Design, Development and Integration of a Wireless Communication Unit in the Concept Car

Master Thesis

University of Kaiserslautern

Department of EIT and Computer Science

Embedded Systems Group

Omair Rafique

January 2013

Supervisory Committee:

Professor Dr. Klaus Schneider (Fachbereich Informatik)
Professor Dr. Dominik Stoffel (Fachbereich EIT)

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Kaiserslautern, den 14.01.2013

Omair Rafique

Danksagung

An dieser Stelle möchte ich allen danken, die es mir ermöglicht haben diese Arbeit zu verfassen. Dazu zählen vor allem meine Freunde und meine Kollegen in der Arbeitsgruppe Eingebettete Systeme. Des Weiteren danke ich besonderes meinem tutor Manuel Gesell für die geleistete Arbeit. Meinem Betreuer Prof. Dr. Schneider danke ich vorallem für das angenehme Arbeitsklima und die gute Betreuung während der Arbeit. Ausserdem, danke ich meinem Betreuer Prof. Dr. Stoffel vorallem für die gute Betreuung während der Arbeit.

Zusammenfassung

Das Concept Car ist ein experimentelles Eingebettetes System mit dem Ziel, moderne und zukünftige Funktionalitäten im Automobilbereich zu testen und überprüfen. Es ist eine Forschungsplattform, die zur Zeit nur durch eine Funkfernsteuerung (Transmitter) gesteuert und angetrieben wird. Darüber hinaus gibt es zwei Wege um an Informationen über die Peripherie des Autos (Energie-Status, Fahrzeuggeschwindigkeit, ECU-Status etc.) zu kommen. Die erste Möglichkeit ist der Online-Modus: Das Auto ist mit einem Computer mittels CAN Viewer Hardware verbunden, somit können die Informationen dort überwacht werden. Die zweite Möglichkeit ist der Offline-Modus: Die Daten werden auf einer SD-Karte gespeichert, damit die Daten später ohne direkte Verbindung zum Concept Car (offline) überwacht werden können. Die Idee dieser Arbeit ist nun, den Status des Fahrzeugs (drahtlos) zu überwachen, während es angetrieben wird, und das Auto ohne Einsatz der Funkfernsteuerung zu betreiben, dabei werden Eigenschaften der beiden Modi kombiniert. Da Smartphones heutzutage sehr verbreitet sind und eine Vielfalt von speziellen Hardware-Features (wie Beschleunigungsmesser, Bluetooth, NFC, etc.) beherbergen, ist die Idee, diese speziellen Funktionen des Smartphones zu benutzen um das Concept Car zu steuern und überwachen.

Abstract

The Concept Car is an experimental embedded system with the objective of testing and verifying modern future car features by deploying different classes of applications. It is a research platform that by the time, has only been controlled and driven by a radio controlled transmitter system. Additionally, there are two ways of getting the information about the car internals (power status, car speed, ECU's status etc.). The first option is the *Online Mode*: The car is connected to the CAN- Viewer hardware by means of wire, which allows to monitor the information on the PC or laptop. The second option is the *Offline mode*: The data is stored on an SD-Card, which allows to monitor the stored data offline. The idea now is to *lively* monitor the status (wirelessly) of the car internals on the fly while it is being driven, and to drive the car without using the battery operated radio transmitter system, thereby combining features of both the modes. Since smartphones are immensely used these days and can come up with variety of special hardware features (like accelerometers, Bluetooth, NFC etc.), the idea is to use these special hardware (software controlled) features of the smartphone to monitor and control the Concept Car mainly, by interacting with the ECUs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

'Exploring the Idea'

This chapter explains the context by introducing the basis of this topic i.e. the 'Concept Car'. It defines the basic functionality and architecture of the Concept Car, defines the area of problem and finally explores the basic idea of solving it. Furthermore, the structure of the thesis is outlined.

1.1 The Context

Before going deep into the specific area of problem, let's first discuss the platform that forms the basis of the thesis, namely the Concept Car.

The Concept Car is an experimental embedded system with the objective of testing and verifying modern future car features by deploying different classes of applications. The Concept Car is a research platform remotely operated via a standard 2-channel (throttle and steering) 27MHz radio transmitter system. It incorporates a set of sensors (wheel speed, gyro/accelerometer, distance etc.) for interacting with the environment and surroundings. It uses an air-cooled sensorless brushless electrical motor for throttle, and a servo motor for steering. Depending on the ground conditions, it is capable of driving at a speed up to 50 km/h.

The basic architecture of the Concept Car, as depicted in Figure 1.1, is comprised of three processing units: input processing, data processing and output processing.

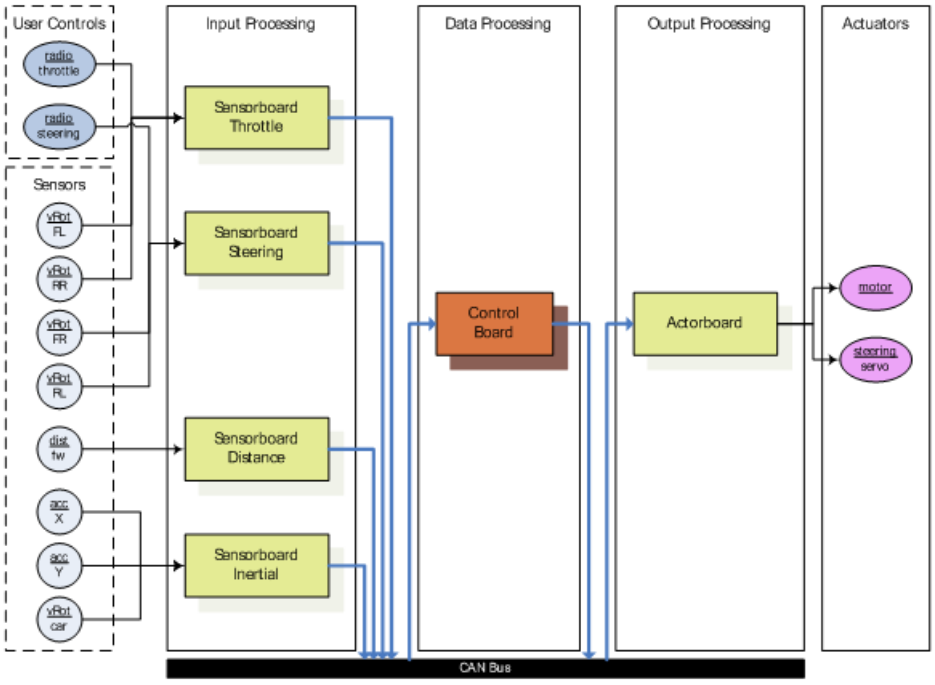


Figure 1.1: Basic Building Block Diagram of the Concept Car

The **input processing unit** is comprised of all the sensor boards, where each board forms a separate electronic control unit (ECU). Each sensor board

strictly interacts with the appropriate sensor components (as depicted in the *User Controls* part in Figure 1.1), receives the sensory signal, processes it, and places it to the centralized CAN bus (if required). For instance, SensorBoard throttle ECU decodes the throttle PWM from the radio receiver and the wheel speed sensors from the front left and rear right wheels. Based on the received throttle signal, it then creates the corresponding acceleration signal (CANID_THROTTLE), and finally places it on the CAN bus. Each sensor board ECU is implemented on an 8 bit AT90CAN128 [1] micro-controller. **The data processing unit** is meant for performing complex mathematical computations. For this purpose, it uses an AT91SAM7A2 [2] board that incorporates ARM7TDMI embedded processor with a 32-bit RISC architecture. This board is provided with several embedded peripherals like CAN interfaces, SPI bus, PWM modules, Timer/Counters etc. **The output processing unit** is mainly comprised of a single ECU namely ActorBoard. This ECU is held responsible for creating the PWM signals to drive the actuators (dc motor and servo). It receives the CAN messages and generates the respective PWM signals for controlling the actuators.

The Concept Car features two independent power supply trains (see Figure 1.2):

- 5S1P LiPo, 5000mAh, 18.5V: for the "heavy-load" electric system (actuators of the car)

- 3S1P LiPo, 5000mAh, 11.1V: for the ECUs interfaced with the CAN bus

The heavy-load electric system includes an engine controller, a radio receiver and a steering servo; is controlled by the 18,5V LiPo battery. Since the maximum voltage that the radio receiver and the steering servo can upheld is 6V, the idea is to use a step-down voltage regulator that steps down the voltage to 6V. All the ECUs (including the ARM board) are supplied with power using a separate 11.1V LiPo battery. Every ECU internally incorporates a voltage regulator for stepping down this voltage to 5V (discussed in detail in the design phase). Since two different batteries are incorporated into the design, it uses the *Galvanic Isolator* for isolating functional sections of electrical systems to prevent current flow.

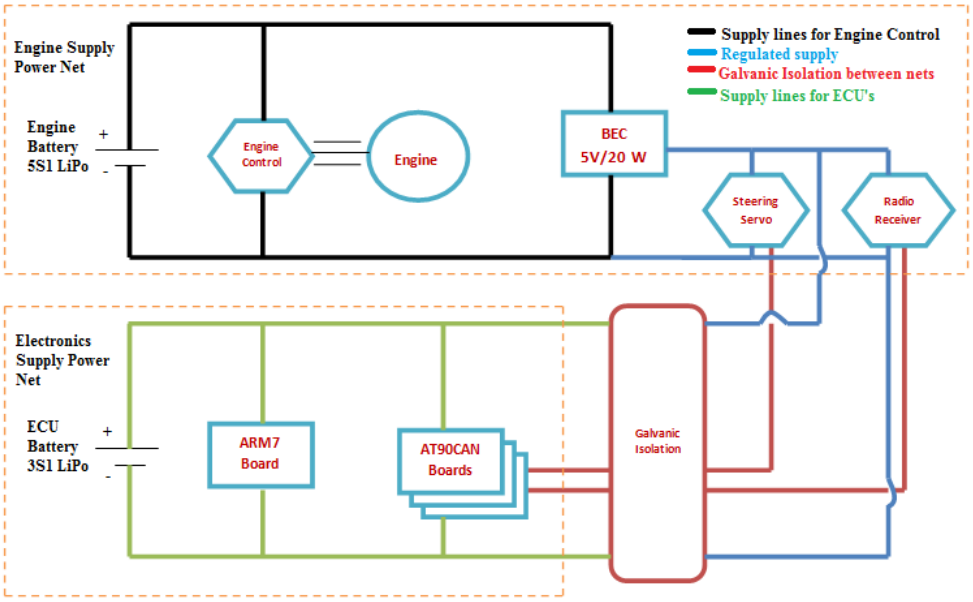


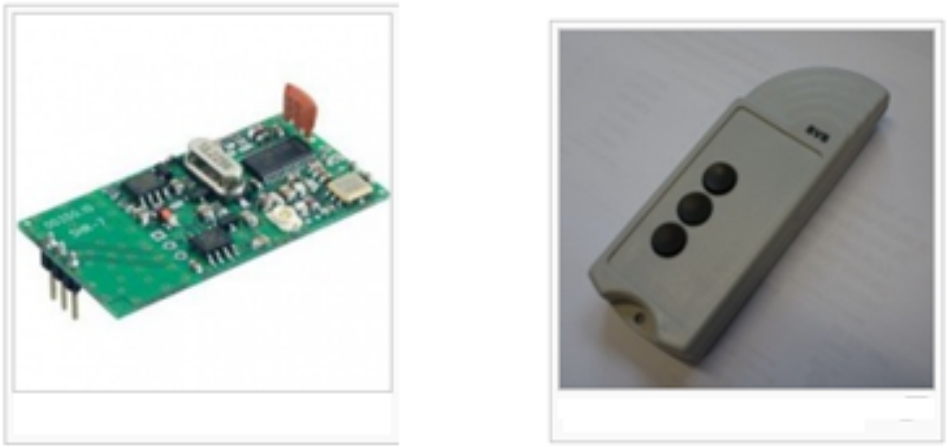
Figure 1.2: The Concept Car's Power Train

This galvanic isolation has been implemented by incorporating a separate board namely EmergencyBoard that mainly serves two purposes:

1. Isolates the actuators from the other boards

2. Provides the user a feature of invoking an emergency stop

This board uses an additional transceiver module from SVS [3], with the receiver (SHR-7) being placed on the board. The SHR-7 sender remote signal can reach up to 1.5 km. Pushing any of the three buttons on the emergency remote control sets the car in a stop mode (see Figure 1.3). Secondly it isolates the actuators from all the other ECU's, thereby removing any possibility of damaging the other ECUs due to any power consumption issue related to the motor and servo. This ECU has been implemented on an 8 bit ATmega88 [4] microcontroller. As depicted in Figure 1.1, all the ECUs (except EmergencyBoard) communicate with each other via centralized CAN bus, with the maximum achievable data transfer rate of 1 Mbps.



http://www.svs-funk.com/

Figure 1.3: The Emergency ECU Transceiver Module

The Driving Mechanism

After getting a firm idea about the architectural view of the Concept Car, it is the right time to discuss how different ECUs are involved in steering and driving the Concept Car. As discussed, a standard 2-channel 27MHz radio transmitter system is responsible for generating throttle and steering signals. This radio transmitter generates the pulse width modulation (PWM) [5] signals for each channel, with 20ms cycle length and 1ms to 2ms duty cycle, where 1.5ms duty cycle represents the idle position. As shown in Figure 1.4, these signals are fed in to the SensorBoard steering and SensorBoard throttle via the EmergencyBoard (1), where the PWM signals are calculated and normalized. This normalized data is then finally placed on the CAN interface (2). In case if complex mathematical computations are required, the ARMBoard receives the normalized data, performs the desired calculations, and places the processed data on the CAN bus with a different id. The selector switch on the ActorBoard chooses the source of data, either receiving processed data from the ARMBoard or normalized data from the SensorBoards (3). Independent of the source, the ActorBoard, based on these signals, finally produces the desired PWM signals (4), and passes these signals to the actuators (servo motor and dc motor) via the EmergencyBoard (5).

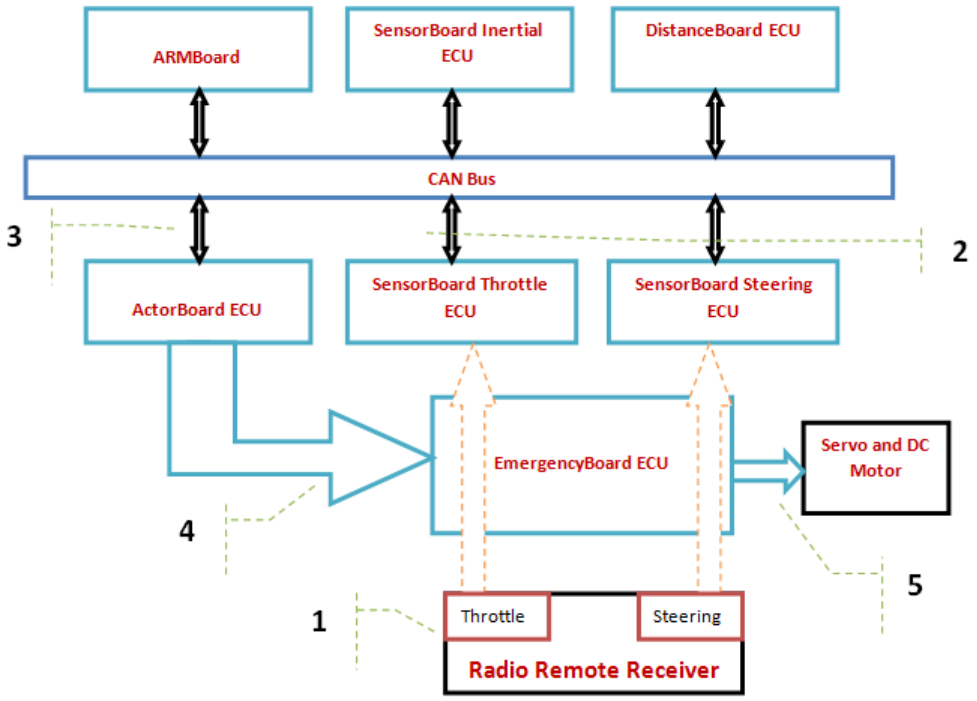


Figure 1.4: The Concept Car's Driving Mechanism

It is important to comprehend the services offered by the EmergencyBoard. Having no access to the CAN bus, this board only accepts the input from the radio receiver and the ActorBoard, and bypasses it to the SensorBoards and the actuators (servo and DC motor) respectively. The EmergencyBoard only bypasses the PWM signals as provided by the radio remote receiver and the ActorBoard, if there is no emergency signal being sent from the SVS transceiver module, thereby bringing in the feature of emergency stop and the galvanic isolation (isolating the actuators from the rest of the ECUs)

1.2 The Problem

Since there are two different problems to consider, this section is split in two different sub-sections: the idea of monitoring the car internals lively, on the fly, while the car is being driven, and making the car controllable with an alternative to radio transmitter system.

