VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

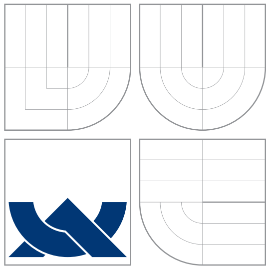
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TOUCH SCREEN USER INTERFACE FOR ELECTRIC CAR

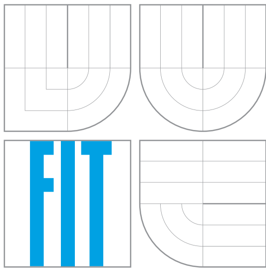
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ONDŘEJ MARTINÁK
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DOTYKOVÉ UŽIVATELSKÉ ROZHRANÍ ELEKTROMOBILU

TOUCH SCREEN USER INTERFACE FOR ELECTRIC CAR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. ONDŘEJ MARTINÁK
AUTHOR

VEDOUCÍ PRÁCE — doc. Dr. Ing. DUŠAN KOLÁŘ
SUPERVISOR

BRNO 2010

Abstrakt

Tato práce navrhuje dotykové uživatelské rozhraní pro elektromobily. Rozhraní bylo navrženo tak, aby bylo pohodlné na použití pro řidiče a aby bylo snadné z něj vyčíst potřebné informace. Aby bylo možné toto rozhraní integrovat do elektromobilu, tato práce také navrhuje potřebnou softwarovou a systémovou (hardwarovou) architekturu. Nakonec shrnuje co je ještě potřeba udělat, aby bylo možné celý systém integrovat.

Abstract

This work proposes a touch screen user interface for elecric cars. The user interface aims to be comfortable for the driver to use and to offer easy way of getting information about various car functions. To be able to integrate this interface to the car, this work also proposes necessary software and system (hardware) architecture. In the end it summarizes what should be done next to integrate the whole system.

Klíčová slova

carputer, linux, dotyková obrazovka, uživatelské rozhraní

Keywords

carputer, linux, touch screen, user interface

Citace

Ondřej Martinák: Touch Screen User Interface for Electric Car, diplomová práce, Brno, FIT VUT v Brně, 2010

Touch Screen User Interface for Electric Car

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Dr. Ing. Dušana Koláře.

. . . .
Ondřej Martinák
May 25, 2010

Poděkování

Chtěl bych poděkovat svému konzultantovi Ing. Jaromíru Marušincovi, Ph.D. MBA za jeho rady a vedení během mé práce.

© Ondřej Martinák, 2010.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1 Introduction 4
1.1 Thesis Focus . . . 4
1.2 Thesis Structure . . . 5

2 System Architecture 6
2.1 Overview . . . 6
2.2 Computer . . . 7
2.2.1 VIC NAVISURFER . . . 7
2.2.2 mp3car Store . . . 7
2.2.3 Mini-Box . . . 7
2.3 Touch screen . . . 7

3 Software Architecture 9
3.1 Overview . . . 9
3.2 Abstract Architecture . . . 10
3.2.1 CAN Bus . . . 10
3.2.2 Operating System . . . 11
3.2.3 Modules . . . 11
3.2.4 Mediator . . . 11
3.2.5 Command Line Interface . . . 11
3.2.6 Graphical User Interface . . . 11

4 Graphical User Interface Design 13
4.1 Users and Tasks . . . 13
4.2 Conceptual model . . . 15
4.3 Task scenarios . . . 17
4.4 Touch Screen Usability . . . 18
4.5 Widgets . . . 19
4.5.1 Car Range . . . 19
4.5.2 Immediate Consumption . . . 20
4.5.3 Batteries' Energy Levels . . . 20
4.5.4 Quick Charging . . . 20
4.5.5 Menu . . . 21
4.5.6 On Screen Keyboard . . . 21
4.6 Screens . . . 22
4.6.1 Standing Car . . . 22
4.6.2 Moving Car . . . 22
4.7 Applications . . . 23

4.7.1 Login . . . 23
4.7.2 Collision Detection . . . 23
4.7.3 Charger . . . 24
4.7.4 Rear Camera . . . 25
4.7.5 Navigation . . . 25
4.7.6 Multimedia Player . . . 25
4.7.7 Internet Browser . . . 25
4.8 Colors and Ergonomics . . . 26
4.9 Final User Interface . . . 26

5 Implementation **28**
5.1 Overview . . . 28
5.2 LinuxICE . . . 28
5.2.1 nGhost . . . 29
5.2.2 Icepanel . . . 29
5.2.3 Matchbox . . . 29
5.3 Core Implementation . . . 30
5.4 D-Bus Interface . . . 31
5.5 Applications . . . 32
5.5.1 Run Scripts . . . 33
5.5.2 Login . . . 33
5.5.3 Shutdown . . . 34
5.5.4 Main Screen Information . . . 34
5.5.5 Battery Charging . . . 34
5.5.6 Rear Camera . . . 35
5.5.7 Collision Detection . . . 36
5.5.8 Music Player . . . 36
5.5.9 Navigation . . . 36
5.5.10 Web Browser . . . 36
5.5.11 Simulation . . . 36
5.6 Modules . . . 37
5.6.1 Run Scripts . . . 37
5.6.2 Battery . . . 37
5.6.3 Car_state . . . 38
5.6.4 Charger . . . 38
5.6.5 Collision . . . 38
5.6.6 Canbus . . . 38
5.6.7 Logger . . . 38
5.7 Skins and Configuration . . . 39
5.8 LiveDVD Creation . . . 39
5.8.1 Install Script . . . 39
5.8.2 Creating the Image . . . 40

6 Results **42**
6.1 Simulation . . . 42
6.2 Speed . . . 43
6.3 Extensibility and Integration . . . 43

A CarPC Software Overview **46**
A.1 PyCar . . . 46
A.2 Headunit . . . 46
A.3 DashPC . . . 47
A.4 KDE Software Compilation 4 . . . 47
A.5 OpenICE/nGhost 3 . . . 48

B User's Guide **50**
B.1 Running . . . 50
B.2 Installation . . . 50
B.3 Usage . . . 50
B.3.1 Login . . . 50
B.3.2 Main Screen . . . 50

C CD Contents **52**

Chapter 1

Introduction

1.1 Thesis Focus

Recently, electric cars[6] have been growing in popularity as they offer effective and relatively clean means of transport, mainly inside the cities. Some of the world countries (like USA, Germany, . . .) have already started actions that should help this growth even more. Nevertheless, major automobile companies are still somewhat reluctant to incorporate the electric cars in the mass production.

To promote the electric cars in Czech republic a *Superbel*[22] project was initiated. The aim of this project is to take an existing Škoda Superb and alter it to use an electric propulsion. The new car will be a prototype serving as a platform for further research and development in the field of electric cars.

Figure 1.1: Škoda Superb (picture taken from [27])

As the new car will feature new type of propulsion, energy source, quick charger and other parts and controllers to make it all work, it will also need new ways to show the state of the new equipment to the driver. To a certain degree, the car's dashboard might be adjusted to accommodate these needs, but this solution is not very robust.

Another solution would be to use a computer interface featuring a touch screen. This interface can be programmed to present whatever data is necessary to show the driver. It's also relatively easy to modify it or create, for example, new control elements. The use of a touch screen is only natural as the driver wouldn't be able to use traditional input devices like mouse or keyboard.

This thesis aims to create such interface and integrate it with the *Superbel* project.

Figure 1.2: Škoda Superb's Dashboard (picture taken from [21])

1.2 Thesis Structure

This section provides quick overview of each chapter of the thesis. It should serve as a starting point, allowing the reader the get better orientation in the document.

- **System Architecture** describes the high-level architecture, focusing on interaction between various hardware parts. It also describes some options for car computer and touch screen.
- **Software Architecture** describes abstract architecture of the software presenting the overview of the system and underlying techniques used to make the design modular and easily extensible.
- **Graphical User Interface Design** focuses on analysis and design of the interaction layer users will use to communicate with the car computer. It presents requirements on the interface, preferred ways of working and individual graphical elements designs.
- **Implementation** describes the way the system was implemented. It gives an overview of the whole architecture as well as detailed information about the platform, libraries and technologies that were used. It also shows how the application and low-level modules were created and how they interact together.
- **Results** summarize the whole project and presents the goals that were achieved. This chapter also describes simulations done to test the project. In the end, it proposes a few ideas how the project could be extended in the future.
- **Appendices** contains information about various alternative car computer software as well as user's guide and a list of included CD/DVD content.

Chapter 2

System Architecture

2.1 Overview

This project will give the driver (or passengers) a new way to interact with the car. They will be able to monitor the state of batteries or estimated range, they will be able to listen to music as they are used to or set the car charger as needed. They will also be able to track their location or plan their travel with a GPS device. For this purpose the car will need a computer that is more than just a multimedia player. It will need a universal computer that will be able to communicate with the car and will be able to run the software part of the user interface. Picture 2.1 shows how this computer will communicate with its surroundings.

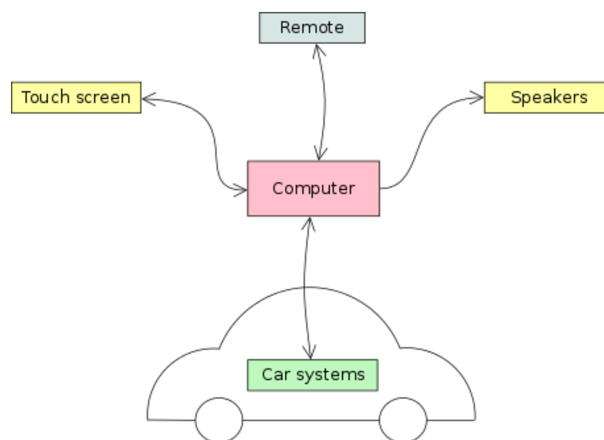


Figure 2.1: System Overview

It will take input from the user through a touch screen while the output can be seen in the form of graphics on the same screen or heard as a sound or music from the car's speakers. Furthermore, engineers or technicians will be able to connect to the computer remotely so they will be able to do whatever maintenance is necessary.

The computer will also communicate with the car through CAN bus[2]. It is a standardized bus used in vehicles for communication between various micro-controllers and devices, like engine control unit, car charger and others. This bus will be a major source of information for the computer and a primary mean to control some of the functions of the

car.

2.2 Computer

Universal car computers (or "carputers") are not a common thing in today mainstream automobiles. The mainstream is usually limited to CD and audio players. The more expensive classes of cars tend to have more sophisticated computers, but they too are usually limited to a few areas of functions like playing multimedia or phone support. But there is also a community of modders[31] who like to upgrade their cars, be it for the entertainment of the modding or the results it gets them. And since there is a demand for computers (or their parts) that can be used for this purpose, there are also stores that offer them.

2.2.1 VIC NAVISURFER

VIC Ltd[26] offers a good selection of car computers as well as accessories (touch screens, GPS modules, . . .). It's solutions cover consumer and commercial needs. One of their products, *Navisurfer*, seems to be an interesting choice for the needs of this project. It comes in variations with or without stand-alone touch screen and apart from the standard equipment it includes features like GPS module, wireless network connection and bluetooth. It's also adapted to work in a car by anti-shock hard drive mounting and battery discharge prevention.

2.2.2 mp3car Store

E-shop on the mp3car website[13] offers a variety of tools for car diagnostics, multimedia players, navigation modules, computer parts and some models of car computers. One example could be the *Fanless Micro Intel Vehicle Computer* that contains all the standard equipment (CPU, graphics adapter, hard drive, operating system, . . .), but anything else needs to be added externally.

