
- Suitable for systems with fixed memory requirements.

##### Example:

```
int buffer [100]; // Memory allocated statically for 100 integers.
```

##### Advantages:

- Simple and efficient in resource-constrained environments.
- No fragmentation or memory leak risks.

##### Disadvantages:

- Limited flexibility; memory usage cannot adapt dynamically.
- Over-allocation can lead to wasted resources.

### 2. Dynamic Memory Allocation

Dynamic memory allocation occurs during runtime, using functions like `malloc()`, `calloc()`, `realloc()`, and `free()`. This technique is suitable for scenarios where memory requirements are variable and unpredictable.

#### Features:

- Allocates memory on demand at runtime.
- Offers flexibility in memory usage.

#### Example:

```
int *dynamicBuffer = (int *)malloc(100 * sizeof(int)); // Allocate
memory for 100 integers.
if (dynamicBuffer == NULL) {
    // Handle allocation failure.
}
free(dynamicBuffer); // Deallocate memory.
```

#### Advantages:

- Provides flexibility for dynamic applications.
- Efficient use of memory in variable workloads.

#### Disadvantages:

- Higher runtime overhead for allocation and deallocation.
- Risk of fragmentation and memory leaks.

### 3. Memory Pool Allocation

Memory pool allocation is a hybrid approach where a block of memory is pre-allocated and divided into fixed-size chunks. Tasks request chunks as needed, avoiding runtime allocation overhead and fragmentation.

#### Features:

- Memory is pre-allocated but used dynamically during program execution.

# Programming for Embedded Systems

## Unit 2

- Ensures consistent allocation and deallocation performance.

### Example:

```
#define POOL_SIZE 10
int memoryPool[POOL_SIZE];
int *getMemoryChunk() {
    static int index = 0;
    return (index < POOL_SIZE) ? &memoryPool[index++] : NULL;
}
```

### Advantages:

- Deterministic behavior, ideal for real-time systems.
- Avoids fragmentation and reduces allocation overhead.

### Disadvantages:

- Requires careful pre-allocation planning.
- Less flexible than fully dynamic allocation.

## 4.1 Comparison of Memory Allocation Techniques

Feature	Static Memory Allocation	Dynamic Memory Allocation	Memory Pool Allocation
Allocation Timing	Compile-time	Runtime	Pre-allocated during initialization
Flexibility	Low	High	Moderate
Performance Overhead	None	High (due to runtime calls)	Low
Fragmentation Risk	None	High (external fragmentation)	None (fixed-sized blocks)
Predictability	Very High	Low	High
Ease of Use	Simple to implement	Requires careful management	Moderate (requires planning)
Memory Safety Risks	None	High (memory leaks, dangling pointers)	None
Real-Time Suitability	Excellent	Poor	Excellent
Best Use Case	Fixed memory requirements	Variable memory needs	Real-time or deterministic systems

### Best Practices

1. **Prefer Static Allocation for Simplicity:** Use static allocation where memory requirements are fixed and predictable.
2. **Avoid Dynamic Allocation in Real-Time Systems:** Dynamic allocation should be minimized or avoided in time-critical applications due to unpredictability and overhead.

3. **Use Memory Pools for Efficiency:** For systems requiring predictable memory behavior, memory pools provide a balanced approach.
4. **Monitor Memory Usage:** Tools and practices for tracking memory leaks and fragmentation help maintain system reliability.
5. **Optimize Data Types:** Use the smallest data types that fulfill application requirements to conserve memory.

## 5 Input and Output Ports Interfacing on AVR

The number and designation of ports in the AVR microcontroller family depend on the total number of pins available on the specific chip variant. These ports provide the interface for input and output operations, but their configuration must be programmed appropriately for the desired functionality. Below is an overview of the port distribution across different AVR microcontroller packages:

Microcontroller Variant	Number of Ports	Port Labels
8-Pin AVR	1	PORTB
40-Pin AVR	4	PORTA, PORTB, PORTC, PORTD
64-Pin AVR	6	PORTA – PORTF
100-Pin AVR	12	PORTA – PORTL

Each I/O port in AVR microcontrollers is associated with three essential registers that enable configuration and operation. These registers control the behavior of the port pins for input, output, and data retrieval. The registers are:

- **Data Register (PORTx)**
- **Data Direction Register (DDRx)**
- **Input Pins Address Register (PINx)**

Here, x represents the port name (A, B, C, or D), depending on the port being addressed. Below is a detailed explanation of each register.

### 1. Data Direction Registers (DDRx)

- **Purpose:**  
Configures the pins of a port as either input or output.



# Programming for Embedded Systems

## Unit 2

- **Key Features:**
  - These are 8-bit registers.
  - Writing 1 to a bit in the register sets the corresponding pin as an **output** pin.
  - Writing 0 to a bit in the register sets the corresponding pin as an **input** pin.
  - These registers can be both read and written to.
  - By default, all bits are initialized to 0, meaning all pins are configured as input pins upon reset.
- **Examples:**
  1. **Configuring Port D as an Output Port:**
    - `DDRD = 0xFF; // Sets all 8 pins of Port D as output (0xFF = 0b11111111)`
  2. **Configuring Port D as an Input Port:**
    - `DDRD = 0x00; // Sets all 8 pins of Port D as input (0x00 = 0b00000000)`

## 2. Data Registers (PORTx)

- **Purpose:**

Controls the logic state (HIGH or LOW) of the port pins.
- **Key Features:**
  - These are 8-bit registers.
  - Writing 1 to a bit in the register sets the corresponding pin to a **logic HIGH** (5V).
  - Writing 0 to a bit in the register sets the corresponding pin to a **logic LOW** (0V).
  - These registers can be both read and written to.
  - The default value of all bits is 0 (LOW state).

### Example:

Writing a specific value to Port D:

- `PORTD = 0x55; // Sets alternating pins of Port D to HIGH (0x55 = 0b01010101)`

### 3. Input Pins Address Registers (PINx)

- **Purpose:**

Reads the logic levels present on the pins of the port.

- **Key Features:**

- These are 8-bit **read-only** registers.
- The value of each bit represents the current logic level of the corresponding pin.
- These registers cannot be written to.

- **Example:**

Reading the value of Port D:

```
uint8_t port_value;  
port_value = PIND; // Reads the current logic levels on  
Port D and stores them in `port_value`
```

#### Register Overview

Port A in AVR microcontrollers is managed through three 8-bit registers: **PORTA**, **DDRA**, and **PINA**. These registers work together to enable various input/output operations on Port A. These registers provide the necessary control and monitoring capabilities for efficient interaction with external devices. The figure below illustrates their structure and role in managing Port A.

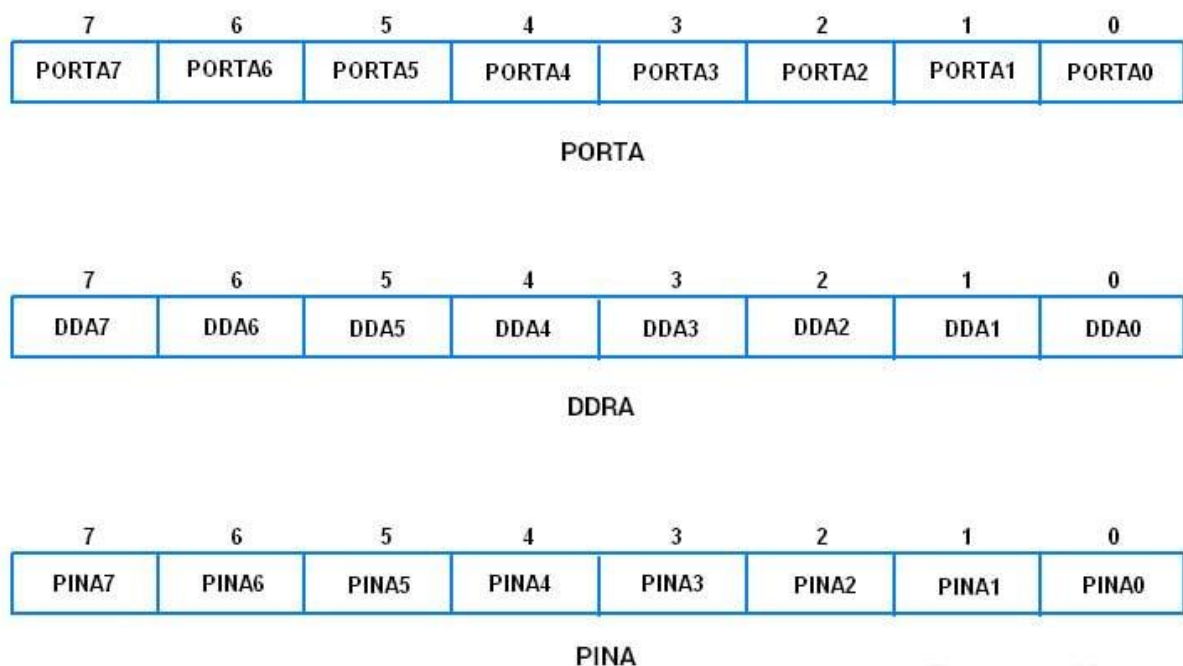
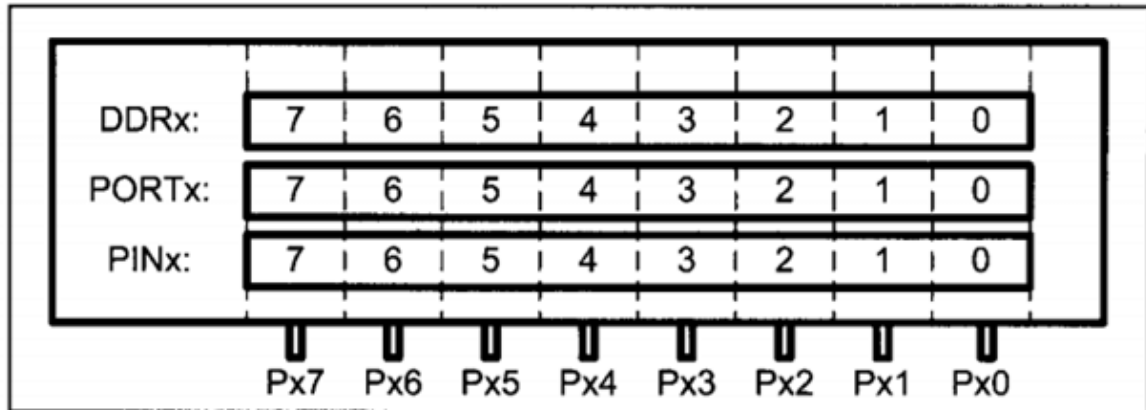


Figure 5-1: Register Associated With PORTA

# Programming for Embedded Systems

## Unit 2

This configuration process is similar for other ports (e.g., Port B, Port C, and Port D), where the appropriate registers (**PORTx**, **DDRx**, and **PINx**) are used to perform analogous operations.



### Practical Notes:

#### 1. Configuration Dependency:

- Ports must be properly configured using the **DDRx** register before performing input or output operations.
- Unconfigured pins may lead to undefined behavior.

#### 2. Versatility of PORTx Registers:

- Can be used to set or clear multiple pins simultaneously, making them highly efficient for embedded system tasks.

## 5.1 DDRx Register in Input and Output Operations

The **Data Direction Register (DDRx)** is a vital 8-bit register in AVR microcontrollers used to configure the functionality of the pins within a specific port (e.g., Port A, B, C, or D). This register determines whether each pin acts as an **input** or an **output**, leveraging the mechanism of a **tri-state buffer**. A tri-state buffer controls an I/O pin by allowing it to output either a **high or low voltage**, or to enter a **high-impedance** state where it effectively disconnects. In high-impedance mode, the pin does not affect or interfere with other devices on the bus.

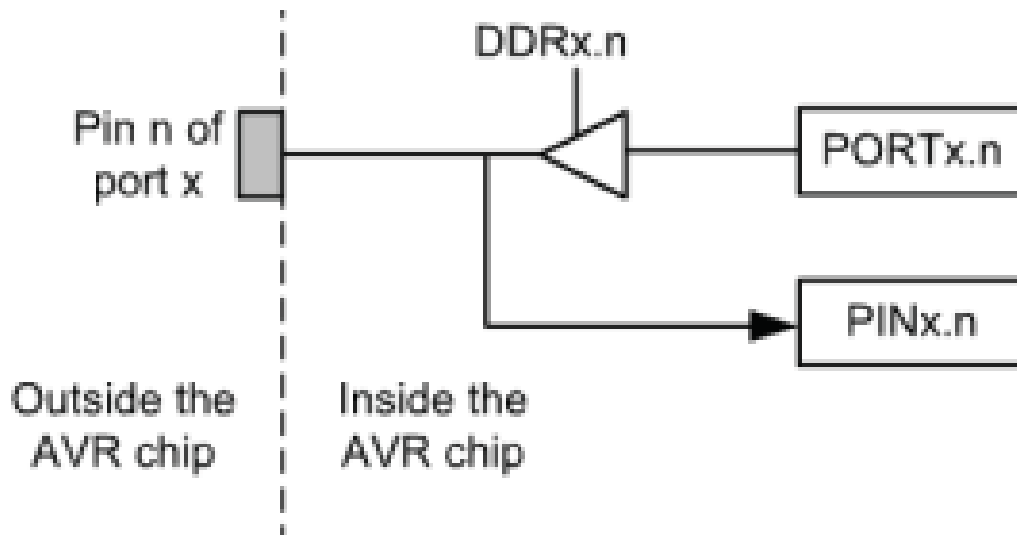


Figure 5-2: The I/O Port in AVR

### 5.1.1 Operation of DDR<sub>x</sub> in Output Mode (DDR<sub>x</sub> = 1)

When a bit in the **DDR<sub>x</sub>** register is set to 1:

- The **tri-state buffer** for the corresponding pin is **activated**, allowing the pin to drive a specific logic level.
- The logic level (HIGH or LOW) is determined by the corresponding bit in the **PORT<sub>x</sub>** register.
- The pin exits the high-impedance state and can deliver current to external devices.

```
DDRD = 0xFF; // Sets all 8 pins of Port D as output (0xFF = 0b11111111)
```

### 5.1.2 Operation of DDR<sub>x</sub> in Input Mode (DDR<sub>x</sub> = 0)

When a bit in the **DDR<sub>x</sub>** register is set to 0:

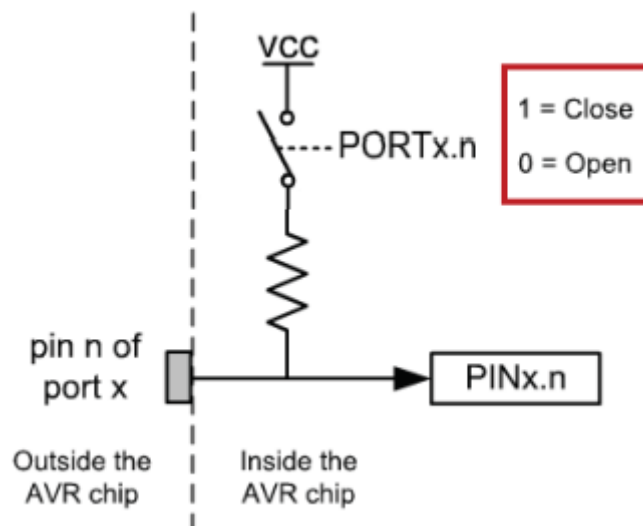
- The **tri-state buffer** for the corresponding pin is **deactivated**, placing the pin in a **high-impedance (Z)** state.
- In this state, the pin does not drive any signal, ensuring it does not interfere with external devices.
- The current logic level on the pin can be read via the **PIN<sub>x</sub>** register.