1.2.1 Problem 1: Monitoring the Car Internals

As discussed in the previous section, all the ECU's interact with each other via CAN bus (except EmergencyBoard). For testing the deployed applications it is extremely important to have access over the CAN bus, to analyze and monitor the messages sent by the ECUs. This eventually corresponds to monitoring the internals of the car, as each message corresponds to a certain parameter for example: wheel speed, steering data, throttle data, battery status etc. By the time, we are provided with two different options. The first option is monitoring in the *Online Mode*. This corresponds to connecting the CAN bus of the car to a PC or laptop, by using the CAN viewer hardware via USB interface (it is discussed in detail in Section 5.4). This approach is only applicable and useful if the car stays static. The second option is monitoring in the *Offline Mode*. This approach relies on logging the values on the SD card, and analyzing this logged data later offline. The problem arises when it comes to *lively* monitoring the CAN bus (car internals) on the fly, while the car is being driven. Thereby, the solution to lively monitoring the car internals is desired.

1.2.2 Problem 2: Driving the Car Independent of the Radio Transmitter System

By the time, the only way of driving the car is to use the standard 2-channel (throttle and steering) 27MHz radio transmitter system. This radio system is battery operated (12V) and is capable of transmitting signals up to a distance of 1.5 Km. Logically speaking, we don't need the car to move that far when it comes to driving over shorter range of distances for testing purposes, inside a room or a lab. Secondly, it uses the peripherals of the Sensorboard ECUs for measuring the pulses for throttle and steering. This affects the use of energy and CPU utilization overhead for the Sensorboards. Thereby, a solution capable of replacing the radio transmitter system is desired.

1.3 The Solution

It is quite obvious from the previous sections that the basic solution to both the problems lies in the idea of introducing the means of some wireless communication within the Concept Car. But still it is important to analyze the structure of integrating it with the already existing platform. The key to the

answer lies in the centralized CAN bus architecture of the car. Since we are aware of the fact that each ECU sends the desired messages over the CAN bus, it is quite obvious that either the task of monitoring (getting into the internals) or the task of driving the car, is completely possible if we have access over the CAN bus. Firstly, the idea is to incorporate a wireless communicator within the Concept Car. This communicator should be blessed with features like:

- A wireless communication technology based on Radio Frequency (RF)
- Capable of accessing all the parameters of the car
- Able to interact with all the ECU's

Secondly, at the user end, we require a device or a computer for replacing the CAN viewer (for monitoring) and radio controlled transmitter system (for driving the car). This device must incorporate some important software controlled hardware features like:

- A wireless communication technology based on Radio Frequency (RF) for interacting with the car
- Sensing capabilities like accelerometer for producing steering and throttle signals
- A user interface for monitoring the car internals

The combination of both, the wireless communicator within the car and the user-end device incorporating the features mentioned above, forms the basic solution to the problem (see Figure 1.5). This solution is elaborated in the Chapter *Design*.

Figure 1.5: The Idea

1.4 State of the Art

The idea of designing and implementing a remotely controlled car is not a new one. Initially the remote controllers developed were based on the visible light, however, by the time, most of these implementations are radio controlled (RF based). There exist a number of remotely controlled electric cars, each incorporating various features and attributes. Many designs and implementations use their specific architectures, while some incorporate proprietary hardware like LEGO MINDSTORMS [6]. Most of these designs although resembling a conventional car, incorporate a single ECU responsible for carrying out all the desired operations. The Concept Car, on the other hand, uses the idea of a modern car architecture with different ECUs responsible for different operations, interacting with each other using the centralized CAN bus. The architecture specific implementation as presented by [7, 8], brings in a smartphone based Bluetooth controlled car. Both of these designs incorporate an on-board device (PDA or Laptop) at the device side. Unlike the Concept Car (as presented in this work), these designs do not incorporate individual ECUs for managing different operations. Another architecture specific design as presented in [9], enables a car to be controlled wirelessly by the Bluetooth interface using a PC. A serial Bluetooth device is connected to the PC for generating instructions to the Bluetooth module at the device end. Likewise the first two designs discussed, it also uses a single control unit at the device side, responsible for managing all the operations. The proprietary hardware designs as presented in [6, 10], uses the LEGO MINDSTORMS NXT, can be controlled using smartphones. These designs offers the features of steering and driving the LEGOs by generating the commands over the Bluetooth interface of the smartphone. However, they do not incorporate the monitoring features, as offered by the Smart Monitor of the Concept Car. Apart from Bluetooth used as the wireless technology, there exist implementations based on other wireless technologies. One example is the implementation of a general packet radio service (GPRS) [11] based *Monorail Car* [12]. This vehicle is driven using a smartphone over set up rails, for the purpose of picking fruits in the mountain areas. Although it involves the full duplex communication, but still a single ARM 9 architecture based embedded processor is used as the control unit. Another example as presented in [13], uses WiFi as a communication technology, for controlling the car from a PC. Likewise the previous implementation, it incorporates both the monitor and the controller, while using an ARM 9 processor as the only ECU available. The idea of controlling a modern car with a smartphone is already being attempted by the AutoNOMOS laboratories [14]. The cur-

rent version of this design namely iDriver, uses iPhone 3GS for steering and driving the modern car. Although it receives the video stream from the car's built in camera, it still does not monitor the internals of the car, as featured by the Smart Monitor of the Concept Car.

1.5 The Structure

This report is intended to present a self explanatory guide from design to delivery. This section briefly describes the structure of the whole thesis section-by-section. Being inspired from the title *Design, Development and Integration of a Communication unit in the Concept Car*, the whole project is organized into four main phases. Let's discuss each phase briefly:

Design

This phase presents all the efforts that are put in for getting the directions to proceed. It starts of discussing the basic design of the wireless communicator and the integration to the already existing structure of the Concept Car. It explains the idea by taking into consideration a simple example. It also discusses about the options available for forming the wireless communication and then discusses the feasibility criteria of selecting one of the communication technology. It discusses in detail the hardware tools/components required for the wireless communicator and the feasibility criteria of selecting the components for the design. Finally, it discusses about the software tools availability and their selection criteria.

Development

As the name suggests, this phase collects all the information from the design and starts implementing the directions gained from the previous section. It starts of presenting and discussing the hardware layout of the wireless communicator, explaining how hardware components selected in the design phase are merged together. It then presents the initial testing and configuration of the wireless communicator. It also presents the first communication test i.e. testing the communicator with the loopback test. Finally, this phase presents the development of all the required software components/libraries (at the device side and as well as at the user end) that are used for the final applications. This include the application specific communication protocol, development of sensor libraries and data handling.

Integration

In the development phase we successfully developed all the required hardware and software components. We initiated the tests for configuring and running the hardware without using any software. In the integration phase we integrate all the independent hardware and make it run with the software libraries we developed in the previous phase. This include running the hardware at both ends i.e. the device end (Concept Car) and the user end, using the software libraries developed at both ends. This phase also tests and verifies the application specific communication protocol by initiating the data validation tests.

Deployment and Delivery

In this phase we play with the real deal i.e. gaining the desired results. The idea of just forming a wireless communication path between the Concept Car and the user-end device would end up gaining nothing unless we test and verify it with the real time applications. For the purpose, this phase presents two main applications that incorporate all the features we *designed, developed and integrated* in the previous phases. These applications involve the use of all the hardware and software features at both ends i.e. the Concept Car and a user end device.

Chapter 2

Design

'Getting the Directions'

This chapter proceeds with the basic idea (as presented in the previous chapter) and formulates the basic idea to a basic design. It actually explains how we are going to manage the basic solution in the already existing structure of the car. The available communication technologies and the selection criteria are discussed. Finally, it explores the availability and selection of the hardware tools required for the Wireless ECU and the selection of the software tools for the Wireless ECU and a smartphone.

2.1 The Basic Design

In the previous chapter we discussed the basic solution, which comprises of two different parts i.e.

1. Introducing a *wireless communicator* within the Concept Car
2. Introducing a *user-end device* at least capable of some wireless communication

Let's formulate each idea into a basic design:

2.1.1 From Wireless Communicator to Wireless ECU

In the previous chapter we emphasized on one of the basic architectural features of the Concept Car i.e. the centralized CAN bus architecture. Taking advantage of this structure, we know that the only possible way of interacting with all the ECUs is simply getting an access to the CAN bus. So now the idea of introducing a wireless communicator formulates to introducing a new ECU in the Concept Car namely the *Wireless ECU*. Likewise all the other ECUs, this ECU would be capable of placing and getting messages from the CAN bus as it would be directly connected to the interface. Most importantly, it should incorporate some mean of sending/receiving data wirelessly.

Let's examine this design by considering a simple scenario, as shown in Figure 2.1. The idea is to receive data wirelessly and drive the actuators based on the data values. Stepping up from point 1 - 6 serves the purpose. Starting from Step 1, the desired data is received wirelessly by the Wireless ECU. It then places this data with a specific id on the CAN bus, in Step 2. In Step 3, the SensorBoard ECU particularly receives the data with this id, performs the desired mathematical operations and places the modified data on the CAN bus again with a different id. In case if intensive mathematical computations are required, 32 bit ARMBoard (processing unit) is used, which has the access to the CAN bus. In Step 4, the ActorBoard receives this modified data and generates the required signals. In Step 5, it passes the calculated signals through to the EmergencyBoard. Finally, the EmergencyBoard passes on these signals for driving the actuators through the galvanic isolation.

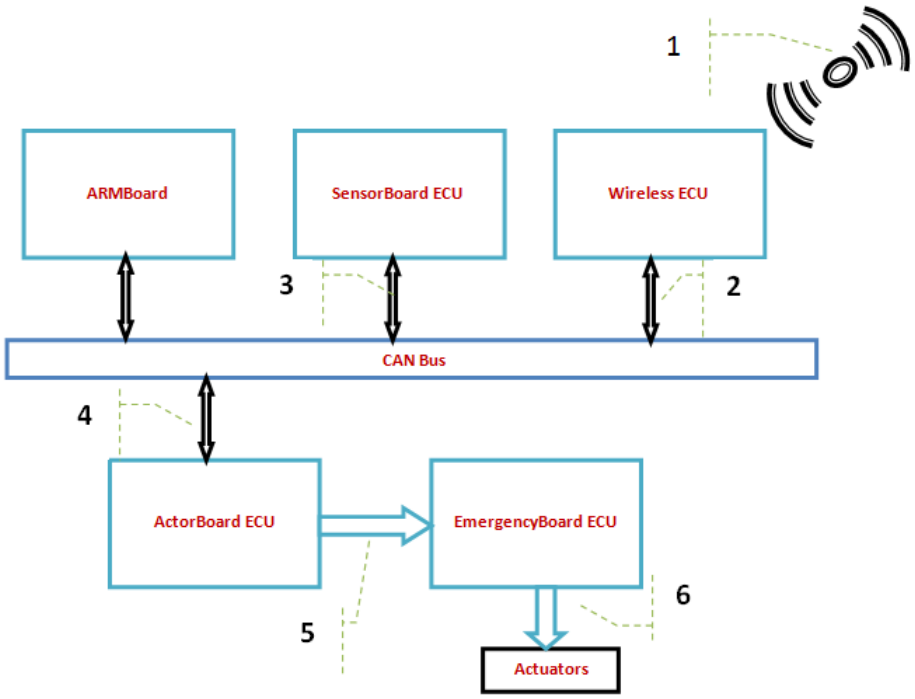


Figure 2.1: A Basic Example for the Wireless ECU

2.1.2 From User-End Device to Smartphone

In the previous chapter we talked about having a user-end device, capable of communicating wirelessly and incorporated with some software controlled hardware features. We can come up with any device incorporating a microprocessor, memory, a display and a wireless communication technology. The most commonly used device that can fulfill our cause is a *smartphone*. A smartphone is similar to a common phone, with advanced computing capabilities and powerful features. It is now even being used very commonly in the area of application development. Since smartphones come up with many interesting software controlled hardware features like accelerometers, near field communication (NFC) [15], WiFi [16], Bluetooth [17] etc and are widely used these days, the idea is to implement these special software controlled hardware features for monitoring and controlling the Concept Car. One of the most important advantage of using a smartphone is that we don't need additional devices like accelerometers, wireless communication device etc., instead we rely on the built in features of a smartphone.

2.1.3 The Basic Design in a Nutshell

We finally reached the point of formulating our basic solution to the basic design. As discussed in detail in the previous two subsections, we can finally define our basic design in a nutshell as:

1. Introducing a separate ECU incorporating some wireless communication i.e. introduction of the Wireless ECU

2. Introducing a software tool exploiting the special hardware features, at the smartphone side

Before making an attempt to develop and implement this design, an effort is required to select the most feasible resources from the lot.

2.2 Availability and Selection

Keeping in mind the basic design we discovered in the previous section, this section (comprised of three sub-sections) discusses in depth about the possible resources (the wireless communication technology, hardware components and software components) that could be considered for this design. It presents the tradeoff between selecting different resources, while the discussion ends up selecting the most suitable resources for the particular design.

2.2.1 The Communication Standard

In the previous section, we finally chose a smartphone as a user-end device. This complies to restricting our choices with the available built in wireless communication technologies, in smartphones. Thereby, we are up to the task of discussing various feasible wireless communication technologies available with the smartphones, eventually ending up with one selected technology. There are three most commonly used wireless communication standards: NFC, WiFi and Bluetooth. All of these three standards are based on radio frequency with different; range of operating frequencies, range of distances, power consumption, package size, interface, cost and data transfer rates. Let's discuss each of the technology independently:

NFC

NFC is a wireless standard for smartphones and similar devices to introduce the radio communication between devices by almost touching them together or at least bringing them as close as few centimeters (4 cm to be exact). Using the NFC standard, communication can also be established between an NFC device and an unpowered NFC chip, called a Tag. The NFC standard cover communications protocols and data exchange formats, and are based on existing radio-frequency identification (RFID) standards including ISO/IEC 14443 and FeliCa [18]. NFC standard is based upon RFID systems that allows two-way communication between endpoints.

WiFi

WiFi is another popular wireless communication technology that allows smartphones (and other electronic devices) to exchange data, including high speed internet connections. It is also based on RF communication and exchanges data in the form of radio waves. It is based on the IEEE 802.11 standard [19]. WiFi operates in the 2.4, 3.6 and 5GHz frequency bands. Depending on the antenna power and environment, typically, it is capable of emitting a radio wave to travel 20 meters. But this range could easily be increased to 100 meters by increasing the transmit power. According to the IEEE 802.11n standard, as per October 2009, it is capable of transmitting at a maximum net data rate from 54 Mbit/s to 600 Mbit/s.

Bluetooth

Bluetooth is another wireless technology standard commonly used for exchanging data over short distances. Likewise WiFi, it is based on radio frequency and exchanges data in the form of radio waves. It operates using short-wavelength radio transmissions in the frequency band from 2.40 to 2.48 GHz. In the early years of its development, it is conceived as the wireless alternative of the RS232 serial data cables [20]. Depending on the antenna power and environment, typically (at 0dB), it is capable of communicating at up to 5 meters. With the introduction of the Bluetooth v2.0+EDR (enhanced data rate), it is capable of transmitting at 3 Mbit/s covering the distance up to 20 meters. As the name suggests, EDR enhances the data transfer rate of v2.0 up to 3 Mbit/s. Bluetooth transceivers are not easily affected by the noise because of the hopping mechanism [9]. The complete specification of the Bluetooth standard is given in [21][22].

The Criteria

Before comparing the discussed technologies and selecting one, it is extremely important to set up the feasibility and selection criteria. As discussed in Chapter 1, a 27MHz radio transmitter capable of communicating at a range more than one Km is currently used. We are more up to the task of finding an alternative to this approach, that is more energy efficient than the other competitive technologies available, regardless of being capable of covering long distances. In a nutshell, we are up to the task of selecting a technology that is capable of at least covering the distance for testing the applications, preferably in a room or lab covering at most 20 meters, with low power consumption. Let's select one of the three discussed technologies based on this criteria.

Comparison and Selection

We briefly discussed each of the available wireless communication technologies in smartphones. Let's end up this section by comparing them and selecting the most appropriate one for our cause.

Looking into the range of the NFC standard, the term contactless is more appropriate than wireless. For this reason, it is widely used in applications like: contactless payment systems, similar to those currently used in credit cards and electronic ticket smartcards [23]. Generally, it is a low speed and short distance protocol, and can be possibly used for bootstrapping more capable wireless standards like Bluetooth, WiFi etc. One idea would be enabling, pairing and connecting to a Bluetooth/WiFi device, once the NFC device finds the desired tag [24]. Due to its short range and low speed, it is totally inappropriate, even for testing the applications. However, it can be incorporated later in the design for implementing a smart connection mechanism.

