Table of Contents

List of Tables

List of Figures

I. Introduction

Car park gantry provides assures full control of monitoring, access and revenue operations. It offers reliable range of electronics access control systems tailored to car park applications using antenna, cash cards and complimentary cards. The system enables vehicles to raise barriers and record times of entry and exit which can be linked to data loggers and enables settlement of parking fees without parking tickets and cash. The interface diagram is shown in Fig.1,

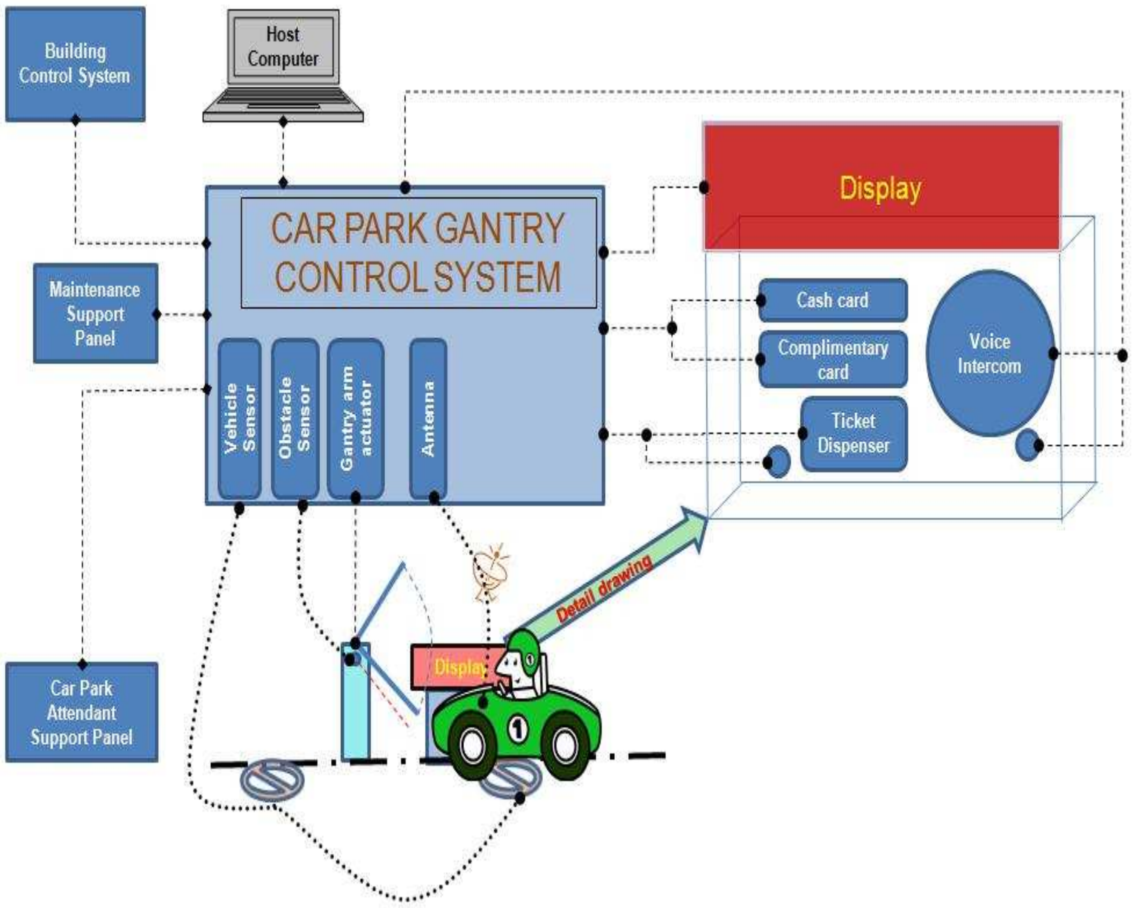


Figure 1: Car Park Gantry Interface Diagram

II. Software Requirements

2.1 Scope

Car Park Gantry system enables access, controls vehicular flow, support automatic or manual collection of payment, and provide necessary accounting and administrative reports. The system's flexibility makes it possible to enlarge the capacity, in terms of the number of vehicle gates, payment units, software performance and possible interfacing with external systems which improve the use of service.

The application makes it possible to connect several parking areas, manageable from a single remote station, thus centralising control and operational accounting.

The system is developed using QNX Neutrino platform using "C" language.

2.2 References

- Car park Gantry Interface Control Document, Version 2
- Electronic Parking System for Singapore, Mitsubishi Heavy Industries, Ltd. Technical Review Vol.40 No.3 (Jun. 2003)

2.3 Functional Requirements

2.3.1 Car park Entry Scenario**:**

- Vehicle presence to be detected at the entrance, for car to access the gantry.
- Users can access the gantry using IU card, Cash Card or Complimentary Card.

- User entry to be validated by matching either the IU label or Complimentary Card label with the database and to verify the minimum available balance.
- After validation the gantry gate to be opened for the user to access the car park.
- Once car cleared the gantry gate to be closed. Verifying no obstacle is detected while closing the gate.
- Once user entry is confirmed - label information, cash balance, time & date to be stored to a database.

2.3.2 Car park Exit Scenario:

- Vehicle presence to be detected before the exit gate, for car to exit the gantry.
- Users can pay the car park usage charges using IU card, Cash Card or Complimentary Card.
- A charging unit to be implemented for calculating the car park usage charges during exit.
- After detecting the charges the gantry gate to be opened for the user to exit the car park.
- Once car cleared the gantry gate to be closed. Verifying no obstacle is detected while closing the gate.
- Once user exit is confirmed - the user record to be deleted from the database.

2.3.3 Maintenance Scenario:

- Car park maintenance team decides to put the system in maintenance, and command the application to switch to manual mode.
- On maintenance mode the application updates the user display to indicate the system status.
- Maintenance team can either open or close the gantry arm manually.
- After Maintenance, a test can be simulated by requesting the application to run a self test.
- On successful completion of self test, the application can indicate ready signal to the maintenance team to put the system in auto mode.

2.3.4 Attendant Support:

- The scenario when the user is unable to access the car park, the user can communicate with the car park gantry attendant via. Voice intercom.
- Car park gantry attendant can override the application and can perform manual operation of the system – if required.
- The Car park gantry attendant can authorize an entry manually by overriding the automatic gantry operation.

2.3.5 Obstacle Sensor:

- Obstacle sensor input provides a safety feature to the system, by preventing possible hazard while lowering the gantry arm.
- Obstacle sensor detects any obstruction in the range while lowering the gantry arm and prevents the application from lowering the arm to avoid accidents.

2.3.6 Emergency Scenario:

- Confirmed Fire or Power Fail cases are considered as emergency scenarios.
- During such unexpected events, the application force opens the gantry arm irrespective of the present state.

2.3.7 Other Scenarios

- When the user presses the ticket dispenser button at gantry exit, the ticket is provided to the user with the following details – entry time, exit time and the fees; Applicable for the hourly users
- Whenever the lots become full, the display will indicate the same; the gantry arm will not operate to allow parking
- To ease the movement of traffic, the Gantry arm will not close completely; the opening of the Gantry arm will override the closing of the arm.
- When the user does not place his card in the IU unit, the display prompts the user to enter a card; If the user places a card in the IU unit then the antenna will be able to retrieve the IU and user details

2.4 Assumptions

- The application will have the following databases:
    - Database containing the list of the season users
    - Database containing the list of Valid IU users
    - Database to store the details – time of entry/exit, IU/ Cash card/ Complimentary card details, User category, Maximum parking time

- The various units have redundant hardware support which will take over in case of a failure
- Online diagnostics available for various units including redundant units which will report to the attendant in case of a failure so that manual maintenance can be carried on
- Additional messages can be added if required by the implementation in case of enhanced functionality, apart from the messages specified in the Gantry ICD
- Regular feedback taken from the customer

2.5 Quality Attributes

2.5.1 Flexibility

- Allows the configuration of the various parameters such as total number of lots, both season lots and hourly lots, parking charges, minimum balance, and maximum parking time.
- One application can be tailored for multiple Gantries (Entry / Exit).

2.5.2 Reliability

- Validity period verification of the season pass user before allowing to enter; hourly charges on expiry
- Users parking above maximum parking time will be indicated in the support system
- Warning issued on insufficient balance via display
- User alarmed if the card is left in the slot (after a certain configurable period by continuous beep); Gantry arm will open only after card has been retrieved.

- During maintenance, options available for built in tests improving the Reliability of the system

2.5.3 Availability

- Redundant load sensors inputs are assumed for car detection.
- Antenna module accounts for delayed user response and makes the system more available.
- Card reader module acts as a secondary access option providing system availability.
- Ease of parking - IU unit users do not have to exit out of the car
- Intercom support for insufficient balance; other emergency scenarios

2.5.4 Performance

- User minimum balance is verified at gantry entrance.
- Tickets are provided to Car park user with required information on request.
- In case of any failure the intercom is available for the user to request for assistance.
- Display behaves an effective medium to communicate in case of a fault/ error apart from other normal information.

2.5.5 Secured

- Validation of the user label (IU or Complimentary card label) provides secured access to the Gantry.
- Obstacle sensor input prevents accidents.
- On emergency notification from building control systems the application will keep the gantry open.

III. Design

The implementation of car park gantry was developed and simulated in QNX Neutrino using C language. The ability to ‘break the problem apart’ into several independent problems is a powerful concept. It's also at the heart of Neutrino. The design of the car park gantry was based on this powerful approach. The main purpose of the car park gantry software was to allow user to enter the car park for parking and later exit the car park. These responsibilities were break down into different process. Refer Fig.2, for the design framework. The entry of the gantry is a process and also the exit of gantry will be process. The simulator developed in order to test the developed application is also a process. Entry process is tested using the simulator corresponding to the entry and similarly the exit process is tested using simulator corresponding to the exit.

