Helwan University
Faculty of Engineering – Helwan
Department of Electronics and Communication

Information Technology Institute (ITI) — Embedded Systems AVR (162 hrs.)

Summer Training Report

Student: AlHussien Mustafa Hassan

Year & Section: 3rd Year – Section 2

Supervisor: Dr. Ehab Abd Alwahab

Training Duration: 13 August 2025 – 17 September 2025

Academic Year: 2024 / 2025

Abstract

This report meticulously documents the comprehensive summer training curriculum focused on **Embedded Systems (AVR)** and advanced **C Programming**.
The core emphasis was placed on mastering C fundamentals, microcontroller architecture (ATmega32), essential peripherals (Timers, Interrupts, DIO, LCD, Keypad), and critical serial communication protocols (**UART**, **SPI**, **I^{2}C**).
The theoretical knowledge was rigorously consolidated through two detailed, multi-faceted engineering projects: the **Clinic Management System**, which demonstrated robust modular software design and input validation; and the **Dual Microcontroller Smart Home Automation System**, which provided a complex, real-world application showcasing inter-controller communication via SPI, secure configuration storage using I^{2}C EEPROM, remote control via UART/Bluetooth, and autonomous environmental management utilizing PWM.
This documentation serves as evidence of proficiency in designing and implementing reliable, high-integrity embedded solutions.

Contents

List of Figures

1 Introduction

Embedded systems play a crucial role in today's technological landscape, seamlessly integrating into devices that surround us in our daily lives. At the heart of many embedded systems lies the Advanced Virtual RISC (Reduced Instruction Set Computing) microcontroller (AVR). This report delves into the fundamental aspects of AVR programming, emphasizing the mas- tery of C programming essentials as a gateway to harnessing the power of AVR.

The curriculum encompasses a wide array of technical disciplines, ranging from fundamental pro- gramming constructs, flow control mechanisms, and data structures (arrays, pointers) to intricate elements of hardware control:

- **Microcontroller Fundamentals:** Mastery of core peripherals including Timers, Interrupts, DIO control, LCD interfacing, and Keypad input processing.
- **Communication Protocols:** Detailed implementation of critical serial interfaces including **UART**, **SPI**, and **I2C**.
- **Engineering Practices:** Integration of essential industry disciplines such as **Software Testing**, understanding **Layered Architecture**, and utilizing **Embedded Systems Tooling**.
- **Industrial Context:** Exploration of domain-specific technologies, notably **Automotive Bus Technology**.

This report serves as a comprehensive guide, covering each facet with precision and clarity, demonstrating not only the theoretical underpinnings but also their practical application through complex projects. The capabilities presented here affirm the readiness to engage in the design and development of reliable, high-integrity embedded solutions.

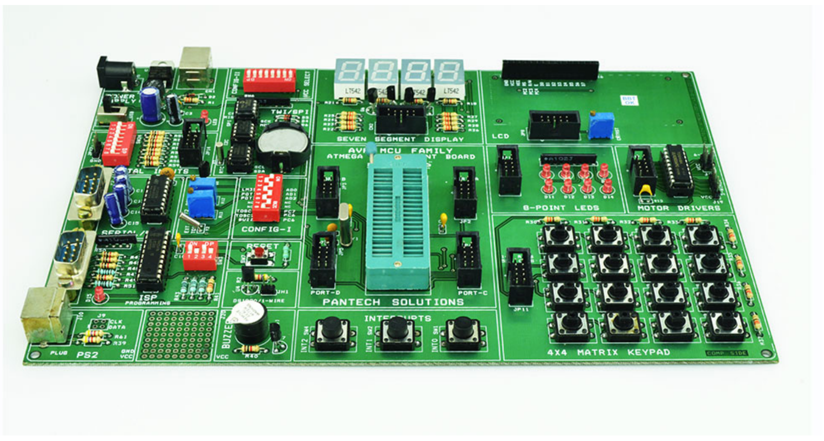


Figure 1: AVR development board.

2 Summary

This report encapsulates an intensive Embedded Systems internship, framed by the mastery of C programming fundamentals and advanced AVR microcontroller concepts. The curriculum provides a cohesive pathway, moving beyond theoretical knowledge to practical, high-integrity system implementation.

The core of the report details the fundamental constructs of C programming, microcontroller interfacing (DIO, Timers, Interrupts), and essential serial communication protocols (UART, SPI, I2C). Significantly, this knowledge base is fortified by the integration of crucial industry best practices:

- **Software Quality:** Introduction to rigorous **Software Testing** methodologies to ensure code reliability and robustness.
- **Industrial Context:** Examination of foundational domain-specific technologies such as **Automotive Bus Technology** and the principles of layered architecture.
- **Development Tooling:** Proficiency gained in utilizing the specialized toolchain necessary for cross-compilation and debugging within the embedded domain.

The journey culminates in two detailed engineering projects—a Clinic Management System and a Dual Microcontroller Smart Home Automation System—which collectively demonstrate a robust toolkit for designing sophisticated, efficient, and reliable embedded solutions tailored to meet the stringent demands of modern electronic applications.

3 C Programming Basics

3.1 Introduction to C Programming

At the heart of the AVR programming journey lies the foundational language of C. Renowned for its efficiency and versatility, C is a procedural programming language that serves as the bedrock for AVR development. Its syntax and principles influence a myriad of modern languages, making it a linchpin in the world of programming.

3.2 C-Based Language

C's influence extends far beyond its own syntax. As a progenitor of C-based languages such as C++, C#, and Objective-C, understanding C lays a robust foundation for navigating a broad spectrum of programming paradigms. This subsection explores the lineage and shared principles that form the basis of these languages.

3.3 Key Applications of C

Delving into C programming isn't merely an academic exercise; it's a practical journey with real-world implications. C is used extensively in developing operating systems, embedded systems (such as those based on AVR microcontrollers), game engines, and system software. This universality underscores C's significance in both low-level and high-level programming environments.

3.4 Comments in C

In the realm of C programming, comments serve as essential elements for enhancing code clarity and providing insights into the logic behind implementation. Comments are non-executable lines intended for human readers. In C, there are two forms of comments:

- **Single-line comments:** Introduced by `//`, useful for short annotations.
- **Multi-line comments:** Enclosed between `/*` and `*/`, suitable for longer explanations or for temporarily disabling blocks of code.

The judicious use of comments not only clarifies code segments but also facilitates collaboration among developers working on the same codebase.

3.5 Variables in C

Variables are dynamic containers that store and manage data during program execution. They are defined by specifying a data type followed by an identifier. Common data types include `int` (integer), `float` (floating-point), and `char` (character). For example:

```
int num = 10;
```

This statement declares an integer variable named `num` and initializes it with the value 10. Proper variable usage enables flexible, reusable, and dynamic program logic.

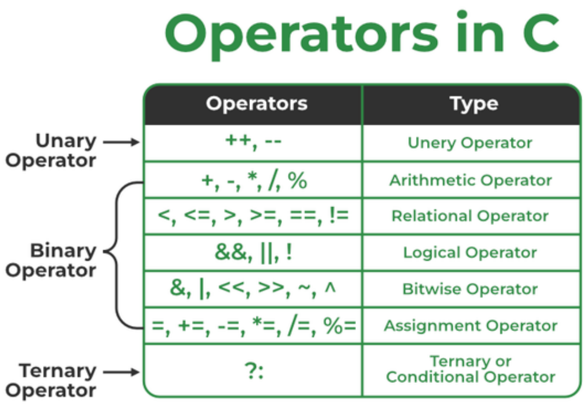
3.6 Operators in C

Operators in C are symbols that perform operations on operands, facilitating various computations. They are categorized as follows:

- **Arithmetic operators:** +, −, ∗, /, %
- **Relational operators:** ==, !=, <, >
- **Logical operators:** &&, \||, !**Bitwise operators:**&, \|, ˆ, ~, <<, >>

- **Assignment operators:** =, +=, −=, ∗=, /=
- **Miscellaneous operators:** Ternary (?:), `sizeof`, and comma operators.

Operator precedence determines the order in which expressions are evaluated. For instance, multiplication (∗) has higher precedence than addition (+), ensuring the correct computation of complex expressions.


Operators in C
Unary Operator
Binary Operator
Ternary Operator

Operators	Type
++, --	Unery Operator
+, -, *, /, %	Arithmetic Operator
<, <=, >, >=, ==, !=	Relational Operator
&&, ||, !	Logical Operator
&, |, <<, >>, ~, ^	Bitwise Operator
=, +=, -=, *=, /=, %=	Assignment Operator
?:	Ternary or Conditional Operator

Figure 2: Operators in C

4 Selection Statements

Conditional logic lies at the heart of dynamic and responsive programming. In the C programming language, this capability is harnessed through a fundamental concept known as **selection statements**. These statements empower developers to introduce decision-making into their code, allowing it to adapt and respond intelligently to varying conditions. As we explore the intricacies of conditional logic, we'll uncover the foundational principles that enable C programs to take diverse paths, enhancing their versatility and functionality.

4.1 if Statement in C

The `if` statement in C is a fundamental control flow structure that allows the execution of a block of code based on a specified condition. The syntax of the `if` statement is straightforward. It begins with the keyword `if`, followed by parentheses `()` containing the condition to be evaluated. If the condition evaluates to true, the code inside the associated block is executed. The block is enclosed within curly braces `{}`. If the condition evaluates to false, the block is skipped, and program execution continues to the next statement. The `if` statement can also be followed by an optional `else` block, which is executed when the condition is false. The general structure is as follows:

```
if (condition) {
    // Code block to execute if condition is true
} else {
    // Code block to execute if condition is false (optional)
}
```

Here, ”condition” represents the expression that determines the execution path. The `else` block is entirely optional, and its absence implies that no action is taken if the condition is false. The `if` statement provides a powerful mechanism for introducing decision- making capabilities into C programs, enabling dynamic and context-aware execution paths.

4.2 switch Statement in C

The `switch` statement in C is a powerful control flow structure designed to handle multiple conditions efficiently. It provides an alternative to the `if-else` construct when dealing with situations where a variable or expression is compared against a series of constant values. The typical scenario involves evaluating the expression and then directing the program's flow to a specific case that matches the evaluated value. The basic syntax of the `switch` statement is as follows:

```
switch (expression) {
    case constant1:
        // Code block for constant1
        break;
    case constant2:
        // Code block for constant2
        break;
    default:
        // Optional default block if no match is found
}
```

Here, "expression" is the variable or expression being evaluated, and each `case` represents a potential value to compare against. If a match is found, the corresponding block of code is executed until the `break` statement is encountered, which exits the `switch` statement. The `default` case is optional and provides a fallback option for situations where none of the specified cases match the evaluated expression. The `switch` statement enhances code readability and maintainability in scenarios where multiple conditional branches are required.

5 Loops

Loops are fundamental constructs in C programming that enable the repetition of a certain block of code. These structures play a pivotal role in automating repetitive tasks, offering a concise and efficient means to execute a sequence of statements multiple times. Loops are particularly useful when dealing with situations where a task needs to be performed iteratively, eliminating the need for redundant code. In C, the incorporation of loops enhances code efficiency and readability, providing a versatile mechanism for managing repetitive operations within a program.

5.1 while Loop in C

The `while` loop in C is a versatile control flow structure that allows a block of code to be repeatedly executed as long as a specified condition remains true. This loop is particularly useful when the number of iterations is not known beforehand and depends on a dynamically changing condition. The basic syntax of the `while` loop is as follows:

```
while (condition) {
    // Code block to execute repeatedly
    // Must contain logic to eventually make the condition false
}
```

Here, ”condition” is an expression that is evaluated before each iteration. If the condition is true, the code within the loop is executed. This process continues until the condition becomes false, at which point the control moves to the next statement after the `while` loop. It’s important to ensure that the condition eventually becomes false to avoid an infinite loop. The `while` loop provides a flexible and concise way to implement repetitive tasks in C, and its simplicity makes it a fundamental building block for various algorithms and program structures.

5.2 for Loop in C

The `for` loop in C is a compact and expressive control flow structure designed for situations where the number of iterations is known beforehand. It combines loop initialization, condition checking, and iteration expression updates within a single line, providing a concise way to manage loop control. The basic syntax of the `for` loop is as follows:

```
for (initialization; condition; iteration) {
    // Code block to execute repeatedly
}
```

Here, ”initialization” is an expression executed once at the beginning of the loop, ”condition” is the loop continuation condition checked before each iteration, and ”iteration” is an expression executed at the end of each iteration. If the condition is true, the code within the loop is executed, and the process repeats until the condition becomes false. The `for` loop is versatile and widely used in C programming for tasks that involve a predetermined number of iterations, such as traversing arrays or implementing numerical algorithms.