### 5.2 PIN register role in inputting data

To read the data from the pins, we use the **PINx** register. It's important to note that the **PINx** register is used to read the input data from the pins, while the **PORTx** register is used to send data out to the pins. Additionally, the **PORTx** register controls the output level (logic HIGH or LOW) for output pins, whereas the **PINx** register only allows reading the current state of input pins. This distinction ensures proper handling of data in both input and output modes.

### 5.3 PORT register role in inputting data

Each AVR pin has an internal pull-up resistor. When we set the corresponding bits in the **PORTx** register to 1, the pull-up resistors are enabled. This is particularly useful when no device is connected to the pin or when connected devices have high impedance, as the pull-up resistor ensures the pin is pulled to a logic HIGH state. Conversely, setting the bits in the **PORTx** register to 0 disables the pull-up resistors, allowing the pin to either float or be controlled by an external signal.



The pins of an AVR microcontroller can be in one of four possible states based on the values of the **PORTx** and **DDRx** registers. These states determine the behavior of the pin as either an input or output and whether it is actively driving a signal or in a high-impedance state. Below are the four states:

# Programming for Embedded Systems

## Unit 2

PORTx	DDRx	0	1
0		Input & high impedance	Out 0
1		Input & pull-up	Out 1

### 5.4 Dual Role of Port A and B

The AVR microcontroller optimizes the use of its I/O pins by multiplexing certain functions, which allows it to perform multiple tasks with the same physical pins. Port A, for example, is multiplexed with the analog-to-digital converter (ADC) to save I/O pins, enabling it to handle analog input signals. The alternate functions of Port A's pins, including their role in the ADC, are detailed in **Table 5.1**. This is an important feature because the ADC is widely used in embedded projects, and, as such, Port A is often dedicated to analog-to-digital conversion rather than simple digital I/O operations.

Similarly, Port B is also multiplexed with several functions to maximize pin usage. The alternate functions for the pins of Port B are outlined in **Table 5.2**, providing further flexibility for connecting various peripherals while minimizing the number of required I/O pins. This allows designers to optimize the layout and functionality of their projects without sacrificing performance.

**Table 5.1 Port A Alternative Function**

Port A Alternative Function	
Bits	Alternate Function (ADC)
PA0	ADC0
PA1	ADC1
PA2	ADC2
PA3	ADC3
PA4	ADC4
PA5	ADC5
PA6	ADC6
PA7	ADC7

**Table 5.2 Port B Alternative Function**

Port B Alternative Function	
Bits	Alternate Function
PB0	XCT/T0
PB1	T1
PB2	INT2/AIN0
PB3	OC0/AIN0
PB4	SS
PB5	MOSI
PB6	MISO
PB7	SCK

### 5.5 Dual Role of Port C and D

The alternate functions of Port C and Port D are shown below. To utilize these alternate functions, the pins must be configured properly. Depending on the specific use case—whether for digital I/O or specialized functions like communication protocols (e.g., UART, SPI), external interrupts, or PWM signals—each pin must be set accordingly to enable the desired functionality.

Port C Alternative Function	
Bits	Alternate Function
PC0	SCL
PC1	SDA
PC2	TCK
PC3	TMS
PC4	TDO
PC5	TDI
PC6	TOSCI
PC7	TOSC2

Port D Alternative Function	
Bits	Alternate Function
PDO	PSPO/CIIN+
PD1	PSPI/CIIN-
PD2	PSP2/C21N+
PD3	PSP3/C21N-
PD4	PSP4/ECCP1/PIA
PD5	PSP5/PIB
PD6	PSP6/PIC
PD7	PSP7/PID

### 5.6 Accessing Individual Bits in AVR I/O Ports

In many embedded applications, there is a need to manipulate only one or two bits of a port rather than the entire 8-bit register. AVR microcontrollers provide the powerful capability to access and modify individual bits of an I/O port without affecting the other bits. This feature ensures precise control over specific pins while preserving the state of the remaining pins in the port. For all AVR I/O ports, you can:

- Access and modify all 8 bits simultaneously (e.g., setting the entire port to a specific value).
- Access and modify individual bits without altering the rest of the bits in the port.

This flexibility is crucial in scenarios where multiple devices or functions share the same I/O port.

#### 1. Setting a Specific Bit to HIGH

# Programming for Embedded Systems

## Unit 2

Suppose we want to set bit 5 of an 8-bit register (e.g., PORTA) to HIGH without altering the other bits:

```
PORTA |= (1 << PA5); // Set bit 5 of PORTA to HIGH
```

Which is equivalent to

```
PORTA |= (1 << 5); // Set bit 5 of PORTA to HIGH
```

Which is equivalent to

```
PORTA = PORTA | (1 << 5); // Set bit 5 of PORTA to HIGH
```

Here,

- `1 << PA5` shifts a binary 1 to bit position 5.
- The `|=` operator ensures that only bit 5 is modified, while the rest remain unchanged.

### 2. Clearing a Specific Bit to LOW

To set bit 5 of PORTA to LOW without affecting the other bits:

```
PORTA &= ~(1 << PA5); // Set bit 5 of PORTA to LOW
```

Here,

- `~(1 << PA5)` creates a mask with bit 5 cleared (set to 0) and all other bits set to 1.
- The `&=` operator ensures that only bit 5 is cleared, leaving the other bits untouched.

### 3. Toggling a Specific Bit

To invert the current state of bit 5 in PORTA:

```
PORTA ^= (1 << PA5); // Toggle bit 5 of PORTA
```

Here,

The `^=` operator flips the state of bit 5, changing it from 1 to 0 or vice versa.

### 4. Checking the State of a Specific Bit

To read the state of bit 5 in PORTA:

```
if (PORTA & (1 << PA5)) {  
    // Bit 5 of PORTA is HIGH  
} else {  
    // Bit 5 of PORTA is LOW  
}
```

Here,



- `PORTA & (1 << PA5)` isolates bit 5.
- The condition evaluates to true if bit 5 is HIGH and false if it is LOW.

### 5. Writing to Multiple Bits Simultaneously

To set bits 4 and 5 of PORTA to HIGH without altering other bits:

```
PORTA |= (1 << PA4) | (1 << PA5); // Set bits 4 and 5 of PORTA to HIGH
```

Here,

- `(1 << PA4) | (1 << PA5)` creates a mask with bits 4 and 5 set to 1.
- The `|=` operator modifies only bits 4 and 5, preserving the state of the others.

### 6. Clearing Multiple Bits Simultaneously

To clear bits 4 and 5 of PORTA to LOW:

```
PORTA &= ~( (1 << PA4) | (1 << PA5) ); // Clear bits 4 and 5 of PORTA
```

Here,

- `~((1 << PA4) | (1 << PA5))` creates a mask with bits 4 and 5 cleared (set to 0).
- The `&=` operator clears these bits, leaving the rest unchanged.

The ability to access individual bits in an AVR microcontroller provides exceptional control and flexibility for embedded system design. By leveraging bitwise operations, developers can manipulate specific pins without affecting the state of others, ensuring efficient use of I/O resources. These techniques are essential for applications involving multiple peripherals or devices sharing the same port.

## ATmega32 GPIO Port Configuration: Step by Step

Here are the concise steps for programming an entire GPIO port as an output for the ATmega32:

#### 1. Set the Port Direction:

- Use the **Data Direction Register (DDRx)** to configure all pins of the port as output by writing `0xFF` to it.

#### 2. Initialize the Port State:

- Use the **PORTx** register to set the initial state of all pins (e.g., LOW or HIGH).

### 3. Write to the Port:

- Update the state of all pins by writing a value to the **PORTx** register (e.g., to set all pins HIGH, LOW, or in a specific pattern).

### 4. Compile and Upload:

- Compile the code and upload it to the ATmega32 using an appropriate programmer.

### 5. Test the Circuit:

- Verify the output by observing the connected devices (e.g., LEDs).

## 5.7 Programming Examples

### Example 1: Controlling an Entire GPIO Port on ATmega32

#### Code:

```
#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

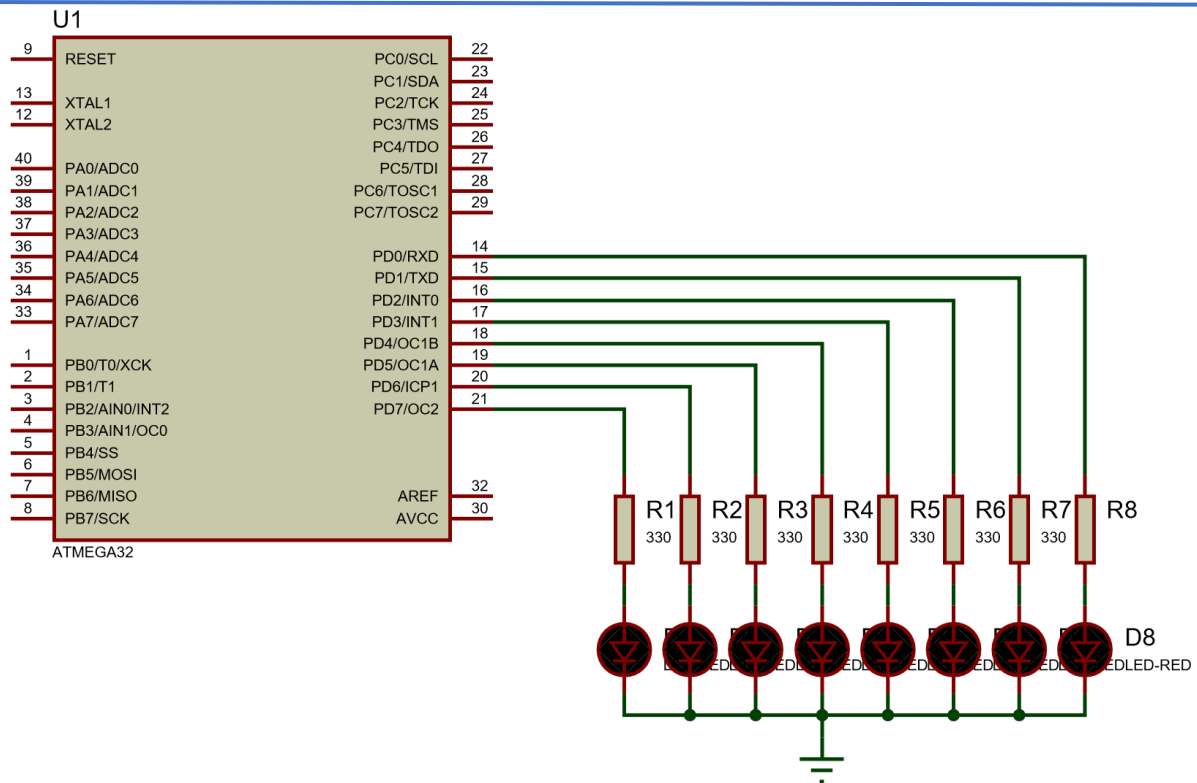
int main(void)
{
    DDRD = 0xFF;          /* Making all 8 pins of Port D as output pins
    */

    while(1)
    {
        PORTD = 0xFF;      /* Making PORTD high. This will make LED ON */
        _delay_ms(100);
        PORTD = 0x00;      /* Making PORTD low. This will make LED OFF */
        _delay_ms(100);
        /* Do not keep very small delay values. Very small delay will lead
        to LED appearing continuously ON due to persistence of vision */
    }
    return 0;
}
```

#### Circuit Diagram

# Programming for Embedded Systems

## Unit 2



### Example 2: Toggling an Entire GPIO Port on ATmega32

#### Code:

```
/*
 * GccApplication1.c
 *
 * Created: 12/21/2024 5:02:37 PM
 * Author : Deepesh
 */
// Toggle Alternate Lights

#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRD = 0xFF;          /* Making all 8 pins of Port D as output pins */

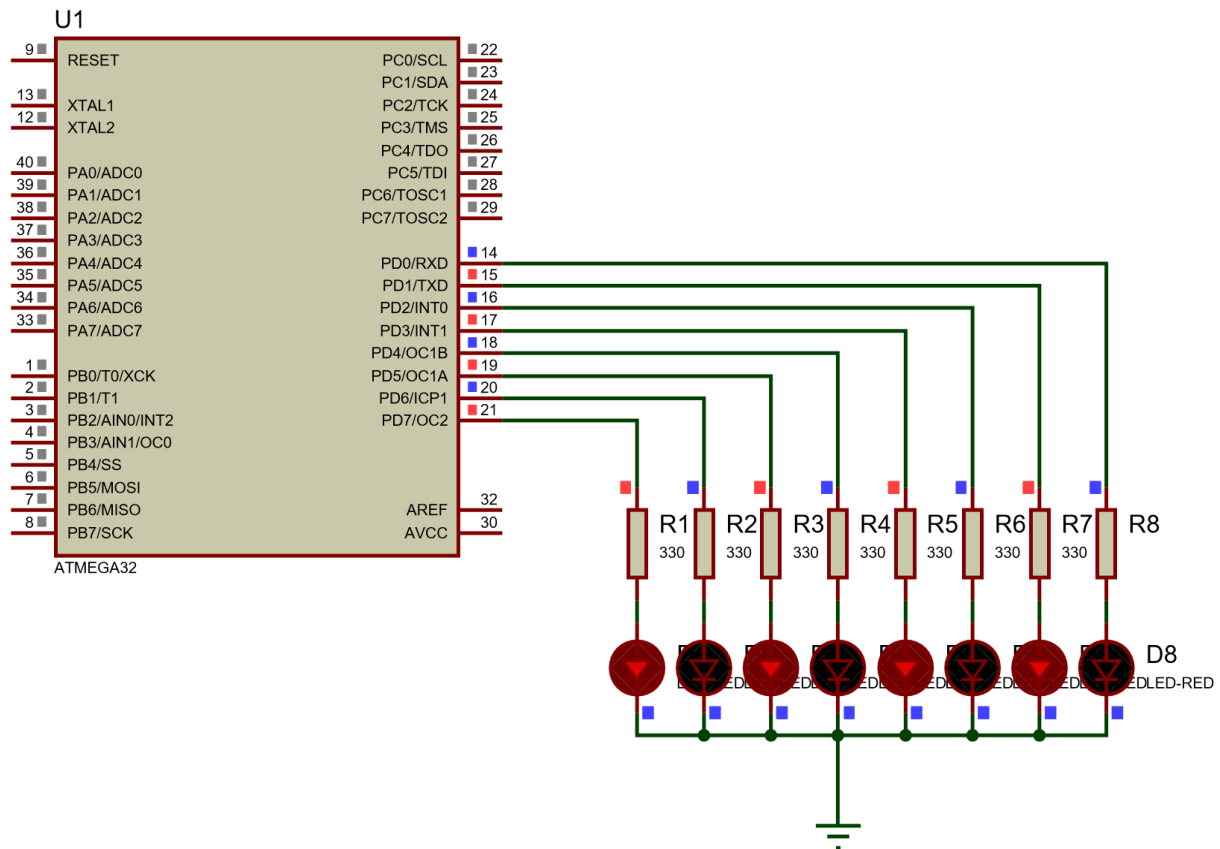
    while(1)
    {
        PORTD = 0x55;     /* Making PORTD Low and High alternate
01010101. */
        _delay_ms(100);
        PORTD = 0xAA;     /* Making PORTD High and Low in Alternate way */
        _delay_ms(100);
        /* Do not keep very small delay values. Very small delay will lead
to LED appearing continuously ON due to persistence of vision */
    }
    return 0;
}
```