Although both Bluetooth and WiFi seem quite appropriate in terms of our selecting criteria, however, each technology dominates over the other in one or the other feature. We are going to select one based on how the features offered by these communication standards match the desired criteria. Table 2.1 presents the most common specifications for both standards:

Property	Bluetooth	WiFi
Frequency (GHz)	2.4	2.4, 3.6 and 5
Bandwidth	800 Kbps	11 Mbps
Power	Low	High
Range (meters)	20	100
Bit Rate (Mbps)	3	600
Cost	Low	High

Table 2.1: Common Characteristics of Bluetooth and WiFi

Although WiFi and Bluetooth operate in the same 2.4 GHz frequency band, WiFi uses more transmit power to cover almost 5 times the distance covered by the Bluetooth. Depending upon the application, WiFi may use as much as 40 times the power used by the Bluetooth. The data rate offered by WiFi is far better than Bluetooth, but still with Bluetooth v2.0+EDR, an intermediate bit rate of 3 Mbps can be achieved. In terms of cost, Bluetooth is quite economical than WiFi. The low power consumption and low cost, in combination with acceptable data rate and range, Bluetooth better suits our selection criteria and is the selected wireless technology for this thesis.

2.2.2 The Hardware Tools

The term hardware tools means the hardware components required to design the Wireless ECU. The four most important components that would be required to design the Wireless ECU are:

1. Bluetooth Device
2. Microcontroller
3. CAN Driver
4. Voltage Regulator

In the previous section, we decided to use Bluetooth as the wireless communication technology. Now, it is important to select one of the many available Bluetooth devices, that fulfills our design requirements like; class, size, range and speed. Secondly, for the design of an ECU, we require; a competitive, fast and low power processing unit i.e. a Microcontroller. We also need a CAN driver for protecting the bus lines against transients. Finally, in order to provide a 5V power source to all the components on the board, we require

a voltage regulator to step down the voltage taken from 3S1 LiPo battery (see Figure 1.2). Generally, it is not possible to present all the hardware tools available (hundreds of them), and then select one. Instead we are going to present the selected ones, and are going to discuss the selection criteria. Let's discuss each of these separately:

Selecting the Bluetooth Device

There are number of Bluetooth modules available in the market these days, classified by; power class, standard, size and cost. Based on these characteristics, Table 2.2 compares the Bluetooth devices:

Class	Maximum Permitted Power (mW)	Range (meters)
Class 1	**100**	**~100**
Class 2	**2.5**	**~10**
Class 3	**1**	**~5**

Table 2.2: Bluetooth Modules Classification

The most important change that one can notice between the Bluetooth 1.0 standard and the 2.0 standard is range. According to the Bluetooth special interest group specifications, devices with 1.0 standard mostly cover a radius of only 5 meters, though this range can be increased to 100 meters (by increasing the antenna power), power consumption issues make the use of these devices very rare. On the other hand, devices adhering to 2.0 standard, has improved power consumption and typically covers the range of about 20 meters.

For the Wireless ECU, we are using the **RN-42 Bluetooth Module** [25]. It is a Class 2 Bluetooth Module that adheres to Bluetooth v2.0+EDR. It is incorporated with an on board chip antenna. It is a small size module with dimensions as small as: 13.4mm x 20mm x 2 mm. It is a low power (26uA during sleep, 3mA connected, 30mA transmit) and a high speed (up to 3.0 Mbps) Bluetooth module that covers a distance of 20 meters. It supports both UART (universal asynchronous receive transmit) [26] and USB (universal serial bus) [27] interface connections. A UART is an individual or part of an integrated circuit used for communicating serially over a computer or peripheral device serial port, such as micro-controllers

Selecting the Microcontroller

The most basic element of the ECU is the processing unit. We need a processing unit that offers a good trade off between high speed and low power consumption, and allows the access to special peripherals like Timers, Pulse Width Modulators, Interrupts etc. As a part of the design, we need to have the access over the CAN bus, so a micro-controller equipped with a CAN controller is desired.

Likewise the other ECUs (SensorBoards, ActorBoard) we decided to go with the same controller i.e. **AT90CAN128**. The reason is getting satisfactory results for the other ECUs. It is an 8 bit microcontroller, with an advanced reduced instruction set computer architecture (RISC). It includes 128K Bytes of in-system reprogrammable flash, 4K bytes EEPROM, 4K bytes internal SRAM and up to 64K bytes optional external memory space. It incorporates peripheral features as: programmable watchdog timer with on-chip oscillator, 8-bit/16-bit synchronous and asynchronous timer/counter with PWM, JTAG (IEEE std. 1149.1 Compliant) interface etc. Most importantly, it also supports the CAN controller, with a maximum data transfer rate of 1 Mbps.

Selecting the CAN Driver

A CAN driver is the interface between a CAN protocol controller and the physical bus. The device provides differential transmit capability to the bus and differential receive capability to the CAN controller. Likewise the other ECUs, we are going to use the same CAN Driver i.e. **PCA82C250** [28], as it turned out to work efficiently.

The PCA82C250 CAN driver is primarily intended for high-speed automotive applications (up to 1 MBd). The device provides differential transmit capability to the bus and differential receive capability to the CAN controller. A current limiting circuit protects the transmitter output stage against short-circuit to positive and negative battery voltage. Although the power dissipation is increased during this fault condition, this feature will prevent destruction of the transmitter output stage. It is fully compatible with the ISO 11898 standard. It offer features like: Slope control to reduce radio frequency interference (RFI), differential receiver with wide common-mode range for high immunity against electro-magnetic interference (EMI), thermally protected, short-circuit proof to battery and ground, low-current standby mode, an unpowered node does not disturb the bus lines and at least 110 nodes can be connected.

Selecting the Voltage Regulator

Likewise all the other ECUs, all the components on the Wireless ECU are required to be powered up with a 5V source. As being successfully used with all the other ECUs, we are going to use a voltage regulator namely MAX1837 [29]. It is a high-efficiency step-down converter that provides a preset 5V output voltage from supply voltages as high as 24V. It is available in a 6-pin SOT23 package, making them ideal for low-cost, low power, space-sensitive applications.

2.2.3 The Software Tools

This section is intended to discuss the software tools required for writing, compiling and programming code for the Wireless ECU and a smartphone. It presents the operating system, development environments, compilers and programmers (for flashing the ECU) to be used at both ends i.e. for the Wireless ECU and a smartphone. Let's discuss it for each end:

Tools for the Wireless ECU

Likewise all the other ECU's, the software code for the Wireless ECU is being written under **Linux OS (Ubuntu 9.04)**, using high level language **C++**. There is no special integrated development environment (IDE) used for writing the code, instead we are using the simple text editor. The code is compiled using the **GNU Compiler Collection (GCC) g++** compiler [30], using console commands. The programmer used for flashing the code into the flash memory of the AT90CAN128 controller is **AVRISP mkII** [31]. It comes up with a USB interface, for connecting it to the PC or laptop. It can be connected to the AT90CAN128 controller using in-system programmable (ISP) interface. The software driver used for programming the code using AVRISP is **AVRDUDE** [32].

Tools for the smartphone

A smartphone can come up with a different mobile operating system, depending upon the manufacturer. Currently the leading operating systems on the smartphone market are iOS, BlackBerry, Android, Symbian and Windows. Android and iOs lead by far the other available operating systems for the smartphones. As a matter of fact, many manufacturers including;

HTC, Samsung, LG, Motorolla and many more, are building their phones on Android as the mobile operating system. It would not be wrong to say that Android is the most common platform used by majority of the smartphone manufacturers. To cover more range of devices (smartphones), we are going with the Android platform.

Android Inc was founded in California, U.S. in 2003, and later was acquired by Google in 2005. After original release there have been number of updates in the original version of Android. Android is a complete operating environment based upon the Linux® V2.6 kernel, offering a complete set of tools, namely software development kit (SDK), for developing apps for Android devices [33].

For this thesis, the software code for the smartphone is being written under **Windows OS (Windows 7)**, using high level language Java. The integrated development environment (IDE) used for writing the code is **'Eclipse' (v3.7.1)** [34]. The plug-in ***Android Development Tools* (ADT)** is integrated with the Eclipse IDE for providing a powerful, integrated environment for building Android applications [35]. ADT extends the capabilities of Eclipse for setting up new Android projects, creating an application UI, adding packages based on the Android Framework API, debugging applications using the Android SDK tools, and even exporting signed (or unsigned) .apk files in order to distribute application. With the guided project setup it provides, as well as tools integration, custom XML editors, and debug output pane, ADT gives an incredible boost in developing Android applications.

Summary

In this Chapter, we started of explaining the idea of structuring the Wireless ECU into the already existing platform of the Concept Car, whereas a smartphone has been selected as the user-end device. We compared different competing wireless technologies, while following our selection criteria, Bluetooth has been selected as a wireless technology. We discussed selecting the important hardware tools for the Wireless ECU where; RN-42 Bluetooth module has been selected as the Bluetooth device, AT90CAN128 as the micro-controller, PCA82C250 as the CAN driver and MAX1837 as the voltage regulator. Finally, we presented in detail the selected software tools including; IDEs, compilers and programmers, for the Wireless ECU and the smartphone.

Chapter 3

Development

'Getting Things Done'

This chapter proceeds with the directions received from the design phase and starts implementing the design. It starts of explaining the hardware development of the Wireless ECU, by presenting the layout of the Wireless ECU and the Bluetooth board. This phase for the very first time, works with the Bluetooth module, performs configuration and the loopback test. In the second part, the software communication protocol for having a smooth communication between the Wireless ECU and a smartphone is elaborated. Finally, the software components for explaining the software libraries to be used at both ends are presented.

3.1 Hardware Development

This section presents the development of the *Wireless ECU* and the *Bluetooth Board*. The Wireless ECU is incorporated with two simple connectors, for the sake of accommodating two different modules (see Figure 3.1). These connectors provide supply and transceiver signals to the modules. One of the connectors is used for connecting the RN-42 Bluetooth module for the current work, while the other is reserved for connecting any other wireless communication module in future, such as a WiFi module. So the idea is to produce a separate board for the RN-42 Bluetooth device, independent of the Wireless ECU board. Connecting the module to the ECU is as easy as plugging the connections into the connector on the ECU. This ensures safety and ease in troubleshooting the boards.

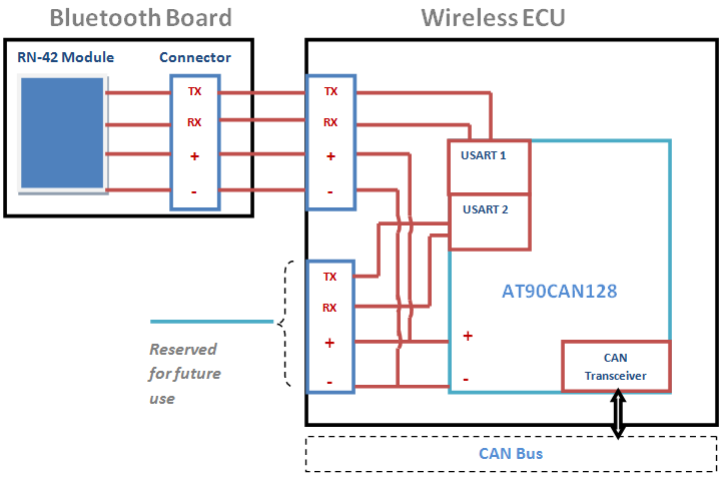


Figure 3.1: The Wireless ECU vs the Bluetooth Board

It is important to note that this Figure does not reveal the complete design of the Wireless ECU, instead it focuses on the connection phenomenon. As discussed in the previous chapter, the RN-42 Bluetooth module comes up with both UART and USB interfaces. For our design, we are using the UART interface for establishing communication between the Wireless ECU and the RN-42 module. Consequently, wireless transmit and receive operations can simply be invoked by invoking UART module of the AT90CAN128 controller. The Bluetooth module is powered from the Wireless ECU via power connections (+ and -), where "+" symbol represents the 5V input, and "-" symbol represents ground. The communication between the module and the ECU

is carried out via transceiver signals (TX and RX), where TX denotes the transmitter, and RX denotes the receiver. Secondly, other connector is reserved for future use. This could be any wireless communication device, that supports UART interface. The Wireless ECU is also connected to the CAN bus via AT90CAN128 built in CAN controller. Let's now discuss the layout of both the independent boards:

3.1.1 Developing the Wireless ECU

This section presents and discusses in detail the hardware development of the Wireless ECU. Likewise all the other ECUs, it is designed and developed using the CadSoft's Eagle PCB design software [36]. The complete layout of the Wireless ECU is shown in Figure *3.2*.

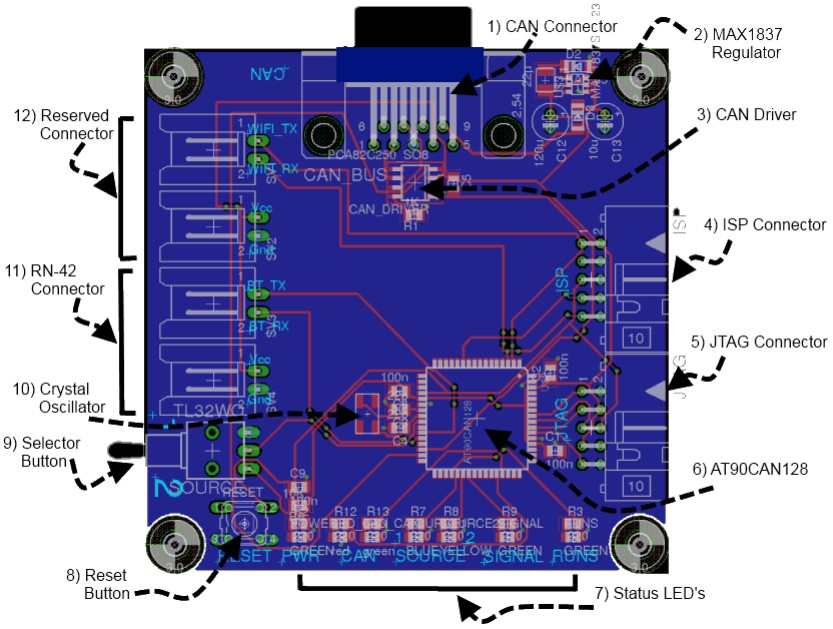


Figure 3.2: Layout Design for the Wireless ECU

The Wireless ECU is comprised of twelve important components: CAN connector (1) is a DB9 serial connector for connecting the Wireless ECU to the CAN interface. It also brings in power supply from a 3S1 LiPo, 11.1 V battery (see Figure 1.2). This voltage is stepped down to 5V by using a MAX1837 regulator (2). The regulated voltage is then used as the power source for all the components on the board. The CAN driver (3) connects

the CAN physical bus to the built in CAN controller of the AT90CAN128 controller (6). The AT90CAN128 controller is the main processing unit that brings in all the desired features to the Wireless ECU. It is clocked from a 16MHz crystal oscillator (10), for executing instructions and driving peripherals. Apparently, there are two ways of programming this controller, either by using an ISP (4) or Joint Test Action Group (JTAG) (5) connector. Likewise all the other ECUs, we are using an ISP compatible hardware programmer i.e. **AVRISP mkII**, therefore the ISP connector is used. For reset operation, a Reset Button (8) is employed. The Wireless ECU incorporates a series of status LED's (7), with each LED indicating a different operation. Starting from right to left, *the RUNS LED* indicates continuous execution of the code. *The SIGNAL LED* indicates the established connection with the RN42 Bluetooth module. *The SOURCE LEDs* represent the mode selected by the user using the selector button (9). This button allows the user to choose between a smartphone and the radio controlled system, as the user end device. Finally, *the PWR LED* indicates that the board is correctly powered. As discussed in the previous section, the Bluetooth module is connected to the Wireless ECU via a separate connector namely the RN42-Connector (11). The ECU also incorporates a reserved connector (12), for bringing in another UART based wireless communication module in the future. The schematic design of the ECU is shown in Figure 3.3:

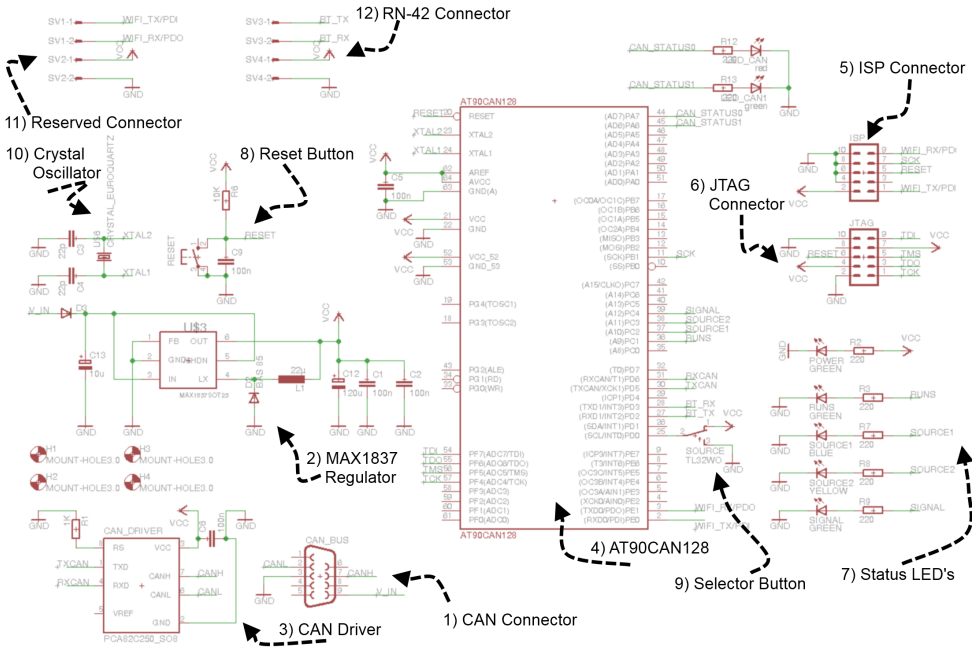


Figure 3.3: Schematic Design for the Wireless ECU

3.1.2 Developing the Bluetooth Board

Building a separate board for the RN-42 Bluetooth module reduces the complexity within the already populated Wireless ECU, eventually bringing in ease in troubleshooting both the boards. Secondly, it avoids re-fabricating the ECU even if the Bluetooth board gets damaged. This board is also developed using the Eagle software for PCB design. The layout for the Bluetooth board is presented in Figure 3.4:

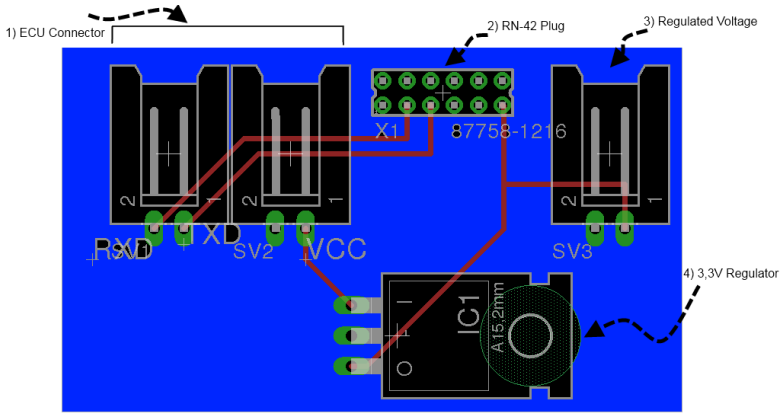


Figure 3.4: Layout Design for the Bluetooth Board

The data sheet of the RN-42 module recommends powering up the device with 3,3V. Starting with the ECU connector (1), it connects the RN-42 module with the *Wireless ECU*. This board comprises of two separate 2-pin connectors, one for power and other for transmission signals. A separate connector namely RN-42 plug (2), connects the module on this board. Again, this ensures that in case if a module is damaged, it can be replaced easily. A 3,3V regulator (4) is employed for stepping down the voltage. For this purpose, a volt regulator is incorporated into the design. Finally, a 2-pin connector at the extreme right (3), provides the regulated voltage as an output, that can be used to power other 3,3V devices like MAX3232 [37] circuit for testing purposes. As shown in Figure 3.5, the schematic version of the Bluetooth board is highlighted with the same components.

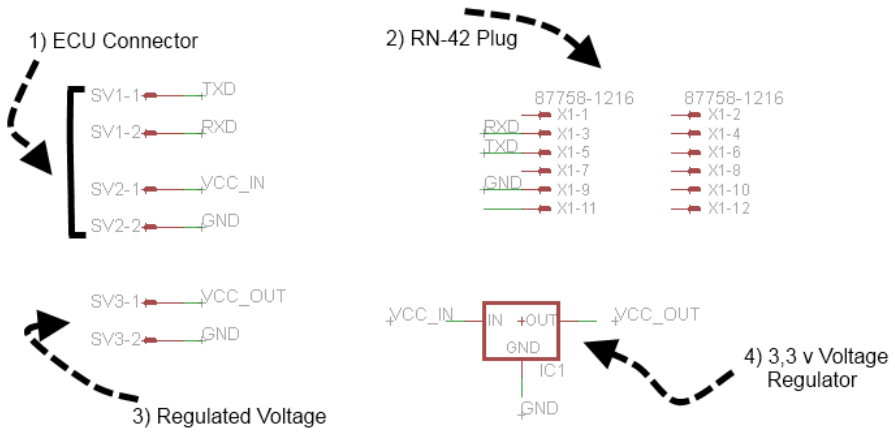


Figure 3.5: Schematic Design for the Bluetooth Board

3.2 Getting Started with the Bluetooth Module

This section presents a *QuickStart* guide to the RN-42 Bluetooth module. The best way of getting started is to configure the device. According to the user manual, it is capable of operating in two different modes namely; *Data Mode* and *Command Mode*. By default, when the device is powered, it starts with the data mode. In order to test and verify that the module is correctly connected, powered and working, we start off configuring the device at first, by entering the command mode. This configuration includes; getting current settings, changing device name, setting up data transfer rate, setting up a profile, leaving command mode etc. In the second part, the loopback test is performed. The idea is to quickly test the module without using any software code.

3.2.1 Configuring the Module

For configuring and programming the RN-42 module for the very first time, we connect it to a PC by using RS232 serial port connections. Since modern PC's only have USB, we are going to use an inexpensive RS232 to USB adapter cable from Pollin [38]. The RN-42 module is a 3,3V transistor-transistor logic (TTL) compatible device [39], whereas RS232 uses different

levels of voltage as defined by the RS232 standard [40]. TTL is a class of digital circuits built from bipolar junction transistors (BJT) and resistors. It is termed as transistor–transistor logic because both the logic gating function and the amplifying function are performed by transistors. To make the signal levels compatible from TTL to RS232 and vice versa, we employ a MAX3232. It is an integrated circuit, operated at 3,3 volts, that converts signals from RS-232 serial port to signals suitable for use in TTL compatible digital logic circuits. It is a dual driver/receiver and typically converts the RX, TX, CTS and RTS signals. The complete connection diagram is shown in Figure 3.6.

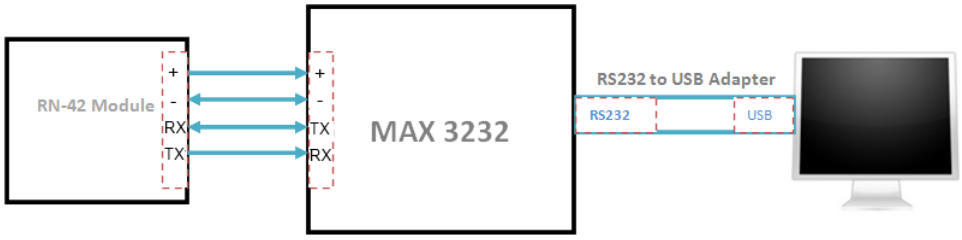


Figure 3.6: Connection Diagram of MAX3232

The MAX3232 is powered from the RN-42 module, from the regulated output voltage connector (see Figure 3.4). It converts the RX and TX signals (TTL level) to RS232, and vice versa. It also reveals the use of RS232 to USB adapter cable. The USB side of the cable is connected to a PC, whereas the RS232 side is connected to a MAX3232. For issuing commands to the RN-42 module, we require a terminal emulator at the PC like; HyperTerminal [41], PuTTY [42], TeraTerm [43] etc. We are going to use TeraTerm for issuing commands serially, to configure our device. For setting up terminal emulators for connecting and communicating with serial devices, I refer the interested reader to the online tutorial at [44].

Before issuing commands from the TeraTerm, it is important to discuss the desired configuration settings. By default the RN-42 module shows up with SPP profile [45], with a baud rate of 115Kbps, no parity, slave mode, default name as FireFly-xxxx (xxxx are the last four nibbles of Bluetooth MAC address) and the pin code is 1234. We are going to change the device's default name to ConceptCar, while staying with the default baud rate unless we find this rate inappropriate for the final applications.

After successfully setting up a connection and keeping in mind the desired settings, we are going to invoke some commands for configuring the device:

Step 1 - Enter Command Mode

Upon power up, the device starts in the data mode. To enter the command mode, send the characters $$$ through the serial port. If the device responds with *CMD*, it shows that the cable and communication settings are correct. While in the command mode, the device will accept ASCII bytes as commands. Valid commands will return an *AOK*, response, and invalid ones will return *ERR*. Commands that are not recognized will return a ?.

```
>$$$
CMD
```

As shown above, the top box presents the typed command ($$$), and the bottom box presents the response received from the device.

Step 2 - Get Current Parameters

A quick check to see if you are in command mode is to type the *D and E* commands after entering the command mode. This will show up the parameters, such as the device name, class of device and serial port settings.

```
>D
***Settings***
BTA=000666430DA8
BTName=FireFly-0DA8
Baudrt=115K
Parity=None
Mode=Slav
Authen=0
Encryp=0
PinCod=1234
Bounded=0
Rem=00143500139B
```

The current device name is FireFly-0DA8. The serial port settings involve; default baud rate = 115K and parity = none. The default mode is slave mode, which corresponds to, this device is not going to be the initiator for establishing connections. Since we need the user to initiate the communication, smartphone is going to be the initiator, so we stay with the same setting for mode.

Step 3 - Change Device's Default Name

As mentioned in Step 2, the default name of the module is FireFly-0DA8. For the sake of verifying that our module is rightly connected and working, we are going to invoke the commands for changing the name of our module to *ConceptCar*.

```
>SN,ConceptCar
AOK
```

Sending the command *SN,ConceptCar*, receives an AOK command in return. This indicates that the command is valid and has been successfully invoked.

Step 4 - Get Updated Parameters

Again invoking the D command brings in the settings again, this time with the changed name i.e. ConceptCar.

```
>D
***Settings***
BTA=000666430DA8
BTName=ConceptCar
Baudrt=115K
Parity=None
Mode=Slav
Authen=0
Encryp=0
PinCod=1234
Bounded=0
Rem=00143500139B
```

It is important to note that the changed settings only come into play after the successfull reboot of the RN-42 module.

Step 5 - Exit Command Mode

After invoking all the desired commands, we exit the command mode by typing - - - (three minus signs). This is verified with the received *END* command on the screen.

```
>—
END
```

The original snapshot of the TeraTerm window while the device was configured is shown in Figure 3.7. It summarizes all the steps as discussed in this section:

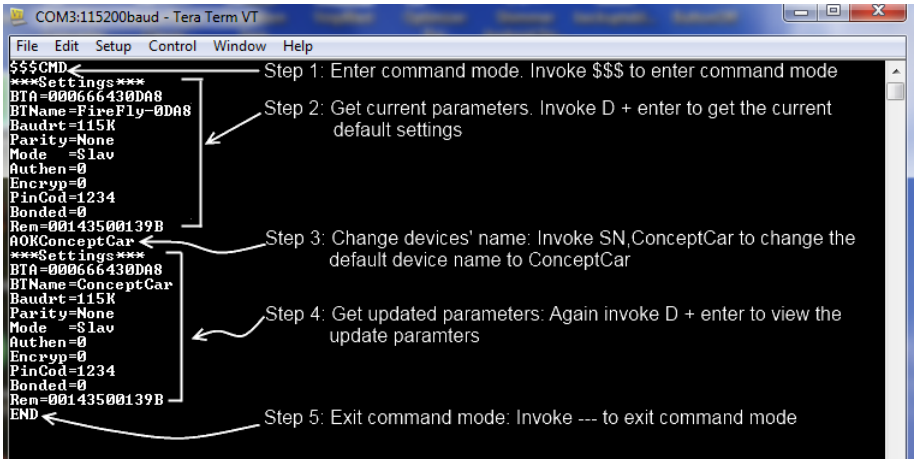


Figure 3.7: Configuring the RN-42 Module

Note that there are a number of ASCI commands for bringing in different features to the RN-42 module. Invoking all the commands is out of scope for this thesis. Refer to the reference manual, to get the complete set of commands. Also the module can directly be configured by using a Bluetooth app on a smartphone, capable of transmitting ASCI bytes over the Bluetooth medium. This avoids the use of the MAX3232 circuitry and the terminal emulator.

3.2.2 RN-42 Loopback Test

After successfully configuring the Bluetooth module, it is the time to establish a quick setup for the wireless communication, without using any piece of software code. This would verify that our Bluetooth transceiver is working properly. For this cause, we are going to implement a *Loopback Test*. Loopback testing incorporates the way of routing electronic signals, digital data streams, or flows of items from the transmitting end to the receiving end of the source without intentional processing or modification [46]. It is primarily intended for performing transmission tests of access lines, thereby testing and verifying the transmission or transportation infrastructure.

As per the definition stated above, we short out the RX and TX lines of the Bluetooth module, as shown in Figure 3.8.

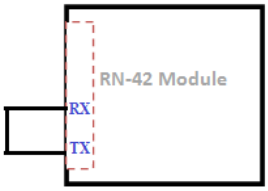


Figure 3.8: Loopback Test for the RN-42 Module

The loopback test can be performed either by using the same method used for configuration, or by using a device capable of Bluetooth communication. Since we already worked with the serial terminal emulator approach, it is the right time to introduce a smartphone for wireless communication testing, for the very first time. We are going to use one of the free Bluetooth communication app, namely *BlueTerm* [47], for sending some data wirelessly to the RN-42 module. Since we binded the RX and TX lines at the RN-42 module, it is expected to receiving an echo of the transmitted byte of data on a smartphone. While sending the set of bytes as ***Hello ConceptCar!***, the results obtained are shown in Figure 3.9.

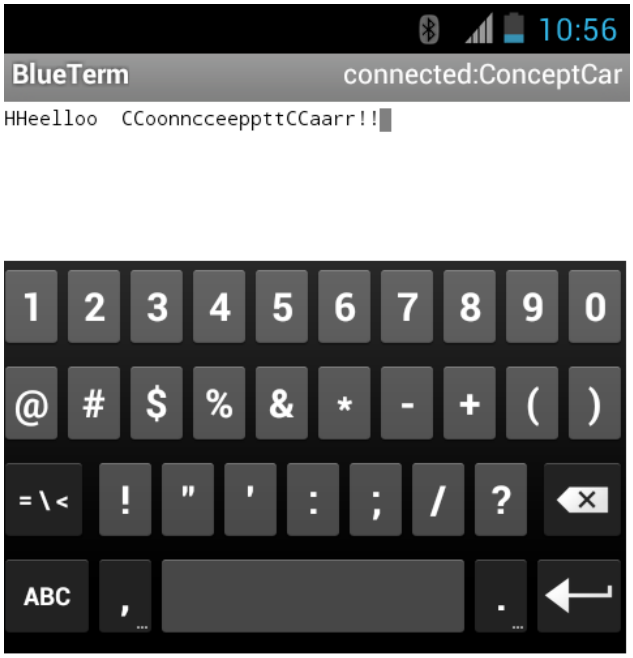


Figure 3.9: Loopback Test Results

The bytes sent from a smartphone, are echoed back and received on the smartphone. The right top of Figure 3.9 reveals the connection to the RN-42

module i.e. ConceptCar. With this, we finally tested and verified that our module is rightly configured and the transceiver is properly working. In the next section we are finally going to develop our software library for establishing the wireless communication with software.

3.3 Software Development

In the previous sections, we successfully developed the Wireless ECU and the Bluetooth board. Furthermore, the Bluetooth module was successfully configured and tested without any software development. In this section, we set up a wireless communication protocol for establishing the wireless communication between the Wireless ECU and the smartphone. The term protocol means forming up an application specific standard data package, that would be followed by both the Wireless ECU and the smartphone. In the second part of this section, the software libraries for managing the wireless communication are discussed. This software library incorporates the components, to be used for the final applications namely; the *Smart Monitor* and the *Smart Controller*.

3.3.1 Setting-Up the Communication Protocol

This section deals with setting up the standard data package for both ends (*the Wireless ECU* and the smartphone). Figure 3.10 presents the standard data package:

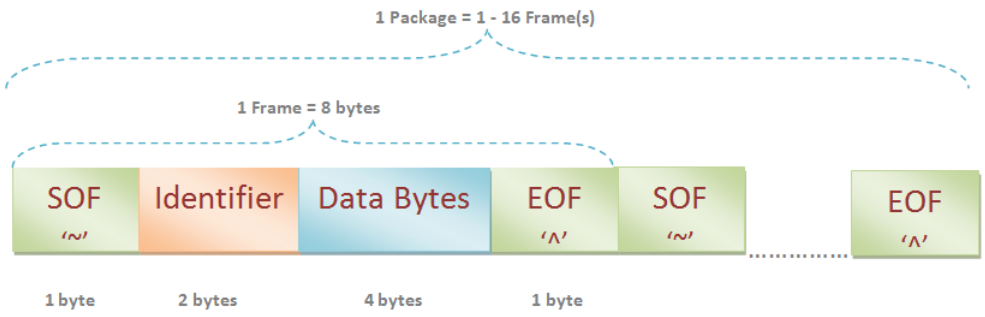


Figure 3.10: The Standard Data Package

The standard data package comprises of at most 16 data frames, where each frame is composed of 8 bytes. Each individual data frame directs to a specific

message. Since it has been successfully tested with 16 frames, we restrict a package to 16 data frames, however, this limit can be increased to a higher number upon further testing. It is important to note that 16 is the maximum number of frames that can be enclosed in a package, however, it can be any number from 1 to 16, depending upon the requirements of the application. Let's discuss each element within a single data frame.

