Es (4351102) - Problem Solver

Antigravity AI

June 4, 2026

Es

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Contents

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

CHAPTER 1

Overview of Embedded System

This chapter focuses on the computational aspects involved in the design and analysis of embedded systems. These quantitative methods are essential for determining memory requirements, system performance, power management, and selecting appropriate microcontrollers for specific applications.

1.1 Memory Capacity and Address Decoding

In an embedded system, the processor must communicate with memory. The size of the memory is directly governed by the number of address lines available on the microcontroller's system bus.

Methodology:

Memory Capacity and Address Range Calculation To calculate the total memory capacity and determine the address range of an embedded system:

1. **Identify Address Lines:** Determine the number of address lines (n) of the microcontroller. The total number of uniquely addressable memory locations is given by 2^n.
2. **Identify Data Bus Width:** Determine the word size (data bus width). If the data bus is 8 bits wide, each memory location holds 1 byte.
3. **Calculate Total Memory Capacity:** Multiply the number of locations by the word size:

   $$\text{Memory Capacity} = 2^n \times \text{Word Size}$$

   Convert the result to Kilobytes (KB) or Megabytes (MB) using 1 KB = 1024 bytes and 1 MB = 1024 KB. Note that in binary conventions, $2^{10} = 1024$.
4. **Determine Address Range:** The lowest address is when all address lines are 0. The highest address is when all address lines are 1. Convert these binary values to hexadecimal to represent the address range.

Worked Example:

Calculate memory capacity and address range A microcontroller has 16 address lines and an 8-bit data bus. Calculate the total memory capacity in Kilobytes (KB) and determine the memory address range in hexadecimal.
Step 1: Find total addressable locations.
Number of address lines $n = 16$.
Total locations $= 2^{16} = 65536$.
Step 2: Determine total memory capacity.
Since the data bus is 8-bit, the word size is 1 byte.
Capacity in bytes = 65536 bytes.
Capacity in KB $= \frac{65536}{1024} = 64$ KB.

Step 3: Determine the address range.
Lowest address in binary (16 bits): $0000\ 0000\ 0000\ 0000_2$
Convert to Hexadecimal: 0000_{16}.
Highest address in binary (16 bits): $1111\ 1111\ 1111\ 1111_2$
Convert to Hexadecimal: $FFFF_{16}$.
Final Answer: The total memory capacity is 64 KB, and the address range is from 0000H to FFFFH.

1.2 System Performance and Execution Time

The speed of an embedded system depends on its clock frequency and the architecture of the processor, which dictates how many clock cycles are required to execute a single machine cycle.

Methodology:

Instruction Execution Time Calculation To calculate the time required for a microcontroller to execute an instruction:

1. **Determine Clock Period:** Given the clock frequency (f_{clk}), calculate the time period of one clock cycle (T_{clk}) using the formula:
$$T_{clk} = \frac{1}{f_{clk}}$$
2. **Calculate Machine Cycle Time:** Identify the number of clock cycles per machine cycle (C_{MC}) for the specific microcontroller architecture. Compute the machine cycle time (T_{MC}):
$$T_{MC} = C_{MC} \times T_{clk}$$
3. **Calculate Execution Time:** Determine how many machine cycles (M) the instruction requires. The total execution time (T_{exec}) is:
$$T_{exec} = M \times T_{MC}$$

Worked Example:

Calculate instruction execution time An 8051 microcontroller is operating at a crystal frequency of 11.0592 MHz. In the 8051 architecture, one machine cycle consists of 12 clock cycles. Calculate the execution time for an instruction that takes 2 machine cycles to complete.
Step 1: Determine Clock Period (T_{clk}).
Clock frequency $f_{clk} = 11.0592$ MHz $= 11.0592 \times 10^6$ Hz.
$T_{clk} = \frac{1}{11.0592 \times 10^6} \approx 90.42 \times 10^{-9}$ seconds $= 90.42$ ns.
Step 2: Calculate Machine Cycle Time (T_{MC}).
Clock cycles per machine cycle $C_{MC} = 12$.
$T_{MC} = 12 \times 90.42$ ns $= 1085$ ns $= 1.085\mu$s.
Step 3: Calculate Total Execution Time (T_{exec}).
Number of machine cycles $M = 2$.
$T_{exec} = 2 \times 1.085\mu$s $= 2.17\mu$s.
Final Answer: The execution time for the instruction is 2.17μs.

1.3 Power Consumption and Battery Life Estimation

Power management is a critical aspect of embedded system design, particularly for portable or remote battery-operated devices. Devices often alternate between a high-power active mode and a low-power sleep mode to conserve energy.

Methodology:

Battery Life Estimation Method To estimate the operational lifespan of a battery-powered embedded system:

1. **Identify Battery Specifications:** Obtain the battery capacity in milliampere-hours (mAh). Apply a derating factor (D) to account for self-discharge and environmental effects:

$$\text{Effective Capacity} = \text{Nominal Capacity} \times D$$

2. **Calculate Average Current Consumption:** If the system runs in multiple modes (e.g., active and sleep), find the average current (I_{avg}) over a complete operation cycle of duration T_{total}:

$$I_{avg} = \frac{(I_{active} \times T_{active}) + (I_{sleep} \times T_{sleep})}{T_{total}}$$

where $T_{total} = T_{active} + T_{sleep}$. Make sure all time units and current units are consistent.

3. **Calculate Battery Life:** Divide the effective capacity by the average current consumption to find the life in hours, then convert to days or years as appropriate:

$$\text{Battery Life (hours)} = \frac{\text{Effective Capacity (mAh)}}{I_{avg}\ \text{(mA)}}$$

Worked Example:

Estimate battery life for an IoT sensor An embedded IoT sensor operates using a 1200 mAh battery. The sensor wakes up every 10 seconds to take a reading and transmit data. The active mode lasts for 50 milliseconds drawing 30 mA. During the remaining time of the 10-second cycle, the system is in sleep mode drawing 15μA. Assuming a battery derating factor of 0.85, calculate the estimated battery life in days.

Step 1: Calculate Effective Battery Capacity.

Nominal Capacity = 1200 mAh. Derating Factor $D = 0.85$.

Effective Capacity = $1200 \times 0.85 = 1020$ mAh.

Step 2: Determine times for each mode.

Total cycle time $T_{total} = 10$ s.

Active time $T_{active} = 50$ ms $= 0.05$ s.

Sleep time $T_{sleep} = 10\text{ s} - 0.05\text{ s} = 9.95$ s.

Step 3: Calculate Average Current (I_{avg}).

Active current $I_{active} = 30$ mA.

Sleep current $I_{sleep} = 15\mu\text{A} = 0.015$ mA.

$$I_{avg} = \frac{(30 \times 0.05) + (0.015 \times 9.95)}{10}$$

$$I_{avg} = \frac{1.5 + 0.14925}{10} = \frac{1.64925}{10} = 0.164925\text{ mA}$$

Step 4: Calculate Battery Life.

Battery Life (hours) $= \frac{1020\text{ mAh}}{0.164925\text{ mA}} \approx 6184.6$ hours.

Battery Life (days) $= \frac{6184.6}{24} \approx 257.7$ days.

Final Answer: The estimated battery life for the IoT sensor is approximately 257.7 days.

1.4 Microcontroller Selection Analysis

Choosing the right microcontroller for a specific embedded application is critical. Using a weighted scoring matrix allows designers to objectively evaluate multiple microcontroller options based on design constraints like cost, speed, power, and peripheral support.