2.2.3 Mini-Box

Another e-shop that deals in car computers and their parts is Mini-Box[12]. It offers a similar carputer model as in the previous case, *VoomPC-2 Car PC Barebone*. Again, this model contains only the standard equipment so it would be needed to add the extra functions.

2.3 Touch screen

The car is already equipped with a touch screen integrated to the middle of the dashboard as can be seen in picture 1.2 or in more detail in picture 2.2. This display is well placed as it is easily reachable by the driver or passenger and it fits nicely to the dashboard. Therefore it is natural to use this screen for user-computer interaction in the car.

Another option is to use external touch screen, perhaps one shipped with the car computer. The reasons for an external display might be better parameters like bigger or brighter screen. But it would either be in the way of the present dashboard controls or it would require to exchange it with the existing touch screen, which would result in alteration of

Figure 2.2: Škoda Superb's Touch Screen (picture taken from [21])

the dashboard. Therefore, the existing touch screen is a primary choice for the needs of this project.

Chapter 3

Software Architecture

3.1 Overview

The previous chapter summarized the hardware part of the system and as was indicated, the car computer will run a software that will present it's users with a user interface for monitoring and controlling the car. You can see an overview of how this software will be composed in picture 3.1.

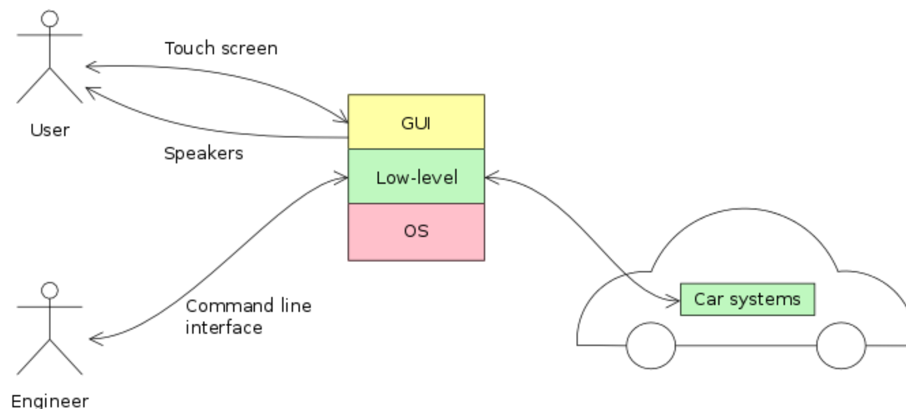


Figure 3.1: Software Architecture Overview

The software will be build from the ground up, which means that the lowest part is an operating system.

The next part consists of low-level tools and utilities that will be responsible for communication with various car functions as well as collection of data about the system or its control. These tools will be in the form of modules and each one of them will take care of one specific task. This level will also contain command line utilities for the maintenance of the system.

The top level consists of a graphical user interface that will present any data collected by the low-level modules to the user and will also allow him to control the system.

3.2 Abstract Architecture

Picture 3.2 shows detailed view of the system described above. This view is abstract which means that it is not tied to any concrete software technology yet. It shows what the system should do and how its components will interact. Concrete implementation will be presented in chapter 5.

The whole idea is based on the mediator[29] and data bus[28] design patterns. The mediator object acts as a common storage for all the data the system uses. Furthermore, clients (low-level modules and high-level frontends) can subscribe to it and the mediator will notify them when, for example, new data is available. Clients can also push data to the mediator. Thus, the mediator defines a common interface for communication between different parts of the system and offers a common place to store the data.

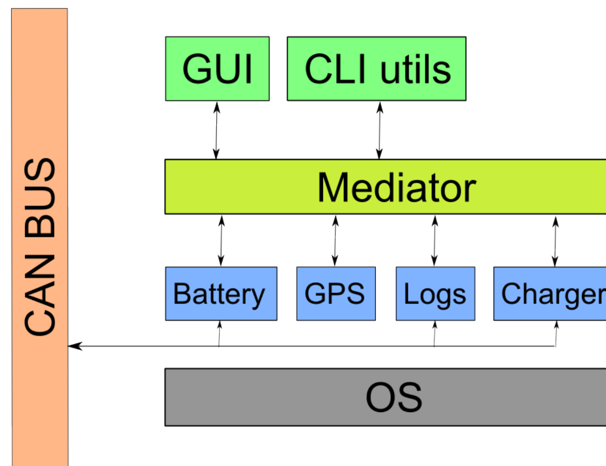


Figure 3.2: Software Architecture Detail

It is then relatively easy to extend such a system. If you would like to, say, add support for cooling monitoring, you would create a module that will collect the necessary data and pass them to the mediator. Then, in the GUI, you would create an element that would subscribe for this data and show it to the user.

The system adheres to the Unix philosophy[1] as it is heavily modular and every module takes care of one specific task. This kind of design was chosen to make the system clean and easily extensible. It is supposed that in the future new modules and GUI elements will be needed to monitor and control new car equipment. This design should also make it possible to use the system in different electric cars or even vehicles.

3.2.1 CAN Bus

The CAN bus is a standardized bus that delivers data from one device to another. It uses a message based protocol, which means the data is transferred as message frames. Each frame contains an ID and up to eight bytes of data. The messages are then broadcasted through the bus so every connected device receives them. The device then decides what to do with it based on its id.

[1]Unix philosophy generally says that a program should "do one thing and do it well"[25]

The devices are usually connected to the bus through a host processor and a CAN controller. To connect a computer with USB port a CAN-to-USB converter should be sufficient.

3.2.2 Operating System

As the software is build from the ground, the computer needs an operating system. Such system should be suited to run in a car computer and it should also make it possible and comfortable to develop and run all the applications needed for this project.

3.2.3 Modules

The lower level of the system is made up of modules that will be responsible for communication with the environment (getting data from GPS module, getting data from batteries, and so on). Each one of these modules will be responsible for one specific task, like monitoring the batteries, tracking GPS location and so on.

These modules should be designed to communicate with the generalized interface of the mediator. If they would need to communicate with each other, they should do so through the mediator, not directly. This will maintain clarity in the system and such system will be easier to maintain and extend.

3.2.4 Mediator

The role of the mediator object is crucial in the design as it will bring together independent parts of the system. As was noted earlier, it will act as a bus with a data storage capability.

It will also present a unified interface for all the parts of the system to communicate with. These parts (be it low-level modules, user interface widgets or anything else) will subscribe with the mediator in order receive data from it or push data to it. They will therefore act as clients while the mediator will act as a server. The clients will identify themselves and the kind of data they are interested in during the subscription. The server will deliver the data according to this information.

3.2.5 Command Line Interface

The system will feature a command line interface (CLI) which will be a suite of tools and utilities for accessing the data and controlling the system without any graphical interface. Thus, it will be targeted mainly on engineers and technicians that would be doing a maintenance of the system. The scope of this suite will therefore not be the same as the scope of the GUI. Its focus will be on accessing the system under conditions not allowing to run a graphical interface.

3.2.6 Graphical User Interface

The system will also feature a graphical user interface (GUI) that will be focused on the ease of user-computer interaction. Its purpose will be to allow the user to comfortably control or monitor the car functions while standing still or during the ride. Actual design and look of the interface will be discussed in more detail in the next chapter.

What's left to discuss here is the underlying architecture. Picture 3.3 shows its overview, with the main screen in the top left. The main screen will be composed of various elements (widgets) to show data acquired from the car systems (energy level of batteries, immediate

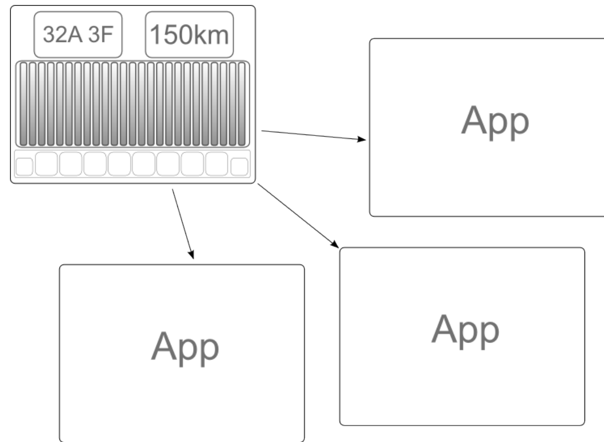


Figure 3.3: GUI Architecture Overview

power consumption, ...) as well as to provide means of control (battery charging, ...). It might also be possible to transition between multiple main screens to provide context sensitive information. An example might be one main screen for standing car and one for moving car. Each one of them will contain slightly different widgets.

It will also be possible to run applications from the main screen. Applications will be stand-alone tools for various purposes like programming the quick car charger, or 3rd party ones, like multimedia player or internet browser.

Chapter 4

Graphical User Interface Design

This chapter summarizes the requirements on the user interface and identifies a set of common tasks the user will perform. This will define a base for the user interface. The next sections then propose a design that will address the requirements and provide ways to perform the tasks.

The process of GUI design used here is based on recommendations and experience described in [30].

4.1 Users and Tasks

First of all, users, their behavior, needs and work patterns and the tasks they would need to do have to be identified. Answering the following questions (the questions are quoted from [30]) can help to do that.

"For whom is this software being designed? Who are the intended users? Who are the intended customers (not necessarily the users)?"

The primary users will be drivers of electric cars, but drivers of normal cars with proper electronics might use the software as well. This means the software should not distract the driver and it should be easily and quickly accessible.

The secondary users will be passengers that might use the software in a way that doesn't require them to keep their attention on the road.

Finally, engineers and technicians should have low-level access to the software so they can analyze logs and do maintenance.

"What is the software for? What activity is it intended to support? What problems will it help users solve? What value will it provide?"

The software will provide information about the car status (energy level in batteries, car range, power consumption and other information provided by the car electronics). It will also provide means to control some of the functions of the car like the ability to charge the car batteries, plan and track the route, play multimedia and so on.

It should not duplicate information and controls present on the dashboard but rather extend them. It should help the driver to plan the ride, keep it under control and enjoy it.

"What problems do the intended users have now? What do they like and dislike about the way they work now?"

The drivers have only limited way of getting information about devices introduced to the car with the new electric propulsion. Therefore, they will need a new way to get this information as they will need a new way to interact with it.

The engineers and technicians will also need their own way of communicating with the new devices and they may benefit from the functions designed for drivers.

“What are the skills and knowledge of the intended users? Are they motivated to learn? How? Are there different classes of users, with different skills, knowledge, and motivation?”

The drivers are proficient in driving the car and presumably are able to use the dashboard controls. It is also presumed that they have at least basic computer usage knowledge and they should be familiar with common car systems (like energy distribution in the car). But there will probably be a wide spectrum of drivers as anyone with driving license can drive the car.

The passengers may vary even more than the drivers as they doesn’t have to have any driving skills or car systems knowledge. Such users will require simple and understandable interface for the most common tasks like playing multimedia or browsing the web.

The engineers and technicians are proficient computer users with good knowledge of the car systems. It is therefore supposed that they will use low-level tools that don’t have unnecessary dependencies.

The drivers and technicians will certainly be motivated to learn the new software as they will need it to efficiently drive the car or maintain it. But the motivation might be lower for the passengers as it will be only an optional tool for them.

“What are the intended users’ preferred ways of working? How will the software fit into those ways? How will it change them?”

The drivers sit in a driver’s seat and control the car with steering wheel, pedals, shift lever and various controls positioned on the dashboard. Because of the drivers’ limited movement, every control has to be easily accessible from the seated position. It also has to be laid out in an ordered manner so that the controls that are often used together are grouped together.

