ITE 1942 – ICT PROJECT

PROJECT REPORT

VEHICLE PARKING MANAGEMENT SYSTEM

Submitted by:

A.I.N.HATHARASINGHE

E2340065

Bachelor of Information Technology (External Degree)
Faculty of Information Technology
University of Moratuwa

CONTENT

FIGURES LIST

1. CHAPTER 01 – INTRODUCTION

1.1 Background and motivation

This project report, presented by A.I.N. Hatharasinghe, a student at the University of Moratuwa pursuing a Bachelor of Information Technology, outlines a comprehensive plan to address a key challenge in the field of IT. The proposed Vehicle Parking Management System aims to enhance efficiency and user experience by leveraging innovative technologies such as Visual Studio, C#, and an SQL database. The system will provide real-time parking availability, automate parking spot allocation, and implement access control for authorized vehicles, significantly reducing congestion and wait times. Through rigorous research and development, this project not only addresses operational inefficiencies but also contributes valuable insights and practical applications to the broader academic and professional community.

Currently, the manual parking management system presents several inefficiencies, delays, and security risks. Staff members and visitors often face challenges such as spending considerable time searching for available parking spaces, which leads to congestion at the main gate of the company. Additionally, the lack of a streamlined process results in unauthorized vehicles sometimes gaining access to restricted areas, posing security risks. Furthermore, the manual system lacks the ability to provide real-time information about parking availability, making it difficult for employees and visitors to plan their entry and exit efficiently.

The inefficiencies of the current manual system directly impact several stakeholders, including employees, visitors, and security staff. Employees often face delays in parking, which can reduce productivity and increase frustration. Visitors experience similar challenges, leading to potential negative impressions of the company's operations. Security staff are also burdened with manually checking vehicles, leading to increased workload and the potential for human error. The cumulative result is a chaotic and unorganized parking experience, contributing to lower satisfaction and higher operational costs.

Several existing solutions have been implemented in various organizations to address similar problems. These systems typically incorporate technologies such as RFID (Radio Frequency Identification) tags for vehicle identification, mobile applications for parking space reservation, and automated barriers for access control. While these solutions have shown some success, they often require expensive hardware installation and significant infrastructure changes. Moreover, they may not be fully integrated with real-time parking information systems, resulting in delays in updating parking availability. Some solutions also lack advanced features such as automated allocation of parking spaces based on vehicle type or specific user requirements (e.g., visitors vs. employees), further limiting their efficiency.

This project aims to overcome the limitations of existing solutions by developing a customized Vehicle Parking Management System designed to meet the specific needs of

the company. The system will be installed at the main gate and will automate key processes, including real-time parking availability updates, automated parking space allocation, and access control for authorized vehicles. By integrating an SQL database and using a user-friendly interface built with Visual Studio and C#, the system will provide seamless interaction for both users and administrators. It will allow security staff to monitor vehicle entry and exit efficiently, reducing the likelihood of unauthorized access.

In addition, the system will offer real-time data on parking availability, minimizing the time spent searching for spots and ensuring smooth traffic flow at the main gate. The automation of parking spot allocation will be based on predefined rules, ensuring that employees and visitors are assigned suitable spots efficiently. The system's built-in reporting capabilities will provide valuable insights into parking trends and usage patterns, enabling further optimization.

The key motivation behind this project is to enhance operational efficiency and employee convenience. By offering real-time parking information, the system will reduce search time, improve security by ensuring only authorized vehicles gain access, and ultimately increase productivity. Furthermore, the system aligns with environmental sustainability goals by minimizing vehicle idling and emissions, creating a more efficient, secure, and user-friendly parking experience for employees and visitors alike.

In conclusion, this project will address a critical operational inefficiency by implementing an automated and intelligent Vehicle Parking Management System. It will streamline parking operations at the main gate, providing a modern, secure, and efficient solution to the challenges posed by the existing manual system.

1.2 Problem In Brief

At company headquarters, managing parking spaces for staff vehicles has become an increasingly difficult task. With a large number of employees and a limited number of parking slots, the lack of an efficient system often results in confusion, delays, and frustration. The current manual method of managing parking is inefficient, and staff members frequently face difficulties in locating available spaces. This leads to wasted time and unnecessary stress as they search for a spot, especially during peak hours when demand for parking is at its highest. The absence of a system that provides real-time information at the main entrance only adds to the problem, causing congestion at the entry point and impeding the smooth flow of vehicles. Employees often have no way of knowing whether a space is available until they enter the parking lot, resulting in additional delays and further contributing to overcrowding.

The primary problem faced by the company is the inefficient management of parking spaces for staff vehicles. Due to the manual nature of the current system, staff members are not informed of the availability of parking spots before entering the premises. This leads to congestion at the main gate as employees try to find parking, often resulting in frustration and lost productivity. The lack of coordination and real-time updates causes delays, as employees must manually check each area of the parking lot to locate an available spot. Furthermore, the absence of any structured allocation or tracking system makes it difficult for management to effectively utilize the limited parking resources, leading to an uneven distribution of vehicles across the parking areas.

In addition to these challenges, there is no mechanism in place to prevent unauthorized use of parking spaces or to monitor the duration for which vehicles are parked. This can result in overuse of the available spaces, creating additional problems for both staff and management. Without an automated system to streamline the process, the company struggles to optimize parking space utilization, resulting in operational inefficiencies and a poor experience for employees.

To address these issues, the implementation of an automated Vehicle Parking Management System at the company's main gate is essential. This system will provide real-time information on available parking spaces as staff members arrive, allowing for efficient allocation of spots and reducing congestion at the entrance. The software will integrate with sensors or manual input to update the availability of parking slots in real time, ensuring that employees can quickly locate and occupy a space. This automated system will optimize the use of limited parking resources, minimize delays, and significantly improve the parking experience for staff members.

1.3 Aim & Objectives

The aim of this project is to develop a Vehicle Parking Management System for the company headquarters, designed to address the challenges of inefficient parking space utilization. This system will utilize Visual Studio, C#, and an SQL database to enable staff members to efficiently identify available parking spaces in real-time, thereby improving overall parking management and reducing the time spent searching for parking.

To achieve the stated aim, the following objectives will be accomplished:

- Develop a User-Friendly Interface: Create an intuitive interface that allows users to easily navigate the system and access parking information without difficulty.

- Real-time Monitoring of Parking Space Availability: Implement a mechanism for real-time monitoring that updates the status of parking spaces as vehicles enter or leave the parking area. This feature will ensure that staff members have immediate access to current parking availability.
- Design a Database: Create a comprehensive database to store and manage parking spot information, staff vehicle data, and parking availability. The database will be designed to ensure quick access and reliable data storage, facilitating seamless operations within the parking management system.
- Critical Analysis of System Requirements: Conduct a thorough analysis of system requirements to ensure that the developed solution meets the specific needs of the company headquarters. This analysis will guide the design and functionality of the system.
- Design and Develop a System for Solving the Problem: Utilize the insights gained from the requirements analysis to design and implement a robust Vehicle Parking Management System that effectively addresses the identified parking challenges.
- Evaluation of the Proposed System: Carry out rigorous testing of the system to ensure it operates smoothly under various conditions. The evaluation phase will include user feedback and adjustments to enhance system performance.
- Structured Deployment: After thorough testing and validation, deploy the system in a structured manner to ensure minimal disruption to operations during the transition.

By accomplishing these objectives, the Vehicle Parking Management System will significantly improve parking efficiency at the company headquarters, ultimately saving time and enhancing user satisfaction.

1.4 Summary

This chapter presents a comprehensive overview of the proposed Vehicle Parking Management System developed by A.I.N. Hatharasinghe at the University of Moratuwa. The motivation behind this project stems from the inefficiencies of the current manual parking management system, which leads to congestion, delays, and security risks at the company headquarters. Employees face significant challenges in locating available parking spaces, resulting in wasted time and frustration. Additionally, unauthorized vehicles occasionally gain access to restricted areas, further complicating the parking process.

The project aims to address these issues by implementing a software-based solution that leverages Visual Studio, C#, and an SQL database. The system will automate parking management, providing real-time information on parking availability, optimizing space allocation, and enhancing security through access control measures. Key objectives include developing a user-friendly interface, implementing real-time monitoring,

designing a comprehensive database, conducting a critical analysis of system requirements, and evaluating the proposed system before deployment.

In the next chapter, "Related Work," the discussion will focus on the challenges associated with the parking management problem and explore existing solutions in detail. This will include a critical analysis of various technologies and systems that have been implemented to address similar parking issues, highlighting their strengths and limitations, as well as how the proposed system aims to surpass them.

2. CHAPTER 02 – RELATED WORKS

2.1 Challenges

2.1.1 Inefficiency in Parking Management

The management of parking spaces is often hampered by outdated manual systems that rely heavily on human intervention. Staff members face significant challenges in locating available parking spots, particularly during peak hours. For example, companies without automated systems often experience congestion, as employees search for parking spaces while others attempt to enter or exit the lot. Research indicates that this inefficiency can lead to substantial time loss, resulting in increased operational costs and reduced employee productivity.

2.1.2 Lack of Real-Time Information

Many organizations lack the ability to provide real-time updates on parking space availability. Without an integrated system that can communicate this information promptly, employees are left uncertain about whether a parking space is free until they physically check. This uncertainty can contribute to traffic bottlenecks at entry points, as multiple vehicles may attempt to enter the lot simultaneously. Solutions that utilize mobile applications to communicate real-time availability have shown promise, yet many still fall short of comprehensive integration with existing infrastructure.

2.1.3 Security Risks

Manual parking systems often leave organizations vulnerable to security breaches. Unauthorized vehicles can gain access to restricted areas, posing potential threats to safety and security. Studies have highlighted instances where manual checks have failed, resulting in unauthorized access. Automated access control systems using technologies such as RFID have emerged as solutions to enhance security; however, they often require substantial investment in new hardware and training.

2.1.4 Limited Data Management and Reporting

Existing solutions frequently struggle with data management, resulting in inefficient reporting capabilities. Many systems do not provide the necessary tools to analyze parking trends and usage patterns effectively. Without robust reporting, management cannot make informed decisions regarding parking resource allocation and utilization. This lack of data oversight leads to missed opportunities for optimization, ultimately perpetuating inefficiencies.

2.1.5 Increased Environmental Impact

The inefficiencies of manual parking systems contribute to increased vehicle idling and emissions, negatively impacting environmental sustainability efforts. The lack of a streamlined process encourages prolonged searches for parking, leading to higher fuel consumption and greenhouse gas emissions. In light of growing environmental concerns, solutions that minimize idling and enhance traffic flow are essential for modern organizations.

2.2 Proposed Solution

2.2.1 Real-Time Parking Availability Monitoring

The Vehicle Parking Management System will integrate sensors or a manual input system to provide real-time updates on parking availability. This feature will ensure that employees receive immediate access to current information, thereby reducing search times and congestion at the entry point.

2.2.2 Automated Parking Spot Allocation

Automating the allocation of parking spots based on predefined rules will streamline the process for both employees and visitors. This system will consider factors such as vehicle type and user requirements (e.g., visitors vs. employees), thereby optimizing the use of available resources.

2.2.3 Vehicle Registration and Access Control

The proposed system will include a secure vehicle registration process to manage access control effectively. By integrating access control mechanisms, only authorized vehicles will be allowed entry, reducing security risks and enhancing overall safety.

2.2.4 Centralized Database Management

An SQL database will serve as the backbone of the system, facilitating efficient data management and retrieval. This centralized approach will allow for quick access to information regarding parking space availability, user registrations, and historical data.

2.2.5 Notifications and Alerts

The system will incorporate a notification mechanism to inform users about parking space availability, reservation confirmations, and other relevant updates. This proactive communication will enhance user experience and reduce confusion.

2.2.6 Reporting and Analytics

The built-in reporting capabilities will allow management to analyze parking trends and usage patterns over time. This data-driven approach will enable informed decision-making regarding resource allocation and optimization.

2.2.7 User-Friendly Interface

The system will feature an intuitive user interface designed to enhance user interaction. By simplifying navigation and access to parking information, the interface will improve overall user satisfaction.

2.3 Summary

The development of the Vehicle Parking Management System aims to address the various challenges associated with manual parking management through innovative solutions. By implementing real-time monitoring, automated allocation, and enhanced security measures, this system will significantly improve parking efficiency, reduce congestion, and contribute positively to the organization's overall operational performance.

3. CHAPTER 03 - SOLUTION

This chapter delves into a comprehensive analysis of the Vehicle Parking Management System (VPMS) to identify and articulate the functional and non-functional requirements necessary for addressing the key challenges of inefficient parking management at the company headquarters. By analyzing each aspect of the proposed system, we aim to ensure that all critical requirements are adequately captured, paving the way for a successful implementation.

3.1. System Requirements

The Vehicle Parking Management System's requirements are essential for creating a comprehensive solution that addresses the challenges of inefficient parking management at the company headquarters. These requirements can be categorized into two main types: functional and non-functional.

Functional Requirements define the specific features and functionalities that the system must support to ensure effective operations. One of the primary functional requirements is real-time monitoring of parking space availability. This feature enables users to access up-to-date information about available parking spots, reducing the time spent searching for spaces and minimizing congestion at entry points. Additionally, the system must incorporate an automated parking spot allocation mechanism. This allows for efficient assignment of parking spaces based on predefined criteria, such as vehicle type or user role (e.g., employee or visitor), thereby optimizing the use of available resources.

Another critical functional requirement is the management of vehicle registration and access control. The system should allow for the registration of vehicles and provide access control to ensure that only authorized vehicles can enter the premises. This enhances security and prevents unauthorized access to restricted areas. Furthermore, the system should support robust reporting and analytics capabilities, allowing management to generate insights into parking trends, usage patterns, and overall system performance. This data will aid in decision-making and further optimization of parking operations.

Non-functional Requirements, on the other hand, pertain to the performance and usability aspects of the system. A crucial non-functional requirement is the need for a user-friendly interface. The system must be intuitive and easy to navigate, ensuring that both employees and visitors can quickly access the necessary information without technical difficulties. This is vital for enhancing user satisfaction and ensuring efficient interactions with the system.

The system's integration with an SQL database is another significant requirement. This integration will provide reliable data storage and management capabilities, ensuring that parking information is stored securely and can be retrieved quickly. Scalability is also a non-functional requirement; the system should be designed to accommodate future growth,

allowing for the addition of new features or an increase in the number of users and vehicles without compromising performance.

3.2. Functional Requirements

- Real-time Parking Availability: The system continuously monitors and displays the availability of parking spaces in real-time. This feature ensures that both staff and visitors are aware of the current status of the parking lot before entering the premises. It minimizes congestion at the main gate by providing up-to-date information, allowing users to decide whether to enter or seek alternative parking options, thus saving time and reducing frustration.
- Parking Spot Registration and Allocation**:** The system allows users to register and allocate parking spots based on specific criteria, such as vehicle type (car, motorcycle, etc.) or user status (employee or visitor). This automated allocation ensures efficient use of space by assigning suitable spots to users based on their needs. It also eliminates manual processes, improving the overall efficiency and organization of parking operations.
- Vehicle Registration**:** For efficient access control and parking monitoring, the system includes a vehicle registration feature. Users must register their vehicles in the system, which enables tracking of vehicle entry and exit. It also helps manage access, ensuring that only registered vehicles are allowed into specific areas, thereby increasing security and optimizing space usage.
- Profile Registration: Profile registration allows the system to personalize user experiences and manage user-specific settings. This feature enables individual users, whether employees or visitors, to create profiles with details such as vehicle type, preferred parking spots, and access permissions. Personalization improves user convenience and the system's ability to manage parking allocations effectively.
- Entries Management: The system records and manages vehicle entries, capturing details like the vehicle’s entry time and identification. This data helps in monitoring the use of parking spaces and ensures that the system updates parking availability in real-time. By efficiently managing vehicle entries, the system reduces delays and ensures smooth traffic flow within the parking area.
- Exit Management: Like entries, the system handles vehicle exits by updating the availability of parking spaces in real-time as vehicles leave. Exit management is crucial for maintaining accurate parking availability and ensuring that the system reflects the current state of the parking lot. It also helps in generating accurate reports on parking usage.
- System Reset: To maintain operational continuity, an administrative feature for resetting the system is essential. In the event of technical issues or operational errors, the reset function allows administrators to address problems quickly without interrupting the system's functionality. This ensures minimal downtime and enhances the system's reliability.

- Reporting and Analytics: The system provides robust analytical tools that generate reports on parking usage, trends, and occupancy rates. These reports assist management in making informed decisions regarding parking space utilization, operational improvements, and future planning. Analytics also offer insights into peak usage times, enabling better allocation strategies and resource optimization.

3.3. Non-Functional Requirements

- Scalability: This feature ensures that the system can accommodate an increasing number of users and vehicles without performance degradation. Scalability allows the application to handle growth—whether by adding more users or vehicles—while maintaining efficient operations, ensuring that the system remains responsive even under high demand.
- Security: Security measures are essential to protect sensitive user and vehicle data from unauthorized access or breaches. This includes implementing secure authentication mechanisms, data encryption, and access controls to ensure that only authorized personnel can interact with the system, thereby safeguarding personal and vehicle information.
- Performance: The system should deliver real-time updates with minimal latency, allowing users to receive timely information regarding parking availability and registration processes. High performance is critical for user satisfaction, as delays can lead to frustration and inefficiencies in parking management.
- Reliability: A reliable system is one that consistently performs its intended functions without failure. This feature ensures that the parking management system remains operational with minimal downtime, allowing users to access the service whenever needed. Reliability can be achieved through robust architecture and thorough testing.
- Usability: A user-friendly interface is crucial for enhancing user experience. The system should have intuitive navigation, clear instructions, and easily accessible features, making it simple for users of varying technical backgrounds to interact with the system effectively and efficiently.
- Maintainability: This feature allows for easy updates, enhancements, and bug fixes within the system. A maintainable system is designed to simplify the process of making changes, ensuring that it can evolve with user needs and technological advancements while minimizing disruptions to service.
- Efficiency: The system should be optimized to minimize resource consumption, including memory, processing power, and network bandwidth. Efficiency improves overall system performance and reduces operational costs, making the application more sustainable in the long run. Efficient systems also enhance user experience by providing quicker responses and reducing waiting times.

3.4. Summary

In summary, the systems analysis for the Vehicle Parking Management System outlines a comprehensive framework that incorporates both functional and non-functional

requirements. This framework ensures that the system not only addresses the immediate challenges of inefficient parking management but also aligns with the company's long-term operational goals. By leveraging technologies such as Visual Studio, C#, and SQL, the project aims to deliver a robust solution that enhances the parking experience for employees and visitors, while providing management with valuable insights into parking utilization. Through rigorous analysis and a focus on user-centric design, the Vehicle Parking Management System is poised to create a more organized, efficient, and secure parking environment.

4. CHAPTER 04 – SYSTEM DESIGN

4.1. Flow Chart

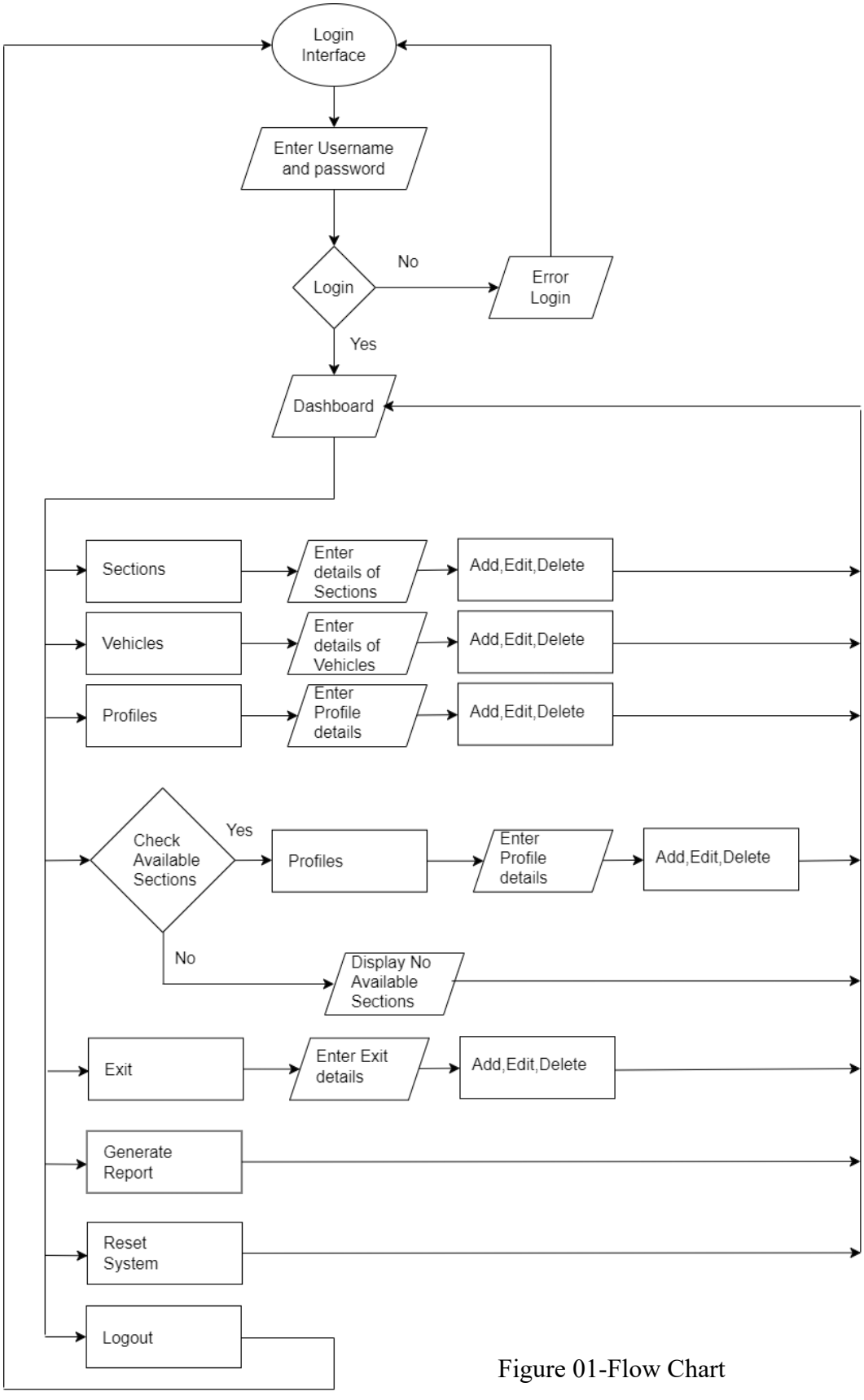


Figure 01-Flow Chart

4.2. Pseudocode

1.Login Interface

Start

Display "Enter Username and Password"

Input Username

Input Password

If Username and Password are correct:

    Go to Dashboard

Else:

    Display "Error: Invalid Login"

    Return to Login Interface

2.Dashboard

Display Dashboard options:

    1. Sections

    2. Vehicles

    3. Profiles

    4. Check Available Sections

    5. Exit

    6. Generate Report

    7. Reset System

    8. Logout

User selects an option:

    If user selects "Sections":

        Go to Sections Module

    If user selects "Vehicles":

        Go to Vehicles Module

    If user selects "Profiles":

        Go to Profiles Module

If user selects "Check Available Sections":

Go to Check Available Sections Module

If user selects "Exit":

Exit the system

If user selects "Generate Report":

Generate Report Module

If user selects "Reset System":

Reset System Module

If user selects "Logout":

Logout

3.Sections Module

Display "Enter Details of Sections"

Input Section Details

Display "Do you want to Add, Edit, or Delete Section?"

If "Add" selected:

Add Section to Database

Display "Section added successfully"

If "Edit" selected:

Display available Sections

Input Section to edit

Update Section details in Database

Display "Section updated successfully"

If "Delete" selected:

Display available Sections

Input Section to delete

Delete Section from Database

Display "Section deleted successfully"

Return to Dashboard

4.Vehicles Module

Display "Enter Details of Vehicles"

Input Vehicle Details

Display "Do you want to Add, Edit, or Delete Vehicle?"

If "Add" selected:

  Add Vehicle to Database

  Display "Vehicle added successfully"

If "Edit" selected:

  Display available Vehicles

  Input Vehicle to edit

  Update Vehicle details in Database

  Display "Vehicle updated successfully"

If "Delete" selected:

  Display available Vehicles

  Input Vehicle to delete

  Delete Vehicle from Database

  Display "Vehicle deleted successfully"

Return to Dashboard

5.Profiles Module

Display "Enter Profile Details"

Input Profile Details

Display "Do you want to Add, Edit, or Delete Profile?"

If "Add" selected:

  Add Profile to Database

  Display "Profile added successfully"

If "Edit" selected:

  Display available Profiles

  Input Profile to edit

Update Profile details in Database

Display "Profile updated successfully"

If "Delete" selected:

Display available Profiles

Input Profile to delete

Delete Profile from Database

Display "Profile deleted successfully"

Return to Dashboard

6.Check Available Sections Module

Check available sections in the parking lot

If sections are available:

Display "Available Sections"

Display List of Available Sections

Else:

Display "No Available Sections"

Return to Dashboard

7.Exit Module

Display "Enter Exit Details"

Input Exit Details

Display "Do you want to Add, Edit, or Delete Exit Details?"

If "Add" selected:

Add Exit Details to Database

Display "Exit details added successfully"

If "Edit" selected:

Display available Exit records

Input Exit details to edit

Update Exit details in Database

Display "Exit details updated successfully"

If "Delete" selected:

Display available Exit records

Input Exit details to delete

Delete Exit details from Database

Display "Exit details deleted successfully"

Return to Dashboard

8.Generate Report Module

Generate a detailed report of parking activities

Display "Report Generated Successfully"

Return to Dashboard

9.Reset System Module

Confirm "Do you want to reset the system? (Y/N)"

If "Y":

Reset all data and configurations

Display "System reset successfully"

If "N":

Return to Dashboard

10.Logout Module

Log the user out

Display "You have successfully logged out"

Return to Login Interface

4.3. Summary

This chapter presents the system design through detailed flowcharts and pseudocodes for each module of the Vehicle Parking Management System. The flowcharts provide a high-level visual representation of the system's behavior, while the pseudocode details the step-by-step logic for user interactions and system functions. Together, these components ensure that the system's architecture is robust, scalable, and easy to maintain. The pseudocode covers all the core functionalities, from login authentication to section management, vehicle registration, profile updates, report generation, and system reset, making sure that every functional requirement is adequately addressed.

5. CHAPTER 05 – SYSTEM IMPLEMENTATION

The implementation of the Vehicle Parking Management System was carried out using SQL for database management, C# for backend processing, and Visual Studio as the development environment. The system was designed to help manage parking spaces efficiently by allowing users to view available spots, handle vehicle entries and exits, and update parking statuses in real-time. Below is a detailed explanation of the primary user interfaces and their functionalities, along with an overview of key processes.

5.1. User Interfaces

5.1.1. Login Interface

The login interface serves as the entry point to the system, providing a secure authentication mechanism. Users are required to input their credentials to access the system. This ensures that only authorized personnel can manage or view parking-related data.

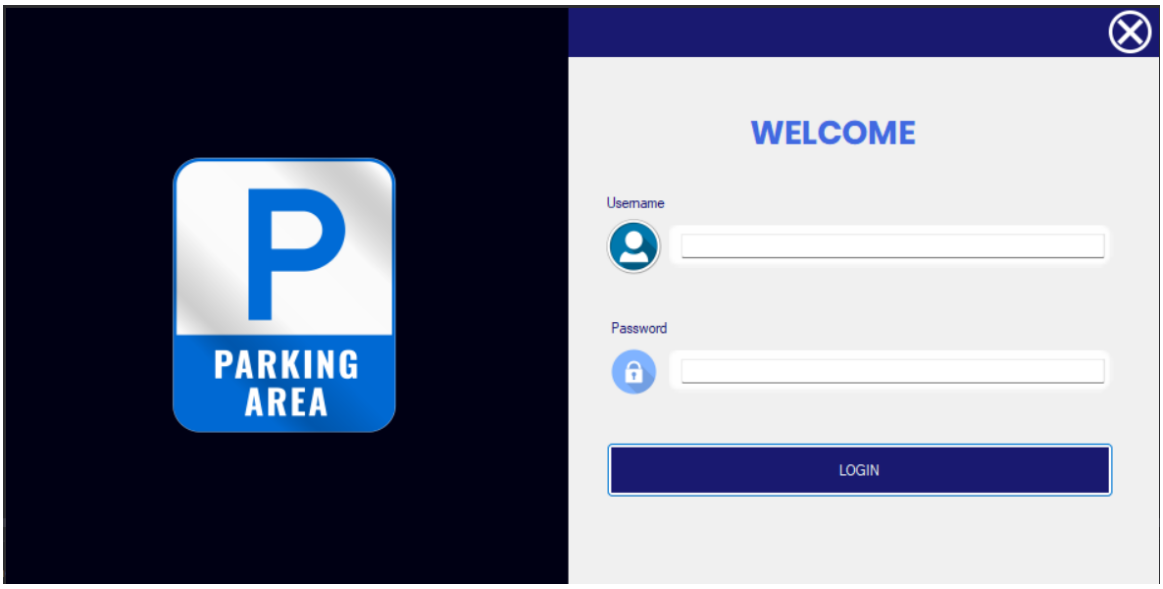


Figure 02-Login Interface

5.1.2. Dashboard

The dashboard provides an overview of the system's status, including the number of available parking spaces, currently occupied spots, and other essential information. It serves as the control center where users can navigate through the various sections of the system.