- *SOF:* It is a hard coded character, denoted by '~'. It suggests the beginning of a new frame. It takes one byte (8 bits) of a frame
- *Identifier:* It is a 2 byte code, representing a specific message for example: Throttle data from a smartphone is represented by the identifier 0x102
- *Data bytes:* It takes 4 bytes of a frame, and as the name suggests it carries data from the source as identified by the id
- *EOF:* It is also a hard coded character, denoted by '^'. It suggests the ending of the current frame. It takes one byte of a frame

The Initialization/Acknowledgment Package

As discussed in the previous section, a smartphone acts as the initiator i.e. it establishes the Bluetooth connection with the Concept Car. Though a smartphone establishes the connection, it waits for the Wireless ECU to respond before it starts any communication. The idea of initially synchronizing the devices at both ends leads to the *initialization/acknowledgment* Package. It serves two main purposes:

1. Indicates the connection availability for communication
2. Reveals how many frames would be enclosed inside each package (until the termination of connection), corresponding to the number of messages enclosed in each package

The Package sent from the Wireless ECU to the smartphone is termed as the *Initialization Package*, whereas the package sent as a response to an initialization package from the smartphone to the ECU is termed as the *Acknowledgment Package.* The *number of frames* field denotes the number of messages that will be enclosed inside each package from that end, for the rest of the application. After initiating the connection, smartphone waits for the Concept Car to respond with an initialization package. As soon as it receives the

package, it extracts the package size, sends an acknowledgment package, and immediately starts communication. On the other hand, the Wireless ECU, as soon as it receives the acknowledgment package, extracts the package size, and immediately starts communication. The initialization/acknowledgment package is shown in Figure 3.11:

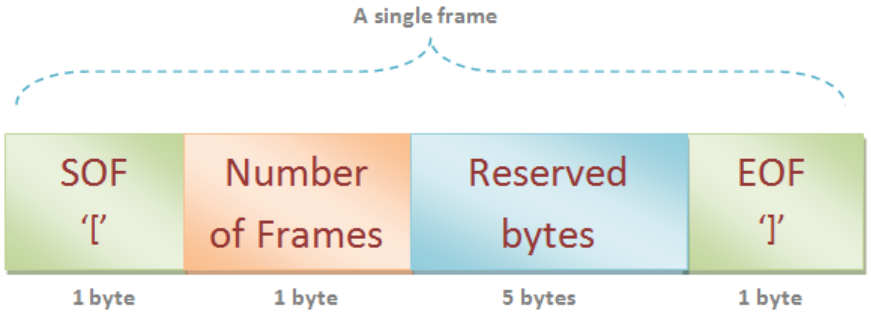


Figure 3.11: The Initialization/Acknowledgment Package

An initialization/acknowledgment package is a special data package comprised of a single data frame, consequently comprising of total 8 bytes. It offers different SOF and EOF fields as compared to a standard data frame. Let's discuss each element of the frame:

- *SOF:* It is a hard coded character, denoted by '['. It suggests the beginning of an initialization/acknowledgment package. It takes one byte (8 bits) of a frame
- *Number of frames:* It represents the package size, corresponding to the total number of frames enclosed inside a package. This package size is used for the rest of the application, until the communication terminates
- *Reserved bytes:* 5 bytes of this frame are reserved for future use
- *EOF:* It is also a hard coded character, denoted by ']'. It suggests the ending of an initialization/acknowledgment package. It takes one byte (8 bits) of a frame

3.3.2 Discovering the Software Library

This section discusses the software library developed for dealing with the wireless communication, accessing sensor namely smartphone's accelerometer, data handling etc. at both ends:

Smartphone: Connect Device

This software component provides the user the list of paired and newly discovered devices. It generally involves user interaction for selecting a Bluetooth device from the list. Upon selection it extracts the address of the device and passes it to the next library component i.e. Bluetooth service.

Smartphone: Bluetooth Service

It accepts the address from the connect device component, and attempts to connect to the specified device. Upon success, it starts a service (a separate thread) for reading and writing data stream from/to the connected Bluetooth device. Upon failure, it prompts the user about the failure.

Smartphone: Data Handler

This component provides the method of encoding and decoding messages in accordance with the standard package frame (as discussed in the previous section). It takes the input stream, decodes it and extracts data and id from the stream. The encoder accepts the tuple (data, id) and encodes it to the standard data package. It also allows logging data in a file on a smartphone.

Smartphone: Accelerometer Handler

It deals with finding, discovering, accessing and monitoring the accelerometer movements, from the smartphone.

Wireless ECU: UART Handler

This component provides complete access to the UART peripheral of the micro-controller. It initializes the UART parameters like; setting up the baud rate and other serial port settings. Note that this baud rate must match the configured rate on the Bluetooth module, otherwise data communication would be faulty. It also provides the UART read and write operations, eventually transmitting and receiving data over the Bluetooth module. It also incorporates the interrupt service routines (ISR's) for the read and write operations. An interrupt service routine is a software routine that hardware invokes in response to an interrupt (event) for example: Timer interrupt executes its ISR upon the completion of the specified time.

Wireless ECU: Package Handler

Package handler combines the feature of the *smartphone: Bluetooth Service* and the *smartphone: Data Handler* as it additionally incorporates the methods of sending/receiving data stream over the RN-42 Bluetooth module. It does that by incorporating the UART handler's read/write methods, after encoding and decoding data in accordance with the standard data package (as discussed in Section 3.3.1).

Wireless ECU: CAN Handler

This component brings in the methods required for accessing the CAN bus. This includes; Initializing the CAN interface, sending/receiving messages to/from the CAN bus, updating the CAN status LED's etc.

Wireless ECU: Inpin, Outpin Handler

This components provides the access to the I/O (input/output) pins on the AT90CAN128 controller. This includes; setting up a specific pin on the controller as an input or an output, placing a high or a low level logic on the pins etc. It is generally used for setting, activating and deactivating the staus LED's on the *Wireless ECU*.

Summary

In this Chapter, we presented the complete hardware development of the Wireless ECU and the Bluetooth, from layout to the practical implementation of the boards. For the very first time, we made the module configured and tested without using a single line of code. For configuration, we used a terminal emulator for issuing commands serially to the Bluetooth module. Also for the very first time, we introduced a smartphone for successfully testing the Bluetooth transceiver, by performing the loopback test. We finally presented the Bluetooth communication protocol (standard data package), and discussed the software library components at both ends.

The snapshot of the developed boards is shown in Figure 3.12.

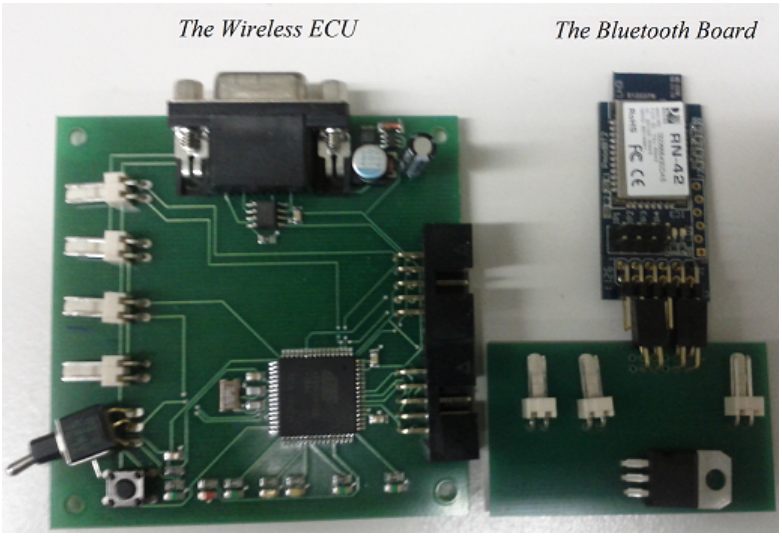


Figure 3.12: The Wireless ECU and the Bluetooth Board

Chapter 4

Integration

'Getting Things Together'

This chapter takes into consideration the hardware and software components built in the development phase, and proceeds with integrating these components together, while performing verification and validation tests. For the very first time, this chapter presents the real implementation of the developed hardware components and software libraries. It brings in the practical implementation of the Bluetooth communication, by presenting a simple test application involving both ends (the Wireless ECU and a smart phone), using the developed software libraries.

4.1 Integrating the Developed Components

In the development phase, we successfully managed and developed the hardware and software components, required for the final applications (Smart Monitor and Smart Controller). Before implementing the final applications, the idea is to; test, verify and validate the components developed in the previous phase. The best way of doing it is to design a simple application involving each end, eventually testing and verifying the software and hardware components. It is important to note that for this test application, the standard data package protocol (as discussed in the previous chapter) is strictly followed, as it is also a part of the software library. In the next section an application designed for testing the developed components is presented.

4.2 The Test Application

In the previous chapter, we successfully configured and tested the Bluetooth module without making the use of the software library created for the Wireless ECU. Similarly, we used a free app namely BlueTerm, without making the use of the software library created for a smartphone. Now, the idea is to design a test application that practically implements, tests and brings in all the components developed in the development phase. Though it is a very simple application comprising of two basic components namely an LED and a button/switch on the Wireless ECU, it utilizes all the software components being developed at both ends (explained in detail in the next section). This application is composed of two parts:

- Toggling the LED on the Wireless ECU by invoking the toggle button on a smartphone
- Getting the status of the selector button of the Wireless ECU on a smartphone

The basic design for this test application is shown in Figure 4.1:

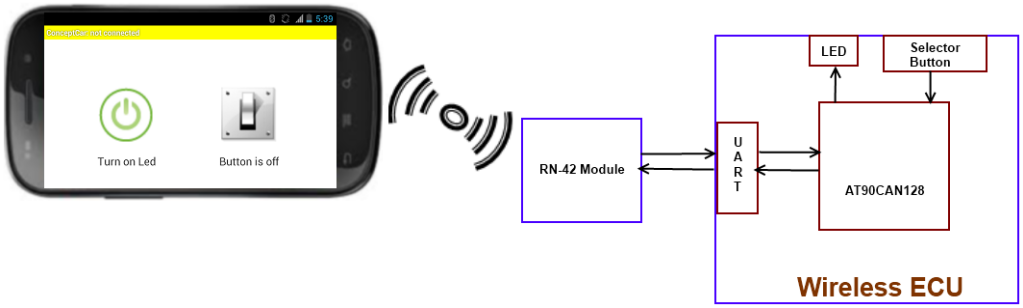


Figure 4.1: The Test App

So it is quite clear now that the LED on the Wireless ECU corresponds to the LED on the user interface (smartphone), and the selector button on the Wireless ECU corresponds to the button on the user interface. Invoking the LED button on the smartphone toggles the LED on the Wireless ECU. Similarly, turning the selector button on or off, changes the status of the button on the smartphone accordingly. Since a smartphone acts as the initiator for establishing the wireless connection, invoking the menu button on the smartphone allows the user to start the connection process.

Figure 4.2: Setting-up the Test App

The procedure for connecting to the Wireless ECU is shown in Figure 4.2. Pressing the menu button brings in the option for connecting to the Concept Car. Invoking the *'Connect to Concept Car'* button as shown in Step 1, switches to an interface as shown in Step 2. This interface allows the user to scan new Bluetooth devices and allows to connect with the desired device. Touching the Concept Car option from the list, attempts to connect to this

device. Upon success, it prompts the user with a toast message *'Connected to Concept Car'*, as shown in Step 3. After the connection has been successfully established, the user can invoke the buttons on the smartphone and on the Wireless ECU.

4.2.1 Analyzing the Data Package for the Test App

Referring to Section 3.3, we discussed about forming the standard data protocol for ensuring that the data received at one end is same as being sent from the other end. We also discussed a special kind of data packages comprising of only one frame namely initialization/acknowledgment package, that ensures initially that both ends are perfectly synchronized in terms of sharing data with each other, once the connection has been established. One of the reasons for presenting this test app is to test and verify this standard data protocol, eventually validating the data sent from one end to the other. This test app requires the transmission of the selector button status (either on or off) from the Wireless ECU, and the LED button status from the smartphone. For the purpose, LED status from the smartphone and button status from the Wireless ECU, is enclosed inside the standard data package. The use of standard data package and an initialization/acknowledgment package is shown in Figure 4.3:

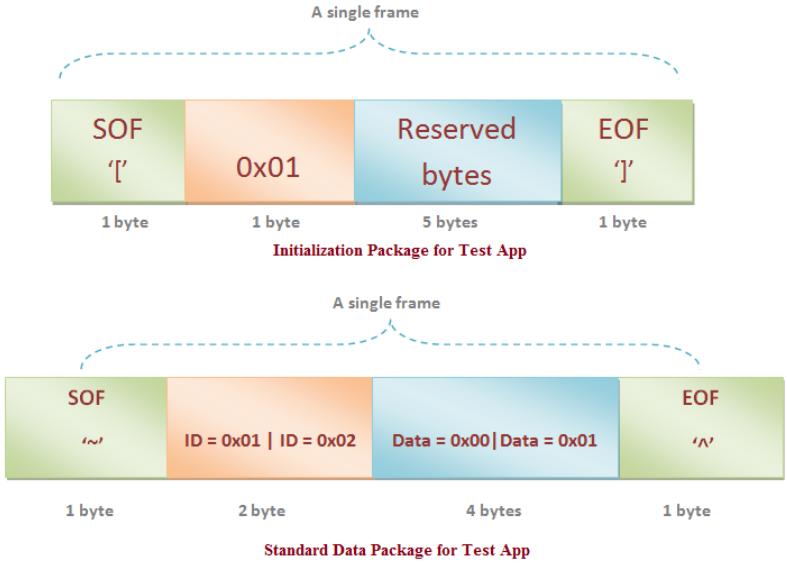


Figure 4.3: The Test App Data Frames

Since from each end, we need to send wirelessly a single message i.e. the LED data from the smartphone, and the button data from the Wireless ECU, the initialization and the acknowledgment package carry 0x01 in the number of frames field. This indicates to the smartphone and the ECU that each data package being sent contains a single frame. As shown in the Figure, the standard data package comprises of only one frame (total 8 bytes). Analyzing the standard data package for the test app, identifier = 0x01 represents the LED message from the smartphone, whereas identifier = 0x02 represents the button status from the Wireless ECU. The data field can be filled with either 0x00 or 0x01. The data = 0x00 represents the *OFF* state, whereas data = 0x01 represents the *ON* state for each control (LED and button). Pressing the LED button on the smartphone, or pressing the selector button on the Wireless ECU changes the state from *ON* to *OFF* and vice versa.

4.2.2 Analyzing the Algorithm at the SmartPhone Side

This section presents the complete algorithm being implemented on the smartphone. The flow diagram of the test app is shown in Figure 4.4.

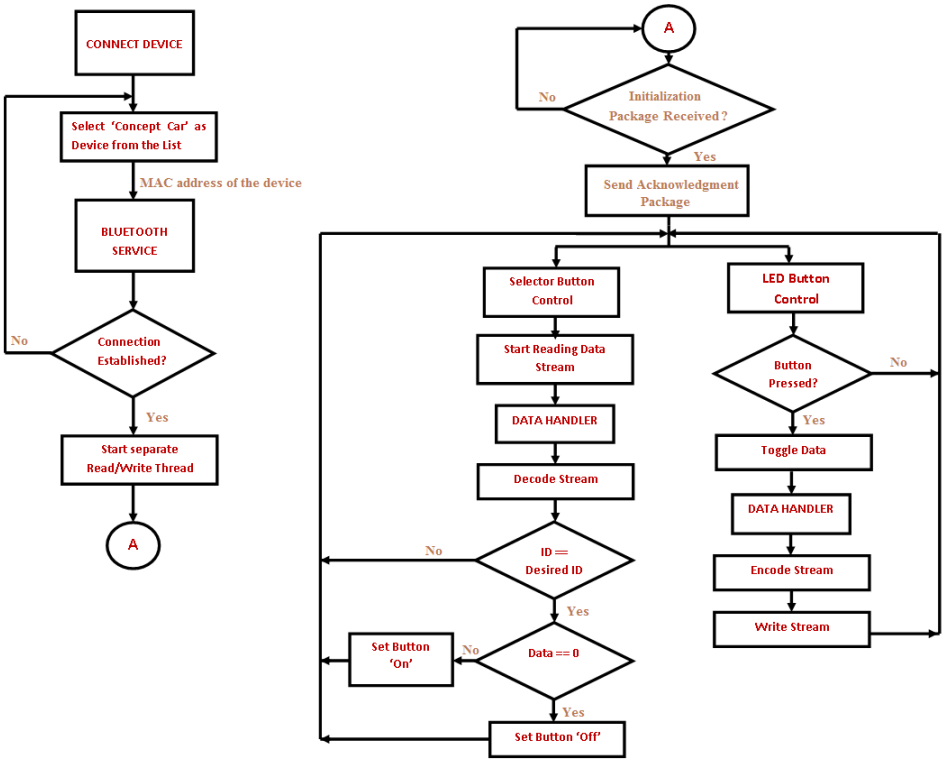


Figure 4.4: The Smartphone Test App Flow Diagram

The Test app starts with the *Connect Device class*, that provides the user a list of all the available devices, eventually selecting the desired device. As the Concept Car is selected from the list, it passes the MAC address to the *Bluetooth Service class*, which attempts to connect with the Concept Car, and upon success it starts a separate thread for reading/writing data stream over the Bluetooth interface. It prompts the user, in case the connection failed. After a successful start of the new thread, initialization package is awaited. As soon as the package is received, it extracts the number of frames field (number of messages to be enclosed inside each package) and sends back an acknowledgment package. This ensures that the connection has been successfully established and both the devices are perfectly synchronized. The user is now allowed to invoke the controls (LED button and the selector button) on the smartphone, and on the Wireless ECU. Pressing the LED button, toggles the LED data and passes this data to the *Data Handler class*. The Data Handler class encodes the provided data along with the identifier (ID = 0x01 for the LED), in accordance with the standard data package. This data package is then transmitted over the Bluetooth interface using the Bluetooth Service write routine. Similarly, the selector button status is received in accordance with the standard data package from the Wireless ECU. This time the Data Handler class accepts the package, decodes it, and finally extracts the identifier and data from it. If the identifier matches the button's identifier i.e. ID = 0x02, it changes the state of the button control on the smartphone, as directed by the data. Since this app only accepts the data enclosed inside a standard data package, therefore it ignores any corrupt form of data.