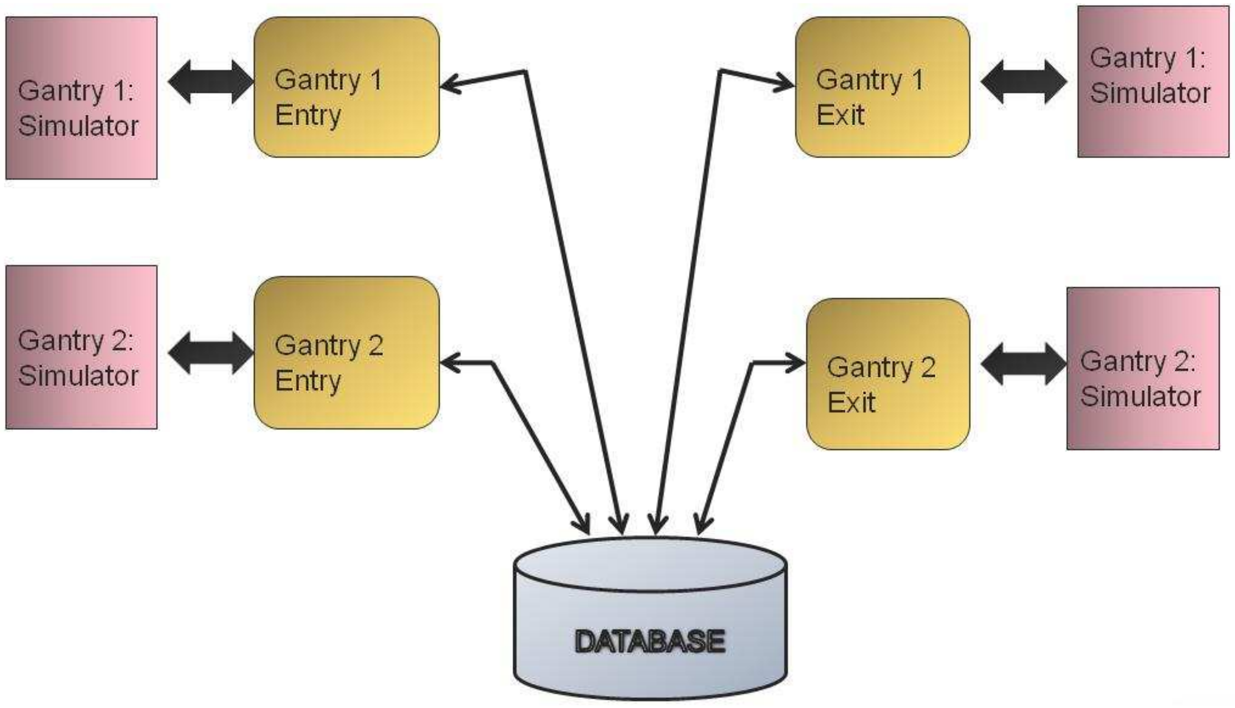


Figure 2: Design Framework Diagram

All the processes are connected to the database as shown in the Fig.2. The application is mainly based on a state machine design approach.

3.1 Architecture:

Our application is divided into two main parts.
1. Simulator (Server)
2. Application (Client)

Main driving part for this kind of architecture was to keep the application code separated from simulation. Simulation is supposed to only simulate the external world. Application should not care as to where the inputs are coming from as long as the required inputs are made available to the application. Hence it increases the ease of testing the application or the simulator. Testability of both simulator and the application increases due to this approach. Another driving force from point of implementation was that simulator and the application can now be two separate processes. This will help us implement this solution in QNX. The same application and simulator processes can be replicated to run and simulate multiple gantries. This architecture is scalable. This architecture is very close to how application would be run in an actual car park. All these applications running on different gantries connect to the same database.

Simulator does the part of getting all the inputs from the external world into the application and sending outputs to the external world. Simulator is designed to simulate all the inputs got from gantry user, car and from all the sensors. These inputs are then sent to the application where logical decisions are made based on the current state of the gantry and from the input received. There is no application logic written in our simulator. All decisions regarding the gantry will be made in the application. Simulator has two kinds of display. One display shows will show the current state of the application which more for testing and for the developer. The other display is for the gantry user. This display guides the gantry user and also shows information regarding cash, date and time. These displays improve the usability of this application.

This architecture required establishing a connection between simulator and application. This architecture also dictated that a message format be fixed so that both simulator and the application can talk to each other. Messages are identified by message IDs. Format of all messages is known to both simulator and to the application. Once the message ID is known, application knows where to get the required information.

All the inputs received from car, its user and from the sensors are listed. Simulator is designed to be an event based architecture. All buttons or other text boxes trigger events whenever their state or input changes. This is used to form an appropriate message based on the button’s state or input and is sent to the application.

1. Signal from load sensor before gantry arm. A button or a radio button is used for this purpose.
2. Message to simulate data sent from IU unit inside the car to the gantry.
3. Signal from obstacle sensor.
4. Signal from load sensor after the gantry arm.
5. Activation of intercom (button press from user).
6. Message from cash card or complementary card.
7. Season and hourly slots available.

Apart from the above listed items which are common to both entry and exit gantries, exit gantries contain.

1. Button to request ticket.
2. Ticket details as seen by the user.

Message format is same as that described in the requirements document. Number of bytes used to send messages are also same as that given in the requirements document.

3.2 Application:

Application is made to react to messages from the simulator. Application is independent of where the messages are coming from. It only reacts if it receives a correct message depending on its state. State machine based architecture is used. As operations involved in an operation of a gantry are sequential, every operation can represent a state. An appropriate message will get the control out of the current state and into the next state. Modifiability is high due to the use of state machine architecture. Application waits for a message. Once a message is received, it is processed in the current state.

For e.g. when the application is in a state where it waiting for a valid message from an IU label, message received is processed and is checked for its validity. If it is a valid message and sufficient cash is available, then application moves onto the next stage where it will wait for message from obstacle sensor.

Apart from the sequence flow defined in the flow chart a few other scenarios can be handled by this design approach as follows:

1. Application can give three chances to the user to either have a valid IU label or cash card or a complementary card. If the user is not able to produce a

valid card even after three attempts, only way for him to enter gantry is be calling the operator using intercom.

2. Cash available must be more than three dollars for the user to use the car park.
3. Application can activate voice intercom no matter what state the application is currently in.

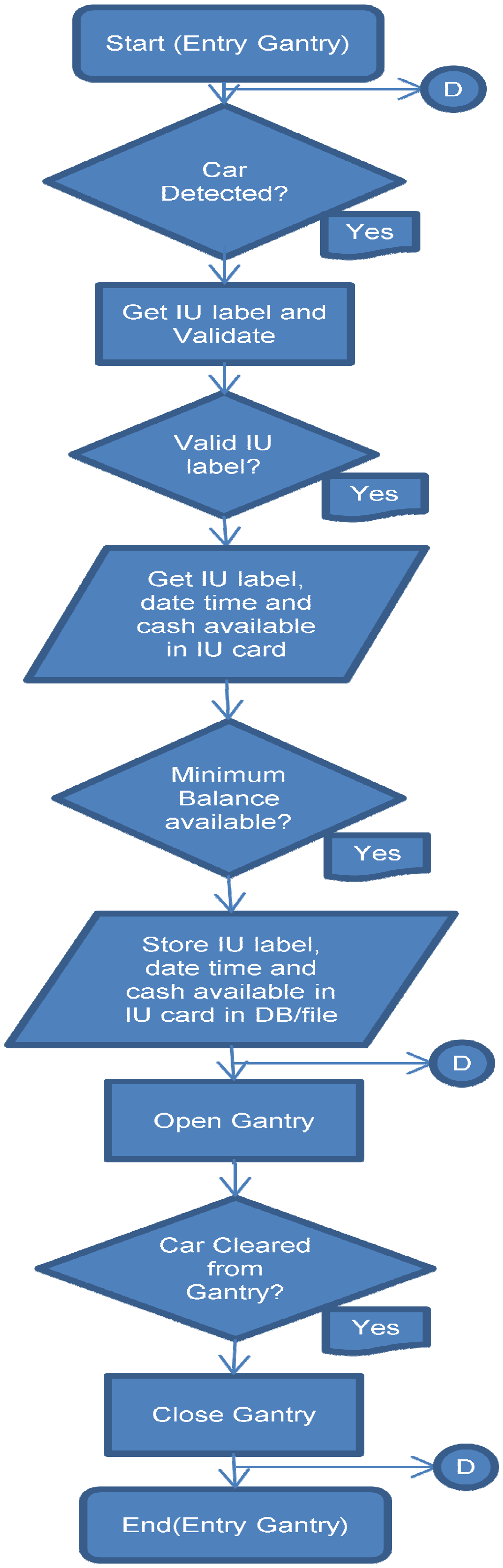

Start (Entry Gantry)
D
Car Detected?
Yes
Get IU label and Validate
Valid IU label?
Yes
Get IU label, date time and cash available in IU card
Minimum Balance available?
Yes
Store IU label, date time and cash available in IU card in DB/file
D
Open Gantry
Car Cleared from Gantry?
Yes
Close Gantry
D
End(Entry Gantry)

Figure 3: Entry flow chart

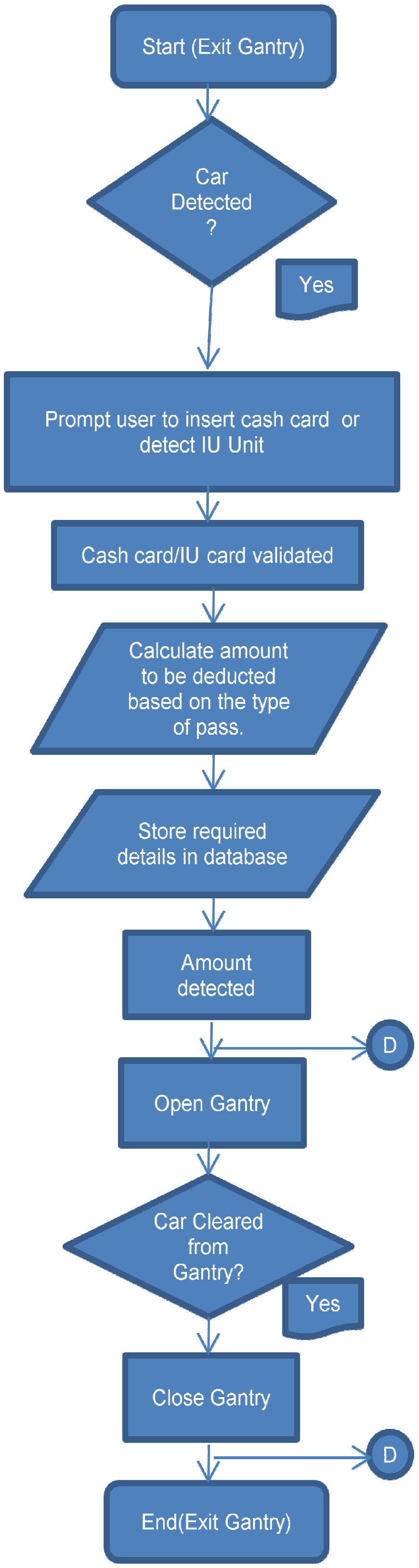

Start (Exit Gantry)
Car Detected ?
Yes
Prompt user to insert cash card or detect IU Unit
Cash card/IU card validated
Calculate amount to be deducted based on the type of pass.
Store required details in database
Amount detected
D
Open Gantry
Car Cleared from Gantry?
Yes
Close Gantry
D
End(Exit Gantry)

Figure 4: Exit flow chart

Various modules are defined to maintain modularity in the application:

1. Module to extract data from message received.
2. Module to do database operations. Write user details to database, validate IU label from database etc.
3. Module to form return messages based on state and message received.

3.3 Safety features:

Following safety features are taken care of by software:

1. Fire alarm. When there is a fire in the building, all gantries will be opened.
2. User can call the operator any time by pressing the intercom button.
3. Gantry arm will never be closed when the vehicle is below the gantry. Application will check for the vehicle to clear obstacle sensor and then load sensor after the gantry to close the Gantry arm.