Methodology:

Weighted Scoring Matrix for Microcontroller Selection

1. **Define Criteria and Weights:** Identify the key requirements of the application. Assign a weight (W_j) to each criterion such that the sum of all weights equals 100% (or 1.0).
2. **Assign Scores:** For each microcontroller candidate i, assign a score (S_{ij}) on a standardized scale (e.g., 1 to 10) for each criterion j, where a higher score indicates better performance in that category.
3. **Calculate Weighted Scores:** Multiply the score by its corresponding weight for each criterion:

$$\text{Weighted Score}_{ij} = S_{ij} \times W_j$$

4. **Compute Total Score:** Sum the weighted scores for each microcontroller.

$$\text{Total Score}_i = \sum_j (S_{ij} \times W_j)$$

The candidate with the highest total score is the optimal choice.

Worked Example:

Select microcontroller for a portable medical device A design team must select a microcontroller for a portable, battery-operated medical device. The selection criteria and their respective weights are: Power Efficiency (45%), Unit Cost (25%), Available Memory (20%), and Number of I/O Pins (10%). Two microcontrollers, MCU A and MCU B, are evaluated and scored out of 10 for each criterion as follows:

- MCU A: Power Efficiency (9), Cost (6), Memory (7), I/O Pins (8)
- MCU B: Power Efficiency (7), Cost (9), Memory (8), I/O Pins (7)

Determine the optimal microcontroller using the weighted scoring matrix.
Step 1: List the Weights.
$W_{\text{Power}} = 0.45$, $W_{\text{Cost}} = 0.25$, $W_{\text{Memory}} = 0.20$, $W_{\text{IO}} = 0.10$.
Step 2: Calculate Weighted Scores for MCU A.
Power: $9 \times 0.45 = 4.05$
Cost: $6 \times 0.25 = 1.50$
Memory: $7 \times 0.20 = 1.40$
I/O: $8 \times 0.10 = 0.80$
Total Score for MCU A $= 4.05 + 1.50 + 1.40 + 0.80 = 7.75$.
Step 3: Calculate Weighted Scores for MCU B.
Power: $7 \times 0.45 = 3.15$
Cost: $9 \times 0.25 = 2.25$
Memory: $8 \times 0.20 = 1.60$
I/O: $7 \times 0.10 = 0.70$
Total Score for MCU B $= 3.15 + 2.25 + 1.60 + 0.70 = 7.70$.
Step 4: Compare Total Scores.
MCU A Total Score = 7.75
MCU B Total Score = 7.70
Since MCU A has the higher total score, it is the optimal choice for this specific application.
Final Answer: MCU A is the optimal microcontroller with a score of 7.75.

CHAPTER 2

AVR Microcontroller Architecture and Pin Diagram

2.1 AVR Memory Architecture and Organization

The ATmega32 utilizes a Harvard architecture with separate memories and buses for program and data. The memory is broadly divided into Program Memory (Flash), Data Memory (Internal SRAM, General Purpose Registers, and I/O Memory), and EEPROM. Problem-solving in this domain typically involves mapping physical memory addresses based on memory sizes.

Methodology:

Memory Address Calculation To find the start and end addresses of any memory segment in the AVR architecture:

1. **Identify the starting address** of the specific memory segment (e.g., 0x0000 for Program Memory, 0x0000 for GPRs, 0x0020 for I/O Registers, 0x0060 for ATmega32 Internal SRAM).
2. **Determine the size** of the segment in bytes.
3. **Calculate the end address** using the formula:
   End Address = Start Address + Size in Bytes − 1.
4. Convert all decimal values to hexadecimal for standard memory mapping representation.

Worked Example:

Calculating SRAM and EEPROM Memory Ranges The ATmega32 microcontroller features 2 KB of Internal SRAM and 1024 bytes of EEPROM. Calculate the hexadecimal start and end addresses for both memory spaces, given that internal SRAM starts after the I/O memory at 0x0060 and EEPROM starts at 0x0000.
Solution:

1. **Internal SRAM:**
   - Start Address: 0x0060
   - Size: 2 KB = $2 \times 1024 = 2048$ bytes.
   - Convert size to hex: $2048_{10} = 0800_{16}$.
   - End Address: $0x0060 + 0x0800 - 1 = 0x085F$.
2. **EEPROM:**
   - Start Address: 0x0000
   - Size: 1024 bytes.
   - Convert size to hex: $1024_{10} = 0400_{16}$.
   - End Address: $0x0000 + 0x0400 - 1 = 0x03FF$.

2.2 I/O Port and Pin Configuration

The ATmega32 has 32 general-purpose I/O pins configured through four 8-bit ports (PORTA, PORTB, PORTC, PORTD). Three memory-mapped registers control each port: DDRx (Data Direction Register), PORTx (Data Register), and PINx (Port Input Pins).

Methodology:

Configuring Port Direction and Pull-up Resistors To configure I/O pins for digital input or output:

1. **Data Direction (DDRx):** Set the bit to `1` to configure the pin as an output. Clear the bit to `0` to configure it as an input.
2. **Data Output (PORTx - when Output):** If the pin is an output, setting the PORTx bit to `1` drives it HIGH. Clearing it to `0` drives it LOW.
3. **Pull-up Resistor (PORTx - when Input):** If the pin is an input, setting the PORTx bit to `1` enables the internal pull-up resistor. Clearing it leaves the pin floating (tri-state).
4. Combine the bit values into a single 8-bit binary number and convert to hexadecimal.

Worked Example:

Generating Hex Values for IO Configuration Configure PORTB of the ATmega32 such that the lower nibble (Pins 0 to 3) acts as input with internal pull-up resistors enabled, and the upper nibble (Pins 4 to 7) acts as output driving a LOW state. Provide the hex values for DDRB and PORTB.
Solution:

1. **Determine DDRB:**
   - Upper nibble (Output): Bits 7, 6, 5, 4 must be `1`.
   - Lower nibble (Input): Bits 3, 2, 1, 0 must be `0`.
   - Binary value: `0b11110000`
   - Hex value: `0xF0`
2. **Determine PORTB:**
   - Upper nibble (Output LOW): Bits 7, 6, 5, 4 must be `0`.
   - Lower nibble (Pull-up Enable): Bits 3, 2, 1, 0 must be `1`.
   - Binary value: `0b00001111`
   - Hex value: `0x0F`

2.3 Status Register and Arithmetic Operations

The Status Register (SREG) contains flags that indicate the results of the most recent Arithmetic Logic Unit (ALU) operation. The key flags are Carry (C), Zero (Z), Negative (N), Overflow (V), Sign (S), and Half-carry (H).

Methodology:

Evaluating Status Register Flags To find the state of the flags after an 8-bit addition or subtraction:

1. Perform the binary operation on the 8-bit operands.
2. **C Flag:** Set to 1 if there is a carry-out from the 8th bit (MSB), else 0.
3. **H Flag:** Set to 1 if there is a carry-out from the 4th bit (bit 3 to bit 4), else 0.

4. **Z Flag:** Set to 1 if the entire 8-bit result is strictly 0x00, else 0.
5. **N Flag:** Set to 1 if the MSB (bit 7) of the result is 1, else 0.