4.2.3 Analyzing the Algorithm at the Wireless ECU Side

This section presents the complete algorithm being implemented on the Wireless ECU. The flow diagram of the test app is shown in Figure 4.5. The test app starts with sending the initialization package continuously until the smartphone acknowledges the connection, by sending an acknowledgment package. As soon as it receives the acknowledgment package, the number of frames field is extracted. This denotes the number of messages (frames) enclosed inside each package sent from the smartphone, for the rest of the application.

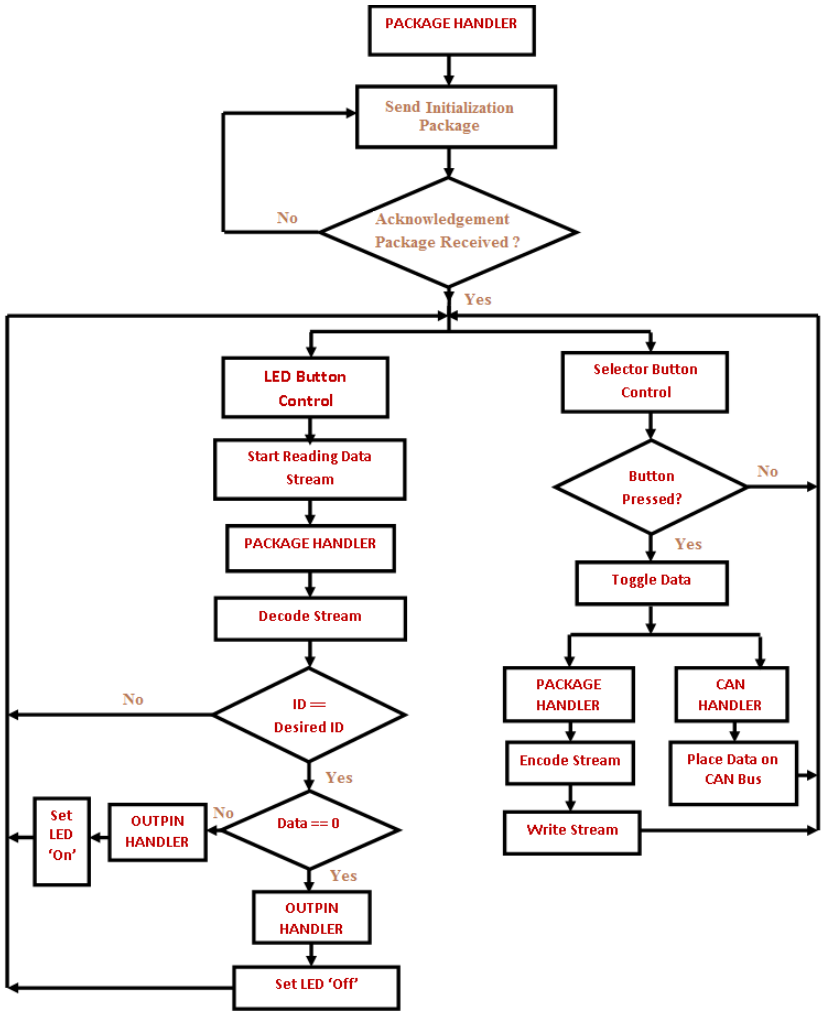


Figure 4.5: The Wireless ECU Test App Flow Diagram

The user can now invoke the controls (LED button and the selector button) on the smartphone, and on the Wireless ECU. Pressing the selector button, toggles the button status and passes this data to the *Package Handler class* and the *CAN Handler class*. The Package Handler class encodes the provided data along with the identifier (ID = 0x02 for the button), in accordance with the standard data package, as shown in Figure 4.3. This data package is then transmitted over the Bluetooth interface, whereas the CAN Handler class places the same data over the CAN bus. Similarly, LED data is received in accordance with the standard data package from the smartphone. This time the Package Handler class accepts the package, decodes it, and finally extracts the identifier and data from it. If the identifier matches the LED

identifier i.e. ID = 0x01, it changes the state of the LED on the Wireless ECU, as directed by the data, by using the *Outpin Handler class*. Since this app only accepts the data enclosed inside a standard data package, therefore it ignores any corrupt form of data.

4.2.4 The Test App Results

The algorithm discussed in the previous sub-sections, has been successfully tested with the expected results. As discussed in the beginning of this section, the test app comprises of two main tasks. The first task is to toggle the LED on the Wireless ECU, by invoking the LED button on the smartphone. The second task is getting the status of the selector button of the Wireless ECU, on the smartphone. The results collected, while the controls have been invoked, are shown in Figure 4.6:

Figure 4.6: The Test App Results

Pressing the LED button on the smartphone and pressing the selector button on the Wireless ECU changes the states of the controls. With this, we also validate the data being sent from one point to other. This successfully verifies that the developed hardware and the software libraries operate as expected. With these results, we can now move towards the implementation of our final applications namely the *Smart monitor and* the *Smart controller*, without being any more concerned about the hardware and software components.

Chapter 5

Deployment and Delivery

'Gaining Desired Results'

This chapter finally presents the solution to the problems (as discussed in Chapter 1) by implementing the final applications, using the hardware tools and software libraries designed, developed, tested and verified in the previous phases. The idea of monitoring the car internals lively, and making the car controllable independent of the radio transmitter system, has been practically implemented, by developing two separate applications.

5.1 Implementing the Solution

Before presenting the final applications, let's recall the problems under consideration and their devised solutions. This thesis is subjected to solve two main problems, on the existing platform of the Concept Car:

1. Monitoring the car internals: *Lively* monitoring the car internals on the fly, while the car is being driven

2. Driving the car independent of the radio transmitter system: Driving the Concept Car over shorter range of distances for testing purposes, by using a user end device (a smartphone), as an alternative to the long range radio transmitter system

The devised solution for both the problems is getting a wireless access over the centralized CAN bus. For this purpose, we added two important tools:

1. A separate ECU incorporating wireless communication i.e. introduction of the Wireless ECU

2. A software tool exploiting the special hardware features, at the smartphone side

Based on these devised solutions, in the previous phases; we designed, developed, integrated and tested all the hardware and software components required to realize a wireless access to the Concept Car. Although a wireless path of interaction with the Concept Car has been successfully established, but still it offers no use unless some real time applications are implemented, that solve the discussed problems. In this final phase, the final two applications that utilize all the hardware and software components, developed and tested in the previous phases, are presented. Each application presents the solution to the respective area of problem, for instance, the *Smart Monitor* targets the problem of monitoring the car internals, and the *Smart Controller* targets the problem of driving the car independent of the transmitter system. Practically, both of these applications are implemented as two parts of the same application, at both ends (the Wireless ECU and the smartphone). Consequently, at the smartphone, both applications share the same startup interface (discussed in detail in the next section).

5.2 The Startup Interface and the Connection Mechanism

Before going deep into the individual applications, it is important to present the startup interface and the connection mechanism (connecting to the Concept Car), as it is shared by both the applications. Starting with the startup screen (as shown in Figure 5.1), it comprises of both the applications namely the Smart Monitor and the Smart Controller. The color and the text of the header indicates the connection status. The user can select any of the application from the startup screen, only once the device has been connected to the Concept Car. The application does not proceed further until the smartphone is connected to the Concept Car.

Figure 5.1: The Startup Interface

The connection mechanism corresponds to connecting with the Concept Car. There are two different ways of connecting with the device, either manually or automatically. Starting with the manual connection, this mechanism is as simple as going through couple of screens (as shown in Figure 5.2). Going through from step 1 to step 3 serves the purpose.

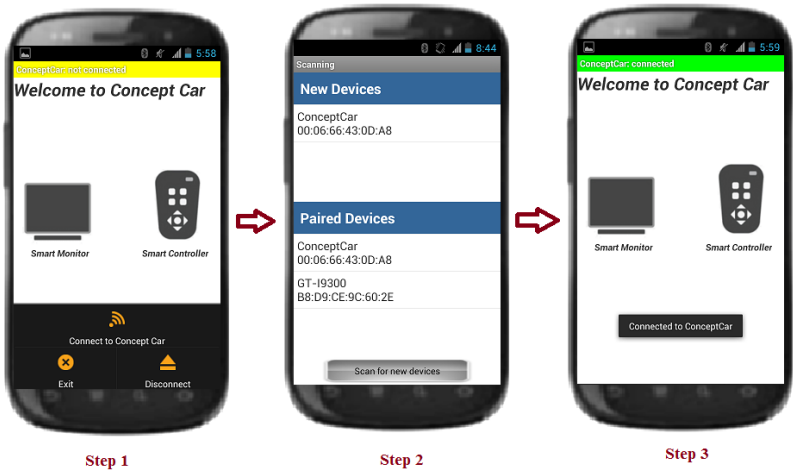


Figure 5.2: The Manual Connection Mechanism

Pressing the menu button brings in the option for connecting to the device, as shown in Step 1. Touching the *Connect to Concept Car* option on the menu bar displays the next interface as shown in Step 2. This interface presents the already paired devices and the list of newly discovered devices. As soon as the desired device is selected, it switches back to the startup screen while attempting to connect with the selected device. Upon success a toast message (*Connected to ConceptCar*) pops up, and the header turns green with the text *ConceptCar: Connected*, as shown in Step 3. Upon failure a toast message (*Unable to connect to device*) appears, and the header remains unchanged. A connected device can be disconnected at any time by invoking the *Disconnect* option in the menu bar.

Introducing the Smart Key

As discussed in Chapter 2, the NFC technology can be possibly used for bootstrapping more capable wireless standards like Bluetooth, WiFi etc. One idea could be connecting to a Bluetooth/WiFi device, once the NFC device finds the desired tag. The idea of connecting to a Bluetooth device using the NFC technology leads to the *Automatic Connection Mechanism*. An NFC tag written with a plain text *ConceptCar* is installed downside on the front guard of the Concept Car. Bringing the smartphone in contact with the written tag, automatically attempts to connect with the Concept Car. This automatic way of connection, avoids going through multiple screens (as in case of manual connection mechanism), and is termed as the *Smart Key* (as shown in Figure 5.3).

Figure 5.3: The Smart Key

As soon as the smartphone is brought in contact with the programmed tag, the Smart Key attempts to connect with the Concept Car. Upon success a toast message (*Connected to ConceptCar*) pops up, and the header turns green with the text *ConceptCar: Connected*. Upon failure a toast message (*Unable to connect to device*) appears, and the header remains unchanged.

5.3 Analyzing the Initialization/Acknowledgment Package

It is important to note that the Smart Monitor and the Smart Controller are implemented as two parts of the same application, at both ends (at the smartphone and the Wireless ECU). As discussed in Chapter 3, the initialization/acknowledgment package apart from initially synchronizing both ends, also presents the package size (frames enclosed inside each package) for the rest of the application, from each end. Therefore, as soon as the connection is initiated from the smartphone, it is important to convey the number of messages enclosed inside each package, from each end. Since the Smart Monitor's app is tested with sixteen frames (sixteen different CAN messages), the Wireless ECU sends the initialization frame with sixteen in the *number of frames* field. With this the smartphone realizes that each package contains

sixteen frames. Similarly, for the Smart Controller, we need three messages (steering, throttle and emergency) enclosed inside each package, sent from the smartphone to the Wireless ECU. Therefore, in response to the initialization package, the smartphone sends an acknowledgment package with three in the *number of frames* field. The initialization/acknowledgment package as sent from each end is shown in Figure 5.4:

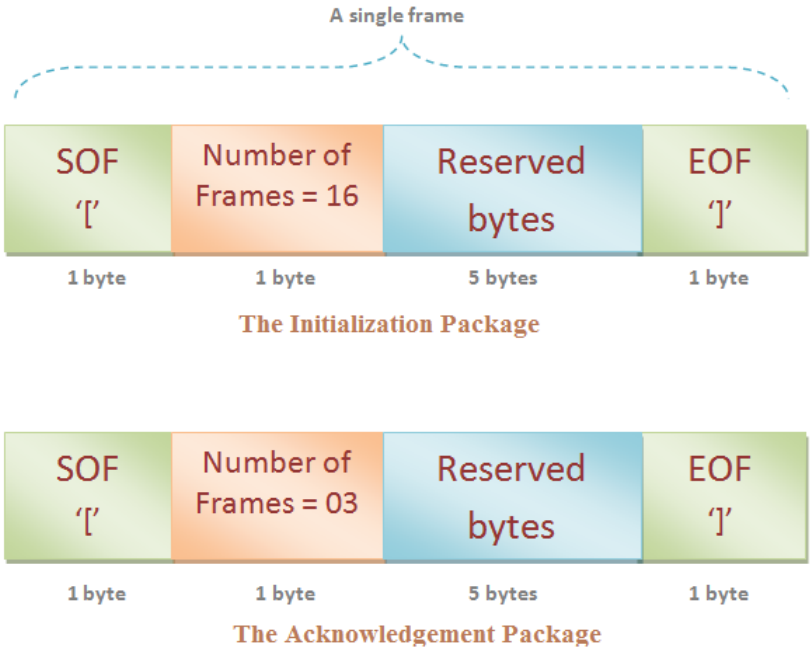


Figure 5.4: The Initialization/Acknowledgment Package

5.4 The Concept Car's Smart Monitor

As the name suggests, this app specifically targets the first problem i.e. lively monitoring the car internals on the fly. As discussed in Chapter 1, there exists two modes pertaining to monitoring the car internals, namely the *Online Mode* (uses the wired CAN viewer hardware) for the static car, and the *Offline Mode* uses the secure digital card (SD card). The idea of implementing the Smart Monitor is to combine the features of both the modes, thereby allowing the user to monitor the parameters of the car lively while it is being driven, or later using the SD card or internal memory of the smartphone. Before the design of this app is further elaborated, let's discuss the already existing online and offline modes of monitoring:

Online Mode Using the CAN Viewer

The online mode uses a USB based CAN adapter namely Tiny-CAN [48]. The USB interface connects the Concept Car to the PC or laptop. Since it uses the wired connection, it is only used when the car is static or driven while suspended. Once connected to the PC, data over the CAN bus can be monitored using the user interface as shown in Figure 5.5.

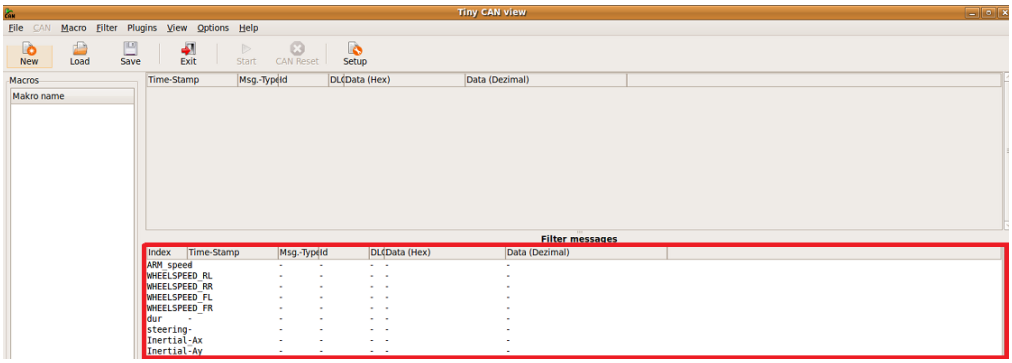


Figure 5.5: The Online Monitoring Mode

As highlighted, using the CAN viewer, the user can filter out undesired messages, so as to receive the messages only with the desired identifiers. This interface displays; the timestamps, CAN message type (either standard or extended), id, data length and data.