Safety features can be easily added into the application as application uses state machine architecture. When any safety critical message is received by application, it abandons the current state and directly jumps to the state where it can handle this safety critical message.

3.4 Event Tree Analysis

The safety of the system is one of the important quality attribute that has to be considered while designing. In the car park gantry system, it is necessary to ensure the safety of car from being damaged due to gantry arm. Also it is necessary to ensure that the car should be safely exit without damage in case of fire and other critical scenarios.

To prevent the damage of car being hit by the gantry, an obstacle sensor is been provided which detects the presence of car while crossing the gantry. An active high obstacle sensor indicates the presence of car crossing the gantry arm. It can also happen that the obstacle sensor can fail and hence can cause damage to the vehicle. To avoid this scenario, an additional safety interlock is provided in our system by incorporating the load sensor2 input. The gantry arm will close only when the load sensor2 goes low which indicate that the car has crossed the gantry arm.

An event tree analysis diagram is shown in the Appendix below, in which the car crossing event is considered as a critical event which can affect the system and tracks them forward to determine the possible consequence in case of obstacle sensor failure or load sensor2 failure.
The entire ETA can be explained in different cases.

1) The obstacle sensor detects and the load sensor2 goes high: - Indicate the normal operation of the gantry, The gantry will close if and only if both the sensors goes low. No damage done to the vehicle.
2) The obstacle sensor detects and load sensor2 fails: The gantry still works without causing any damage to the vehicle because of the signal from obstacle sensor even though load sensor2 fails to detect the presence of car. The system is still operable.
3) Obstacle sensor fails and load sensor2 detects: This scenario is a hazard because of the obstacle sensor failure, but because of the feedback from load sensor2, the arm is prevented from being closed and avoids damage.
4) Both Obstacle sensor and load sensor2 fails: This event is an unwanted scenario which can cause damage to the vehicle. If this condition prevails, the mechanical assembly of gantry arm is made in such a way that the actuator de- energize to open which is the fail safe scenario. Also it informs the operator about the failure of obstacle sensor and load sensor2 and displays about immediate attention.

Appendix shows another event tree analysis where the outcome of car park gantry is been considered for the events such as power failure, building fire alarm and software failures are considered. When any of these events occurs, the application detects the event and causes the gantry arm to open so that the vehicles parked can be evacuated without much delay and thereby avoiding damage to man and machine. But if the application fails to detect the presence of any of these events will lead to an absolute chaos. To avoid this scenario, the gantry arm is been designed in such a way to open if there is a failure to the application i.e., fails to open.

Thus by make use of event tree analysis, safety of the system can be incorporated during design and implementation phase.

IV. Implementation

Once the gantry software is compiled and when it runs the simulator of the entry process spawns the entry application process, exit application process and its simulator process. So now four processes will run and the GUI of the entry process and the exit process are active. The user can interact with the entry and exit simulator simultaneously. The purpose of the simulator is to test the embedded software and the simulator has the responsibility for generating and receiving the messages to be supplied to the embedded application. The format of the message interface is followed as defined in the Gantry ICD document given.

4.1 Inter process communication

The inter process communication is my message passing. We have followed the traditional client/server model. The client sends message to the server, the server receives it from the client and the server replies to the client. In order to achieve this kind of approach we need Node Descriptor (ND), a Process ID (PID), and a Channel ID (CHID). If one process creates the other, then it's easy — the process creation call returns with the process ID of the newly created process. Both the creating process can pass its own PID and CHID on the command line to the newly created process or the newly created process can issue the getppid () function call to get the parent PID and assume a well-known CHID.

In our case we four process and they are like strangers so the problem comes in communicating between these processes. There are many ways in doing this,

1) Open a well-known filename and store the ND/PID/CHID there. This is the traditional approach taken by UNIX-style servers, where they open a file, write their process ID there as an ASCII string, and expect that clients will open the file and fetch the process ID.
2) Use global variables to advertise the ND/PID/CHID information. This is typically used in multi-threaded applications
3) Use the name-location functions.
4) Take over a portion of the pathname space and become a resource manager.

The complication by adapting these techniques for communicating is,

1) The first approach is very simple, but can suffer from “pathname pollution" the process that created the file can die without removing the file, there's no way of knowing whether or not the process is still alive until you try to send a message to it.

2) The second approach, where we use global variables to advertise the ND/PID/CHID values, is not a general solution, as it relies on the client's being able to access the global variables. And since this requires shared memory, it certainly won't work across a network.
3) The third approach, where we use the name_attach () and name_detach () function calls and it works well for simple client/server situations
4) The last approach, where the server becomes a resource manager, is definitely the cleanest and is the recommended general-purpose solution for this kind of problem.

Considering the above four methods method 3 and 4 seems to work out well for our design. But method 4 takes a considerable amount of time and due to the time constraints put on us we had to choose method 3. In this name declaration method we declare a name in the program and using this name the processes can communicate each other, as the names are visible to all the process since it is defined in the namespace of the kernel.

4.2 Database

In our design framework diagram we had showed that all the processes are connected to the database. QNX provides us a database which is called QDB (Qnx Data Base). It is small-footprint, embeddable SQL database server that supports most SQL-92 syntax. It is designed as an easy-to-configure Neutrino resource manager. Some of the key features of QDB are it provides concurrent access, synchronous safe writes, simple API for accessing the database, result storing for repeated use. Results can also be passed from one application to another. By default 64 threads can access the database. Considering all these advantages we decided to design the QDB for our project. A QDB configuration file is used to create the database. In our implementation we considered two databases.

4.2.1 Season User Database

The list of the season user is inserted in this database. The fields present in the database are shown in Table 1. The entry process connects to this database using the function calls provided by the QDB library. The Entry process check whether he is a season user or hourly user based on presence or not a decision is taken and an entry made in the operational database with S/H type.

Table 1: Season user

Car number	**IU card no**	**Zone ***	**Commencement month**	**End month**	**Amount paid**
ES6103	1D78		March	June	90

*incase customers wants a different zones of parking for season user and hourly users.

4.2.2 Operational Database

All entries in the car park gantry are registered in this database. Table 2 shows the field used in the database. The entry process and the exit process are connected to this database using the functional calls provided under the QDB library. The entry process first makes an entry into the database with the details such as the date, label type, cash available; entry time and type of the user such as season user or hourly user are entered during the entry. In case of the exit scenario the cash balance is calculated based on the type of user and the details after calculation are entered back into the database. Once backup of the database which is optional is done the label details can be deleted from the operational database.

Table 2: Operational database

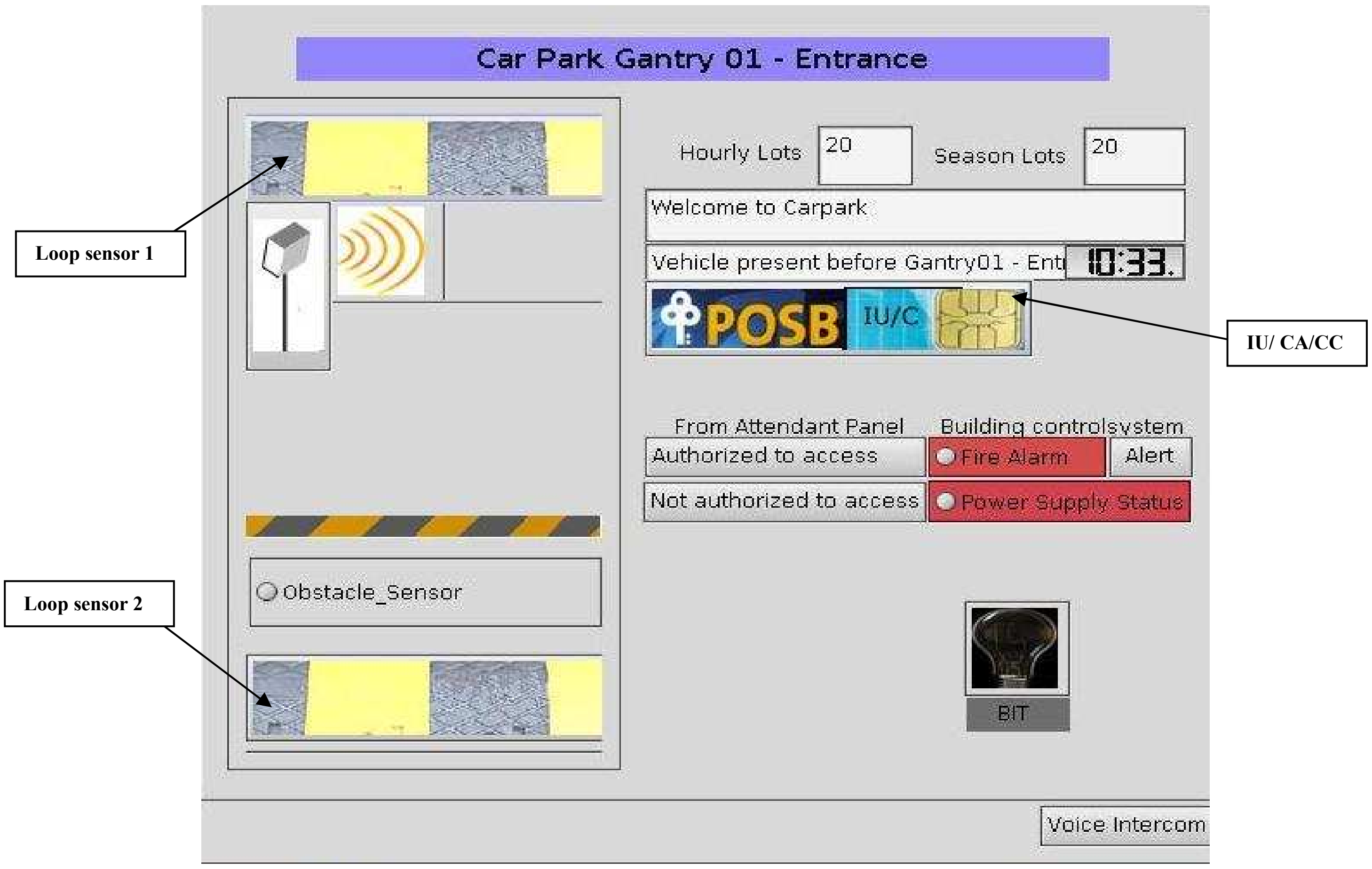
Date	**Type**	**User**	**Label**	**Entry time**	**Exit time**	**Amount**
YYYY:MM:DD	**IU/CA/CO**	**S/H/F**		**HH:MM:SS**	**HH:MM:SS**	**XXX.XX**
	IU	**S**	1D78			**-**
	IU	**H**	1C45			**2.40**

The simulator of the entry and the exit process are also connected to the operational database using the function call provided under the QDB library. The reasons for doing this are that the entry and exit process can update the number of lots for the hourly and the season users when a car enters/exits the car park gantry.