# Programming for Embedded Systems

## Unit 2

}

### Circuit Diagram



### Example 3: Scrolling LED

#### Code

```
/*
 * GccApplication1.c
 *
 * Created: 12/21/2024 5:02:37 PM
 * Author : Deepesh
 */
// Scrolling Leds

#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRD = 0xFF; // Set all pins of PORTD as output
    PORTD = 0x01; // Initialize with the first LED ON (PD0)

    while (1) {
        for (uint8_t i = 0; i < 8; i++) {
            PORTD = (1 << i); // Shift the ON state to the next pin
            _delay_ms(200);    // Wait for 200 ms
        }
    }
}
```

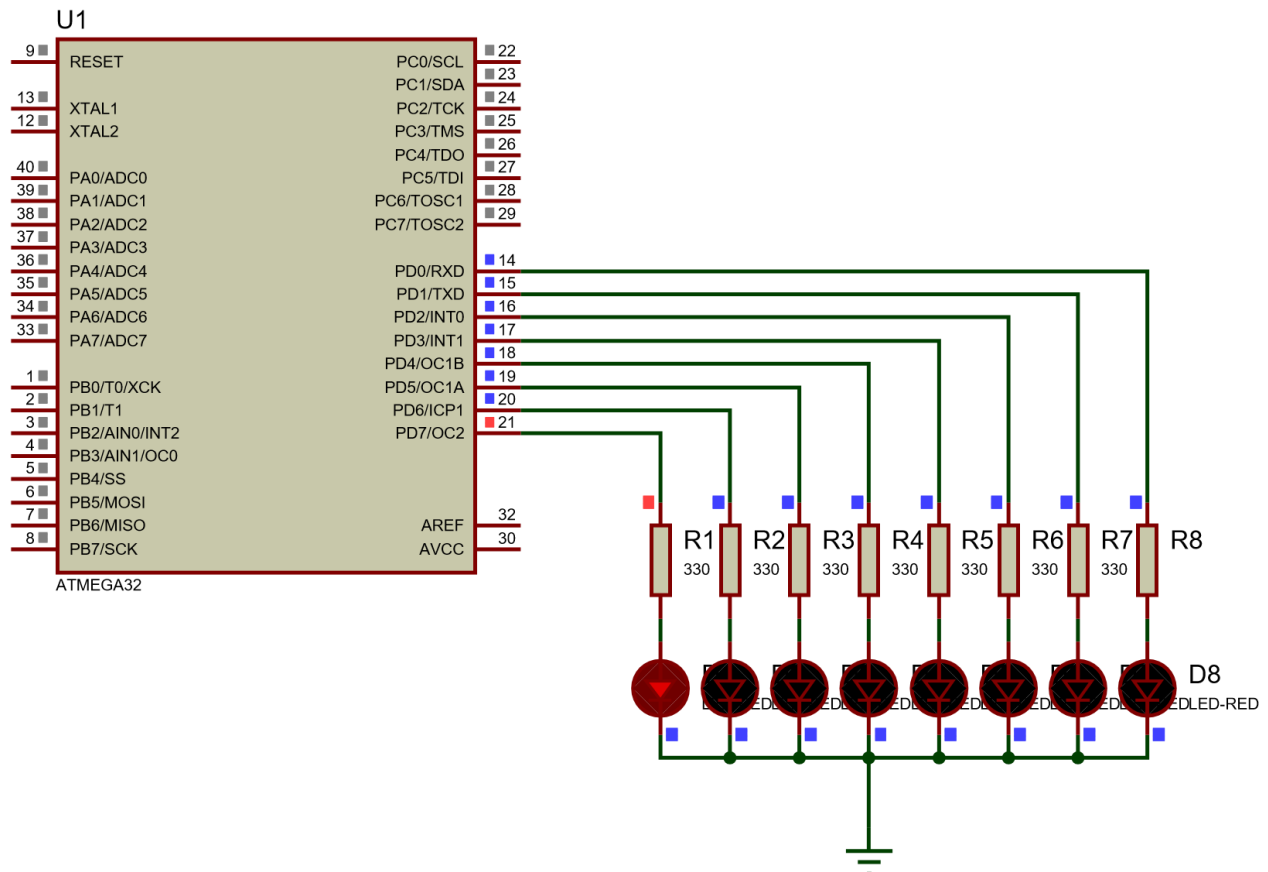
```

    }
}

return 0;
}

```

### Circuit Diagram



### Steps to Interface Individual Pins of Ports on ATmega32

- Set the Port Directions:**
  - Use the Data Direction Register (**DDRD**) to configure:
- Enable Internal Pull-Up for the Input Pin:**
  - Use the **PORTx** register to enable the internal pull-up resistor by writing 1 to it.
- Read the Input Pin Status:**
  - Use the **PIND** register to check the status of bit by masking it.
- Control the Output Pin:**
  - Based on the state of the input pin activate or deactivate led by writing to **PORTx** register.
- Implement an Infinite Loop:**
  - Continuously monitor the input pin and update the output pin accordingly in an infinite loop.
- Compile and Upload:**
  - Compile the code and upload it to the ATmega32 using an appropriate programmer.

### 7. Test the Circuit:

- Verify the behavior by connecting a switch to and an LED, observing how the LED responds to the switch's state.

#### Example 4: Switch-Controlled LED

```
/*
 * GccApplication1.c
 *
 * Created: 12/21/2024 5:02:37 PM
 * Author : Deepesh
 */
// Switching Led on/off based on switch

#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    // Set PD3 as output and PD2 as input
    DDRD = DDRD | (1 << 3); // Set PD3 as output (High for input)
    DDRD = DDRD & ~(1 << 2); // Set PD2 as input (Low for input)

    // Enable pull-up resistor on PD2
    PORTD = PORTD | (1 << 2); // Enable pull-up on PD2

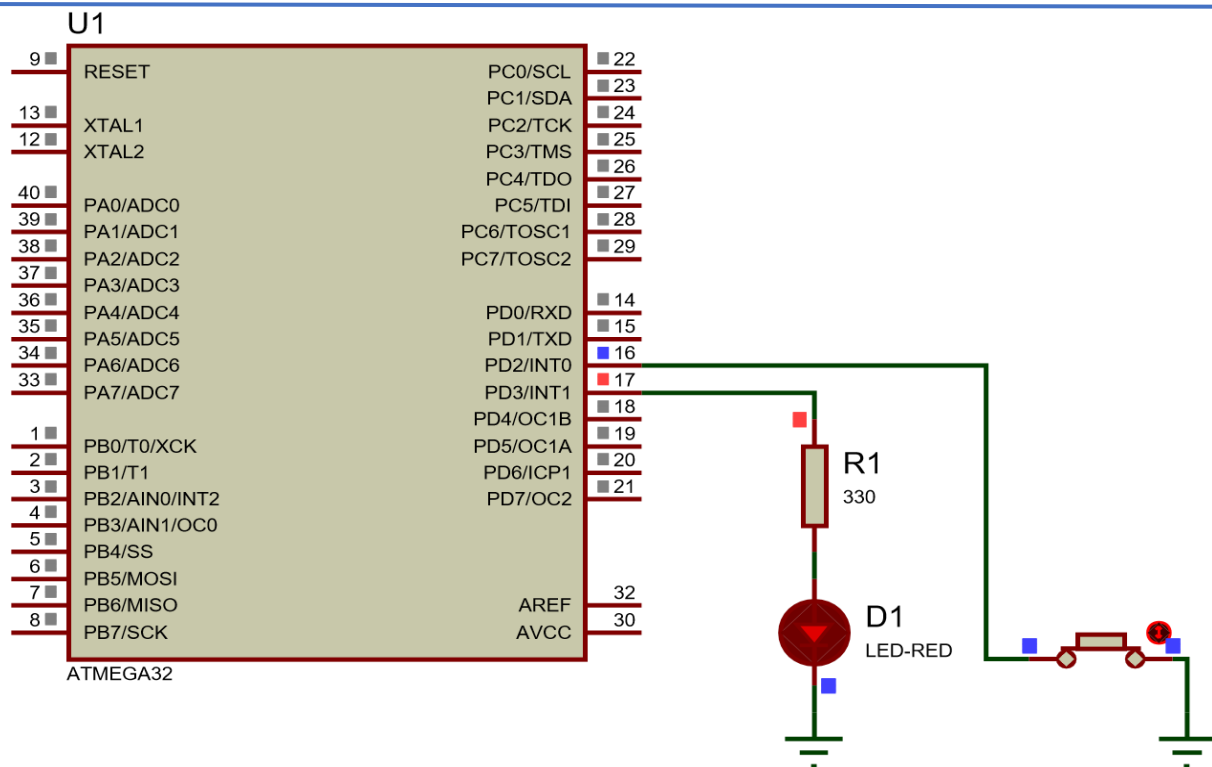
    while(1)
    {
        // Check if the switch (PD2) is pressed (low state)
        if (!(PIND & (1 << 2))) // Switch pressed (logic LOW)
        {
            PORTD = PORTD | (1 << 3); // Turn ON the LED (PD3)
        }
        else // Switch not pressed (logic HIGH)
        {
            PORTD = PORTD & ~(1 << 3); // Turn OFF the LED (PD3)
        }

        _delay_ms(50); // De-bounce delay
    }

    return 0;
}
```

# Programming for Embedded Systems

## Unit 2



### Example 5: Switch-Controlled Motor Direction with L293D

```
/*
 * GccApplication1.c
 *
 * Created: 12/21/2024 5:02:37 PM
 * Author : Deepesh
 */
// Switching Led on/off based on switch

#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    // Set PD0 and PD1 as output (motor driver control pins)
    DDRD = (1 << PD0) | (1 << PD1);

    // Set PA0 and PA1 as input (switch pins)
    DDRA = 0x00;

    // Enable pull-up resistors on PA0 and PA1
    PORTA = (1 << PA0) | (1 << PA1);

    while(1)
    {
        // Check if switch on PA0 is pressed (LOW) and PA1 is not pressed
        if (!(PINA & (1 << PA0)) && (PINA & (1 << PA1)))
        {
            // Switch on PA0 is pressed and PA1 is not pressed
        }
    }
}
```

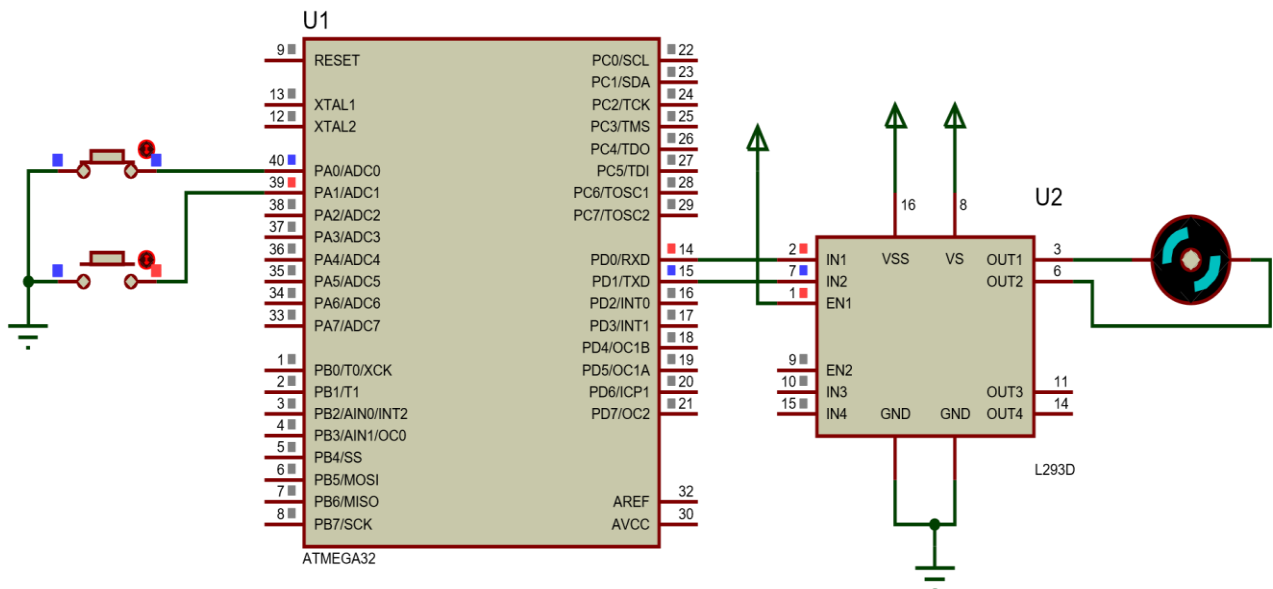
# Programming for Embedded Systems

## Unit 2

```
// Set motor to one direction (e.g., forward)
PORTD = (1 << PD0); // Motor direction forward
PORTD &= ~(1 << PD1); // Ensure PD1 is LOW
}
// Check if switch on PA1 is pressed (LOW) and PA0 is not pressed
else if (!(PINA & (1 << PA1)) && (PINA & (1 << PA0)))
{
    // Set motor to the opposite direction (e.g., backward)
    PORTD = (1 << PD1); // Motor direction backward
    PORTD &= ~(1 << PD0); // Ensure PD0 is LOW
}
else
{
    // Stop motor if both switches are released
    PORTD &= ~((1 << PD0) | (1 << PD1)); // Stop the motor
}

_delay_ms(100); // Simple debounce delay
}

return 0;
}
```



## 6 Timers and Counters in AVR

Timers and counters are essential components in microcontrollers, enabling event counting and time delay generation. These features are implemented using **counter registers**, as shown in Figure 9-1.