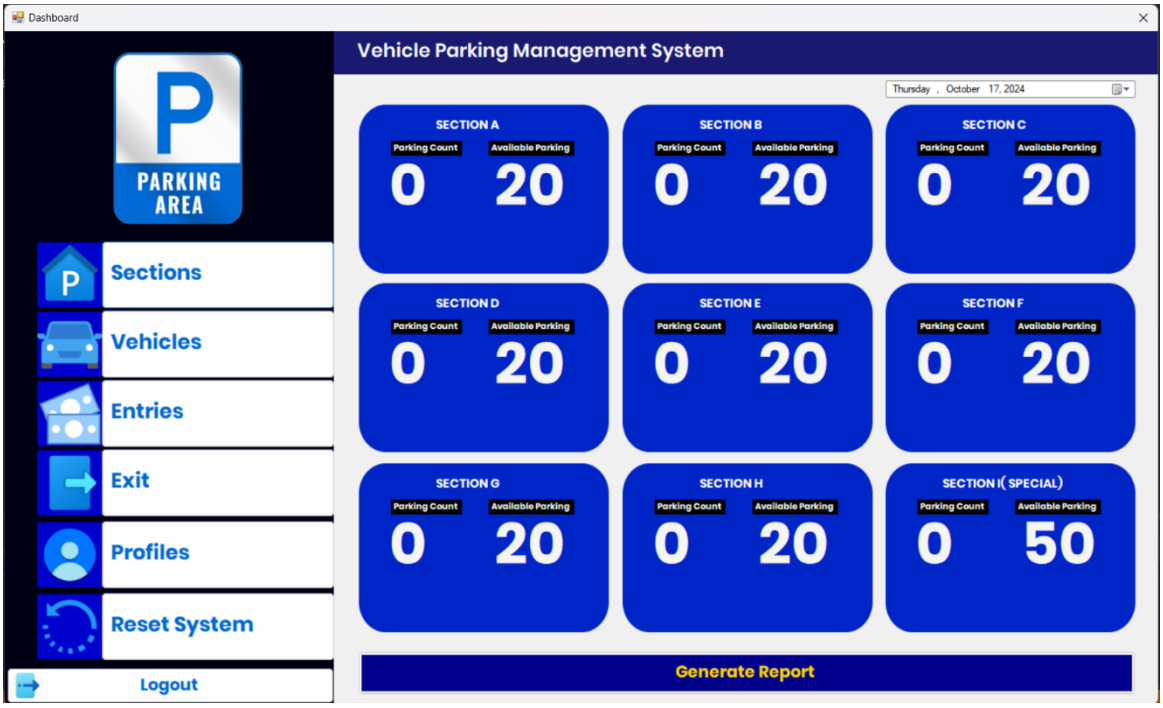


Figure 03-Dashboard Interface

5.1.3. Sections

This section allows for the configuration and management of different parking areas within the facility. It enables administrators to categorize parking spaces based on location or vehicle type (e.g., employee, visitor).

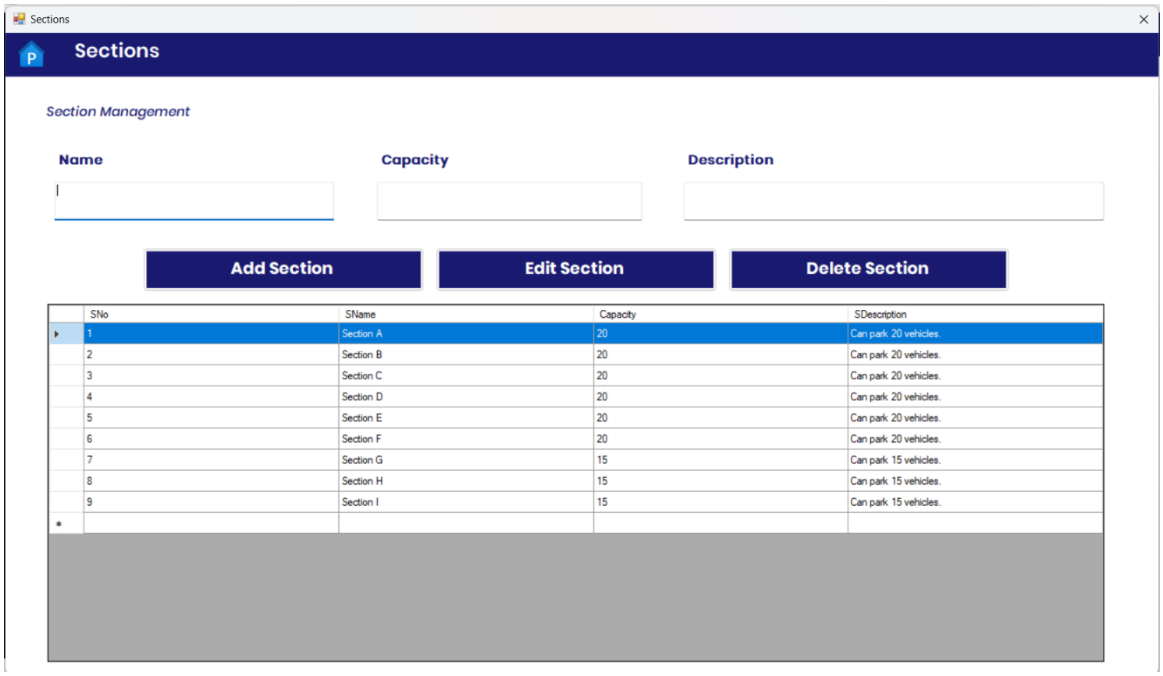
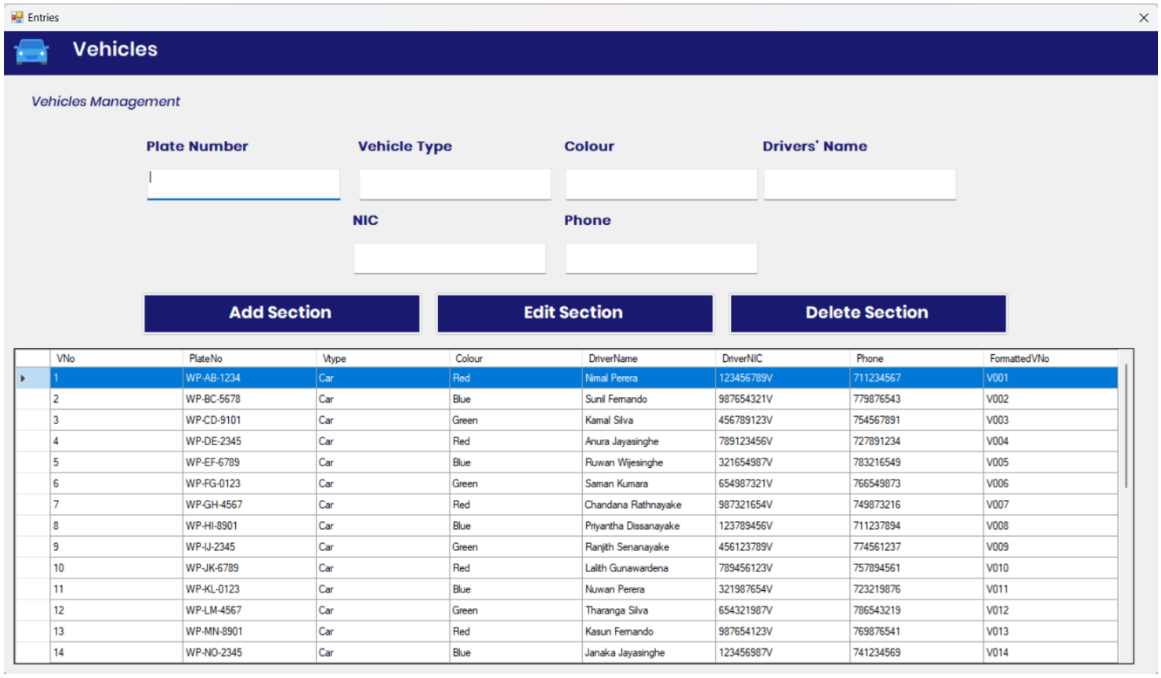


Figure 04-Sections Interface

5.1.4. Vehicles

This interface allows users to register and manage vehicle details. It includes fields for vehicle numbers, types, and owners, which are stored in the SQL database for easy access and retrieval.



VNo	PlateNo	Vtype	Colour	DriverName	DriverNIC	Phone	FormattedVNo
1	WP-AB-1234	Car	Red	Nimal Perera	123456789V	711234567	V001
2	WP-BC-5678	Car	Blue	Sunil Fernando	987654321V	779876543	V002
3	WP-CD-9101	Car	Green	Kamal Silva	456789123V	754567891	V003
4	WP-DE-2345	Car	Red	Anura Jayasinghe	789123456V	727891234	V004
5	WP-EF-6789	Car	Blue	Ruwan Wijesinghe	321654987V	783216549	V005
6	WP-FG-0123	Car	Green	Saman Kumara	654987321V	766549873	V006
7	WP-GH-4567	Car	Red	Chandana Rathnayake	987321654V	749873216	V007
8	WP-HI-8901	Car	Blue	Priyantha Dissanayake	123789456V	711237894	V008
9	WP-IJ-2345	Car	Green	Ranjith Senanayake	456123789V	774561237	V009
10	WP-JK-6789	Car	Red	Lalith Gunawardena	789456123V	757894561	V010
11	WP-KL-0123	Car	Blue	Nuwan Perera	321987654V	723219876	V011
12	WP-LM-4567	Car	Green	Tharanga Silva	654321987V	786543219	V012
13	WP-MN-8901	Car	Red	Kasun Fernando	987654123V	769876541	V013
14	WP-NO-2345	Car	Blue	Janaka Jayasinghe	123456987V	741234569	V014

Figure 05-Vehicles Interface

5.1.5. Profiles

The profile management feature allows users to maintain and update their information. Each user has a dedicated profile, and administrators can manage access levels, ensuring the system operates with the appropriate level of security.

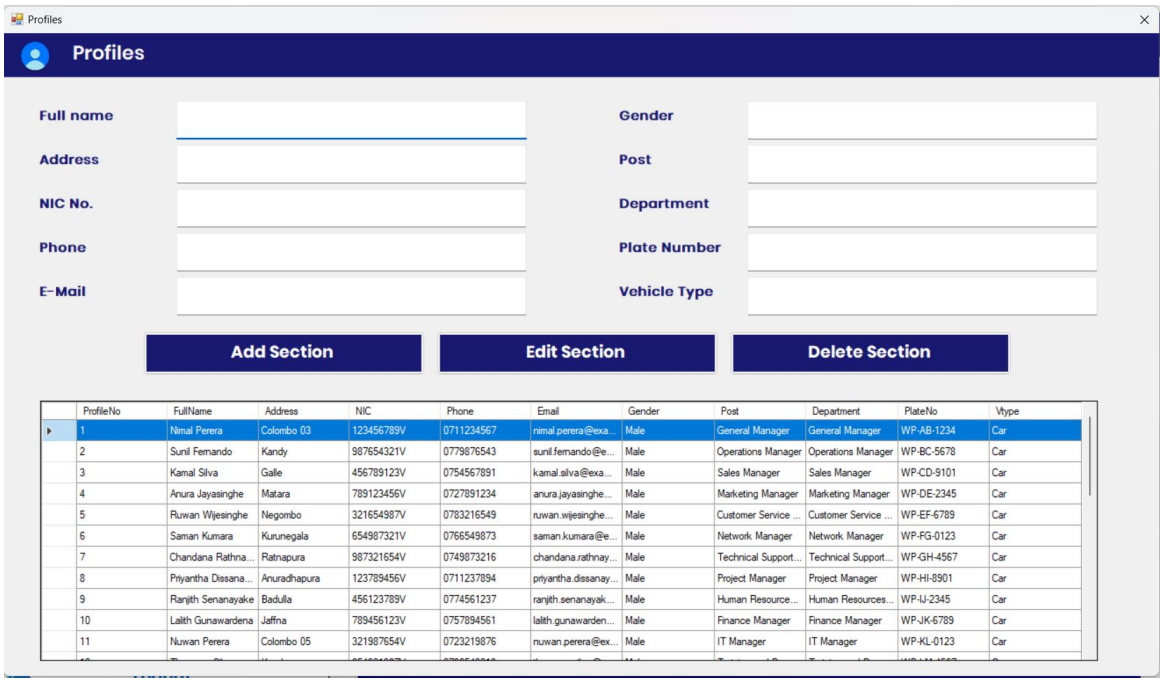


Figure 06-Profiles Interface

5.1.6. Entries

The entry management system tracks vehicles entering the parking facility. It logs the vehicle's details, including time of entry, and allocates a parking spot, updating the availability in the system.

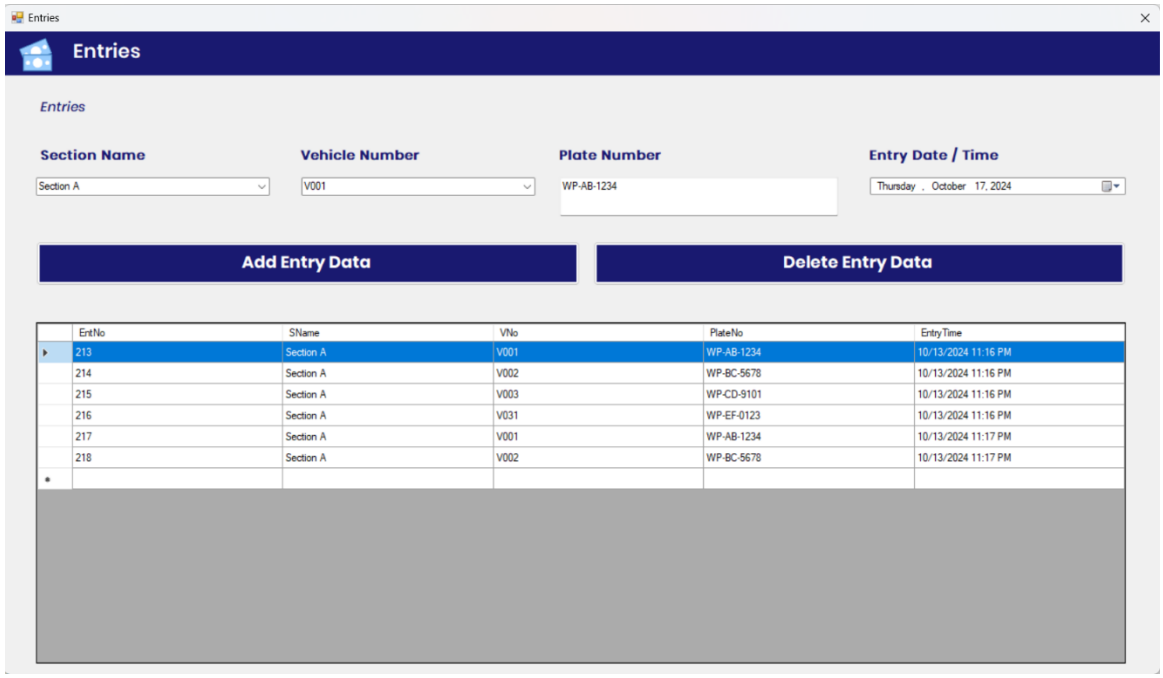


Figure 07-Entries Interface

5.1.7. Exit

Similar to the entry system, the exit interface records the time a vehicle leaves the parking facility. Upon exit, the system frees up the corresponding parking spot, making it available for future use.

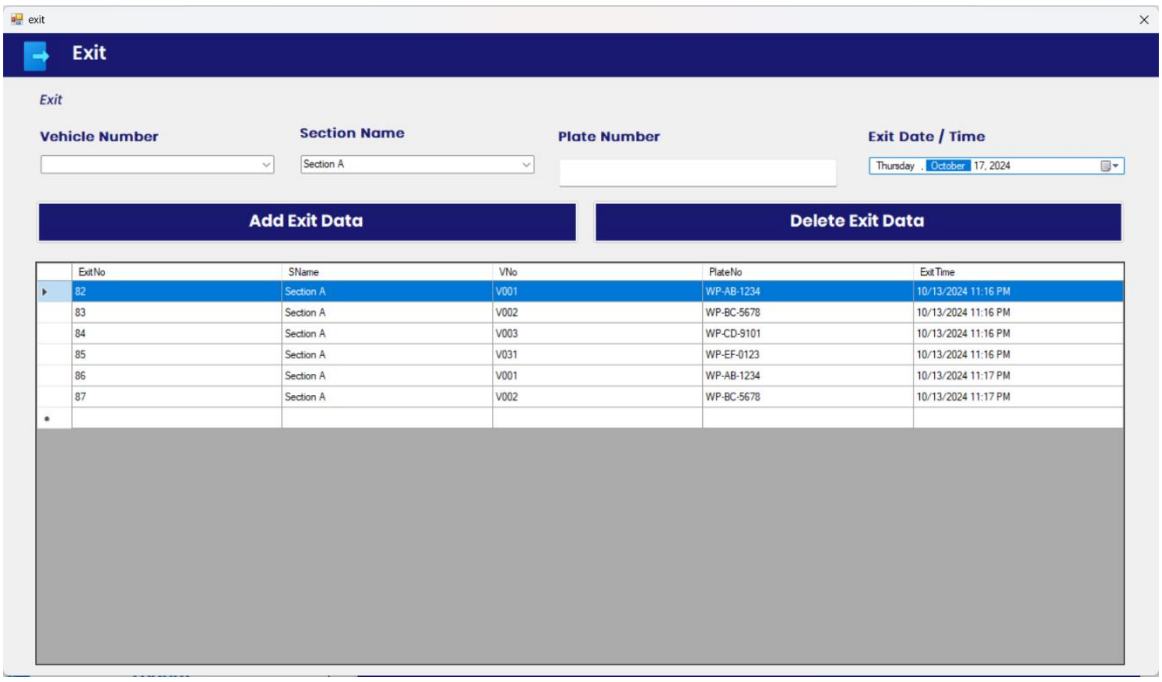


Figure 08-Exit Interface

5.2. Code Implementation

Class Definition and Constructor

```
class Functions{

        private SqlConnection Con;

        private SqlCommand Cmd;

        private string ConStr;

public Functions(){

ConStr=@"Data Source = (LocalDB)\v11.0;AttachDbFilename =
C:\Users\ISHAN\Desktop\BIT Project\Vehicle Parking Management System
Project\Vehicle Parking Management System Project\parkingMSDB.mdf;Integrated
Security=True";
```

```
Con = new SqlConnection(ConStr);
Cmd = new SqlCommand(); Cmd.Connection = Con;
  }
 }
```

This section defines the Functions class with three private members: Con, Cmd, and ConStr. Con is a SqlConnection object used to establish a connection to the database. Cmd is a SqlCommand object for executing SQL queries. The constructor initializes the ConStr variable with a connection string pointing to the database file on the local system. It then creates a new SqlConnection and associates it with the SqlCommand.

GetData Method

```
public DataTable GetData(string Query, SqlParameter[] parameters = null)
{
   DataTable dt = new DataTable();
   using (SqlConnection con = new SqlConnection(ConStr))
   {
      using (SqlCommand cmd = new SqlCommand(Query, con))
      {
         if (parameters != null)
         {
            cmd.Parameters.AddRange(parameters);
         }
         try
         {
            con.Open();
            SqlDataAdapter sda = new SqlDataAdapter(cmd);
            sda.Fill(dt);
         }
```

```
        catch (Exception ex)
        {
            Console.WriteLine("Error retrieving data: " + ex.Message); // Log errors
            throw; // Re-throw the exception for further handling if needed
        }
    }
  }
  return dt;
}
```

The GetData method is designed to execute a SQL query and return the results as a DataTable. It takes a query string and an optional array of SqlParameter objects for parameterized queries. A new SqlConnection is created within a using statement to ensure proper resource management. If parameters are provided, they are added to the SqlCommand. The method attempts to open the connection and fill a DataTable with the results using a SqlDataAdapter. If an error occurs, it is logged, and the exception is re-thrown for further handling.

SetData Method

```
public int SetData(string Query, SqlParameter[] parameters = null)
{
  int Cnt = 0;
  using (SqlConnection con = new SqlConnection(ConStr))
  {
    using (SqlCommand cmd = new SqlCommand(Query, con))
    {
      if (parameters != null)
      {
        cmd.Parameters.AddRange(parameters);
      }

      try
```

```
            {
                con.Open();
                Cnt = cmd.ExecuteNonQuery();
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error executing command: " + ex.Message); // Log
errors
                throw; // Re-throw the exception for further handling if needed
            }
        }
    }
    return Cnt;
}
```

The SetData method handles data modification operations such as insertions, updates, or deletions. Similar to GetData, it establishes a new SqlConnection and prepares a SqlCommand using the provided query and parameters. It opens the connection and executes the command using ExecuteNonQuery, which returns the number of affected rows. Any errors encountered during execution are logged, and the exception is re-thrown.

Fetch Combined Data from Multiple Tables

```
public DataTable GetCombinedData()
{
    DataTable combinedData = new DataTable();
    try
    {
        string query = @"
            SELECT
                e.VNo AS 'Vehicle Number',
                e.PlateNo AS 'Plate Number',
                c.VType AS 'Vehicle Type',
```

```
            c.DriverName AS 'Driver Name',
            c.Phone AS 'Phone',
            e.EntryTime AS 'Entry Time',
            x.ExitTime AS 'Exit Time'
        FROM
            EntryTbl e
        LEFT JOIN
            ExitTbl x ON e.VNo = x.VNo
        LEFT JOIN
            CarTbl c ON e.PlateNo = c.PlateNo
        ORDER BY
            e.EntryTime";

        combinedData = GetData(query);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error retrieving combined data: " + ex.Message);
        throw;
    }
    return combinedData;
}
```

GetCombinedData retrieves data by joining three tables: EntryTbl, ExitTbl, and CarTbl, based on vehicle number and plate number. The query selects information such as vehicle number, plate number, vehicle type, driver details, entry time, and exit time. The method uses the GetData function to fetch this combined data into a DataTable. It is designed for cases where related data from multiple tables is needed in a single result set.

5.3. Inputs

SectionTbl – Use to add data of the sections

- Section Number
- Section Name
- Section Capacity
- Section Description

SNo	SName	Capacity	SDescription
1	Section A	20	Can park 20 vehicles.
2	Section B	20	Can park 20 vehicles.
3	Section C	20	Can park 20 vehicles.
4	Section D	20	Can park 20 vehicles.
5	Section E	20	Can park 20 vehicles.
6	Section F	20	Can park 20 vehicles.
7	Section G	15	Can park 15 vehicles.
8	Section H	15	Can park 15 vehicles.
9	Section I	15	Can park 15 vehicles.

Figure 09-SectionTbl

CarTbl – Use to add data of the Vehicles

- Vehicle Number
- Vehicle Plate Number
- Vehicle Type
- Vehicle Color
- Vehicle Driver Name
- Vehicle Driver NIC
- Phone

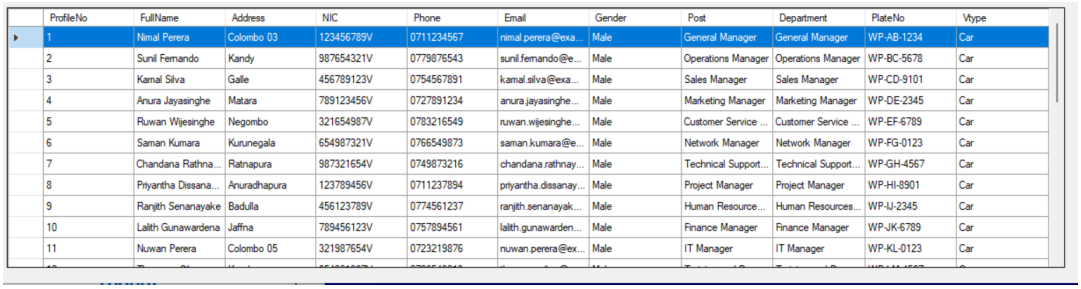
VNo	PlateNo	Vtype	Colour	DriverName	DriverNIC	Phone	FormattedVNo
1	WP-AB-1234	Car	Red	Nimal Perera	123456789V	711234567	V001
2	WP-BC-5678	Car	Blue	Sunil Fernando	987654321V	779876543	V002
3	WP-CD-9101	Car	Green	Kamal Silva	456789123V	754567891	V003
4	WP-DE-2345	Car	Red	Anura Jayasinghe	789123456V	727891234	V004
5	WP-EF-6789	Car	Blue	Ruwan Wijesinghe	321654987V	783216549	V005
6	WP-FG-0123	Car	Green	Saman Kumara	654987321V	766549873	V006
7	WP-GH-4567	Car	Red	Chandana Rathnayake	987321654V	749873216	V007
8	WP-HI-8901	Car	Blue	Priyantha Dissanayake	123789456V	711237894	V008
9	WP-IJ-2345	Car	Green	Ranjith Senanayake	456123789V	774561237	V009
10	WP-JK-6789	Car	Red	Lalith Gunawardena	789456123V	757894561	V010
11	WP-KL-0123	Car	Blue	Nuwan Perera	321987654V	723219876	V011
12	WP-LM-4567	Car	Green	Tharanga Silva	654321987V	786543219	V012
13	WP-MN-8901	Car	Red	Kasun Fernando	987654123V	769876541	V013
14	WP-NO-2345	Car	Blue	Janaka Jayasinghe	123456987V	741234569	V014

Figure 10-CarTbl

ProfileTbl – Use to add data of the People

- Profile Number
- Full Name
- Address
- NIC
- Phone
- Email

- Gender
- Post
- Department
- Plate Number
- Vehicle Type

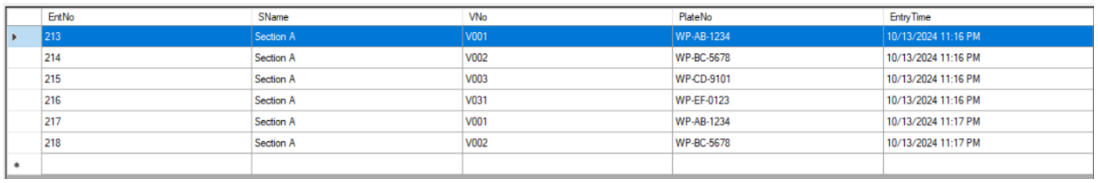


ProfileNo	FullName	Address	NIC	Phone	Email	Gender	Post	Department	PlateNo	Vtype
1	Nimal Perera	Colombo 03	123456789V	0711234567	nimal.perera@exa...	Male	General Manager	General Manager	WP-AB-1234	Car
2	Sunil Fernando	Kandy	987654321V	0779876543	sunil.fernando@e...	Male	Operations Manager	Operations Manager	WP-BC-5678	Car
3	Kamal Silva	Galle	456789123V	0754567891	kamal.silva@exa...	Male	Sales Manager	Sales Manager	WP-CD-9101	Car
4	Anura Jayasinghe	Matara	789123456V	0727891234	anura.jayasinghe...	Male	Marketing Manager	Marketing Manager	WP-DE-2345	Car
5	Ruwan Wijesinghe	Negombo	321654987V	0783216549	ruwan.wijesinghe...	Male	Customer Service ...	Customer Service ...	WP-EF-6789	Car
6	Saman Kumara	Kurunegala	654987321V	0766549873	saman.kumara@e...	Male	Network Manager	Network Manager	WP-FG-0123	Car
7	Chandana Rathna...	Ratnapura	987321654V	0749873216	chandana.rathnay...	Male	Technical Support...	Technical Support...	WP-GH-4567	Car
8	Priyantha Dissana...	Anuradhapura	123789456V	0711237894	priyantha.dissanay...	Male	Project Manager	Project Manager	WP-HI-8901	Car
9	Ranjith Senanayake	Badulla	456123789V	0774561237	ranjith.senanayak...	Male	Human Resource...	Human Resources...	WP-IJ-2345	Car
10	Lalith Gunawardena	Jaffna	789456123V	0757894561	lalith.gunawarden...	Male	Finance Manager	Finance Manager	WP-JK-6789	Car
11	Nuwan Perera	Colombo 05	321987654V	0723219876	nuwan.perera@ex...	Male	IT Manager	IT Manager	WP-KL-0123	Car

Figure 11-ProfileTbl

EntryTbl – Use to add data of the Entries

- Entry Number
- Section Name
- Vehicle Number
- Plate Number
- Entry Time

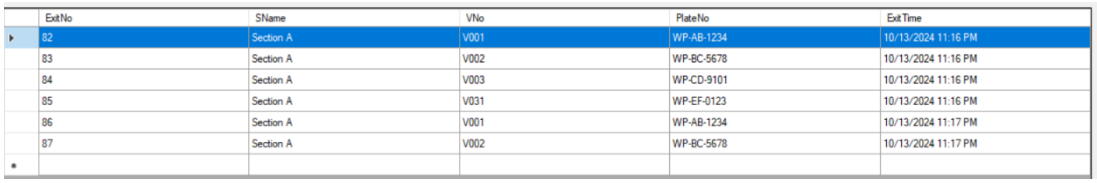


EntNo	SName	VNo	PlateNo	EntryTime
213	Section A	V001	WP-AB-1234	10/13/2024 11:16 PM
214	Section A	V002	WP-BC-5678	10/13/2024 11:16 PM
215	Section A	V003	WP-CD-9101	10/13/2024 11:16 PM
216	Section A	V031	WP-EF-0123	10/13/2024 11:16 PM
217	Section A	V001	WP-AB-1234	10/13/2024 11:17 PM
218	Section A	V002	WP-BC-5678	10/13/2024 11:17 PM

Figure 12-EntryTbl

ExitTbl – Use to add data of the Exits

- Exit Number
- Section Name
- Vehicle Number
- Plate Number
- Exit Time



ExitNo	SName	VNo	PlateNo	ExitTime
82	Section A	V001	WP-AB-1234	10/13/2024 11:16 PM
83	Section A	V002	WP-BC-5678	10/13/2024 11:16 PM
84	Section A	V003	WP-CD-9101	10/13/2024 11:16 PM
85	Section A	V031	WP-EF-0123	10/13/2024 11:16 PM
86	Section A	V001	WP-AB-1234	10/13/2024 11:17 PM
87	Section A	V002	WP-BC-5678	10/13/2024 11:17 PM

Figure 13-ExitTbl

5.4. Outputs

SectionTbl – Use to add data of the sections

- "Section Added" Message
- "Section Deleted" Message
- "Section Updated" Message