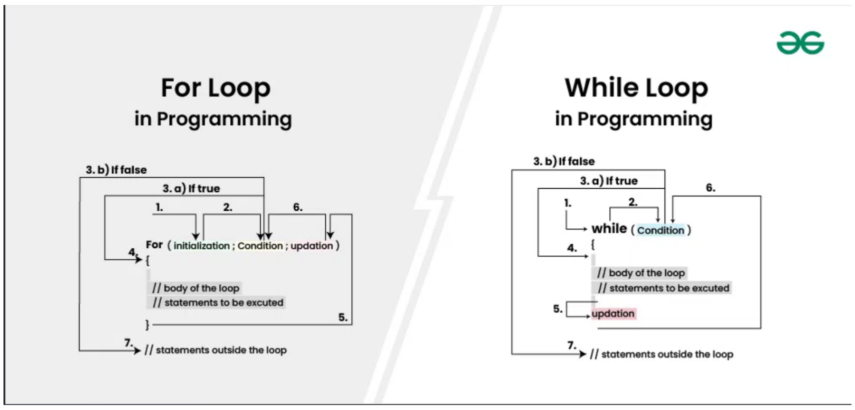


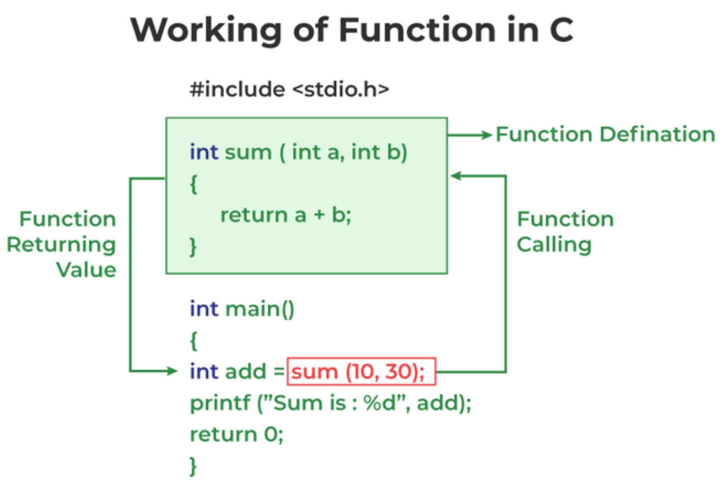
Figure 3: Difference between For Loop and While Loop in Programming

6 Functions

Functions in C are modular, self-contained blocks of code that perform a specific task. They play a crucial role in organizing and simplifying code by allowing the division of a program into smaller, manageable units. A function typically consists of a function signature, which includes the return type, function name, and parameters, followed by a block of code enclosed in curly braces. The basic syntax of a function declaration is as follows:

```
return_type function_name(parameter1_type parameter1,
                         parameter2_type parameter2, ...)
{
    // Body of the function
    // ...
    return value; // Optional return statement
}
```

Here, ”return_type” indicates the type of value the function returns, ”function_name” is the user-defined name of the function, and parameters are input values that the function may require. The `return` statement is optional and is used to send a value back to the calling code. To use a function, it must be declared or defined before its invocation. Function calls include the function name followed by arguments enclosed in parentheses. Functions are pivotal for **code reuse**, **abstraction**, and maintaining a **modular structure**, making C programs more organized and comprehensible.



```
#include <stdio.h>
int sum ( int a, int b)
{
    return a + b;
}
int main()
{
int add = sum (10, 30);
printf ("Sum is : %d", add);
return 0;
}
```

Figure 4: Working of Function in C

7 Arrays

An Array is a data structure consisting of a number of data values that have the same type. These values, called **elements**, can be individually accessed by their position within the array.

7.1 One-dimensional Array in C

A one-dimensional array in C is a structured collection of elements of the same data type, arranged in a linear sequence. It allows programmers to store and manipulate a series of values under a single variable name. Each element in the array is identified by its **index**, starting from 0 for the first element. The basic syntax for declaring and initializing a one-dimensional array is as follows:

```
data_type array_name[size];
```

Here, ”`data_type`” represents the type of data each element will hold (e.g., `int`, `float`, `char`), ”`array_name`” is the user-defined name for the array, and ”`size`” indicates the number of elements in the array. For example, to declare an integer array named ”numbers” with a size of 5:

```
int numbers[5];
```

The elements of the array can be accessed and manipulated using their indices. For instance, to assign a value to the third element (index 2):

```
numbers[2] = 100;
```

Arrays provide a compact and efficient way to work with sets of data in C, and their flexibility makes them a fundamental component of many algorithms and data structures.

7.2 String Representation in C

In C, a **string** is represented as an array of characters terminated by a **null character (`'
0'`)**. This null character marks the end of the string. Strings in C are typically declared using character arrays. The basic syntax for declaring and initializing a string is as follows:

```
char string_name[size];
```

Here, ”`char`” specifies the data type for each character, ”`string_name`” is the user-defined name for the string, and ”`size`” indicates the maximum number of characters in the string, including the null character. For example:

```
char greeting[10] = "Hello";
```

In this example, the size is 10 to accommodate the characters in the word ”Hello” plus the null character. C provides a convenient shorthand for initializing strings:

```
char name[] = "John Doe";
```

In this case, the size is automatically determined based on the length of the string literal. Strings can be manipulated using various functions from the `<string.h>` library, and individual characters within the string can be accessed using array notation. For instance:

```
char first_char = greeting[0]; // first_char will be 'H'
```

Strings in C are mutable, allowing you to change individual characters. However, care must be taken to ensure that the null character is always present at the end of the string to maintain its proper representation.

7.3 Two-Dimensional Array in C

A **two-dimensional array** in C is a structured collection of elements organized in rows and columns. It can be visualized as a table with rows and columns, where each element is identified by its row and column indices. The basic syntax for declaring and initializing a two- dimensional array is as follows:

```
data_type array_name[row_size][column_size];
```

Here, ”`data_type`” represents the type of data each element will hold (e.g., `int`, `float`, `char`), ”`array_name`” is the user-defined name for the array, ”`row_size`” indicates the number of rows, and ”`column_size`” indicates the number of columns. For example, to declare a 3×4 integer array named ”matrix”:

```
int matrix[3][4];
```

This array has three rows and four columns. Elements in a two-dimensional array can be accessed and manipulated using their row and column indices. For instance, to assign a value to the element in the second row (index 1) and third column (index 2):

```
matrix[1][2] = 50;
```

Two-dimensional arrays are particularly useful for representing grids of data, such as matrices or tables. They provide a structured way to organize and work with data in a grid-like fashion.

8 Pointers

Pointers in C are **variables that store memory addresses**. They provide a way to manipulate data indirectly by referencing the memory location where the data is stored. Pointers play a crucial role in dynamic memory allocation, function parameter passing, and various other advanced programming scenarios. The basic syntax for declaring a pointer is as follows:

```
data_type *pointer_name;
```

Here, ”`data_type`” represents the type of data the pointer will point to, and ”`pointer_name`” is the user-defined name for the pointer. For example, to declare a pointer to an integer:

```
int *pointer_to_number;
```

To initialize a pointer with the address of a variable, you use the **address-of operator (&)**. For instance:

```
int number = 42;
int *pointer_to_number = &number;
```

In this example, `pointer_to_number` now holds the memory address of the `number` variable. To access the value at the memory location pointed to by a pointer, you use the **dereference operator (*)**. For example:

```
int value = *pointer_to_number; // value will be 42
```

This retrieves the value stored at the memory address pointed to by `pointer_to_number`. Pointers are versatile and powerful, allowing for dynamic memory management, efficient data manipulation, and advanced programming techniques in C.

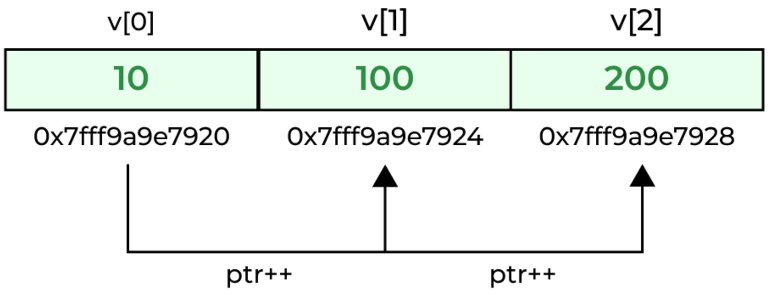


Figure 5: Working of a Pointer in C

9 Data Modifiers

Data modifiers in C are keywords used to alter the properties of basic data types, affecting their range and behavior. Two primary data modifiers in C are **signed** and **unsigned**. These modifiers are commonly applied to integer data types to specify whether the variable can represent both positive and negative numbers (`signed`) or only non-negative numbers (`unsigned`). The basic syntax for using data modifiers is as follows:

```
data_modifier data_type variable_name;
```

Here, ”`data_modifier`” represents the modifier keyword (`signed` or `unsigned`), ”`data_type`” is the basic data type (e.g., `int`, `char`), and ”`variable_name`” is the user-defined name for the variable. For example:

```
unsigned int positive_count;
signed char temperature; // 'signed' is usually the default for int and char
```

When no modifier is specified, the default is usually `signed`.

9.1 Other Important Data Qualifiers

- **The `const` keyword** in C is used to declare constants—variables whose values cannot be modified after initialization. Example:

  ```
  const int MAX_VALUE = 100;
  ```

- **The `static` keyword** declares static variables, retaining their values between function calls and initializing only once. Example:

  ```
  static int counter = 0;
  ```

- **The `volatile` qualifier** indicates that a variable’s value may change unexpectedly (e.g., by hardware), preventing certain compiler optimizations. Example:

  ```
  volatile int sensor_reading;
  ```

Data modifiers are particularly relevant in scenarios where the range of representable values is crucial. For instance, an `unsigned int` can represent a larger positive range compared to a regular `int`, but it cannot represent negative values. It’s important to choose the appropriate data modifier based on the requirements of the specific variable or data being represented.

10 User Defined Data Types

User-defined data types in C allow programmers to create custom data types based on their specific requirements. There are three primary types of user-defined data types in C: **structures (`struct`)**, **unions (`union`)**, and **enumerations (`enum`)**.

10.1 Structures (`struct`)

Structures allow bundling together different data types under a single name.

- **Syntax:**

```
struct structure_name {
    data_type member1;
    data_type member2;
    // ...
};
```

- **Example:**

```
struct Point {
    int x;
    int y;
};
```

10.2 Unions (`union`)

Unions enable storing different data types in the **same memory location**.

- **Syntax:**

```
union union_name {
    data_type member1;
    data_type member2;
    // ...
};
```

- **Example:**

```
union Data {
    int i;
    float f;
};
```

10.3 Enumerations (`enum`)

Enumerations provide a way to create named constant values representing integer constants.

- **Syntax:**

```
enum enumeration_name {
    constant1,
    constant2 = value,
    // ...
};
```

- **Example:**

```
enum Days {
    SUNDAY = 1,
    MONDAY, // Automatically 2
    TUESDAY
};
```

11 Preprocessor Directive

A Preprocessor Directive is a command that is executed **before** a program is compiled, to perform some operations before actual compilation.

11.1 C Building Process

1. **Preprocessing:** Directives (#include, #define) are handled.
2. **Compilation:** Preprocessed code is translated into assembly code.
3. **Assembly:** Assembly code is translated into object code (machine code).
4. **Linking:** Object files and libraries are combined to create the final executable program.

Preprocessor directives start with the # symbol and are **not** terminated by a semicolon.

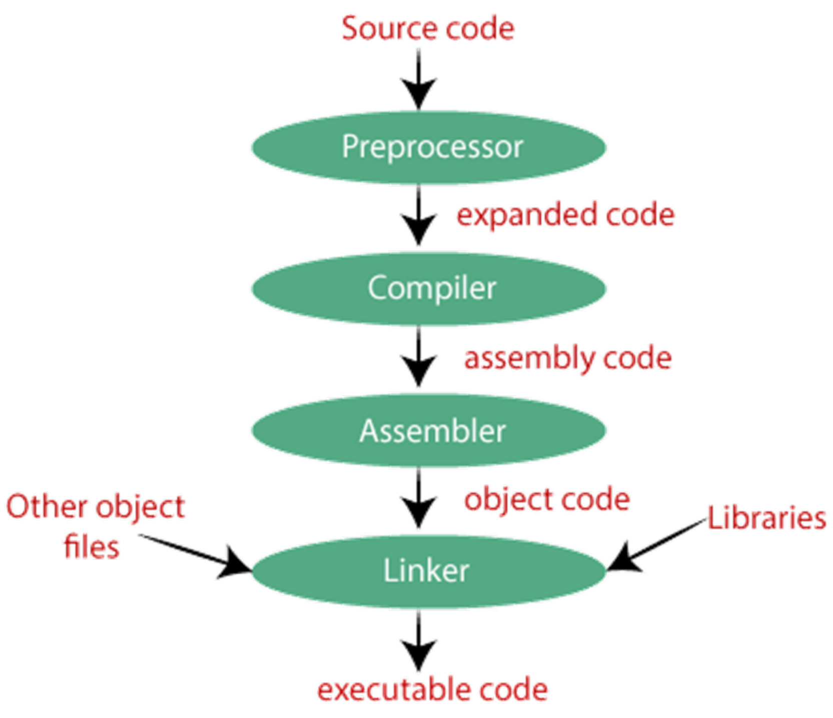


Figure 6: C Build Process

11.2 Common Preprocessor Directives

- **Include Directive (`#include`):** This directive is used to include the contents of another file (typically header files) in the source code.

  ```
  #include <filename.h> // For system headers
  #include "filename.h" // For local headers
  ```

- **Define Directive (`#define`):** This directive is used to create symbolic constants and macros, which are replaced with specified expressions during preprocessing.

  ```
  #define MACRO_NAME value
  #define PI 3.14159
  ```

- **Conditional Compilation (`#ifdef`, `#ifndef`, `#else`, `#endif`):** These directives are used for conditional compilation, allowing certain parts of the code to be included or excluded based on conditions.

  ```
  #ifdef DEBUG
  // Code to include if DEBUG is defined
  #else
  // Code to include if DEBUG is not defined
  #endif
  ```

- **Undefine Directive (`#undef`):** This directive is used to undefine a previously defined macro or symbol.

  ```
  #undef PI
  ```

Preprocessor directives provide a way to enhance code modularity, manage configurations, and control compilation based on specific conditions. Their use is widespread in C programming, especially in large projects and library development.