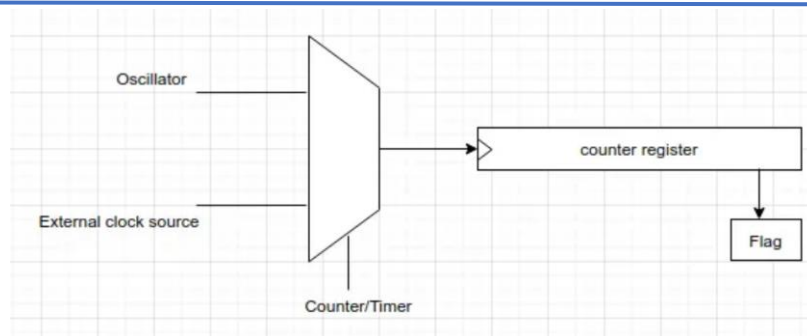


Figure 9-1. A General View of Counters and Timers/Counter in AVR Microcontrollers

If we want to set it as timer we must select Timer i.e **selection pin as 0** and if we want to set it as counter we must select **selection pin as 1**.

### Counting Events

To count events, an external signal is connected to the clock pin of the counter register. Each time the external event occurs, the counter increments. The value stored in the counter register then reflects the total number of events. This method is useful for tracking external occurrences such as sensor triggers or signal pulses.

### Generating Time Delays

For time delays, the counter's clock pin is connected to an oscillator. Each oscillator tick increments the counter, and the elapsed time is calculated based on the oscillator frequency.

- For example, with a **1 MHz oscillator**, the counter increments once per microsecond. To create a delay of **100 microseconds**, clear the counter at the start and wait until its value reaches 100.

### Using Overflow Flags for Timing

Microcontrollers also provide an **overflow flag** for each counter.

- When the counter exceeds its maximum value (e.g., 255 for an 8-bit counter), it wraps back to zero, and the overflow flag is set.
- The flag must be cleared in software.

### Example with an 8-bit Timer:

- Suppose the timer operates at **1 MHz** (1 tick per microsecond).
- To generate a **3-microsecond delay**, load the counter with the value  $\$FD$  (253).
  - Tick 1: Counter increments to  $\$FE$  (254).
  - Tick 2: Counter increments to  $\$FF$  (255).
  - Tick 3: Counter overflows to  $\$00$ , and the flag is set.

### 6.1 Timers in ATmega32

ATmega32 provides three timers/counters for versatile applications:

- **Timer0**: 8-bit timer
- **Timer1**: 16-bit timer
- **Timer2**: 8-bit timer

#### 1. 8-bit Timers (Timer0 and Timer2):

Ideal for smaller delays or counting fewer events due to their limited range (0–255).

#### 2. 16-bit Timer (Timer1):

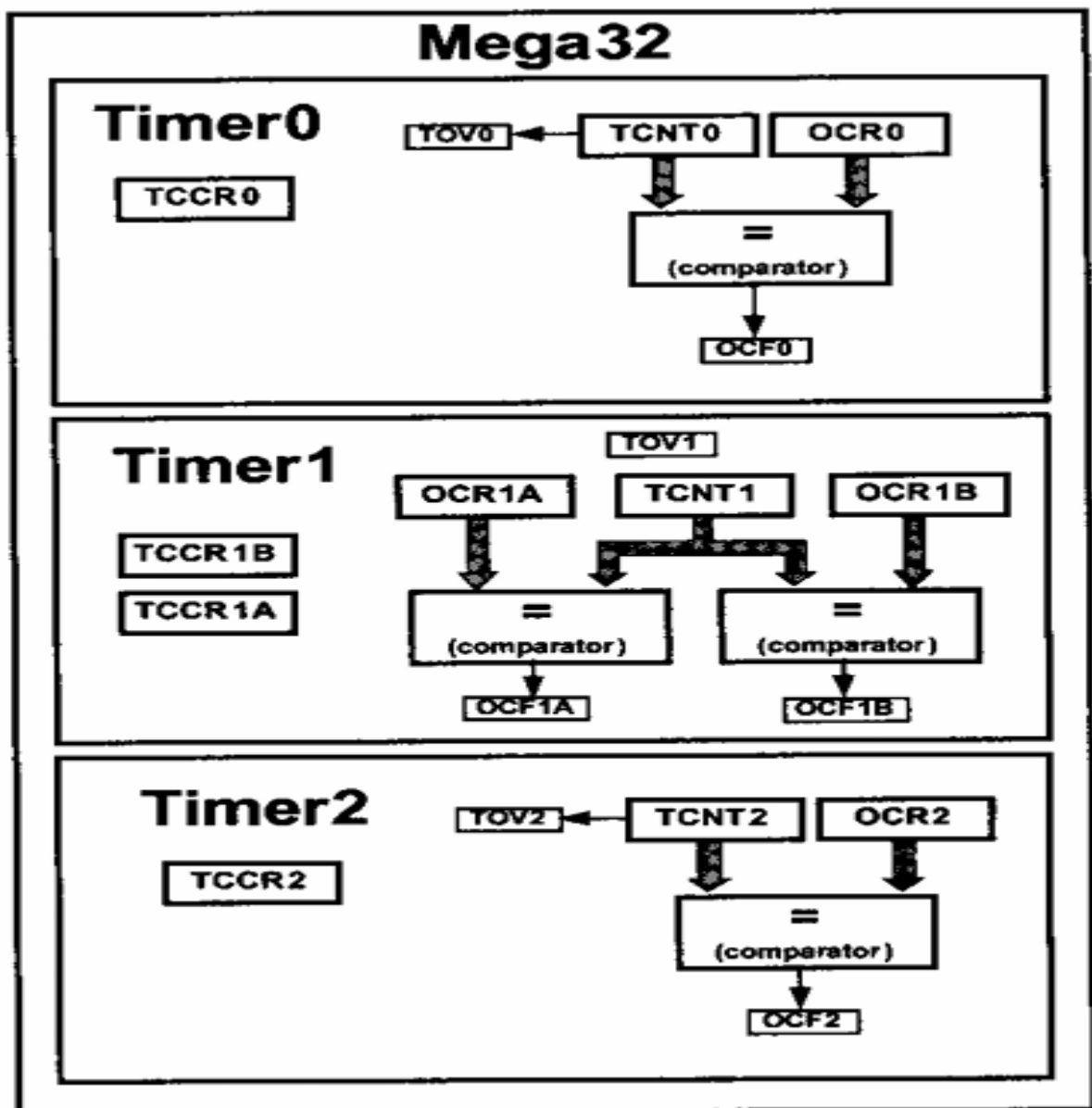
Suitable for longer delays or higher-precision timing as it can count up to 65535.

#### 6.1.1 Registers Associated with Timers in ATmega32

Timers in ATmega32 are controlled by several key registers:

- **TCNTn (Timer/Counter Register)**: Holds the current timer/counter value.
- **TOVn (Timer Overflow Flag)**: Indicates when a timer overflow occurs.
- **TCCRn (Timer/Counter Control Register)**: Configures the timer/counter's operation mode and clock source.
- **OCRn (Output Compare Register)**: Compares its value with  $TCNTn$  to trigger specific actions.

These registers allow precise control of timers for tasks like generating delays, counting events, and producing PWM signals.



### 1. TCNT<sub>n</sub> (Timer/Counter Register)

- **Function:**

- Holds the current value of the timer or counter.
- Automatically increments with each clock pulse.
- Can be read or written to set or monitor the counter's value.

### 2. TOV<sub>n</sub> (Timer Overflow Flag)

- **Function:**

- Set when the timer overflows (e.g., from 255 to 0 for 8-bit timers).
- Must be cleared manually in software.

### 3. TCCRn (Timer/Counter Control Register)

- **Function:**
  - Configures the timer's mode of operation, prescaler, and clock source.
  - Key settings include Normal Mode, CTC Mode, and PWM modes.

### 4. OCRn (Output Compare Register)

- **Function:**
  - Compared with the current value in TCNTn.
  - When the values match, the **Output Compare Flag (OCFn)** is set, which can be used to trigger interrupts or toggle pins.

#### 6.1.2 Programming Timer 0

Before programming Timer 0 we need to understand major registers associated with this timer. In **Normal Mode**, Timer 0 increments from 0x00 to 0xFF (8-bit), then overflows and starts again from 0x00. This overflow can trigger an **Overflow Interrupt** or set the **TOV0 (Timer Overflow Flag)**, which allows us to measure time intervals or count events.

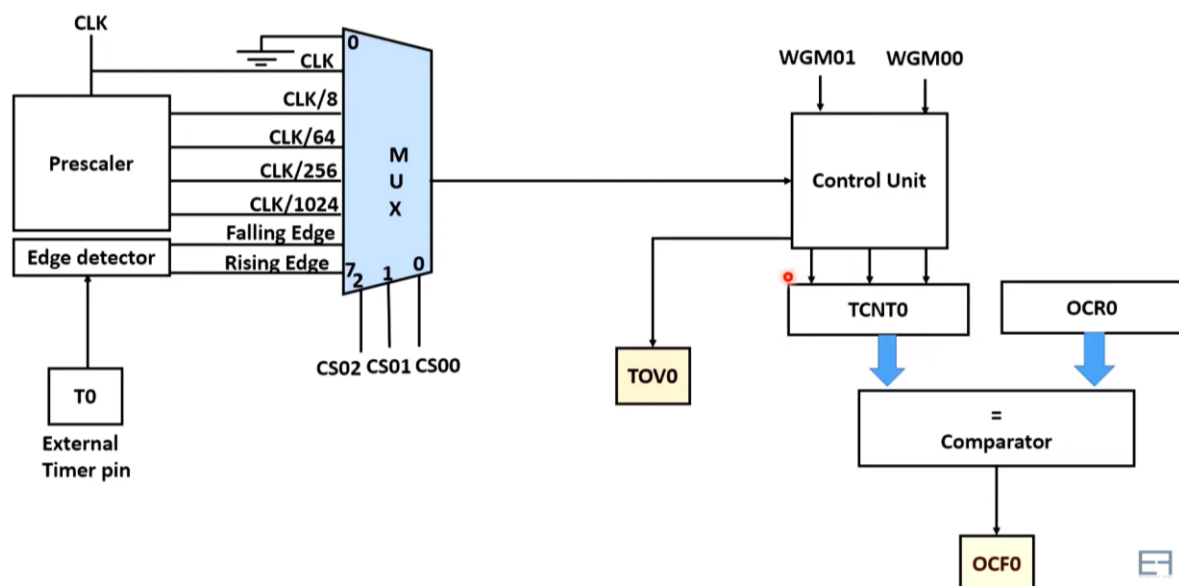


Figure 6-1: Timer 0 overview in Detail

#### 6.1.3 Key Registers of Timer 0

##### 1. TCNT0 (Timer/Counter Register 0)

# Programming for Embedded Systems

## Unit 2

- **Type:** 8-bit register
- **Purpose:** Holds the current count value. It increments with each clock pulse.
- **Initial Value:** 0x00 after reset.

TCNTO	D7	D6	D5	D4	D3	D2	D1	D0
-------	----	----	----	----	----	----	----	----

### 2. TCCR0 (Timer/Counter Control Register 0)

- **Type:** 8-bit register
- **Purpose:** Configures the mode of operation and clock source selection for Timer 0.

7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

#### i Bit 7- FOC0: Force compare match

Write only a bit, which can be used while generating a wave. Writing 1 to this bit causes the wave generator to act as if a compare match has occurred.

#### ii Bit 6,3 - WGM00, WGM01: Waveform Generation Mode

WGM00	WGM01	Timer0 mode selection bit
0	0	Normal
0	1	CTC (Clear timer on Compare Match)
1	0	PWM, Phase correct
1	1	Fast PWM

#### iii Bit 5:4 - COM01:00: Compare Output Mode

These bits control the waveform generator. We will see this in the compare mode of the timer.

#### iv Bit 2:0 - CS02:CS00: Clock Source Select

These bits are used to select a clock source. When CS02: CS00 = 000, then timer is stopped. As it gets a value between 001 to 101, it gets a clock source and starts as the timer.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer / Counter stopped)
0	0	1	clk (no pre-scaling)

# Programming for Embedded Systems

## Unit 2

CS02	CS01	CS00	Description
0	1	0	clk / 8
0	1	1	clk / 64
1	0	0	clk / 256
1	0	1	clk / 1024
1	1	0	External clock source on T0 pin. Clock on falling edge
1	1	1	External clock source on T0 pin. Clock on rising edge.

### 3. TIFR: Timer Counter Interrupt Flag register

**Purpose:** Contains flags indicating the status of Timer/Counter events, such as overflow or compare match.

7	6	5	4	3	2	1	0
OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0

- **Bit 0 - TOV0:** Timer0 Overflow flag

0 = Timer0 did not overflow

1 = Timer0 has overflown (going from 0xFF to 0x00)

- **Bit 1 - OCF0:** Timer0 Output Compare flag

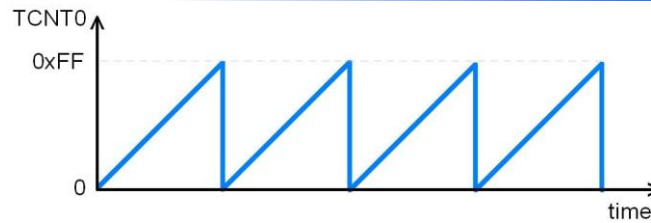
0 = Compare match did not occur

1 = Compare match occurred

- **Bit 2 - TOV1:** Timer1 Overflow flag
- **Bit 3 - OCF1B:** Timer1 Output Compare B match flag
- **Bit 4 - OCF1A:** Timer1 Output Compare A match flag
- **Bit 5 - ICF1:** Input Capture flag
- **Bit 6 - TOV2:** Timer2 Overflow flag
- **Bit 7 - OCF2:** Timer2 Output Compare match flag

### 4. Timer0 Overflow

- Normal mode: When the counter overflows i.e. goes from 0xFF to 0x00, the TOV0 flag is set.



### 6.1.4 Steps to Create a Delay Using Timer 0 in Normal Mode

Using Timer 0 in normal mode to generate a delay involves the following steps:

#### 1. Initialize the Timer

- Load the starting value into the **TCNT0** register (e.g., 0x25). This value determines how long it will take for the timer to overflow.