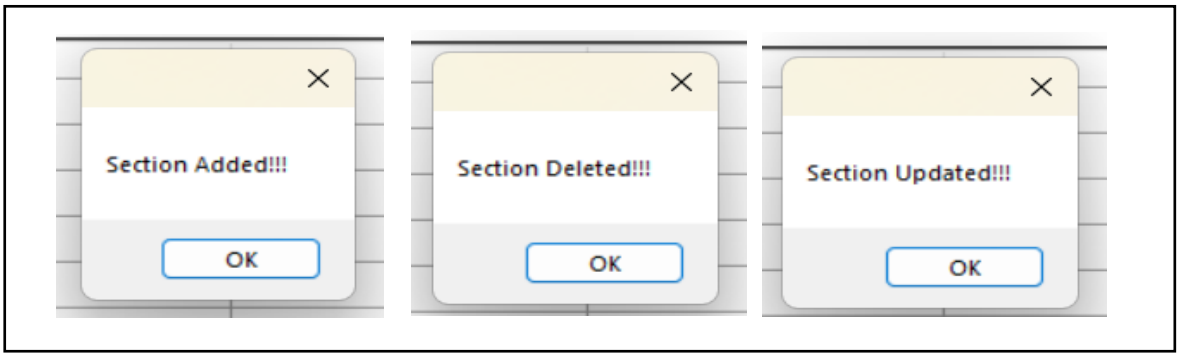


Figure 14-Outputs of Sections

CarTbl – Use to add data of the Vehicles

- "Vehicle Added" Message
- "Vehicle Entry Updated" Message
- "Vehicle Data Deleted" Message

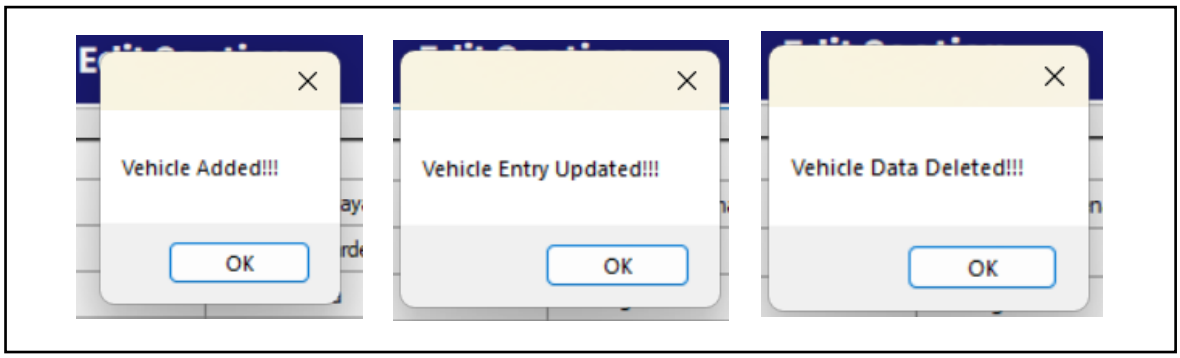


Figure 15-Outputs of Vehicles

ProfileTbl – Use to add data of the People

- "Profile Added" Message
- "Profile Updated" Message
- " Profile Deleted" Message

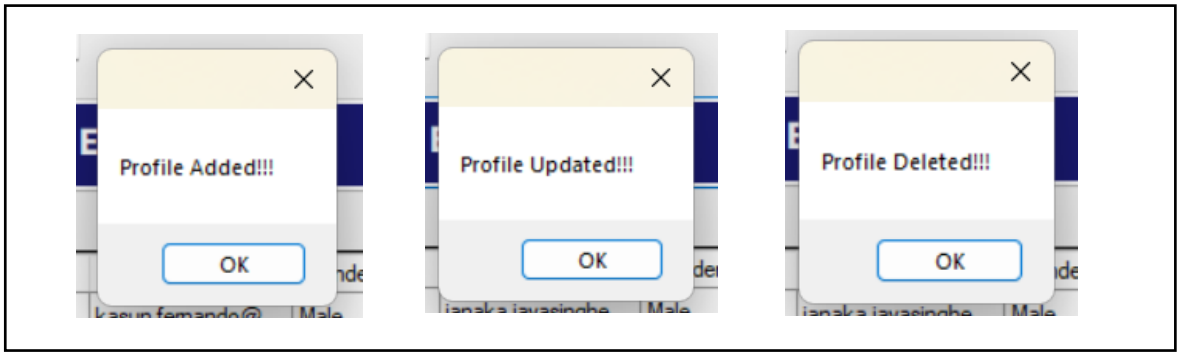


Figure 16-Outputs of Profiles

EntryTbl – Use to add data of the Entries

- "Entry Added" Message
- "Record Deleted" Message

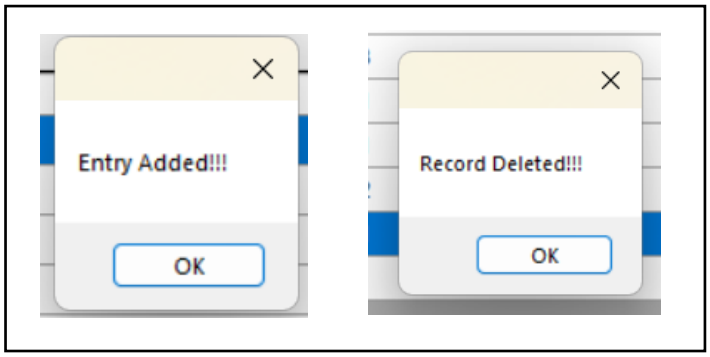


Figure 17-Outputs of Entries

ExitTbl – Use to add data of the Exit

- "Entry Added" Message
- "Record Deleted" Message

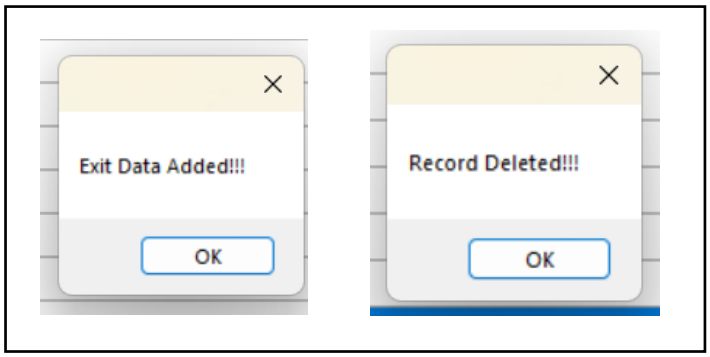


Figure 18-Outputs of Exit

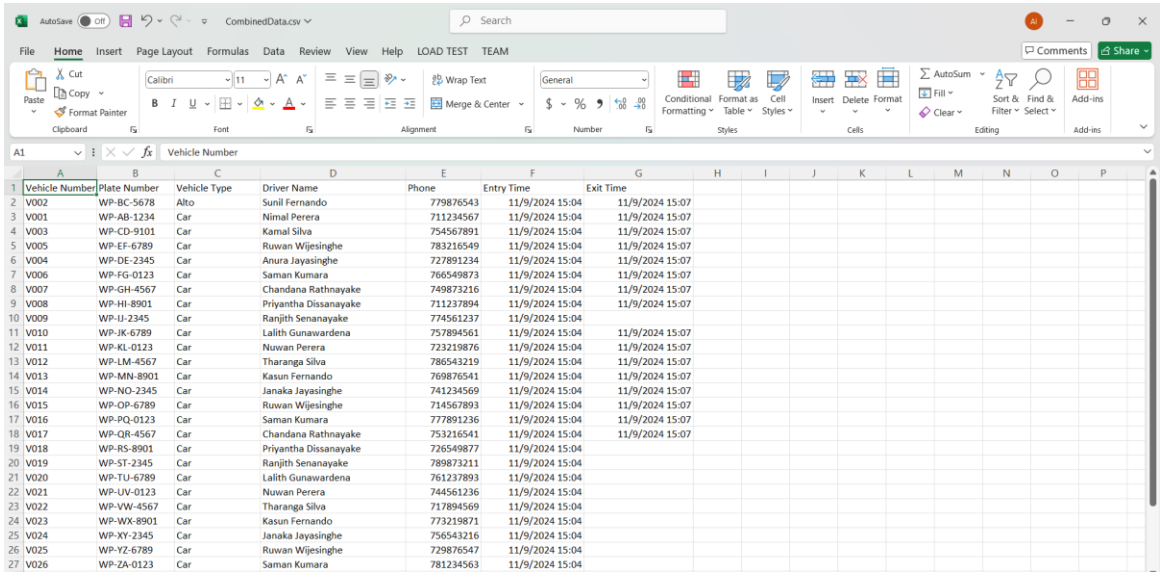
Dashboard

- Realtime Parking section Availability

Figure 19-Dashboard

Generate Report

- "Data Exported Successfully" Message
- Generate csv file



Vehicle Number	Plate Number	Vehicle Type	Driver Name	Phone	Entry Time	Exit Time
V002	WP-BC-5678	Alto	Sunil Fernando	779876543	11/9/2024 15:04	11/9/2024 15:07
V001	WP-AB-1234	Car	Nimal Perera	711234567	11/9/2024 15:04	11/9/2024 15:07
V003	WP-CD-9101	Car	Kamal Silva	754567891	11/9/2024 15:04	11/9/2024 15:07
V005	WP-EF-6789	Car	Ruwan Wijesinghe	783216549	11/9/2024 15:04	11/9/2024 15:07
V004	WP-DE-2345	Car	Anura Jayasinghe	727891234	11/9/2024 15:04	11/9/2024 15:07
V006	WP-FG-0123	Car	Saman Kumara	766549873	11/9/2024 15:04	11/9/2024 15:07
V007	WP-GH-4567	Car	Chandana Rathnayake	749873216	11/9/2024 15:04	11/9/2024 15:07
V008	WP-HI-8901	Car	Priyantha Dissanayake	711237894	11/9/2024 15:04	11/9/2024 15:07
V009	WP-IJ-2345	Car	Ranjith Senanayake	774561237	11/9/2024 15:04	
V010	WP-JK-6789	Car	Lalith Gunawardena	757894561	11/9/2024 15:04	11/9/2024 15:07
V011	WP-KL-0123	Car	Nuwan Perera	723219876	11/9/2024 15:04	11/9/2024 15:07
V012	WP-LM-4567	Car	Tharanga Silva	786543219	11/9/2024 15:04	11/9/2024 15:07
V013	WP-MN-8901	Car	Kasun Fernando	769876541	11/9/2024 15:04	11/9/2024 15:07
V014	WP-NO-2345	Car	Janaka Jayasinghe	741234569	11/9/2024 15:04	11/9/2024 15:07
V015	WP-OP-6789	Car	Ruwan Wijesinghe	714567893	11/9/2024 15:04	11/9/2024 15:07
V016	WP-PQ-0123	Car	Saman Kumara	777891236	11/9/2024 15:04	11/9/2024 15:07
V017	WP-QR-4567	Car	Chandana Rathnayake	753216541	11/9/2024 15:04	11/9/2024 15:07
V018	WP-RS-8901	Car	Priyantha Dissanayake	726549877	11/9/2024 15:04	
V019	WP-ST-2345	Car	Ranjith Senanayake	789873211	11/9/2024 15:04	
V020	WP-TU-6789	Car	Lalith Gunawardena	761237893	11/9/2024 15:04	
V021	WP-UV-0123	Car	Nuwan Perera	744561236	11/9/2024 15:04	
V022	WP-VW-4567	Car	Tharanga Silva	717894569	11/9/2024 15:04	
V023	WP-WX-8901	Car	Kasun Fernando	773219871	11/9/2024 15:04	
V024	WP-XY-2345	Car	Janaka Jayasinghe	756543216	11/9/2024 15:04	
V025	WP-YZ-6789	Car	Ruwan Wijesinghe	729876547	11/9/2024 15:04	
V026	WP-ZA-0123	Car	Saman Kumara	781234563	11/9/2024 15:04	

Figure 20-Output of Generate Report button(CSV file)

6. CHAPTER 06 – APPENDIX

6.1. Appendix A: Login Interface

6.1.1. Overview

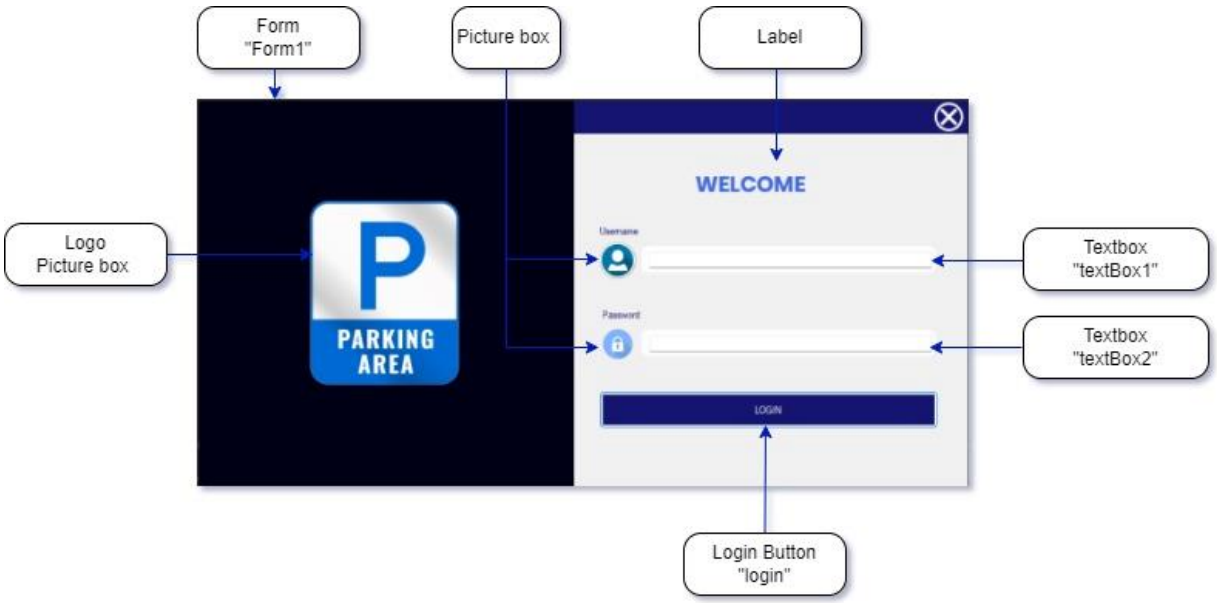


Figure 21-Login Interface overview

This appendix provides an overview of the login form design for the Vehicle Parking Management System Project. The primary interface, labeled Form1, serves as the main window for user interaction.

At the top left of the form is the Logo, displayed in a PictureBox component. This logo represents the "Parking Area," helping to establish the system's identity and purpose. Next to the logo is a secondary PictureBox that contains visual icons relevant to the input fields, enhancing the user experience by providing visual cues.

The form prominently features a Label that reads "WELCOME," situated above the login input fields. This welcoming message is designed to create a friendly atmosphere for users as they prepare to log in. Below this label, the user is presented with two TextBoxes. The first TextBox, referred to as textBox1, is designated for the username input and includes an accompanying user icon for clarity. The second TextBox, textBox2, is intended for the password. This field is designed to mask the input for security purposes and is paired with a key icon, further guiding users in entering their credentials.

Central to the login form is the Login Button, labeled simply as "LOGIN." When clicked, this button initiates the login process, validating the credentials entered in the TextBoxes. The button is strategically placed to encourage users to proceed after filling out the required information.

Additionally, an optional Close Button (represented by an "X" in the top-right corner) is included to allow users to exit the login screen easily. This button ensures that users have the flexibility to close the application if needed.

Overall, the design of the login form emphasizes user-friendliness and security, providing a clear and engaging interface for users as they access the Vehicle Parking Management System.

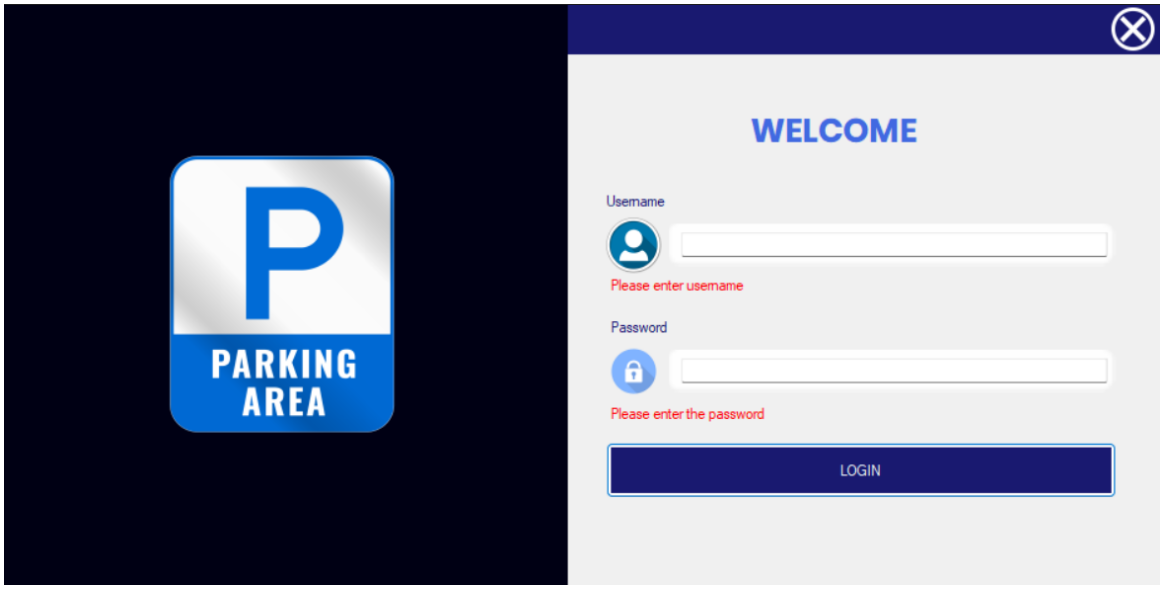


Figure 22-Login page show error when Without enter Username or Password

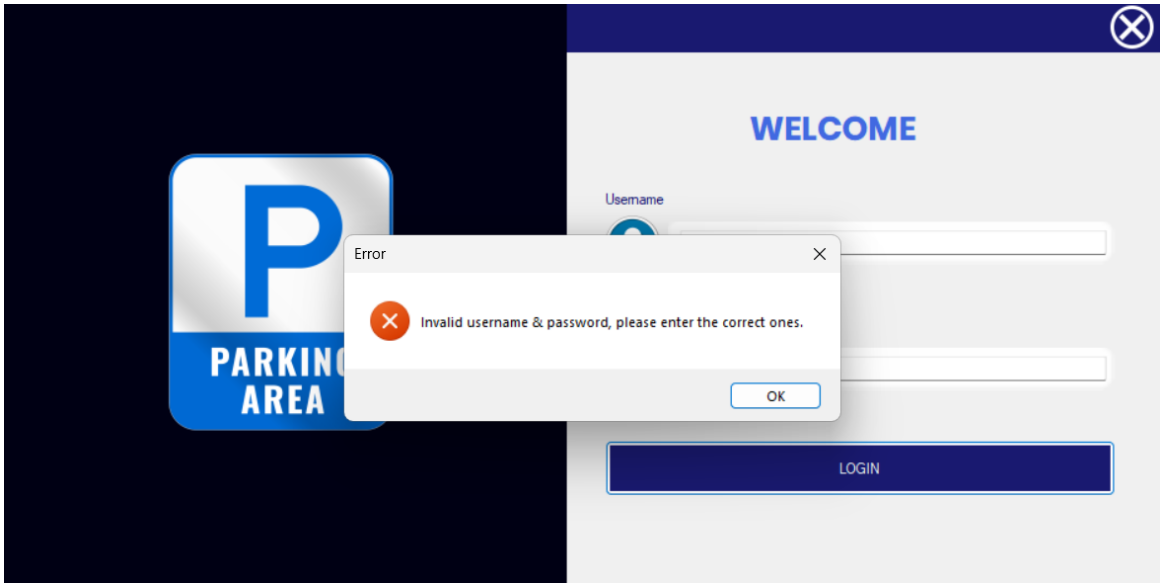


Figure 23-Login page show error when enter Wrong Username or Password

6.1.2. Colors

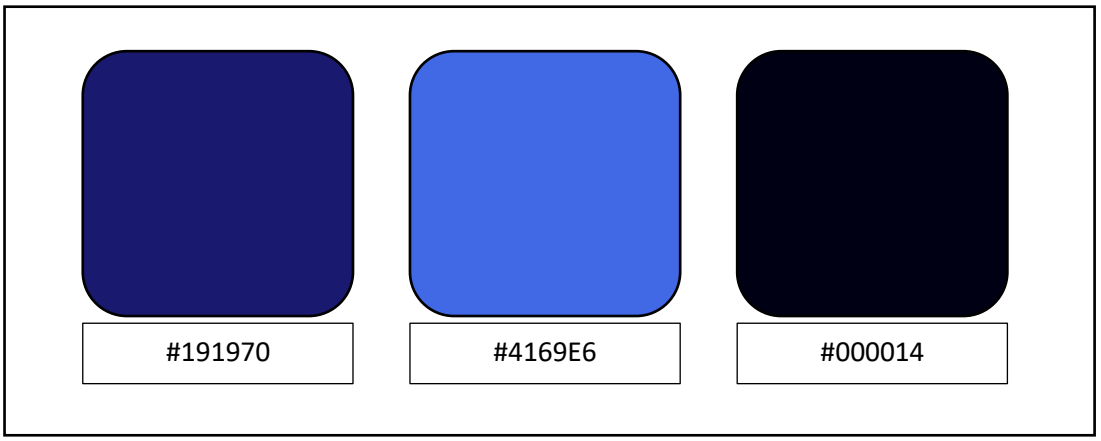


Figure 24-Color used in Login Interface

6.1.3. Full Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Vehicle_Parking_Management_System_Project
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         this.KeyPreview = true; // Allows the form to capture key events
```

```
        this.KeyDown += new KeyEventHandler(Form1_KeyDown); // Attach the
KeyDown event handler
    }

    // This method is triggered when the user clicks the login button
    private void login_Click(object sender, EventArgs e)
    {
        // Get the values entered in TextBox1 and TextBox2
        string username = textBox1.Text;
        string password = textBox2.Text;

        // Check if both fields are empty
        if (string.IsNullOrEmpty(username) && string.IsNullOrEmpty(password))
        {
            label4.Text = "Please enter username";
            label5.Text = "Please enter the password";
        }
        // Check if only the username field is empty
        else if (string.IsNullOrEmpty(username))
        {
            label4.Text = "Please enter username";
        }
        // Check if only the password field is empty
        else if (string.IsNullOrEmpty(password))
        {
            label5.Text = "Please enter the password";
        }
        else
        {
```

```
            // Validate credentials
            if (username == "admin" && password == "123")
            {
                // If valid, close the current form and open the next form
                this.Hide(); // Hide the login form
                Form nextForm = new main(); // Replace with your next form class
                nextForm.ShowDialog(); // Open the next form
                this.Close(); // Close the login form
            }
            else
            {
                // Check if both username and password are incorrect
                if (username != "a" && password != "1")
                {
                    MessageBox.Show("Invalid username & password, please enter the
correct ones.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
                // Check if only the username is incorrect
                else if (username != "a")
                {
                    MessageBox.Show("Invalid username, please enter the correct one.",
"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
                // Check if only the password is incorrect
                else if (password != "1")
                {
                    MessageBox.Show("Invalid password, please enter the correct one.",
"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
                else
```

```
            {
                // If valid, close the current form and open the next form
                this.Hide(); // Hide the login form
                Form nextForm = new main(); // Replace with your next form class
                nextForm.ShowDialog(); // Open the next form
                this.Close(); // Close the login form
            }
        }
    }
}

// This method captures key presses in the form
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    // If the Enter key is pressed, trigger the login button click
    if (e.KeyCode == Keys.Enter)
    {
        login.PerformClick(); // Trigger the login button's click event
    }
}

private void pictureBox5_Click(object sender, EventArgs e)
{
    // Close the application completely when the close button is clicked
    Application.Exit();
}
}
}
```

6.1.4. Code Introduction and Process

I. Class Definition and Constructor

```
namespace Vehicle_Parking_Management_System_Project
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         this.KeyPreview = true; // Allows the form to capture key events
         this.KeyDown += new KeyEventHandler(Form1_KeyDown); // Attach the KeyDown event handler
      }
```

In this section, the class Form1 is defined, inheriting from the Form class, which is part of the Windows Forms framework. The constructor initializes the form and sets the KeyPreview property to true, allowing the form to capture keyboard events before they are passed to any control on the form. The KeyDown event handler is attached to the form, enabling the application to respond to key presses.

II. Login Button Click Event

The login_Click method handles the event when the user clicks the login button. It retrieves the username and password entered into the respective text boxes. The method first checks if both fields are empty, providing appropriate feedback by updating labels label4 and label5. If either field is empty, it prompts the user to enter the missing information.

If both fields are filled, the method then validates the credentials. If the username is "admin" and the password is "123," it successfully logs in the user by hiding the login form and opening the next form (represented by main). If the credentials are incorrect, the method provides specific error messages depending on whether the username, password, or both are incorrect, using MessageBox.Show to inform the user.

```
private void login_Click(object sender, EventArgs e)
{
    // Get the values entered in TextBox1 and TextBox2
    string username = textBox1.Text;
    string password = textBox2.Text;

    // Check if both fields are empty
    if (string.IsNullOrEmpty(username) && string.IsNullOrEmpty(password))
    {
        label4.Text = "Please enter username";
        label5.Text = "Please enter the password";
    }
    // Check if only the username field is empty
    else if (string.IsNullOrEmpty(username))
    {
        label4.Text = "Please enter username";
    }
    // Check if only the password field is empty
    else if (string.IsNullOrEmpty(password))
    {
        label5.Text = "Please enter the password";
    }
    else
    {
        // Validate credentials
        if (username == "admin" && password == "123")
        {
            // If valid, close the current form and open the next form
            this.Hide(); // Hide the login form
```

```
            Form nextForm = new main(); // Replace with your next form class
            nextForm.ShowDialog(); // Open the next form
            this.Close(); // Close the login form
        }
        else
        {
            if (username != "a" && password != "1")
            {
                MessageBox.Show("Invalid username & password, please enter the correct
ones.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
            else if (username != "a")
            {
                MessageBox.Show("Invalid username, please enter the correct one.",
"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
            else if (password != "1")
            {
                MessageBox.Show("Invalid password, please enter the correct one.",
"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
            else
            {
                this.Hide(); // Hide the login form
                Form nextForm = new main(); // Replace with your next form class
                nextForm.ShowDialog(); // Open the next form
                this.Close(); // Close the login form
            }
        }
    }
```

III. KeyDown Event Handler

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    // If the Enter key is pressed, trigger the login button click
    if (e.KeyCode == Keys.Enter)
    {
        login.PerformClick(); // Trigger the login button's click event
    }
}
```

The Form1_KeyDown method captures key presses when the form is active. Specifically, if the Enter key is pressed, it triggers the PerformClick method on the login button, effectively simulating a button click. This feature enhances user experience by allowing them to log in by simply pressing Enter after filling in their credentials.

IV. Close Application Event

```
private void pictureBox5_Click(object sender, EventArgs e)
{
    // Close the application completely when the close button is clicked
    Application.Exit();
}
```

The pictureBox5_Click method is triggered when a specific picture box (likely a close button) is clicked. This method calls Application.Exit(), which terminates the application completely. This feature provides users with an easy way to exit the application without needing to close the form manually.

6.1.5. Summary

Overall, this code outlines the core functionality of the login interface for a Vehicle Parking Management System. It includes event handling for user interactions, validation of login credentials, and mechanisms to navigate between forms, enhancing usability and security within the application. Each section of the code contributes to creating an intuitive and efficient user experience.

6.2. Appendix B: Dashboard Interface

6.2.1. Overview

Figure 25-Dashboard Interface overview

The main form of the Vehicle Parking Management System is designed to offer a user-friendly interface with intuitive navigation and clear displays of key parking information. The form is titled "main", and it combines functionality with a clean design, ensuring users can access all important features efficiently.

At the top-left corner of the form is the Logo Picture Box, which displays a logo representing a "Parking Area." This visual element reinforces the system's purpose and helps users easily recognize the application's theme.

On the left side of the form is the Navigation Panel, which consists of several buttons aligned vertically for quick access. These buttons include the Sections Button ("sections"), which allows users to manage different parking sections, and the Vehicles Button ("vehicles") for vehicle management. Users can also manage parking entries with the Entries Button ("entries"), exit the application using the Exit Button ("exit"), and access the user profiles through the Profiles Button ("profiles"). Additionally, there is a Reset System Button ("reset") for resetting the system and a Logout Button ("logout") to securely log out.

The central portion of the form is dedicated to displaying the status of various parking sections. This area consists of multiple Labels, each representing a section (e.g., "SECTION A", "SECTION B"). Each section label is accompanied by two numeric displays showing Parking Used (the number of occupied spots) and Available Parking (the number of free spots). This layout ensures that users can quickly monitor parking availability across sections. There are six sections, each with 20 parking spaces, along with a special section, "SECTION G (SPECIAL)", which accommodates 50 parking spots.

In the top-right corner of the form, there is a dateTimePicker control, allowing users to view or select the current date and time. This feature is particularly useful for time-sensitive parking management and entry tracking, making it easier for users to manage time-based parking records.

At the bottom of the form is a prominent Generate Report Button ("genreport"), which enables users to generate reports based on current parking data. This functionality supports monitoring and record-keeping, ensuring that users can produce reports on parking activity whenever needed.

Overall, the form's design is centered around providing clear, organized information and easy access to key functionalities. The combination of labels, buttons, and parking status displays ensures that users can manage vehicles, parking sections, and system operations with ease. Each element is placed logically, enhancing the user experience while maintaining functionality.