12 C Code Examples: Practical Illustrations (12 Examples)

12.1 Selection Statements (if-else and switch)

1. **if-else Example:**

```
int score = 75;
if (score >= 50) {
    printf("Passed.\n");
} else {
    printf("Failed.\n");
}
```

2. **switch Example:**

```
char grade = 'B';
switch (grade) {
    case 'A':
        printf("Excellent.\n");
        break;
    case 'B':
        printf("Good.\n"); // This will execute
        break;
    default:
        printf("Needs improvement.\n");
}
```

12.2 Loops (for and while)

3. **for Loop Example:**

```
// Print numbers from 1 to 5
for (int i = 1; i <= 5; i++) {
    printf("%d ", i);
} // Output: 1 2 3 4 5
```

4. **while Loop Example:**

```
int j = 0;
// Print numbers from 0 until j is less than 3
while (j < 3) {
    printf("%d ", j);
    j++;
} // Output: 0 1 2
```

12.3 Functions

5. **Function with Return Value:**

```
int multiply(int x, int y) {
    return x * y;
}

int result = multiply(4, 5); // result is 20
```

6. **Void Function (No Return):**

```
void greet(char name[]) {
    printf("Hello, %s!\n", name);
}

greet("Ahmed"); // Prints: Hello, Ahmed!
```

12.4 Arrays

7. **1D Array Declaration and Access:**

```
int numbers[3] = {10, 20, 30};
int third_element = numbers[2]; // third_element is 30
```

8. **String (Char Array) Example:**

```
char str[] = "AVR";
printf("The string is: %s\n", str); // Output: AVR
```

12.5 Pointers

9. **Basic Pointer Usage (Address and Dereference):**

```
int x = 10;
int *p = &x; // p stores address of x
printf("Value is: %d\n", *p); // Output: 10
```

12.6 Data Modifiers

10. **unsigned and const Modifiers:**

```
const int MAX_USERS = 50;
unsigned char status_flags = 255;
// MAX_USERS cannot be changed.
```

12.7 User Defined Data Types

11. **Structure (`struct`) Example:**

```
struct Student {
    int id;
    float gpa;
};
struct Student s1;
s1.id = 101;
```

12.8 Preprocessor Directive

12. **#define Directive Example:**

```
#include <stdio.h>
#define BUFFER_SIZE 1024

int main() {
    char data[BUFFER_SIZE]; // Uses 1024
    // ...
}
```

13 Embedded System Concepts

Embedded systems are specialized computing systems dedicated to specific tasks within larger systems. They are designed to operate in real-time and are often found in various devices and applications, ranging from consumer electronics and automotive systems to industrial machinery and medical devices. Here are key concepts associated with embedded systems:

- **Microcontroller/Microprocessor:** The core processing unit in an embedded system is typically a microcontroller or microprocessor. These compact chips integrate CPU, memory, and peripherals on a single chip, optimizing space and power consumption.
- **Peripheral Devices:** Embedded systems interact with the external world through peripherals such as sensors, actuators, displays, and communication interfaces. These devices enable the system to sense and control its environment.
- **Memory Management:** Memory in embedded systems is often constrained, and efficient memory management is crucial. Systems may use Flash memory for program storage and RAM for data storage. Some embedded systems employ read-only memory (ROM) for firmware.
- **Power Efficiency:** Embedded systems are frequently battery-powered or have power constraints. Power-efficient design is essential to maximize the system's operational time. Techniques like low-power modes and optimized algorithms contribute to energy savings.
- **Communication Protocols:** Embedded systems often communicate with other devices or systems. Common communication protocols include UART, SPI, I2C, and CAN. Networking protocols like Ethernet and wireless protocols (Wi-Fi, Bluetooth) are also used.
- **Development Tools and IDEs:** Embedded systems development relies on specialized tools and Integrated Development Environments (IDEs) that support cross- compilation, debugging, and simulation for the target architecture.
- **Cross-Compilation:** Due to resource constraints on embedded systems, development often involves cross-compiling software on a host machine for the target architecture.

Understanding these concepts is vital for designing and developing effective embedded systems that meet the specific requirements and constraints of their intended applications. The field of embedded systems is continually evolving with advancements in hardware, software, and application domains.

14 Digital Input/Output (DIO)

”DIO” typically stands for **Digital Input/Output** in the context of embedded systems or microcontrollers. Digital Input/Output refers to the ability of a microcontroller to interact with the external world using digital signals, where a signal can be either in a high (1) or low (0) state.

14.1 ATMEGA32 Pinouts

Here’s the pinouts of the ATMEGA 32
In microcontrollers or embedded systems, digital I/O functionality is often associated with specific registers and pins. Registers are memory locations within the microcontroller that control various aspects of its behavior, including I/O operations. Here’s a breakdown of the essential concepts related to Digital Input/Output:

14.2 General-Purpose I/O (GPIO)

Many microcontrollers provide GPIO pins that can be configured as digital inputs or outputs. These pins allow the microcontroller to interface with external components, such as sensors, actuators, or other microcontrollers.

14.3 Data Direction Register (DDR)

The **Data Direction Register (DDR)** determines whether a particular pin is configured as an input or an output. Each bit in the DDR corresponds to a specific pin, where setting the bit to 1 configures the pin as an output, and setting it to 0 configures the pin as an input.

14.4 Port Register (PORT)

The **Port Register (PORT)** is used to write data to output pins or read data from input pins. When a pin is configured as an output, writing a 1 to the corresponding bit in the PORT register sets the pin to a high state, and writing a 0 sets it to a low state. When configured as an input, reading from the PORT register retrieves the logic level on the pin.

14.5 Pin Register (PIN)

The **Pin Register (PIN)** is used to read the state of input pins. Reading from the PIN register returns the logic level on each input pin.

14.6 Digital Output and Input Summary

- **Digital Output:** To set a pin as a digital output, you would set the corresponding bit in the Data Direction Register (DDR) to `1`. To write a logic level to the output pin, you use the Port Register (PORT).
- **Digital Input:** To set a pin as a digital input, you would set the corresponding bit in the Data Direction Register (DDR) to `0`. To read the logic level on the input pin, you use the Pin Register (PIN).

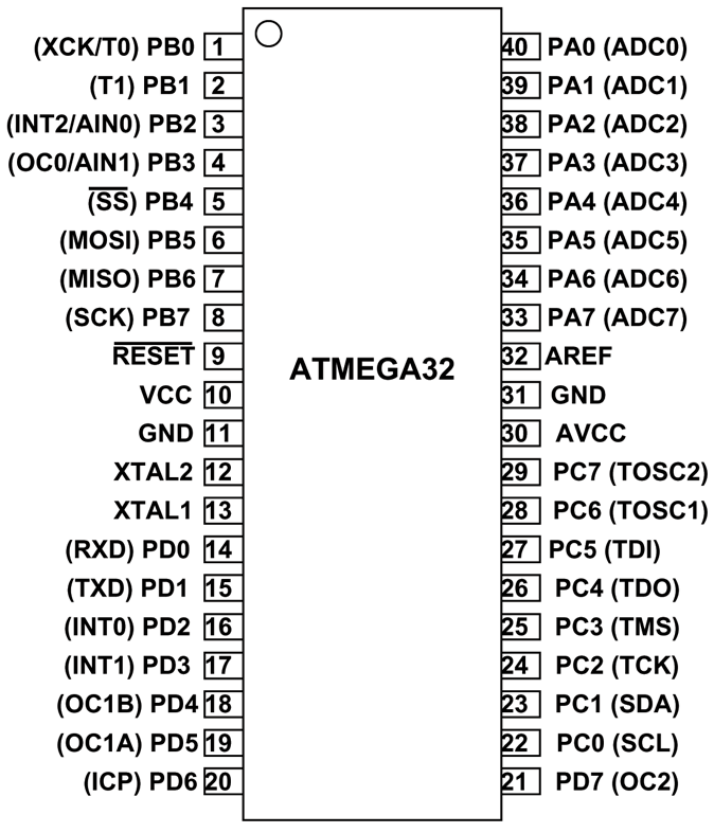


Figure 7: Pinouts ATmega32

15 Layered Architecture

Layered architecture is a software design pattern where the system is divided into distinct, horizontal layers, each performing a specific role. This structure is foundational for developing complex, scalable, and maintainable embedded software.

15.1 Advantages of Layered Architecture

1. **Modularity:** In a layered architecture, we separate the user application from the hardware drivers and from the microcontroller specific drivers. This clear separation makes each part easier to understand and manage independently.

2. **Portability:** Changing any part of the software would affect its layer only. For example, if we need the same application with a new microcontroller, we shall only change the **Microcontroller Abstraction Layer (MCAL)** while keeping the application logic and other layers intact.

3. **Reusability:** Code could be easily reused in different applications and systems. Hardware-independent code, typically residing in the application layer, can be ported across different hardware platforms without modification.

4. **Maintainability:** Debugging and testing are now much easier in small parts of the software instead of having a very long and complex one. When an issue is detected, it is generally localized to a specific layer, simplifying the process of maintenance and updates.

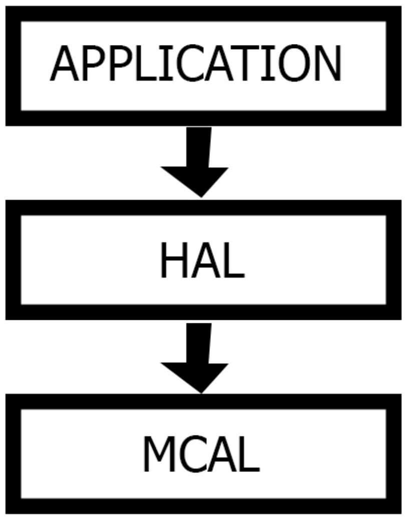


Figure 8: Layered Architecture

16 Liquid Crystal Display (LCD)

An **LCD (Liquid Crystal Display)** is a flat-panel display technology that uses liquid crystals to modulate light and produce images. LCDs are commonly used in embedded systems for displaying information to users. The integration of LCDs with embedded systems allows for the creation of user interfaces, status displays, and other visual feedback mechanisms.

16.1 LCDs and Embedded Systems

1. **Display Output:** LCDs provide a visual output for embedded systems, enabling the presentation of information in a structured and user-friendly manner (e.g., in digital thermometers and electronic meters).
2. **User Interface:** LCDs serve as a visual interface between the user and the embedded system. They can display alphanumeric characters, graphics, and icons, providing a means for users to interact with and interpret data.
3. **Feedback Mechanism:** LCDs offer a real-time feedback mechanism, allowing users to receive information about the system's state, measurements, or any relevant data, which is essential in applications requiring immediate visual feedback.
4. **Low Power Consumption:** Many LCDs have relatively low power consumption, making them suitable for battery-powered embedded systems where energy efficiency is crucial.
5. **Compact Size:** LCDs are available in various sizes and form factors, allowing for flexibility in design and integration into compact embedded systems.

16.2 LCD Pinout (Character LCD)

The pinout of an LCD varies based on its type. Here is a general overview of the common pins found in a character LCD (such as a 16x2):

1. **VCC and GND:**
   - **VCC (Voltage):** Power supply for the LCD.
   - **GND (Ground):** Ground reference for the LCD.
2. **RS (Register Select):** Determines whether data sent to the LCD is a **command** (Low) or **character/data** (High).
3. **RW (Read/Write):** Determines the direction of data transfer. High for reading from the LCD, low for writing to the LCD.

4. **E (Enable):** Enables the LCD to latch in data or commands. Rising edge triggers the operation.

5. **D0-D7 (Data Lines):** 8-bit or 4-bit parallel data lines for sending commands or character data.

6. **A (Anode) and K (Cathode):** Backlight connections (if the LCD has a backlight). Connect to power for backlighting.

16.3 LCD Memory Organization

In an LCD, the memory is divided into three main areas: **CGRAM**, **CGROM**, and **DDRAM**.

16.3.1 CGRAM (Character Generator RAM)

- **Purpose:** CGRAM is dedicated to storing **custom character patterns** defined by the user.
- **Custom Character Creation:** The programmer specifies a bitmap pattern in CGRAM (a series of on/off pixels) that defines the shape of the custom character.
- **Addressing:** CGRAM is addressed using specific addresses for each custom character slot (e.g., Addresses `0x00` to `0x3F` for the first 64 addresses).

16.3.2 CGROM (Character Generator ROM)

- **Purpose:** CGROM stores **predefined character patterns** from the standard ASCII character set.
- **Fixed Patterns:** The character patterns in CGROM are fixed and **cannot be modified** by the programmer.
- **Addressing:** CGROM is addressed using ASCII codes.

16.3.3 Display RAM (DDRAM)

- **Purpose:** DDRAM (Display Data RAM) is the primary area used for **displaying characters** on the LCD screen. It holds the data to be displayed (from both CGRAM and CGROM).
- **Addressing:** DDRAM is addressed by row and column coordinates. The programmer sets the cursor position in DDRAM, and subsequent data is displayed at that position.
- **Example (16x2):** DDRAM addresses might be `0x80` to `0x8F` (for the first row) and `0xC0` to `0xCF` (for the second row).