#### 2. Set Timer Configuration

- Configure the **TCCR0** register to enable Normal Mode and select a clock prescaler.
- The prescaler divides the system clock to adjust the timer speed. When configured, the timer begins counting.

#### 3. Monitor Timer Overflow

- Keep track of the **TOV0** (Timer 0 Overflow) flag in the **TIFR** register:
  - If using **polling**, continuously check for the flag to become 1.
  - Alternatively, use an **interrupt** to detect the overflow event automatically.

#### 4. Stop the Timer

- To stop the timer, set the clock source bits in **TCCR0** to 0. This disconnects the timer clock and halts counting.

#### 5. Clear the Overflow Flag

- To reset the **TOV0** flag, write 1 to the **TOV0** bit in the **TIFR** register. This prepares the timer for the next operation.

#### 6. Return to the Main Program

- Once the delay is complete, return to the main program and proceed with other tasks.

### 6.1.5 Calculating Values for Timer Delay

When programming a timer, determining the appropriate value to load into the TCNT0 register is crucial for achieving the desired delay. The following steps outline how to calculate this value:

#### 6.1.6 Steps to Determine Timer Initial Value (TCNT0)

##### 1. Calculate the Timer Clock Period ( $T_{\text{clock}}$ ):

- The timer's clock period is the reciprocal of the timer's clock frequency ( $F_{\text{Timer}}$ ):

$$T_{\text{clock}} = 1 / F_{\text{Timer}}$$

- In no prescaler mode, the timer frequency equals the oscillator frequency ( $F_{\text{oscillator}}$ ).

- Example: If  $F_{\text{oscillator}} = 1 \text{ MHz}$ , then  $T_{\text{clock}} = 1 \mu\text{s}$ .

##### 2. Determine the Required Timer Counts ( $n$ ):

- Divide the desired delay ( $T_{\text{delay}}$ ) by the timer clock period ( $T_{\text{clock}}$ ):

$$n = T_{\text{delay}} / T_{\text{clock}}$$

This gives the total number of timer increments required for the desired delay.

##### 3. Calculate the Initial Count Value ( $256 - n$ ):

- In an 8-bit timer, the total count range is 256 (from 0 to 255). To determine the starting point:

$$\text{Initial Count} = 256 - n$$

This ensures the timer overflows exactly after  $n$  counts.

##### 4. Convert to Hexadecimal:

- Convert the initial count value from decimal to hexadecimal format, as required by the TCNT0 register.

- Example: If the initial count is 50 (decimal), its hexadecimal equivalent is 0x32.



### 5. Load the Calculated Value into TCNT0:

- Set the TCNT0 register to the computed value:

TCNT0 = Initial Count in Hexadecimal

### 6.1.7 Example Calculation Desired Delay: $T_{\text{delay}} = 2 \text{ ms}$ , Oscillator Frequency: $F_{\text{oscillator}} = 1 \text{ MHz}$ and Prescaler: None (Clock directly used by the timer).

#### Step 1: Calculate $T_{\text{clock}}$ :

$$T_{\text{clock}} = 1 / F_{\text{oscillator}} = 1 / 1 \text{ MHz} = 1 \mu\text{s}$$

#### Step 2: Determine n:

$$n = T_{\text{delay}} / T_{\text{clock}} = 2 \text{ ms} / 1 \mu\text{s} = 2000 \text{ ticks}$$

#### Step 3: Calculate Initial Count:

Since 2000 ticks exceed 256 (8-bit timer limit), a pre-scaler is needed. Assume a pre-scaler of 8:

$$T_{\text{clock\_prescaled}} = 8 \times T_{\text{clock}} = 8 \mu\text{s}$$

$$n = T_{\text{delay}} / T_{\text{clock\_prescaled}} = 2 \text{ ms} / 8 \mu\text{s} = 250 \text{ ticks}$$

$$\text{Initial Count} = 256 - n = 256 - 250 = 6$$

#### Step 4: Convert to Hexadecimal:

Initial Count in Hex = 0x06

#### Step 5: Load into TCNT0:

Set the timer with: **TCNT0 = 0x06;**

## 6.2 Programming Examples

### A. Timer as Delay

### 1. Program to create a delay using Timer 0, Normal Mode and No Prescaler

#### Code:

```
/*
 * GccApplication2.c
 *
 * Created: 12/23/2024 6:40:52 PM
 * Author : Deepesh
 */
#define F_CPU 8000000UL
#include <avr/io.h>

void T0delay();

int main(void)
{
    DDRD = 0xFF;          /* PORTD as output*/

    while(1)              /* Repeat forever*/
    {
        PORTD=0x55;
        T0delay();        /* Give some delay */
        PORTD=0xAA;
        T0delay();
    }
}

void T0delay()
{
    TCNT0 = 0x25;          /* Load TCNT0*/
    TCCR0 = 0x01;          /* Timer0, normal mode, no pre-scalar */

    while((TIFR&0x01)==0); /* Wait for TOV0 to roll over */
    TCCR0 = 0;
    TIFR = 0x1;            /* Clear TOV0 flag*/
}
```

The time delay generated by above code

As  $F_{osc} = 8 \text{ MHz}$

$T = 1 / F_{osc} = 0.125 \mu\text{s}$

Therefore, the count increments by every **0.125  $\mu\text{s}$** .

In above code, the number of cycles required to roll over are:

$0xFF - 0x25 = 0xDA$  i.e. decimal 218

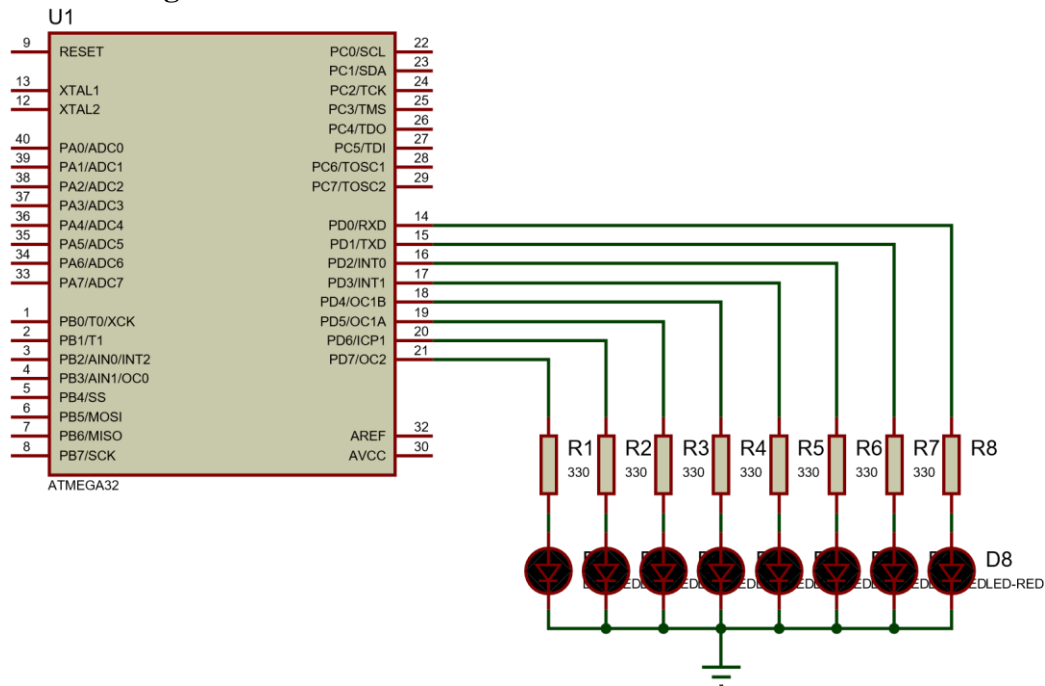
Add one more cycle as it takes to roll over and raise TOV0 flag: 219

**Total Delay** =  $219 \times 0.125 \mu\text{s} = 27.375 \mu\text{s}$

# Programming for Embedded Systems

## Unit 2

### Circuit Diagram :



### 2. Program to generate a square waveform having 10 ms high and 10 ms low time Using Timer 0 Normal Mode and Pre-Scalar:

#### Code:

```
/*
 * GccApplication2.c
 *
 * Created: 12/23/2024 6:40:52 PM
 * Author : Deepesh
 */
#define F_CPU 8000000UL
#include <avr/io.h>

void T0delay();

int main(void)
{
    DDRB = 0xFF;          /* PORTB as output */
    PORTB=0;
    while(1)              /* Repeat forever */
    {
        PORTB= ~ PORTB;
        T0delay();
    }
}

void T0delay()
{
    TCCR0 = (1<<CS02) | (1<<CS00); /* Timer0, normal mode, /1024 prescaler */
    TCNT0 = 0xB2;                  /* Load TCNT0, count for 10ms */
    while((TIFR&0x01)==0); /* Wait for TOV0 to roll over */
    TCCR0 = 0;
    TIFR = 0x1;                  /* Clear TOV0 flag */
}
```

# Programming for Embedded Systems

## Unit 2

Let us generate a square waveform having 10 ms high and 10 ms low time:

First, we have to create a delay of 10 ms using timer0.

$$*F_{\text{osc}} = 8 \text{ MHz}$$

Use the pre-scalar 1024, so the timer clock source frequency will be,

$$8 \text{ MHz} / 1024 = \mathbf{7812.5 \text{ Hz}}$$

$$\text{Time of 1 cycle} = 1 / 7812.5 = \mathbf{128 \mu\text{s}}$$

Therefore, for a delay of 10 ms, number of cycles required will be,

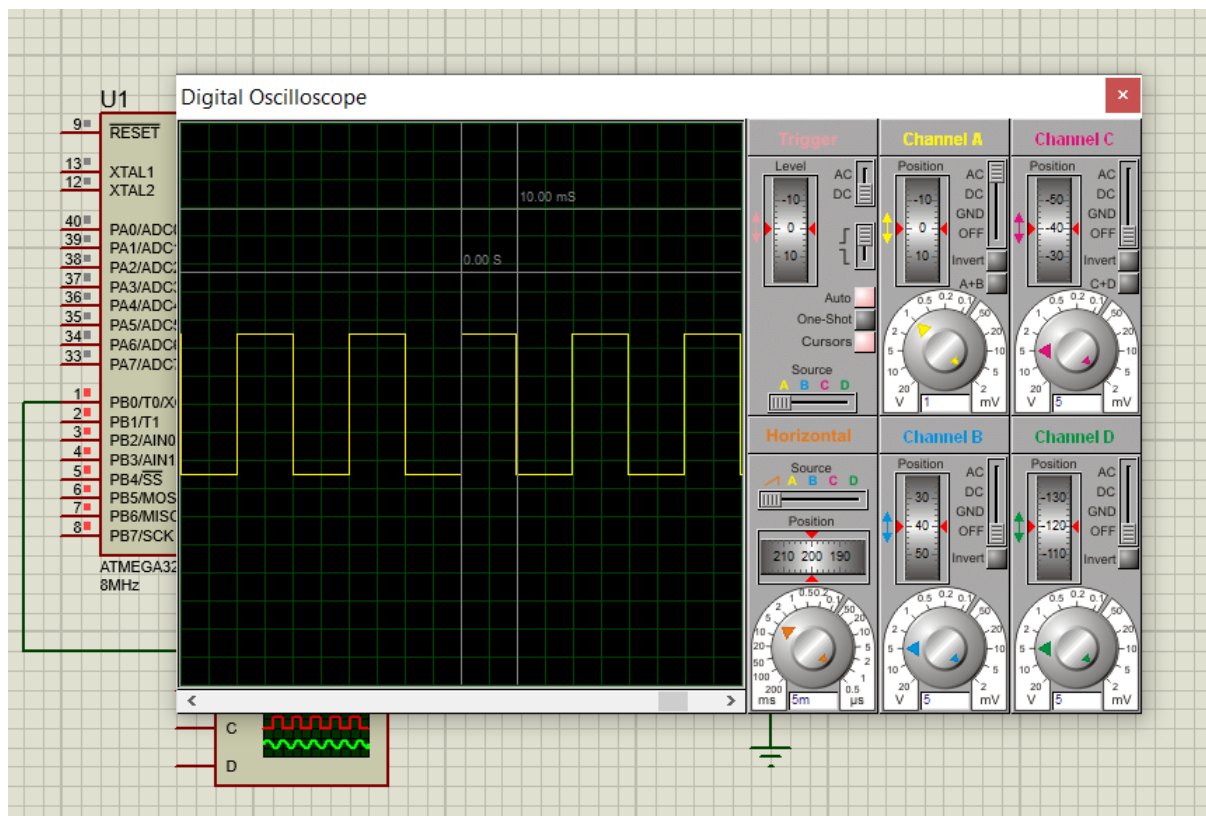
$$10 \text{ ms} / 128 \mu\text{s} = \mathbf{78 \text{ (approx)}}$$

We need 78 timer cycles to generate a delay of 10 ms. Put the value in TCNT0 accordingly.

Value to load in TCNT0 = 256 – 78 (78 clock ticks to overflow the timer)

$$= \mathbf{178 \text{ i.e. } 0xB2 \text{ in hex}}$$

Thus, if we load 0xB2 in the TCNT0 register, the timer will overflow after 78 cycles i.e. precisely after a delay of 10 ms.



### B. Timer as Counter Example

3. Assuming that a 1 Hz clock pulse is fed into pin PB0, use the TOV0 flag to extend Timer0 to a 16-bit counter and display the counter on PORTC and PORTD.