Worked Example:

Determining Flag States after Addition Determine the 8-bit result and the state of the C, H, Z, and N flags after the ALU adds `0x85` and `0x92`.
Solution:

1. Convert to binary and add:

```
  1000 0101  (0x85)
+ 1001 0010  (0x92)
-------------------
1 0001 0111  (Result = 0x17, Carry Out = 1)
```

2. **Result:** `0x17`
3. **Carry Flag (C):** A carry is generated from the 8th bit. $\mathbf{C = 1}$.
4. **Half-carry Flag (H):** Looking at bit 3 (the 4th bit): $0101 + 0010 = 0111$. No carry is generated from bit 3 to bit 4. $\mathbf{H = 0}$.
5. **Zero Flag (Z):** The result (`0x17`) is not zero. $\mathbf{Z = 0}$.
6. **Negative Flag (N):** The MSB (bit 7) of the 8-bit result (`00010111`) is 0. $\mathbf{N = 0}$.

2.4 Clock Circuits Reset and Execution Time

The AVR execution speed is directly proportional to the system clock frequency configured via Fuse Bits. The AVR architecture executes most instructions in a single machine cycle.

Methodology:

Calculating Instruction Execution Time

1. Identify the system oscillator frequency (f_{osc}).
2. Calculate the Machine Cycle Time (T_{cycle}) using:

$$T_{cycle} = \frac{1}{f_{osc}}$$

3. Multiply the machine cycle time by the number of clock cycles required by the specific instruction.

Worked Example:

Instruction Timing Calculation An ATmega32 is configured via its fuse bits to use an external crystal oscillator running at 16 MHz. Calculate the time taken to execute an `RCALL` instruction, which requires 3 clock cycles.
Solution:

1. System Frequency: $f_{osc} = 16,000,000$ Hz.

2. Machine Cycle Time:
$$T_{cycle} = \frac{1}{16,000,000} = 62.5 \text{ ns}$$
3. Execution Time:
$$\text{Time} = 3 \times 62.5 \text{ ns} = 187.5 \text{ ns}$$

2.5 Timers and Counters Operation Modes

The ATmega32 includes two 8-bit timers (Timer0, Timer2) and one 16-bit timer (Timer1). They can operate in Normal, Clear Timer on Compare Match (CTC), and Fast Pulse Width Modulation (PWM) modes.

Methodology:

Timer Preload Calculation for Delay (Normal Mode) To calculate the starting value (preload) required to generate a specific time delay:

1. Determine the target delay (T_{delay}), clock frequency (f_{osc}), and prescaler (N).
2. Calculate the required number of timer ticks (Count):
$$\text{Count} = \frac{T_{delay} \times f_{osc}}{N}$$
3. Calculate the Preload Value based on the timer resolution ($n = 8$ or 16 bits):
$$\text{Preload Value} = 2^n - \text{Count}$$
4. Convert the preload value to hexadecimal to load into the `TCNTx` register.

Worked Example:

8-bit Timer Preload Calculation Calculate the `TCNT0` preload value required to generate a 1 ms delay using Timer0. The ATmega32 operates at an 8 MHz clock frequency, and a prescaler of 64 is selected.
Solution:

1. Given: $T_{delay} = 1$ ms $= 0.001$ s, $f_{osc} = 8,000,000$ Hz, $N = 64$.
2. Calculate the required ticks:
$$\text{Count} = \frac{0.001 \times 8,000,000}{64} = \frac{8000}{64} = 125$$
3. Timer0 is an 8-bit timer, so $n = 8$ and $2^8 = 256$.
4. Calculate the Preload Value:
$$\text{Preload} = 256 - 125 = 131$$
5. Convert to Hexadecimal: $131_{10} = 0\text{x}83$.
6. The value to load into `TCNT0` is `0x83`.

Methodology:

Square Wave Frequency Calculation in CTC Mode In CTC mode, the timer counts up to the value stored in the Output Compare Register (`OCRnx`) and then resets. If configured to toggle the `OCnx` pin on compare match, a square wave is generated.

1. Identify the oscillator frequency (f_{osc}), prescaler (N), and the `OCRnx` register value.
2. Use the formula to calculate the output frequency:

$$f_{OCnx} = \frac{f_{osc}}{2 \cdot N \cdot (1 + \text{OCRnx})}$$

Worked Example:

CTC Mode Frequency Analysis Calculate the frequency of the square wave generated on the OC0 pin using Timer0 in CTC mode. The clock frequency is 16 MHz, the prescaler is 256, and OCR0 is set to 124.
Solution:

1. Given: $f_{osc} = 16,000,000$ Hz, $N = 256$, OCR0 $= 124$.
2. Apply the CTC frequency formula:

$$f_{OC0} = \frac{16,000,000}{2 \cdot 256 \cdot (1 + 124)}$$

$$f_{OC0} = \frac{16,000,000}{512 \cdot 125} = \frac{16,000,000}{64,000} = 250 \text{ Hz}$$

Methodology:

Fast PWM Frequency and Duty Cycle Calculation In Fast PWM non-inverting mode, the timer counts from 0 to MAX. The output is cleared on compare match and set at BOTTOM.

1. Calculate the PWM frequency for an 8-bit timer:

$$f_{PWM} = \frac{f_{osc}}{N \cdot 256}$$

2. Calculate the duty cycle percentage:

$$\text{Duty Cycle } (\%) = \left(\frac{\text{OCRnx} + 1}{256} \right) \times 100$$

Worked Example:

PWM Signal Generation Properties Timer0 is configured in Fast PWM non-inverting mode on an ATmega32 running at 8 MHz. If the prescaler is 64 and OCR0 is 63, calculate the PWM frequency and the duty cycle.
Solution:

1. **PWM Frequency:**

$$f_{PWM} = \frac{8,000,000}{64 \cdot 256} = \frac{8,000,000}{16,384} \approx 488.28 \text{ Hz}$$

2. **Duty Cycle:**

$$\text{Duty Cycle} = \left(\frac{63 + 1}{256} \right) \times 100 = \frac{64}{256} \times 100 = 25\%$$

2.6 On-chip ADC Features and Hardware Considerations

The ATmega32 features a 10-bit successive approximation Analog-to-Digital Converter (ADC). It supports multiple multiplexed input channels and requires careful hardware considerations, such as filtering the AVCC supply, to minimize noise.

Methodology:

Calculating ADC Digital Output Value The 10-bit ADC converts an analog input voltage (V_{in}) into a digital number between 0 and 1023.

1. Identify the analog input voltage (V_{in}) and the reference voltage (V_{ref}).
2. Calculate the digital output using the formula:

$$\text{ADC Value} = \text{round}\left(\frac{V_{in} \cdot 1024}{V_{ref}}\right)$$

3. Ensure V_{in} does not exceed V_{ref}, otherwise the output saturates at 1023.

Worked Example:

ADC Value Computation Compute the 10-bit digital result for an analog input voltage of 3.2 V applied to an ADC channel, given that the reference voltage (V_{ref}) is 5.0 V.
Solution:

1. Given: $V_{in} = 3.2$ V, $V_{ref} = 5.0$ V.
2. Apply the ADC conversion formula:

$$\text{ADC Value} = \frac{3.2 \times 1024}{5.0} = \frac{3276.8}{5} = 655.36$$

3. Rounding to the nearest integer, the ADC output is **655** (or `0x28F` in hex).

Methodology:

Determining ADC Conversion Time The ADC requires a clock frequency between 50 kHz and 200 kHz for maximum resolution. The conversion time depends on this clock and the phase of the ADC operation.

1. Identify the system clock (f_{osc}) and the ADC prescaler division factor (N_{ADC}).
2. Calculate the ADC clock frequency:

$$f_{ADC} = \frac{f_{osc}}{N_{ADC}}$$

3. Calculate the ADC clock period: $T_{ADC} = \frac{1}{f_{ADC}}$.
4. Multiply T_{ADC} by the number of cycles needed. A normal conversion takes 13 ADC clock cycles (the first conversion takes 25 cycles).