Offline Mode Using SD Card

For the offline mode, one of the SensorBoard namely InertialBoard writes data to a FAT16-formatted SD card that is inserted into the card reader slot. When the board is booted and a valid card is detected, a new file is created and data from all messages on the CAN bus is written into this file. There is no mechanism to specifically close the log file except remove the card or power down the board. There are two status LED's on the board that specifically indicate the logging status. The LED *LOGGER ON* (while in the ON state) indicates the SD card is present (it has been detected) and the file could be opened for writing. The LED *LOGGER WRITE* blinks once each time a 512 bytes sector is dumped to the SD card. It is a good indicator if logging actually works. For each CAN message the following data fragment is written, as shown in Figure 5.6:

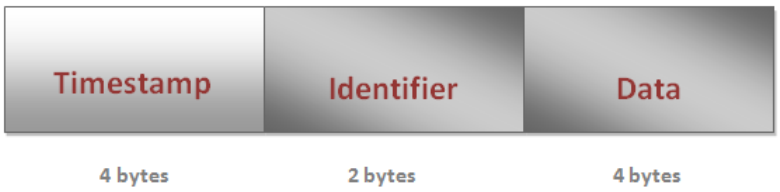


Figure 5.6: SD Card Log Data Format

The timestamps field holds the time in ms since the board was booted up. The remainder of the data fragment is the CAN message's id and content. For each received CAN message, a new fragment of this sort is appended to the log file. Since there is no mechanism for explicitly closing the log file, the last message fragment in the log file is usually cut off. The data is stored to a newly created file with the naming scheme LOG<nr>.CAN. The <nr> part is a consecutive number to name all files differently and not to overwrite an older log. An internal counter increments each time a new log file is written, and the counter state is conserved in EEPROM over reboots. Reflashing the board resets the counter to 0. A specimen of a logged file, as stored in the SD card according to the described format is shown in Figure 5.7.

```
1  Timestamp:Id:Data
2  500:16:2061
3  500:17:4294936427
4  500:18:137
5  500:8:63
6  500:9:28007
7  500:10:22171
8  500:11:16194
9  500:34:1904
10 500:37:1982
11 500:32:1
12 1289:16:1893
13 1289:17:2043
14 1289:18:136
15 1290:16:2251
16 1290:17:1978
17 1290:18:137
18 1290:9:23325
19 1290:10:21903
20 1290:11:15323
21 1293:16:1868
22 1293:17:2069
23 1293:18:136
24 1293:8:63
25 1295:16:2147
26 1295:17:2059
27 1295:18:138
28 1296:16:2226
29 1296:17:1904
30 1296:18:138
```

Figure 5.7: The Offline Mode Logged File

5.4.1 The Smart Monitor's Design

As discussed, this app is specifically designed with the features of the already existing online and offline modes, while adding new values to them. Starting with the online mode, it allows the user to monitor the Concept Car in real time only under the condition the car stays static. This is mainly due to the wired connectivity of the Tiny-CAN hardware. The Smart Monitor overcomes this restriction as it incorporates the wireless connectivity. The user can lively monitor all the parameters of the car on the fly, while the car is being driven. Additionally, this app allows the user to store all the desired data (internals of the Concept Car) on the SD card or internal memory of the smartphone, thereby provides the feature of the offline mode. Let's now discuss how these features are designed and implemented within the Smart Monitor's app.

Once the smartphone is connected to the Concept Car (either manually or using Smart Key), invoking the Smart Monitor control on the startup screen opens the Smart Monitor interface, as shown in Figure 5.8.

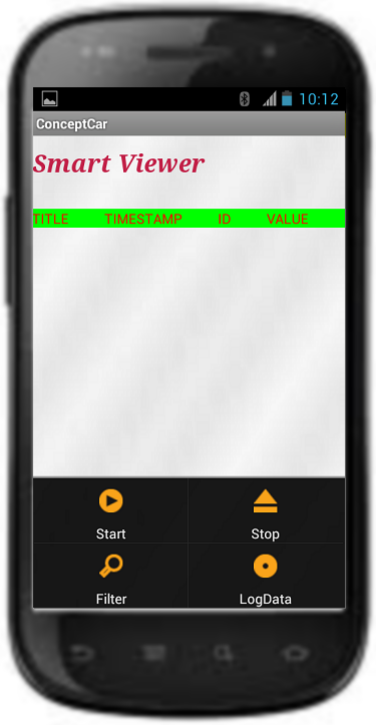


Figure 5.8: The Smart Monitor's Interface

Pressing the menu button brings in four different options. Starting with the Start option, the Smart Monitor begins to collect and display data from the received stream, once this option has been invoked. The Stop option, as the name suggests, allows the user to stop displaying the messages at any point in time. The Filter, as the name suggests, allows the user to filter out the messages that are not desired, so as to receive only the messages with the desired identifiers. It allows the user to select messages from the default list and add new messages. Invoking this option displays a dialog box as shown in Figure 5.9.

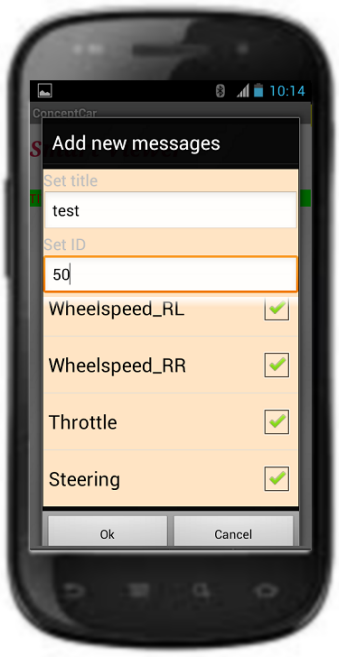


Figure 5.9: The Smart Monitor's Filter

This dialog box is programmed with a list of important messages, with each message having its own unique id. The user can select any of the default messages by simply ticking out the box against the desired message. The user can also add a new message with a unique id, by typing in the title and id on the text boxes provided at the top. Note that this unique id must match one of the CAN id's of the message sent from the Wireless ECU. Once done, pressing the OK button switches back to the Smart Monitor interface, this time, with the newly added messages, as shown in Figure 5.10. This is the same feature as provided by the user interface of the online mode (Tiny CAN viewer).

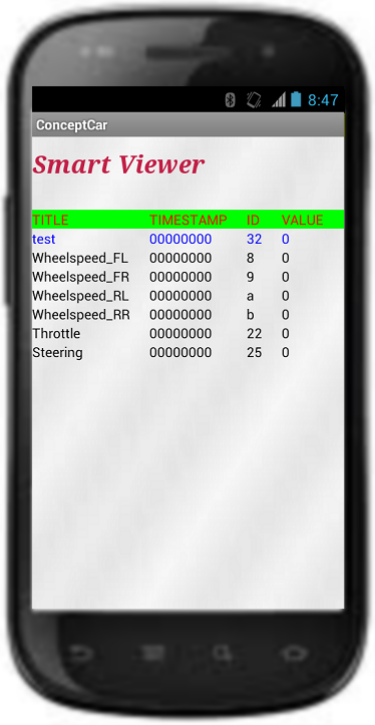


Figure 5.10: The Filter's Newly Added Messages

As shown in the figure above, each message is displayed with; a title (entered in the dialog box), a timestamps indicating the time in ms since the Start option is invoked, the unique id (entered in the dialog box) and the data received against the id. The order with which the Start and Filter option is invoked, is unimportant.

Last but not least, the LogData option from the menu bar (as shown in Figure 5.8), allows the user to log all the added messages in a file on the smartphone. The logged file is stored as a Log.file in the internal memory/SD card of the smartphone. This data logger exactly follows the same format as discussed in the previous section (see Figure 5.6). However unlike the offline mode, it overwrites the previously created log file. It is recommended, therefore, to rename the already stored file before logging the data again. The data logger once started, continues to log until the termination of the Smart Monitor's app. The results obtained are discussed in Section 5.4.4.

5.4.2 Analyzing the Implementation at the Smartphone Side

This section elaborates the implementation of the Smart Monitor by presenting the flow diagram and the code examples of the app, at the smartphone side. Let's start with discussing the flow diagram of the app, as shown in Figure 5.11:

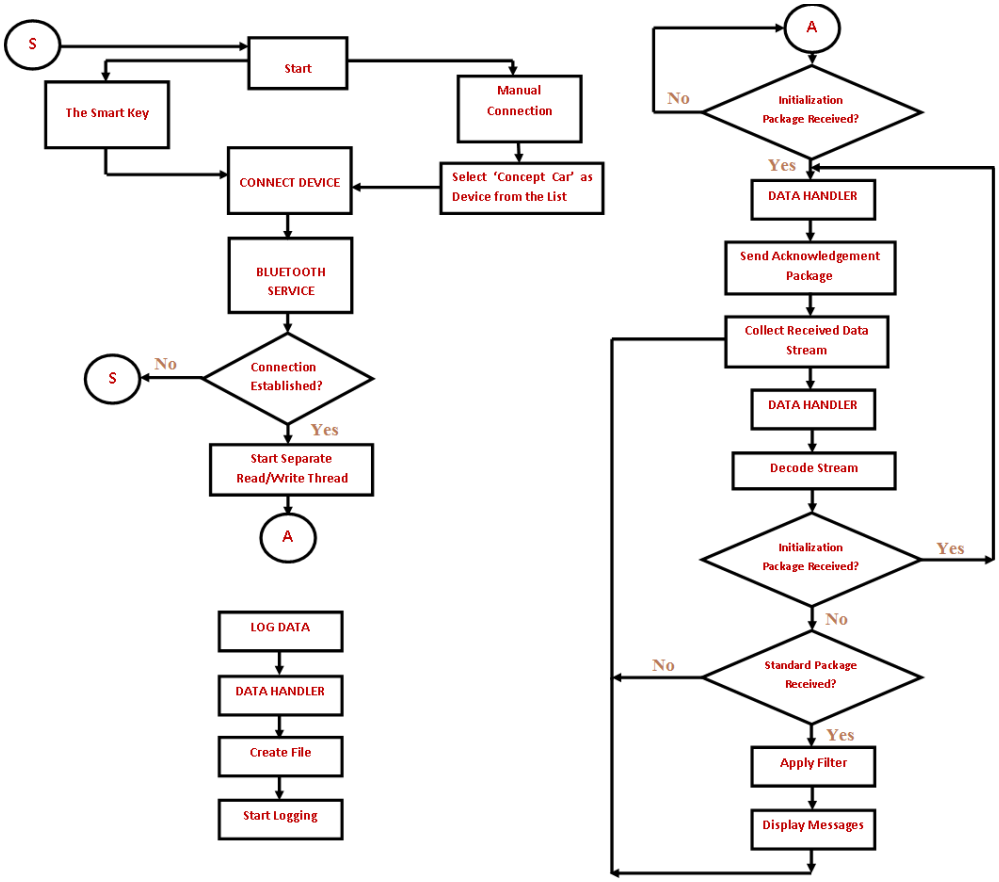


Figure 5.11: The Smart Monitor's Flow Diagram at the Smartphone Side

The app starts with initiating the connection from the smartphone, either manually or using the *Smart Key*. In either case, the name of the device is used by the *Connect Device class*, that extracts the mac address and passes it to the *Bluetooth Service class*. This class attempts to connect by using the provided mac address. Upon success, prompts the user with a toast message, and creates a thread for reading/writing data stream over the Bluetooth interface. It also prompts the user, in case the connection failed. After a

successful start of the new thread, an initialization package is awaited. As soon as the package is received, the *Data Handler class* extracts the number of frames field (number of messages enclosed inside each package), and sends back an acknowledgment package. The initialization/acknowledgment packages are sent as discussed in Section 5.3. This ensures that the connection has been successfully established and both the devices are perfectly synchronized. With this it now starts to collect the received stream from the *Bluetooth Service class*, and passes it to the *Data Handler class*, that decodes the tuple (id, data) from the stream. The messages with the received identifiers are compared against the identifiers set in the filter, and the matched ones are displayed on the interface. The rest are said to be filtered out. It continues to collect, decode, filter and display messages until the termination of the connection. The user at any point in time, using the menu bar, can invoke the *LogData* feature. Once invoked, the *Data Handler class* creates a file and continues to log data until the termination of this app. Since this app only accepts the data enclosed inside a standard data package, therefore it ignores any corrupt form of data.

Code Snippets

Let's now present some of the important methods of the library implemented in different classes, responsible for initiating connection, data handling and data streaming:

There exist two versions of connectDevice method that are responsible for initiating connections. One of the method uses the manual method as:

```
1  /** Method that connects the Concept car
2  * @param data Mac address received from the
       ConnectDevice class*/
3  private void connectDevice(Intent data) {
4    mConnectionSource = false;
5    // Get the device MAC address
6    String address = data.getExtras()
7         .getString(connectdevice.EXTRA_DEVICE_ADDRESS);
8    // Get the BluetoothDevice object
9    BluetoothDevice device = mBtAdapter.getRemoteDevice(
       address);
10   // Attempt to connect to the device
11   mBtService.connect(device);
12 }
```

In line

- 6) the Mac address from the Connect Device class is retrieved, in the user understandable string format
- 9) the Bluetooth object is requested for the Mac address received in the previous step
- 11) finally, the device object is passed to the Bluetooth Service class, that attempts to connect with the device

The next method is called only when an NFC event occurs.

```
1
2  /**
3  * Method that connects directly to Concept Car, using
4  * the Mac address received from the NFC event
5  * @param address
6  */
7  private void connectDevice(String address) {
8    mConnectionSource = true;
9    // Get the BluetoothDevice object
10   final BluetoothDevice device = mBtAdapter.
       getRemoteDevice(address);
11   //since a UI component has to be accessed, it should
       run on UI thread
12   if (!connectionPending) {
13     connectionPending = true;
14     runOnUiThread(new Runnable() {
15       public void run() {
16         mConnectionProgress =
17         ProgressDialog.show(ConceptCar.this, "",
18                   "Connecting to Concept Car");
19         //Attempt to connect to the device
20         mBtService.connect(device);
21          }
22     });
23
24   }
25
26 }
```

In line

- 10) the Bluetooth object is requested based on the address received from the NFC event
- 14) executes a separate thread on the user interface thread, as a progress dialog is to be initiated
- 16) a progress dialog is initiated indicating the connection status
- 20) meanwhile the connection method is called

The decoder method that extracts the tuple (id, data) from the received stream, is implemented in the Data Handler class as:

```
1  /**
2  * Extracts the tuple (id, data) from the stream
3  */
4  public int decodePackage(byte[] readbuf, int frmcnt) {
5    byte j = 0;
6    byte k = 0;
7    if(readbuf[0] == '[' && readbuf[7] == ']' &&
       framecount > 0) {
8      return -1;
9    }
10   for(int i = 0; i<framecount; i++,j+=8) {
11     if(readbuf[j] == '~' && readbuf[j+7] == '^') {
12       mId[k] = (short) ((short) readbuf[j+1] |
13                         (short) readbuf[j+2]<<8);
14       mMessage[k] =
15         (int) ((int) (readbuf[j+3] & 0xFF)|
16         (int) ((readbuf[j+4]&0xFF)<< 8) |
17         (int) ((readbuf[j+5] & 0xFF) << 16)|
18         (int) ((readbuf[j+6] & 0xFF) << 32));
19       mCurrentTime = System.currentTimeMillis()/1000;
20       mTimeStamp[k] = mCurrentTime.toString();
21       k++;
22     }
23   }
24   mMessageCount = k;
25   j=0;
26   return k;
27 }
```

In line

- 7) this if statement keeps track of the initialization package
- 10) extracts all the messages from the package along with their identifiers
- 19) calculates the timestamps in ms
- 26) returns total number of messages received

The updateLog method is implemented inside the Data Handler class, and is responsible for writing the messages on the SD card or ROM:

```
1  /**
2  * Updates the log file with the newly received messages
3  * @param islogenabled if the LogData control is invoked
4  */
5  public void updateLog(boolean islogenabled) {
6    if(mFileWriter != null) {
7      try {
8        if((islogenabled) &&
9          (!mCurrentTime.equals(mPreviousTime))) {
10         for(int i = 0; i<mMessageCount; i++) {
11           mLogText =
12            mTimeStamp[i] + ":" + HexString((mId[i]))
13            + ":" + String(mMessage[i]) + "\n";
14           mFileWriter.write(mLogText);
15           mPreviousTime = mCurrentTime;
16                 }
17           }
18         }
19         catch (IOException e) {
20           // TODO Auto-generated catch block
21           e.printStackTrace();
22         }
23   }
24 }
```

In line

- 6) this if statement checks if the file is already created

- 8) this if statement checks if the user has invoked the LogData option from the menu bar
- 10) this for loop is responsible for writing the file according the defined format