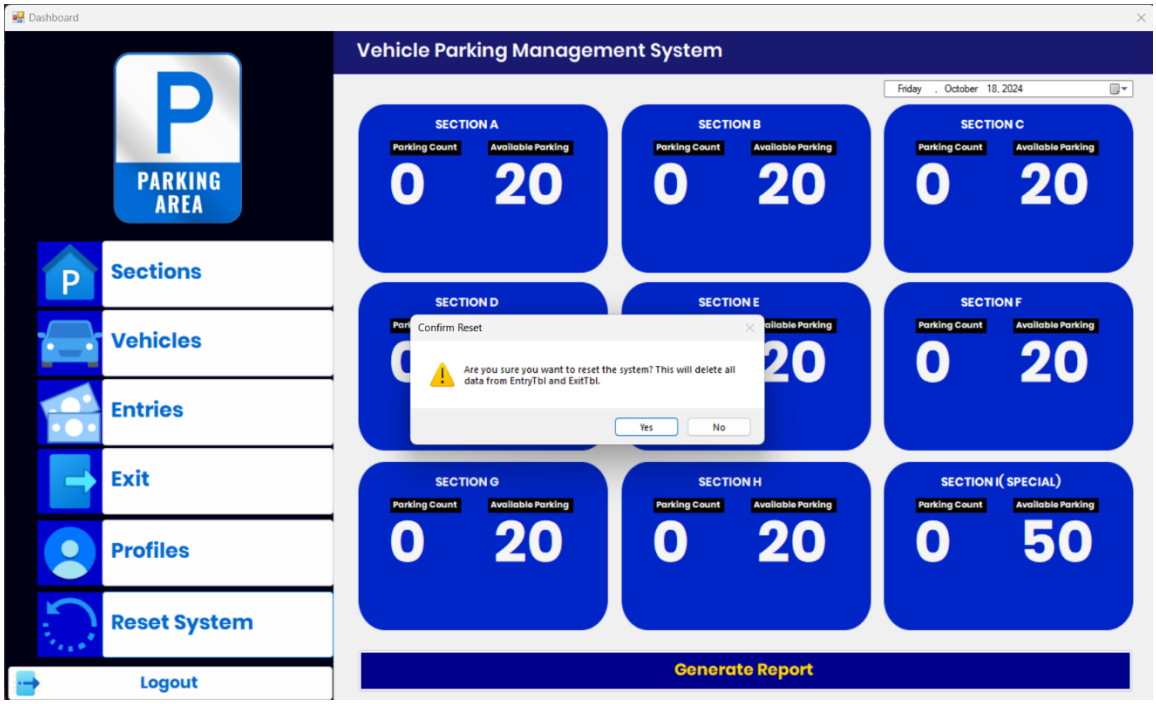


Figure 26-Dashboard page show message box when click the Reset System Button

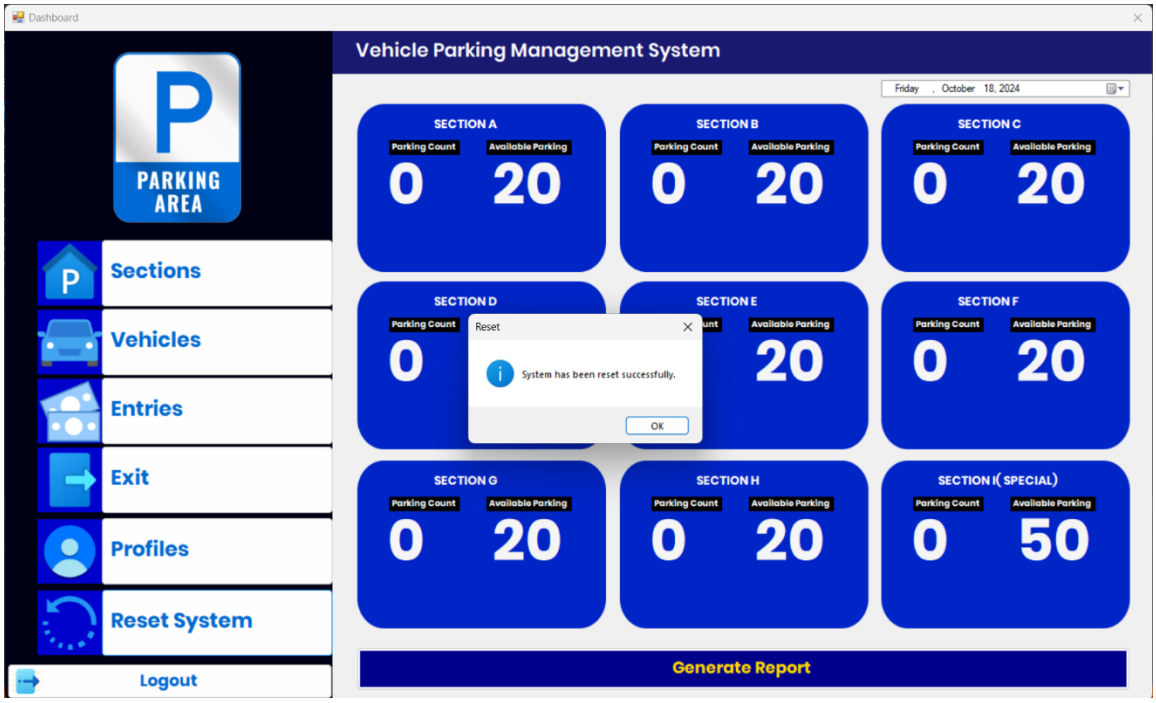


Figure 27-Dashboard page show message box System Has been reset successfully

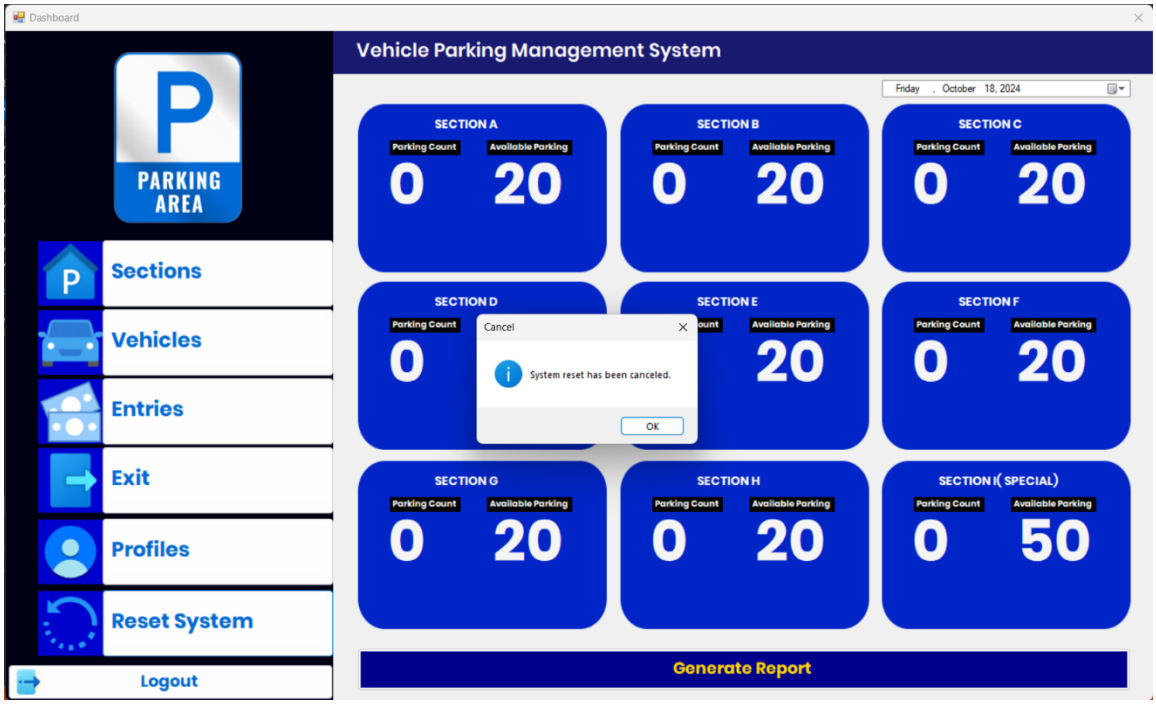


Figure 28-Dashboard page show message box System reset has been canceled

6.2.2. Colors

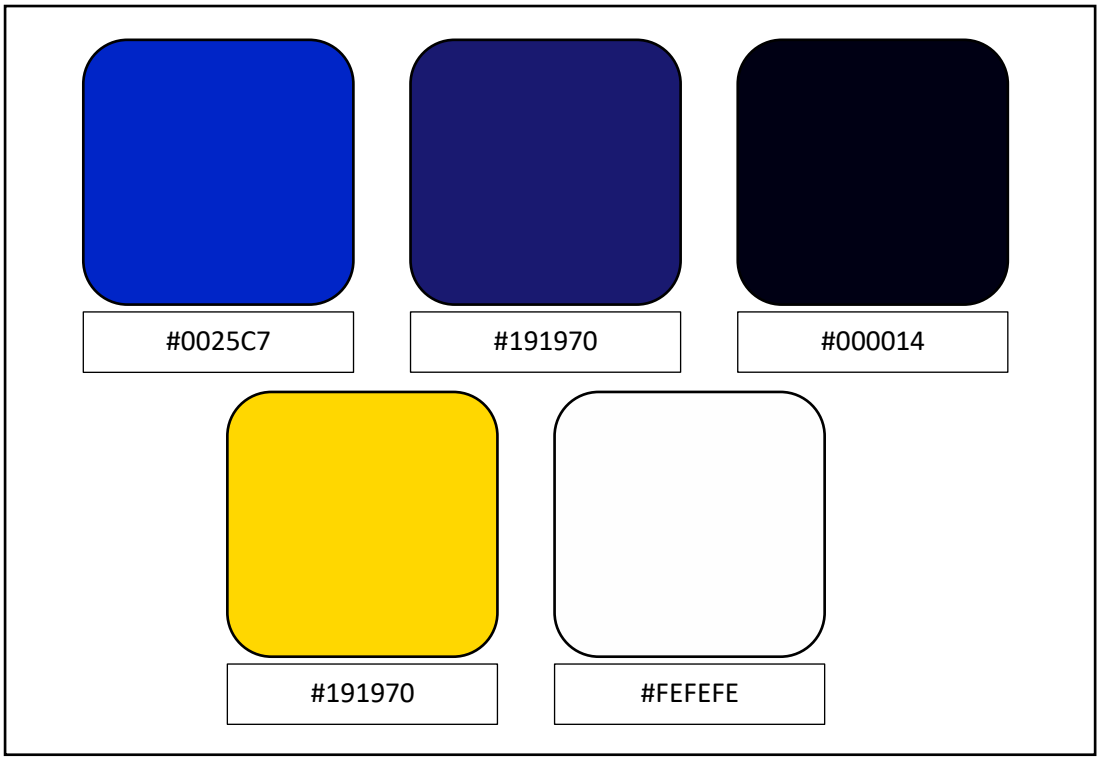


Figure 29-Color used in Dashboard Interface

6.2.3. Full Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace Vehicle_Parking_Management_System_Project
{
    public partial class main : Form
```

```
{
    private Timer timer;
    public main()
    {
        InitializeComponent();
        Con = new Functions();
        InitializeTimer(); // Initialize the timer
    }
    Functions Con;
    private void close_Click(object sender, EventArgs e)
    {
        // Close the application completely when the close button is clicked
        Application.Exit();
    }
    private void logout_Click(object sender, EventArgs e)
    {

        // If valid, close the current form and open the next form
        this.Hide(); // Hide the login form
        Form nextForm = new Form1(); // Replace with your next form class
        nextForm.ShowDialog(); // Open the next form
        this.Close();
    }
    private void login_Click(object sender, EventArgs e)
    {
        Form nextForm2 = new sections(); // connect to sections
        nextForm2.ShowDialog(); // Open the next form
    }
    private void LoadSectionACountSectionA()
```

```
    {
        try
        {

            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section A'";
            DataTable dtEntries = Con.GetData(queryEntries);
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section A'";
            DataTable dtExits = Con.GetData(queryExits);


            int countEntries = 0;
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
               countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
            }
            // Get count of exits (vehicles that left), ensure it is not less than 0
            if (dtExits.Rows.Count > 0)
            {
               countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
            }


            // Calculate the current count by subtracting exits from entries, ensure non-
negative
            int currentCount = Math.Max(countEntries - countExits, 0);
            count1.Text = currentCount.ToString();


            // Calculate the available parking spots (20 total spots - current parked vehicles)
            int availableCount = 20 - currentCount;
```

```
            available1.Text = availableCount.ToString();


            // If available spots are 0, show "No Available Parking" in noavailable1 label
            if (availableCount == 0)
            {
                noavailable1.Text = "No Available Parking";
            }
            else
            {
                noavailable1.Text = string.Empty; // Clear the message if parking is available
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error loading count: " + ex.Message);
        }
    }
    private void LoadSectionACountSectionB()
    {
        try
        {


            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section B'";
            DataTable dtEntries = Con.GetData(queryEntries);
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section B'";
            DataTable dtExits = Con.GetData(queryExits);


            int countEntries = 0;
```

```
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
                countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
            }

            // Get count of exits (vehicles that left), ensure it is not less than 0
            if (dtExits.Rows.Count > 0)
            {
                countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
            }

            // Calculate the current count by subtracting exits from entries, ensure non-
negative
            int currentCount = Math.Max(countEntries - countExits, 0);
            count2.Text = currentCount.ToString();

            // Calculate the available parking spots (20 total spots - current parked vehicles)
            int availableCount = 20 - currentCount;
            available2.Text = availableCount.ToString();

            // If available spots are 0, show "No Available Parking" in noavailable1 label
            if (availableCount == 0)
            {
                noavailable2.Text = "No Available Parking";
            }
            else
            {
                noavailable2.Text = string.Empty; // Clear the message if parking is available
```

```
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error loading count: " + ex.Message);
        }
    }

    private void LoadSectionACountSectionC()
    {
        try
        {
            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section C'";
            DataTable dtEntries = Con.GetData(queryEntries);
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section C'";
            DataTable dtExits = Con.GetData(queryExits);

            int countEntries = 0;
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
                countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
            }

            // Get count of exits (vehicles that left), ensure it is not less than 0
            if (dtExits.Rows.Count > 0)
            {
                countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
```

```
            }

            // Calculate the current count by subtracting exits from entries, ensure non-
negative
            int currentCount = Math.Max(countEntries - countExits, 0);
            count3.Text = currentCount.ToString();

            // Calculate the available parking spots (20 total spots - current parked vehicles)
            int availableCount = 20 - currentCount;
            available3.Text = availableCount.ToString();

            // If available spots are 0, show "No Available Parking" in noavailable1 label
            if (availableCount == 0)
            {
                noavailable3.Text = "No Available Parking";
            }
            else
            {
                noavailable3.Text = string.Empty; // Clear the message if parking is available
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error loading count: " + ex.Message);
        }
    }
    private void LoadSectionACountSectionD()
    {
        try
```

```
        {
            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section D'";
            DataTable dtEntries = Con.GetData(queryEntries);
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section D'";
            DataTable dtExits = Con.GetData(queryExits);

            int countEntries = 0;
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
               countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
            }

            // Get count of exits (vehicles that left), ensure it is not less than 0
            if (dtExits.Rows.Count > 0)
            {
               countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
            }
            // Calculate the current count by subtracting exits from entries, ensure non-
negative
            int currentCount = Math.Max(countEntries - countExits, 0);
            count4.Text = currentCount.ToString();

            // Calculate the available parking spots (20 total spots - current parked vehicles)
            int availableCount = 20 - currentCount;
            available4.Text = availableCount.ToString();

            // If available spots are 0, show "No Available Parking" in noavailable1 label
```

```
            if (availableCount == 0)
            {
                noavailable4.Text = "No Available Parking";
            }
            else
            {
                noavailable4.Text = string.Empty; // Clear the message if parking is available
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error loading count: " + ex.Message);
        }
    }

    private void LoadSectionACountSectionE()
    {
        try
        {
            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section E'";
            DataTable dtEntries = Con.GetData(queryEntries);
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section E'";
            DataTable dtExits = Con.GetData(queryExits);

            int countEntries = 0;
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
```

```
            countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
        }

        // Get count of exits (vehicles that left), ensure it is not less than 0
        if (dtExits.Rows.Count > 0)
        {
            countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
        }
        // Calculate the current count by subtracting exits from entries, ensure non-
negative
        int currentCount = Math.Max(countEntries - countExits, 0);
        count5.Text = currentCount.ToString();

        // Calculate the available parking spots (20 total spots - current parked vehicles)
        int availableCount = 20 - currentCount;
        available5.Text = availableCount.ToString();

        // If available spots are 0, show "No Available Parking" in noavailable1 label
        if (availableCount == 0)
        {
            noavailable5.Text = "No Available Parking";
        }
        else
        {
            noavailable5.Text = string.Empty; // Clear the message if parking is available
        }
    }
    catch (Exception ex)
    {
```

```
            MessageBox.Show("Error loading count: " + ex.Message);
        }
    }
    private void LoadSectionACountSectionF()
    {
        try
        {
            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section F'";
            DataTable dtEntries = Con.GetData(queryEntries);
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section F'";
            DataTable dtExits = Con.GetData(queryExits);

            int countEntries = 0;
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
                countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
            }

            // Get count of exits (vehicles that left), ensure it is not less than 0
            if (dtExits.Rows.Count > 0)
            {
                countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
            }

            // Calculate the current count by subtracting exits from entries, ensure non-
negative
            int currentCount = Math.Max(countEntries - countExits, 0);
```

```
            count6.Text = currentCount.ToString();

            // Calculate the available parking spots (20 total spots - current parked vehicles)
            int availableCount = 20 - currentCount;
            available6.Text = availableCount.ToString();

            // If available spots are 0, show "No Available Parking" in noavailable1 label
            if (availableCount == 0)
            {
                noavailable6.Text = "No Available Parking";
            }
            else
            {
                noavailable6.Text = string.Empty; // Clear the message if parking is available
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error loading count: " + ex.Message);
        }
    }
    private void LoadSectionACountSectionG()
    {
        try
        {

            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section G'";
            DataTable dtEntries = Con.GetData(queryEntries);
```

```
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section G'";
            DataTable dtExits = Con.GetData(queryExits);


            int countEntries = 0;
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
               countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
            }


            // Get count of exits (vehicles that left), ensure it is not less than 0
            if (dtExits.Rows.Count > 0)
            {
               countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
            }
            // Calculate the current count by subtracting exits from entries, ensure non-
negative
            int currentCount = Math.Max(countEntries - countExits, 0);
            count7.Text = currentCount.ToString();
            // Calculate the available parking spots (20 total spots - current parked vehicles)
            int availableCount = 20 - currentCount;
            available7.Text = availableCount.ToString();


            // If available spots are 0, show "No Available Parking" in noavailable1 label
            if (availableCount == 0)
            {
               noavailable7.Text = "No Available Parking";
            }
            else
```

```
            {
                noavailable7.Text = string.Empty; // Clear the message if parking is available
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error loading count: " + ex.Message);
        }
    }
    private void LoadSectionACountSectionH()
    {
        try
        {

            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section H'";
            DataTable dtEntries = Con.GetData(queryEntries);
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section H'";
            DataTable dtExits = Con.GetData(queryExits);
            int countEntries = 0;
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
                countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
            }

            // Get count of exits (vehicles that left), ensure it is not less than 0
            if (dtExits.Rows.Count > 0)
            {
```

```
            countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
        }

        // Calculate the current count by subtracting exits from entries, ensure non-
negative
        int currentCount = Math.Max(countEntries - countExits, 0);
        count8.Text = currentCount.ToString();

        // Calculate the available parking spots (20 total spots - current parked vehicles)
        int availableCount = 20 - currentCount;
        available8.Text = availableCount.ToString();

        // If available spots are 0, show "No Available Parking" in noavailable1 label
        if (availableCount == 0)
        {
            noavailable8.Text = "No Available Parking";
        }
        else
        {
            noavailable8.Text = string.Empty; // Clear the message if parking is available
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error loading count: " + ex.Message);
    }
}
private void LoadSectionACountSectionI()
{
```

```
        try
        {

            string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName
= 'Section I'";
            DataTable dtEntries = Con.GetData(queryEntries);
            string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName =
'Section I'";
            DataTable dtExits = Con.GetData(queryExits);

            int countEntries = 0;
            int countExits = 0;
            if (dtEntries.Rows.Count > 0)
            {
               countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
            }

            // Get count of exits (vehicles that left), ensure it is not less than 0
            if (dtExits.Rows.Count > 0)
            {
               countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
            }

            // Calculate the current count by subtracting exits from entries, ensure non-
negative
            int currentCount = Math.Max(countEntries - countExits, 0);
            count9.Text = currentCount.ToString();

            // Calculate the available parking spots (20 total spots - current parked vehicles)
            int availableCount = 50 - currentCount;
```

```
            available9.Text = availableCount.ToString();

            // If available spots are 0, show "No Available Parking" in noavailable1 label
            if (availableCount == 0)
            {
                noavailable9.Text = "No Available Parking";
            }
            else
            {
                noavailable9.Text = string.Empty; // Clear the message if parking is available
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error loading count: " + ex.Message);
        }
    }
    private void main_Load(object sender, EventArgs e)
    {
        // Load data count when form loads
        LoadSectionACountSectionA();
        LoadSectionACountSectionB();
        LoadSectionACountSectionC();
        LoadSectionACountSectionD();
        LoadSectionACountSectionE();
        LoadSectionACountSectionF();
        LoadSectionACountSectionG();
        LoadSectionACountSectionH();
        LoadSectionACountSectionI();
```

```
}

private void InitializeTimer()
{
    // Initialize the timer for real-time updates
    timer = new Timer();
    timer.Interval = 5000; // Set the interval to 5 seconds (5000 ms)
    timer.Tick += new EventHandler(OnTimerTick);
    timer.Start(); // Start the timer
}
private void OnTimerTick(object sender, EventArgs e)
{
    // Update the count in real-time every 5 seconds
    LoadSectionACountSectionA();
    LoadSectionACountSectionB();
    LoadSectionACountSectionC();
    LoadSectionACountSectionD();
    LoadSectionACountSectionE();
    LoadSectionACountSectionF();
    LoadSectionACountSectionG();
    LoadSectionACountSectionH();
    LoadSectionACountSectionI();
}

private void button1_Click(object sender, EventArgs e)
{
    Form nextForm2 = new vehicles(); // connect to entries
    nextForm2.ShowDialog(); // Open the next form
}
```

```
private void button2_Click(object sender, EventArgs e)
{
    Form nextForm2 = new Entries(); // connect to entries
    nextForm2.ShowDialog(); // Open the next form
}

private void button3_Click(object sender, EventArgs e)
{
    Form nextForm2 = new exit(); // connect to entries
    nextForm2.ShowDialog(); // Open the next form
}

private void button4_Click(object sender, EventArgs e)
{
    Form nextForm2 = new Profiles(); // connect to entries
    nextForm2.ShowDialog(); // Open the next form
}

private void button6_Click(object sender, EventArgs e)
{
    // Fetch combined data from EntryTbl and ExitTbl using the Functions instance
    DataTable combinedData = Con.GetCombinedData();

    // Define the file path where the CSV file will be saved
    string folderPath = @"C:\Users\ISHAN\Desktop\BIT Project\ExitData";

    // Create the directory if it doesn't exist
    if (!Directory.Exists(folderPath))
```

```
        {
            Directory.CreateDirectory(folderPath);
        }

        string filePath = Path.Combine(folderPath, "CombinedData.csv"); // Use
Path.Combine for better path handling

        try
        {
            // Export the DataTable to CSV
            ExportDataTableToCSV(combinedData, filePath);

            // Notify the user that the export was successful
            MessageBox.Show("Data exported successfully!");
        }
        catch (Exception ex)
        {
            // Handle any errors that may occur during the export
            MessageBox.Show("Error exporting data: " + ex.Message);
        }
    }
    // Function to export DataTable to CSV
    private void ExportDataTableToCSV(DataTable dt, string filePath)
    {
        using (StreamWriter sw = new StreamWriter(filePath))
        {
            // Write the header row
            for (int i = 0; i < dt.Columns.Count; i++)
            {
```

```
                sw.Write(dt.Columns[i]);
                if (i < dt.Columns.Count - 1)
                {
                    sw.Write(","); // Add comma between column names
                }
            }
            sw.WriteLine();

            // Write all rows
            foreach (DataRow row in dt.Rows)
            {
                for (int i = 0; i < dt.Columns.Count; i++)
                {
                    sw.Write(row[i].ToString());
                    if (i < dt.Columns.Count - 1)
                    {
                        sw.Write(","); // Add comma between column values
                    }
                }
                sw.WriteLine();
            }
        }
    }
    private void reset_Click(object sender, EventArgs e)
    {
        // Display a confirmation dialog asking if the user is sure they want to reset
        DialogResult result = MessageBox.Show(
            "Are you sure you want to reset the system? This will delete all data from
EntryTbl and ExitTbl.",
```

```
            "Confirm Reset",
            MessageBoxButtons.YesNo,
            MessageBoxIcon.Warning);

        if (result == DialogResult.Yes)
        {
            // If the user clicks "Yes", proceed with the reset
            try
            {
                Con.ClearTables(); // Call the method to clear the tables
                MessageBox.Show("System has been reset successfully.", "Reset",
MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error resetting the system: " + ex.Message, "Error",
MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }
        else if (result == DialogResult.No)
        {
            // If the user clicks "No", do nothing and return
            MessageBox.Show("System reset has been canceled.", "Cancel",
MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }

  }
}
```

6.2.4. Code Introduction and Process

I. Class Definition and Constructor

```
public partial class main : Form
{
    private Timer timer;
    public main()
    {
        InitializeComponent();
        Con = new Functions();
        InitializeTimer(); // Initialize the timer
    }
    Functions Con;
```

Here, the main class inherits from Form, indicating that it represents a Windows Form. A Timer object and a Functions instance (Con) are declared. The constructor (main()) calls InitializeComponent() to set up the form's components. The Con object is used for handling database interactions, and InitializeTimer() starts a timer for updating parking information periodically.

II. Close Application (close_Click) Method

```
private void close_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

When the close button is clicked, this method is triggered, which calls Application.Exit(), terminating the entire application.

III. Logout

```
private void logout_Click(object sender, EventArgs e)
{
    this.Hide();
    Form nextForm = new Form1(); // Replace with your next form class
    nextForm.ShowDialog();
    this.Close();
}
```

The logout_Click method hides the current form (this.Hide()), opens a new form (Form1) using ShowDialog(), and then closes the current form once the new one is active. This simulates logging out and transitioning to a new form in the system.

IV. Sections

```
private void login_Click(object sender, EventArgs e)
{
    Form nextForm2 = new sections(); // Connect to sections form
    nextForm2.ShowDialog();
}
```

This method handles the login button click. When clicked, a new form (sections) is opened using ShowDialog(), allowing the user to navigate to the sections management page.

V. Vehicles

```
private void button1_Click(object sender, EventArgs e)
{
    Form nextForm2 = new vehicles(); // connect to vehicles form
    nextForm2.ShowDialog(); // Open the next form
}
```

When button1 is clicked, it opens a form related to vehicle management (vehicles form). This allows the user to manage or view vehicles, likely for adding or viewing parked vehicles.

VI. Entries, Exit, Profiles

```
private void button2_Click(object sender, EventArgs e)
{
    Form nextForm2 = new Entries(); // connect to entries form
    nextForm2.ShowDialog(); // Open the next form
}

private void button3_Click(object sender, EventArgs e)
{
    Form nextForm2 = new exit(); // connect to exit form
    nextForm2.ShowDialog(); // Open the next form
}

private void button4_Click(object sender, EventArgs e)
{
    Form nextForm2 = new Profiles(); // connect to profiles form
    nextForm2.ShowDialog(); // Open the next form
}
```

These buttons open other forms in the system: the Entries form (to manage or view entries of vehicles), the exit form (likely for vehicle exits), and the Profiles form (for user or system profiles). All these forms open using ShowDialog(), which creates a modal window, preventing interaction with the main form until the new form is closed.

VII. Export CSV file

This method handles the process of exporting data from two tables (EntryTbl and ExitTbl) into a CSV file. It retrieves the combined data using Con.GetCombinedData(), checks if the target folder exists (and creates it if necessary), then exports the data to a CSV file (CombinedData.csv). If the export is successful, a confirmation message is shown; if not, an error message appears.

```
private void button6_Click(object sender, EventArgs e)
{
    // Fetch combined data from EntryTbl and ExitTbl using the Functions instance
    DataTable combinedData = Con.GetCombinedData();
    // Define the file path where the CSV file will be saved
    string folderPath = @"C:\Users\ISHAN\Desktop\BIT Project\ExitData";
    // Create the directory if it doesn't exist
    if (!Directory.Exists(folderPath))
    {
        Directory.CreateDirectory(folderPath);
    }
    string filePath = Path.Combine(folderPath, "CombinedData.csv"); // Use
Path.Combine for better path handling


    try
    {
        // Export the DataTable to CSV
        ExportDataTableToCSV(combinedData, filePath);
        // Notify the user that the export was successful
        MessageBox.Show("Data exported successfully!");
    }
    catch (Exception ex)
    {
        // Handle any errors that may occur during the export
        MessageBox.Show("Error exporting data: " + ex.Message);
    }
}
```

```
private void ExportDataTableToCSV(DataTable dt, string filePath)
{
    using (StreamWriter sw = new StreamWriter(filePath))
    {
        for (int i = 0; i < dt.Columns.Count; i++)
        {
            sw.Write(dt.Columns[i]);
            if (i < dt.Columns.Count - 1)
            {
                sw.Write(","); // Add comma between column names
            }
        }
        sw.WriteLine();
        foreach (DataRow row in dt.Rows)
        {
            for (int i = 0; i < dt.Columns.Count; i++)
            {
                sw.Write(row[i].ToString());
                if (i < dt.Columns.Count - 1)
                {
                    sw.Write(","); // Add comma between column values
                }
            }
            sw.WriteLine();
        }
    }
}
```

VIII. Resetting System

```
private void reset_Click(object sender, EventArgs e)
{
   DialogResult result = MessageBox.Show(
      "Are you sure you want to reset the system? This will delete all data from EntryTbl and ExitTbl.",
      "Confirm Reset",
      MessageBoxButtons.YesNo,
      MessageBoxIcon.Warning);
   if (result == DialogResult.Yes)
   {
      try
      {
         Con.ClearTables(); // Call the method to clear the tables
         MessageBox.Show("System has been reset successfully.", "Reset", MessageBoxButtons.OK, MessageBoxIcon.Information);
      }
      catch (Exception ex)
      {
         MessageBox.Show("Error resetting the system: " + ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
      }
   }
   else if (result == DialogResult.No)
   {
      MessageBox.Show("System reset has been canceled.", "Cancel", MessageBoxButtons.OK, MessageBoxIcon.Information);
   }
}
```

This method handles the process of resetting the system by clearing all data from EntryTbl and ExitTbl. A confirmation dialog is displayed to the user, asking for confirmation before proceeding with the reset. If the user confirms (Yes), the Con.ClearTables() method is called to clear the tables, and a success message is displayed. If an error occurs during the reset, an error message is shown. If the user cancels (No), the reset is canceled, and a cancellation message is shown.