The user interface was developed using the Photon Application Builder. Numerous buttons are created and the callbacks are active so that whenever the user clicks the button the corresponding actions are taken.

4.3 Entry Scenario

The startup screen of the entry is shown in Fig.5 and first state is the detection of loop sensor so when the user clicks the loop sensor button 1 and the following message is displayed as shown in the Fig.5.



Welcome to Carpark

Vehicle present before Gantry01 - Ent

Figure 5: Start up screen

The next state would be clicking the IU/CA/CC button. On clicking you get a pop window as shown in Fig.6 asking the user to click either IU label (IU) or cash card (CA) or complimentary card (CC).

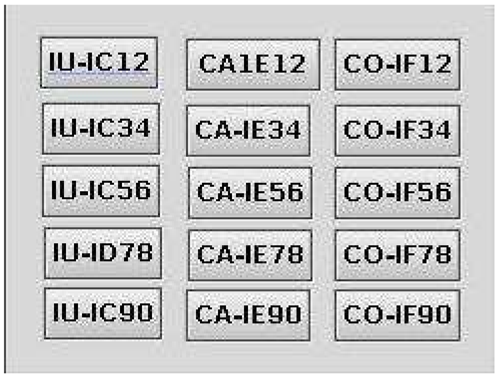
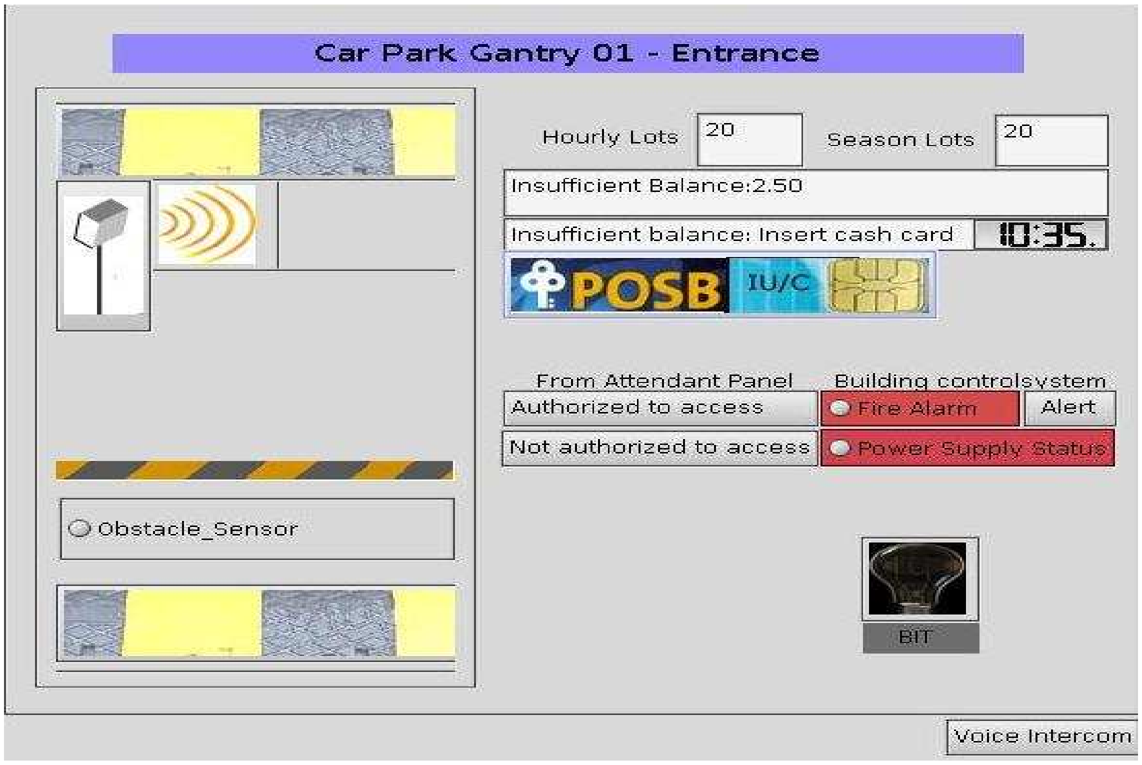


Figure 6: Labels list

The user can chose any of the labels. The next state would be the validation of the label. The application connects to the database of the season user and checks whether the label is a season user or not. If he is a season user then the user type would be S and on exit the cash is not deducted. If he is an hourly user then user type would be H. The available cash balance is checked if it is less than the minimum balance then the software asks the user to insert his cash card as shown in Fig.7.



Insufficient Balance:2.50

Insufficient balance: Insert cash card 10:35.

Figure 7: In sufficient balance scenario

If he is a valid user then the entry is written in the operational database and the gantry arm is open and the lots are reduced as shown in Fig.8.

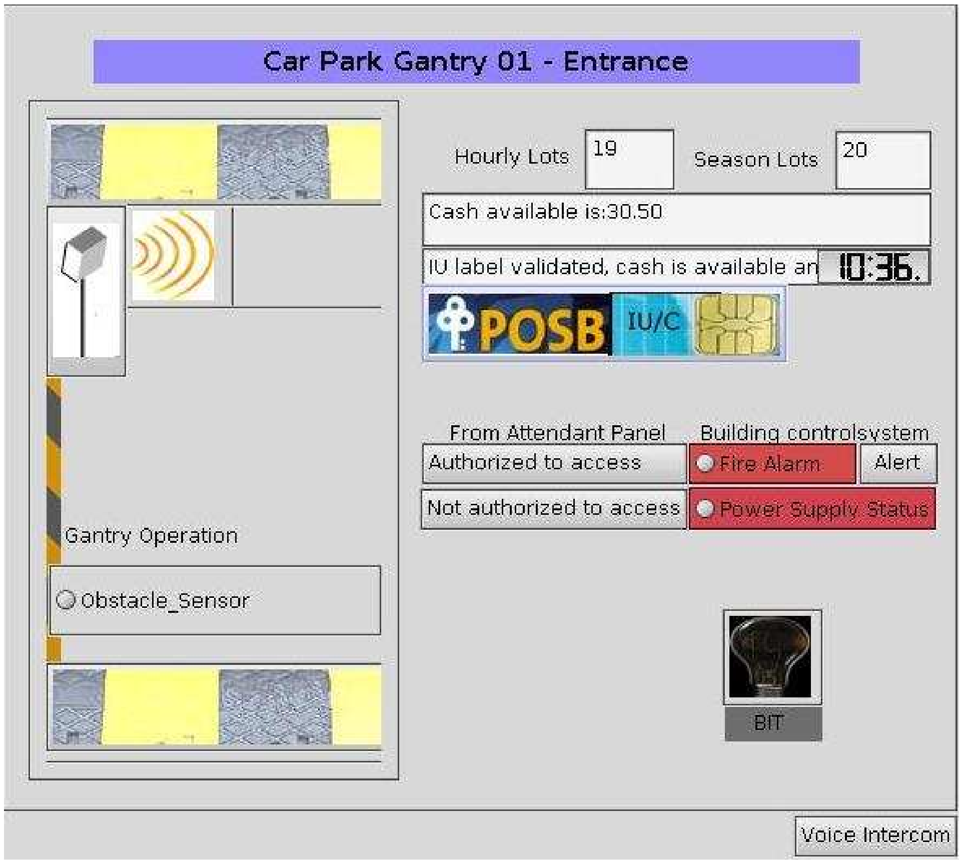
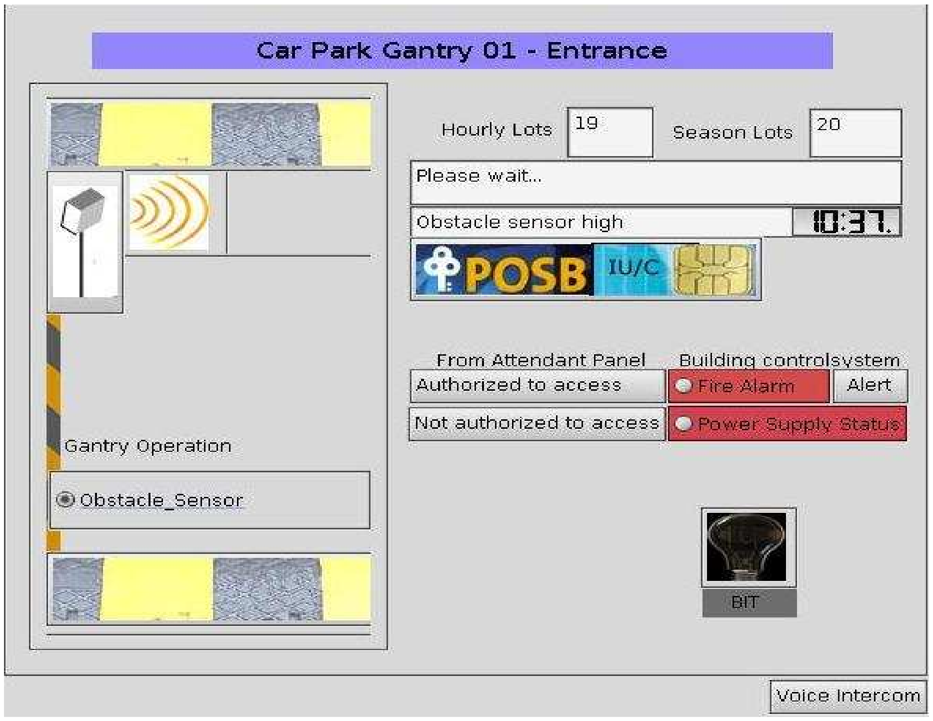


Figure 8: Validation and Entry made in the database

The next state would be the detection by the obstacle sensor so the user clicks it and sensor goes high so the gantry arm will not close as long as the sensor output are high as shown in Fig.9. The next state would be the load sensor 2 going high and then the obstacle sensor is clicked again so as to make it low and gantry arm closes, then the load sensor 2 goes low. The application then waits for the next car to come. In case if the user is stuck he can always press the voice intercom and then the car park attendant will allow him as authorized access or not authorized access as shown in Fig 10 and Fig 11. In case of the season users the season user slots is decreased as shown in the Fig 12 and the remaining states are executed as mentioned in the above.