Worked Example:

ADC Timing Calculation Calculate the time taken for a normal ADC conversion in an ATmega32 operating at 16 MHz if the ADC prescaler is set to 128.
Solution:

1. Calculate ADC Clock Frequency:

$$f_{ADC} = \frac{16,000,000}{128} = 125,000 \text{ Hz} = 125 \text{ kHz}$$

2. (Note: 125 kHz is well within the optimal 50-200 kHz range).
3. Calculate ADC Clock Period:

$$T_{ADC} = \frac{1}{125,000} = 8\,\mu\text{s}$$

4. Calculate Normal Conversion Time:

$$\text{Time} = 13 \times T_{ADC} = 13 \times 8\,\mu\text{s} = 104\,\mu\text{s}$$

CHAPTER 3

AVR Programming in C

This chapter focuses on the practical aspects of programming AVR microcontrollers using the C language. It covers essential techniques such as bitwise operations for port manipulation, time delay generation, control statements for logic implementation, and fundamental data conversion algorithms required for embedded systems.

3.1 Bitwise Operations for Register Manipulation

In embedded C programming for AVR microcontrollers, modifying specific bits of Special Function Registers (SFRs) without affecting other bits is a critical operation.

Methodology:

Setting Clearing and Toggling Bits Use the following bitwise logical operators to manipulate specific bits in a register (e.g. PORTX DDRX or PINX):

1. **Setting a Bit (Making it 1):** Use the bitwise OR operator (|).

$$\text{Register} = \text{Register} \mid (1 \ll \text{BitPosition});$$

2. **Clearing a Bit (Making it 0):** Use the bitwise AND operator (&) with the bitwise NOT operator (~).

$$\text{Register} = \text{Register} \ \& \ \sim (1 \ll \text{BitPosition});$$

3. **Toggling a Bit (Inverting it):** Use the bitwise XOR operator (^).

$$\text{Register} = \text{Register} \ \hat{} \ (1 \ll \text{BitPosition});$$

4. **Checking a Bit Status:** Use bitwise AND to mask all other bits.

$$\text{Status} = \text{Register} \ \& \ (1 \ll \text{BitPosition});$$

Worked Example:

Bitwise Port Manipulation Write C statements to perform the following operations on PORTB: 1. Set bit 3 to 1. 2. Clear bit 5 to 0. 3. Toggle bit 7.
Solution:

1. **Set bit 3:**

$$\text{PORTB} = \text{PORTB} \mid (1 \ll 3);$$

   Alternatively in C code: `PORTB |= 0x08;`

2. **Clear bit 5:**

$$\text{PORTB} = \text{PORTB} \ \& \ \sim (1 \ll 5);$$

   Alternatively in C code: `PORTB &= ~0x20;`

3. **Toggle bit 7:**

$$\text{PORTB} = \text{PORTB} \hat{} (1 \ll 7);$$

Alternatively in C code: `PORTB ^= 0x80;`

3.2 Digital I/O Port Programming

AVR ports can be configured as inputs or outputs using the DDR (Data Direction Register). Data is written to the PORT register and read from the PIN register.

Methodology:

Configuring and Accessing IO Ports

1. **Configure Port as Output:** Write 1 to the specific bits of the DDRX register. To make the entire port an output write `0xFF`.
2. **Configure Port as Input:** Write 0 to the specific bits of the DDRX register. To make the entire port an input write `0x00`.
3. **Enable Internal Pull-up Resistor:** Configure the pin as input then write 1 to the corresponding bit in the PORTX register.
4. **Read from Input Pin:** Read the value from the PINX register.
5. **Write to Output Pin:** Write the value to the PORTX register.

Worked Example:

Switch and LED Interface Write an AVR C program to monitor the status of a switch connected to pin PB2. If the switch is pressed (reads LOW) turn ON an LED connected to pin PC5 (active HIGH). Otherwise turn OFF the LED.
Solution:

1. **Initialization:**
   - Make PB2 an input: `DDRB &= ~(1 << 2);`
   - Enable pull-up on PB2: `PORTB |= (1 << 2);`
   - Make PC5 an output: `DDRC |= (1 << 5);`
2. **Logic implementation (Infinite Loop):**
   - Read PINB bit 2. If it is 0 the switch is pressed.
   - If pressed set PORTC bit 5 to 1.
   - If not pressed clear PORTC bit 5 to 0.

C Code snippet:

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

int main(void) {
    DDRB &= ~(1<<2); // PB2 as input
    PORTB |= (1<<2); // Enable pull-up for PB2
    DDRC |= (1<<5);  // PC5 as output

    while(1) {
        if((PINB & (1<<2)) == 0) {
```

```
            PORTC |= (1<<5);  // Turn ON LED
        } else {
            PORTC &= ~(1<<5); // Turn OFF LED
        }
    }
    return 0;
}
```

3.3 Generating Time Delays

Delays are frequently required for debouncing switches creating visual effects with LEDs or meeting timing requirements of external peripherals.

Methodology:

Using the Delay Header File AVR GCC provides built-in functions for accurate delay generation.

1. **Define CPU Clock Frequency:** Define `F_CPU` indicating the microcontroller clock frequency before including the delay library.

   #define F_CPU 8000000UL // Example for 8 MHz

2. **Include Library:** Include the delay header file.

   #include <util/delay.h>

3. **Millisecond Delay:** Use `_delay_ms(double ms)` for delays in milliseconds.

4. **Microsecond Delay:** Use `_delay_us(double us)` for delays in microseconds.

Worked Example:

Blinking an LED Write an AVR C program to toggle all pins of PORTD continuously with a 500 ms delay between each toggle. Assume an 8 MHz clock frequency.

Solution:

1. Set the entire PORTD as output by loading `0xFF` into DDRD.
2. Use a `while(1)` loop for continuous operation.
3. Inside the loop toggle the PORTD register using the XOR operator.
4. Call `_delay_ms(500)` to generate the required delay.

C Code snippet:

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

int main(void) {
    DDRD = 0xFF; // Configure PORTD as output
    while(1) {
        PORTD ^= 0xFF;  // Toggle all bits of PORTD
        _delay_ms(500); // Wait for 500 milliseconds
    }
    return 0;
}
```

3.4 Data Conversion Algorithms

Embedded systems frequently require converting data from one format to another for display or transmission purposes. Common conversions include Unpacked BCD to ASCII and Packed BCD to ASCII.

Methodology:

BCD to ASCII Conversion To convert a Packed BCD byte (e.g. `0x45`) to two ASCII characters ('4' and '5'):

1. **Extract the Lower Nibble:** Mask the upper 4 bits using bitwise AND with `0x0F`.

$$\text{LowerNibble} = \text{Data} \ \& \ 0x0F;$$

2. **Convert Lower Nibble to ASCII:** Add `0x30` (or `'0'`) to the result.

$$\text{ASCII}_L = \text{LowerNibble} + 0x30;$$

3. **Extract the Upper Nibble:** Mask the lower 4 bits using bitwise AND with `0xF0` and shift right by 4 positions.

$$\text{UpperNibble} = (\text{Data} \ \& \ 0xF0) \gg 4;$$

4. **Convert Upper Nibble to ASCII:** Add `0x30` to the shifted result.

$$\text{ASCII}_U = \text{UpperNibble} + 0x30;$$

Worked Example:

Converting Packed BCD to ASCII Write an AVR C program to read a packed BCD number from PORTB convert it to two ASCII characters and send the upper digit to PORTC and the lower digit to PORTD.
Solution:

1. Configure PORTB as input and PORTC PORTD as outputs.
2. Read the byte from PINB.
3. Apply the BCD to ASCII conversion method.
4. Write the ASCII value of the upper digit to PORTC and the lower digit to PORTD.