VIII. Load Count for Sections

```
private void LoadSectionACountSectionD()
{
   try
   {
      string queryEntries = "SELECT COUNT(*) FROM EntryTbl WHERE SName = 'Section D'";
      DataTable dtEntries = Con.GetData(queryEntries);
      string queryExits = "SELECT COUNT(*) FROM ExitTbl WHERE SName = 'Section D'";
      DataTable dtExits = Con.GetData(queryExits);
      int countEntries = 0;
      int countExits = 0;
      if (dtEntries.Rows.Count > 0)
      {
         countEntries = Math.Max(Convert.ToInt32(dtEntries.Rows[0][0]), 0);
      }
      if (dtExits.Rows.Count > 0)
      {
         countExits = Math.Max(Convert.ToInt32(dtExits.Rows[0][0]), 0);
      }
      int currentCount = Math.Max(countEntries - countExits, 0);
      count4.Text = currentCount.ToString();

      int availableCount = 20 - currentCount;
      available4.Text = availableCount.ToString();
```

```
        if (availableCount == 0)
        {
            noavailable4.Text = "No Available Parking";
        }
        else
        {
            noavailable4.Text = string.Empty;
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error loading count: " + ex.Message);
    }
}
```

The LoadSectionACountSectionA method works the same way as the previous two but is designed for "Section A." It retrieves the entry and exit counts, calculates the current count, updates the available spots, and displays relevant messages about parking availability.

IX. Timer Set

```
private void InitializeTimer()
{
    timer = new Timer();
    timer.Interval = 5000; // Set the interval to 5 seconds (5000 ms)
    timer.Tick += new EventHandler(OnTimerTick);
    timer.Start(); // Start the timer
}
```

This method sets up a timer to refresh the parking section counts every 5 seconds. It sets the interval to 5000 milliseconds (5 seconds) and attaches an event handler (OnTimerTick) to execute every time the timer ticks.

```
private void OnTimerTick(object sender, EventArgs e)
{
    // Update the count in real-time every 5 seconds
    LoadSectionACountSectionA();
    LoadSectionACountSectionB();
    LoadSectionACountSectionC();
    LoadSectionACountSectionD();
    LoadSectionACountSectionE();
    LoadSectionACountSectionF();
    LoadSectionACountSectionG();
    LoadSectionACountSectionH();
    LoadSectionACountSectionI();
}
private void main_Load(object sender, EventArgs e)
{
    // Load data count when form loads
    LoadSectionACountSectionA();
    LoadSectionACountSectionB();
    LoadSectionACountSectionC();
    LoadSectionACountSectionD();
    LoadSectionACountSectionE();
    LoadSectionACountSectionF();
    LoadSectionACountSectionG();
    LoadSectionACountSectionH();
    LoadSectionACountSectionI();
}
```

This method is executed when the form loads. It calls several functions (LoadSectionACountSectionA, LoadSectionACountSectionB, etc.) to load and display the count of vehicles in sections A through I. Each of these methods likely retrieves the current vehicle count for a specific parking section.

The OnTimerTick method is called every time the timer ticks (every 5 seconds). This method reloads the vehicle counts for all the parking sections from A to I, ensuring that the displayed data remains up to date in real-time.

6.2.5. Summary

The code consists of several methods designed to manage parking sections in a Vehicle Parking Management System. It includes functions to load and display the counts of parked vehicles in different sections (Sections A through I) by querying entry and exit tables in a database. Each method calculates the current count of vehicles by subtracting the number of exits from the number of entries, ensuring that counts do not go negative. It also computes the number of available parking spots and displays a message if no spots are available. Additionally, the code features a timer to update these counts in real time every five seconds, several button click event handlers to navigate between forms (for vehicle entries, exits, and profiles), a method for exporting combined data from the entry and exit tables to a CSV file, and a reset function that prompts the user for confirmation before clearing the data in both tables.

6.3. Appendix C: Sections Interface

6.3.1. Overview

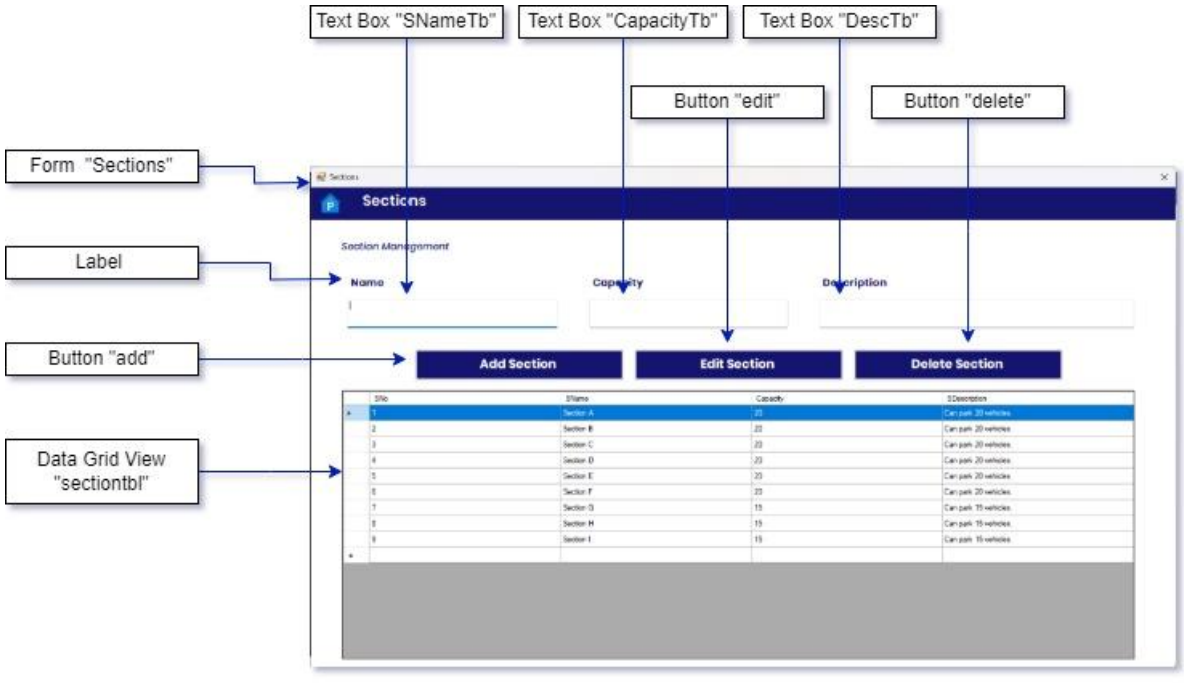


Figure 30-Sections Interface overview

The "Sections" form is a well-organized interface designed for efficient management of sections within a system. At the top, the form features a clear label, "Section Management," indicating its purpose. Below the label are three input fields, each serving a specific function: the SNameTb text box allows users to input the section name, the CapacityTb text box is for specifying the section's capacity, and the DescTb text box captures additional descriptive details about the section. These fields are logically aligned to ensure easy data entry.

Directly beneath the input fields, three prominently placed buttons enable key operations: the Add Section button allows users to insert new sections into the system, the Edit Section button facilitates the modification of existing sections, and the Delete Section button supports the removal of sections. These action buttons are intuitively positioned, allowing for streamlined interaction with the form.

At the bottom of the form, a DataGridView control, named sectiontbl, displays a detailed overview of all current sections, including their names, capacities, and descriptions. This grid allows users to select sections for editing or deletion, providing a visual representation of the data. Overall, the form's design is clean and user-friendly, making section management simple and efficient by combining input fields, action buttons, and a data table into a cohesive layout.

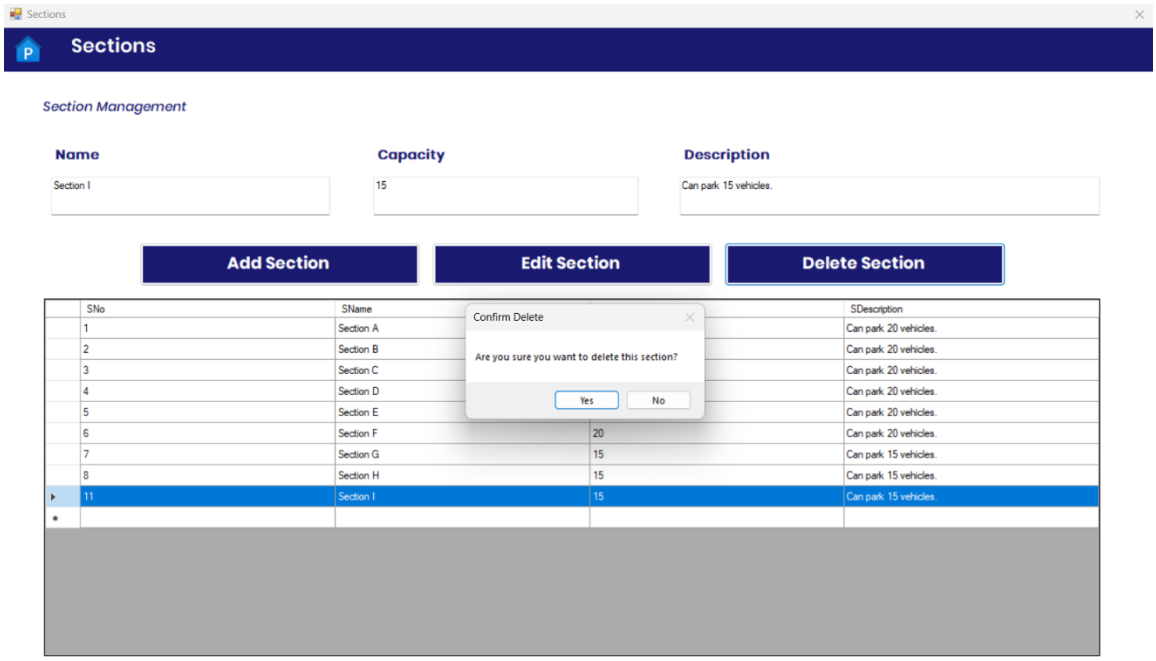


Figure 31-Sections page show message box when click Delete Button

6.3.2. Colors

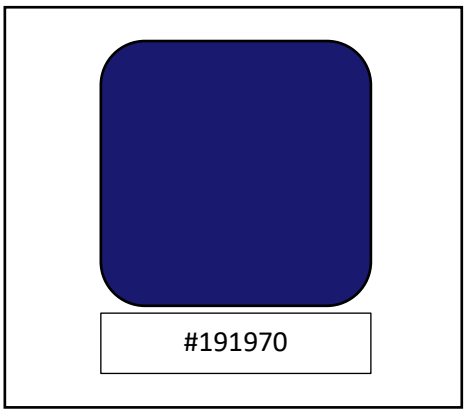


Figure 32-Color used in Dashboard Interface

6.3.3. Full Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace Vehicle_Parking_Management_System_Project
{
   public partial class sections : Form
   {
      public sections()
      {
         InitializeComponent();
         Con = new Functions();
         ShowSection();
      }
```

```
Functions Con;
private void ShowSection()
{
    string Query = "SELECT * FROM SectionTbl";
    SectionList.DataSource = Con.GetData(Query);
    SectionList.Refresh(); // Refresh the DataGridView if needed
}
private void close_Click(object sender, EventArgs e)
{
    // Close the application completely when the close button is clicked
    this.Close();
}
private void add_Click(object sender, EventArgs e)
{
    if (SNameTb.Text == "" || CapacityTb.Text == "" || DescTb.Text == "")
    {
        MessageBox.Show("Missing data!!!");
    }
    else
    {
        try
        {
            string Name = SNameTb.Text;
            string Cap = CapacityTb.Text;
            string Desc = DescTb.Text;
            string Query = "INSERT INTO SectionTbl VALUES ('{0}', {1}, '{2}')";
            Query = string.Format(Query, Name, Cap, Desc);
            // Execute the insert command
            Con.SetData(Query);
```

```
            MessageBox.Show("Section Added!!!");
            ShowSection();
        }
        catch (Exception Ex)
        {
            MessageBox.Show(Ex.Message);
        }
    }
}
int Key = 0;
private void SectionList_CellContentClick(object sender,
DataGridViewCellEventArgs e)
{
    // Check if a valid row is clicked
    if (e.RowIndex >= 0)
    {
        // Automatically select the entire row
        SectionList.CurrentRow.Selected = true;
        // Populate text boxes with the selected row's data
        SNameTb.Text = SectionList.CurrentRow.Cells[1].Value.ToString();
        CapacityTb.Text = SectionList.CurrentRow.Cells[2].Value.ToString();
        DescTb.Text = SectionList.CurrentRow.Cells[3].Value.ToString();
        Key = Convert.ToInt32(SectionList.CurrentRow.Cells[0].Value);
    }
    else
    {
        Key = 0; // No valid selection
    }
}
```

```
private void delete_Click(object sender, EventArgs e)
{
    if (Key == 0)
    {
        MessageBox.Show("Select a section to delete.");
        return;
    }
    try
    {
        // Adjust the query to avoid specifying the identity column
        string Query = "DELETE FROM SectionTbl WHERE SNo = {0}";
        Query = string.Format(Query, Key);
        // Execute the delete command and get affected rows
        int rowsAffected = Con.SetData(Query);
        if (rowsAffected > 0)
        {
            MessageBox.Show("Section Deleted!!!");
            ShowSection(); // Refresh data
        }
        else
        {
            MessageBox.Show("No section found with the specified ID.");
        }
    }
    catch (Exception Ex)
    {
        MessageBox.Show("Error: " + Ex.Message);
    }
}
```

```
        private void edit_Click(object sender, EventArgs e)
        {
            try
            {
                string Name = SNameTb.Text;
                string Cap = CapacityTb.Text;
                string Desc = DescTb.Text;
                // Adjust the query to avoid specifying the identity column
                string Query = "Update SectionTbl set SName='{0}', Capacity={1},
SDescription='{2}' where SNo={3}";
                Query = string.Format(Query, Name, Cap, Desc, Key);
                // Execute the update command
                Con.SetData(Query);
                MessageBox.Show("Section Updated!!!");
                ShowSection();
            }
            catch (Exception ex)
            {
                // Show the error message
                MessageBox.Show("An error occurred while updating the section: " +
ex.Message);
            }
        }
        private void close_Click_1(object sender, EventArgs e)
        {
            // Close the application completely when the close button is clicked
            this.Close();
        }
    }
}
```

6.3.4. Code Introduction and Process

I. Class Definition and Constructor

```
public partial class sections : Form
{
    public sections()
    {
        InitializeComponent();
        Con = new Functions();
        ShowSection();
    }
    Functions Con;
}
```

The sections class is a partial class that inherits from the Form class, indicating that it is a Windows Forms application. In the constructor (sections()), it initializes the form's components using InitializeComponent() and creates an instance of the Functions class, which is likely responsible for database operations. The ShowSection() method is called to populate the data grid with sections when the form is first loaded.

II. ShowSection Method

```
private void ShowSection()
{
    string Query = "SELECT * FROM SectionTbl";
    SectionList.DataSource = Con.GetData(Query);
    SectionList.Refresh(); // Refresh the DataGridView if needed
}
```

The ShowSection() method is responsible for retrieving data from the database. It executes a SQL query to select all records from the SectionTbl table. The results are then set as the data source for SectionList, a DataGridView control. After updating the data source, the Refresh() method is called to ensure the DataGridView displays the most current data.

III. Add Button Click Event

```
private void add_Click(object sender, EventArgs e)
{
   if (SNameTb.Text == "" || CapacityTb.Text == "" || DescTb.Text == "")
   {
      MessageBox.Show("Missing data!!!");
   }
   else
   {
      try
      {
         string Name = SNameTb.Text;
         string Cap = CapacityTb.Text;
         string Desc = DescTb.Text;
         string Query = "INSERT INTO SectionTbl VALUES ('{0}', {1}, '{2}')";
         Query = string.Format(Query, Name, Cap, Desc);
         // Execute the insert command
         Con.SetData(Query);
         MessageBox.Show("Section Added!!!");
         ShowSection();
      }
      catch (Exception Ex)
      {
         MessageBox.Show(Ex.Message);
      }
   }
}
```

In the add_Click event handler, the method first checks if any of the required fields (SNameTb, CapacityTb, or DescTb) are empty. If any fields are missing, a message box

alerts the user. If all fields are filled, it retrieves the input data and constructs an SQL INSERT query to add a new section to the SectionTbl. After executing the query using Con.SetData(), it shows a success message and refreshes the DataGridView to display the updated data. If any exceptions occur during this process, an error message is displayed.

IV. Cell Content Click Event

```
private void SectionList_CellContentClick(object sender,
DataGridViewCellEventArgs e)
{
   // Check if a valid row is clicked
   if (e.RowIndex >= 0)
   {
      // Automatically select the entire row
      SectionList.CurrentRow.Selected = true;
      // Populate text boxes with the selected row's data
      SNameTb.Text = SectionList.CurrentRow.Cells[1].Value.ToString();
      CapacityTb.Text = SectionList.CurrentRow.Cells[2].Value.ToString();
      DescTb.Text = SectionList.CurrentRow.Cells[3].Value.ToString();
      Key = Convert.ToInt32(SectionList.CurrentRow.Cells[0].Value);
   }
   else
   {
      Key = 0; // No valid selection
   }
}
```

The SectionList_CellContentClick method is triggered when a cell in the SectionList DataGridView is clicked. If the click is valid (i.e., a row index greater than or equal to 0), the selected row is marked as selected. The data from the clicked row is then used to populate the text boxes (SNameTb, CapacityTb, and DescTb), allowing users to edit the section details. The Key variable is assigned the primary key (SNo) of the selected section, which will be used for update or delete operations. If no valid row is selected, Key is set to 0.

V. Delete Button Click Event

```
private void delete_Click(object sender, EventArgs e)
{
    if (Key == 0)
    {
        MessageBox.Show("Select a section to delete.");
        return;
    }
    // Ask for confirmation before deletion
    var confirmResult = MessageBox.Show("Are you sure you want to delete this section?",
                                        "Confirm Delete",
                                        MessageBoxButtons.YesNo);
    if (confirmResult == DialogResult.No)
    {
        // User chose not to delete
        return;
    }
    try
    {
        // Adjust the query to avoid specifying the identity column
        string Query = "DELETE FROM SectionTbl WHERE SNo = {0}";
        Query = string.Format(Query, Key);
        // Execute the delete command and get affected rows
        int rowsAffected = Con.SetData(Query);
        if (rowsAffected > 0)
        {
            MessageBox.Show("Section Deleted!!!");
            ShowSection(); // Refresh data
        }
```

```
        {
            MessageBox.Show("Section Deleted!!!");
            ShowSection(); // Refresh data
        }
        else
        {
            MessageBox.Show("No section found with the specified ID.");
        }
    }
    catch (Exception Ex)
    {
        MessageBox.Show("Error: " + Ex.Message);
    }
}
```

The delete_Click method handles the deletion of a section. It first checks if a section has been selected by verifying if Key is not zero. If no section is selected, a message prompts the user to select one. If a section is selected, it constructs a SQL DELETE query using the primary key and executes it. The number of affected rows is checked to confirm the deletion. If successful, a success message is displayed, and the section list is refreshed. If no section matches the specified ID, an appropriate message is shown. Any exceptions encountered during this process are caught and displayed.

VI. Edit Button Click Event

The edit_Click method is responsible for updating a selected section's details. It retrieves the values from the text boxes and constructs an SQL UPDATE query, targeting the section with the specified primary key. The query is executed using Con.SetData(). If the operation is successful, a message is displayed, and the section list is refreshed to show the updated data. Any errors encountered during the process are caught, and an error message is displayed to the user.

```
private void edit_Click(object sender, EventArgs e)
{
    try
    {
        string Name = SNameTb.Text;
        string Cap = CapacityTb.Text;
        string Desc = DescTb.Text;
        // Adjust the query to avoid specifying the identity column
        string Query = "Update SectionTbl set SName='{0}', Capacity={1}, SDescription='{2}' where SNo={3}";
        Query = string.Format(Query, Name, Cap, Desc, Key);
        // Execute the update command
        Con.SetData(Query);
        MessageBox.Show("Section Updated!!!");
        ShowSection();
    }
    catch (Exception ex)
    {
        // Show the error message
        MessageBox.Show("An error occurred while updating the section: " + ex.Message);
    }
}
```

6.3.5. Summary

Overall, the sections class manages sections within a vehicle parking management system. It allows users to view, add, edit, and delete sections, with the user interface driven by a DataGridView. Each operation is handled with error checking and feedback to the user, ensuring a smooth interaction experience.

6.4. Appendix D: Vehicles Interface

6.4.1. Overview

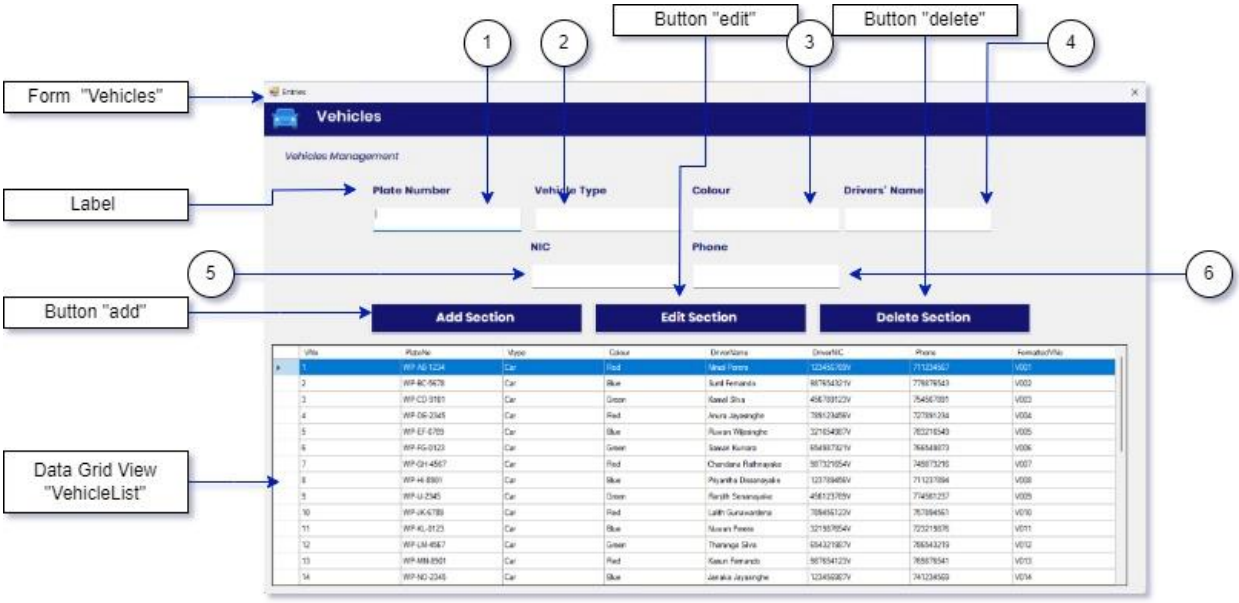


Figure 33-Vehicles Interface overview

1-TextBox “plateno”

2-TextBox “vtype”

3- TextBox “colour”

4- TextBox “drivername”

5- TextBox “nic”

6- TextBox “phone”

The interface form in the image showcases a well-structured layout for managing vehicle entries in a parking management system. This form is designed to provide users with an intuitive and efficient way to input and manage vehicle entry data.

At the top of the form, there is likely a title or header that indicates its purpose, such as “Vehicle Entry Management.” This helps users quickly identify the function of the form. Below the title, various input fields are arranged logically to streamline the data entry process.

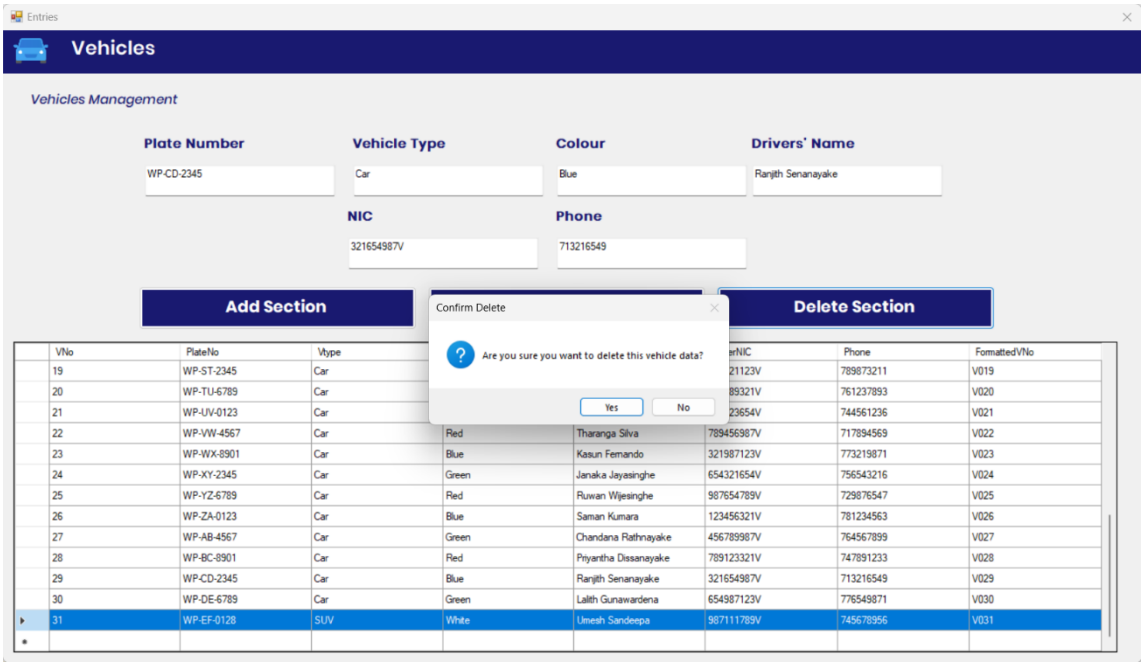
The first section of the form includes a ComboBox labeled "Section Number." This allows users to select the specific section in the parking lot where a vehicle will be parked. The use of a ComboBox ensures that users can only select from predefined options, reducing the risk of errors during data entry. Next to the "Section Number" ComboBox, there may

be additional fields such as "Vehicle Number" (labeled as "VNo") for users to enter the unique identification of the vehicle. This field could also be a ComboBox, enabling users to select from a list of registered vehicles.

The form also includes fields for entering the date and time of the vehicle entry. The "DateTimePicker" control, indicated in the instructions, allows users to easily select the appropriate date and time, enhancing user experience by providing a calendar view for date selection and a time picker for entering the time.

Furthermore, there might be buttons at the bottom of the form for actions such as "Submit," "Edit," "Delete," and "View." These buttons enable users to perform various operations, such as adding new entries, editing existing ones, deleting records, and viewing the list of parked vehicles. The placement of these buttons is critical for usability, as they should be easily accessible without requiring excessive scrolling or navigation.

Overall, this interface form effectively combines user-friendly design elements with practical functionality, making it an essential tool for managing vehicle entries in a parking management system. By organizing input fields and controls logically and intuitively, it aims to enhance user efficiency and reduce the likelihood of errors during data entry.



VNo	PlateNo	Vtype				Phone	FormattedVNo
19	WP-ST-2345	Car			21123V	789873211	V019
20	WP-TU-6789	Car			89321V	761237893	V020
21	WP-UV-0123	Car			23654V	744561236	V021
22	WP-VW-4567	Car	Red	Tharanga Silva	789456987V	717894569	V022
23	WP-WX-8901	Car	Blue	Kasun Fernando	321987123V	773219871	V023
24	WP-XY-2345	Car	Green	Janaka Jayasinghe	654321654V	756543216	V024
25	WP-YZ-6789	Car	Red	Ruwan Wijesinghe	987654789V	729876547	V025
26	WP-ZA-0123	Car	Blue	Saman Kumara	123456321V	781234563	V026
27	WP-AB-4567	Car	Green	Chandana Rathnayake	456789987V	764567899	V027
28	WP-BC-8901	Car	Red	Priyantha Dissanayake	789123321V	747891233	V028
29	WP-CD-2345	Car	Blue	Ranjith Senanayake	321654987V	713216549	V029
30	WP-DE-6789	Car	Green	Lalith Gunawardena	654987123V	776549871	V030
31	WP-EF-0128	SUV	White	Umesh Sandeepa	987111789V	745678956	V031

Figure 34-Vehicles page show message box when click Delete Button

6.4.2. Colors

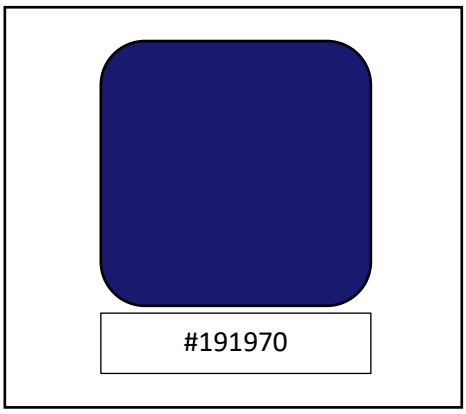


Figure 35-Color used in vehicles Interface

6.4.3. Full Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Vehicle_Parking_Management_System_Project
{
   public partial class vehicles : Form
   {
      public vehicles()
      {
         InitializeComponent();
         Con = new Functions();
         ShowSection();
```

```
    }
    Functions Con;
    private void ShowSection()
    {
        string Query = "SELECT * FROM CarTbl"; // Adjust table name if necessary
        VehicleList.DataSource = Con.GetData(Query);
        VehicleList.Refresh(); // Refresh the DataGridView if needed
    }
    private void entries_Load(object sender, EventArgs e)
    {


    }
    int Key = 0;


    private void VehicleList_CellContentClick(object sender,
DataGridViewCellEventArgs e)
    {
        if (e.RowIndex >= 0) // Ensure a row is selected
        {
            VehicleList.CurrentRow.Selected = true;
            plateno.Text = VehicleList.CurrentRow.Cells[1].Value.ToString();  // Plate
Number
            vtype.Text = VehicleList.CurrentRow.Cells[2].Value.ToString();    // Vehicle
Type
            colour.Text = VehicleList.CurrentRow.Cells[3].Value.ToString();  // Colour
            drivername.Text = VehicleList.CurrentRow.Cells[4].Value.ToString(); // Driver
Name
            nic.Text = VehicleList.CurrentRow.Cells[5].Value.ToString();      // Driver NIC
            phone.Text = VehicleList.CurrentRow.Cells[6].Value.ToString();     // Phone
            Key = Convert.ToInt32(VehicleList.CurrentRow.Cells[0].Value); // Assuming
the first column is the ID
```

```
        }
        else
        {
            Key = 0; // No valid selection
        }
    }
    private void add_Click(object sender, EventArgs e)
    {
        if (plateno.Text == "" || vtype.Text == "" || colour.Text == "" || drivername.Text == "" || nic.Text == "" || phone.Text == "")
        {
            MessageBox.Show("Missing data!!!");
        }
        else
        {
            try
            {
                string PlateNo = plateno.Text;
                string Vtype = vtype.Text;
                string Colour = colour.Text;
                string Name = drivername.Text;
                string Nic = nic.Text;
                string Pnum = phone.Text;

                // Correct the SQL query to properly format string values
                string Query = "INSERT INTO CarTbl (PlateNo, Vtype, Colour, DriverName, DriverNIC, Phone) VALUES ('{0}', '{1}', '{2}', '{3}', '{4}', '{5}')";
                Query = string.Format(Query, PlateNo, Vtype, Colour, Name, Nic, Pnum);

                // Execute the insert command
```

```
            Con.SetData(Query);
            MessageBox.Show("Vehicle Added!!!");
            ShowSection();
        }
        catch (Exception Ex)
        {
            MessageBox.Show(Ex.Message);
        }
    }
}
private void delete_Click(object sender, EventArgs e)
{
    if (Key == 0)
    {
        MessageBox.Show("Select a section to delete.");
        return;
    }
    // Show confirmation dialog
    var confirmResult = MessageBox.Show("Are you sure you want to delete this
vehicle data?",
                                        "Confirm Delete",
                                        MessageBoxButtons.YesNo,
                                        MessageBoxIcon.Question);

    if (confirmResult == DialogResult.Yes)
    {
        try
        {
            // Adjust the query to avoid specifying the identity column
```

```
            string Query = "DELETE FROM CarTbl WHERE VNo = {0}";
            Query = string.Format(Query, Key);

            // Execute the delete command and get affected rows
            int rowsAffected = Con.SetData(Query);

            if (rowsAffected > 0)
            {
                MessageBox.Show("Vehicle Data Deleted!!!");
                ShowSection(); // Refresh data
            }
            else
            {
                MessageBox.Show("No vehicle found with the specified ID.");
            }
        }
        catch (Exception Ex)
        {
            MessageBox.Show("Error: " + Ex.Message);
        }
    }
    else
    {
        // User chose not to delete
        MessageBox.Show("Deletion cancelled.");
    }
}
private void edit_Click(object sender, EventArgs e)
{
```

```
            try
            {
                string PlateNo = plateno.Text;
                string Vtype = vtype.Text;
                string Colour = colour.Text;
                string Name = drivername.Text;
                string Nic = nic.Text;
                string Pnum = phone.Text;
                // Adjust the query to update the selected entry
                string Query = "UPDATE CarTbl SET PlateNo = '{0}', Vtype = '{1}', Colour =
'{2}', DriverName = '{3}', DriverNIC = '{4}', Phone = '{5}' WHERE VNo = '{6}'";
                Query = string.Format(Query, PlateNo, Vtype, Colour, Name, Nic, Pnum, Key);
                // Execute the update command
                Con.SetData(Query);
                MessageBox.Show("Vehicle Entry Updated!!!");
                ShowSection();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
        private void panel3_Paint(object sender, PaintEventArgs e)
        {
            // This method is currently empty. You may want to implement any necessary
painting here.
        }
    }
}
```