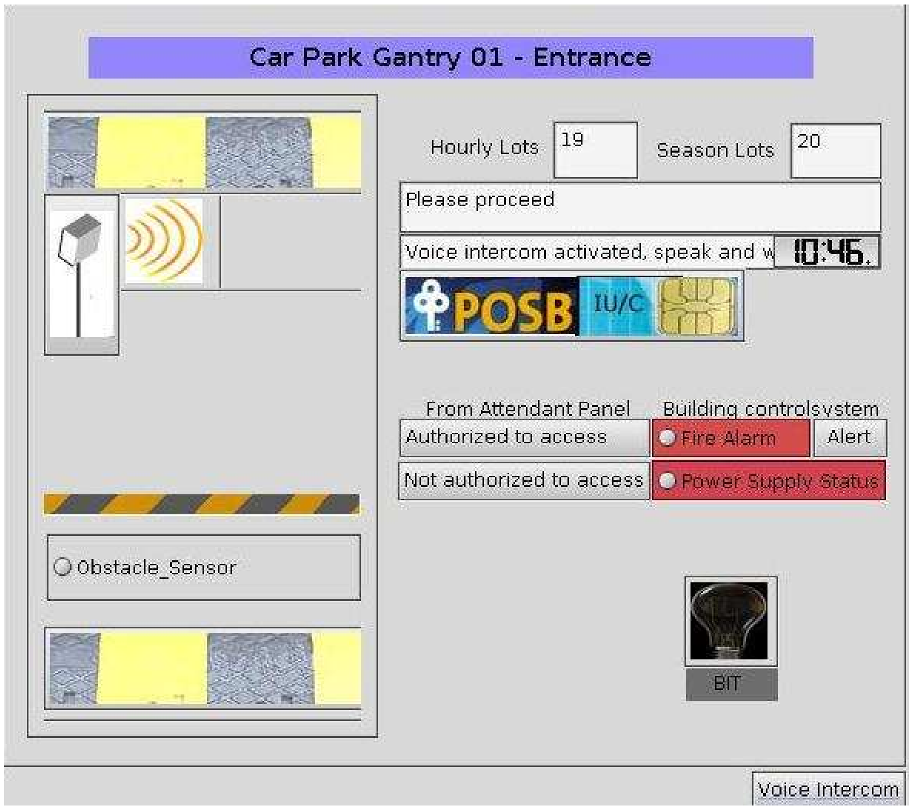
Car Park Gantry 01 - Entrance

Hourly Lots 19 Season Lots 20

Please wait...

Obstacle sensor high 10:37.

Figure 9: Obstacle sensor going high



Car Park Gantry 01 - Entrance

Hourly Lots 19 Season Lots 20

Please proceed

Voice intercom activated, speak and w 10:46.

Figure 10: Voice intercom pressed in case of assistance

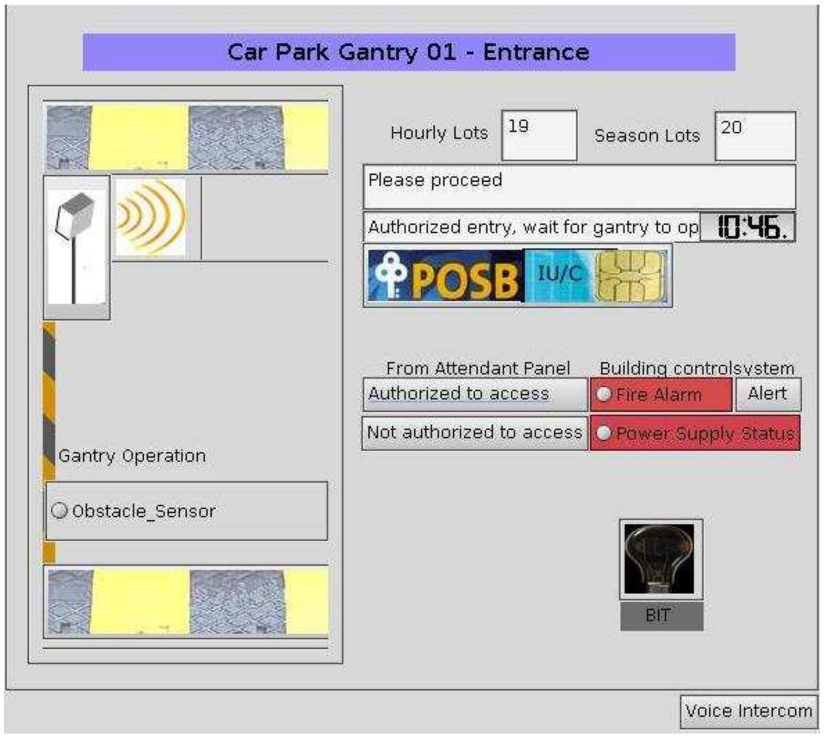


Figure 11: Authorizatoin access or not

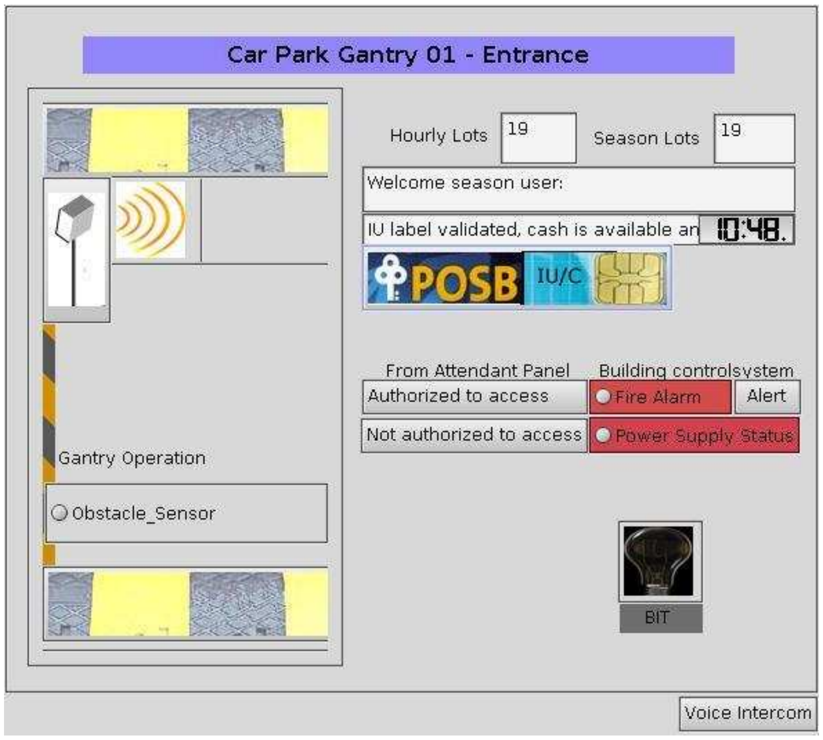


Figure 12: Season user entry

4.4 Exit Scenario

On exit the load sensor 1 go high and then the user clicks label which he wants to exit. Lets say he wants to exit a car which is already in the parking lot. The application checks the user type i.e., whether he is a season user or hourly user and based on it the cash is deducted or not. After cash deduction the gantry arm is opened for the car to go out of the gantry as shown in Fig.13. In case of the hourly user the cash is deducted as 10 cents per min and it can be configured based on the customer.

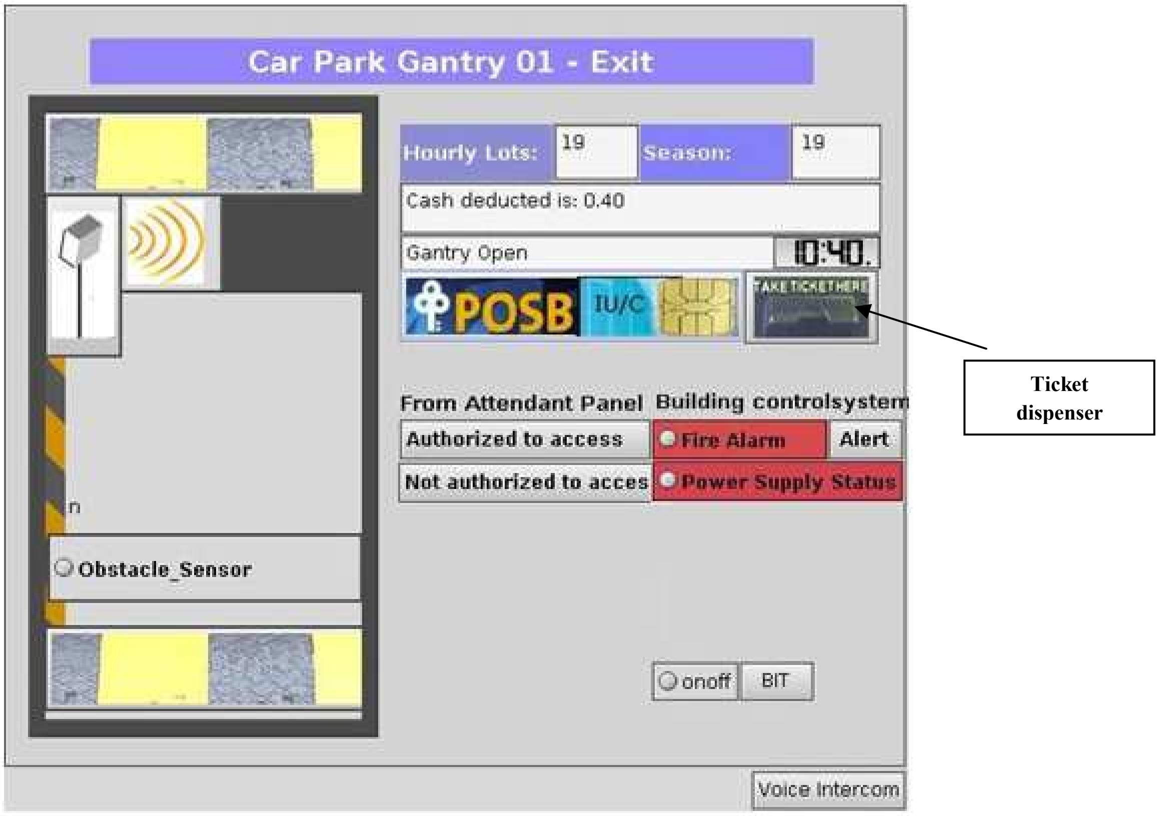
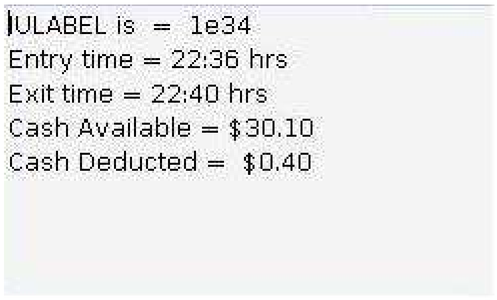