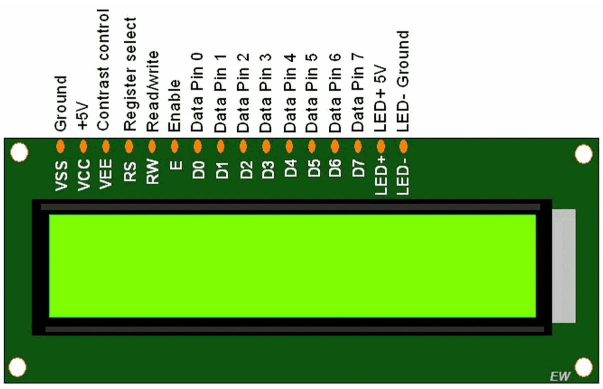


Figure 9: LCD Pins

17 Keypad: The Matrix Input

A **keypad** is a fundamental input device that allows users to enter data or make selections by pressing buttons arranged in a matrix. It's the trusty input method you find everywhere, from calculators to security systems.

17.1 Keypad Structure

1. **Matrix Layout:** Keypads are organized in a grid with **rows** and **columns**. The beauty of this design is that the intersection of a specific row and column represents a **unique button**.
2. **Buttons:** Each button corresponds to a specific input, such as a digit (0-9), letter, or a special function like ”Enter,” ”Clear,” or ”Function.”

17.2 How Keypads Work: Matrix Scanning

Keypads don't need a separate wire for every button; instead, they use a clever technique called **matrix scanning** to save microcontroller pins.

1. **Row and Column Connection:** When a button is pressed, it acts as a mechanical switch, **bridging a connection** between one specific row line and one specific column line.
2. **Microcontroller Scanning:** The microcontroller systematically scans:
   - It sets one **row pin** to LOW (or HIGH) at a time.
   - It then **monitors all the column pins**.
3. **Key Identification:** If the microcontroller detects a signal (LOW or HIGH, depending on setup) on a column pin while a specific row is active, it knows exactly which row and column intersected, thus identifying the **unique pressed key**.

18 Interrupts: Event-Driven Processing

An **interrupt** is the superhero mechanism of embedded systems. It's a combination of hardware and software that **forces the processor to stop** its current activity and immediately execute a highly specific piece of code called an **Interrupt Service Routine (ISR)**, usually in response to a critical event.

18.1 Interrupt Basics

1. **Event-Driven Handling:** Interrupts are triggered by events, such as a physical button press (External Interrupts), a timer finishing its count (Internal Interrupts), or the arrival of new communication data.
2. **Interrupt Service Routine (ISR):** This is the dedicated function that runs when the interrupt fires. The ISR is solely responsible for handling the triggering event quickly and efficiently.
3. **Interrupt Vector:** This is essentially a map (a table of addresses) that tells the processor exactly where to find the correct ISR for each type of interrupt.
4. **Interrupt Priority:** When multiple events happen simultaneously, the system uses priority levels to decide which ISR should run first, ensuring the most urgent tasks are handled immediately.

18.2 Interrupt Handling Process

1. **Interrupt Request (IRQ):** An external or internal event triggers a request.
2. **Processor Response:** The processor stops its main program, **saves its current state (context)**, and jumps to the address specified by the Interrupt Vector.
3. **Execute ISR:** The Interrupt Service Routine runs, handling the event (e.g., toggling an LED, reading a sensor).
4. **Context Switch:** After the ISR completes, the processor **restores its saved state** and resumes the main program exactly where it left off.

18.3 Interrupt Example (AVR C Code)

This code snippet shows the structure for handling an External Interrupt (INT0) on an AVR Microcontroller:

```
// Define the Interrupt Service Routine for External Interrupt 0
ISR(INT0_vect) {
    // Code to run IMMEDIATELY when the external event occurs
    LED_Toggle(); // Example: Toggle an LED
    _delay_ms(50); // Debounce delay
}

int main() {
    // 1. Configure INT0 Pin
    // 2. Set the trigger condition (e.g., falling edge)
    // 3. Enable the specific interrupt (EIMSK register)

    sei(); // Enable GLOBAL interrupts!

    while(1) {
        // Main program loop runs here, waiting for the event
    }
    return 0;
}
```

19 Timers: The Embedded Clockwork

In the ATmega32 and other AVR microcontrollers, **Timers/Counters** are vital peripherals that provide a versatile way to measure time, generate precise delays, and create periodic events. The ATmega32 features 8-bit (Timer 0, Timer 2) and 16-bit (Timer 1) options.

19.1 Timer/Counter 0 Overview (8-bit)

Timer/Counter 0 is an 8-bit timer, meaning it counts from 0 up to 255.

19.1.1 Key Registers

- **TCNT0 (Timer/Counter Register):** Holds the **current count value**.
- **OCR0 (Output Compare Register):** Holds a value for comparison in CTC mode.
- **TCCR0 (Control Register):** Controls the timer's **mode** (Normal, CTC, PWM) and the **prescaler**.
- **TIMSK (Timer Interrupt Mask Register):** Enables or disables specific timer interrupts (e.g., Overflow, Compare Match).
- **Prescaler:** A divider that slows down the clock frequency to adjust the timer's resolution and maximum period. (Common values: 1, 8, 64, 256, 1024).

19.1.2 Common Modes of Operation

1. **Normal Mode:** The timer counts up until it reaches the maximum value (255) and then **overflows (resets to 0)**. Suitable for basic timekeeping.
2. **CTC Mode (Clear Timer on Compare Match):** The timer counts up until it matches the value stored in the **OCR0** register. Upon match, the timer **clears to 0**, and an interrupt can be triggered. Ideal for generating precise, regular time intervals.
3. **PWM Mode (Pulse Width Modulation):** Used to generate square wave signals with varying duty cycles, essential for motor control (LED Dimming).

19.2 Timer Example (CTC Interrupt)

This ISR is executed precisely when the timer count matches the Compare Match Register (in CTC mode), ensuring accurate timing:

```
// ISR for Timer 0 Compare Match (executed periodically)
ISR(TIMER0_COMP_vect) {
    static volatile uint16_t timer_count = 0;

    // Increment the count every time the interrupt fires
    timer_count++;

    // Check if 1 second has passed (based on OCR0 and prescaler setup)
    if (timer_count >= 1000) {
        // Perform the time-critical task
        printf("1 second elapsed!\n");
        timer_count = 0;
    }
}

// Example setup for CTC mode (TCCR0 = 0x0A; OCR0 = 125; etc.)
// ... TCCR0 = 0x0A; is a simplified setting
// The final configuration depends on CPU clock and desired frequency.
```

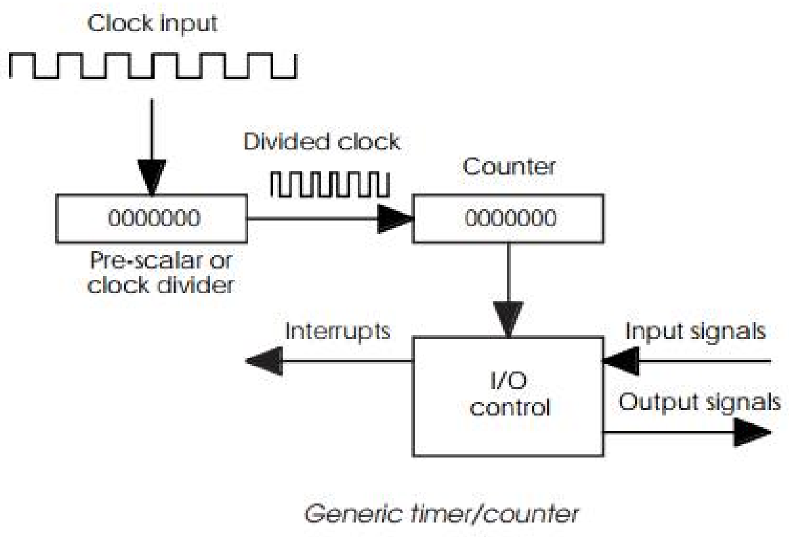


Figure 10: Timer/Counter Diagram

20 UART Communication

The ATmega32 microcontroller features a versatile **UART (Universal Asynchronous Receiver/Transmitter)** module that facilitates serial communication in embedded systems using C programming.
UART is a crucial component for transmitting and receiving **asynchronous serial data** between the microcontroller and external devices.

In C programming for the ATmega32, configuring and utilizing the UART involves initializing the necessary registers, such as:

- **UBRR:** Used for **Baud Rate** control.
- **UCSRnA/B/C:** Used for status and control settings (e.g., enabling transmitter/receiver, defining data frame).
- **UDR:** Used for data transmission (writing the byte to transmit or reading the received byte).

By setting these registers appropriately, developers can establish communication protocols, define baud rates, and implement efficient data exchange with other devices.
The UART functionality in the ATmega32 is instrumental for interfacing with peripherals, sensors, or other microcontrollers, enabling seamless communication in embedded systems applications.

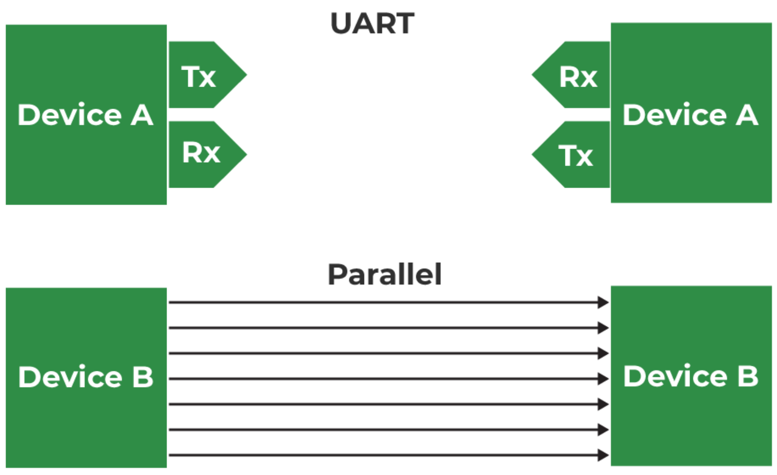


Figure 11: UART Diagram

21 SPI Communication

The ATmega32 microcontroller boasts a robust **Serial Peripheral Interface (SPI)** module, a key feature for high-speed **synchronous serial communication** in C programming. SPI facilitates **full-duplex** communication between the ATmega32 and external devices, such as sensors, memory chips, or other microcontrollers.

In C programming for the ATmega32, configuring the SPI involves setting specific registers, including:

- **SPCR (SPI Control Register):** Controls parameters like data order, clock polarity, clock phase, and setting the device as **Master** or **Slave**.
- **SPDR (SPI Data Register):** Used for data exchange (writing the byte to transmit or reading the received byte).

Developers can tailor parameters like data order, clock polarity, and clock phase to meet the requirements of connected devices. Leveraging SPI's capabilities allows for efficient data exchange, making it particularly useful for applications requiring rapid and synchronized data transfer.

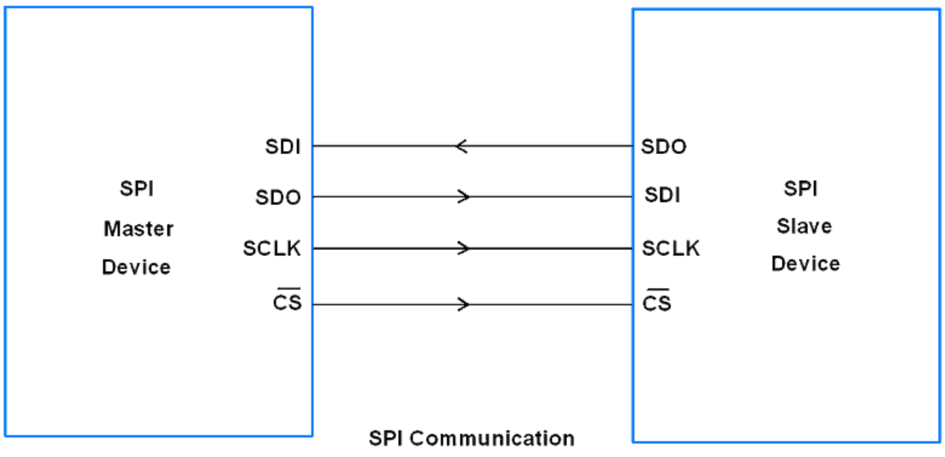


Figure 12: SPI Communication

22 I2C Communication

The ATmega32 microcontroller incorporates a versatile **Inter-Integrated Circuit (I2C)** module (often referred to as TWI - Two-Wire Interface), a crucial component for serial communication in C programming. I2C is a **multi-master, multi-slave** protocol that facilitates communication between various devices on a shared bus.

In C programming for the ATmega32, utilizing the I2C module involves configuring specific registers like:

- **TWBR (Bit Rate Register):** Used to set the clock frequency (SCL rate).
- **TWSR (Status Register):** Indicates the current status of the I2C operation.
- **TWDR (Data Register):** Used for data transfer operations.

Developers can define the microcontroller as either a **Master** or a **Slave**, and handle data transfer operations seamlessly using the device's unique address. The I2C protocol is particularly valuable for connecting a multitude of devices, such as sensors and EEPROMs, in embedded systems.