The touch screen with the user interface has to fit into this layout in a way that the driver will be able to touch it by merely extending his hand. It also shouldn’t be in a position that will force the driver to keep the eyes off the road for too long. Moreover, the passenger should be able to easily access the touch screen as well. This leads to a position in the center of the car which many car manufacturers have already adopted.

The driver is used to look at the dashboard through his steering wheel to check his speed or fuel. While the touch screen will be a little bit more to the side it should offer the most important information in a way that will catch the driver’s attention immediately (but on the other hand it must not distract him).

Answers to these questions give an idea about who the users are and what they want to do and introduce the tasks they will need to do to achieve their goals. Tables 4.1 and 4.2 summarize all the tasks in a clear manner. Some of the tasks are pretty simple and straightforward while others are complex and consists of a number of partial operations. The complex ones will be described in more detail in section 4.3.

The following list presents ideas of how the system could be extended. These ideas won’t be part of this project but they were considered during the design and they would

Check energy level in batteries
Check estimated car range
Check immediate power consumption
Check energy coming from panels (if present)
Check rear camera video feed
Check GPS location
Check multimedia information
Check collision information
Log into the system

Table 4.1: Simple tasks

Set the car charger
Play multimedia
Plan routes
Find nearest or reachable charging points

Table 4.2: Complex tasks

be interesting to implement in the future.

- Night and thermo vision
- Satellite maps of weather forecast
- Range estimation according to terrain and aspects
- Support for payment for the charging at charging points
- Traffic analysis, RDS, radar
- Highway traffic cameras

4.2 Conceptual model

The conceptual model is a model that users build in their minds about the system. It describes their knowledge of the system from their high-level point of view. It contains tasks they can perform and objects they may manipulate with. Since it's a user's point of view, the model doesn't contain anything related to the underlying architecture that the user cannot interact with directly (like database backends or internal containers for graphical objects).

The conceptual model can be represented in the form of a matrix that maps user actions to individual objects the user may interact with. This matrix has objects listed in the leftmost column while the actions are listed in the topmost row. The matrix itself then shows which actions may be used on which objects with the 'x' symbol.

A matrix representing a good selection of objects and actions is generally a compact one where many of the tasks are applicable on most of the objects. An ideal conceptual model would be represented by a matrix where every action can be performed on every object.

	Observe	Route	Find CP	Play	Charge	Launch	Browse	Log
Rear camera	x							
Collision	x							
Range	x	x	x					
Consumption	x	x	x					
GPS location	x	x	x					
Map	x	x	x					
Chrg. points	x	x	x					
Song	x			x				
Movie	x			x				
Batteries	x				x			
Quick chrg.					x			
Charging					x			
Chrg. presets					x			
Menu						x		
Internet							x	
Login								x

Table 4.3: Objects/Actions Matrix

Observe	Observe the visual or sound representation of the data
Route	Plan a route
Find CP	Look for the nearest or reachable charging point
Play	Play a multimedia file
Charge	Set the charger to charge the car batteries
Launch	Launch an application
Browse	Browse the web
Log	Log into the system

Table 4.4: Actions description

That would mean that the users can learn a set of actions and then apply them on every object in the system they can interact with.

On the other hand, the worst possible model would be represented by a matrix where every object has associated different tasks. Such system would be very complex and the users would have to learn every task for every situation from the beginning. They wouldn't be able to use actions they already know for interaction with other objects.

The ideal models are practically impossible to achieve in reality so good, real-life models are often structured. There are groups of several actions that can be performed on several objects (there can also be a common actions that can be performed on all or almost all objects). These groups then describe a common parts of the model that can be used consistently.

This project's model represented by matrix 4.3 is an example of such design (table 4.4 offers a brief description of the actions). There are identifiable groups of objects/actions that offer a consistent interaction with a part of the system. These groups form the user interface as follows:

- Navigation application (4.7.5) – actions *Route* and *Find CP*, objects *Range*, *Consumption*, *GPS location*, *Map* and *Charging points*
- Multimedia player (4.7.6) – action *Play*, objects *Song* and *Movie*
- Charger application (4.7.3) – action *Charge*, objects *Batteries*, *Quick charging*, *Charging* and *Charging presets*
- Application menu (4.5.5) – action *Launch*, object *Menu*
- Internet browser (4.7.7) – action *Browse*, object *Internet*
- Login application (4.7.1) – action *Log*, object *Login*

There is also a common action, *Observe*, that can be performed on most of the objects. This action represents the user's need to check various car status data. Therefore, it makes sense to present this data in a uniform way. This is done by user interface screens described in section 4.6, rear camera application (4.7.4) and collision detection application (4.7.2).

4.3 Task scenarios

This section details the complex tasks described in table 4.2 in a form of task scenarios (or use cases). Trivial tasks like getting information about current energy level in batteries are not described here.

The purpose of task scenarios is to illustrate how the user will interact with system and how he will perform the given tasks. Based on this information, the GUI can be then designed in a way that will help the user to perform these tasks more effectively.

Number	0
Name	Charge the batteries
Description	User wants to set the car charger to charge the batteries.
Actors	User
Precondition	Power supply is connected to the car.
Basic flow	1. User launches the charger application. 2. Then, he sets the desired intensity, time and target energy level in the application. 3. User starts the charging by taping on the "start" button.
Alternate flow	1. User chooses one of the stored presets in the quick charging in main screen of the standing mode. 2. The charging starts with selected parameters.

Number	1
Name	Play a multimedia
Description	User wants to play a multimedia file.
Actors	User
Precondition	
Basic flow	1. User launches the multimedia player. 2. Then, he chooses the audio file he would like to play. 3. User starts the playing by taping on the "play" button.
Alternate flow	

Number	2
Name	Find the nearest or reachable charging point
Description	User wants to find the nearest or any reachable charging point to recharge car batteries.
Actors	User
Precondition	
Basic flow	1. User launches the navigation application. 2. The application shows his current location (based on GPS data) and the locations of charging points. 3. The charging points in range are highlighted. 4. User can then choose the charging point he likes.
Alternate flow	

Number	3
Name	Plan a route
Description	User wants to plan a route for his travel.
Actors	User
Precondition	
Basic flow	1. User launches the navigation application. 2. The application shows his current location (based on GPS data) and all other necessary navigation data. 3. User can then plan his route and he can do so according to the charging points, if he wishes it.
Alternate flow	

4.4 Touch Screen Usability

Designing user interfaces for touch screens differ in some ways from designing interfaces that use the usual input and output devices, like mouse, keyboard and the computer monitor. Therefore, some guidelines[23, 8] have to be considered to make the user interface well suited for a touch screen.

The users using a touch screen can interact with it only by taping the screen with a fingertip. This can be compared to a mouse with only one button. It means that context sensitive actions (that are usually done with the right mouse button) must be done by a different mechanism. A popular solution to this problem is to tap the given control element and hold the finger in contact with the screen for a short time. Then, a context sensitive menu may be opened for example. Another solution might be gestures. Gestures are stored movements of the fingertip across the screen. These movements might be mapped to different actions.

The left mouse button is frequently used for double-clicking. This mechanism might be used with touch screens as well but it poses certain problems. The user's fingertip will probably never remain in the same position during the double-tap so the mechanism has to be tolerant to this. The motion might also be less comfortable for the user in the first place.

Dragging, another common concept used with the mouse, might be and is used with touch screens. But the continuous movement of the fingertip across the screen might also be a little difficult in certain situations. The places where the dragging will be required should be therefore chosen well.

A popular mechanism for scrolling on touch screen devices is the kinetic scrolling. It scrolls the content even after the user's finger has left the screen slowly coming to stop. This makes it easy to scroll smaller or greater distances while the user isn't really fatigued.

Tapping the screen can't be used for any precise positioning and it also lacks the movement or pointing feedback. When the user taps the screen the given action is done. There is no other state. This behavior might pose some problems as the user might not be able to cancel the current operation.

It should also be taken into consideration that human fingertips can be quite large. The interface should be designed in a way that allows to safely control the interface for a wide variety of people. This means that the controls the user will tap (most commonly push buttons) should have a minimal size defined. The gaps between the controls should also be taken into account.

This, and the fact that touch screens doesn't usually feature large resolutions (common are 640x480 and 800x600) means that the application layouts should work well with screen space. It also helps when the application can run in fullscreen mode. Although it may not always be possible due to the space restrictions, it is good to keep the logically equivalent controls in roughly the same positions across all the applications. It helps the user to effectively navigate through them and lowers the resulting fatigue.

Generally, it could be summarized that touch screens are best suited for pointing or selecting actions. This means that push buttons that the user may tap will probably be the major control element. On the other hand, the worst action for this type of input device is text input. This can be solved by providing a software keyboard. The keyboard might be only numeric (usually arranged as a keypad) or alphanumeric. The numeric only keyboards have the advantage of smaller size so the buttons might be bigger. The alphanumeric keyboard might be arranged as full QWERTY (emulating hardware keyboards) or as a keypad as well. In this case, each buttons contains several letters and the button may have to be tapped several times to get to the desired letter. This principle is known mainly from mobile phones.

As the touch screen is controlled by fingers, the users will naturally want to use their left or right hands based on their preference. The user interface should therefore be designed in a way that it will be the same for left handed and right handed people or it should be possible to switch it from one mode to the other.

Basically, user interfaces that use touch screens are best suited for applications that have a low frequency of use and where accurate positioning is not required.

4.5 Widgets

This section details widgets (GUI elements) that will be later used to construct the user interface screens. They are presented as schematics at this point so that they are not tied to any concrete graphical representation yet.

4.5.1 Car Range

This widget is a simple text display that shows estimated car range. This range is computed based on actual energy levels in batteries and expected maximal range. This value serves for orientation purposes only as it might not reflect the situation accurately.

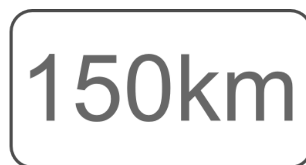


Figure 4.1: Car Range

4.5.2 Immediate Consumption

This is another display widget that shows immediate power consumption in kWh/100km. The power consumption is computed from the immediate measurements of batteries' energy consumption.

Figure 4.2: Immediate Consumption

4.5.3 Batteries' Energy Levels

One of the main additions to the car will be batteries that will power it. The driver will therefore need to know the energy levels in the batteries. The bars representing each battery will use following colors to show different levels for better clarity and to catch the driver's attention more easily in case of low power.

100%-65%	green
65%-30%	yellow
30%-15%	red
15%-0%	dark red

Table 4.5: Battery States

The car may contain a variable number of batteries therefore the widget will detect this number and display the bars representing them appropriately. If the number will be bigger than the available space on the screen then only every second battery will be visualized (or every third and so on) averaging the two together.

4.5.4 Quick Charging

This widget is actually a button that offers a quick way to begin charging the car batteries. It shows last used charging preset (it's one of the presets created and managed in the Charger application, described in section 4.7.3) and when the user taps this button the car will begin charging the batteries. This widget serves only as a quick start, any changes to the charging process will need to be done from the Charger application.