#### CODE

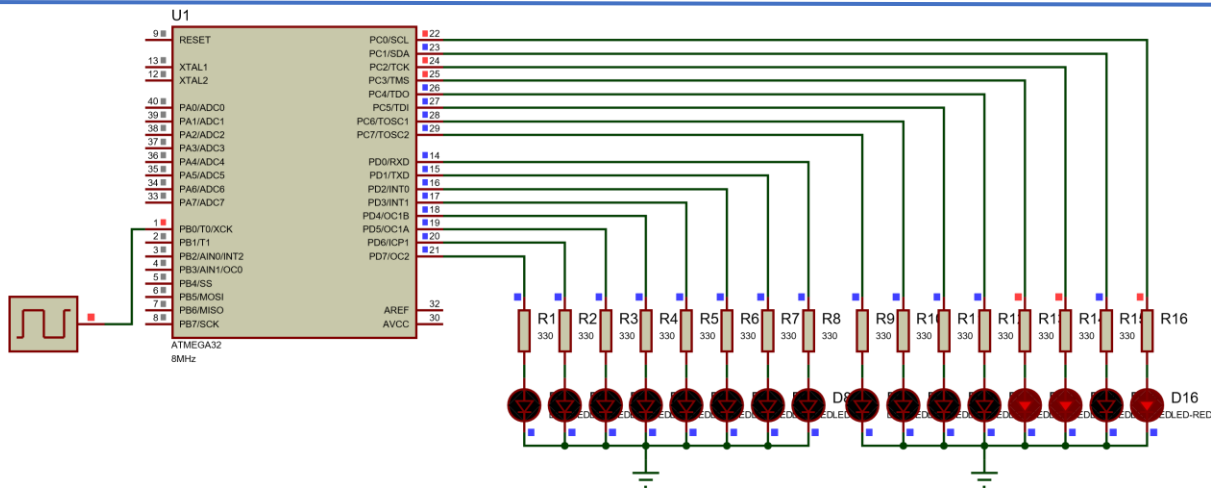
```
/*
 * GccApplication2.c
 *
 * Created: 12/23/2024 6:40:52 PM
 * Author : Deepesh
 */
#define F_CPU 8000000UL
#include <avr/io.h>

int main(void)
{
    PORTB=0x01;
    DDRC= 0xFF;
    DDRD = 0xFF;
    TCCR0=0x06;
    TCNT0=0x00;
    while(1)                /* Repeat forever */
    {
        do{
            PORTC = TCNT0;

        }while((TIFR &(1<<TOV0))==0);
        TIFR=1<<TOV0;
        PORTD++;
    }
}

void T0delay()
{
    TCCR0 = (1<<CS02) | (1<<CS00); /* Timer0, normal mode, /1024 prescaler */
    TCNT0 = 0xB2;                  /* Load TCNT0, count for 10ms */
    while((TIFR&0x01)==0); /* Wait for TOV0 to roll over */
    TCCR0 = 0;
    TIFR = 0x1;                   /* Clear TOV0 flag */
}
```

#### Circuit diagram



## 7 AVR Interrupt

In embedded systems, efficiency and responsiveness are paramount, especially when managing multiple tasks or devices. A microcontroller like the AVR ATmega32 often juggles responsibilities such as monitoring sensors, controlling actuators, or maintaining communication protocols. If we rely on the **polling method**—where the microcontroller continuously checks each device's status—it wastes valuable time monitoring devices that may not even need attention. Furthermore, polling lacks the ability to prioritize critical tasks, leading to delays in responding to urgent events. For example, imagine a system where a timer overflow occurs while the microcontroller is busy polling a sensor with no immediate need for action. The delay in addressing the timer event could lead to system inefficiencies or even failure.

This inherent limitation of polling calls for a better approach: **interrupts**. Interrupts allow the microcontroller to focus on primary tasks, stepping in to address external requests only when necessary. To truly appreciate the advantages of interrupts, let us explore the fundamental differences between **interrupts** and **polling** in the next section.

### 7.1 Interrupts vs. Polling

When managing multiple devices, a microcontroller like the AVR ATmega32 can use either **interrupts** or **polling** to provide service. In the **polling method**, the microcontroller continuously checks the status of each device in a sequential manner. When a device requires attention, the microcontroller performs the service before moving to the next

# Programming for Embedded Systems

## Unit 2

device. While straightforward, polling is inefficient as it wastes processing time monitoring devices that may not need immediate action, and it cannot prioritize critical tasks.

In contrast, the **interrupt method** allows devices to notify the microcontroller only when service is required by sending an interrupt signal. This approach lets the microcontroller focus on other tasks until an interrupt is triggered. When an interrupt occurs, the microcontroller pauses its current operation, executes the relevant **Interrupt Service Routine (ISR)**, and then resumes its work. This method is more efficient and versatile, as it supports task prioritization and the ability to mask or ignore certain requests when necessary.

Let's take the following scenario to highlight the advantage of interrupt over pooling. Imagine you are tasked with designing a system that monitors a temperature sensor in a factory. The objective is to activate a buzzer when the temperature exceeds a predefined threshold.

In the **Polling** method, the microcontroller repeatedly checks the sensor's value in a loop, monitoring it every few milliseconds. While it does this, it is unable to perform other tasks simultaneously.

Pooling Mechanism	Interrupt Mechanism
<pre>void main() {     while (1) {         int tempValue = readTemperatureSensor(); // Function to         read the temperature sensor         if (tempValue &gt; THRESHOLD) {             activateBuzzer(); // Activate buzzer if threshold is             exceeded         }         // Additional function: Check other sensor values or         perform maintenance tasks         performOtherTasks(); // Example of another task     } }</pre>	<pre>void ISR_TemperatureThreshold() {     activateBuzzer(); // Activate buzzer when temperature     exceeds threshold }  void main() {     while (1) {         // Main loop can perform other tasks concurrently         performOtherTasks(); // Example of another task     } }</pre>

In this approach, the microcontroller is constantly occupied with reading the sensor value and comparing it to the threshold. This consumes a lot of processing time, limiting the microcontroller's ability to handle other tasks efficiently. The microcontroller checks the sensor, activates the buzzer when necessary, and then repeats this process. Each iteration wastes processing cycles on the sensor check even if no action is needed.

Whereas, In the **Interrupt** method, the microcontroller uses an ISR (Interrupt Service Routine) that activates only when the sensor's threshold is reached.

When the temperature sensor's value crosses the threshold, it triggers an interrupt. The microcontroller halts its current task, executes the ISR, and then resumes the main loop. While waiting for an interrupt, the microcontroller can continue performing other tasks, such as checking other sensor values or handling communication protocols. This allows the microcontroller to operate more efficiently by not being tied up with unnecessary sensor checks.

The main advantage of interrupts is that they free up the microcontroller from constantly polling the sensor. This reduces wasted processor cycles and allows it to multitask effectively. For instance, in our example, while the microcontroller waits for an interrupt, it can monitor a different sensor or perform maintenance tasks, thus optimizing resource usage and responsiveness.

This scenario clearly illustrates why interrupts are preferable in real-time systems where time-sensitive and concurrent tasks are involved. By using interrupts, the microcontroller can handle events as they occur without wasting valuable processor time, improving system efficiency and performance.

## 7.2 Interrupt Service Routine (ISR) and Interrupt Vector Table

In microcontroller systems, each interrupt requires an associated Interrupt Service Routine (ISR), or interrupt handler. When an interrupt is triggered, the microcontroller automatically begins executing the ISR associated with that interrupt. Every interrupt corresponds to a specific memory location in the microcontroller's memory, where the starting address of its ISR is stored. This collection of memory locations dedicated to storing the addresses of ISRs is referred to as the interrupt vector table.

The Interrupt Vector Table for the ATmega32 microcontroller is a memory map that stores the starting addresses of the Interrupt Service Routines (ISRs). Each entry in this table corresponds to a specific interrupt source. When an interrupt occurs, the microcontroller fetches the ISR's starting address from the appropriate entry in the vector table and begins executing the corresponding ISR. This organized mapping allows the ATmega32 to handle



# Programming for Embedded Systems

## Unit 2

multiple events efficiently and respond quickly to changes in the system's environment. The fixed location for each ISR in the vector table ensures that the microcontroller can execute interrupt routines without delay, making it ideal for real-time applications.

The ATmega32 provides a variety of interrupt sources, including Port Pins (INT0, INT1, INT2), Timers (Timer0, Timer1, Timer2), UART, SPI, ADC, EEPROM, Analog Comparator, and TWI (I2C). Each entry in the vector table maps these interrupt sources to specific memory locations, which hold the starting addresses of their respective ISRs. This organization allows the microcontroller to quickly locate the appropriate ISR and respond in real-time to external and internal events, ensuring efficient handling of multiple tasks without wasting processor cycles.

**The table below summarizes the interrupts for the ATmega32:**

# Programming for Embedded Systems

## Unit 2

Vector No.	Program Address <sup>(2)</sup>	Source	Interrupt Definition
1	\$000 <sup>(1)</sup>	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	\$002	INT0	External Interrupt Request 0
3	\$004	INT1	External Interrupt Request 1
4	\$006	INT2	External Interrupt Request 2
5	\$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	\$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	\$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	\$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	\$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	\$012	TIMER1 OVF	Timer/Counter1 Overflow
11	\$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	\$016	TIMER0 OVF	Timer/Counter0 Overflow
13	\$018	SPI, STC	Serial Transfer Complete
14	\$01A	USART, RXC	USART, Rx Complete
15	\$01C	USART, UDRE	USART Data Register Empty
16	\$01E	USART, TXC	USART, Tx Complete
17	\$020	ADC	ADC Conversion Complete
18	\$022	EE_RDY	EEPROM Ready
19	\$024	ANA_COMP	Analog Comparator
20	\$026	TWI	Two-wire Serial Interface
21	\$028	SPM_RDY	Store Program Memory Ready

This organization allows the ATmega32 to respond to interrupts quickly by directly jumping to the ISR's starting address, making real-time responses seamless. This capability is crucial for real-time applications, enabling the microcontroller to efficiently handle multiple tasks without wasting processor cycles.

### 7.3 Interrupt Handling in ATmega32 Microcontroller

When an interrupt is triggered in the ATmega32 microcontroller, it follows a precise sequence:

# Programming for Embedded Systems

## Unit 2

1. **Complete the Current Instruction:** The microcontroller finishes the instruction it's currently executing and saves the address of the next instruction (program counter) on the stack. This ensures it can return to the exact point where it was interrupted.
2. **Jump to the Interrupt Vector Table:** It then jumps to a fixed memory location, the interrupt vector table, which contains the starting addresses of all ISRs (Interrupt Service Routines). The microcontroller fetches the ISR's starting address associated with the triggered interrupt.
3. **Execute the ISR:** The microcontroller begins executing the ISR, which handles the specific event—such as sensor readings or hardware interactions. The ISR completes its task and reaches the `RETI` instruction.
4. **Return from the ISR:** The microcontroller returns to the point where it was interrupted by popping the program counter address from the stack. This seamless return allows the microcontroller to continue its normal operation without delay.

This interrupt handling mechanism in the ATmega32 enables rapid response to events, ensuring efficient multitasking and minimal overhead in real-time applications.

### 7.4 Enabling and disabling an interrupt

Upon a reset, the microcontroller disables all interrupts, meaning it won't respond to any even if they are triggered. To allow the microcontroller to respond to interrupts, they must be enabled through software.



Figure 10-2. Bits of Status Register (SREG)

The global enabling or disabling of interrupts is controlled by the **I bit** (bit 7) of the **SREG (Status Register)**. To globally disable interrupts during critical tasks, the "CLI" (Clear Interrupt) instruction can be used to set the **I bit** to 0. This provides a simple and effective way to manage interrupt handling during important operations.

### 7.4.1 Steps to Enable an Interrupt

To enable an interrupt, follow these steps:

#### 1. Set the Global Interrupt Enable Bit:

The **I bit** (bit D7) of the **SREG register** must be set to HIGH to allow the microcontroller to respond to any interrupt. This can be done using the "**SEI**" (**Set Interrupt**) instruction.

#### 2. Enable Specific Interrupts:

Each interrupt has its own **Interrupt Enable (IE)** bit located in specific I/O registers. For example, the **TIMSK register** contains the interrupt enable bits for **Timer0**, **Timer1**, and **Timer2**. By setting the corresponding bit to HIGH, the interrupt is enabled for that specific peripheral.

**Note:** Even if an interrupt enable bit is set HIGH, the interrupt will not be processed unless the **I bit** in the **SREG register** is also HIGH.

### 7.5 External Hardware Interrupts

The ATmega32 microcontroller supports three external hardware interrupts, designated as **INT0**, **INT1**, and **INT2**. These interrupts are located on the following pins:

- **INT0:** PD2 (PORTD.2)
- **INT1:** PD3 (PORTD.3)
- **INT2:** PB2 (PORTB.2)

#### Behavior of External Interrupts:

- When one of these pins is activated, the microcontroller halts its current task and jumps to the **Interrupt Vector Table** to execute the corresponding **Interrupt Service Routine (ISR)**.
- The interrupt vector table assigns specific memory locations to these interrupts:
  - **INT0:** Address \$002
  - **INT1:** Address \$004
  - **INT2:** Address \$006

### 7.5.1 Enabling External Interrupts:

External interrupts must be enabled before they can take effect. This is done by setting the corresponding **INTx bit** in the **GICR (General Interrupt Control Register)**

#### Triggering Modes:

- **INT0 and INT1:** These interrupts can be configured to be either **level-triggered** (responding to a constant high/low signal) or **edge-triggered** (responding to a signal transition).
- **INT2:** This interrupt can only be configured as **edge-triggered**.

External interrupts are crucial for handling time-critical tasks or responding to events triggered by external hardware. The flexibility in configuring the triggering mode makes them suitable for a wide range of applications.

### 7.6 Programming External Interrupts of ATmega32

External interrupts in the ATmega32 microcontroller can be controlled and configured using specific registers: **GICR (General Interrupt Control Register)**, **MCUCR (MCU Control Register)**, and **MCUCSR (MCU Control and Status Register)**.

#### 1. GICR Register (General Interrupt Control Register)

The **GICR register** enables or disables external interrupts.

7	6	5	4	3	2	1	0
INT1	INT0	INT2	–	–	–	IVSEL	IVCE

#### Bit 7 – INT1: External Interrupt Request 1 Enable

- 0: Disable external interrupt
- 1: Enable external interrupt

#### Bit 6 – INT0: External Interrupt Request 0 Enable

- 0: Disable external interrupt
- 1: Enable external interrupt

### Bit 5 – INT2: External Interrupt Request 2 Enable





- 0: Disable external interrupt
- 1: Enable external interrupt

### 2. MCU Control Register (MCUCR)

- To define a level trigger or edge trigger on external INT0 and INT1 pins MCUCR register is used.

7	6	5	4	3	2	1	0
SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00





### Triggering Modes for INT0 (ISC01, ISC00):

ISC01	ISC00		Description
0	0		The low level on the INT0 pin generates an interrupt request.
0	1		Any logical change on the INT0 pin generates an interrupt request.
1	0		The falling edge on the INT0 pin generates an interrupt request.
1	1		The rising edge on the INT0 pin generates an interrupt request.

### Triggering Modes for INT1 (ISC11, ISC10):

# Programming for Embedded Systems



## Unit 2

ISC01	ISC00		Description
0	0		The low level on the INT1 pin generates an interrupt request.
0	1		Any logical change on the INT1 pin generates an interrupt request.
1	0		The falling edge on the INT1 pin generates an interrupt request.
1	1		The rising edge on the INT1 pin generates an interrupt request.

### 3. MCUCSR Register (MCU Control and Status Register)

The **MCUCSR register** defines the triggering mode for the INT2 interrupt using the **ISC2 bit**.

7	6	5	4	3	2	1	0
JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF

ISC2		Description
0		The falling edge on the INT2 pin generates an interrupt request.
1		The rising edge on the INT2 pin generates an interrupt request.