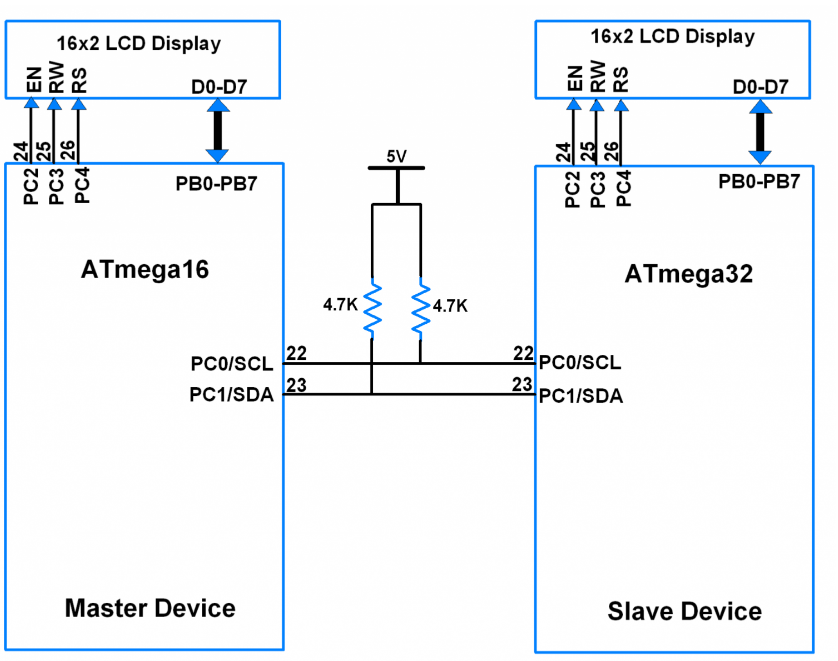


Figure 13: I2C Interfacing

23 Software Testing

Software testing is a critical phase in the development of embedded systems. It ensures that both hardware and software components function as intended and that the system meets all design requirements. The main goal of software testing is to detect and correct errors early, reducing future maintenance costs and improving reliability.

Testing can be divided into several levels. **Unit testing** focuses on verifying individual modules or functions to ensure each performs its specific task correctly. **Integration testing** examines how different modules work together, verifying communication and data flow between them. Finally, **System testing** evaluates the complete embedded system under realistic conditions to confirm overall performance.

Common testing methods include **black-box testing**, which examines functionality without considering internal code structure, and **white-box testing**, which tests internal logic and code paths. Tools like simulators and debuggers are often used to execute and trace program behavior step-by-step.

Effective testing enhances the dependability of embedded products, especially those used in critical fields such as automotive systems, medical devices, and industrial automation. By following structured testing procedures, developers can deliver safe, reliable, and efficient embedded solutions.

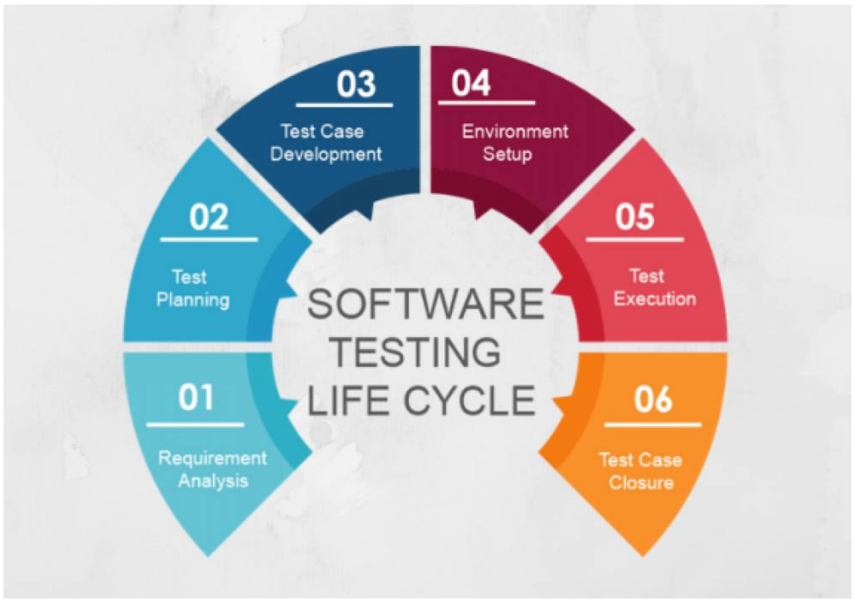


Figure 14: Software Testing Life Cycle

24 Automotive Bus Technology

Modern vehicles rely heavily on electronic control units (ECUs) that communicate with each other through various communication buses. **Automotive bus technology** provides the network backbone that allows sensors, controllers, and actuators to exchange data efficiently and reliably.

The most common automotive communication protocols include:

- **CAN (Controller Area Network):** A robust, high-speed network designed for real-time communication among ECUs. It uses a multi-master architecture and error detection mechanisms for reliability.
- **LIN (Local Interconnect Network):** A simpler, low-cost serial network used for non-critical subsystems such as mirrors, windows, and climate control.
- **FlexRay:** A high-speed deterministic bus suitable for time-critical applications like braking or steering control.
- **Ethernet (Automotive Ethernet):** Used in modern cars for high-bandwidth applications such as cameras, infotainment, and driver-assistance systems.

Each of these technologies serves a specific role based on speed, cost, and reliability requirements. For instance, CAN is ideal for control systems due to its balance between performance and robustness, while LIN is better suited for simple node communication.

Understanding these networks is essential for embedded engineers working in the automotive sector, as communication between ECUs must be both fast and fault-tolerant.

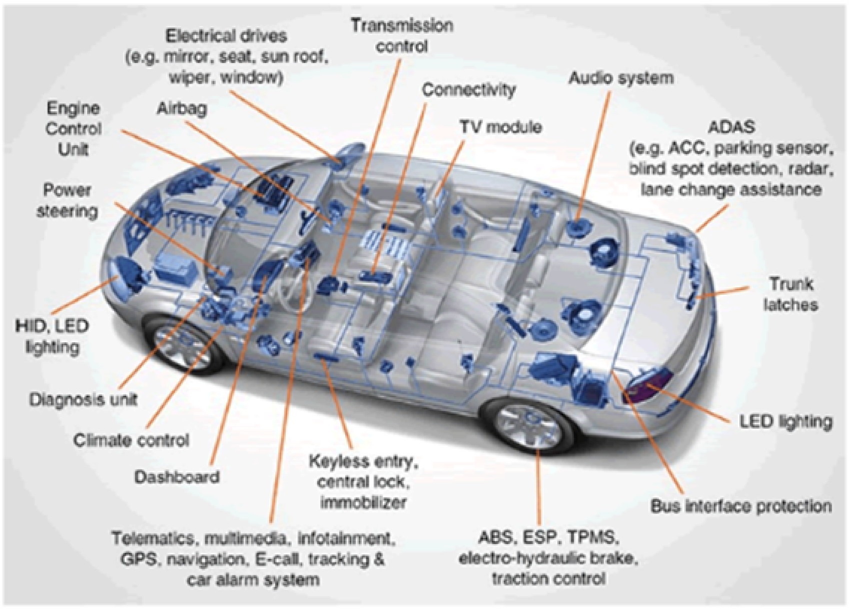

Electrical drives (e.g. mirror, seat, sun roof, wiper, window)
Transmission control
Connectivity
Audio system
Engine Control Unit
Airbag
TV module
ADAS (e.g. ACC, parking sensor, blind spot detection, radar, lane change assistance
Power steering
Trunk latches
HID, LED lighting
Diagnosis unit
LED lighting
Climate control
Dashboard
Keyless entry, central lock, immobilizer
Bus interface protection
Telematics, multimedia, infotainment, GPS, navigation, E-call, tracking & car alarm system
ABS, ESP, TPMS, electro-hydraulic brake, traction control

Figure 15: Automotive Technology Example

25 Embedded Systems Tooling

Developing embedded systems requires a combination of hardware and software tools that help engineers design, build, and test their solutions efficiently. These tools form the embedded systems development ecosystem.

At the software level, **Integrated Development Environments (IDEs)** such as Atmel Studio, MPLAB, or Keil are used to write, compile, and debug C code for microcontrollers. They often include simulators that allow developers to test programs virtually before uploading them to actual hardware.

Hardware tools include **programmers and debuggers** (like AVRISP or ST-Link) that interface with the microcontroller to upload code or monitor runtime behavior. Additionally, instruments like oscilloscopes, logic analyzers, and multimeters help verify signal integrity, timing, and circuit behavior.

Version control tools such as Git also play an important role in collaborative projects, enabling teams to manage source code changes efficiently.

By mastering these tools, embedded developers can shorten development cycles, reduce errors, and ensure that the final system performs exactly as intended—bridging the gap between code and hardware reality.

26 Projects and Practical Application

This section documents the practical application of C programming concepts and embedded system logic through dedicated project examples.

26.1 Project 1: Clinic Management System (C-Console)

A comprehensive console-based system developed in C to manage patient records and doctor appointments in a small clinic. The system operates in two distinct modes: Admin and User.

26.1.1 System Architecture Overview

The system utilizes global arrays for storage (Patient records and reservation slots) and employs functions for modularity. Key concepts implemented include:

- **Data Structures (`struct`):** Used to define patient records and reservation slots.
- **Control Flow:** Extensive use of `if-else` and `while` loops for menus, login trials, and input validation.
- **Functions:** Modular approach for Admin features (Add, Edit, Reserve, Cancel) and User features (View Record, View Reservations).

26.1.2 Admin Mode Features

Accessing the Admin Mode requires a password (`1234`) with three attempts allowed before system closure. Features include:

1. **Add New Patient Record:** Takes Name, Age, Gender, and ensures the entered **ID is unique**.
2. **Edit Patient Record:** Allows modification of patient details after verifying the existence of the entered ID.
3. **Reserve a Slot:** Displays 5 default slots (2pm to 2:30pm, 2:30pm to 3pm, 3pm to 3:30pm, 4pm to 4:30pm, and 4:30pm to 5pm) and assigns an available slot to a patient ID, ensuring the patient does not have a double reservation.
4. **Cancel Reservation:** Removes the patient ID from a reserved slot, making the slot available again.

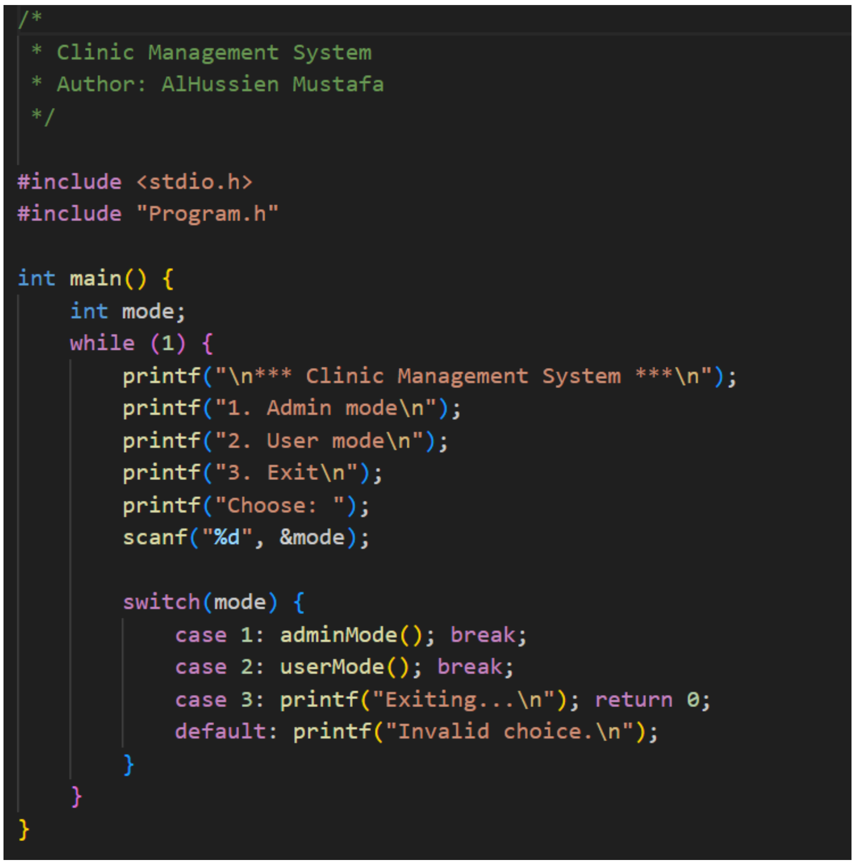
26.1.3 User Mode Features

The User Mode allows read-only access without a password, focusing on viewing information:

1. **View Patient Record:** Displays Name, Age, Gender, and ID upon successful entry of the Patient ID.
2. **View Today's Reservations:** Lists all reserved time slots along with the corresponding Patient ID for each reservation.

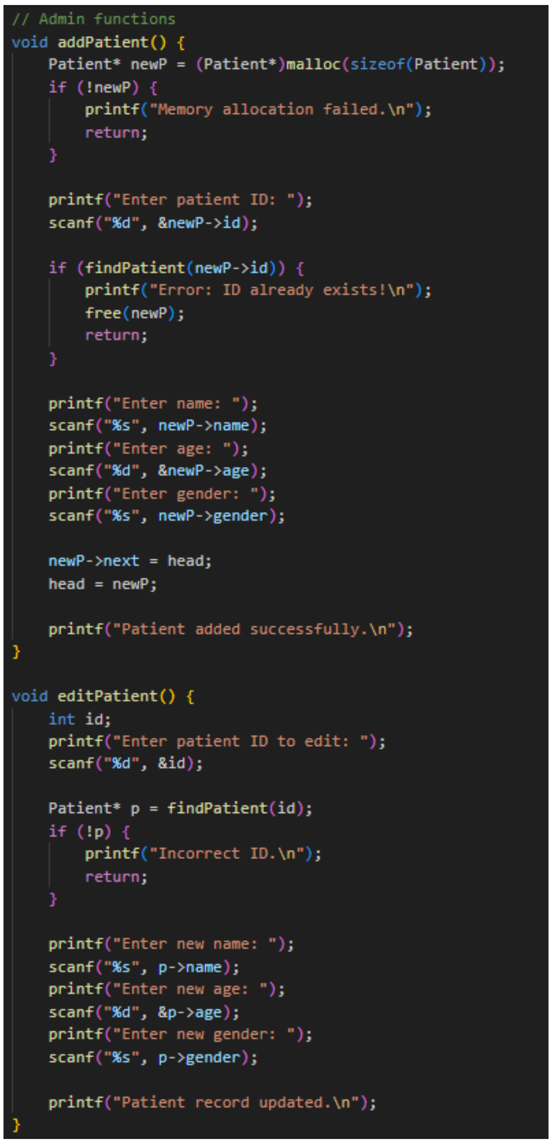
26.1.4 C Code Implementation and Output

The implementation relies on basic C constructs for maximum portability and simplicity.