C Code snippet:

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

int main(void) {
    unsigned char bcd_data, upper_ascii, lower_ascii;

    DDRB = 0x00; // PORTB as input
    DDRC = 0xFF; // PORTC as output
    DDRD = 0xFF; // PORTD as output

    while(1) {
        bcd_data = PINB; // Read packed BCD from PORTB

        // Extract and convert lower nibble
        lower_ascii = (bcd_data & 0x0F) + 0x30;

        // Extract and convert upper nibble
        upper_ascii = ((bcd_data & 0xF0) >> 4) + 0x30;
```

```
        // Output results
        PORTC = upper_ascii;
        PORTD = lower_ascii;
    }
    return 0;
}
```

3.5 Accessing EEPROM in C

The AVR microcontroller includes internal EEPROM memory to store non-volatile data. Reading and writing EEPROM involves configuring specific control registers sequentially.

Methodology:

EEPROM Read and Write Operations

1. **Writing to EEPROM:**
   (a) Wait until the EEPROM Write Enable (EEWE) bit in EECR is zero.
   (b) Write the target EEPROM address to the EEAR register.
   (c) Write the data byte to the EEDR register.
   (d) Set the EEPROM Master Write Enable (EEMWE) bit in EECR to 1.
   (e) Within 4 clock cycles set the EEPROM Write Enable (EEWE) bit to 1.
2. **Reading from EEPROM:**
   (a) Wait until the EEWE bit in EECR is zero.
   (b) Write the target EEPROM address to the EEAR register.
   (c) Set the EEPROM Read Enable (EERE) bit in EECR to 1.
   (d) Read the data byte from the EEDR register.

Worked Example:

EEPROM Write Sequence Write a C function to write the byte `0x55` to EEPROM address `0x005F`.
Solution:

1. Implement a `while` loop to poll the `EEWE` bit of the `EECR` register.
2. Load the address `0x005F` into `EEAR`.
3. Load the data `0x55` into `EEDR`.
4. Set `EEMWE` to 1.
5. Set `EEWE` to 1.

C Code snippet:

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

void EEPROM_write(unsigned int uiAddress, unsigned char ucData) {
    // Wait for completion of previous write
    while(EECR & (1<<EEWE));

    // Set up address and data registers
    EEAR = uiAddress;
```

```
    EEDR = ucData;

    // Write logical one to EEMWE
    EECR |= (1<<EEMWE);

    // Start EEPROM write by setting EEWE
    EECR |= (1<<EEWE);
}

int main(void) {
    EEPROM_write(0x005F, 0x55);
    while(1);
    return 0;
}
```

CHAPTER 4

AVR Interfacing

This chapter covers the computational and algorithmic aspects of interfacing various peripherals with the AVR microcontroller. The focus is on register configuration sequences, code generation for displays, scanning algorithms for keypads, and mathematical calculations for Analog-to-Digital Conversion (ADC) and motor control.

4.1 LED and Switch Interfacing

Interfacing basic Input/Output devices requires configuring the Data Direction Register (`DDRx`), Port Data Register (`PORTx`), and Port Input Pins Register (`PINx`).

Methodology:

IO Port Configuration Algorithm

1. **Direction Control:** To make a pin an output, write 1 to the corresponding bit in `DDRx`. To make it an input, write 0.
2. **Outputting Data:** For output pins, write the desired logic level (0 or 1) to `PORTx`.
3. **Internal Pull-up:** For input pins, write 1 to the corresponding bit in `PORTx` to enable the internal pull-up resistor.
4. **Reading Input:** Read the logical state of input pins from the `PINx` register.

Worked Example:

Switch and LED Logic A push button is connected to pin PA0 and an LED is connected to pin PB0. Write the logic sequence to turn on the LED when the button is pressed (assuming active-low button and active-high LED).

Step 1: Configure Port A pin 0 as input with pull-up

```
DDRA = DDRA & ~(1 « PA0); /* PA0 as input */
PORTA = PORTA | (1 « PA0); /* Enable pull-up on PA0 */
```

Step 2: Configure Port B pin 0 as output

```
DDRB = DDRB | (1 « PB0); /* PB0 as output */
```

Step 3: Read switch and control LED

```
if ((PINA & (1 « PA0)) == 0) { /* Button pressed (Active Low) */
  PORTB = PORTB | (1 « PB0); /* Turn ON LED */
} else {
  PORTB = PORTB & ~(1 « PB0); /* Turn OFF LED */
}
```

4.2 Seven Segment Display Interfacing

Seven-segment displays (SSD) require a specific binary code to turn on the required segments (a through g and dp) to display a decimal digit.

Methodology:

Generating Hex Codes for Seven Segment Display

1. **Identify Display Type:** Determine if the display is Common Cathode (CC) or Common Anode (CA).
2. **Segment Mapping:** Map the microcontroller pins to segments typically ordered as $dp - g - f - e - d - c - b - a$ (from MSB to LSB).
3. **Logic for CC:** To turn ON a segment, send Logic 1. To turn OFF, send Logic 0.
4. **Logic for CA:** To turn ON a segment, send Logic 0. To turn OFF, send Logic 1.
5. **Convert to Hex:** Form the 8-bit binary number and convert it to Hexadecimal.

Worked Example:

Hex Code Calculation for Digit 3 Calculate the hex code to display the digit '3' on a Common Cathode (CC) seven-segment display. Assume the segments are connected to PORTC in the order dp, g, f, e, d, c, b, a (where a is PC0).

Step 1: Identify required segments for '3'

To display '3', segments a, b, c, d, g must be ON. Segments e, f, dp must be OFF.

Step 2: Assign logic levels for Common Cathode

ON = 1, OFF = 0.

- $dp = 0$
- $g = 1$
- $f = 0$
- $e = 0$
- $d = 1$
- $c = 1$
- $b = 1$
- $a = 1$

Step 3: Form binary and convert to Hex

Binary: 01001111_2

Hexadecimal: `0x4F`

Therefore, writing `0x4F` to `PORTC` will display '3'.

4.3 16x2 LCD Interfacing

Interfacing a standard HD44780-based 16x2 LCD in 8-bit mode requires sending specific command and data bytes using control pins: Register Select (RS), Read/Write (RW), and Enable (EN).

Methodology:

LCD Command and Data Write Sequence **Sending a Command:**

1. Send the command byte to the data port.
2. Clear RS (RS = 0) to select the Command Register.
3. Clear RW (RW = 0) for Write operation.
4. Apply a High-to-Low pulse on the EN pin (Set EN, wait, Clear EN).

Sending Data:

1. Send the data byte (ASCII character) to the data port.
2. Set RS (RS = 1) to select the Data Register.
3. Clear RW (RW = 0) for Write operation.
4. Apply a High-to-Low pulse on the EN pin.