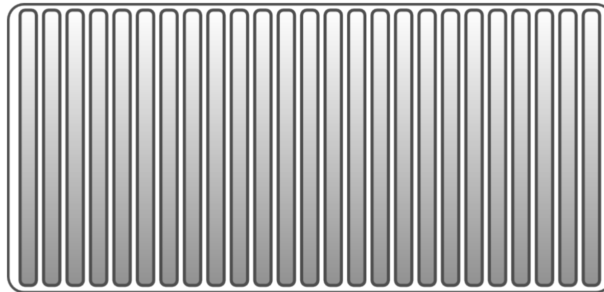


Figure 4.3: Batteries' Energy Levels

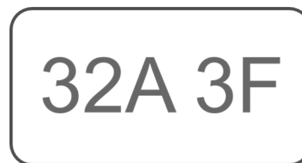


Figure 4.4: Quick Charging

4.5.5 Menu

The menu serves as a tool to launch external applications as well as switching between them. It will be made of buttons laid out in a horizontal pattern and since there is only limited number of applications to run, all the buttons representing the applications will be visible at once. It will also make it easier for the users to maintain an overview of all the applications they can run. Controls arranged in this way will be in reach by both the driver and the passenger. It also make the user interface independent on what side the steering wheel is positioned on.

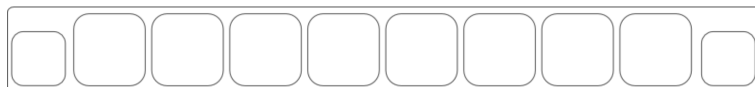


Figure 4.5: Menu

There will also be two special buttons (one on each side). One will be responsible for shutting the system down while the other will bring the software keyboard on the screen. These buttons are smaller because one might be potentially dangerous to tap by mistake while the other serves a lesser role. To protect the user from accidentally tap the shutdown button and shut the system down the button will bring up a menu of shut down options. This means the user must tap twice on a selected button to actually shut the system down. This is a common usability pattern[1].

4.5.6 On Screen Keyboard

The on screen keyboard might not be a true widget but rather a tool that will be available for the users whenever they will need to type some input. Once launched, it will slide onto

the screen rearranging other (application) windows so that it will not overlap them. When closed, the windows will retake their original geometry.

4.6 Screens

The whole GUI will operate in two modes to reflect different needs in different situations. Therefore, there will be two main screens, one for car standing still and the other for a moving car. The main screen will be the first thing the user will be presented with after the start of the system. It will serve as an overview of the car status showing informational data. In a way, it will be similar to the part of dashboard that the driver can see through steering wheel.

4.6.1 Standing Car

The users in a standing car will be interested in information related to charging, batteries status and perhaps route planning or media playback. Therefore, the screen layout will reflect this by offering widgets focusing on this data. The layout is shown in picture 4.6.

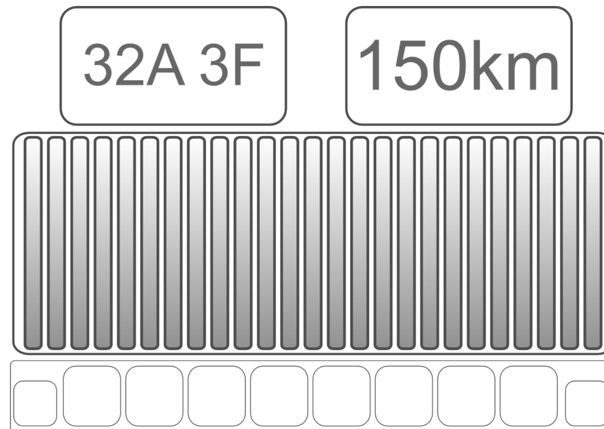


Figure 4.6: Standing Car Layout

The menu is positioned along the bottom line to be equally accessible by the driver and the passenger. Since the horizontal edge of the touch screen is longer than the vertical it also offers more space for the application buttons.

The batteries widget dominates the whole layout as it shows a complex information about the energy status. It will also be the main point to attract the driver's attention when the energy level will be too low. At the top are the widgets for quick charging and estimated car range. Their simple textual information is easily readable even when the controls are smaller in size. But they still have to be large enough to comfortably tap the quick charging button.

4.6.2 Moving Car

The users in a moving car will be more interested in current power consumption and remaining energy which is reflected by layout shown in picture 4.7. Overall, the layout is

similar to the previous one. The main difference is that instead of quick charging options, it shows immediate power consumption.

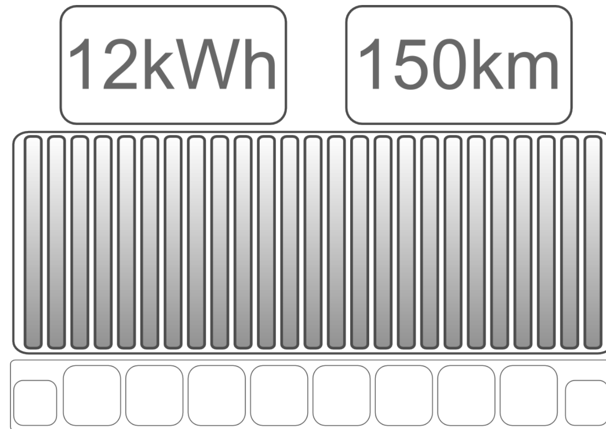


Figure 4.7: Main Screen Schematic for a Moving Car

This layout follows the same rules as the standing car layout, but there should be some logical differences in function in this mode. For example, the playing of videos and other potentially distracting actions should be disabled in this mode.

4.7 Applications

Applications are used to group together controls that are logically tied among themselves but otherwise stand apart the screens described in the previous section. Some of the applications will be part of this project, while others will be 3rd party (these will be pointed out). In the future, it should also be possible for the user to install new applications or remove the old ones. It should also be kept in mind that 3rd party applications that are to be run on the touch screen might need alteration of their user interface.

4.7.1 Login

The login application (picture 4.8) serves as a security solution for the software. It will require a correct password to let the user log into the system. Because typing the password on a touch screen is not practical a different way is chosen. The screen will be divided to colored fields and the user will have to pick four colors as the password.

The user will also be able to change the password here. This will be done by choosing the old password first and then twice choosing the new password. This is a common usability pattern[1] that will make sure that the user really chose the password he or she wanted.

4.7.2 Collision Detection

This application (shown in picture 4.9) will show a top view of the car and its surroundings. There will be zones marked around the car that will represent the positions of the collision sensors. The sensors will be feeding the application with information about the distance to nearest obstacle in the given direction. The application will then convert this information to graphical representation showing green indicator for clear road, yellow one for an object

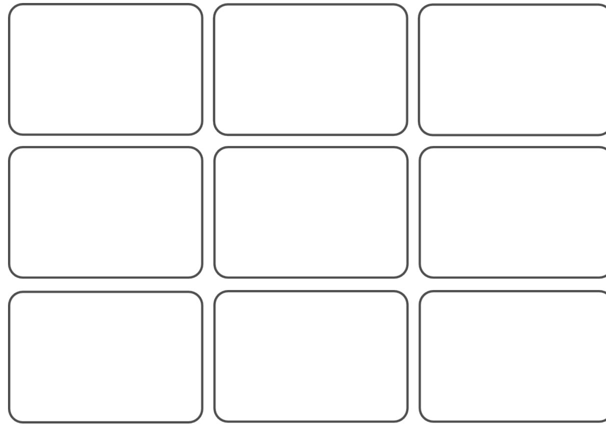


Figure 4.8: Login Application

that is getting closer to the car and a red one when the object will be critically close to the car.

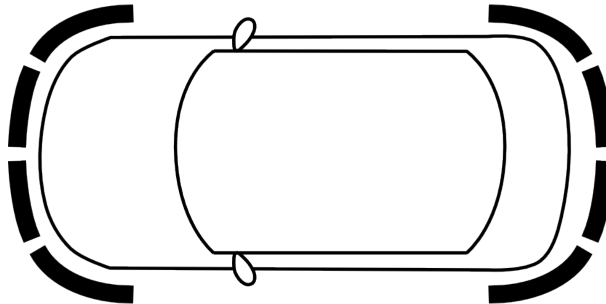


Figure 4.9: Collision Application

4.7.3 Charger

Picture 4.10 shows a schematic of the charger application. This application will be used to program the car quick charger. Follows a list of requirements:

- It should offer options for intensity (speed) of charging and time programmer
- It should detect whether the power supply is plugged in
- It should show progress of charging by animating the battery bars
- User should be able to store his or her frequently used charging settings to be able to use it again easily
- User should be able to start and stop the charging

As can be seen from the picture, the application will allow the user to pick a time of charging, thus allowing him or her to limit the charging, for example, for a night. The user will also be able to store his settings in the form of presets, that he can use again easily. When the charging has started the "Start" button will change to "Stop".

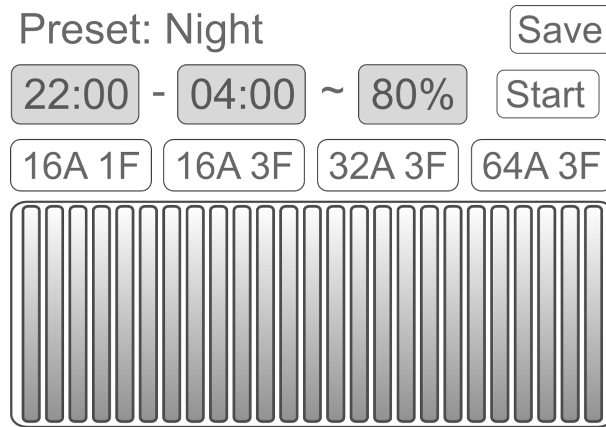


Figure 4.10: Charger Application

4.7.4 Rear Camera

This will be a simple application presenting video feed from the rear camera. It may give the driver better overview of what is behind the car then the rear mirrors.

4.7.5 Navigation

The navigation application will be utilizing some of the existing 3rd party navigation applications and therefore its design will be dependent on the application itself. It is supposed though that a skin will be created that will let the application to suit the whole project.

It will also be extended with the map of charging points in Czech republic. The application should then track the car with a GPS module, indicate nearby or reachable charging points, show available car range and allow the usual route planning.

4.7.6 Multimedia Player

This software will utilize one of the common 3rd party multimedia players. Therefore, the design of the application will be left to the application itself. Again, it is supposed that it will be modified with a skin to suit the rest of the applications better. The skin should also take into account that it will run on a touch screen so it should adhere to the rules described in 4.4.

The player will be able to play only music, not video files, because watching movies could potentially distract the driver.

4.7.7 Internet Browser

Internet browser is another typical candidate for a 3rd party software. There are some solutions targeted specifically at touch screens because of the recent boom of mobile devices. As the previous two applications, it should be skinned to fit the project.

4.8 Colors and Ergonomics

It should be noted that the readability of the touch screen will depend heavily on light conditions in the car. The parameters of the screen, like brightness and contrast, will also influence that. To help make the readability better, some tips[23], [30] should be considered. The colors used in the user interface will have an impact on the ability to read its content and well chosen colors may help make the interface readable even in poor light conditions.

Therefore, brighter colors should be used generally (mainly for backgrounds and larger areas), but then again, these can daze the driver in the night. The optimal solution would be to automatically choose the color scheme based on the time of day, or rather light conditions in the car. But that would require some kind of light sensor in the car. For the sake of this project, the user interface will be skinnable so the user may choose a color representation he likes the most.

While the touch screen interface offers a very flexible tool for creating various controls, the size of the screen is limited. Therefore it would be best not to include controls for common operations, like return to previous screen or return to the main screen. While this can be solved by using gesture controls, an interesting alternative might be to utilize external buttons for these operations. But different cars might have different number of external buttons available (or none at all) so the user interface will contain all the necessary controls itself.