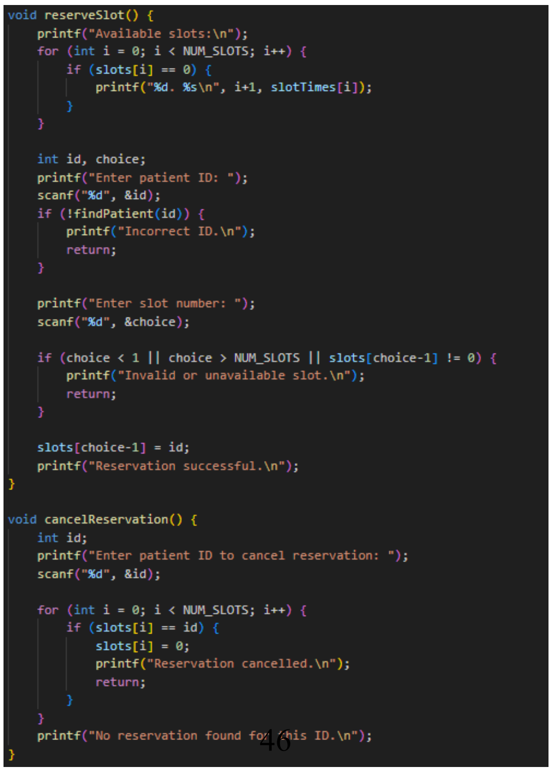
```c
/*
 * Clinic Management System
 * Author: AlHussien Mustafa
 */

#include <stdio.h>
#include "Program.h"

int main() {
    int mode;
    while (1) {
        printf("\n*** Clinic Management System ***\n");
        printf("1. Admin mode\n");
        printf("2. User mode\n");
        printf("3. Exit\n");
        printf("Choose: ");
        scanf("%d", &mode);

        switch(mode) {
            case 1: adminMode(); break;
            case 2: userMode(); break;
            case 3: printf("Exiting...\n"); return 0;
            default: printf("Invalid choice.\n");
        }
    }
}
```

Figure 16: Main Code



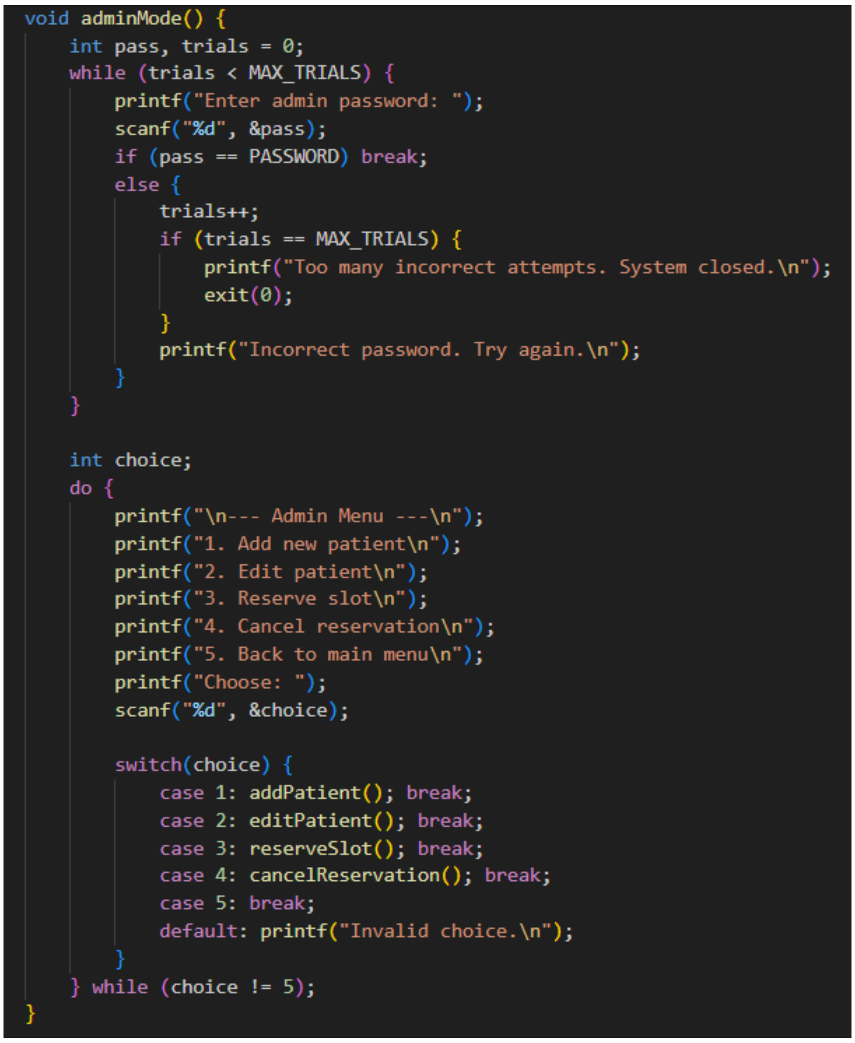
```c
// Admin functions
void addPatient() {
    Patient* newP = (Patient*)malloc(sizeof(Patient));
    if (!newP) {
        printf("Memory allocation failed.\n");
        return;
    }

    printf("Enter patient ID: ");
    scanf("%d", &newP->id);

    if (findPatient(newP->id)) {
        printf("Error: ID already exists!\n");
        free(newP);
        return;
    }

    printf("Enter name: ");
    scanf("%s", newP->name);
    printf("Enter age: ");
    scanf("%d", &newP->age);
    printf("Enter gender: ");
    scanf("%s", newP->gender);

    newP->next = head;
    head = newP;

    printf("Patient added successfully.\n");
}

void editPatient() {
    int id;
    printf("Enter patient ID to edit: ");
    scanf("%d", &id);

    Patient* p = findPatient(id);
    if (!p) {
        printf("Incorrect ID.\n");
        return;
    }

    printf("Enter new name: ");
    scanf("%s", p->name);
    printf("Enter new age: ");
    scanf("%d", &p->age);
    printf("Enter new gender: ");
    scanf("%s", p->gender);

    printf("Patient record updated.\n");
}
```

(a) Admin Functions Code



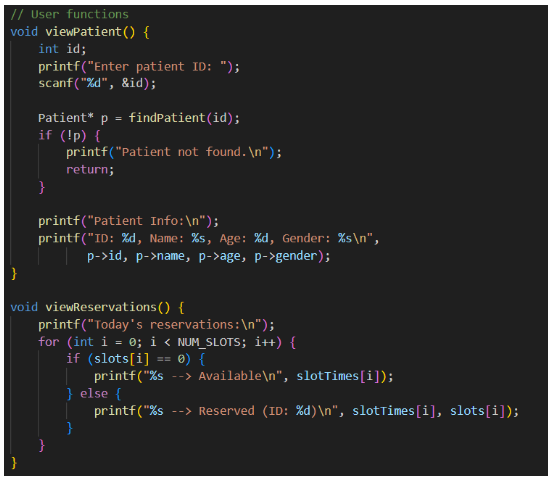
```c
void reserveSlot() {
    printf("Available slots:\n");
    for (int i = 0; i < NUM_SLOTS; i++) {
        if (slots[i] == 0) {
            printf("%d. %s\n", i+1, slotTimes[i]);
        }
    }

    int id, choice;
    printf("Enter patient ID: ");
    scanf("%d", &id);
    if (!findPatient(id)) {
        printf("Incorrect ID.\n");
        return;
    }

    printf("Enter slot number: ");
    scanf("%d", &choice);

    if (choice < 1 || choice > NUM_SLOTS || slots[choice-1] != 0) {
        printf("Invalid or unavailable slot.\n");
        return;
    }

    slots[choice-1] = id;
    printf("Reservation successful.\n");
}

void cancelReservation() {
    int id;
    printf("Enter patient ID to cancel reservation: ");
    scanf("%d", &id);

    for (int i = 0; i < NUM_SLOTS; i++) {
        if (slots[i] == id) {
            slots[i] = 0;
            printf("Reservation cancelled.\n");
            return;
        }
    }
    printf("No reservation found for this ID.\n");
}
```

(b) Admin Functions Code



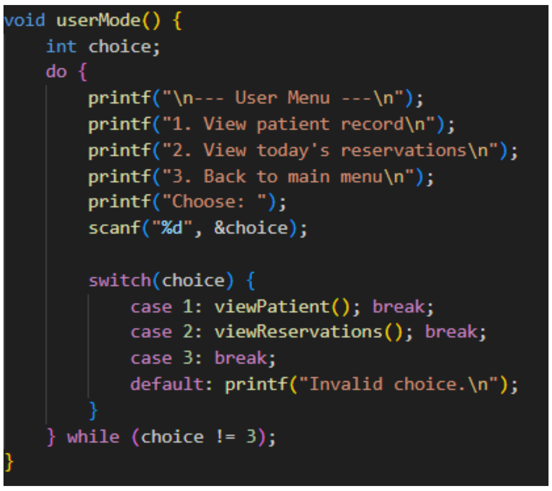
```c
void adminMode() {
    int pass, trials = 0;
    while (trials < MAX_TRIALS) {
        printf("Enter admin password: ");
        scanf("%d", &pass);
        if (pass == PASSWORD) break;
        else {
            trials++;
            if (trials == MAX_TRIALS) {
                printf("Too many incorrect attempts. System closed.\n");
                exit(0);
            }
            printf("Incorrect password. Try again.\n");
        }
    }

    int choice;
    do {
        printf("\n--- Admin Menu ---\n");
        printf("1. Add new patient\n");
        printf("2. Edit patient\n");
        printf("3. Reserve slot\n");
        printf("4. Cancel reservation\n");
        printf("5. Back to main menu\n");
        printf("Choose: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1: addPatient(); break;
            case 2: editPatient(); break;
            case 3: reserveSlot(); break;
            case 4: cancelReservation(); break;
            case 5: break;
            default: printf("Invalid choice.\n");
        }
    } while (choice != 5);
}
```

Figure 17: Admin Mode Function



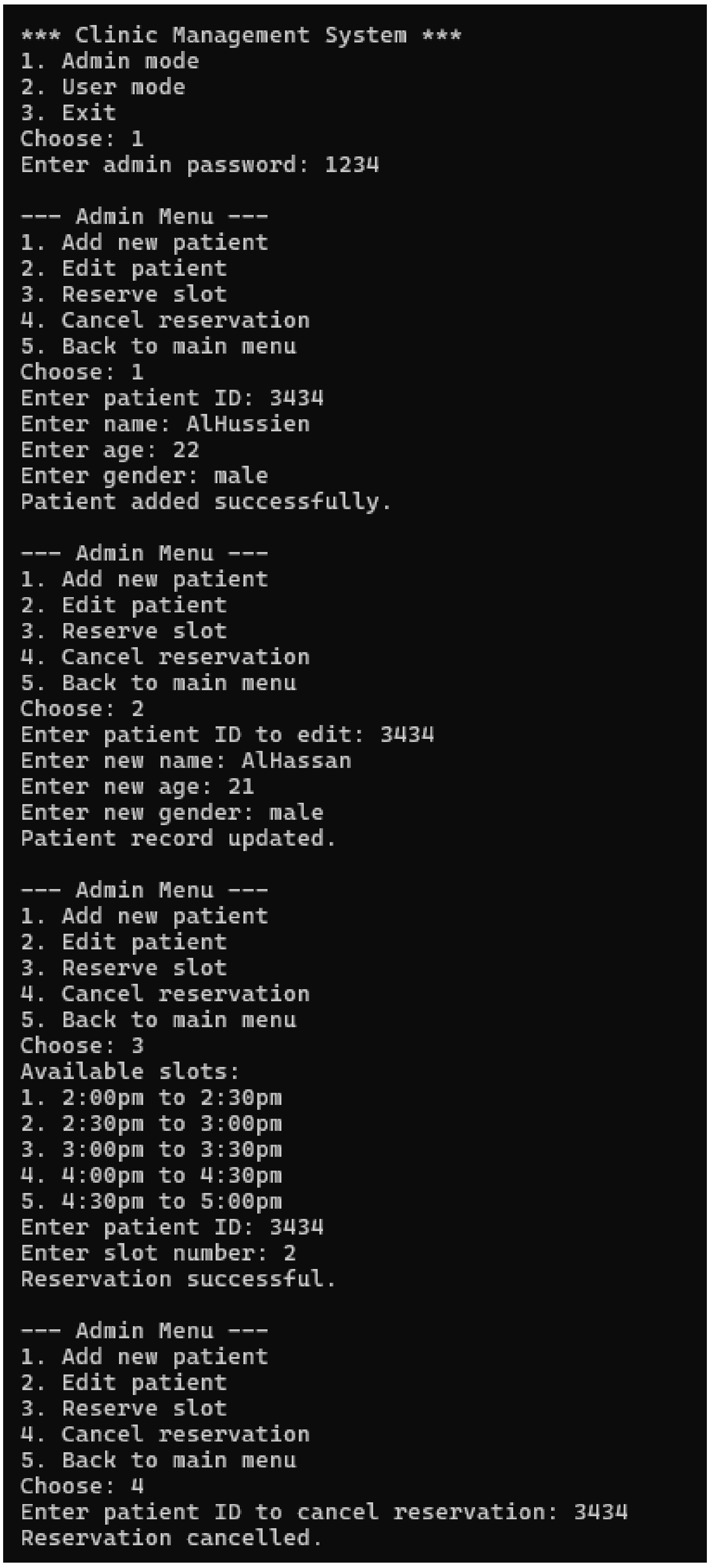
```c
// User functions
void viewPatient() {
    int id;
    printf("Enter patient ID: ");
    scanf("%d", &id);

    Patient* p = findPatient(id);
    if (!p) {
        printf("Patient not found.\n");
        return;
    }

    printf("Patient Info:\n");
    printf("ID: %d, Name: %s, Age: %d, Gender: %s\n",
           p->id, p->name, p->age, p->gender);
}

void viewReservations() {
    printf("Today's reservations:\n");
    for (int i = 0; i < NUM_SLOTS; i++) {
        if (slots[i] == 0) {
            printf("%s --> Available\n", slotTimes[i]);
        } else {
            printf("%s --> Reserved (ID: %d)\n", slotTimes[i], slots[i]);
        }
    }
}
```