Worked Example:

LCD Initialization Algorithm Provide the standard sequence of command codes required to initialize a 16x2 LCD in 8-bit mode.
Step 1: Configure Display Mode
Command `0x38`: Initialize LCD in 8-bit mode, 2 lines, and 5x7 dot matrix.
Step 2: Display Control
Command `0x0C`: Display ON, Cursor OFF. (Alternatively, `0x0E` for Cursor ON).
Step 3: Clear Display
Command `0x01`: Clear the display screen and return the cursor to home position.
Step 4: Entry Mode Set
Command `0x06`: Increment cursor to the right after writing each character.

4.4 Matrix Keypad Interfacing

A matrix keypad organizes switches in a grid of rows and columns, saving microcontroller I/O pins. A 4x4 keypad uses 8 pins instead of 16.

Methodology:

Keypad Scanning Algorithm

1. **Configuration:** Configure row pins as outputs and column pins as inputs with internal pull-ups enabled.
2. **Grounding Rows:** Sequentially ground one row at a time by making its output logic 0, while keeping other rows at logic 1.
3. **Reading Columns:** For each grounded row, read the column pins. If a key is pressed, the corresponding column pin will read logic 0 (due to being shorted to the grounded row).
4. **Key Identification:** Combine the active row and active column indices to uniquely identify the pressed key using a formula or lookup table.

Worked Example:

Detecting a Key Press on a 4x4 Keypad Assume a 4x4 keypad is connected to PORTD. Rows (R0-R3) are outputs on PD0-PD3, and Columns (C0-C3) are inputs on PD4-PD7 with pull-ups. Determine the port states when the key at Row 1, Column 2 is pressed.

Step 1: Output state for Row 1 scanning

To scan Row 1, set PD1 to 0 and PD0, PD2, PD3 to 1. The upper nibble is set to 1 for pull-ups.

`PORTD` = `1111 1101` (Binary) = `0xFD`.

Step 2: Key Press Action

When the key at Row 1, Column 2 is pressed, Column 2 (PD6) is shorted to Row 1 (PD1). Since PD1 is at Logic 0, PD6 is pulled to Logic 0.

Step 3: Reading the Port

The column pins (PD4-PD7) are read. PD4=1, PD5=1, PD6=0, PD7=1.

`PIND` upper nibble = `1011` (Binary) = `0xB`.

The entire `PIND` register will read as `1011 1101` (Binary) = `0xBD`.

The software detects the 0 at bit 6 and bit 1, identifying the intersection (Row 1, Col 2).

4.5 ADC Interfacing

The AVR has a built-in 10-bit Analog-to-Digital Converter (ADC). It converts continuous analog voltages into discrete digital values.

Methodology:

ADC Digital Output Calculation The digital output value from the ADC can be calculated using the formula:

$$\text{ADC}_{value} = \frac{V_{in} \times (2^n - 1)}{V_{ref}}$$

Where:

- V_{in} is the input analog voltage.
- V_{ref} is the reference voltage.
- n is the resolution of the ADC in bits (10 for standard AVRs).
- ADC_{value} is the integer result stored in the ADC data registers (ADCH and ADCL).

Worked Example:

Calculating ADC Output Value An ATmega32 has its ADC configured with a reference voltage $V_{ref} = 5\text{V}$ and 10-bit resolution. Calculate the digital ADC output if the analog input voltage is 3.2V.

Step 1: Identify the given parameters

$V_{in} = 3.2\text{V}$

$V_{ref} = 5.0\text{V}$

$n = 10 \implies 2^{10} - 1 = 1023$

Step 2: Apply the ADC formula

$$\text{ADC}_{value} = \frac{3.2 \times 1023}{5.0}$$

Step 3: Perform the calculation

$$\text{ADC}_{value} = \frac{3273.6}{5} = 654.72$$

Step 4: Determine the final integer value
Since the ADC register stores an integer, the value is rounded or truncated:

$$\text{ADC}_{value} = 654$$

In binary, this is 1010001110_2, which is stored across the `ADCH` and `ADCL` registers.

4.6 Stepper Motor Interfacing

A stepper motor moves in discrete steps by energizing its stator coils in a specific sequence.

Methodology:

Stepper Motor Sequence Generation **Full-Step Sequence (1-Phase ON):** Only one coil is energized at a time.

1. Step 1: Coil A ON (1000_2 = `0x08`)
2. Step 2: Coil B ON (0100_2 = `0x04`)
3. Step 3: Coil C ON (0010_2 = `0x02`)
4. Step 4: Coil D ON (0001_2 = `0x01`)

Full-Step Sequence (2-Phase ON): Two adjacent coils are energized simultaneously for higher torque.

1. Step 1: Coils A and B ON (1100_2 = `0x0C`)
2. Step 2: Coils B and C ON (0110_2 = `0x06`)
3. Step 3: Coils C and D ON (0011_2 = `0x03`)
4. Step 4: Coils D and A ON (1001_2 = `0x09`)

Worked Example:

Stepper Motor RPM Calculation A unipolar stepper motor has a step angle of 1.8°. It is driven in full-step mode (2-Phase ON). If the delay between each step is 10ms, calculate the RPM (Revolutions Per Minute) of the motor.
Step 1: Calculate steps per revolution

$$\text{Steps per revolution} = \frac{360^\circ}{\text{Step Angle}} = \frac{360^\circ}{1.8^\circ} = 200 \text{ steps}$$

Step 2: Calculate time for one revolution

$$\text{Time per step} = 10\text{ms} = 0.01\text{s}$$

$$\text{Time per revolution} = 200 \text{ steps} \times 0.01\text{s/step} = 2\text{s}$$

Step 3: Calculate Revolutions Per Minute

$$\text{RPM} = \frac{60 \text{ seconds}}{\text{Time per revolution in seconds}}$$

$$\text{RPM} = \frac{60}{2} = 30 \text{ RPM}$$

CHAPTER 5

Embedded System Applications

This chapter covers the computational aspects of developing typical embedded system applications. The primary focus is on calculating required register values, signal timings, analog-to-digital conversions, and actuation parameters necessary for hardware interfacing.

5.1 Serial Communication Parameters

Methodology:

UART Baud Rate Generation To calculate the Universal Synchronous and Asynchronous Receiver and Transmitter (USART) Baud Rate Register (UBRR) value in normal asynchronous mode:

1. Identify the system clock frequency f_{osc} and the desired Baud Rate ($BAUD$).
2. Use the standard asynchronous mode formula to find UBRR:

$$UBRR = \frac{f_{osc}}{16 \times BAUD} - 1$$

3. For Double Speed Asynchronous Mode, the multiplier is modified:

$$UBRR = \frac{f_{osc}}{8 \times BAUD} - 1$$

4. Round the calculated UBRR value to the nearest integer.
5. Calculate the actual baud rate generated by substituting the rounded UBRR back:

$$\text{Actual } BAUD = \frac{f_{osc}}{16 \times (UBRR + 1)}$$

6. Compute the percentage error to ensure it is within acceptable limits (typically $< 2\%$ for UART):

$$\text{Error } (\%) = \left(\frac{\text{Actual } BAUD - \text{Desired } BAUD}{\text{Desired } BAUD} \right) \times 100$$

Worked Example:

UBRR Calculation for 9600 Baud Rate Find the UBRR value required to generate a 9600 baud rate with an 8 MHz system clock in normal asynchronous mode. Determine the resulting baud rate error.
Step 1: Identify given parameters. System Clock, $f_{osc} = 8$ MHz $= 8,000,000$ Hz
Desired Baud Rate, $BAUD = 9600$ bps
Step 2: Apply the UBRR formula.