4.9 Final User Interface

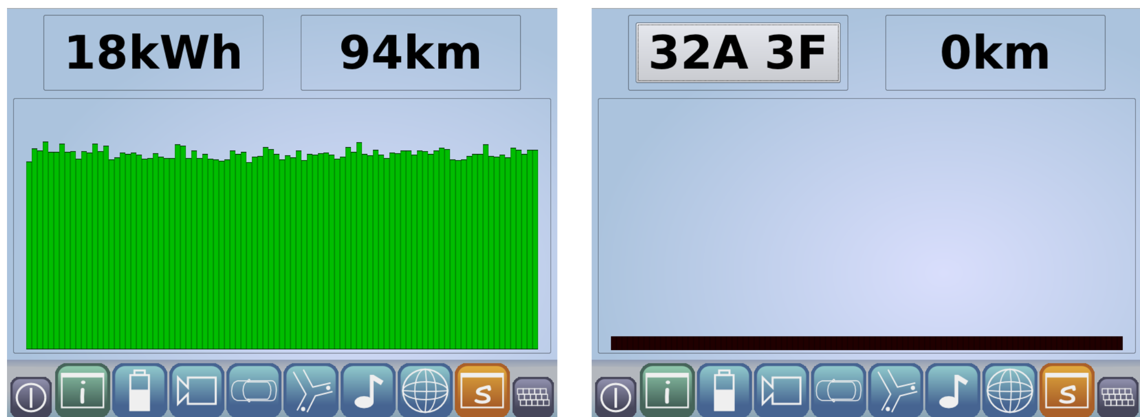


Figure 4.11: Main Screens Screenshots

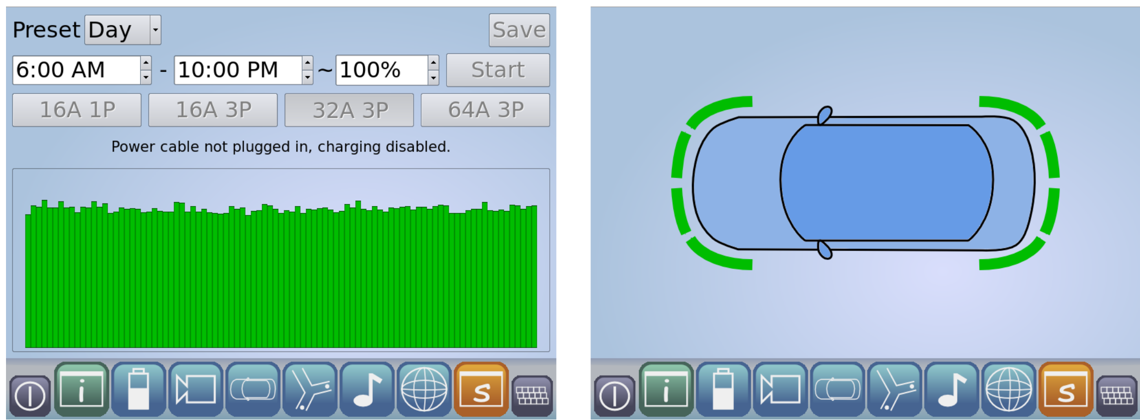


Figure 4.12: Screenshots of Charge and Collision Applications

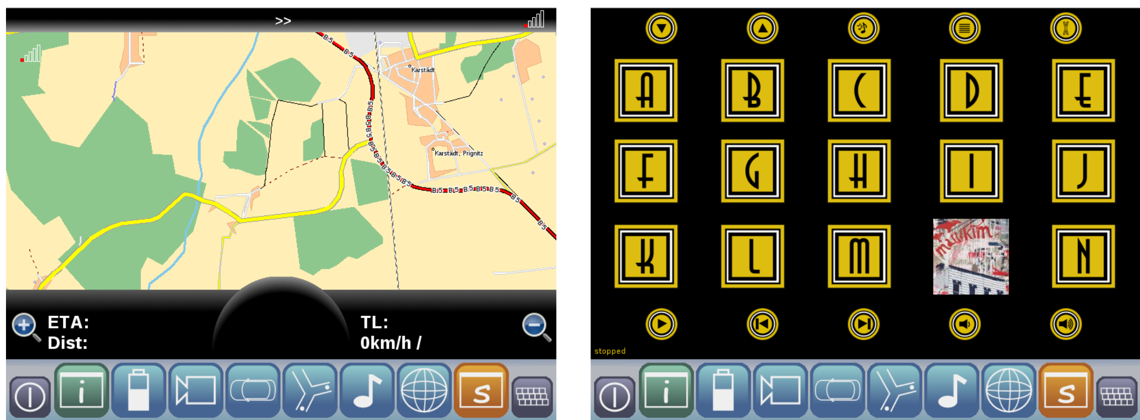


Figure 4.13: Screenshots of Navigation and Music Player Applications

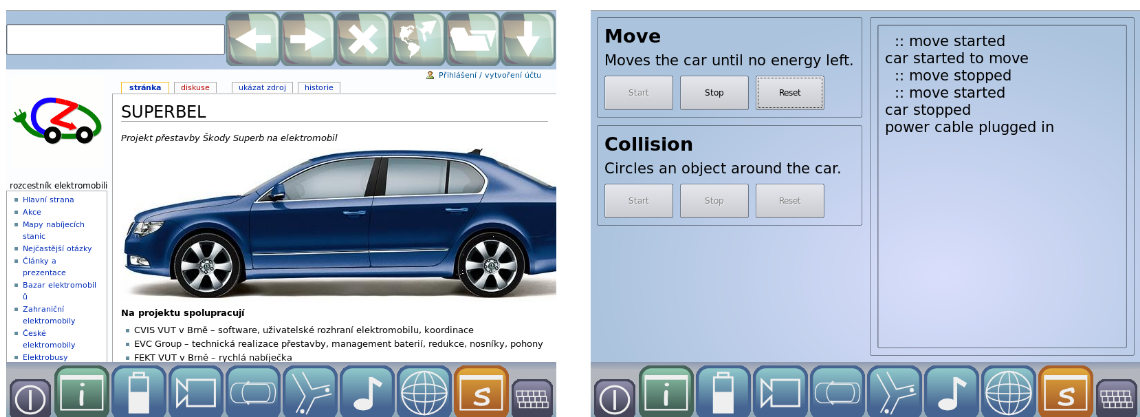


Figure 4.14: Screenshots of Web Browser and Simulation Application

Chapter 5

Implementation

This chapter describes the way the project was implemented. It presents the platform that was chosen to build upon, the modifications that were needed to be done and problems encountered during the process. Then, it presents the applications and modules that were created from scratch. It also shows how this implementation conforms to the abstract architecture build in section 3.2.

5.1 Overview

Picture 5.1 shows the high-level overview of the project implementation. It is build on top of LinuxICE, a Linux distribution specialized for car computers. This distribution offers a few applications to give the user the ability to run other applications and manage them. These applications form the foundation of the project. Then, a number of user-centered applications and modules is present so the user can actually monitor and control the car functions. All of the applications were skinned and configured to fit together nicely.

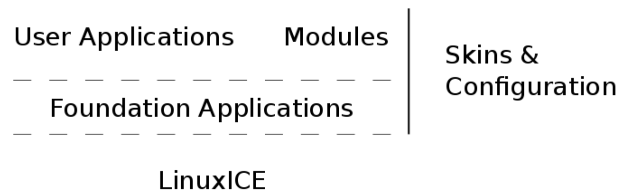


Figure 5.1: Implementation Overview

5.2 LinuxICE

As was indicated in the previous section, LinuxICE[10] (version 2.0.1 was used) is an open source Linux distribution specialized for car computers. It is based on Ubuntu 9.04 and contains several aspects that make it ideal as a car computer operating system like touch screen drivers, GPS support or bluetooth support.

LinuxICE was created according to OpenICE platform[17] which proposes that modern vehicle information and entertainment applications should include following four core features: “media playback, navigation, vehicle information/statistics and handsfree/mobile device integration”.

OpenICE offers three applications that create a layer between the user and the system. These applications are nGhost, Icepanel and Matchbox. They are used to allow the user to run any other applications, manage these applications or play multimedia. Therefore, these applications are targeted to be used as a foundation for any project built on top of the OpenICE platform.

LinuxICE wraps all this together on top of an existing desktop Linux distribution and adds some common, touch screen friendly applications like web browser or navigation. It therefore serves as an ideal base for projects that have the ambition to run in car computers. Another advantage is an active community that keeps the project under development and continually adds new features as well as fixes bugs.

5.2.1 nGhost

nGhost2[15] was considered for use in this thesis as it is an integral part of OpenICE. It as an open source application that primarily serves as a media center but it is fairly customizable and can be used to show or interact with various car related information. It uses skins described in XML that can also define a fair deal of functionality like inter-process communication or usage of events. It's API also makes it possible to write plugins to bring in completely new functionality.

But in the end, nGhost was not used in this thesis. The main reason was that I was unable to modify it so it would conform to the needs of the thesis. This was probably due to version mismatch between the binary libraries used in LinuxICE and the newly compiled nGhost. Furthermore, the advanced features of nGhost wouldn't be used in the end so it was not a big loss. New applications were created instead that offer new flexibility and suits the thesis perfectly. These applications will be described in section 5.5.

It should also be mentioned that new version of nGhost is being developed, as described in the appendix A.5.

5.2.2 Icepanel

Icepanel[7] is an open source application that sits docked at the edge of the screen a provides various control or information elements. It can contain buttons to run other applications, show current time or manage the playback of multimedia. As it is based on the same library as nGhost, it also has means for inter-process communication using Unix sockets or D-Bus.

This application was used to implement the menu described in 4.5.5. A new skin was created that features custom graphics and buttons to launch applications. These buttons actually launch scripts that take care of launching the applications as will be described in 5.5.

5.2.3 Matchbox

LinuxICE uses Matchbox[11] as a window manager. It is an open source manager primarily designed for embedded devices. That means that it stacks the application windows on top of each other while each window takes a size that covers the whole screen (except for any dockable panel windows). This behavior is crucial for embedded devices as their displays are very limited in size.

The window manager can be controlled with keyboard shortcuts (apart from the standard control by mouse or fingers in case of touch screens) which is very useful when debugging or in the development process. This means that the user interface can stay the same

as it should be in the final release while for example a terminal emulator can be launched with a keyboard shortcut.

5.3 Core Implementation

This section describes the implementation of applications, modules (and their interaction) that form the core of the project. It is these applications that gives users the ability to control the car functions or monitor them, play multimedia or browse the web. Picture 5.2 shows an overview of this implementation.

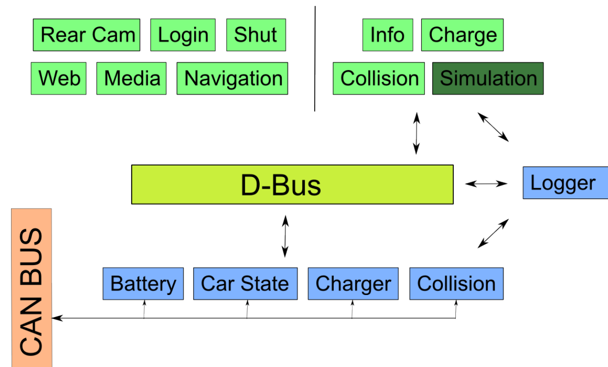


Figure 5.2: Core Implementation

As can be seen from the picture, this implementation is modular and conforms nicely to the abstract architecture described in 3.2.

The base of the structure forms the D-Bus[3] that serves as a mediator between different parts of the system. D-Bus is an inter-process communication interface created by *freedesktop.org* which is fairly widespread in use by desktop applications and environments. It allows the applications to send signals through the bus to other applications or to call their methods remotely. This way is ideal to keep loose coupling between different parts of the system which is desirable to maintain its scalability[32].