User Function Code



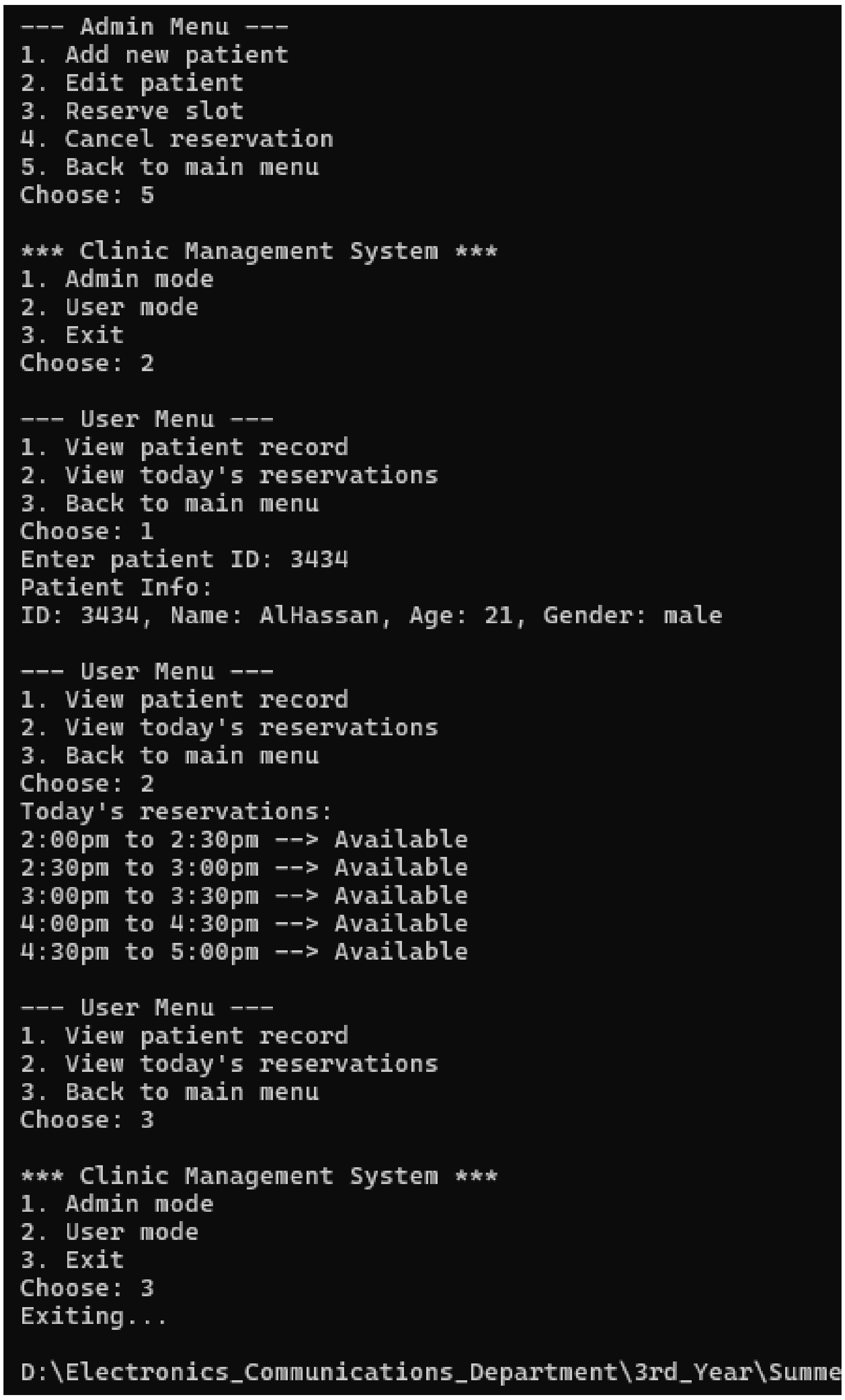
```c
void userMode() {
    int choice;
    do {
        printf("\n--- User Menu ---\n");
        printf("1. View patient record\n");
        printf("2. View today's reservations\n");
        printf("3. Back to main menu\n");
        printf("Choose: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1: viewPatient(); break;
            case 2: viewReservations(); break;
            case 3: break;
            default: printf("Invalid choice.\n");
        }
    } while (choice != 3);
}
```

User Mode Function



```
*** Clinic Management System ***
1. Admin mode
2. User mode
3. Exit
Choose: 1
Enter admin password: 1234

--- Admin Menu ---
1. Add new patient
2. Edit patient
3. Reserve slot
4. Cancel reservation
5. Back to main menu
Choose: 1
Enter patient ID: 3434
Enter name: AlHussien
Enter age: 22
Enter gender: male
Patient added successfully.

--- Admin Menu ---
1. Add new patient
2. Edit patient
3. Reserve slot
4. Cancel reservation
5. Back to main menu
Choose: 2
Enter patient ID to edit: 3434
Enter new name: AlHassan
Enter new age: 21
Enter new gender: male
Patient record updated.

--- Admin Menu ---
1. Add new patient
2. Edit patient
3. Reserve slot
4. Cancel reservation
5. Back to main menu
Choose: 3
Available slots:
1. 2:00pm to 2:30pm
2. 2:30pm to 3:00pm
3. 3:00pm to 3:30pm
4. 4:00pm to 4:30pm
5. 4:30pm to 5:00pm
Enter patient ID: 3434
Enter slot number: 2
Reservation successful.

--- Admin Menu ---
1. Add new patient
2. Edit patient
3. Reserve slot
4. Cancel reservation
5. Back to main menu
Choose: 4
Enter patient ID to cancel reservation: 3434
Reservation cancelled.
```

Figure 18: Admin Mode Output



```
--- Admin Menu ---
1. Add new patient
2. Edit patient
3. Reserve slot
4. Cancel reservation
5. Back to main menu
Choose: 5

*** Clinic Management System ***
1. Admin mode
2. User mode
3. Exit
Choose: 2

--- User Menu ---
1. View patient record
2. View today's reservations
3. Back to main menu
Choose: 1
Enter patient ID: 3434
Patient Info:
ID: 3434, Name: AlHassan, Age: 21, Gender: male

--- User Menu ---
1. View patient record
2. View today's reservations
3. Back to main menu
Choose: 2
Today's reservations:
2:00pm to 2:30pm --> Available
2:30pm to 3:00pm --> Available
3:00pm to 3:30pm --> Available
4:00pm to 4:30pm --> Available
4:30pm to 5:00pm --> Available

--- User Menu ---
1. View patient record
2. View today's reservations
3. Back to main menu
Choose: 3

*** Clinic Management System ***
1. Admin mode
2. User mode
3. Exit
Choose: 3
Exiting...

D:\Electronics_Communications_Department\3rd_Year\Summe
```

Figure 19: User Mode Output

26.2 Project 2: Smart Home Automation System (Dual Microcontroller)

This project implements a sophisticated Smart Home Automation system utilizing two interconnected microcontrollers (MCUs) to manage home environment parameters, offering control via both a local Keypad/LCD interface (Manual) and a remote Bluetooth interface.

26.2.1 Core Hardware and Communication Protocols

The project successfully integrates all three core serial communication protocols of the AVR microcontroller family:

- **MCU Communication (MCU-to-MCU):** High-speed data exchange using the **SPI (Serial Peripheral Interface)** protocol.
- **Bluetooth Connectivity:** Wireless control commands are received via the **UART (Universal Asynchronous Receiver/Transmitter)** protocol.
- **Configuration Storage:** The system password is stored/retrieved from external non-volatile memory (EEPROM) using the **I^{2}C (Inter-Integrated Circuit)** protocol.

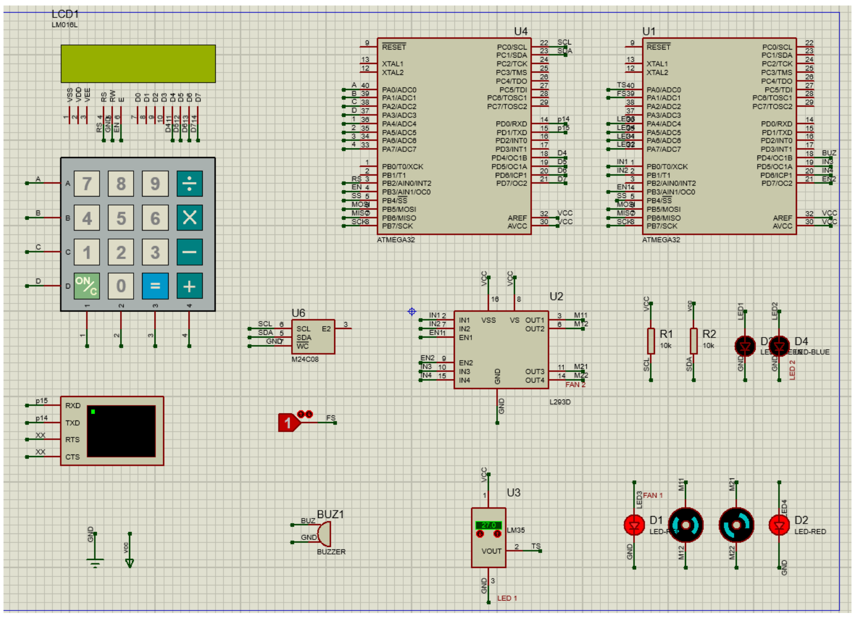


Figure 20: System Implementation

26.2.2 System Flow and Security

The system initiates with a robust security check:

1. **Password Entry and LCD Display:** User enters the password via Keypad.

2. **EEPROM Verification:** Password is compared against the default password stored in **I^{2}C EEPROM**.

3. **Security Breach Handling (Buzzer):** If the password is entered incorrectly for **3 consecutive attempts**, a **Buzzer** is activated and the system locks.

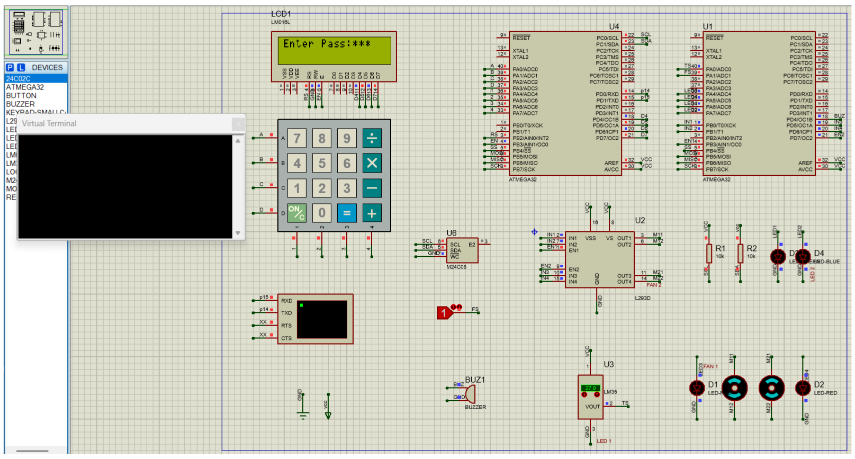


Figure 21: Password Writing

26.2.3 Operation Modes and Control Features

Upon successful login, the user selects between Manual (Keypad) control or Remote (Bluetooth/UART) control.