6.4.4. Code Introduction and Process

I. Class Definition and Constructor

```
namespace Vehicle_Parking_Management_System_Project
{
    public partial class vehicles : Form
    {
        public vehicles()
        {
            InitializeComponent();
            Con = new Functions();
            ShowSection();
        }
```

This code starts with the declaration of a namespace Vehicle_Parking_Management_System_Project, which organizes classes and other types. Inside this namespace, the vehicles class is defined, which inherits from Form, indicating that it's a Windows Forms application. The constructor (public vehicles()) initializes the form components and creates an instance of the Functions class (assumed to handle database operations). It also calls the ShowSection method to populate the vehicle list when the form is loaded.

II. Function Declaration and Data Source Binding

```
Functions Con;


private void ShowSection()
{
    string Query = "SELECT * FROM CarTbl"; // Adjust table name if necessary
    VehicleList.DataSource = Con.GetData(Query);
    VehicleList.Refresh(); // Refresh the DataGridView if needed
}
```

Here, a member variable Con of type Functions is declared to interact with the database. The ShowSection method constructs a SQL query to select all records from the CarTbl table. It uses the Con.GetData(Query) method (assumed to execute the SQL query and

return the results) to set the data source of the VehicleList DataGridView, which displays vehicle entries. After setting the data source, VehicleList.Refresh() ensures that the DataGridView updates to reflect any changes.

III. Handling Cell Clicks in the DataGridView

```
private void VehicleList_CellContentClick(object sender,
DataGridViewCellEventArgs e)
{
   if (e.RowIndex >= 0) // Ensure a row is selected
   {
      VehicleList.CurrentRow.Selected = true;
      plateno.Text = VehicleList.CurrentRow.Cells[1].Value.ToString();  // Plate
Number
      vtype.Text = VehicleList.CurrentRow.Cells[2].Value.ToString();    // Vehicle
Type
      colour.Text = VehicleList.CurrentRow.Cells[3].Value.ToString();  // Colour
      drivername.Text = VehicleList.CurrentRow.Cells[4].Value.ToString(); // Driver
Name
      nic.Text = VehicleList.CurrentRow.Cells[5].Value.ToString();      // Driver NIC
      phone.Text = VehicleList.CurrentRow.Cells[6].Value.ToString();     // Phone
      Key = Convert.ToInt32(VehicleList.CurrentRow.Cells[0].Value); // Assuming
the first column is the ID
   }
   else
   {
      Key = 0; // No valid selection
   }
}
```

The VehicleList_CellContentClick method is triggered when a cell in the VehicleList DataGridView is clicked. It checks if the clicked row index is valid (greater than or equal to 0). If so, it marks the current row as selected and populates various text fields (plateno, vtype, colour, etc.) with values from the selected row. The Key variable is updated to store the ID of the selected vehicle (assumed to be in the first column). If the selection is invalid, Key is reset to 0.

IV. Adding a New Vehicle

```
private void add_Click(object sender, EventArgs e)
{
   if (plateno.Text == "" || vtype.Text == "" || colour.Text == "" || drivername.Text == "" || nic.Text == "" || phone.Text == "")
   {
      MessageBox.Show("Missing data!!!");
   }
   else
   {
      try
      {
         string PlateNo = plateno.Text;
         string Vtype = vtype.Text;
         string Colour = colour.Text;
         string Name = drivername.Text;
         string Nic = nic.Text;
         string Pnum = phone.Text;
         string Query = "INSERT INTO CarTbl (PlateNo, Vtype, Colour, DriverName, DriverNIC, Phone) VALUES ('{0}', '{1}', '{2}', '{3}', '{4}', '{5}')";
         Query = string.Format(Query, PlateNo, Vtype, Colour, Name, Nic, Pnum);
         Con.SetData(Query);
         MessageBox.Show("Vehicle Added!!!");
         ShowSection();
      }
      catch (Exception Ex)
      {
         MessageBox.Show(Ex.Message);
      }
   }
}
```

The add_Click method is executed when the "Add" button is clicked. It first checks if any of the required fields are empty and shows a warning message if so. If all fields have values, it constructs an SQL INSERT query to add a new vehicle entry to the CarTbl table using the values from the text fields. The Con.SetData(Query) method executes the insertion command. Upon successful addition, a confirmation message is displayed, and the vehicle list is refreshed by calling ShowSection().

V. Deleting a Vehicle Entry

```
private void delete_Click(object sender, EventArgs e)
{
   if (Key == 0)
   {
      MessageBox.Show("Select a section to delete.");
      return;
   }
   var confirmResult = MessageBox.Show("Are you sure you want to delete this vehicle data?",
                                       "Confirm Delete",
                                       MessageBoxButtons.YesNo,
                                       MessageBoxIcon.Question);
   if (confirmResult == DialogResult.Yes)
   {
      try
      {
         string Query = "DELETE FROM CarTbl WHERE VNo = {0}";
         Query = string.Format(Query, Key);
         int rowsAffected = Con.SetData(Query);
         if (rowsAffected > 0)
         {
            MessageBox.Show("Vehicle Data Deleted!!!");
            ShowSection(); // Refresh data
         }
         else
```

```
            {
                MessageBox.Show("No vehicle found with the specified ID.");
            }
        }
        catch (Exception Ex)
        {
            MessageBox.Show("Error: " + Ex.Message);
        }
    }
    else
    {
        // User chose not to delete
        MessageBox.Show("Deletion cancelled.");
    }
}
```

In the delete_Click method, the code checks if a vehicle is selected (i.e., Key is not 0). If not, it prompts the user to select a vehicle for deletion. It then shows a confirmation dialog to confirm the deletion action. If the user confirms, it constructs an SQL DELETE query to remove the vehicle from the CarTbl table based on the ID stored in Key. After executing the deletion, it checks how many rows were affected. If a vehicle was deleted, a success message is displayed, and the vehicle list is refreshed. If no rows were affected, it indicates that no vehicle was found. Any exceptions during this process are caught and displayed as error messages.

VI. Editing a Vehicle Entry

The edit_Click method is executed when the "Edit" button is clicked. It retrieves the current values from the text fields and constructs an SQL UPDATE query to modify the selected vehicle entry in the CarTbl table based on the ID stored in Key. It then executes the update command with Con.SetData(Query). If successful, a confirmation message is displayed, and the vehicle list is refreshed. Any exceptions during this process are caught and shown as error messages.

```
private void edit_Click(object sender, EventArgs e)
{
    try
    {
        string PlateNo = plateno.Text;
        string Vtype = vtype.Text;
        string Colour = colour.Text;
        string Name = drivername.Text;
        string Nic = nic.Text;
        string Pnum = phone.Text;
        // Adjust the query to update the selected entry
        string Query = "UPDATE CarTbl SET PlateNo = '{0}', Vtype = '{1}', Colour = '{2}', DriverName = '{3}', DriverNIC = '{4}', Phone = '{5}' WHERE VNo = '{6}'";
        Query = string.Format(Query, PlateNo, Vtype, Colour, Name, Nic, Pnum, Key);
        // Execute the update command
        Con.SetData(Query);
        MessageBox.Show("Vehicle Entry Updated!!!");
        ShowSection();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

6.4.5. Summary

Overall, the vehicles class handles the management of vehicle records in the parking system, allowing for the addition, deletion, and updating of vehicle information. It uses a DataGridView for displaying vehicle data, and the operations are performed via SQL commands executed through the Functions class, which is likely responsible for managing database connections and executing queries.

6.5. Appendix E: Entries Interface

6.5.1. Overview

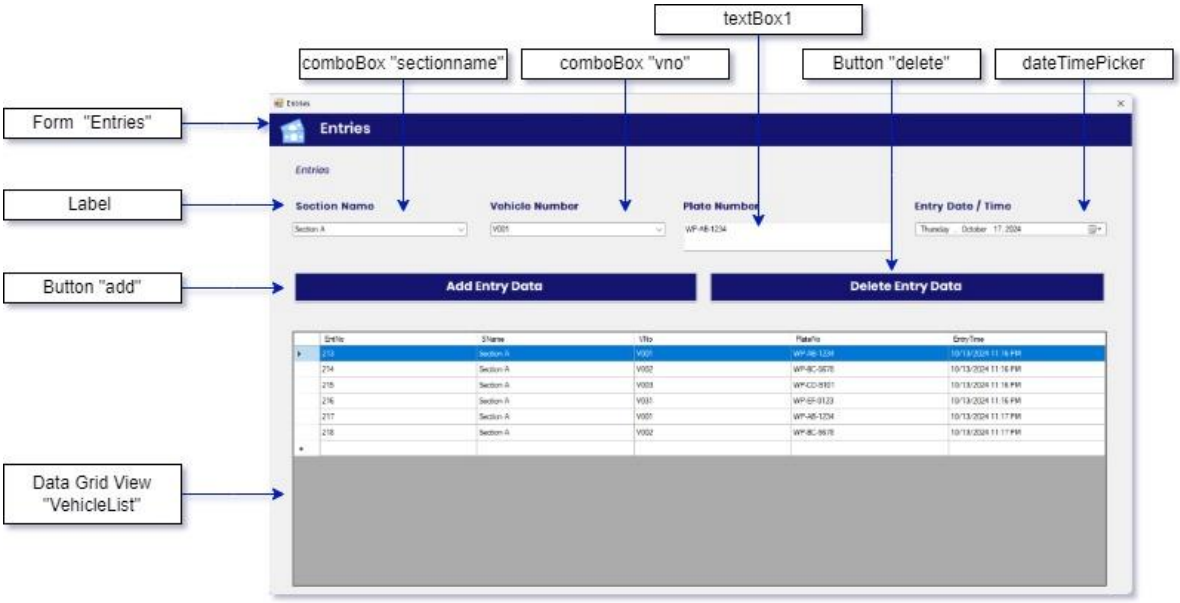


Figure 36-Entries Interface overview

This form, titled "Entries," is part of a Vehicle Parking Management System, designed to allow users to enter and manage vehicle data efficiently. It is divided into two primary sections: one for adding new vehicle entry data and another for displaying and managing existing entries.

At the top, there are several input fields aligned in a horizontal manner, making it user-friendly for data entry. The comboBox "sectionname" allows users to select a section where the vehicle is parked. The next field, comboBox "vno", is designated for the vehicle number, enabling users to pick from a list of pre-existing vehicle numbers. Adjacent to this, a textBox "Plate Number" captures the unique vehicle registration plate number for precise identification.

Further along, the dateTimePicker field enables users to select or input the specific date and time of entry, ensuring that each vehicle's time of arrival is accurately recorded.

There are two buttons below the input fields: the "Add Entry Data" button, which allows the user to add the inputted vehicle information into the system, and the "Delete Entry Data" button, which facilitates the removal of selected entries from the data grid.

Below these buttons is a Data Grid View labeled "VehicleList," which provides a table-like display of all existing vehicle entries. Each row in the table contains information such as the vehicle’s section, number, plate, and entry date/time, allowing the user to easily view and manage the parked vehicles.

This layout promotes ease of use, with clearly labeled fields and buttons, ensuring efficient data entry and retrieval in the vehicle management system.

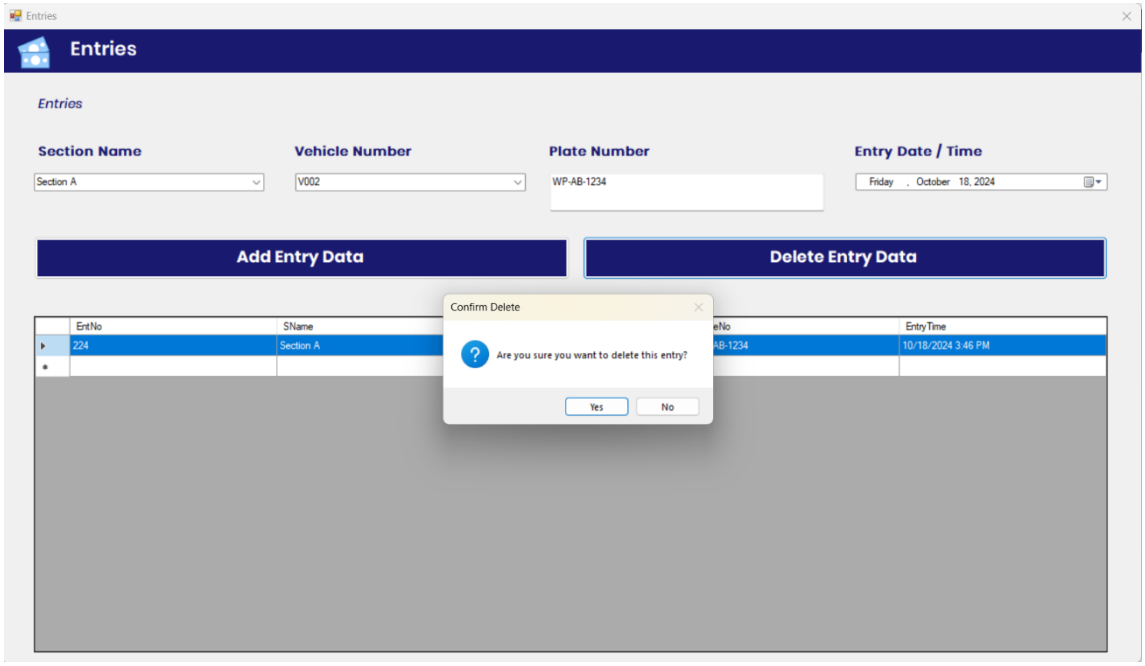


Figure 37-Entries page show message box when click Delete Button

6.5.2. Colors

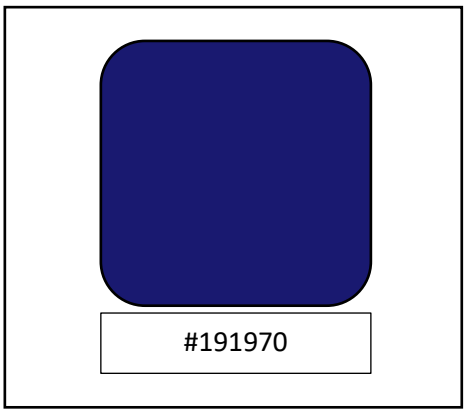


Figure 38-Color used in vehicles Interface

6.5.3. Full Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Data.SqlClient;
namespace Vehicle_Parking_Management_System_Project
{
    public partial class Entries : Form
    {
        public Entries()
        {
            InitializeComponent();
            Con = new Functions();
            ShowSection();
            LoadSections();
            LoadVehicleNumbers(); // Call the method to load vehicle numbers
            vno.SelectedIndexChanged += new EventHandler(vno_SelectedIndexChanged);
            this.Load += new EventHandler(Form2_Load);
        }
        Functions Con;
        private void ShowSection()
        {
            string Query = "SELECT * FROM EntryTbl";
            EntryList.DataSource = Con.GetData(Query);
            EntryList.Refresh();
        }
        private void LoadSections()
        {
            string Query1 = "SELECT SName FROM SectionTbl";
```

```
        DataTable dt = Con.GetData(Query1);
        // Bind the data to ComboBox1
        sectionname.DataSource = dt;
        sectionname.DisplayMember = "SName";
        sectionname.ValueMember = "SName";  // If needed, this can be changed to a
section ID
    }
    private void LoadVehicleNumbers()
    {
        string Query2 = "SELECT FormattedVNo FROM CarTbl";
        DataTable dt1 = Con.GetData(Query2);
        // Bind the data to the ComboBox named 'vno'
        vno.DataSource = dt1;
        vno.DisplayMember = "FormattedVNo";
        vno.ValueMember = "FormattedVNo";
    }
    // Event handler for when a vehicle number is selected
    private void vno_SelectedIndexChanged(object sender, EventArgs e)
    {
        if (vno.SelectedItem != null)
        {
            string selectedVehicleNo = vno.SelectedValue.ToString();
            // Query to fetch the plate number based on the selected vehicle number
            string Query3 = "SELECT PlateNo FROM CarTbl WHERE FormattedVNo = '"
+ selectedVehicleNo + "'";
            DataTable dt = Con.GetData(Query3);
            if (dt.Rows.Count > 0)
            {
                // Assuming PlateNo is in the first column of the result
                textBox1.Text = dt.Rows[0]["PlateNo"].ToString();
```

```
            }
        }
    }
    private void add_Click(object sender, EventArgs e)
    {
        if (string.IsNullOrWhiteSpace(textBox1.Text) || sectionname.SelectedItem == null
|| vno.SelectedItem == null)
        {
            MessageBox.Show("Missing data!!!");
        }
        else
        {
            try
            {
                string SectionName = sectionname.SelectedValue.ToString();
                string VehicleNumber = vno.SelectedValue.ToString();
                string PlateNo = textBox1.Text;
                // Get the selected date and time from dateTimePicker
                DateTime selectedDateTime = dateTimePicker.Value;
                string formattedEntryTime = selectedDateTime.ToString("yyyy-MM-dd
HH:mm:ss");
                string query = "INSERT INTO EntryTbl (SName, VNo, PlateNo, EntryTime)
VALUES ('{0}', '{1}', '{2}', '{3}')";
                query = string.Format(query, SectionName, VehicleNumber, PlateNo,
formattedEntryTime);
                // Execute the insert command
                Con.SetData(query);
                MessageBox.Show("Entry Added!!!");
                // Refresh the data display and update the vehicle numbers
                ShowSection();
                UpdateVehicleNumbers(); // Refresh the ComboBox
```

```
            }
            catch (Exception Ex)
            {
               MessageBox.Show(Ex.Message);
            }
         }
      }
      int Key = 0;
      private void EntryList_CellContentClick(object sender,
DataGridViewCellEventArgs e)
      {
         // Check if a valid row is clicked
         if (e.RowIndex >= 0)
         {
            // Automatically select the entire row
            EntryList.CurrentRow.Selected = true;
            // Populate text boxes with the selected row's data
            sectionname.SelectedItem = EntryList.CurrentRow.Cells[1].Value.ToString();
            vno.SelectedItem = EntryList.CurrentRow.Cells[2].Value.ToString();
            textBox1.Text = EntryList.CurrentRow.Cells[3].Value.ToString();
            Key = Convert.ToInt32(EntryList.CurrentRow.Cells[0].Value);
         }
         else
         {
            Key = 0; // No valid selection
         }
      }
      private void delete_Click_1(object sender, EventArgs e)
      {
```

```
if (Key == 0)
{
   MessageBox.Show("Select a section to delete.");
   return;
}
// Show a confirmation dialog
var result = MessageBox.Show("Are you sure you want to delete this entry?",
                             "Confirm Delete",
                             MessageBoxButtons.YesNo,
                             MessageBoxIcon.Question);
// Check the user's response
if (result == DialogResult.Yes)
{
   try
   {
      // Adjust the query to avoid specifying the identity column
      string Query = "DELETE FROM EntryTbl WHERE EntNo = {0}";
      Query = string.Format(Query, Key);
      // Execute the delete command and get affected rows
      int rowsAffected = Con.SetData(Query);
      if (rowsAffected > 0)
      {
         MessageBox.Show("Entry Deleted!!!");
         ShowSection(); // Refresh data
      }
      else
      {
         MessageBox.Show("No section found with the specified ID.");
      }
```

```
        }
        catch (Exception Ex)
        {
            MessageBox.Show("Error: " + Ex.Message);
        }
    }
    else
    {
        MessageBox.Show("Deletion cancelled.");
    }
}
private void Form2_Load(object sender, EventArgs e)
{
    UpdateVehicleNumbers(); // Call this to set up the ComboBox on form load
}
private void close_Click(object sender, EventArgs e)
{
    this.Close();
}
private void UpdateVehicleNumbers()
{
    string query = "SELECT FormattedVNo FROM CarTbl";
    DataTable dt1 = Con.GetData(query);
    // Create a list of vehicle numbers that are currently in the EntryTbl
    List<string> occupiedVNos = new List<string>();
    // Check EntryTbl for occupied vehicle numbers
    string entryQuery = "SELECT VNo FROM EntryTbl";
    DataTable entryDt = Con.GetData(entryQuery);
    foreach (DataRow row in entryDt.Rows)
```

```
        {
            occupiedVNos.Add(row["VNo"].ToString());
        }
        // Check ExitTbl for vehicle numbers that can be shown again
        string exitQuery = "SELECT VNo FROM ExitTbl";
        DataTable exitDt = Con.GetData(exitQuery);
        foreach (DataRow row in exitDt.Rows)
        {
            string vNo = row["VNo"].ToString();
            if (occupiedVNos.Contains(vNo))
            {
                occupiedVNos.Remove(vNo); // Remove it if it's in the exit table
            }
        }
        // Filter the vehicle numbers based on their occupancy status
        DataTable availableVNos = new DataTable();
        availableVNos.Columns.Add("FormattedVNo", typeof(string));
        foreach (DataRow row in dt1.Rows)
        {
            string formattedVNo = row["FormattedVNo"].ToString();
            if (!occupiedVNos.Contains(formattedVNo)) // Only add non-occupied vehicle
numbers
            {
                availableVNos.Rows.Add(formattedVNo);
            }
        }
        // Bind the filtered data to the ComboBox
        vno.DataSource = availableVNos;
        vno.DisplayMember = "FormattedVNo";
```

```
            vno.ValueMember = "FormattedVNo";
        }
    }
}
```

6.5.4. Code Introduction and Process

I. Class Definition and Constructor

```
public partial class Entries : Form
{
    public Entries()
    {
        InitializeComponent();
        Con = new Functions();
        ShowSection();
        LoadSections();
        LoadVehicleNumbers(); // Call the method to load vehicle numbers
        vno.SelectedIndexChanged += new EventHandler(vno_SelectedIndexChanged);
        this.Load += new EventHandler(Form2_Load);
    }
```

The Entries class is a form that handles the entries for the vehicle parking management system. It inherits from Form, which allows it to function as a Windows Form application. In the constructor, various components are initialized using InitializeComponent(), and an instance of the Functions class is created to interact with the database. The ShowSection, LoadSections, and LoadVehicleNumbers methods are called to populate the data displayed in the form. Additionally, event handlers are attached to the SelectedIndexChanged event of the vehicle number combo box (vno) and the Load event of the form itself.

II. Database Interaction Methods

```
private void ShowSection()
{
    string Query = "SELECT * FROM EntryTbl";
    EntryList.DataSource = Con.GetData(Query);
    EntryList.Refresh();
}
```

This method retrieves all records from the EntryTbl table in the database and displays them in the EntryList DataGridView. The Con.GetData(Query) method is assumed to execute the SQL query and return the results as a DataTable. After setting the DataSource, EntryList.Refresh() is called to update the display.

```
private void LoadSections()
{
    string Query1 = "SELECT SName FROM SectionTbl";
    DataTable dt = Con.GetData(Query1);
    // Bind the data to ComboBox1
    sectionname.DataSource = dt;
    sectionname.DisplayMember = "SName";
    sectionname.ValueMember = "SName";  // If needed, this can be changed to a section ID
}
```

The LoadSections method populates a combo box (sectionname) with section names from the SectionTbl. It retrieves the section names and binds them to the combo box, allowing the user to select a section easily.

```
private void LoadVehicleNumbers()
{
    string Query2 = "SELECT FormattedVNo FROM CarTbl";
    DataTable dt1 = Con.GetData(Query2);
    // Bind the data to the ComboBox named 'vno'
    vno.DataSource = dt1;
    vno.DisplayMember = "FormattedVNo";
    vno.ValueMember = "FormattedVNo";
}
```

This method loads formatted vehicle numbers from the CarTbl and binds them to the vno combo box, which allows users to select a vehicle number. The DisplayMember and ValueMember properties specify what to show in the combo box and what value to return when an item is selected.

III. Event Handlers

```
private void vno_SelectedIndexChanged(object sender, EventArgs e)
{
    if (vno.SelectedItem != null)
    {
        string selectedVehicleNo = vno.SelectedValue.ToString();
        // Query to fetch the plate number based on the selected vehicle number
        string Query3 = "SELECT PlateNo FROM CarTbl WHERE FormattedVNo = '"
+ selectedVehicleNo + "'";
        DataTable dt = Con.GetData(Query3);
        if (dt.Rows.Count > 0)
        {
            // Assuming PlateNo is in the first column of the result
            textBox1.Text = dt.Rows[0]["PlateNo"].ToString();
        }
    }
}
```

The vno_SelectedIndexChanged method is an event handler that triggers when a user selects a vehicle number from the vno combo box. It checks if an item is selected and retrieves the corresponding plate number from the CarTbl based on the selected formatted vehicle number. The plate number is then displayed in textBox1.

IV. Adding Entries

```
private void add_Click(object sender, EventArgs e)
{
   if (string.IsNullOrWhiteSpace(textBox1.Text) || sectionname.SelectedItem == null || vno.SelectedItem == null)
   {
      MessageBox.Show("Missing data!!!");
   }
   else
   {
      try
      {
         string SectionName = sectionname.SelectedValue.ToString();
         string VehicleNumber = vno.SelectedValue.ToString();
         string PlateNo = textBox1.Text;
         DateTime selectedDateTime = dateTimePicker.Value;
         string formattedEntryTime = selectedDateTime.ToString("yyyy-MM-dd HH:mm:ss");
         string query = "INSERT INTO EntryTbl (SName, VNo, PlateNo, EntryTime) VALUES ('{0}', '{1}', '{2}', '{3}')";
         query = string.Format(query, SectionName, VehicleNumber, PlateNo, formattedEntryTime);
         // Execute the insert command
         Con.SetData(query);
         MessageBox.Show("Entry Added!!!");
         ShowSection();
         UpdateVehicleNumbers(); // Refresh the ComboBox
      }
      catch (Exception Ex)
```

The add_Click method handles the addition of new entries to the parking system. It checks if required fields are filled; if not, it displays a warning message. If all fields are valid, it retrieves the selected section name, vehicle number, plate number, and entry time from the dateTimePicker. An SQL INSERT command is constructed and executed using Con.SetData(query). Upon successful addition, it shows a confirmation message, refreshes the displayed entries, and updates the vehicle numbers in the combo box.