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

Step 3: Perform division and subtraction.

$$16 \times 9600 = 153,600$$

$$UBRR = \frac{8,000,000}{153,600} - 1 = 52.083 - 1 = 51.083$$

Rounding to the nearest integer, we get $UBRR = 51$.
Step 4: Check for error.

$$\text{Actual } BAUD = \frac{8,000,000}{16 \times (51+1)} = \frac{8,000,000}{832} = 9615.38 \text{ bps}$$

$$\text{Error } (\%) = \left(\frac{9615.38 - 9600}{9600}\right) \times 100 \approx 0.16\%$$

The error of 0.16% is well within the acceptable tolerance.

5.2 Timer and Delay Generation

Methodology:

Timer Delay Calculation To calculate the initial value to be loaded into the Timer/Counter register (TCNT) for a specific time delay:

1. Identify the system clock frequency f_{osc}, the Prescaler value (N), and the target delay time (T_{delay}).
2. Calculate the timer clock frequency f_{timer} and timer period T_{timer}:

$$f_{timer} = \frac{f_{osc}}{N}$$

$$T_{timer} = \frac{1}{f_{timer}} = \frac{N}{f_{osc}}$$

3. Determine the total number of timer ticks required for the desired delay:

$$\text{Ticks} = \frac{T_{delay}}{T_{timer}} = \frac{T_{delay} \times f_{osc}}{N}$$

4. Calculate the starting value to load into the timer register ($TCNT_{initial}$). For an n-bit timer (e.g. 8-bit timer where MAX = 255 or 16-bit timer where MAX = 65535):

$$TCNT_{initial} = (\text{MAX} + 1) - \text{Ticks}$$

Worked Example:

TCNT0 Value for 10 ms Delay Calculate the initial value to load into an 8-bit timer (TCNT0) to generate an exact 10 ms delay. The system clock is 8 MHz and the chosen prescaler is 1024.
Step 1: Identify given parameters. Timer is 8-bit, so MAX = 255, and MAX + 1 = 256
$f_{osc} = 8,000,000$ Hz
Prescaler $N = 1024$
Delay $T_{delay} = 10$ ms $= 0.01$ s
Step 2: Calculate the timer clock period T_{timer}.

$$T_{timer} = \frac{1024}{8,000,000} = 0.000128 \text{ s} = 128\mu\text{s}$$

Step 3: Calculate the number of ticks required.

$$\text{Ticks} = \frac{10 \text{ ms}}{128\mu\text{s}} = \frac{10 \times 10^{-3}}{128 \times 10^{-6}} = 78.125$$

Rounding to the nearest whole tick, Ticks = 78.
Step 4: Calculate the initial value for TCNT0.

$$TCNT_{initial} = 256 - 78 = 178$$

The decimal value 178 (which is `0xB2` in hexadecimal) must be loaded into the TCNT0 register.

5.3 Analog to Digital Conversion

Methodology:

ADC Digital Output Calculation To convert an analog input voltage to a digital representation using an n-bit ADC:

1. Identify the resolution of the ADC (n bits). The maximum digital value is $2^n - 1$.
2. Identify the reference voltage V_{ref} and the analog input voltage V_{in}.
3. Calculate the step size (voltage resolution per bit):

$$\text{Step Size} = \frac{V_{ref}}{2^n}$$

4. Calculate the decimal digital output value by dividing the input voltage by the step size:

$$\text{Digital Output} = \frac{V_{in}}{\text{Step Size}} = \frac{V_{in} \times 2^n}{V_{ref}}$$

5. The final digital output is an integer. Drop the fractional part (truncate) or round to the nearest whole number to get the final register value.

Worked Example:

ADC Value for 2.5V Input An embedded system utilizes a 10-bit ADC with a reference voltage of 5V. Calculate the expected digital output when a 2.5V analog signal is applied.
Step 1: Identify parameters. ADC Resolution, $n = 10$ bits ($2^{10} = 1024$)
Reference Voltage, $V_{ref} = 5\text{V}$
Input Voltage, $V_{in} = 2.5\text{V}$
Step 2: Calculate the step size.

$$\text{Step Size} = \frac{5\text{V}}{1024} \approx 4.88 \text{ mV/bit}$$

Step 3: Calculate the digital output value.

$$\text{Digital Output} = \frac{2.5\text{V}}{4.88 \times 10^{-3} \text{ V}} = \frac{2.5 \times 1024}{5} = \frac{2560}{5} = 512$$

The 10-bit ADC result register will contain the decimal value 512 (which is `0x200` in hexadecimal).

Methodology:

LM35 Temperature Sensor Calculation The LM35 is an analog temperature sensor that outputs a voltage linearly proportional to the Celsius temperature with a scale factor of 10mV/°C.

1. Read the digital ADC value and calculate the corresponding analog voltage V_{in}:

$$V_{in} = \frac{\text{ADC Value} \times V_{ref}}{2^n}$$

2. Convert the voltage to temperature using the 0.01 V/°C scale factor:

$$\text{Temperature (°C)} = \frac{V_{in}}{0.01} = V_{in} \times 100$$

3. Combining the formulas yields the direct conversion equation:

$$\text{Temperature (°C)} = \left(\frac{\text{ADC Value} \times V_{ref}}{2^n}\right) \times 100$$

Worked Example:

Temperature Calculation from ADC Value A microcontroller with a 10-bit ADC and a 5V reference reads a value of 61 from an LM35 sensor. Calculate the measured temperature.

Step 1: Apply the direct conversion equation. $n = 10$ bits ($2^{10} = 1024$), $V_{ref} = 5\text{V}$, ADC Value = 61

$$\text{Temperature (°C)} = \left(\frac{61 \times 5}{1024}\right) \times 100$$

Step 2: Calculate the intermediate voltage.

$$V_{in} = \frac{305}{1024} \approx 0.2978 \text{ V} = 297.8 \text{ mV}$$

Step 3: Convert voltage to temperature.

$$\text{Temperature} = 0.2978 \times 100 = 29.78°\text{C}$$

The calculated temperature is 29.78°C.

5.4 Pulse Width Modulation (PWM)

Methodology:

PWM Duty Cycle and Register Calculation In Fast PWM mode, the Output Compare Register (OCR) dictates the duty cycle of the generated signal.

1. Identify the maximum timer value (TOP). For an 8-bit timer, TOP = 255.

2. For a non-inverting Fast PWM output, calculate the Duty Cycle percentage:

$$\text{Duty Cycle (\%)} = \left(\frac{OCR + 1}{\text{TOP} + 1}\right) \times 100$$

3. To find the required OCR value for a specific duty cycle, rearrange the equation:

$$OCR = \left(\frac{\text{Duty Cycle (\%)} \times (\text{TOP} + 1)}{100}\right) - 1$$

4. The frequency of the resulting PWM signal (f_{PWM}) is calculated as:

$$f_{PWM} = \frac{f_{osc}}{N \times (\text{TOP} + 1)}$$

where N is the timer prescaler.