Figure 13: Exit scenario

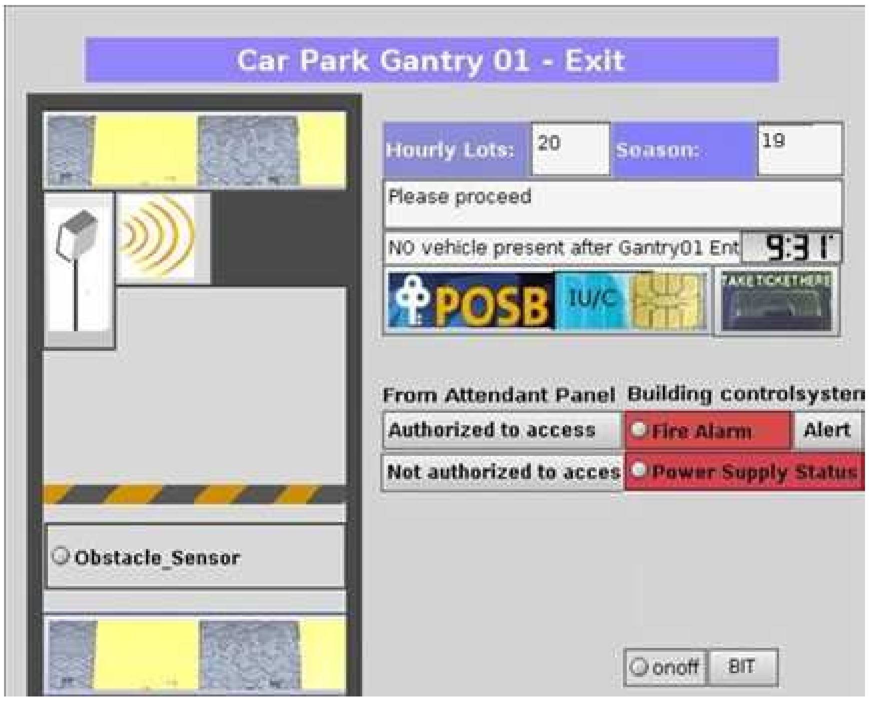
On deduction of cash in case the user wants the ticket, he can press the ticket dispenser and can take the receipt as shown in Fig.14.



```
IULABEL is  =  1e34
Entry time = 22:36 hrs
Exit time = 22:40 hrs
Cash Available = $30.10
Cash Deducted =  $0.40
```

Figure 14: Cash receipt

When the obstacle sensors is cleared the the gantry arm is closed and waits for the next car to exit as shown in Fig.15.


Car Park Gantry 01 - Exit

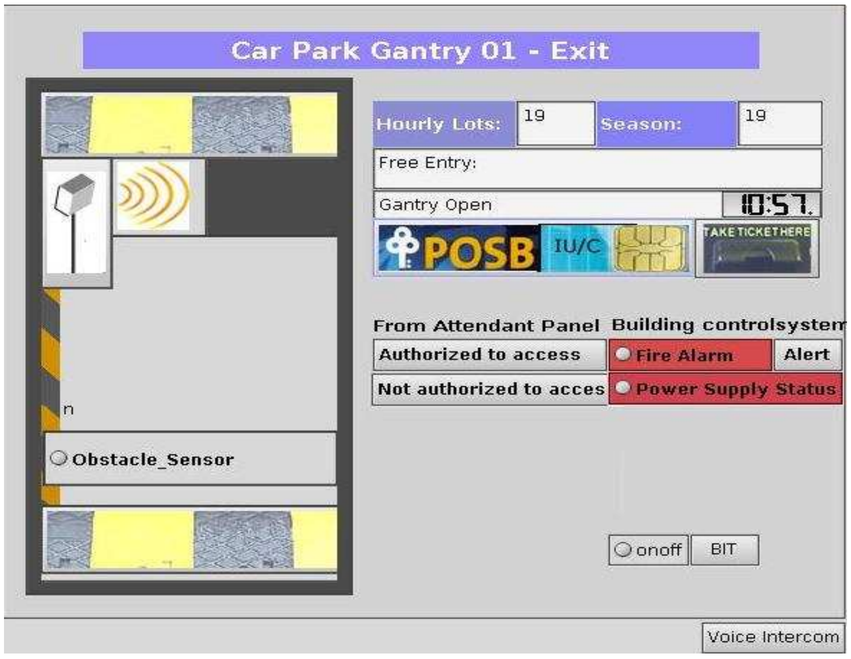
Hourly Lots: 20 Season: 19

Please proceed

NO vehicle present after Gantry01 Ent 9:31

Figure 15: Exit of a car from gantry

In case of season user the cash is not deducted and it as shown in Fig.16.


Car Park Gantry 01 - Exit

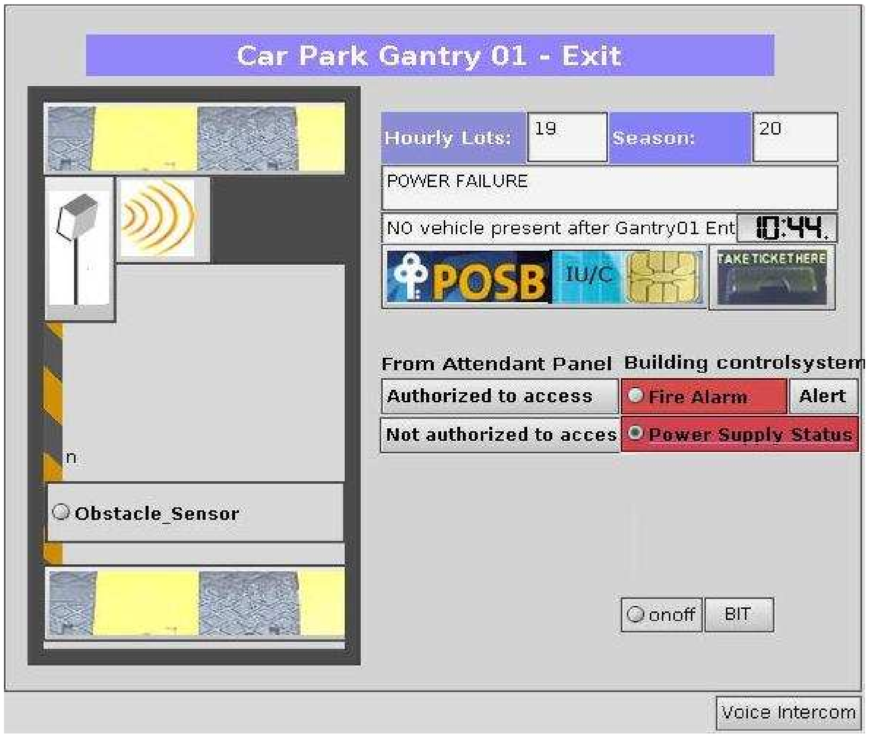
Hourly Lots: 19 Season: 19

Free Entry:

Gantry Open 10:57

Figure 16: Exit of a season user

In case of power failure the gantry arm is opened up and entry and exit are manually operated as shown in Fig.17.

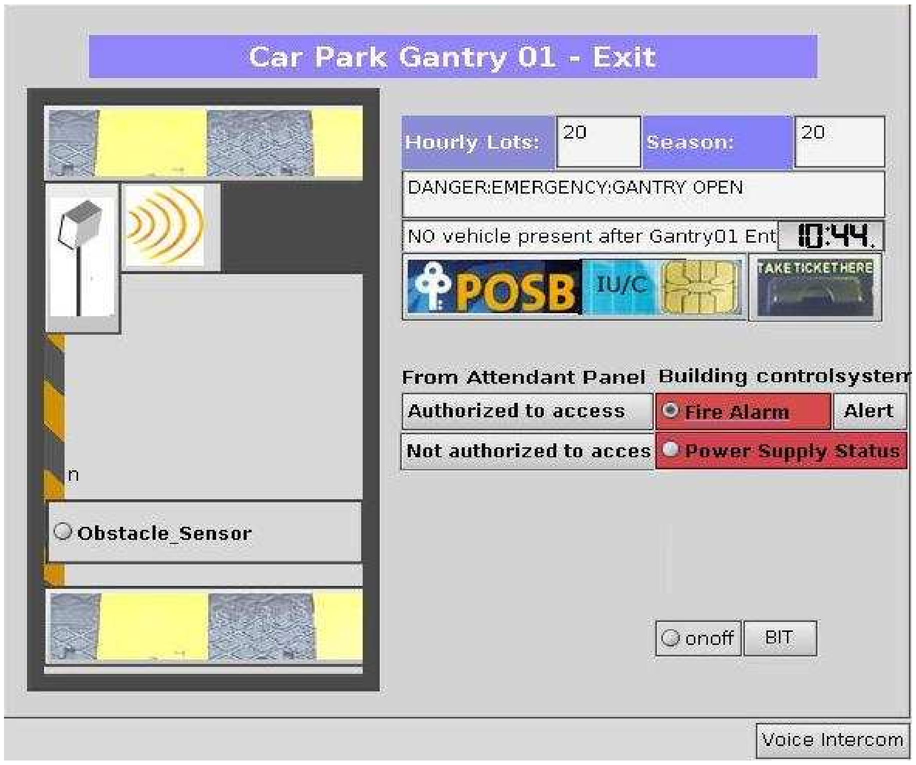


POWER FAILURE

NO vehicle present after Gantry01 Ent 10:44

Figure 17: Power failure scenario

Safety is important and hence in case of any fire in the surrounding premises the gantry open will be opened for the car to leave quickly and the lots are reseted to the initial value as shown in the Fig.18



DANGER:EMERGENCY:GANTRY OPEN

NO vehicle present after Gantry01 Ent 10:44

Figure 18: Fire Alarm

V. Testing

The testing was carried out in phases. Considering the limited time available no exhausted testing was performed, however the testing involved the verification of the implementation of the various critical functions. The entire process of verification and validation was carried out as discussed below:-

5.1 Code Walkthrough

However as specified before the implementation involved a state machine. The section of the code involving the state machine underwent a code review session in the form of a discussion. Individual members performed a walkthrough of the code to verify if the states and the transitions were implemented as discussed during the initial implementation phase.

The output of the code walkthrough involved the following changes to the code:-

a) Implementation was not supporting the scenario involving multiple cars.
b) It was also observed that the handling of the messages were not as per the specification.

After discussion the two issues were resolved accordingly by implementing the required functionality.

5.2 Unit Testing

The modules were handled by individual members and since each member was handling different modules; it was easier to perform the testing. The strategy employed for unit testing involved the testing of the critical functions. The errors that were obtained were rectified accordingly and test cases were re-run to ensure the error was rectified.

The testing involved the database handling functions, Charge calculation, GetData and FormData functions which were used for the message handling. The approach for the testing and the results obtained are discussed as given below:-

5.2.1 Database Functions

The database handling functions which involve interaction with the QDB was considered critical as it was relatively newer concept. Hence it was necessary to test it to ensure that it would support the various functionalities. The tests were designed to verify the following basic functionalities which were critical from the point of the final application:-

1) Insertion of an entry into the database
2) Retrieving an entry from the database by means of a key (here the Label)
3) Deleting an entry from the database by means of a key
4) Deleting all entries from the database
5) Concurrent access to the database

Thus the testing of the above scenarios led to ensuring that the database was working as required and more critically the queries used to retrieve the data were formed correct at runtime.

5.2.2 Testing of Functions GetData and FormData

The functions *GetData* and *FormData* were tested as it were critical functions involving extracting data from the message received and concatenating various data for messages that have to be sent. Further these functions involved the usage of masks and shifting operations, thus the probability of errors occurring were more. The table contains the list of certain sample scenarios that were tested for the two routines.