V. Selecting Entries

```
private void EntryList_CellContentClick(object sender, DataGridViewCellEventArgs
e)
{
   // Check if a valid row is clicked
   if (e.RowIndex >= 0)
   {
      // Automatically select the entire row
      EntryList.CurrentRow.Selected = true;

      // Populate text boxes with the selected row's data
      sectionname.SelectedItem = EntryList.CurrentRow.Cells[1].Value.ToString();
      vno.SelectedItem = EntryList.CurrentRow.Cells[2].Value.ToString();
      textBox1.Text = EntryList.CurrentRow.Cells[3].Value.ToString();

      Key = Convert.ToInt32(EntryList.CurrentRow.Cells[0].Value);
   }
   else
   {
      Key = 0; // No valid selection
   }
}
```

The EntryList_CellContentClick method handles clicks on rows in the EntryList DataGridView. When a row is clicked, it checks if the row index is valid, selects the entire row, and populates the form fields (sectionname, vno, and textBox1) with the

corresponding values from the clicked row. The Key variable is set to the entry ID for later use, such as when deleting an entry.

VI. Deleting Entries

```
private void delete_Click_1(object sender, EventArgs e)
{
    if (Key == 0)
    {
        MessageBox.Show("Select a section to delete.");
        return;
    }
    // Show a confirmation dialog
    var result = MessageBox.Show("Are you sure you want to delete this entry?",
                                 "Confirm Delete",
                                 MessageBoxButtons.YesNo,
                                 MessageBoxIcon.Question);
    // Check the user's response
    if (result == DialogResult.Yes)
    {
        try
        {
            // Adjust the query to avoid specifying the identity column
            string Query = "DELETE FROM EntryTbl WHERE EntNo = {0}";
            Query = string.Format(Query, Key);
            // Execute the delete command and get affected rows
            int rowsAffected = Con.SetData(Query);
            if (rowsAffected > 0)
            {
                MessageBox.Show("Entry Deleted!!!");
                ShowSection(); // Refresh data
            }
```

```
            else
            {
                MessageBox.Show("No section found with the specified ID.");
            }
        }
        catch (Exception Ex)
        {
            MessageBox.Show("Error: " + Ex.Message);
        }
    }
    else
    {
        MessageBox.Show("Deletion cancelled.");
    }
}
```

The delete_Click_1 method facilitates the deletion of an entry. It first checks if a valid entry is selected using the Key variable. If not, it prompts the user to select an entry to delete. If an entry is selected, a confirmation dialog appears. If the user confirms the deletion, an SQL DELETE command is executed to remove the entry from the EntryTbl. A message indicates whether the deletion was successful, and the data is refreshed afterward.

VII. Form Load Event

```
private void Form2_Load(object sender, EventArgs e)
{
    UpdateVehicleNumbers(); // Call this to set up the ComboBox on form load
}
```

The Form2_Load method runs when the form loads. It calls the UpdateVehicleNumbers method to ensure that the vehicle numbers combo box is populated with the latest data when the form is displayed.

6.5.5. Summary

Overall, this code defines a Windows Form for managing vehicle entries, allowing users to add, view, and delete vehicle parking entries while dynamically updating available vehicle numbers and sections. The use of data binding and event handling makes it interactive and user-friendly, facilitating efficient data management within the vehicle parking management system.

6.6. Appendix F: Exit Interface

6.6.1. Overview

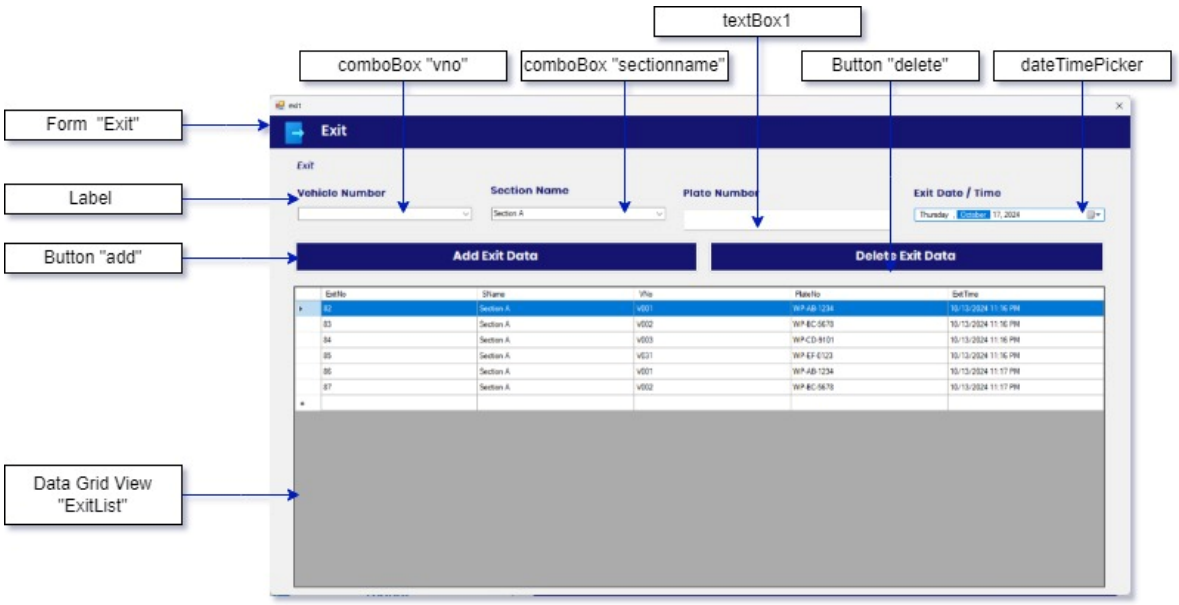


Figure 39-Exit Interface overview

The design of the "Exit" form consists of several elements that facilitate vehicle exit management in a parking system. At the top, the form is labeled "Exit," clearly indicating its purpose. Below the label, the form is divided into sections for user input and data display.

On the left side, there are two combo boxes: the first labeled "Vehicle Number" (comboBox "vno") and the second labeled "Section Name" (comboBox "sectionname"). These fields allow the user to select the vehicle's number and the section where it was parked.

Next, there's a text box (textBox1) for entering the vehicle's plate number, which ensures an additional identifier for the vehicle. On the right side of the form, a dateTimePicker is included to record the exit date and time, providing accurate logging of when a vehicle leaves the parking area.

In the middle of the form, two buttons are positioned: one labeled "Add" to insert exit data and the other labeled "Delete" for removing exit information. These actions enable the user to manage exit records efficiently.

At the bottom of the form, a Data Grid View named "ExitList" is used to display a list of vehicles and their corresponding exit details, such as the section, vehicle number, and exit time. This grid offers a visual summary of past entries and allows users to easily verify and manage exit data.

The overall layout is simple and functional, aimed at streamlining the process of managing vehicle exits from a parking system.

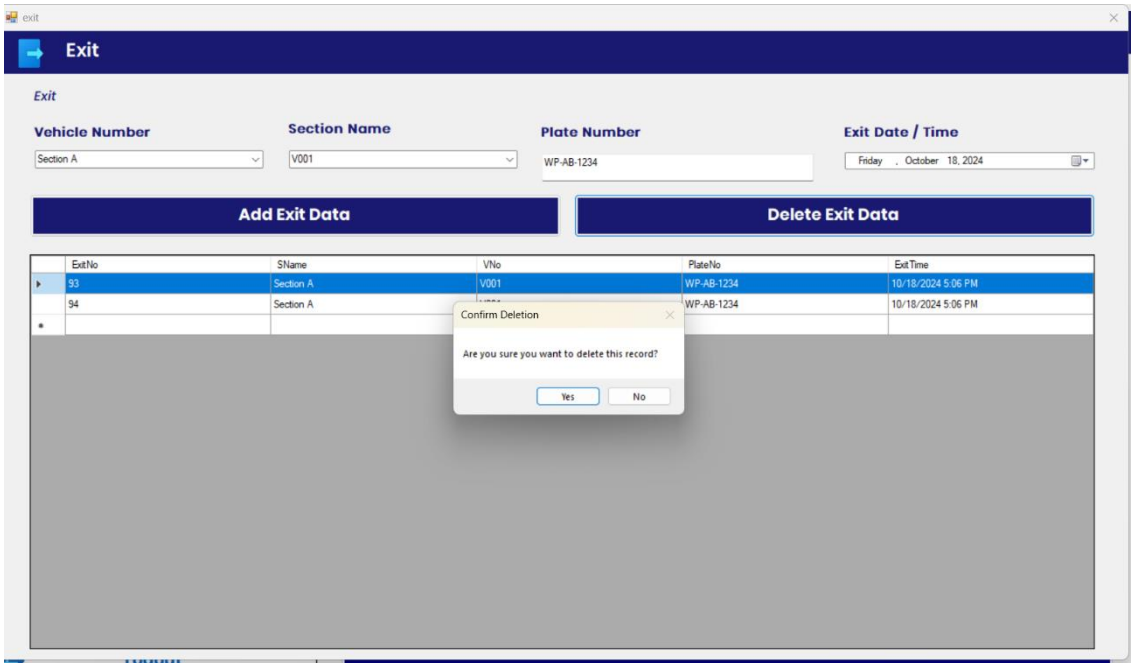


Figure 40-Exit page show message box when click Delete Button

6.6.2. Colors

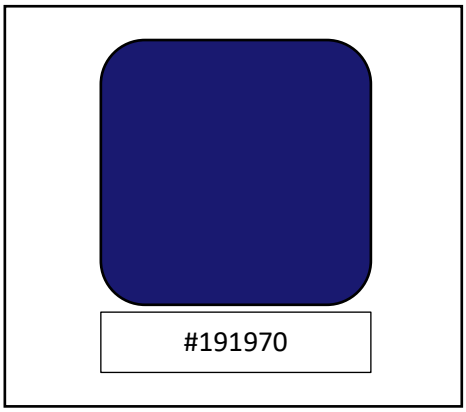


Figure 41-Color used in exit Interface

6.6.3. Full Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Vehicle_Parking_Management_System_Project
{
   public partial class exit : Form
   {
      public exit()
      {
         InitializeComponent();
         Con = new Functions();
         ShowSection();
         LoadSections();
         UpdateVehicleNumbers(); // Call the method to load vehicle numbers initially
         vno.SelectedIndexChanged += new EventHandler(vno_SelectedIndexChanged);
         this.Load += new EventHandler(exit_Load);
      }
      Functions Con;
      int Key = 0;

      private void exit_Load(object sender, EventArgs e)
```

```
{
    UpdateVehicleNumbers();
}
private void UpdateVehicleNumbers()
{
    // Get all vehicle numbers from EntryTbl
    string query = "SELECT VNo FROM EntryTbl";
    DataTable entryDt = Con.GetData(query);
    List<string> availableVNos = new List<string>();
    // Add all vehicle numbers from EntryTbl to the list
    foreach (DataRow row in entryDt.Rows)
    {
        availableVNos.Add(row["VNo"].ToString());
    }
    // Get vehicle numbers from ExitTbl to remove from available list
    string exitQuery = "SELECT VNo FROM ExitTbl";
    DataTable exitDt = Con.GetData(exitQuery);
    foreach (DataRow row in exitDt.Rows)
    {
        string vNo = row["VNo"].ToString();
        if (availableVNos.Contains(vNo))
        {
            availableVNos.Remove(vNo); // Remove vehicle numbers that are in ExitTbl
        }
    }
    // Bind the remaining vehicle numbers to the ComboBox
    vno.DataSource = availableVNos;
}
private void ShowSection()
```

```
        {
            string Query = "SELECT * FROM ExitTbl";
            ExitList.DataSource = Con.GetData(Query);
            ExitList.Refresh();
        }
        private void LoadSections()
        {
            string Query1 = "SELECT SName FROM SectionTbl";
            DataTable dt = Con.GetData(Query1);
            sectionname.DataSource = dt;
            sectionname.DisplayMember = "SName";
            sectionname.ValueMember = "SName";
        }
        private void vno_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (vno.SelectedItem != null)
            {
                string selectedVehicleNo = vno.SelectedValue.ToString();
                string Query3 = "SELECT PlateNo, SName FROM EntryTbl WHERE VNo =
'" + selectedVehicleNo + "'";
                DataTable dt = Con.GetData(Query3);
                if (dt.Rows.Count > 0)
                {
                    textBox1.Text = dt.Rows[0]["PlateNo"].ToString();
                    sectionname.SelectedValue = dt.Rows[0]["SName"].ToString();
                }
            }
        }
        private void add_Click(object sender, EventArgs e)
```

```
    {
        if (string.IsNullOrWhiteSpace(textBox1.Text) || sectionname.SelectedItem == null
|| vno.SelectedItem == null)
        {
            MessageBox.Show("Missing data!!!");
        }
        else
        {
            try
            {
                string SectionName = sectionname.SelectedValue.ToString();
                string VehicleNumber = vno.SelectedValue.ToString();
                string PlateNo = textBox1.Text;
                DateTime selectedDateTime = dateTimePicker.Value;
                string formattedExitTime = selectedDateTime.ToString("yyyy-MM-dd
HH:mm:ss");
                string Query = "INSERT INTO ExitTbl (SName, VNo, PlateNo, ExitTime)
VALUES ('{0}', '{1}', '{2}', '{3}')";
                Query = string.Format(Query, SectionName, VehicleNumber, PlateNo,
formattedExitTime);
                Con.SetData(Query);
                MessageBox.Show("Exit Data Added!!!");
                // Refresh the data display and vehicle numbers
                ShowSection();
                UpdateVehicleNumbers(); // Reload vehicle numbers after adding to ExitTbl
            }
            catch (Exception Ex)
            {
                MessageBox.Show(Ex.Message);
            }
        }
```

```
    }
    private void EntryList_CellContentClick_1(object sender,
DataGridViewCellEventArgs e)
    {
        // Check if a valid row is clicked
        if (e.RowIndex >= 0)
        {
            ExitList.CurrentRow.Selected = true;
            // Populate text boxes with the selected row's data
            vno.SelectedItem = ExitList.CurrentRow.Cells[1].Value.ToString();
            sectionname.SelectedItem = ExitList.CurrentRow.Cells[2].Value.ToString();
            textBox1.Text = ExitList.CurrentRow.Cells[3].Value.ToString();
            Key = Convert.ToInt32(ExitList.CurrentRow.Cells[0].Value);
        }
        else
        {
            Key = 0; // No valid selection
        }
    }
    private void delete_Click(object sender, EventArgs e)
    {
        if (Key == 0)
        {
            MessageBox.Show("Select a record to delete.");
            return;
        }
        // Ask user for confirmation
        DialogResult result = MessageBox.Show("Are you sure you want to delete this
record?", "Confirm Deletion", MessageBoxButtons.YesNo);
```

```
        // If user clicks "Yes", proceed with deletion
        if (result == DialogResult.Yes)
        {
            try
            {
                string Query = "DELETE FROM ExitTbl WHERE ExitNo = {0}";
                Query = string.Format(Query, Key);
                Con.SetData(Query); // Execute the delete query
                MessageBox.Show("Record Deleted!!!");
                ShowSection();  // Refresh the section data
                UpdateVehicleNumbers();  // Reload vehicle numbers after deletion
            }
            catch (Exception Ex)
            {
                MessageBox.Show("Error: " + Ex.Message);
            }
        }
        // If user clicks "No", the code exits the method here, and nothing happens
    }
    private void ExitList_CellContentClick(object sender, DataGridViewCellEventArgs
e)
    {
        // Check if a valid row is clicked
        if (e.RowIndex >= 0)
        {
            ExitList.CurrentRow.Selected = true;


            // Populate text boxes with the selected row's data
            vno.Text = ExitList.CurrentRow.Cells[1].Value.ToString(); // Set the vehicle
number in the ComboBox
```

```
            sectionname.Text = ExitList.CurrentRow.Cells[2].Value.ToString(); // Set the section name in the ComboBox
            textBox1.Text = ExitList.CurrentRow.Cells[3].Value.ToString(); // Set the PlateNo in the TextBox


            // Assign the ExitNo (primary key) from the selected row to 'Key'
            Key = Convert.ToInt32(ExitList.CurrentRow.Cells[0].Value);
        }
        else
        {
            Key = 0; // No valid selection
        }
    }
  }
}
```

6.6.4. Code Introduction and Process

I. Class Definition and Constructor

```
    public exit()
    {
        InitializeComponent();
        Con = new Functions();
        ShowSection();
        LoadSections();
        UpdateVehicleNumbers(); // Call the method to load vehicle numbers initially
        vno.SelectedIndexChanged += new EventHandler (vno_SelectedIndexChanged);
        this.Load += new EventHandler(exit_Load);
    }
    Functions Con;
    int Key = 0;
```

The exit class is a form within the Vehicle_Parking_Management_System_Project namespace. The constructor (exit()) initializes the form by calling InitializeComponent() to set up form controls and binds database-related methods like ShowSection(), LoadSections(), and UpdateVehicleNumbers() to load data from the database.

It also assigns an event handler to vno.SelectedIndexChanged, which will trigger when the selected vehicle number changes in the ComboBox. The this.Load event ensures that UpdateVehicleNumbers() is called when the form is loaded.

II. Loading and Updating Vehicle Numbers

```
private void exit_Load(object sender, EventArgs e)
{
    UpdateVehicleNumbers();
}
private void UpdateVehicleNumbers()
{
    string query = "SELECT VNo FROM EntryTbl";
    DataTable entryDt = Con.GetData(query);
    List<string> availableVNos = new List<string>();
    foreach (DataRow row in entryDt.Rows)
    {
        availableVNos.Add(row["VNo"].ToString());
    }
    string exitQuery = "SELECT VNo FROM ExitTbl";
    DataTable exitDt = Con.GetData(exitQuery);
    foreach (DataRow row in exitDt.Rows)
    {
        string vNo = row["VNo"].ToString();
        if (availableVNos.Contains(vNo))
        {
            availableVNos.Remove(vNo);
        }
```

```
    }


    vno.DataSource = availableVNos;
}
```

The UpdateVehicleNumbers() method fetches vehicle numbers from the EntryTbl using the query "SELECT VNo FROM EntryTbl". These vehicle numbers are stored in a list, availableVNos. Then, it retrieves vehicle numbers from ExitTbl and removes them from the list, assuming that vehicles already in the ExitTbl have exited the parking lot. Finally, the remaining vehicle numbers are set as the data source for the vno ComboBox, making them available for exit processing.

III. Show and Load Section Data

```
private void ShowSection()
{
    string Query = "SELECT * FROM ExitTbl";
    ExitList.DataSource = Con.GetData(Query);
    ExitList.Refresh();
}
private void LoadSections()
{
    string Query1 = "SELECT SName FROM SectionTbl";
    DataTable dt = Con.GetData(Query1);
    sectionname.DataSource = dt;
    sectionname.DisplayMember = "SName";
    sectionname.ValueMember = "SName";
}
```

ShowSection(): This method retrieves all records from ExitTbl and binds the result to a DataGridView named ExitList, which displays the exit records on the form. LoadSections(): This method populates the sectionname ComboBox with section names from SectionTbl. It uses the DisplayMember and ValueMember properties to display and select the SName field in the ComboBox.

IV. Handling Vehicle Selection

```
private void vno_SelectedIndexChanged(object sender, EventArgs e)
{
   if (vno.SelectedItem != null)
   {
      string selectedVehicleNo = vno.SelectedValue.ToString();
      string Query3 = "SELECT PlateNo, SName FROM EntryTbl WHERE VNo = '"
+ selectedVehicleNo + "'";
      DataTable dt = Con.GetData(Query3);

      if (dt.Rows.Count > 0)
      {
         textBox1.Text = dt.Rows[0]["PlateNo"].ToString();
         sectionname.SelectedValue = dt.Rows[0]["SName"].ToString();
      }
   }
}
```

This method triggers when a user selects a vehicle number from the vno ComboBox. It retrieves the vehicle's plate number (PlateNo) and the section name (SName) from EntryTbl where the selected vehicle number matches VNo. These values are then displayed in a TextBox (textBox1) and the sectionname ComboBox, respectively.

V. Adding an Exit Record

The add_Click method is triggered when the user clicks the "Add" button to record an exit. It checks if the required data (vehicle number, plate number, section name) is provided. If any data is missing, it shows an error message. If all data is present, it formats the exit time (dateTimePicker.Value) and inserts the data into the ExitTbl table.

After the record is added, it calls ShowSection() to refresh the displayed exit records and UpdateVehicleNumbers() to update the list of available vehicle numbers.

```
private void add_Click(object sender, EventArgs e)
{
   if (string.IsNullOrWhiteSpace(textBox1.Text) || sectionname.SelectedItem == null || vno.SelectedItem == null)
   {
      MessageBox.Show("Missing data!!!");
   }
   else
   {
      try
      {
         string SectionName = sectionname.SelectedValue.ToString();
         string VehicleNumber = vno.SelectedValue.ToString();
         string PlateNo = textBox1.Text;
         DateTime selectedDateTime = dateTimePicker.Value;
         string formattedExitTime = selectedDateTime.ToString("yyyy-MM-dd HH:mm:ss");
         string Query = "INSERT INTO ExitTbl (SName, VNo, PlateNo, ExitTime) VALUES ('{0}', '{1}', '{2}', '{3}')";
         Query = string.Format(Query, SectionName, VehicleNumber, PlateNo, formattedExitTime);
         Con.SetData(Query);
         MessageBox.Show("Exit Data Added!!!");
         ShowSection();
         UpdateVehicleNumbers();
      }
      catch (Exception Ex)
      {
         MessageBox.Show(Ex.Message);
      }
   }
}
```

VI. Adding an Exit Record

```
private void ExitList_CellContentClick(object sender, DataGridViewCellEventArgs e)
{
    if (e.RowIndex >= 0)
    {
        ExitList.CurrentRow.Selected = true;
        vno.Text = ExitList.CurrentRow.Cells[1].Value.ToString();
        sectionname.Text = ExitList.CurrentRow.Cells[2].Value.ToString();
        textBox1.Text = ExitList.CurrentRow.Cells[3].Value.ToString();
        Key = Convert.ToInt32(ExitList.CurrentRow.Cells[0].Value);
    }
    else
    {
        Key = 0;
    }
}
```

When a user clicks a row in the ExitList DataGridView, this method is triggered. It retrieves the vehicle number, section name, and plate number from the selected row and displays them in the corresponding controls (vno, sectionname, and textBox1). The primary key (ExitNo) is stored in the Key variable to track which record is selected.

VII. Deleting a Record

The delete_Click method is responsible for deleting an exit record. If no record is selected (Key == 0), it shows a message asking the user to select a record. When a valid record is selected, it asks for confirmation through a message box. If the user confirms, it deletes the selected record from ExitTbl and refreshes the data display and vehicle numbers.

```
private void delete_Click(object sender, EventArgs e)
{
    if (Key == 0)
    {
        MessageBox.Show("Select a record to delete.");
        return;
    }
    DialogResult result = MessageBox.Show("Are you sure you want to delete this record?", "Confirm Deletion", MessageBoxButtons.YesNo);
    if (result == DialogResult.Yes)
    {
        try
        {
            string Query = "DELETE FROM ExitTbl WHERE ExitNo = {0}";
            Query = string.Format(Query, Key);
            Con.SetData(Query);
            MessageBox.Show("Record Deleted!!!");
            ShowSection();
            UpdateVehicleNumbers();
        }
        catch (Exception Ex)
        {
            MessageBox.Show("Error: " + Ex.Message);
        }
    }
}
```

6.6.5. Summary

This code efficiently manages vehicle exits by interacting with the EntryTbl and ExitTbl tables, allowing users to add, view, and delete exit records. The ComboBox is dynamically populated with available vehicle numbers, ensuring only vehicles that haven't exited are listed. The system also provides feedback through message boxes and refreshes data after each operation.

6.7. Appendix G: Profiles Interface

6.7.1. Overview

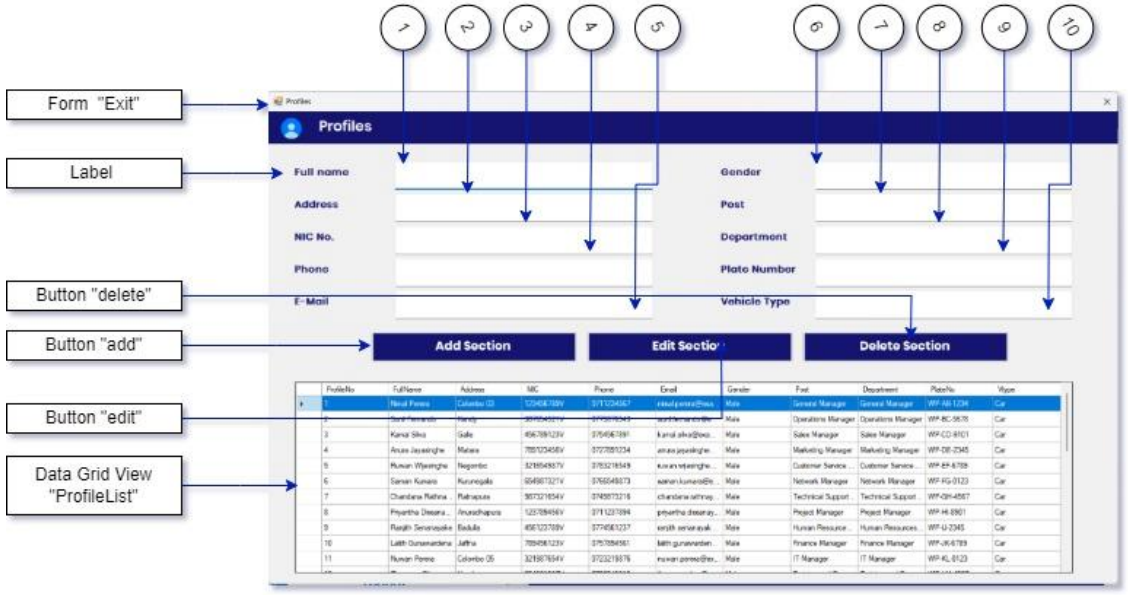


Figure 42-Profiles Interface overview

1-TextBox “name”

2-TextBox “address”

3-TextBox “nic”

4-TextBox “phone”

5-TextBox “email”

6-TextBox “gender”

7-TextBox “post”

8-TextBox “dep”

9-TextBox “plate”

10-TextBox “vtype”

This form design is part of a profile management interface, allowing users to enter or update various details associated with a person's profile, particularly for managing vehicle parking systems.

The TextBox for "Name" is used to capture the full name of the individual. This field is essential for identifying the user and linking their profile to other details in the system. Below the name input, the TextBox for "Address" allows the user to provide their residential or work address, which may be important for administrative purposes or contact.

Next, the TextBox for "NIC" collects the individual's National Identity Card number, ensuring that the system can verify their identity with an official government-issued ID. This field is vital for accurate record-keeping and security. The TextBox for "Phone" is provided for entering a contact number, facilitating communication if needed. Similarly, the TextBox for "Email" allows the user to input their email address, enabling electronic communication or notifications from the system.

On the right side, the form includes more specific profile details. The TextBox for "Gender" collects the user's gender information, which may be used for demographic purposes. The TextBox for "Post" refers to the user's role or position, possibly indicating their job title within an organization. The TextBox for "Department" (labeled "Dep") gathers information on which department the individual is associated with, aiding in internal categorization within an organization.

For the vehicle-related details, the TextBox for "Plate" collects the individual's vehicle registration or license plate number, linking their vehicle to their profile in the parking system. Finally, the TextBox for "VType" (vehicle type) records the type of vehicle the user drives, such as a car, motorbike, or truck, which may be necessary for organizing parking spots based on vehicle size or type.

This comprehensive form ensures that all necessary details for managing both individual profiles and vehicles are included, facilitating efficient parking space management.

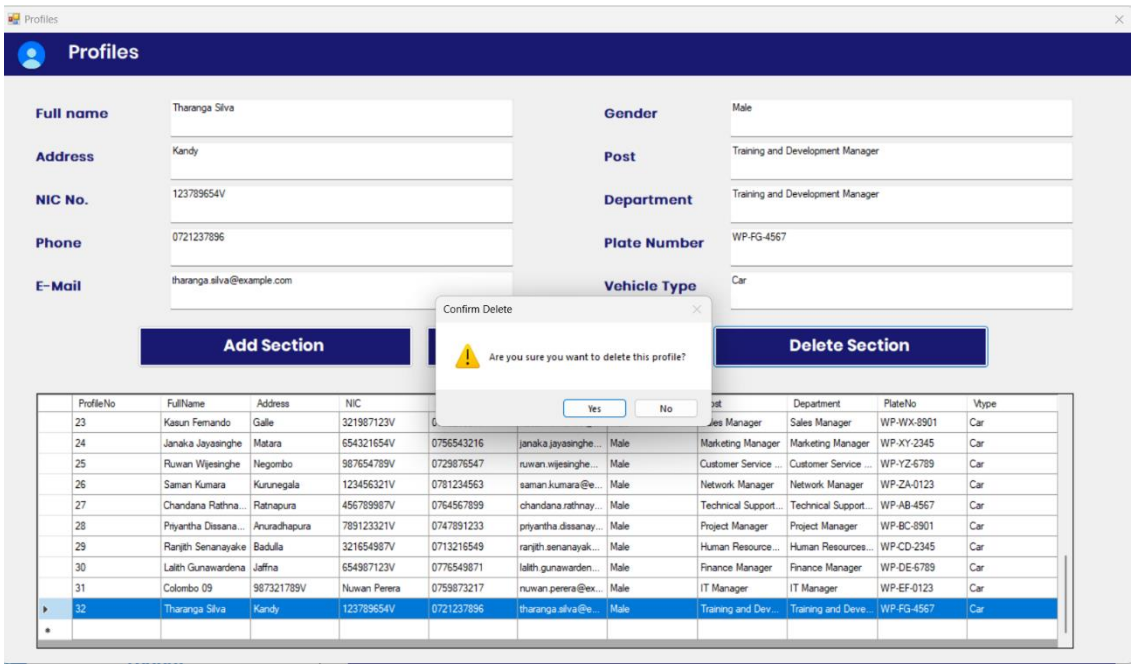


Figure 43-Profile page show message box when click Delete Button

6.7.2. Colors

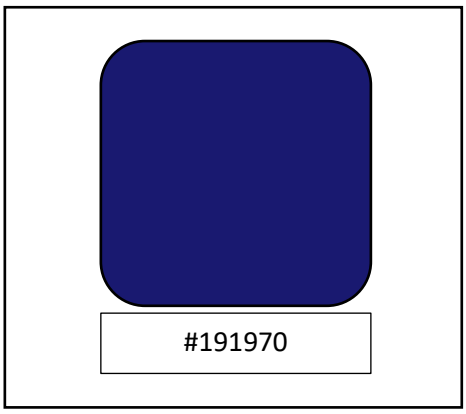


Figure 44-Color used in profiles Interface

6.7.3. Full Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Vehicle_Parking_Management_System_Project
{
   public partial class Profiles : Form
   {
      public Profiles()
      {
         InitializeComponent();
         Con = new Functions();
         ShowSection();
```

```
}
Functions Con;

private void ShowSection()
{
    string Query = "SELECT * FROM ProfileTbl";
    ProfileList.DataSource = Con.GetData(Query);
    ProfileList.Refresh(); // Refresh the DataGridView if needed
}
private void add_Click(object sender, EventArgs e)
{
    if (name.Text == "" ||
        address.Text == "" ||
        nic.Text == "" ||
        phone.Text == "" ||
        email.Text == "" ||
        gender.Text == "" ||
        post.Text == "" ||
        dep.Text == "" ||
        plate.Text == "" ||
        vtype.Text == "")
    {
        MessageBox.Show("Missing data or no selection!!!");
    }
    else
    {
        try
        {
            string Name = name.Text;
```

```
                string Address = address.Text;
                string NIC = nic.Text;
                string Phone = phone.Text;
                string Email = email.Text;
                string Gender = gender.Text;
                string Post = post.Text;
                string Department = dep.Text;
                string PlateNo = plate.Text;
                string Vtype = vtype.Text;
                string Query = "INSERT INTO ProfileTbl
(FullName,Address,NIC,Phone,Email,Gender,Post,Department,PlateNo,Vtype) VALUES
('{0}', '{1}', '{2}', '{3}', '{4}', '{5}', '{6}', '{7}', '{8}', '{9}')";
                Query = string.Format(Query, Name, Address, NIC, Phone, Email, Gender,
Post, Department, PlateNo, Vtype);
                // Execute the insert command
                Con.SetData(Query);
                MessageBox.Show("Profile Added!!!");
                ShowSection();
            }
            catch (Exception Ex)
            {
                MessageBox.Show(Ex.Message);
            }
        }
    }
    int Key = 0;
    private void ProfileList_CellContentClick(object sender,
DataGridViewCellEventArgs e)
    {
        if (e.RowIndex >= 0)
        {
```

```
            ProfileList.CurrentRow.Selected = true;
            name.Text = ProfileList.CurrentRow.Cells[1].Value.ToString();
            address.Text = ProfileList.CurrentRow.Cells[2].Value.ToString();
            nic.Text = ProfileList.CurrentRow.Cells[3].Value.ToString();
            phone.Text = ProfileList.CurrentRow.Cells[4].Value.ToString();
            email.Text = ProfileList.CurrentRow.Cells[5].Value.ToString();
            gender.Text = ProfileList.CurrentRow.Cells[6].Value.ToString();
            post.Text = ProfileList.CurrentRow.Cells[7].Value.ToString();
            dep.Text = ProfileList.CurrentRow.Cells[8].Value.ToString();
            plate.Text = ProfileList.CurrentRow.Cells[9].Value.ToString();
            vtype.Text = ProfileList.CurrentRow.Cells[10].Value.ToString();

            Key = Convert.ToInt32(ProfileList.CurrentRow.Cells[0].Value);
        }
        else
        {
            Key = 0; // No valid selection
        }
    }
    private void edit_Click(object sender, EventArgs e)
    {
        if (name.Text == "" ||
            address.Text == "" ||
            nic.Text == "" ||
            phone.Text == "" ||
            email.Text == "" ||
            gender.Text == "" ||
            post.Text == "" ||
            dep.Text == "" ||
```

```
            plate.Text == "" ||
            vtype.Text == "")
        {
            MessageBox.Show("Missing data or no selection!!!");
        }
        else
        {
            try
            {
                string Name = name.Text;
                string Address = address.Text;
                string NIC = nic.Text;
                string Phone = phone.Text;
                string Email = email.Text;
                string Vtype = gender.Text;
                string PlateNo = post.Text;
                string Gender = vtype.Text;
                string Post = plate.Text;
                string Department = dep.Text;
                string Query = "UPDATE ProfileTbl SET  FullName = '{0}', Address = '{1}',
NIC='{2}',Phone = '{3}', Email = '{4}', Gender = '{5}', Post = '{6}', Department = '{7}',
PlateNo = '{8}', Vtype = '{9}' WHERE ProfileNo = {10}";
                Query = string.Format(Query, NIC, Name, Address, Phone, Email, Gender,
Post, Department, PlateNo, Vtype, Key);
                // Execute the update command
                Con.SetData(Query);
                MessageBox.Show("Profile Updated!!!");
                ShowSection();
            }
            catch (Exception Ex)
```

```
            {
                MessageBox.Show(Ex.Message);
            }
        }
    }
    private void delete_Click(object sender, EventArgs e)
    {
        if (Key == 0)
        {
            MessageBox.Show("No profile selected!");
        }
        else
        {
            DialogResult result = MessageBox.Show("Are you sure you want to delete this profile?", "Confirm Delete", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
            if (result == DialogResult.Yes)
            {
                try
                {
                    string Query = "DELETE FROM ProfileTbl WHERE ProfileNo = {0}";
                    Query = string.Format(Query, Key);
                    // Execute the delete command
                    Con.SetData(Query);
                    MessageBox.Show("Profile Deleted!!!");
                    ShowSection();
                }
                catch (Exception Ex)
                {
                    MessageBox.Show(Ex.Message);
```

```
            }
        }
        else
        {
            MessageBox.Show("Delete operation canceled.");
        }
    }
  }
 }
}
```