Worked Example:

OCR Value for 60 Percent Duty Cycle Determine the OCR value needed to generate a 60% duty cycle PWM signal using an 8-bit timer in non-inverting Fast PWM mode.
Step 1: Identify parameters. Timer is 8-bit, so TOP = 255
Desired Duty Cycle = 60%
Step 2: Substitute into the OCR formula.

$$OCR = \left(\frac{60 \times (255 + 1)}{100}\right) - 1$$

Step 3: Calculate the register value.

$$OCR = \left(\frac{60 \times 256}{100}\right) - 1$$

$$OCR = 153.6 - 1 = 152.6$$

Rounding to the nearest integer, we assign $OCR = 153$.
Step 4: Verify actual duty cycle.

$$\text{Actual Duty Cycle} = \left(\frac{153 + 1}{256}\right) \times 100 = \left(\frac{154}{256}\right) \times 100 = 60.156\%$$

5.5 Stepper Motor Control

Methodology:

Stepper Motor Step Angle and RPM Calculation To calculate rotational parameters of a stepper motor based on step sequences and switching delays:

1. **Step Angle (θ)**: The angle the motor shaft rotates for each discrete step pulse. It can be found if the number of rotor teeth (Z_r) and stator phases (m) are known:

$$\text{Step Angle} = \frac{360^\circ}{m \times Z_r}$$

2. **Steps per Revolution (S)**: The total number of pulses needed for a full 360-degree mechanical rotation:

$$S = \frac{360^\circ}{\text{Step Angle}}$$

3. **Speed in RPM (N)**: The rotational speed. First, find Pulses Per Second (PPS) using the switching delay T_{delay} between steps in seconds:

$$\text{PPS} = \frac{1}{T_{delay}}$$

Then, calculate revolutions per minute:

$$N = \frac{\text{PPS} \times 60}{S}$$

Worked Example:

Stepper Motor RPM Calculation A stepper motor has a step angle of 1.8°. A microcontroller drives the motor by applying a 5 ms delay between successive step pulses. Calculate the resulting motor speed in RPM.

Step 1: Calculate total steps per revolution (S).

$$S = \frac{360^\circ}{1.8^\circ} = 200 \text{ steps/revolution}$$

Step 2: Calculate the pulses per second (PPS). Given delay $T_{delay} = 5$ ms $= 0.005$ s.

$$\text{PPS} = \frac{1}{0.005} = 200 \text{ pulses/second}$$

Step 3: Calculate the motor speed in RPM.

$$N = \frac{\text{PPS} \times 60}{S}$$

$$N = \frac{200 \times 60}{200} = 60 \text{ RPM}$$

The stepper motor rotates at a speed of 60 Revolutions Per Minute.

Formulae

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book.

Unit 1: System Basics and Power Management

- **Clock Period (T_{clk}):**
$$T_{clk} = \frac{1}{f_{clk}}$$
- **Machine Cycle Time (T_{MC}):**
$$T_{MC} = C_{MC} \times T_{clk}$$
Where C_{MC} is the number of clock cycles per machine cycle.
- **Execution Time (T_{exec}):**
$$T_{exec} = M \times T_{MC}$$
Where M is the number of machine cycles required to execute a set of instructions.
- **Memory Capacity:**
$$\text{Memory Capacity} = 2^n \times \text{Word Size}$$
Where n is the number of address lines.
- **Average Current Consumption (I_{avg}):**
$$I_{avg} = \frac{(I_{active} \times T_{active}) + (I_{sleep} \times T_{sleep})}{T_{total}}$$
Useful for calculating power consumption in systems with sleep modes.
- **Effective Battery Capacity:**
$$\text{Effective Capacity} = \text{Nominal Capacity} \times D$$
Where D is the derating factor.
- **Battery Life:**
$$\text{Battery Life (hours)} = \frac{\text{Effective Capacity (mAh)}}{I_{avg}\ \text{(mA)}}$$
- **Component Selection (Weighted Decision Matrix):**
$$\text{Weighted Score}_{ij} = S_{ij} \times W_j$$
$$\text{Total Score}_i = \sum_j (S_{ij} \times W_j)$$
Where S_{ij} is the score of component i for criterion j, and W_j is the weight of criterion j.

Unit 2: Computer Arithmetic and Memory Management

- **Signed Overflow Flag (S):**

$$S = N \oplus V$$

  Where N is the negative flag and V is the overflow flag.

- **Memory Map End Address:**

$$\text{End Address} = \text{Start Address} + \text{Size in Bytes} - 1$$

Unit 4: Analog to Digital Conversion (ADC)

- **ADC Step Size (Resolution):**

$$\text{Step Size} = \frac{V_{ref}}{2^n}$$

  Where V_{ref} is the reference voltage and n is the number of ADC bits.

- **Analog Input Voltage (V_{in}) from ADC Value:**

$$V_{in} = \frac{\text{ADC Value} \times V_{ref}}{2^n}$$

- **ADC Digital Output Value:**

$$\text{Digital Output} = \frac{V_{in}}{\text{Step Size}} = \frac{V_{in} \times 2^n}{V_{ref}}$$

- **Temperature Calculation (e.g., LM35):**

$$\text{Temperature (°C)} = \frac{V_{in}}{0.01} = V_{in} \times 100$$

  Alternatively, using the ADC value directly:

$$\text{Temperature (°C)} = \left(\frac{\text{ADC Value} \times V_{ref}}{2^n} \right) \times 100$$

Unit 5: Timers, PWM, Stepper Motors, and UART

Timers and Delays

- **Timer Clock Frequency (f_{timer}):**

$$f_{timer} = \frac{f_{osc}}{N}$$

  Where f_{osc} is the oscillator frequency and N is the prescaler value.

- **Timer Clock Period (T_{timer}):**

$$T_{timer} = \frac{1}{f_{timer}} = \frac{N}{f_{osc}}$$

- **Timer Ticks for a Given Delay:**

$$\text{Ticks} = \frac{T_{delay}}{T_{timer}} = \frac{T_{delay} \times f_{osc}}{N}$$

- **Timer Initial Value ($TCNT_{initial}$):**

$$TCNT_{initial} = (\text{MAX} + 1) - \text{Ticks}$$

  Where MAX is the maximum counter value (e.g., 255 for 8-bit, 65535 for 16-bit).

Pulse Width Modulation (PWM)

- **PWM Frequency (f_{PWM}):**

$$f_{PWM} = \frac{f_{osc}}{N \times (\text{TOP} + 1)}$$

- **PWM Duty Cycle (%):**

$$\text{Duty Cycle (\%)} = \left(\frac{OCR + 1}{\text{TOP} + 1}\right) \times 100$$

  Where OCR is the Output Compare Register value.

- **OCR Value for Desired Duty Cycle:**

$$OCR = \left(\frac{\text{Duty Cycle (\%)} \times (\text{TOP} + 1)}{100}\right) - 1$$

Stepper Motor Control

- **Step Angle:**

$$\text{Step Angle} = \frac{360^\circ}{m \times Z_r}$$

  Where m is the number of stator phases and Z_r is the number of rotor teeth.

- **Steps per Revolution (S):**

$$S = \frac{360^\circ}{\text{Step Angle}}$$

- **Pulses Per Second (PPS):**

$$\text{PPS} = \frac{1}{T_{step\_delay}}$$

- **Motor Speed (RPM):**

$$\text{RPM} = \frac{\text{PPS} \times 60}{S}$$

UART Communication

- **UART Baud Rate Register (UBRR) - Normal Mode:**

$$UBRR = \frac{f_{osc}}{16 \times BAUD} - 1$$

- **UART Baud Rate Register (UBRR) - Double Speed Mode:**

$$UBRR = \frac{f_{osc}}{8 \times BAUD} - 1$$

- **Actual Baud Rate (Normal Mode):**

$$\text{Actual } BAUD = \frac{f_{osc}}{16 \times (UBRR + 1)}$$

- **Baud Rate Error (%):**

$$\text{Error (\%)} = \left(\frac{\text{Actual } BAUD - \text{Desired } BAUD}{\text{Desired } BAUD}\right) \times 100$$