### 7.7 Best Practices for Interrupt Handling in Embedded Systems

Effective interrupt handling is crucial for reliable and efficient embedded system design. Below are some best practices to follow when working with interrupts:

#### 1. Keep ISRs Short and Efficient

- Minimize the amount of code in the **Interrupt Service Routine (ISR)** to ensure that the microcontroller quickly returns to its main tasks.
- Perform only critical operations in the ISR and defer non-urgent tasks to the main loop or a separate thread.

#### 2. Use Atomic Operations

- Ensure that shared resources accessed by both the ISR and the main program are handled using atomic operations to avoid race conditions.
- Use appropriate mechanisms like **disabling interrupts temporarily** or **mutexes** to protect critical sections.

#### 3. Avoid Blocking Calls in ISRs

- Do not include functions like delays or wait loops in the ISR, as these can disrupt the system's real-time performance.

#### 4. Prioritize Interrupts

- If multiple interrupts are used, assign priorities carefully to ensure that the most critical interrupt gets serviced first.

#### 5. Handle Interrupt Nesting Carefully

- For systems that allow nested interrupts, ensure higher-priority interrupts can preempt lower-priority ones, and restore states properly upon completion.

#### 6. Enable Only Necessary Interrupts

- Limit the number of enabled interrupts to reduce complexity and improve system reliability.

#### 7. Test and Debug Thoroughly

- Simulate and test interrupt conditions to identify potential issues like missed interrupts or unexpected behavior.
- Use tools like logic analyzers or debuggers to trace interrupt behavior.



### Steps to Program External Interrupts

#### 1. Enable Global Interrupts:

- Use the **SREG** (Status Register) to enable the global interrupt system by setting the **I-bit** (Interrupt Enable bit). **This can be done through sei( ), enable global interrupt.**

#### 2. Enable Specific External Interrupts:

- Use the **GICR (General Interrupt Control Register)** to enable the desired external interrupt pins:
  - Set **INT0** for External Interrupt 0 (PD2).
  - Set **INT1** for External Interrupt 1 (PD3).
  - Set **INT2** for External Interrupt 2 (PB2).

#### 3. Select Triggering Mode:

- Configure the type of event (e.g., rising edge, falling edge, or low-level) that will trigger the interrupt:

#### 4. Define the Interrupt Service Routine (ISR):

- Write the ISR in your code to specify what the microcontroller should do when the interrupt is triggered.
- Use the **ISR ( )** macro to define the function for each external interrupt.

### Programming Examples

#### 1. External Interrupt to toggle the PORTC

Code:

```
/*
 * GccApplication3.c
 *
 * Created: 12/24/2024 6:40:11 PM
 * Author : Deepesh
 */

#define F_CPU 8000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

/*Interrupt Service Routine for INT0*/
ISR(INT0_vect)
{
    PORTC=~PORTC;          /* Toggle PORTC */
    _delay_ms(50);          /* Software debouncing control delay */
}
```

# Programming for Embedded Systems

## Unit 2

```
}

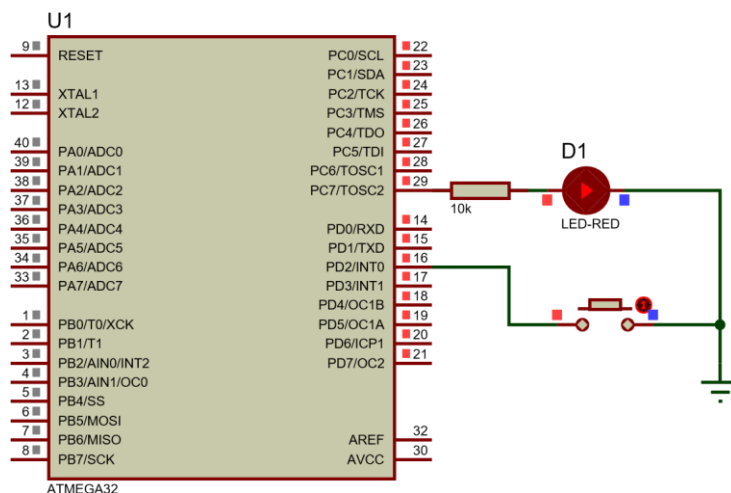
int main(void)
{
    DDRC=0xFF;          /* Make PORTC as output PORT*/
    PORTC=0;
    DDRD=0;              /* PORTD as input */
    PORTD=0xFF;          /* Make pull up high */

    GICR = 1<<INT0;      /* Enable INT0*/
    MCUCR = 1<<ISC01 | 1<<ISC00; /* Trigger INT0 on rising edge */

    sei();                /* Enable Global Interrupt */

    while(1);            /* Void Loop no-work */
}
```

### Circuit Diagram



2. External Interrupt to toggle the PORT C.7, while if interrupt is not raised continuous toggle PORT A.

Code:

```
/*
 * GccApplication3.c
 *
 * Created: 12/24/2024 6:40:11 PM
 * Author : Deepesh
 */

#define F_CPU 8000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

/*Interrupt Service Routine for INT0*/
```

# Programming for Embedded Systems

## Unit 2

```
ISR(INT0_vect)
{
    PORTC=~PORTC;          /* Toggle PORTC */
    _delay_ms(50);         /* Software debouncing control delay */
}

void setup()
{
    DDRC=0xFF;             /* Make PORTC as output PORT*/
    DDRA=0xFF;             /* Make PORTC as output PORT*/
    PORTC=0;
    DDRD=0;                /* PORTD as input */
    PORTD=0xFF;            /* Make pull up high */

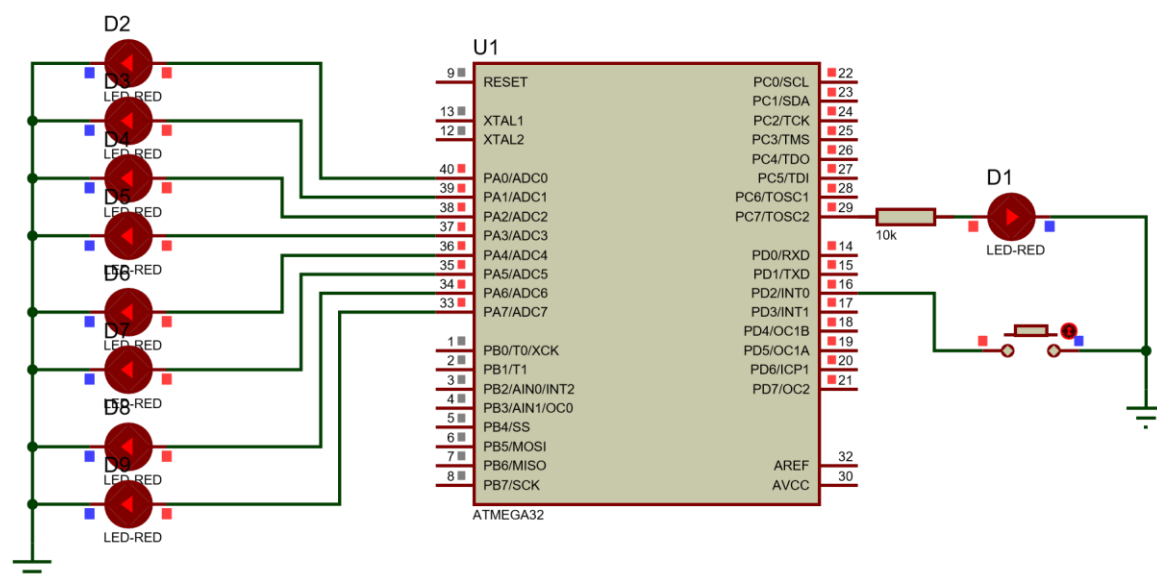
    GICR = 1<<INT0;        /* Enable INT0*/
    MCUCR = 1<<ISC01 | 1<<ISC00; /* Trigger INT0 on rising edge */

    sei();                 /* Enable Global Interrupt */
}

int main(void)
{
    setup();

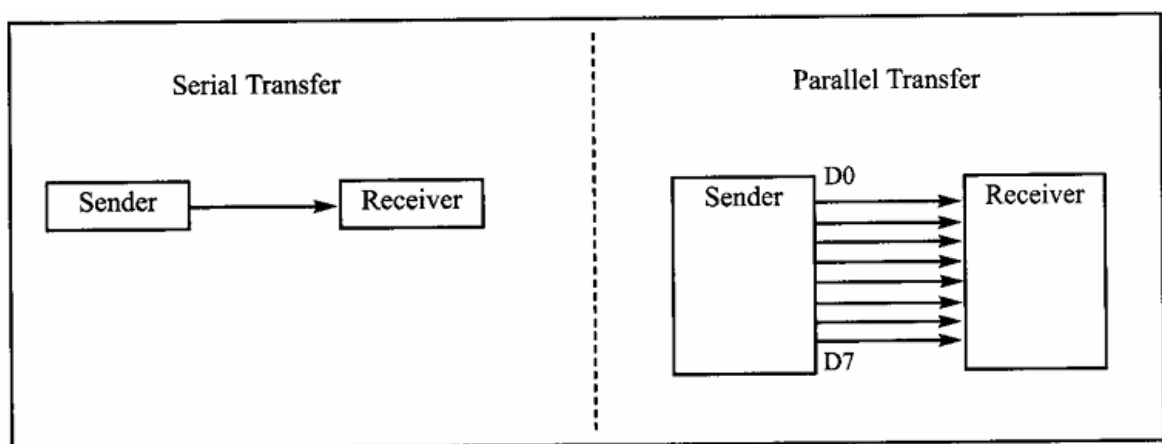
    while(1){
        PORTA=~PORTA;      /* Toggle PORTC */
        _delay_ms(50);     /* Software debouncing control delay */
    }
}
```

### Circuit Diagram



### 8 Serial Communication in AVR

Communication between electronic devices can be classified into two types: **parallel** and **serial**. Parallel communication, which uses multiple data lines (typically 8 or more) to transfer data simultaneously, is fast but practical only over short distances due to signal degradation and interference. It is commonly used in devices like printers and IDE hard drives. For long-distance communication spanning hundreds of feet to miles, however, **serial communication** is preferred. By transmitting data one bit at a time using a single data line, serial communication is cost-effective and more resilient to signal issues.



**Figure 11-1. Serial versus Parallel Data Transfer**

The **AVR ATmega32** microcontroller is well-equipped for serial communication, using its built-in **USART (Universal Synchronous/Asynchronous Receiver-Transmitter)** module. This module streamlines the process of sending and receiving data serially, reducing the need for complex coding.

In serial communication, data in byte-sized chunks (8 bits) is converted to a stream of bits using a **parallel-in-serial-out shift register** for transmission. At the receiving end, the bits are reassembled into bytes using a **serial-in-parallel-out shift register**. Two main methods of serial communication exist:

- **Asynchronous:** Transfers one byte at a time without a shared clock between sender and receiver.
- **Synchronous:** Transfers multiple bytes at a time using a synchronization clock shared between sender and receiver.

While software can be written to handle these methods, the process can be tedious. This is why hardware solutions like UARTs (Universal Asynchronous Receiver-Transmitters) and USARTs are widely used.

### RS232 Standards

To ensure compatibility among data communication equipment from various manufacturers, the **RS232 standard** was introduced by the Electronics Industries Association (EIA) in 1960 and revised over the years (RS232A, RS232B, and RS232C). Today, RS232 remains one of the most widely used serial I/O standards in PCs and numerous devices. However, because RS232 was established before the advent of **TTL logic**, its voltage levels are not directly compatible with modern microcontrollers like ATmega32. Specifically:

- **Logical 1:** Represented by voltages between -3V to -25V.
- **Logical 0:** Represented by voltages between +3V to +25V.
- **Undefined Range:** -3V to +3V.

This necessitates the use of a **voltage level converter**, such as the MAX232 IC, to bridge the gap between RS232 and TTL voltage levels.

### Interfacing ATmega32 with RS232

For Universal Synchronous and Asynchronous Receiver-Transmitter (USART) communication, only three wires are necessary:

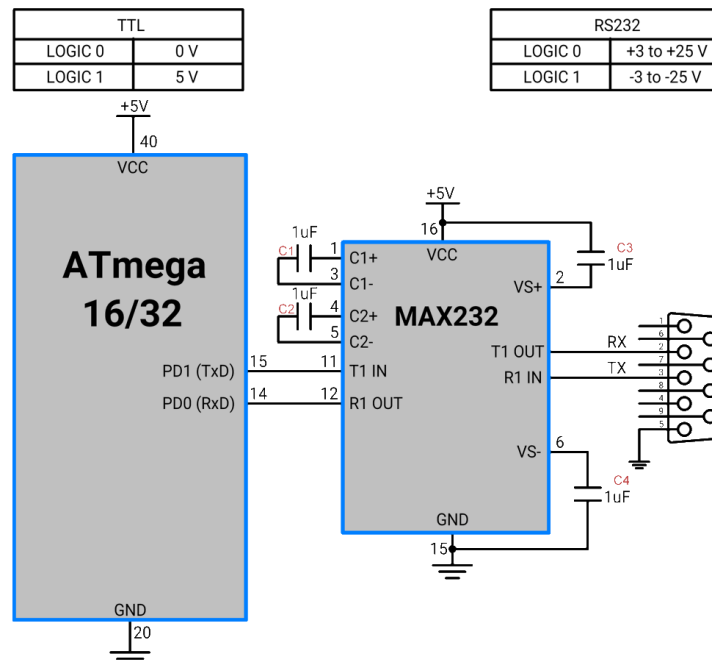
- **Tx (Transmit):** Responsible for sending data from the microcontroller.
- **Rx (Receive):** Responsible for receiving data at the microcontroller.
- **GND (Ground):** Provides a common reference for signal levels between the devices.

The ATmega32 microcontroller operates at **TTL voltage levels**, where Logic 0 is represented by **0V** and Logic 1 is represented by **5V**. However, for RS232 logic 0 is represented by **+3V to +25V** and logic 1 is represented by **-3V to -25V**. This mismatch in voltage levels makes direct communication between the ATmega32 and RS232 devices impossible without a voltage level converter. To facilitate communication between the ATmega32 and RS232 devices, a voltage level converter like the **MAX232 IC** is required.

# Programming for Embedded Systems

## Unit 2

The MAX232 translates TTL signals to RS232 voltage levels and vice versa, enabling seamless data exchange between the two systems.



In older computers, **DB9 connectors** were commonly used to interface with external peripherals. Although the DB9 connector has nine pins, only three are necessary for basic **USART communication**:

- **Pin 2 (Tx):** Connects to the receiving pin (Rx) of the other device.
- **Pin 3 (Rx):** Connects to the transmitting pin (Tx) of the other device.
- **Pin 5 (GND):** Connects to the common ground of both devices.

This simplified setup ensures reliable and efficient serial communication, avoiding the need to utilize all nine pins on the DB9 connector.