The functions involved using various masks and shifting operation. From the unit testing it was found that the main cause of most of the errors was improper masks and in certain cases the shift amount. The necessary changes were made and the test cases were re-run again to ensure the test cases passed as expected.

Parameters	Expected Output	Actual Output	Result
***GetData* Test cases**			

message[0] = 0x04
message[1] = 0x5C
message[2] = 0x55
message[3] = 0x10
message[4] = 0x6F
	msgid is 4
	Label is 0x1C55
		msgid is 4
		Label is 0x55C0.
		(First round)
		Similar failure observed for all Label Cases
			FAIL

	msgid is 4 Label is 0x1C55	msgid is 4 Label is 0x1C55	**PASS**

message[0] = 0x06
message[1] = 0x5E
message[2] = 0x75
message[3] = 0x20
message[4] = 0x0F
	msgid is 6
	Label is 0x1E55
		msgid is 6
		Label is 0x1E55
			PASS

message[0] = 0x04
message[1] = 0x5C
message[2] = 0x55
message[3] = 0x10
message[4] = 0x6F
	msgid is 4
	Dollar value is 10
	Cents values is 60
		msgid is 4
		Dollar value is 10
		Cents values is 60
			PASS

message[0] = 0x06
message[1] = 0x5E
message[2] = 0x75
message[3] = 0x21
message[4] = 0x1F
	msgid is 6
	Dollar value is 21
	Cents values is 10
		msgid is 6
		Dollar value is 21
		Cents values is 10
			PASS

message[0] = 0x08
message[1] = 0x5F
message[2] = 0x77
	msgid is 8
	Label is 0x1F77
		msgid is 8
		Label is 0x1F77
			PASS

message[0] = 0x10
message[1] = 0x10
	msgid is 16
	Alarm Triggered
		msgid is 16
		Alarm Triggered
			PASS

message[0] = 0x10
message[1] = 0x08
	msgid is 16
	Power Supply CutOff
		msgid is 16
		-
		(First Round)
			FAIL

	msgid is 16 Power Supply CutOff	msgid is 16 Power Supply CutOff	**PASS**
FormData Test cases			

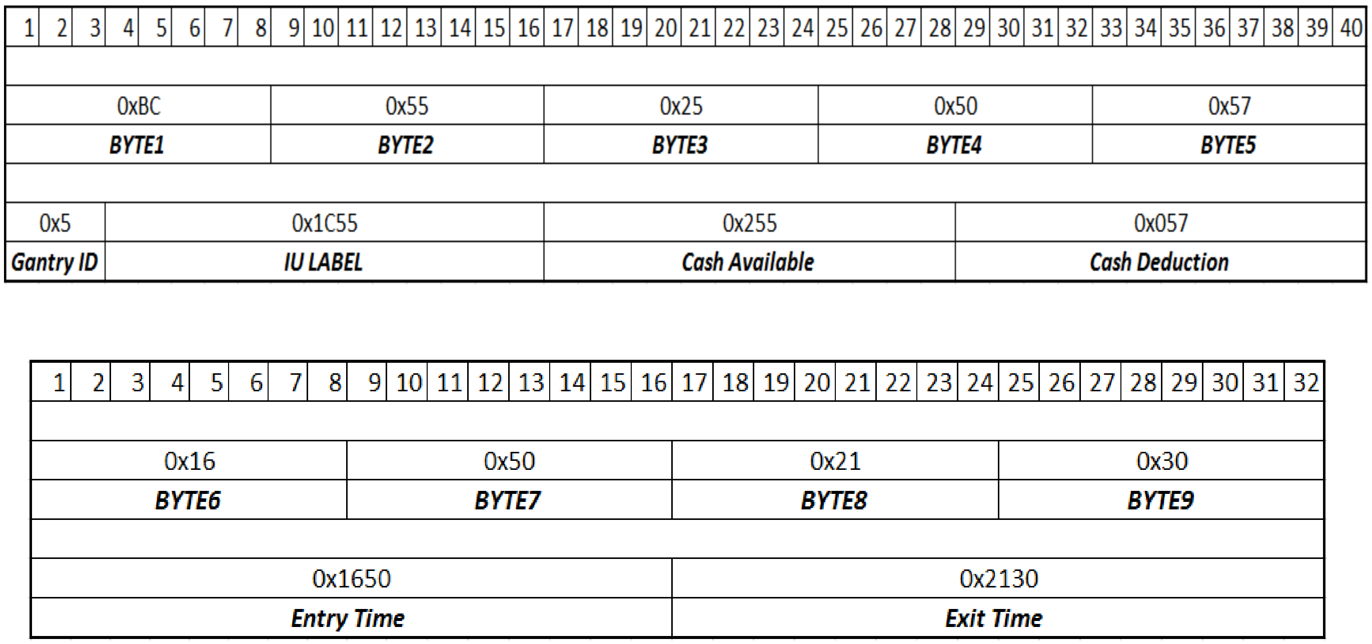
msgtobesent.dollar_deduct = 1;
msgtobesent.cents_deduct = 50;
msgtobesent.dollar_avail = 0;
msgtobesent.cents_avail = 0;
msgtobesent.entry_hr = 0;
msgtobesent.entry_min = 0;
msgtobesent.exit_hr = 0;
msgtobesent.exit_min = 0;
msgtobesent.label = 0;
msgtobesent.msgid = 5;
msgtobesent.gantryid = 0x02;
	Message is
	0x05 0x40 0x15
		Message is
		0x05 0x40 0x15
			PASS

msgtobesent.dollar_deduct = 2;
msgtobesent.cents_deduct = 25;
msgtobesent.dollar_avail = 0;
msgtobesent.cents_avail = 0;
msgtobesent.entry_hr = 0;
msgtobesent.entry_min = 0;
msgtobesent.exit_hr = 0;
msgtobesent.exit_min = 0;
msgtobesent.label = 0;
msgtobesent.msgid = 7;
msgtobesent.gantryid = 0x05;
	Message is
	0x07 0xA0 0x22
		Message is
		0x07 0xa0 0x22
			PASS

msgtobesent.msgid = 11;
msgtobesent.gantryid = 0x05;
msgtobesent.label = 0x1c55;
msgtobesent.dollar_avail = 25;
msgtobesent.cents_avail = 50;
msgtobesent.dollar_deduct = 5;
msgtobesent.cents_deduct = 75;
msgtobesent.entry_hr = 16;
msgtobesent.entry_min = 50;
msgtobesent.exit_hr = 21;
msgtobesent.exit_min = 30;
	Message is
	0x0B
	0xBC
	0x55
	0x25
	0x50
	0x57
	0x16
	0x50
	0x21
	0x30
		Message is
		0x0B
		0xBC
		0x55
		0x25
		0x50
		0x57
		0x16
		0x50
		0x21
		0x30
			PASS

Table 3: Test Cases for GetData and FormData

The final test case involves sending a message to the ticket dispenser and could be understood better by means of the below figure showing the message split into the various fields:-



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
0xBC								0x55								0x25								0x50								0x57							
BYTE1								***BYTE2***								***BYTE3***								***BYTE4***								***BYTE5***							
0x5			0x1C55													0x255												0x057											
Gantry ID			***IU LABEL***													***Cash Available***												***Cash Deduction***											

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0x16								0x50								0x21								0x30							
BYTE6								***BYTE7***								***BYTE8***								***BYTE9***							
0x1650																0x2130															
Entry Time																***Exit Time***															

Figure 19: Message Format for Ticket Dispenser

5.3 Integration Testing

The integration testing was performed in four stages. The activity involved in the four stages can be discussed as follows:

5.3.1 Integration of modules for entry scenario

This activity involved the integration of the various modules to handle the entry scenario. There were two main ***stubs*** that were used for this activity – one stub for the database handling functions and the other stub was used for the simulator. The simulator stub was a process used for the purpose of sending messages in sequence to the application to simulate the entry of the car. The purpose of the testing was to ensure that the state machine was working in the correct sequence and as expected. Scenarios that formed the test cases include

1. Entry of an hourly IU user
2. Entry of a Season IU user

3. Entry of a Cash-Card user
4. Multiple Car Entry
5. Failure scenario

5.3.2 Integration of modules for exit scenario

As the structure of the entry process and the entry process were similar, the reusability was also possible during the testing. The activity for the exit scenario involved the usage of the stubs used for the entry scenario, with the required modifications. Scenarios that formed the test cases include

1. Exit of an hourly IU user
2. Exit of a Season IU user
3. Exit of a Cash-Card user
4. Ticket Dispenser

5.3.3 Integration of entry and exit modules with the database

This activity involved the integration of the entry module and exit module with the database separately. The modules after integration with the database module was subjected to test the scenarios specified above to verify proper functionality and ensure that the integration of the database modules did not result in any unexpected behavior. The database was updated using stubs externally.

5.3.4 Verification of the simulator

The photon based simulator with GUI was also verified to ensure its proper functioning. The application was implemented as a stub which would receive the message from the simulator. The various callbacks used for the buttons on the GUI were verified to ensure that the expected and correct message format was sent from the simulator.

5.3.5 Integration of the simulator with entry and exit

The next stage involved the integration of the entry and the exit modules with the respective simulator. The simulator was designed such that the scenarios that were tested on the exit and entry modules could easily be ported in to the simulator callback functions. It provided with the interface where the user could select any one of the labels and can simulate the entry and the exit scenario accordingly by pressing the buttons on the simulator in a defined sequence. In the exit scenario the simulator was also designed so that it could accept the messages from the application as in the case of the messages to the ticket dispenser, IU unit and to the cash card reader. The ticket dispenser module in the simulator was designed so that it would decode the message and display the various fields.

5.4 System Testing

Once the integration testing phase was completed successfully, the next step involved the integration of the entire system. This involved integrating the two modules, namely the entry and the exit modules along with their simulators (a total of four processes). The integration was successful in the first attempt as there was no problem in running both these processes in tandem. The first scenario that was tested involved the simulation of the entry of a car and after a certain time period to allow for charge calculation the simulation of the exit was performed. The output was as expected with the display messages appearing as expected, the charge deduction and ticket dispenser details were also correct and matching with the details that was registered during the entry process. Further the values displaying the number of lots were updated synchronously on the entry and exit of the car. Following this the other scenarios specified in the integration were tested on the system as a whole.

VI. Maintenance:

After deployment of the project, comes the final stage – Maintenance and Upgrading. The maintenance cost about 40%-60% of the software cost. Depending upon the life span of the software that is been deployed, the maintenance cost also varies. Maintenance of the software holds a key point in life cycle of the Car park gantry software. The typical maintenance cycle involves

- Problem/modification Identification and Classification
- Analysis
- Design
- Implementation
- System test
- Acceptance test
- Delivery

6.1Problem/Modification identification and Classification:

We have followed State machine architecture for the project Car park gantry system. It enables us to handle each of the states independently and perform maintenance. If any problem happens to any of the modules /states, because of the architecture efficiency, it is possible to perform maintenance without affecting the normal running of the rest of the modules. So modularity of the software holds a key point that helps us in flexibility on maintenance of out project.