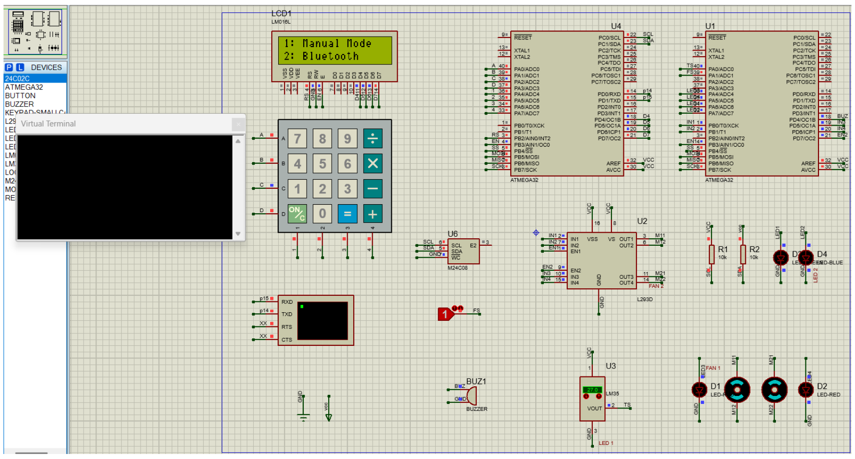


Figure 22: System Modes

Manual and Remote Control Menus The system provides two PWM control sub-menus:

1. **Lights (LEDs) Menu:** Allows the user to control the light brightness using **PWM (Pulse Width Modulation)**, enabling adjustable dimming levels.

2. **Fans (Motors) Menu:** Allows the user to manually control the **ON/OFF status** and **speed** (Low/Medium/High) of the cooling fans, also via PWM.

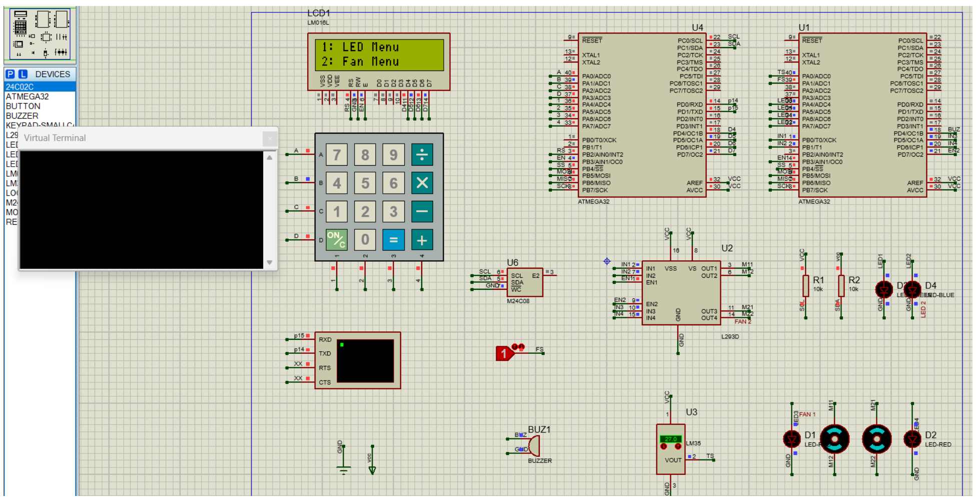


Figure 23: Modes Menu

26.2.4 Environmental Sensing and Automatic Control

The system utilizes sensors for autonomous environmental management:

1. **Temperature Sensor:** Reads the current room temperature.
2. **Smoke Sensor:** Monitors for potential fire or smoke hazards.
3. **Automatic Fan Speed Adjustment:** Based on temperature readings, the system automatically adjusts the fan speed using **PWM**.
4. **Emergency Fire Alert:** If the Smoke Sensor detects a dangerous level of smoke:
   - The **Buzzer** is immediately triggered.
   - A warning message (e.g., **"Wel3aaa!"**) is printed on the LCD screen.

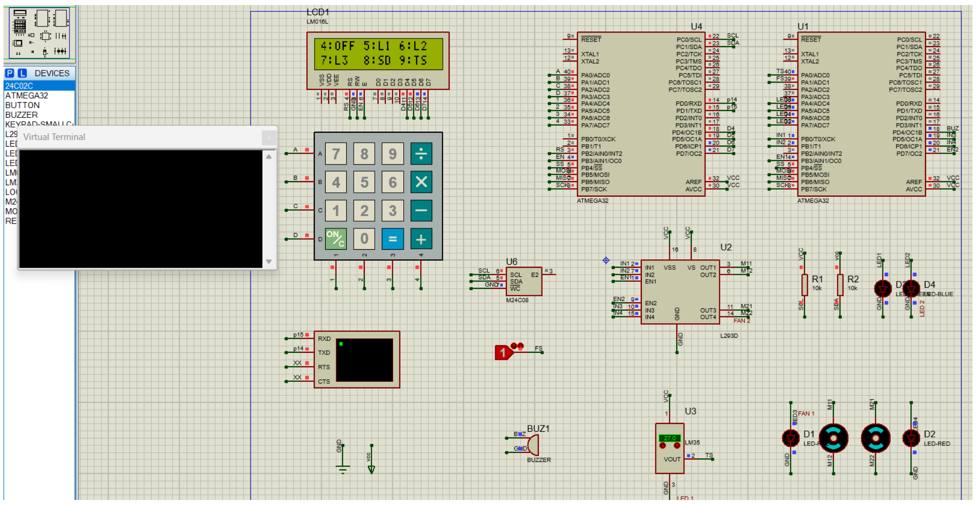
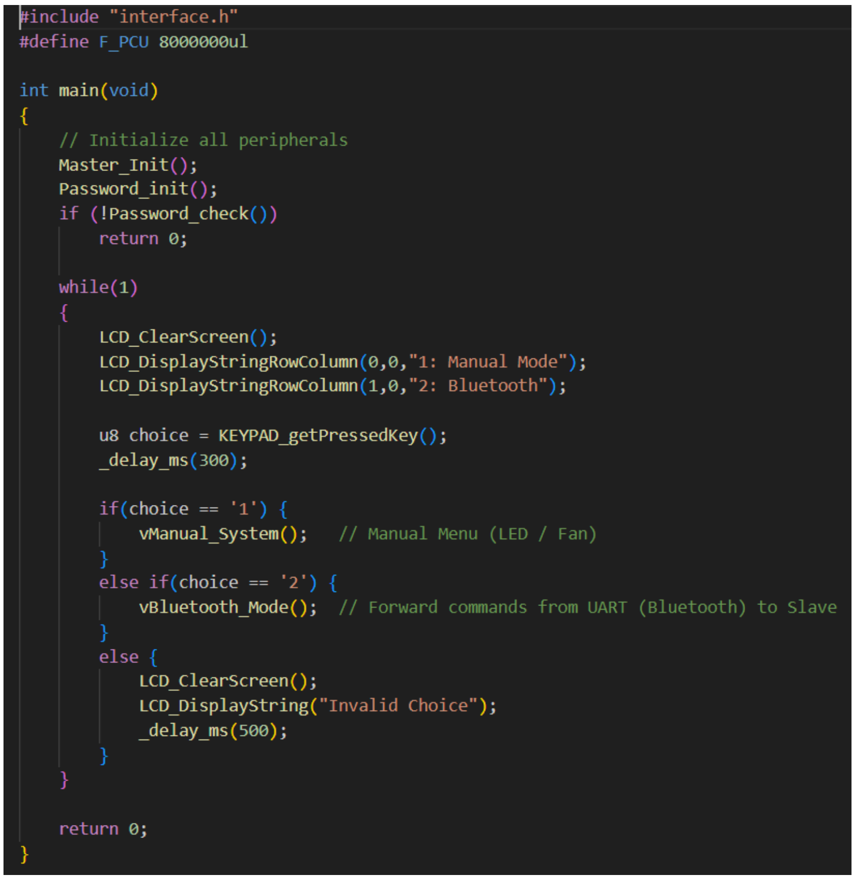


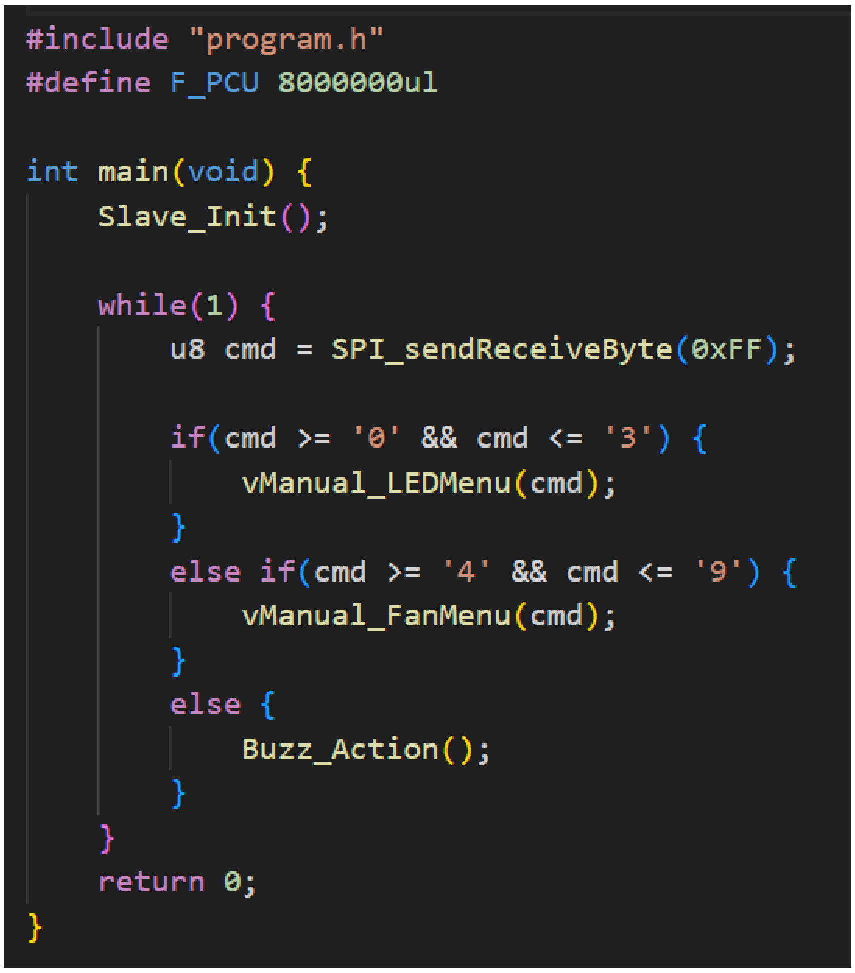
Figure 24: Fan Control

26.2.5 C Code Implementation



```c
#include "interface.h"
#define F_PCU 8000000ul

int main(void)
{
    // Initialize all peripherals
    Master_Init();
    Password_init();
    if (!Password_check())
        return 0;

    while(1)
    {
        LCD_ClearScreen();
        LCD_DisplayStringRowColumn(0,0,"1: Manual Mode");
        LCD_DisplayStringRowColumn(1,0,"2: Bluetooth");

        u8 choice = KEYPAD_getPressedKey();
        _delay_ms(300);

        if(choice == '1') {
            vManual_System();   // Manual Menu (LED / Fan)
        }
        else if(choice == '2') {
            vBluetooth_Mode();  // Forward commands from UART (Bluetooth) to Slave
        }
        else {
            LCD_ClearScreen();
            LCD_DisplayString("Invalid Choice");
            _delay_ms(500);
        }
    }

    return 0;
}
```

Figure 25: Master MC Code



```c
#include "program.h"
#define F_PCU 8000000ul

int main(void) {
    Slave_Init();

    while(1) {
        u8 cmd = SPI_sendReceiveByte(0xFF);

        if(cmd >= '0' && cmd <= '3') {
            vManual_LEDMenu(cmd);
        }
        else if(cmd >= '4' && cmd <= '9') {
            vManual_FanMenu(cmd);
        }
        else {
            Buzz_Action();
        }
    }
    return 0;
}
```

Figure 26: Slave MC Code

Features: Basic arithmetic operations, memory functions, and LCD feedback.

Future Work: Add scientific functions, improved UI, or external communication.

27 Conclusion

This report has provided a comprehensive survey of foundational and advanced concepts in C programming and their critical application within embedded systems, particularly focusing on the ATmega32 microcontroller family. We began by establishing a firm grounding in C syntax, flow control mechanisms (selection and loops), data organization, and memory manipulation.

27.1 Integration of Theory and Practice

The subsequent sections bridged the gap between theory and hardware, introducing core embedded concepts such as DIO control, layered architecture, and indispensable communication protocols (UART, SPI, and I2C). We further explored crucial aspects of the development pipeline:

- **Software Quality:** Addressing the importance of modularity, code robustness, and systematic verification through **Software Testing** methodologies.
- **Industry Relevance:** Understanding industry standards and domain-specific knowledge, exemplified by an overview of **Automotive Bus Technology**.
- **Development Environment:** Highlighting the role of compilers, debuggers, and cross-compilation within **Embedded Systems Tooling**.

27.2 Project Demonstration

Most significantly, the practical implementation showcased through the two detailed projects—the Clinic Management System and the Smart Home Automation System—demonstrates the power of integrating these theoretical foundations:

- **The Clinic System** highlighted the crucial role of modular programming and robust input validation in creating reliable console-based software.
- **The Smart Home Project** provided a complex, real-world scenario that successfully utilized PWM for motor control, implemented security protocols with EEPROM storage via **I2C**, and achieved high-speed inter-controller communication using **SPI** and remote control via **UART**.

In conclusion, the knowledge acquired in C programming, combined with a deep understanding of microcontroller peripherals, communication strategies, and the modern tooling environment, forms the bedrock for developing sophisticated, efficient, and reliable embedded solutions tailored to meet the stringent demands of modern electronic applications and industry quality standards.

28 References

1. ATmega32 Datasheet, Microchip Technology.
2. Mazidi, M. A., *AVR Microcontroller and Embedded Systems*, Pearson.
3. Information Technology Institute (ITI) Embedded Systems Materials, 2025.