In the bottom resides low-level modules that communicate with CAN bus. They prepare data received from it for the applications. The logger is technically a module as well but it also offers an interface that any application can use directly in case it needs to log some events, like errors or important messages.

The CAN bus is represented by another module that is responsible for direct communication with the bus. Currently, this module is a fake one that actually communicates with the simulation application. This module should be replaced by the one that can really interact with the CAN bus. More information about this will be presented in section 6.3.

In the top resides applications that the user can use directly. They can be divided in two groups. Applications from the first group doesn't use D-Bus to communicate with the low-level modules. These applications have their own means of getting information, like playing songs stored on the hard drive. The other group interacts with the modules in a way that will be described in the next section.

5.4 D-Bus Interface

As all the applications and modules created in this project are written in Python, the D-Bus is also accessed through Python bindings[4]. The applications and modules communicate together by sending signals through the bus. The signal may contain additional data. Following code snippet shows how to construct a class that will do just that.

```
class ChargeService(dbus.service.Object):
  def __init__(self):
    bus_name = dbus.service.BusName("org.superbel.charger",
                                    bus=dbus.SessionBus())
    dbus.service.Object.__init__(self, bus_name, "/charger")

  @dbus.service.signal("org.superbel.charger")
  def charge_intensity(self, intensity):
    return
```

This snippet is taken from the charge application and shows a class that defines a D-Bus service that will be able to send a signal through the bus. To send the signal one only needs to create an instance of this class and call its `charge_intensity` method. The application using the service is usually polling for data and then sending it in a loop with given time interval.

What's important to notice here are the strings highlighted in brown. These strings form a unique identifier of each interface and signal that goes through the bus. The resulting identifier of the signal composed from the string in the example would then be: "`org.superbel.charger/charger/charge_intensity`".

The signal has one argument defined (apart from the `self` keyword). This argument is used to transfer user data through the bus to the receiving application. The next example shows the receiving side.

```
class Charger(QtGui.QMainWindow):
  def __init__(self, parent=None):
    QtGui.QWidget.__init__(self, parent)

    bus = dbus.SessionBus()
    bus.add_signal_receiver(self.charge_intensity, "charge_intensity",
                            "org.superbel.charger")

  def charge_intensity(self, intensity):
    self.canbus.setChargeIntensity(intensity)
```

This snippet is taken from the charger module which communicates with the charging application. First, a session bus object is created. A handler that processes the signal is registered afterwards. Appropriate unique identifier has to be used to register the handler to a specific signal. In this example, the handler is tied to the signal defined in the previous example. When the handler receives the signal it extracts its data and passes it to the CAN bus.

Asynchronous receiving and sending of the signals requires a D-Bus aware main application loop. This can be easily solved in a Qt application using the following code (again,

taken from the charger module).

```
if __name__ == "__main__":
  dbus.mainloop.qt.DBusQtMainLoop(set_as_default=True)

  app = QtGui.QApplication(sys.argv)
  charger = Charger()
  sys.exit(app.exec_())
```

It activates the D-Bus aware main loop, creates the application and then enters the loop. This is the basic principle that is used in all the applications and modules that uses D-Bus for inter-process communication.

Table 5.1 describes all the D-Bus interfaces used in this project along with their signals and a short description.

Interface	Signal	Description
org.superbel.charger	charge_intensity	Sets the charging intensity
	charge_start	Starts charging
	charge_stop	Stops charging
	charge_max	Sets the charging maximum
	cable_present	Is the power cable plugged in?
org.superbel.battery	num_batteries	Sets the number of batteries
	battery_levels	Sets the battery levels
	battery_consumption	Sets the immediate consumption
org.superbel.car_state	car_state	Is the car moving or still?
org.superbel.collision	distance	Distance from the car in each dir.
org.superbel.logger	log	Sends a message to log
org.superbel.simulation	num_batteries	
	battery_levels	
	battery_consumption	
	car_state	
	cable_present	
	collision	
	charge_intensity	
	charge_start	
	charge_stop	
	charge_max	

Table 5.1: D-Bus interfaces

Simulation signals follows the same pattern and usage as their counterparts described above. These signals are used to run the simulation and should not be present when the system is implemented into the car.

5.5 Applications

The applications that were created from scratch (they will be noted in the text) were implemented using Python[19, 5] and Qt[20] (utilizing Python bindings PyQt[18]). Python

is a high-level programming language that features among others a dynamic type system and automatic memory management. Qt is an object oriented GUI framework that provides means to create applications easily and comfortably. Combined, these technologies are great for rapid development of either console based or GUI applications. This makes for the loss of not using nGhost as developing new applications was easy and straightforward.

Another advantage of writing custom applications instead of using nGhost was the gain of greater flexibility in UI layouts. The layouts are internally managed by Qt and scale well for example for changing resolutions. The UI design process itself was also easier as Qt offers and advanced tool for this purpose (Qt Designer) as opposed to manually writing XML based skins for nGhost. On the other hand, nGhost skins offered a great deal of functionality tweaking without the need to recompile the application. But then again, the new applications are written in Python which makes it possible to change their functionality without recompiling as well.

The applications can be found in `src/apps`.

5.5.1 Run Scripts

As was mentioned earlier, the applications are not launched directly but rather by invoking a launch script. This script (implemented in Bash) checks whether the application is running or not and if not it runs it. If it was running already it brings it forward so the user may interact with it. This means that switching applications is quicker because they don't have to be killed and launched repeatedly.

A typical run script looks like this:

```
APP="charge"
RUN=`wmctrl -l | grep -i $APP`

if [ -z "$RUN" ]; then
  /home/superbel/apps/charge/charge.py &
else
  wmctrl -a $APP
fi
```

But this solution has it's drawbacks as it means that all the applications may run simultaneously which may slow down the system.

5.5.2 Login

The login application is launched early in the process of establishing X session thus preventing it to go any further until the correct password is given. The password is encoded from the colors the user chooses in the following way:

1. The color is converted to its string representation
2. The strings are concatenated together (for example *"redgreenblueblack"*)
3. An MD5 hash is computed for the resulting string

Then, the computed hash is compared with the stored password (also in the form of MD5 hash) and if the passwords are equal the user may log in.

The potential weakness might be that MD5 is not collision resistant but the point of this application was not to create a bulletproof security solution. After all, the software it secures doesn't control any critical functions.

5.5.3 Shutdown

This application simply calls utilities provided by the system to perform the necessary shutting actions:

- reboot - `reboot`
- shutdown - `poweroff`
- suspend - `pm-suspend`
- hibernate - `pm-hibernate`

5.5.4 Main Screen Information

This application shows information about batteries present in the car, about its immediate consumption or car range. It does so by listening to signals from battery and car_state modules and then processing the acquired information.

The number of batteries, their levels and power consumption are acquired directly from the data attached to the signals. The estimated car range is computed from the current energy levels in batteries and expected range that was set to 120km.

The application also changes the information shown based on the information received from the car_state module. It shows slightly different data when the car is moving and when it is stopped.

A new widget for displaying battery bars was created by inheriting from `QGroupBox`. This widget adapts its geometry to the surrounding layout. It also adapts its graphical representation to the number of batteries it gets information about.

5.5.5 Battery Charging

The charge application is designed to plan the battery charging. Users can just start the charging or they can plan it for a specific time with a specific maximal energy level. This might by useful to quickly charge the batteries as the first two thirds (or a similar percentage) usually charge much quicker than the rest. There are also stored two presets of the charging settings so the user can configure the charging quickly. The natural expansion of this application would then allow users to store their own presets.

The state of charging is shown in the status bar in the middle which also tells the user if the power cable is plugged in or not. In the bottom area is the same battery widget that was created in the previous application to show the overall battery status.

This application communicates two way with the charger module to get information about the power cable status and to issue charging commands. Is also gets information from the battery module about the status of the batteries.

This application also uses an interesting feature of Python called *pickling* to store and load the presets. It's basically a way to serialize given object to a stream of data. This data can then be sent to a file or for example over a network. The data can also be read

back and the corresponding objects will be automatically created by Python's pickle object. Following code snippets show how this is done in the application.

```
def loadPresets(self):
  try:
    f = open(os.path.expanduser("~/apps/charge/presets.pcl"), "rb")

    self.presets = pickle.load(f)

    f.close()
  except IOError as (errno, strerror):
    print("Preset file load error:\n\t\t%d - %s" % (errno, strerror))
  except pickle.PickleError:
    print("Error while pickling presets from file")
```

```
def savePresets(self):
  prs = self.presetsTemplate()

  try:
    f = open(os.path.expanduser("~/apps/charge/presets.pcl"), "wb")

    pickle.dump(prs, f)

    f.close()
  except IOError as (errno, strerror):
    print("Preset file save error:\n\t\t%d - %s" % (errno, strerror))
  except pickle.PickleError:
    print("Error while pickling presets to file")
```

The presets object used in the examples is a list of dictionaries (a dictionary is basically an associative map). Each dictionary holds information about one preset. The whole list is serialized to the file or read from it with the simple code shown above.

The time planning is activated only when the start and stop time are different. That means that if the user wants to charge independently on the time he just sets the times to the same value. The changes to these times are polled every 6 seconds so it might take a little while until the application picks them but since the resolution of the times is 1 minute it's not a problem.

If the users has trouble to tap the arrows in time spin boxes (as they may be small for some people) they may always double-tap the time itself and change it directly.

5.5.6 Rear Camera

The application that shows video feed from the rear camera was implemented using OpenCV 2.1[16]. OpenCV is a computer vision library that among other things wraps camera management and also provides some basic GUI capability. It was therefore used to create the window, create the camera capture object and then to present the images from the camera in the window. As OpenCV takes care of all the low-level work, it takes only a few lines of code to create such application.

5.5.7 Collision Detection

This application is used to monitor the car surroundings and watch for any potential collisions with other objects. It does so by showing the distance at certain points around the car provided by the counterpart collision module.

When the distance becomes critical a corresponding indicator turns red. A natural expansion of this application would be to provide an acoustic indication as well, for example in the form of beeping.

5.5.8 Music Player

A 3rd party open source application called JukX[9] was chosen as a music player. It uses XMMS2 as a backend which must be installed and running for the application to work. It is designed for use on touch screens and while it is still in development it works pretty well. The only drawback right now is that the list of albums must be regenerated manually through XMMS in case it changed.

A three free songs[1][2][3] were added to the player for testing purposes.

5.5.9 Navigation

A 3rd party open source application called Navit[14] is used to take care of car navigation. This application was already part of LinuxICE and was a primary choice for navigation. It is probably the most advanced open source navigation application available today.

5.5.10 Web Browser

The final 3rd party application (open source as well) called Carbon is used as a web browser. It is a very lightweight application based on WebKit designed to work well on touch screens. It allows the user to simply browse the web and features a kinetic scrolling. A kinetic scrolling is a type of scrolling popular on mobile devices where the screen contents keep scrolling even after the user has lifted his finger from the display. The scrolling then slowly stops.

5.5.11 Simulation

This is a special application that allows the user to run simulations of the system. It serves as a testing tool and shouldn't be needed when the project is implemented into the car.

The simulation application communicates through D-Bus with the fake CAN bus module. It basically simulates whatever data would be coming through the CAN bus while the bus module serves as its frontend. A system implemented in this way maintains the necessary loose coupling needed for simple replacement with the real CAN bus module.