6.2Analysis:

The project is been designed in such a way that further modification and future implementation can be done. The system architecture that we followed (module/ state machine approach) provides us with a comparable degree of freedom for modification. Modification of charging unit can be done depending upon the requirement of the customer at any stage of the system.

6.3 Design:

The future modification can be implemented on the existing software with ease. The coding of the existing software is written in such a way that it will be easy for a new programmer or developer to understand and follow the logic and do the modification according to the customer requirement.

6.4 Implementation ad testing:

Coding of the software is easy to follow as all the variables, structures and pointers are clearly mentioned with proper comments. So it will be easy to implement and test the code because of the modular structure as well as the standard that is been followed for the development of the program.

Some of the success factors that are incorporated in the software are

6.5 Functionality:

The functionality of the entire system is been maintained i.e., even if any sort of modification or future implementation and maintenance to be done, then it can be performed without affecting the functionality or end aim of the software. Hence the actual functionality of the system is been maintained.

6.6 Quality:

The quality of the system is preserved. While designing and coding, extreme care is been taken so that software integrity is not compromised.

6.7 Code Quality:

The entire system is been developed based on unique coding standard so that it will be easy for the maintenance people as well as new program developer to analyze, understand and modify if required. Also we have followed proper commenting of each line and hence it is easy for a third person to have a look at the code and understand the functionality of each modules and it is inter connectivity between the modules.

6.8 User satisfaction:

The developed system have covered almost 90% of the customer requirements such as

- Normal gantry entry - exit scenario
- Cash card reader and IU card reader facility
- Available lot display
- Ticket dispensing unit
- Display unit
- Obstacle sensor for car damage avoidance
- Building Fire alarm protection

Also it provide a central data base facility and charging unit re configurability so that the customer can change the hourly lot according to the change of demand. Because of the software structure that is followed in the development of the system, it allows flexibility in maintenance of the entry or the exit scenario independently without disrupting the working of other.

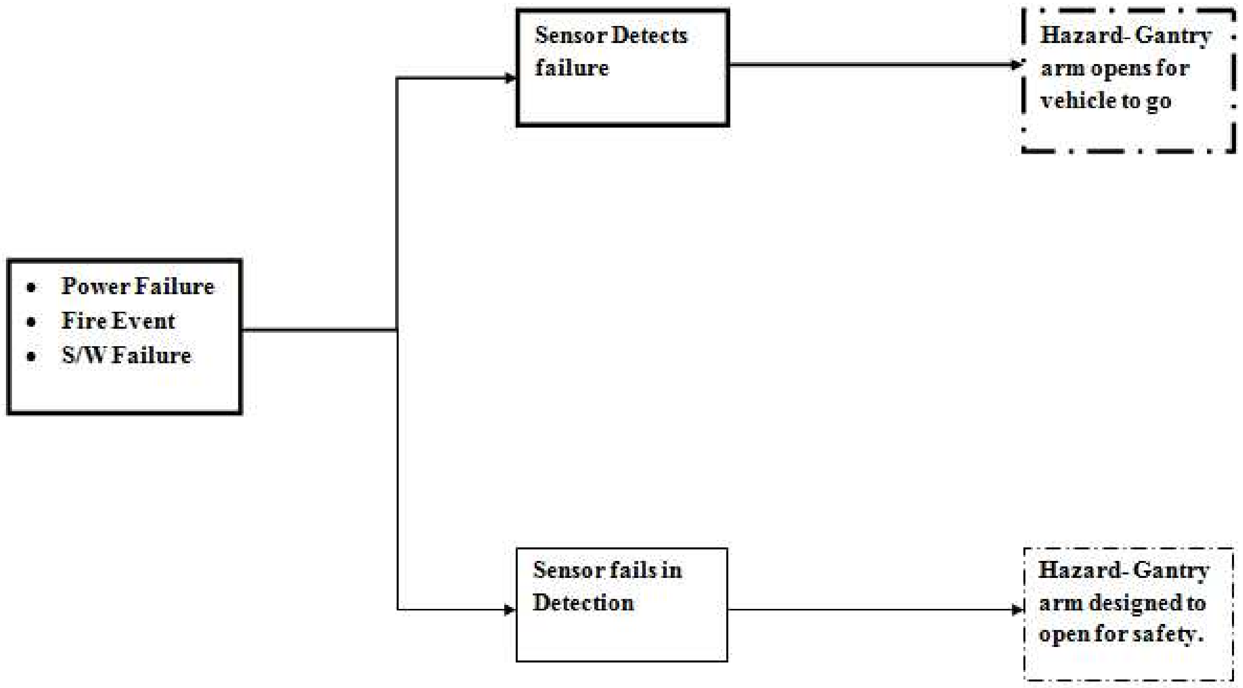
In addition to the above mentioned functionality, the modification is also possible because of the coding style and implementation methodology. This is one of the quality attribute that is been followed by us throughout the development of the project. The stand out feature of the project is the database unit holding the seasoned user and hourly user separately. Because of this feature, it is possible for the customer to configure the database at any point of time i.e., adding/ deletion of seasoned user, modification of normal user data base are possible without any disruption of the system. Even the CHU [charging Unit] can be easily reconfigured.

Modular / state transition approach allows us to upgrade the system at any point of time. All we have to do is to add new module to the exiting one after testing. Because of the modular approach, the up gradation of the system is pretty easy because it allows the developer to upgrade the system in a peaceful manner without ripping off the entire program. All we have to do is to write a new module that describing add-on feature and plug into the system and make necessary calls to the modules as and when required. This type of approach avoids the ripple effect which normally occurs for unstructured program development. Also portability is been achieved because of the same modular architecture and also minimum sharing of data. Only necessary information's are passed between each state.

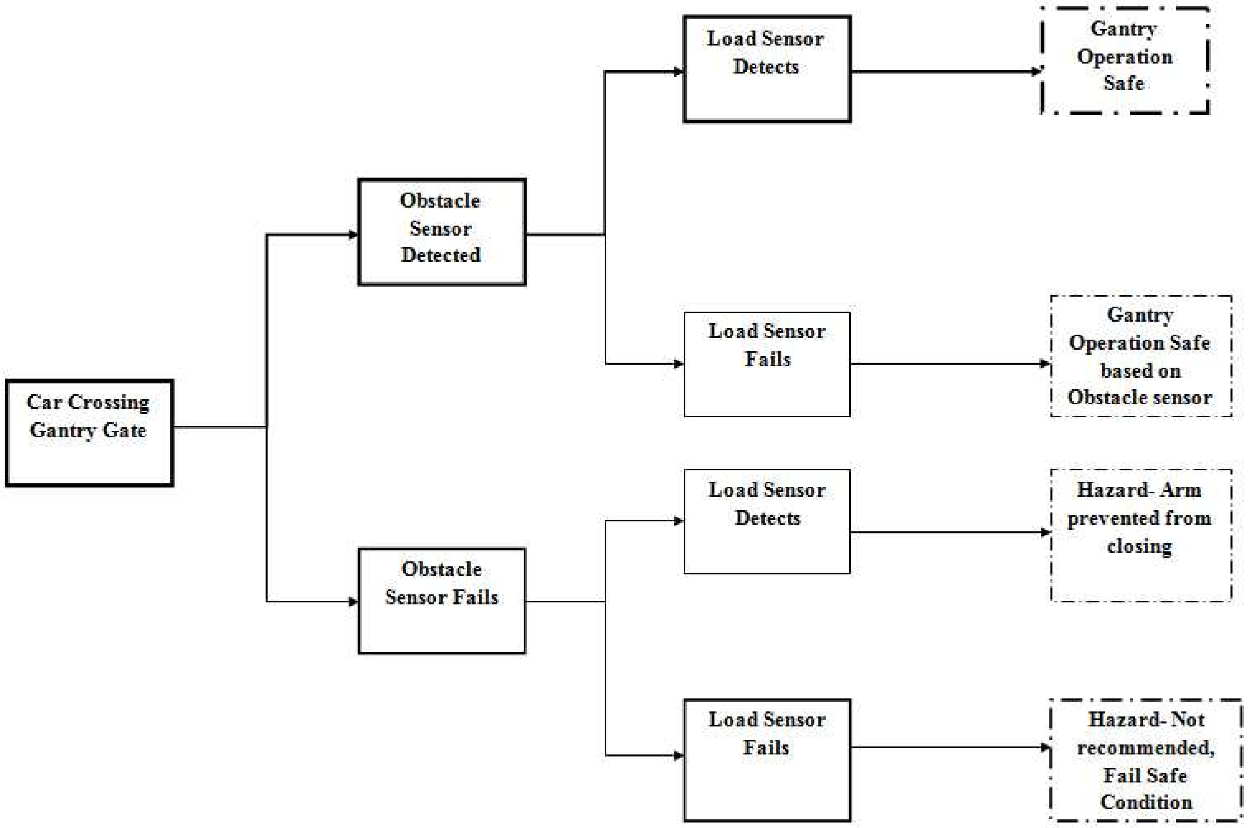
VII. Conclusion:

The process of learning continues throughout one’s life says an old adage. As a person goes on learning new things, he realizes that there are many more things which he needs to learn to quench his thirst for knowledge. In the process of doing this project we as a team learnt that we need to be proactive, which is taking responsibility. Each of us took responsibility from the requirement gathering to the release of the project. We started this project with the end in mind. Since it is a six member team numerous ideas and approaches towards the design and implementation were discussed. We decided to put first things first such as the quality attributes required for the software and the safety critical scenarios. At the end of all brain storming sessions it will be difficult to convey the ideas. The best way to convey your ideas is first seek to understand, then to be understood. Always two heads are better than one we all of us had worked as a team. The simulator team was developing the user interface and another team was developing the application. Both the teams were communicating during the implementation stage. This project made all of us to portray the skills which we perceive and also made us to sharpen the saw. Apart from the fulfilling the coursework criteria each of us shared whatever they had experienced during the progress of this project. Last but not the least it is always better to think out of box because winner don’t do different things they do things differently. We did things differently.

VIII. Appendix: Event Tree Analysis



• Power Failure
• Fire Event
• S/W Failure
Sensor Detects failure
Hazard- Gantry arm opens for vehicle to go
Sensor fails in Detection
Hazard- Gantry arm designed to open for safety.


Car Crossing Gantry Gate
Obstacle Sensor Detected
Obstacle Sensor Fails
Load Sensor Detects
Load Sensor Fails
Load Sensor Detects
Load Sensor Fails
Gantry Operation Safe
Gantry Operation Safe based on Obstacle sensor
Hazard- Arm prevented from closing
Hazard- Not recommended, Fail Safe Condition