### 8.1 Programming USART in AVR

USART (Universal Synchronous and Asynchronous Receiver-Transmitter) in AVR microcontrollers requires a thorough understanding of its basic registers to ensure proper functionality. Below is an overview of the essential registers and their functionalities:

### 8.1.1 AVR Basic Registers for USART

#### 1. UDR (USART Data Register):

- The UDR serves as both the transmit (Tx) and receive (Rx) buffer.
- Writing to the UDR register stores data in the Tx buffer, while reading from it retrieves data from the Rx buffer.
- A FIFO (First In, First Out) shift register ensures efficient data handling during transmission.

#### 2. UCSRA (USART Control and Status Register A):

The USART Control and Status Register A (UCSRA) is primarily used to manage control and status flags for the USART module. Additionally, there are two other control and status registers, UCSRB and UCSRC, which serve complementary roles in configuring and monitoring USART operations.

#### 3. UBRR

The UBRR (USART Baud Rate Register) is a 16-bit register used to configure the baud rate for USART communication, ensuring the correct data transmission speed.

#### UCSRA (USART Control and Status Register A) Bit Descriptions

7	6	5	4	3	2	1	0
RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM

- **Bit 7 – RXC (USART Receive Complete):**

This flag is set when there is unread data in the UDR register. It can be used to trigger a Receive Complete interrupt.

- **Bit 6 – TXC (USART Transmit Complete):**

This flag is set when the entire frame from the transmit buffer (Tx Buffer) is shifted out and no new data is currently present in the transmit buffer (UDR). The TXC flag is automatically cleared upon the execution of a Transmit Complete interrupt or by writing a 1 to this bit. It can also generate a Transmit Complete interrupt.

- **Bit 5 – UDRE (USART Data Register Empty):**

# Programming for Embedded Systems

## Unit 2

This flag indicates that the transmit buffer (UDR) is empty and ready to receive new data. A 1 in this flag suggests the buffer is empty. The UDRE flag can generate a Data Register Empty interrupt and is set by default after a reset.

- **Bit 4 – FE (Frame Error):**

This bit is set when a frame error is detected, typically caused by incorrect start or stop bits in the received data.

- **Bit 3 – DOR (Data OverRun):**

This flag is set if a Data OverRun condition occurs. A Data OverRun happens when the receive buffer is full (two characters), and a new character arrives in the receive shift register, overwriting existing data.

- **Bit 2 – PE (Parity Error):**

This bit is set when a parity error is detected during data reception.

- **Bit 1 – U2X (Double the USART Transmission Speed):**

Writing 1 to this bit doubles the USART transmission speed, improving communication efficiency.

- **Bit 0 – MPCM (Multi-Processor Communication Mode):**

This bit is used to enable Multi-Processor Communication Mode, allowing the USART to distinguish between data and address frames in a multi-processor setup.

### UCSRB (USART Control and Status Register B) Bit Descriptions

7	6	5	4	3	2	1	0
RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8

- **Bit 7 – RXCIE (RX Complete Interrupt Enable):**

Setting this bit to 1 enables an interrupt when the RXC flag (Receive Complete) is set.

- **Bit 6 – TXCIE (TX Complete Interrupt Enable):**

Setting this bit to 1 enables an interrupt when the TXC flag (Transmit Complete) is set.

- **Bit 5 – UDRIE (USART Data Register Empty Interrupt Enable):**

Setting this bit to 1 enables an interrupt when the UDRE flag (Data Register Empty) is set.

- **Bit 4 – RXEN (Receiver Enable):**

Writing 1 to this bit activates the USART receiver, allowing it to receive data.

- **Bit 3 – TXEN (Transmitter Enable):**



# Programming for Embedded Systems

## Unit 2

Writing 1 to this bit activates the USART transmitter, enabling it to send data.

- **Bit 2 – UCSZ2 (Character Size):**

The UCSZ2 bit, along with the UCSZ1:0 bits in the UCSRC register, defines the number of data bits (character size) in a frame used by the receiver and transmitter.

- **Bit 1 – RXB8 (Receive Data Bit 8):**

This bit stores the 8th data bit of the received frame when using 9-bit data communication.

- **Bit 0 – TXB8 (Transmit Data Bit 8):**

This bit holds the 8th data bit for transmission when 9-bit data communication is enabled.

### UCSRC (USART Control and Status Register C) Bit Descriptions

7	6	5	4	3	2	1	0
URSEL	UMSEL	UPM1	UPM0	USBS	USCZ1	USCZ0	UCPOL

- **Bit 7 – URSEL (Register Select):**

This bit determines whether the UCSRC or UBRRH register is accessed.

- 1 = UCSRC register selected for writing.
- 0 = UBRRH register selected for writing.

- **Bit 6 – UMSEL (USART Mode Select):**

This bit sets the USART operating mode:

- 0 = Asynchronous Mode.
- 1 = Synchronous Mode.

- **Bits 5:4 – UPM1:0 (Parity Mode):**

These bits enable and define the type of parity used for error detection. If a parity mismatch occurs, the PE (Parity Error) flag in UCSRA is set.

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

- **Bit 3 – USBS (Stop Bit Select):**

# Programming for Embedded Systems

## Unit 2

This bit selects the number of stop bits transmitted:

- 0 = 1 Stop Bit.
- 1 = 2 Stop Bits.
- **Bits 2:1 – UCSZ1:0 (Character Size):**

Along with the UCSZ2 bit in UCSRB, these bits define the number of data bits (character size) in a frame:

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
<b>0</b>	<b>1</b>	<b>1</b>	<b>8-bit</b>
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

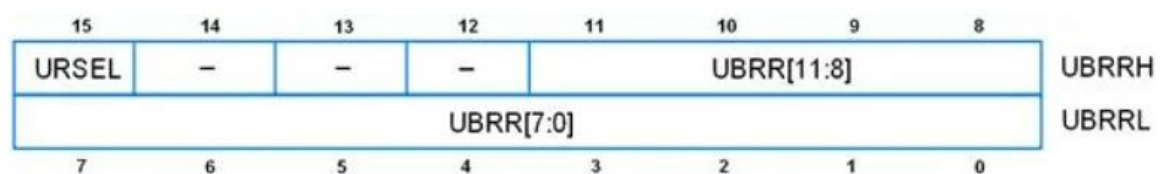
- **Bit 0 – UCPOL (Clock Polarity):**

This bit is only applicable in synchronous mode and determines the clock polarity:

- 0 = Data changes on rising edge and is sampled on falling edge of the clock.
- 1 = Data changes on falling edge and is sampled on rising edge of the clock.

In asynchronous mode, this bit must be set to 0.

### UBRRL and UBRRH: USART Baud Rate Registers



The USART Baud Rate Registers (UBRRL and UBRRH) are used to set the baud rate for communication. These two registers work together to form a 12-bit value, where:

- **UBRRL (USART Baud Rate Register Low):** Holds the lower 8 bits of the 12-bit UBRR value.

# Programming for Embedded Systems

## Unit 2

- **UBRRH (USART Baud Rate Register High):** Holds the upper 4 bits of the 12-bit UBRR value.
- **Bit 15 – URSEL (Register Select):**
  - Selects between UCSRC and UBRRH registers, as both share the same address.
  - Set **URSEL = 1** to access UCSRC.
  - Set **URSEL = 0** to access UBRRH.
- **Bits 11:0 – UBRR11:0 (Baud Rate Setting):**
  - Used to define the baud rate for data transmission and reception.
  - Formula to calculate the UBRR value:

$$\text{UBRR} = \frac{F_{osc}}{16 \times \text{BaudRate}} - 1$$

- Formula to calculate the baud rate from UBRR:

$$\text{BaudRate} = \frac{F_{osc}}{16 \times (\text{UBRR} + 1)}$$

### Example Calculation:

Suppose  $F_{osc} = 8 \text{ MHz}$  and the required baud rate is 9600 bps

1. Using the formula:

$$\text{UBRR} = \frac{8,000,000}{16 \times 9600} - 1 = 51.088$$

2. Round the result:

$$\text{UBRR} = 51$$

The lower 8 bits (**51 in decimal = 00110011 in binary**) are stored in UBRR1, and the upper 4 bits (**0000 in binary**) are stored in UBRRH.

### Steps to Program UART

#### 1. Set the Baud Rate:

- Use the **UBRR (USART Baud Rate Register)** to define the communication speed.
  - Calculate the UBRR value using the formula:
  - Split the value into **UBRRH (High byte)** and **UBRRL (Low byte)** registers.

#### 2. Configure USART Control Registers:

# Programming for Embedded Systems

## Unit 2

- Use the **UCSRB (USART Control and Status Register B)** to enable transmitter and receiver:
  - Set the **TXEN** bit to enable the transmitter.
  - Set the **RXEN** bit to enable the receiver.
- Optionally, enable interrupt-driven communication by setting the **RXCIE** (Receive Complete Interrupt Enable) and/or **TXCIE** (Transmit Complete Interrupt Enable) bits.

### 3. Set Frame Format:

- Use the **UCSRC (USART Control and Status Register C)** to configure the frame format:
  - **UCSZ1, UCSZ0**: Select data size (e.g., 8 bits, 9 bits).
  - **UPM1, UPM0**: Set parity mode (e.g., no parity, even, odd).
  - **USBS**: Choose stop bit (1 or 2 stop bits).

### 4. Transmit Data:

- Write data to the **UDR (USART Data Register)** to send it through the UART.
- Ensure the **UDRE (Data Register Empty)** flag in **UCSRA** is set before writing to avoid overwriting previous data.

### 5. Receive Data:

- Read data from the **UDR (USART Data Register)** after checking that the **RXC (Receive Complete)** flag in **UCSRA** is set.
- Optionally, handle errors by checking the **FE (Frame Error)**, **DOR (Data OverRun)**, and **PE (Parity Error)** bits in **UCSRA**.

## Programming Examples

1. Program to transfer the letter “G” continuously using 8-bit data and 1 stop bit with crystal of 8MHz frequency

### Code:

```
/*
 * GccApplication4.c
 *
 * Created: 12/26/2024 6:37:26 PM
 * Author : Deepesh
 */
```

# Programming for Embedded Systems

## Unit 2

```
#define F_CPU 8000000UL          /* Define frequency here its 8MHz */
#include <avr/io.h>
#include <util/delay.h>

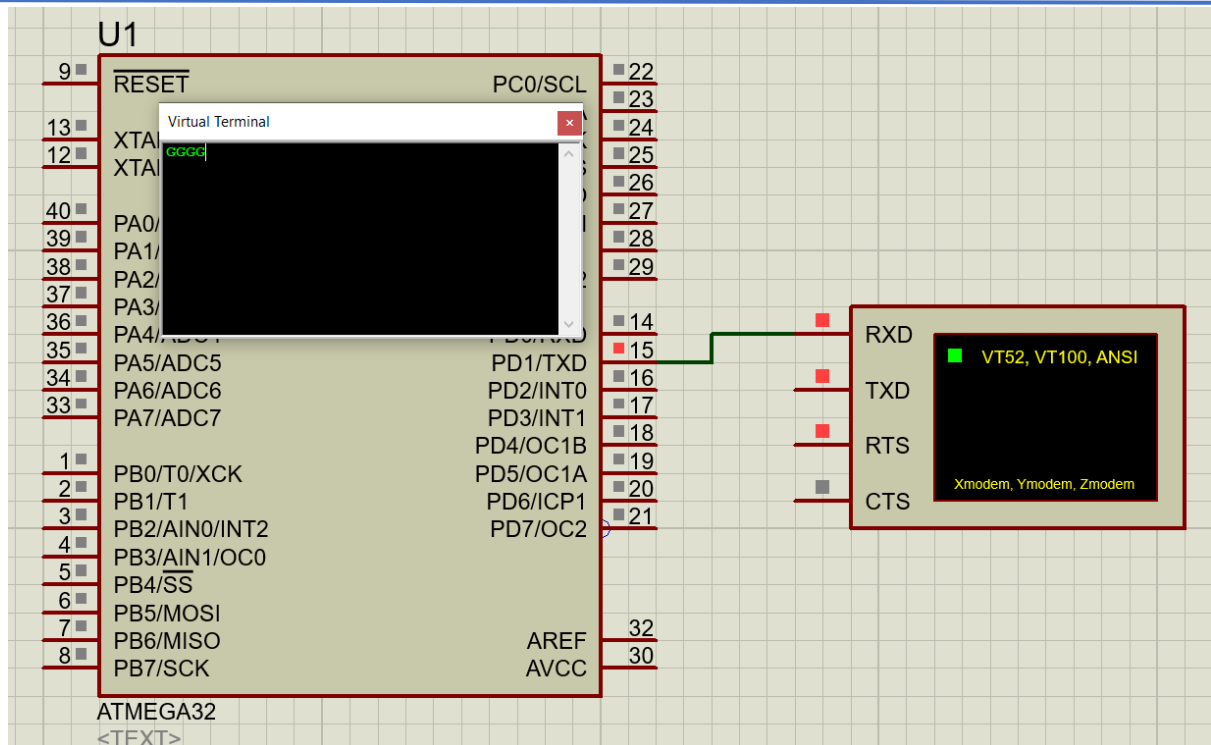
void usart_init (void)
{
    UCSRB = (1<<TXEN); /* Turn on transmission */
    UCSRC = (1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL); /*asynchronous mode, 8-bit, 1-stop
bit*/
    UBRRL = 0X33;
}

void usart_send(unsigned char ch)
{
    while(!(UCSRA &(1<< UDRE))); /* Wait for empty transmit buffer */
    UDR =ch;
}

int main(void)
{
    usart_init ();
    while(1)
    {
        usart_send('G');
        _delay_ms(1000);
    }
}
```

# Programming for Embedded Systems

## Unit 2



2. Program to send the message "The Earth is but One Country" to the serial port continuously using the previous setting.

### Code

```
/*
 * GccApplication4.c
 *
 * Created: 12/26/2024 6:37:26 PM
 * Author : Deepesh
 */

#define F_CPU 8000000UL /* Define frequency here its 8MHz */
#include <avr/io.h>
#include <util/delay.h>

void usart_init (void)
{
    UCSRB = (1<<TXEN); /* Turn on transmission */
    UCSRC = (1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL); /*asynchronous mode, 8-bit, 1-stop
bit*/
    UBRRL = 0X33;
}

void usart_send(unsigned char ch)
{
    while(!(UCSRA &(1<< UDRE))); /* Wait for empty transmit buffer */
    UDR =ch;
}
```

# Programming for Embedded Systems

## Unit 2

```
}  
int main(void)  
{  
    unsigned char str[30]="The Earth is but One Country. ";  
    unsigned char StrLength = 30;  
    unsigned char i =0;  
    usart_init ();  
    while(1)  
    {  
        usart_send(str[i++]);  
        if(i >= StrLength)  
            i = 0;  
        _delay_ms(100);  
    }  
    return 0;  
}
```

### Circuit Diagram