6.7.4. Code Introduction and Process

I. Class Definition and Constructor

```
public partial class Profiles : Form
{
    public Profiles()
    {
        InitializeComponent();
        Con = new Functions();
        ShowSection();
    }
    Functions Con;
}
```

This section defines the constructor for the Profiles form. When the form is initialized, it also creates an instance of the Functions class (which handles database interactions) and calls the ShowSection() method to load and display the existing profiles in the form. InitializeComponent() is a method generated by the designer to initialize the form's controls.

II. Displaying Data

```
private void ShowSection()
{
    string Query = "SELECT * FROM ProfileTbl";
    ProfileList.DataSource = Con.GetData(Query);
    ProfileList.Refresh(); // Refresh the DataGridView if needed
}
```

ShowSection() is responsible for displaying data from the ProfileTbl table. It executes an SQL SELECT query to retrieve all the records from the database and binds them to a DataGridView called ProfileList. The Refresh() method ensures that the data displayed is up-to-date.

III. Add Button Event (Adding a New Profile)

```
private void add_Click(object sender, EventArgs e)
{
    if (name.Text == "" || /* other validations */ vtype.Text == "")
    {
        MessageBox.Show("Missing data or no selection!!!");
    }
    else
    {
        try
        {
            string Query = "INSERT INTO ProfileTbl (FullName, Address, ...) VALUES ('{0}', '{1}', ...)";
            Query = string.Format(Query, Name, Address, ...);
            Con.SetData(Query); // Insert into database
            MessageBox.Show("Profile Added!!!");
            ShowSection(); // Refresh the data
        }
```

```
        }
        catch (Exception Ex)
        {
            MessageBox.Show(Ex.Message);
        }
    }
}
```

This block handles the event when the user clicks the add button. It first checks if all the necessary fields (e.g., name, address, etc.) are filled out. If not, it shows an error message. If all fields are filled, it constructs an SQL INSERT query to add the new profile to the ProfileTbl table. It uses Con.SetData(Query) to execute the query, then refreshes the ProfileList to show the updated data. Any exceptions are caught and displayed as a message.

IV. Selecting a Profile from the List

```
private void ProfileList_CellContentClick(object sender, DataGridViewCellEventArgs e)
{
    if (e.RowIndex >= 0)
    {
        // Populate the text fields with the selected profile's details
        ProfileList.CurrentRow.Selected = true;
        name.Text = ProfileList.CurrentRow.Cells[1].Value.ToString();
        // Continue for other fields like address, nic, etc.
        Key = Convert.ToInt32(ProfileList.CurrentRow.Cells[0].Value);
    }
    else
    {
        Key = 0; // No valid selection
    }
}
```

When a user clicks on a row in the ProfileList (DataGridView), this method populates the form fields with the selected profile's details. It assigns the values from the row's cells to corresponding text fields (e.g., name, address, nic). The Key variable stores the ProfileNo (primary key) of the selected profile, which will be used later for updating or deleting the profile.

V. Edit Button Event (Updating a Profile)

```
private void edit_Click(object sender, EventArgs e)
{
   if (name.Text == "" || /* other validations */ vtype.Text == "")
   {
      MessageBox.Show("Missing data or no selection!!!");
   }
   else
   {
      try
      {
         // Update query with user inputs
         string Query = "UPDATE ProfileTbl SET FullName = '{0}', Address = '{1}',
... WHERE ProfileNo = {10}";
         Query = string.Format(Query, Name, Address, ..., Key);
         Con.SetData(Query); // Execute update
         MessageBox.Show("Profile Updated!!!");
         ShowSection(); // Refresh the data
      }
      catch (Exception Ex)
      {
         MessageBox.Show(Ex.Message);
      }
   }
}
```

This section handles the edit button click event. Similar to the add operation, it first validates the input fields. If valid, it constructs an SQL UPDATE query to modify the selected profile's information based on the Key value (the ProfileNo). After executing the query, it shows a success message and refreshes the ProfileList to reflect the changes.

VI. Delete Button Event (Deleting a Profile)

```
private void delete_Click(object sender, EventArgs e)
{
   if (Key == 0)
   {
      MessageBox.Show("No profile selected!");
   }
   else
   {
      DialogResult result = MessageBox.Show("Are you sure you want to delete this profile?", "Confirm Delete", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
      if (result == DialogResult.Yes)
      {
         try
         {
            // Delete query
            string Query = "DELETE FROM ProfileTbl WHERE ProfileNo = {0}";
            Query = string.Format(Query, Key);
            Con.SetData(Query); // Execute delete
            MessageBox.Show("Profile Deleted!!!");
            ShowSection(); // Refresh the data
         }
         catch (Exception Ex)
         {
            MessageBox.Show(Ex.Message);
         }
      }
```

```
        else
        {
            MessageBox.Show("Delete operation canceled.");
        }
    }
}
```

This section defines the delete button functionality. It first checks if a profile is selected (i.e., Key != 0). If not, it shows a warning message. If a profile is selected, the system asks for confirmation from the user before proceeding. If confirmed, an SQL DELETE query is executed to remove the profile from the database. Finally, the ProfileList is refreshed to remove the deleted profile from the list.

6.7.5. Summary

This code effectively manages user profiles, allowing CRUD (Create, Read, Update, Delete) operations through a Windows Forms interface. It leverages SQL queries to interact with the database and uses the DataGridView control to display profile data dynamically. The Functions class (not shown) is assumed to handle database connections and query execution, making this form a powerful tool for managing profile data.

6.8. Appendix H: Functions

6.8.1. Overview

This C# code is part of a Vehicle Parking Management System that interacts with a SQL database to perform various operations such as retrieving data, executing SQL commands (insert, update, delete), and clearing tables. The code is housed within the Functions class, which is responsible for handling database-related functionalities.

The constructor of the Functions class initializes a connection to the SQL database by defining a connection string (ConStr). This connection string points to a .mdf file, which is the SQL Server database used for the parking management system. Additionally, the constructor sets up an SqlCommand object, which is used to execute SQL queries.

The GetData method retrieves data from the database based on a SQL query. It accepts the query as a string and an optional array of SqlParameter objects to prevent SQL injection. Inside this method, a SqlConnection is opened, and a SqlDataAdapter is used to fill a DataTable with the result of the query. This method handles exceptions by logging errors and re-throwing them for further handling.

The SetData method is responsible for executing SQL commands that modify the data in the database (insert, update, or delete). Similar to GetData, this method accepts a query and optional parameters. The method opens a connection to the database and uses ExecuteNonQuery to execute the command. The number of affected rows is returned, indicating how many records were modified. Errors are handled by logging and re-throwing exceptions.

The GetCombinedData method retrieves combined data from three tables: EntryTbl, ExitTbl, and CarTbl, based on a common key—VNo (Vehicle Number) and PlateNo (Plate Number). The SQL query used here joins these tables to produce a consolidated view of vehicle information, such as vehicle type, driver details, entry time, and exit time. The method uses GetData to execute the query and return the result.

The ClearTables method is designed to clear all data from two specific tables: EntryTbl and ExitTbl. It constructs two separate SQL delete queries to remove all records from these tables. The SetData method is called to execute these queries, ensuring that the tables are emptied. Any errors during this process are logged and re-thrown.

Throughout the code, error handling is implemented using try-catch blocks. In case of exceptions, errors are logged to the console, providing details of what went wrong. The exceptions are then re-thrown, allowing the system to manage them as needed.

In summary, this class provides functionality for interacting with the SQL database, handling basic CRUD operations, retrieving data from multiple tables, and clearing records. It effectively encapsulates all database-related logic needed for the vehicle parking management system.

6.8.2. Full Code

```
using System;

using System.Data;

using System.Data.SqlClient;

namespace Vehicle_Parking_Management_System_Project

{

  class Functions

  {

    private SqlConnection Con;

    private SqlCommand Cmd;

    private string ConStr;


    public Functions()
```

```
        {
            // Corrected connection string
            ConStr = @"Data
Source=(LocalDB)\v11.0;AttachDbFilename=C:\Users\ISHAN\Desktop\BIT
Project\Vehicle Parking Management System Project\Vehicle Parking Management
System Project\parkingMSDB.mdf;Integrated Security=True";

            Con = new SqlConnection(ConStr);
            Cmd = new SqlCommand();
            Cmd.Connection = Con;
        }
        // Get data from the database
        public DataTable GetData(string Query, SqlParameter[] parameters = null)
        {
            DataTable dt = new DataTable();
            using (SqlConnection con = new SqlConnection(ConStr))
            {
                using (SqlCommand cmd = new SqlCommand(Query, con))
                {
                    if (parameters != null)
                    {
                        cmd.Parameters.AddRange(parameters);
                    }
                    try
                    {
                        con.Open();
                        SqlDataAdapter sda = new SqlDataAdapter(cmd);
                        sda.Fill(dt);
                    }
                    catch (Exception ex)
```

```
            {
                Console.WriteLine("Error retrieving data: " + ex.Message); // Log errors
                throw; // Re-throw the exception for further handling if needed
            }
        }
    }
    return dt;
}
// Insert, update, delete operations
public int SetData(string Query, SqlParameter[] parameters = null)
{
    int Cnt = 0;
    using (SqlConnection con = new SqlConnection(ConStr))
    {
        using (SqlCommand cmd = new SqlCommand(Query, con))
        {
            if (parameters != null)
            {
                cmd.Parameters.AddRange(parameters);
            }
            try
            {
                con.Open();
                Cnt = cmd.ExecuteNonQuery();
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error executing command: " + ex.Message); // Log
errors
```

```
                throw; // Re-throw the exception for further handling if needed
            }
        }
    }
    return Cnt;
}
public DataTable GetCombinedData()
{
    DataTable combinedData = new DataTable();
    try
    {
        // SQL query to fetch data from EntryTbl, ExitTbl, and CarTbl based on VNo
(Vehicle Number)
        string query = @"
    SELECT
        e.VNo AS 'Vehicle Number',
        e.PlateNo AS 'Plate Number',
        c.VType AS 'Vehicle Type',
        c.DriverName AS 'Driver Name',
        c.Phone AS 'Phone',
        e.EntryTime AS 'Entry Time',
        x.ExitTime AS 'Exit Time'
    FROM
        EntryTbl e
    LEFT JOIN
        ExitTbl x ON e.VNo = x.VNo
    LEFT JOIN
        CarTbl c ON e.PlateNo = c.PlateNo
    ORDER BY
```

```
            e.EntryTime";
            // Fetch combined data based on the query
            combinedData = GetData(query);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error retrieving combined data: " + ex.Message);
            throw; // Re-throw the exception for further handling if needed
        }
        return combinedData;
    }
    public void ClearTables()
    {
        try
        {
            // Delete all records from EntryTbl
            string deleteEntryQuery = "DELETE FROM EntryTbl";
            SetData(deleteEntryQuery);
            // Delete all records from ExitTbl
            string deleteExitQuery = "DELETE FROM ExitTbl";
            SetData(deleteExitQuery);
            Console.WriteLine("All data cleared from EntryTbl and ExitTbl.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error clearing tables: " + ex.Message);
            throw; // Re-throw the exception for further handling if needed
        }
    }
```

6.8.3. Code Introduction and Process

I. Class Definition and Constructor

```
namespace Vehicle_Parking_Management_System_Project
{
   class Functions
   {
        private SqlConnection Con;
        private SqlCommand Cmd;
        private string ConStr;
        public Functions()
        {
           ConStr = @"Data
        Source=(LocalDB)\v11.0;AttachDbFilename=C:\Users\ISHAN\Desktop\BIT
        Project\Vehicle Parking Management System Project\Vehicle Parking
        Management System Project\parkingMSDB.mdf;Integrated Security=True";
           Con = new SqlConnection(ConStr);
           Cmd = new SqlCommand();
           Cmd.Connection = Con;
        }
```

In this section, the Functions class is defined to encapsulate database operations. The class has three private members: Con (a SqlConnection object for establishing connections with the database), Cmd (a SqlCommand object for executing SQL queries), and ConStr (a connection string to locate and authenticate the database). In the constructor, the connection string is assigned to ConStr, and the Cmd.Connection is set to Con for executing commands within the same connection.

II. GetData Method

This method retrieves data from the database. It takes an SQL query as a string (Query) and an optional array of SQL parameters (parameters). Inside, a DataTable is used to store the result set. The method creates a new SqlConnection and SqlCommand object, checks if parameters are provided, and adds them to the command if available. The SqlDataAdapter object fills the DataTable with the results of the query. Errors are caught in a try-catch block and logged to the console, and any exceptions are re-thrown for further handling.

```
public DataTable GetData(string Query, SqlParameter[] parameters = null)
{
   DataTable dt = new DataTable();
   using (SqlConnection con = new SqlConnection(ConStr))
   {
      using (SqlCommand cmd = new SqlCommand(Query, con))
      {
         if (parameters != null)
         {
            cmd.Parameters.AddRange(parameters);
         }

         try
         {
            con.Open();
            SqlDataAdapter sda = new SqlDataAdapter(cmd);
            sda.Fill(dt);
         }
         catch (Exception ex)
         {
            Console.WriteLine("Error retrieving data: " + ex.Message); // Log
errors
            throw; // Re-throw the exception for further handling if needed
         }
      }
   }
   return dt;
}
```

III. SetData Method

```
public int SetData(string Query, SqlParameter[] parameters = null)
{
   int Cnt = 0;
   using (SqlConnection con = new SqlConnection(ConStr))
   {
      using (SqlCommand cmd = new SqlCommand(Query, con))
      {
         if (parameters != null)
         {
            cmd.Parameters.AddRange(parameters);
         }

         try
         {
            con.Open();
            Cnt = cmd.ExecuteNonQuery();
         }
         catch (Exception ex)
         {
            Console.WriteLine("Error executing command: " + ex.Message); // Log errors
            throw; // Re-throw the exception for further handling if needed
         }
      }
   }
   return Cnt;
}
```

This method executes non-query operations such as INSERT, UPDATE, and DELETE. It accepts an SQL query and optional SQL parameters. The method opens a connection, executes the command using ExecuteNonQuery(), and returns the number of rows affected by the query. If any errors occur, they are caught, logged, and re-thrown for further processing.

IV. GetCombinedData Method

```
public DataTable GetCombinedData()
{
   DataTable combinedData = new DataTable();


   try
   {
      // SQL query to fetch data from EntryTbl, ExitTbl, and CarTbl based on
VNo (Vehicle Number)
      string query = @"
      SELECT
         e.VNo AS 'Vehicle Number',
         e.PlateNo AS 'Plate Number',
         c.VType AS 'Vehicle Type',
         c.DriverName AS 'Driver Name',
         c.Phone AS 'Phone',
         e.EntryTime AS 'Entry Time',
         x.ExitTime AS 'Exit Time'
      FROM
         EntryTbl e
      LEFT JOIN
         ExitTbl x ON e.VNo = x.VNo
      LEFT JOIN
         CarTbl c ON e.PlateNo = c.PlateNo
      ORDER BY
         e.EntryTime";
```

```
        combinedData = GetData(query);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error retrieving combined data: " + ex.Message);
                throw; // Re-throw the exception for further handling if needed
            }

            return combinedData;
        }
```

This method retrieves combined data from three tables: EntryTbl, ExitTbl, and CarTbl. It joins these tables using the vehicle number (VNo) and plate number (PlateNo) to fetch details like vehicle type, driver name, phone, entry time, and exit time. A LEFT JOIN is used to ensure that even if there is no matching record in ExitTbl, data from EntryTbl will still appear. The query is passed to the GetData() method to execute, and the results are returned in a DataTable.

V. ClearTables Method

```
        public void ClearTables()
        {
            try
            {
                // Delete all records from EntryTbl
                string deleteEntryQuery = "DELETE FROM EntryTbl";
                SetData(deleteEntryQuery);
                // Delete all records from ExitTbl
                string deleteExitQuery = "DELETE FROM ExitTbl";
                SetData(deleteExitQuery);
                Console.WriteLine("All data cleared from EntryTbl and ExitTbl.");
            }
            catch (Exception ex)
```

```
        {
            Console.WriteLine("Error clearing tables: " + ex.Message);
            throw; // Re-throw the exception for further handling if needed
        }
    }
```

This method clears all data from two tables: EntryTbl and ExitTbl. It uses DELETE queries and calls the SetData() method to execute them. After clearing both tables, a message is logged to the console. If any errors occur during the deletion process, they are caught, logged, and re-thrown.

6.8.4. Summary

This class serves as a utility for interacting with a vehicle parking management system's database. It allows for retrieving, inserting, updating, deleting, and clearing data in the system, ensuring smooth database operations. The code is structured to handle potential errors and ensure that resources (such as database connections) are managed correctly using using statements.

6.9. Appendix I: SQL Codes

6.9.1. Overview

The given SQL code defines the structure of several database tables, each serving a specific purpose in a system likely related to vehicle parking and profile management.

The SectionTbl is used to store information about different sections, possibly representing areas in a parking system. It includes a unique section number (SNo), the section's name (SName), its capacity (Capacity), and a description of the section (SDescription). The primary key of this table is the SNo, which ensures each section is uniquely identifiable.

The ProfileTbl captures personal information for individuals, which could include drivers or vehicle owners. This table stores data such as the individual's full name (FullName), address (Address), national identity card number (NIC), and contact details like phone and email. Additional fields include the gender, post (job title or role), department, vehicle plate number, and vehicle type. The ProfileNo serves as the primary key, uniquely identifying each profile entry.

The ExitTbl and EntryTbl represent the exit and entry points in the system, respectively, likely tracking vehicles' movements in and out of a parking area. Both tables store details such as the section name (SName), vehicle number (VNo), plate number (PlateNo), and the

associated time of entry or exit (EntryTime or ExitTime). The primary keys, ExitNo and EntNo, ensure unique entries for each record.

The CarTbl stores details related to the vehicles in the system. Fields such as the vehicle number (VNo), plate number (PlateNo), vehicle type (Vtype), and color (Colour) describe the vehicle. Additionally, it stores the driver's name (DriverName), NIC (DriverNIC), and phone number. A computed column, FormattedVNo, generates a formatted version of the vehicle number for easier reference, ensuring consistency in representing vehicle numbers. The VNo is the primary key, uniquely identifying each vehicle.

Together, these tables create a structure for managing sections, vehicle profiles, entries and exits, and vehicle-specific data in a system, most likely a parking management system.

6.9.2. Codes

```
CREATE TABLE [dbo].[SectionTbl] (
    [SNo]         INT          IDENTITY (1, 1) NOT NULL,
    [SName]       VARCHAR (10)  NULL,
    [Capacity]    INT          NULL,
    [SDescription] VARCHAR (100) NULL,
    PRIMARY KEY CLUSTERED ([SNo] ASC)
);
CREATE TABLE [dbo].[ProfileTbl] (
    [ProfileNo]  INT          IDENTITY (1, 1) NOT NULL,
    [FullName]   VARCHAR (100) NULL,
    [Address]    VARCHAR (255) NULL,
    [NIC]        VARCHAR (20)  NOT NULL,
    [Phone]      VARCHAR (20)  NULL,
    [Email]      VARCHAR (100) NULL,
    [Gender]     VARCHAR (10)  NULL,
    [Post]       VARCHAR (50)  NULL,
    [Department] VARCHAR (100) NULL,
    [PlateNo]    VARCHAR (20)  NULL,
    [Vtype]      VARCHAR (50)  NULL,
    PRIMARY KEY CLUSTERED ([ProfileNo] ASC)
```

```
);

CREATE TABLE [dbo].[ExitTbl] (
    [ExitNo]   INT         IDENTITY (1, 1) NOT NULL,
    [SName]    VARCHAR (10) NULL,
    [VNo]      VARCHAR (10) NULL,
    [PlateNo]  VARCHAR (10) NULL,
    [ExitTime] DATETIME     NULL,
    PRIMARY KEY CLUSTERED ([ExitNo] ASC)
);

CREATE TABLE [dbo].[EntryTbl] (
    [EntNo]     INT         IDENTITY (1, 1) NOT NULL,
    [SName]     VARCHAR (10) NULL,
    [VNo]       VARCHAR (10) NULL,
    [PlateNo]   VARCHAR (10) NULL,
    [EntryTime] DATETIME     NULL,
    PRIMARY KEY CLUSTERED ([EntNo] ASC)
);

CREATE TABLE [dbo].[CarTbl] (
    [VNo]         INT         IDENTITY (1, 1) NOT NULL,
    [PlateNo]     VARCHAR (10) NULL,
    [Vtype]       VARCHAR (50) NULL,
    [Colour]      VARCHAR (50) NULL,
    [DriverName]  VARCHAR (50) NULL,
    [DriverNIC]   VARCHAR (13) NULL,
    [Phone]       VARCHAR (13) NULL,
```

```
    [FormattedVNo] AS   ('V'+right('000'+CONVERT([varchar](3),[VNo]),(3)))
PERSISTED,
    PRIMARY KEY CLUSTERED ([VNo] ASC)
);
```

6.9.3. Summary

The provided SQL code establishes a set of database tables for a vehicle parking management system. The SectionTbl stores details about parking sections, including section number, name, capacity, and description. The ProfileTbl captures personal information of individuals associated with vehicles, such as name, address, and contact details. The ExitTbl and EntryTbl track vehicle movements in and out of the parking area, recording section name, vehicle number, plate number, and entry or exit time. Lastly, the CarTbl holds vehicle-specific information, including vehicle number, plate number, type, color, and driver details, with a computed column for a formatted vehicle number. Collectively, these tables facilitate comprehensive management of sections, profiles, and vehicle movements within the system.

7. CHAPTER 07 – REFERENCES

[1] Demirović, D., & Durić, N. (2014). *Advanced software system for optimization of car parking services in urban area*. ResearchGate. https://www.researchgate.net/publication/261280697_Advanced_software_system_for_optimization_of_Car_parking_services_in_urban_area

[2] Sharma, M., & Chaudhary, D. (2022). *Design and development of smart parking system using IoT technology*. *International Journal of Science Technology and Management*, 11(6), 39-46. https://www.ijstm.com/images/short_pdf/1656135872_8123.pdf

[3] Parada, J. (2023). *A web-based parking lot management system designed and developed using multi-paradigm programming languages*. ResearchGate. https://www.researchgate.net/publication/371876871_PARADAJuan_A_Web-Based_Parking_Lot_Management_System_Designed_and_Developed_Using_Multi-Paradigm_Programming_Languages

[4] Kashipara. *Vehicle Management System Project in C#.Net* [Online]. Available at: https://www.kashipara.com/project/idea/c-net/vehicle-management-system_3620.html

[5] MyCodeSpace. (n.d.). *Vehicle Parking Management System using C# and SQL Server*. [YouTube channel]. Retrieved from https://www.youtube.com/@MyCodeSpace1

[6] TECHNO IS TUTI. (n.d.). *Automobile Vehicle Management System*. Retrieved from https://technoistuti.in/automobile-vehicle-management-system/

[7] Weerarathna, M. M. (2023, May 19). *Parking Management System: C# & SQL Server*. Medium. Retrieved from https://medium.com/@mmweerarathna123/parking-management-system-c-sql-server-6ebbb69be07b

[8] Draw.io. (n.d.). *Draw.io Diagrams Application*. Retrieved from https://app.diagrams.net/

[9] Ma Tutor. (n.d.). *YouTube channel*. YouTube. Retrieved [insert retrieval date here], from https://www.youtube.com/@matutor2190

[10] KIMTOOFLEX. (n.d.). *YouTube channel*. YouTube. Retrieved [insert retrieval date here], from https://www.youtube.com/@KIMTOOFLEX