When the project is implemented into the car all that is needed is to create the real CAN bus module. This module would feed the other modules with live data from the car and would also send commands from the applications through the bus. The simulation application would then be naturally cut out of the system.

The simulation application itself works in the following manner. It creates an object from a class describing a specific simulation. It then periodically updates this simulation

[1] http://music2ten.com/2010/05/18/plank-arse-nick-mp3/

[2] http://music2ten.com/2010/04/21/the-most-serene-republic-heavens-to-purgatory-mp3/

[3] http://music2ten.com/2010/05/07/matt-kim-no-more-long-years-mp3/

object as well as manage the simulation as a whole, like pausing it, stopping it or resetting it.

The simulation application expects the specific objects to have a minimal uniform interface. This interface should look like this:

```
class ConcreteSimulation():
  def __init__(self, service, label):
    # initialization

  def reset(self):
    # reset the simulation

  def update(self):
    # generate simulation data and send the signals
```

The `reset` method is called to put the simulation into the starting state so it can be run anew.

The `update` method is periodically called by the main simulation application and is responsible for updating or generating the simulation data. When the data is prepared it is sent through the D-Bus to the CAN bus module. The concrete simulation object can also listen to incoming signals thus allowing a two way communication with the environment.

There are two simulations present at this time. The first one drives the car until it runs out of energy. The car then stops and a power cable is plugged in so the car batteries might be charged. This simulates arrival at a charging point. The second simulation circles an object close to the car. This simulation demonstrates the collision detection system.

5.6 Modules

The modules implement low-level functionality that takes care of presenting the data from the car (acquired through the CAN bus) to the applications. They also allow to control certain car systems by sending commands through the bus.

Each module is implemented as a Qt application without any window or other GUI elements. They basically run as a background daemons. The Qt framework is used here to simplify interaction with D-Bus as was described in 5.4.

The modules can be found in `src/modules`.

5.6.1 Run Scripts

Two scripts were created (in Bash) to simplify the process of running and killing the modules. These scripts reside in the modules directory and basically start all the modules or kill them.

5.6.2 Battery

This module is responsible for extracting the information from the CAN bus related to the car batteries. After it gets the required information it sends it through the D-Bus with the identifier `org.superbel.battery`.

5.6.3 Car_state

The car_state module asks for the state of the car. The state might be either *moving* or *standing*. The module then sends the information about the state through the D-Bus with the identifier `org.superbel.car_state`.

5.6.4 Charger

This module communicates with the CAN bus in two ways. It polls the bus for information about the state of power cable connection and it also listens to signals coming through the D-Bus. It can then issue charging commands based on these signals.

The charger module communication is identified with `org.superbel.charger`.

5.6.5 Collision

The collision module polls the CAN bus for information coming from the collision detection sensors. These sensors collect information about distance in several points around the car. The module then sends this information with identifier `org.superbel.collision`.

5.6.6 Canbus

The canbus module is actually a fake one. Normally this module would wrap the necessary functionality required for other modules to get data from or put data to the CAN bus.

Currently, this module communicates with the simulation application and serves as its frontend (or interface). This module exposes an interface that other modules may use directly while it talks with the simulation application through D-Bus with the identifier `org.suprebel.simulation`.

5.6.7 Logger

The logger consists of two parts. The first part is a daemon that runs in the background and listens to signals sent through the D-Bus. It can then log information about this signals (and about the data attached to the signals) to a file.

One file is created for one day where each file is named with current date. Each log entry is stamped with the time when the event occurred. Then the event level (or severity) follows and in the end the event itself is described. The event severity may be one of the following values:

- **SIG** - an `org.superbel.*` signal was delivered through the bus
- **ERR** - an error has occurred
- **DBG** - messages used for debugging purposes
- **MSG** - informative messages with lower severity

The logger can be set to log only certain levels so the logs contain only interesting data. Logging with SIG level is currently not used as the signals are coming very quickly and the information in the log files becomes hard to read. Furthermore, the file sizes may grow very big in a short time. As an extension, a solution to this problem would be some kind of

intelligent logging that would for example say that certain event occurred X times during a certain period of time.

The second part of the logger is an interface that any application may use directly. This interface basically simplifies the logging process for the application because it takes care of sending the logging signals. Following code snippet illustrates the interface:

```
class LogMod():
  def logErr(self, text):
    # send an error logging signal

  def logDbg(self, text):
    # send a debug logging signal

  def logMsg(self, text):
    # send a common message logging signal
```

5.7 Skins and Configuration

LinuxICE foundation applications, Matchbox and Icepanel, were created in a way that they can be easily modified graphically as well as functionally. As the representation of these applications is specific to their use, it was necessary to create skins for them and to alter their configuration. Skins form the graphical representation of the given application and are made of bitmap images and textual information that gives these images proper position. Skins also contain information about the functionality of the control elements they are made of. For example, it tells what action should be done when the user taps certain button. Configuration files then define a common behavior of the application, like setting preferred window dimensions.

This applies to the user applications, web browser and music player, as well. It is a way to modify the applications to a certain degree without the need to recompile them.

Directory `src/skins` contains the aforementioned skins and configurations files.

5.8 LiveDVD Creation

The result of this project is delivered as LiveDVD. It is a Linux distribution packed with everything described in this project and ready to be run directly from the DVD. This makes it easy to test or try the project without complicate or long installation. Even better, it can be run in a virtual machine to see what it can do.

But it is also possible to install it from the LiveDVD to hard drive. This has the advantage of quicker and smoother run and lower demands on system memory.

This section describes the process of creating this LiveDVD which is greatly simplified by the installation script written for this purpose.

5.8.1 Install Script

The installation script is written in Python and operates in three modes. The modes are described in the script help that can be acquired by executing it without any parameters (or with wrong ones).

```
Syntax:
  superbel.py update PATH  - updates system files (doesn't install
                             any packages)
  superbel.py install PATH - installs from PATH to current machine
  superbel.py livedvd PATH - installs from PATH to current machine
                             and creates liveDVD

Example:
  superbel.py install /mnt/dvd/superbel
  superbel.py install user@host:~/superbel
```

The script is supposed to be run from the destination machine (machine where the project will be installed into). The machine should be a properly installed LinuxICE with the user named as "superbel". It should also have automatic login enabled (these steps will be described in the next part in more detail).

The script then copies everything from the given path to a temporary directory located in the user's home directory. Then, depending on the mode chosen, it either only updates installed applications and modules (used mainly in development) or installs the whole project or installs the whole project and creates a LiveDVD. Afterwards, the temporary directory is removed.

All necessary packages are installed during the installation from the Ubuntu repositories except for two, OpenCV 2.1 and JukX. These packages were not available in the repositories and therefore were built from sources, packed and included in the project. They are installed separately from the others.

The installation script is located in `src/bin`.

5.8.2 Creating the Image

A detailed steps will be provided in this section that will describe the whole process of creating the *iso* image of the LiveDVD. The process will begin from scratch and will end with the resulting image being made. It is then a user's/developer's choice whether to burn the image or run it in virtual machine.

The liveDVD can be crated in a virtual or real machine. In either case, the process will need at least twice as much free disk space as is the size of the liveDVD – 15GB should be sufficient. The process also requires internet connectivity as it will download necessary packages. The liveDVD creation should take about 30 to 60 minutes (depending on the machine).

The necessary steps follows:

1. Get the LinuxICE liveDVD from [10] (section *downloads*).
2. Boot the liveDVD.
3. Go to *Apps* -> *Setup* and choose *Install*.
4. Install the system like you would any other Ubuntu (see [24]). Be sure to name the user "**superbel**".
5. After successful installation boot the system.

6. Go to *Apps* -> *Setup* and choose *ICE Configuration Util.* Then type in the user's name (it should be "superbel") and click "Go!".
7. Go to *Apps* -> *TerminalEmulator* and choose *Terminal* or use keyboard shortcut *CTRL+ALT+X*.
8. Copy the superbel installation script to home directory and run it like this (PATH must lead to the project `src` directory):

```
./superbel.py livedvd PATH
```

9. When the installer asks whether to install new packages choose *yes* (package *remastersys* requires one more *yes* as it is not authenticated).
10. The created image `superbel.iso` is located in `/home/remastersys/remastersys/`.

Chapter 6

Results

The result of this project is a car computer software targeted for electric cars. The software was designed to work well with touch screens so the users can interact with it easily even while driving.

The software is based on an open source multi-purpose car computer Linux distribution. This gave the project a solid platform to work on. The project presents a number of new applications and modules targeted specifically at electric cars. The modules collect data from the car and issue commands, like starting a charging of batteries. The applications present the information and let the user control some car functions. Some 3rd party applications are also included to do a common work, like music playback or web browsing.

The whole software was designed to be greatly modular and easily extensible. New applications or modules, for example for interaction with new hardware, may be introduced to the system with very little or no changes to the software at all. This gives the project great potential for future extensions as there are areas that still require development.

This project also tried to cover a gap in the open source software field as there are very few (and often limited) solutions for car computers, and probably none targeted at electric cars at all. Most of the car computer software available today focuses on media playback. Some of the other projects like this one are presented in the appendix A.

6.1 Simulation

As was already mentioned in the previous text, the resulting project was not implemented into the car but rather a simulation was created. The project should have been used in the *superbel* electric car described in the introduction 1.1. The car was being altered from combustion engine to electric propulsion which required a lot of work. As the car was part of a greater project and agenda, the main focus was on getting the car to be able to ride. The user interface created in this project wasn't a top priority as it is not crucial for the car. Therefore, the hardware needed for this project wasn't present in the car at the time of its creation and so it couldn't be implemented.

Instead, a simulation was created to show some of the software's functions and capabilities.

6.2 Speed

The software had some problems with speed during the testing as it reacted slowly to user commands from time to time. This might be due to several circumstances.

The first one is that some of the modules polls quite frequently for new data (every 100ms) and the application tries to update the presented data in that frequency as well. This may result in a number of signals being sent and processed as well as demanding graphical redrawing. A solution to this problem would be to use greater time intervals for the polling (which might create a badly responsive user interface) or use a different mechanism then polling altogether.

Another idea would be to try to identify the bottlenecks and optimize them using a lower-level language, like C++.

It could also be caused by running a number of applications simultaneously. Currently, applications are not killed so their switching can be fast (because they don't need to be started again). A solution might be an intelligent approach where only a few most used applications would be kept alive.

6.3 Extensibility and Integration

There is a number of areas where this project could be better or extended. One of the areas is the navigation that should contain a map of charging points. This map could then be used to plan the route accordingly because the car has only limited range.

Another area for extension is screen resolution and graphical layouts. Currently, the applications and their skins are optimized for 800x600px which is fairly common resolution on touch screens. Some of the applications are already prepared for easy transition to other resolutions but some are fixed, like the web browser for example.

Another important future extension is the possible integration into the car. Thanks to the modular and loosely coupled nature of the system, this should be as simple as writing an actual CAN bus module and removing the simulation application as described in 5.5.11. Of course, not all the features could be tested in the simulation (like GPS for example) so these might pose a potential source of problems during the integration.

Bibliography

[1] The brighton usability pattern collection. [Online] Available http://www.cmis.brighton.ac.uk/research/patterns/home.html, May 23, 2010.

[2] Controller area network. [Online] Available http://en.wikipedia.org/wiki/Controller_area_network, January 3, 2010.

[3] D-bus. [Online] Available http://www.freedesktop.org/wiki/Software/dbus, May 23, 2010.

[4] D-bus python bindings. [Online] Available http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html, May 23, 2010.

[5] Dive into python. [Online] Available http://diveintopython.org/, May 22, 2010.

[6] Electric cars. [Online] Available http://www.elektromobily.org, January 6, 2010.

[7] Icepanel. [Online] Available http://wiki.openice.org/index.php?title=Icepanel, May 23, 2010.

[8] Interaction design guide for touchscreen applications. [Online] Available http://www.sapdesignguild.org/resources/tsdesigngl/Index.htm, May 25, 2010.

[9] Jukx. [Online] Available http://jukx.sourceforge.net/, May 23, 2010.

[10] Linuxice. [Online] Available http://wiki.openice.org/index.php?title=LinuxICE, May 23, 2010.

[11] Matchbox window manager. [Online] Available http://matchbox-project.org/, May 13, 2010.

[12] Mini-box. [Online] Available http://www.mini-box.com/s.nl/sc.8/category.101/.f, January 4, 2010.

[13] mp3store. [Online] Available http://store.mp3car.com/Car_Computer_Systems_s/25.htm, January 4, 2010.

[14] Navit. [Online] Available http://www.navit-project.org/, May 23, 2010.

[15] nghost2. [Online] Available http://wiki.openice.org/index.php?title=The_nGhost_Project, May 23, 2010.

[16] Opencv. [Online] Available http://opencv.willowgarage.com/wiki/, May 23, 2010.

[17] Openice. [Online] Available http://wiki.openice.org/index.php?title=Main_Page, May 23, 2010.

[18] Pyqt. [Online] Available http://www.riverbankcomputing.co.uk/software/pyqt/intro, May 22, 2010.

[19] Python. [Online] Available http://www.python.org/, May 23, 2010.

[20] Qt. [Online] Available http://qt.nokia.com/products, May 23, 2010.

[21] Skoda superb launch in february 2009. [Online] Available http://indianautosblog.com/2008/11/2009-skoda-superb-launch-in-feburary-2009, May 25, 2010.

[22] Superbel. [Online] Available http://www.elektromobily.org/wiki/SUPERBEL, December 18, 2009.

[23] Touchscreen application tips. [Online] Available http://www.elotouch.com/Support/TechnicalSupport/10tips.asp, December 5, 2009.

[24] Ubuntu installation guide. [Online] Available https://help.ubuntu.com/9.04/installation-guide/i386/, May 24, 2010.

[25] Unix philosophy. [Online] Available http://en.wikipedia.org/wiki/Unix_philosophy, January 3, 2010.

[26] Vic limited. [Online] Available http://www.vic-ltd.com/products.html, January 4, 2010.

[27] Výfuky na vozy Škoda. [Online] Available http://www.bawel-autodily.cz/sortiment/tyll/skoda/, May 25, 2010.

[28] Douglass, B. P.: *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems.* Addison-Wesley Professional, 2002. ISBN 0-201-69956-7.

[29] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994. ISBN 0-201-63361-2.

[30] Johnson, J.: *GUI Bloopers 2.0: Common User Interface Design Don'ts and Dos.* Morgan Kaufmann Publishers, 2008. ISBN 0-123-70643-0.

[31] Rahimzadeh, A.: *Geek My Ride: Build the Ultimate Tech Rod.* Wiley Publishing, Inc., 2005. ISBN 0-7645-7876-6.

[32] Shalloway, A. and Trott, J.: *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Addison-Wesley Professional, 2001. ISBN 0-201-715945-0.

Appendix A

CarPC Software Overview

This appendix gives an overview of the software considered as a base for this project. It gives a short description of each one and offers reasons why it wasn't chosen.

A.1 PyCar

PyCar (http://pymedia.org/pycar/index.html) is a media center for car computer. It allows users to listen to music and watch movies in all the common formats used today. It also helps to organize the multimedia files and to use and create m3u playlists. It supports touch screens of various sizes and resolutions.

Figure A.1: PyCar Screenshot

Since PyCar is written in Python it can be easily used on Windows or Linux without any changes to the system. It can be extended with external scripts, modules and plugins and it can also be customized to show the information in a way that the user likes.

While PyCar is only a media center it was not suitable for this project.

A.2 Headunit

Headunit (http://sourceforge.net/projects/headunit) is a car computer software written from scratch using Qt library. It allows users to play and organize multimedia files while it remains independent of any external application to do so. It also supports radio and GPS

and the visual representation may be changed with MediaCar skins. Headunit is also ported to Windows.

The project seems to be dead now, last update done 4 years ago. According to its developer the project should have filled the gap when the other car pc software solutions were still immature.

A.3 DashPC

DashPC (http://www.dashpc.com) is a complex car computer software solution which was created in 1999. It is based on Linux from Scratch Linux distribution and it offers features like multimedia playing, navigation, engine diagnostics and wireless network connection. It supports variety of input devices like wireless keyboards, pointing devices and touchscreens.

Figure A.2: DashPC Screenshot

Last commit to the project code was done 5 years ago so DashPC doesn't seem to be in active development anymore.

A.4 KDE Software Compilation 4

Including KDE (http://www.kde.org) here might seem a little bit odd as it is by origin a desktop environment, which means it offers a suite of tools to use and to control personal computers by graphical user interfaces. These tools range from a window manager and panels to application launcher menus, applications themselves and others. What made it interesting for this project is its architecture. It makes it possible to create small applications, called plasmoids, that reside on the desktop. These applications can be as simple as clock or more sophisticated like RSS readers.

Plasmoids can be seen as GUI modules (something like widgets in standard user interfaces) that can be used to present data to user and to offer means to control the data. Furthermore, the user can control the nature of plasmoids by changing their size, rotation or position. Users can also close them or easily add new ones. Thus, KDE creates very interesting and flexible platform to build user controlled graphical interfaces upon.

Since the plasmoids presenting various car functions would be independent, separate entities serving only one purpose, they would conform nicely to the architecture described in section 3.2. This level of modularity would make it easy to create and add new plasmoids presenting data from new car systems.

Figure A.3: Screenshot of KDE with Various Plasmoids

The idea was to use this platform, use the plasmoids to build the GUI. It would require to create necessary plasmoids to show the information about car status and to control its functions. The users would then be able to arrange them as they would like, thus easily creating personalized user interface.

But the biggest advantage of using KDE for this project (flexibility and modularity) is also it's biggest disadvantage as it doesn't offer any car computer specific features. It's only a platform on which everything would have to be built from scratch. Therefore, KDE was considered not suitable for this project.

A.5 OpenICE/nGhost 3

nGhost 3 (http://www.mp3car.com/vbulletin/linuxice/122445-nghost-3-prototype.html) is a next development (still work in progress) version of nGhost 2, described in subsection 5.2.1. The biggest change is usage of Clutter library as a backend. Clutter allows to create advanced, visually rich and animated graphical user interfaces. It uses OpenGL as a renderer and it offers features like scene graph, JSON scripting to describe the layout, multiple input pointing devices and more.

The new GUI is based on this library and so it offers visually more interesting user interface with smooth transitions between screens and elements. There are some examples available on youtube[1].

Another changes to the nGhost 2 structure contains event handling improvements, network/IPC improvements and new plugins. nGhost 3 also tries to maintain backward compatibility with existing plugins.

The last information about nGhost 3 is that it's developers think they are "heading

[1] http://www.youtube.com/results?search_query=nghost+3&aq=f

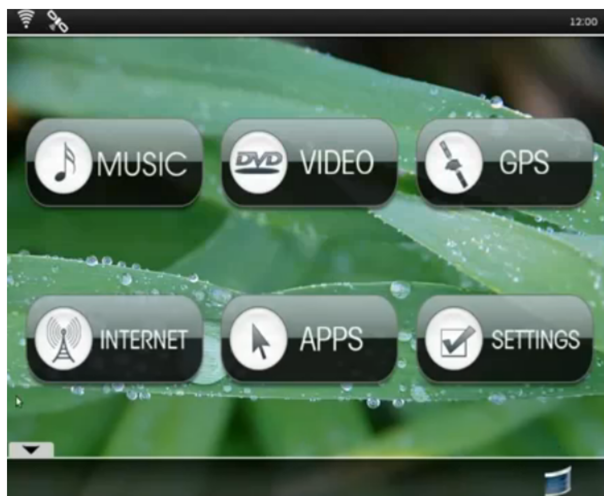


Figure A.4: nGhost 3 Screenshot

down a technological deadend"[2], therefore a major rewrite might be needed.

[2]http://forums.openice.org/viewtopic.php?f=1&t=465&start=15

Appendix B

User's Guide

B.1 Running

To run the system simply boot the attached superbel LiveDVD.

B.2 Installation

To install the software on a hard drive follow these steps:

1. Boot the attached superbel LiveDVD.
2. When the boot process asks what action to take choose *install*.
3. Then proceed according to the installation wizard (see [24]). Be sure to name the user "**superbel**".
4. After successful installation reboot the machine.
5. You should be now greeted by the log in screen of the newly installed system.

B.3 Usage

B.3.1 Login

You will be greeted by the login application after booting the system. To successfully log in you have to give a correct password which consists of four colors. You can also change the current password here. Should you fail three times in choosing the password the attempt will be logged.

B.3.2 Main Screen

After successful login you will arrive at the main screen of the system (shown in picture B.1). This screen gives an overview of the car status and you can navigate from here to the other applications.

The numbered buttons in the menu in the bottom have the following meaning:

1. Brings up the shutdown options

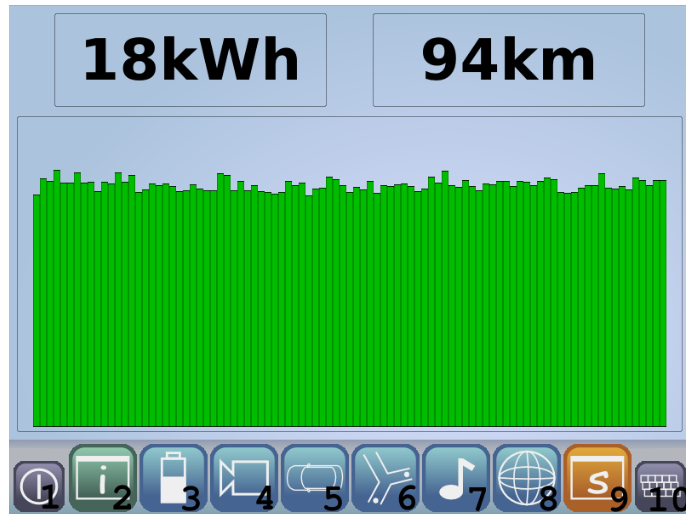


Figure B.1: Main Screen

2. Shows the main screen
3. Shows the battery charging application
4. Shows the rear camera application
5. Shows the collision detection application
6. Shows the navigation
7. Shows the music player
8. Shows the web browser
9. Shows the simulation application
10. Brings up the software keyboard

Appendix C

CD Contents

Two discs are attached to the thesis. Their contents are described below.

- LiveDVD - Contains the final redistributable project. It can be run from the DVD or installed to the hard drive.
- Sources CD
  - `docs` - Contains the thesis documentation.
  - `docs/src` - Latex sources of this documentation.
  - `docs/doc` - The final PDF documentation.
  - `src` - Contains the thesis source code.
  - `src/apps` - Source code of the applications.
  - `src/bin` - The installation script.
  - `src/data` - Data attached to the project.
  - `src/modules` - Source code of the modules.
  - `src/skins` - Skins and configuration files for the applications.
  - `src/jukx-0.4.2b.tar.gz` - JukX package built from sources.
  - `src/opencv-2.0.1-pkg.tar.gz` - OpenCV package built from sources.