The Bluetooth Service class is held responsible for streaming data in and out from the Bluetooth interface. Below is the part of the code, that receives the data stream:

```
1  // Keep listening to the InputStream while connected
2  while (true) {
3    try {
4      if(buffer[0] == '[' && buffer[7] == ']') {
5            ackData = mDataHandler.acknowledgePackage(
                 MESSAGE_COUNT);
6            WritetoDevice(ackData);
7      }
8      dinput.readFully(buffer, 0, num_of_bytes);
9      mHandler.obtainMessage(ConceptCar.MESSAGE_READ,
10           num_of_bytes, -1, buffer).sendToTarget();
11     } catch (IOException e) {
12         Log.e(TAG, "disconnected", e);
13         connectionLost();
14         break;
15     }
16 }
```

In line

- 2) loop until the end of application or disconnection
- 4) this if statement keeps track of the initialization package
- 5) get acknowledgment package from DataHandler
- 6) method that sends the acknowledgment package
- 8) receives the data bytes as indicated by the initialization package
- 13) this method informs the handler, if the connection is lost
- 14) finishes the current thread

5.4.3 Analyzing the Implementation at the Wireless ECU Side

This section elaborates the implementation of the Smart Monitor by presenting the flow diagram and the code examples of the app, at the Wireless ECU side. Let's start with discussing the flow diagram of the app, as shown in Figure 5.12:

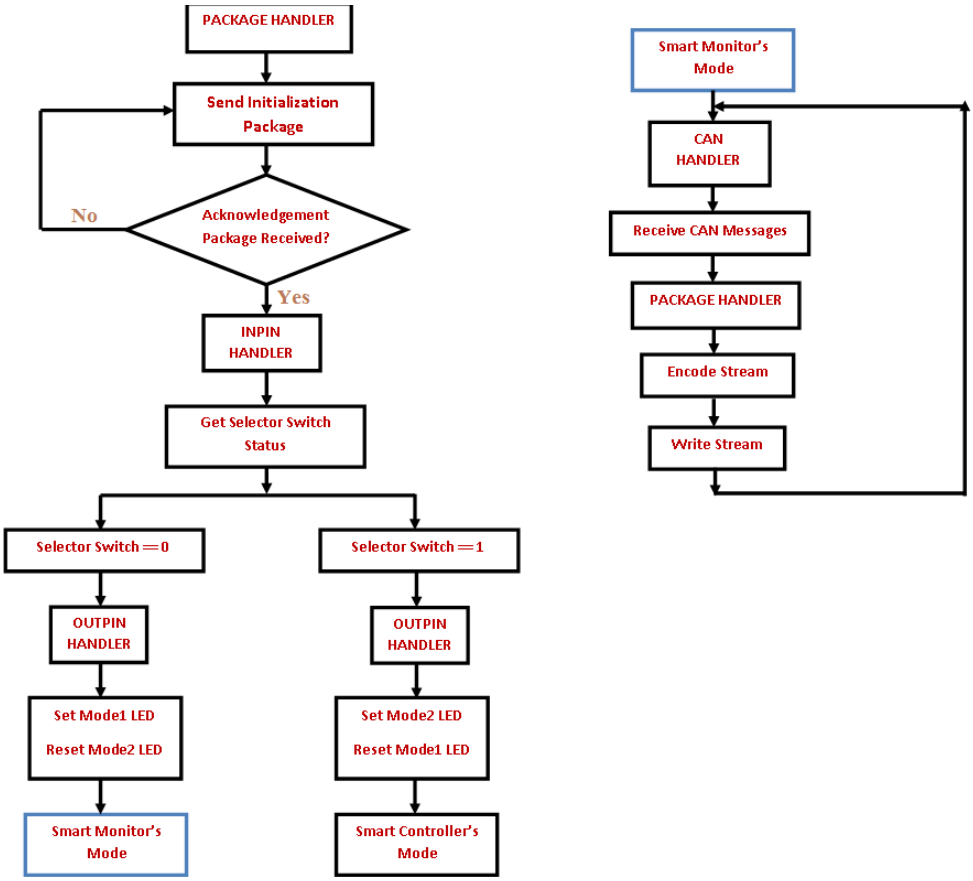


Figure 5.12: The Smart Monitor's Flow Diagram at the ECU Side

At the Wireless ECU, the Smart Monitor and the Smart Controller are implemented as two different modes within the same application. At any point in time, the user can switch between the modes by using the selector switch. In this section the Smart Monitor's mode implemented at the ECU is discussed. The app starts with the *Package Handler class* sending the initialization package, until the acknowledgment package is received. As soon as it receives the acknowledgment package, extracts the *number of frames* field,

and the *Inpin Handler class* gets the status of the selector button. This status is also sent to the CAN bus using the *CAN Handler class*. The initialization/acknowledgment packages are sent as discussed in Section 5.3. If the selector switch provides a logic 0, the app starts executing the Smart Monitor mode. This mode uses the CAN handler for continuously receiving the CAN messages. The desired messages are then encoded using the *Package Handler class*, in accordance with the standard data package. Finally, the data package is sent over the Bluetooth interface using the *UART Handler class*. It continues to transmit the received CAN messages until the mode changes or the ECU gets reset. Since this app only accepts the data enclosed inside a standard data package, therefore it ignores any corrupt form of data.

Code Snippets

Let's now present some of the important methods of the library implemented, responsible for data handling and data streaming at the ECU:

The Wireless ECU keeps sending the initialization package until it receives the acknowledgment package, using the DEVICE_init method implemented in the Package Handler class as:

```
uint8_t DEVICE_init(uint8_t number_of_frames){
  /* Wait for data to be received */
  while ( ! (UCSR1A & (1<<RXC1))){
    PACKAGE_init(number_of_frames);
        _delay_ms(1000);
  }
  /* Get and return received data from buffer */
  currentByte = UDR1;

  if(SOF_ACK == currentByte){
    currentByte = UART1_Receive();
    while(currentByte != EOF_ACK){
          data[index] = currentByte;
          index = index + 1;
          currentByte = UART1_Receive();
    }
    return(data[0]);
  }
  return 0;
}
```

In line

- 3) this while loop continues to send initialization package until the first byte is received
- 10) this if statement checks if the received package is the acknowledgment from the smartphone
- 17) returns the number of frames field to the calling function

The PACKAGE_send method implemented in the Package Handler class, is responsible for encoding the messages and id into a stream, and transmitting this stream via the Bluetooth interface:

```
1  void PACKAGE_send(uint32_t *data, uint16_t *id, uint8_t
       framesize) {
2    for(i = 0; i<framesize; i++, j+=8, y++) {
3      txPackage[x] = SOF;
4      x = x + 1;
5      for(k = 0; k<2; k++) {
6        txPackage[x] =
7          (uint8_t) (*(id + y) >> SHIFTCOUNT[k]);
8        x = x + 1;
9      }
10     for(k = 0; k<4; k++) {
11       txPackage[x] =
12         (uint8_t) (*(data + y) >> SHIFTCOUNT[k]);
13       x = x + 1;
14     }
15     txPackage[x] = EOF;
16     x = x + 1;
17   }
18   //begin writing the stream encoded
19   for(i = 0; i<framesize*8; i++)
20   {
21     UART1_Transmit(txPackage[i]);
22   }
23 }
```

In line

- 2) this for loop is the most outer loop that runs frame wise

- 3) Inserts the SOF byte for the current frame
- 5) this for loop is responsible for encoding the id in the stream
- 7) extracts the id byte-by-byte
- 10) this for loop is responsible for encoding the data in the stream
- 12) extracts the data byte-by-byte
- 15) inserts the EOF for the current frame
- 19) this for loop is responsible for writing the encoded stream over the Bluetooth interface
- 21) uses the transmit method from the UART Handler class

The PACKAGE_send method internally uses the UART Handler's UART_transmit method for writing the stream over the Bluetooth interface as:

```
1  void UART1_Transmit (uint8_t data)
2
3  {
4
5    /* Wait for empty transmit buffer */
6    while ( ! ( UCSR1A & (1<<UDRE1)));
7
8    /* Put data into buffer, sends the data */
9    UDR1 = data;
10
11 }
```

In line

- 6) wait until the transmit buffer is empty
- 9) send the data byte over the Bluetooth interface

This method, as implemented in the UART Handler class, sends only one byte from the Bluetooth interface. It is used in a loop by the PACKAGE_SEND method for streaming out data according to the standard data package.

A CD/DVD is attached at the end of the report for complete set of code.

5.4.4 Results

As discussed in Chapter 3, a standard data package is tested with at most 16 frames/messages, the Smart Monitor's app has also been tested with this limit. The Wireless ECU receives 16 CAN messages from the CAN bus, and sends it to the smartphone. These messages as received by the smartphone are displayed on the Smart Monitor as shown in Figure 5.13:

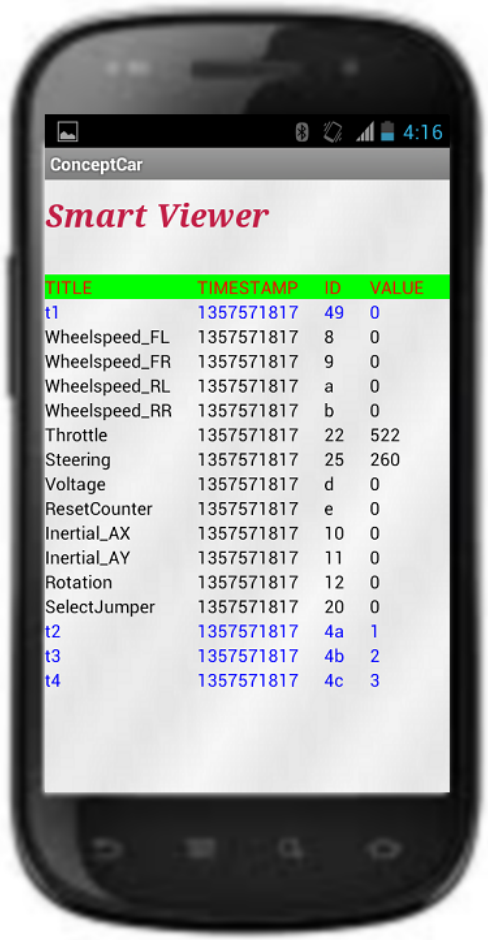


Figure 5.13: The Smart Monitor's Received Messages

As discussed, using the Filter option from the menu bar, the user can select 13 default messages that can be received from the CAN bus from different ECUs. Additionally, the user can add messages by typing in the title and id in the text boxes. These messages are highlighted with a blue color as shown in the figure.

Invoking the LogData option from the menu bar, creates a Log.file on the SD card or internal memory of the smartphone, and begins writing the added messages. This feature allows the user to record the parameters of the Concept Car, that can be used later for monitoring and analyzing purposes. A Log.file has been stored while using the Smart Monitor's app, is shown in Figure 5.14:

```
1  9-Mittwoch-Januar
2
3  TIMESTAMP:ID:DATA
4  1357759264:25:245
5  1357759264:22:0
6  1357759264:a:0
7  1357759264:8:0
8  1357759264:9:0
9  1357759264:b:0
10 1357759264:d:0
11 1357759264:e:0
12 1357759264:10:0
13 1357759264:11:0
14 1357759264:12:0
15 1357759264:20:0
16 1357759264:49:0
17 1357759264:4a:1
18 1357759264:4b:2
19 1357759264:4c:3
20 1357759265:25:245
21 1357759265:22:0
22 1357759265:a:0
23 1357759265:8:0
24 1357759265:9:0
25 1357759265:b:0
26 1357759265:d:0
27 1357759265:e:0
28 1357759265:10:0
29 1357759265:11:0
```

Figure 5.14: The Smart Monitor's Logged File

The LogData feature exactly follows the same format as that of the offline mode. It is important to note that unlike the offline mode, LogData always overwrites the file created previously. Therefore it is recommended to rename the logged file once the logging is done.

5.5 The Concept Car's Smart Controller

As the name suggests, this app specifically targets the second problem i.e. controlling the Concept Car with an alternative to already existing radio controlled transmitter system. Recall from Chapter 1, this radio transmitter generates the PWM signals for steering and driving the car. Now the idea is to replace and map these signals using the Smart Controller app. Apart from steering and driving the car, this controller also brings in the emergency mode, which is previously implemented by using a separate transmitter system, thereby combining features of both the transmission systems. The idea of implementing the Smart Controller is to introduce a user controllable interface for driving the Concept Car over shorter range of distances, for testing purposes. Let's proceed now with discussing the design of the Smart Controller:

5.5.1 The Smart Controller's Design

Once the smartphone is connected to the Concept Car (either manually or using Smart Key), invoking the Smart Controller's control on the startup screen brings in three different modes; Sensor Mode, Manual Mode, and Emergency Mode, as shown in Figure 5.15:

Figure 5.15: The Smart Controller's Modes

Sensor Mode

As the name suggests, this mode brings in another hardware feature of the smartphone namely accelerometer. This mode is implemented with the aim of steering the car by generating the linear movements. For this purpose, acceleration values in the range -8m/s^2 to 8m/s^2 along the y-axis are mapped to the steering signals from 1ms to 1.8ms duty cycles respectively. A lookup table has been created so that each value of acceleration corresponds to a specific steering signal. An acceleration of -8m/s^2 corresponds to 1ms duty cycle (extreme left position), whereas an acceleration of 0m/s^2 corresponds to approximately 1.5ms (idle position), and finally an acceleration of 8m/s^2 corresponds to approximately 1.8 ms duty cycle (extreme right position). The formula used for creating this lookup table is given as:

$$LookUpTable[ScaledAcc] = STEERING_{min} + CONST.ScaledAcc$$

where,

$STEERING_{min}$ corresponds to 1ms duty cycle

$CONST$ corresponds to increment factor, which is equal to 5

$ScaledAcc$ corresponds to mapping the acceleration of (-8m/s^2 to 8m/s^2) to the range from 0 to 160. It is scaled as:

$$ScaledAcc = Acceleration_{y-axis}.10 + 80$$

As the acceleration along y-axis changes by a factor of 0.1m/s^2, duty cycle changes by a factor of 0.005ms, therefore, this method generates the steering signals linearly from 1ms to 1.8ms, as the acceleration changes from -8m/s^2 to 8m/s^2. For throttle, firstly, we have to drive the car for shorter range of distances, inside a lab or a room. Secondly, the car runs quite faster even at the lowest possible driving signals (a PWM signal with 1.6ms duty cycle). Therefore for every mode the throttle signal is generated with a fixed lowest driving value. Using Sensor Mode, a fixed value corresponding to a duty cycle 1.6ms is sent each time the acceleration along x-axis falls below 0m/s^2. The Sensor Mode interface is shown in Figure 5.16:

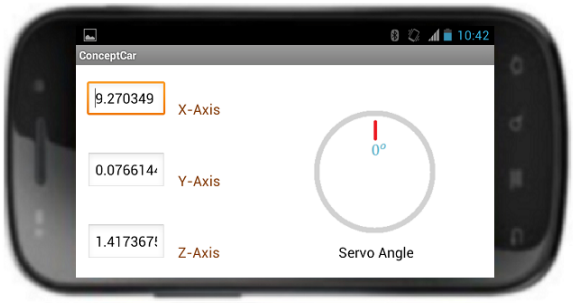


Figure 5.16: The Smart Controller's Sensor Mode

This interface graphically indicates the servo angle, as depicted by the acceleration signals. At 1ms duty cycle, the servo angle reaches approximately to -90 degrees. Similarly, at 1.5ms it reaches 0 degrees and around 2ms it reaches 90 degrees. This interface indicates only these three positions, when the acceleration reaches the corresponding values. As already discussed, each package sent from the smartphone includes three messages (steering, throttle and emergency). The user can invoke the emergency stop from any of the three modes. Pressing the menu button brings in the feature of forcing the emergency stop.

Manual Mode

This mode generates the steering signals with the right and left arrow button, and the throttle signal with the up and down arrow buttons, as shown in Figure 5.17. Unlike the Sensor Mode, it does not produce the linear movements for steering, instead it generates the extreme position signals. Pressing the left arrow button steers the car to the possible extreme left position by generating a value that corresponds to a PWM signal with 1ms duty cycle, and pressing the right arrow steers the car to the possible extreme right position by generating a value that corresponds to a PWM signal with 1.8ms duty cycle. As already discussed in the Sensor Mode, for throttle, a fixed lowest possible driving signal with a duty cycle of 1.6ms is used. Pressing the up button generates this fixed throttle signal.

Likewise the Sensor Mode, invoking the ForceEmergencyStop option in the menu bar forces the emergency stop.

Figure 5.17: The Smart Controller's Manual Mode

Emergency Mode

Although each mode incorporates the feature of enforcing the emergency stop, the Smart Controller also presents it as a separate mode. This mode along with the emergency status fills the package with the ideal values of steering and throttle. As shown in Figure 5.18, touching the green button on the interface sends the emergency signal to the Wireless ECU, that forwards it to the ActorBoard.

Figure 5.18: The Smart Controller's Emergency Mode

5.5.2 Analyzing the Implementation at the Smartphone Side

This section elaborates the implementation of the Smart Controller by presenting the flow diagram and the code examples of the app, at the smartphone side. Let's start with discussing the flow diagram of the app, as shown in Figure 5.19:

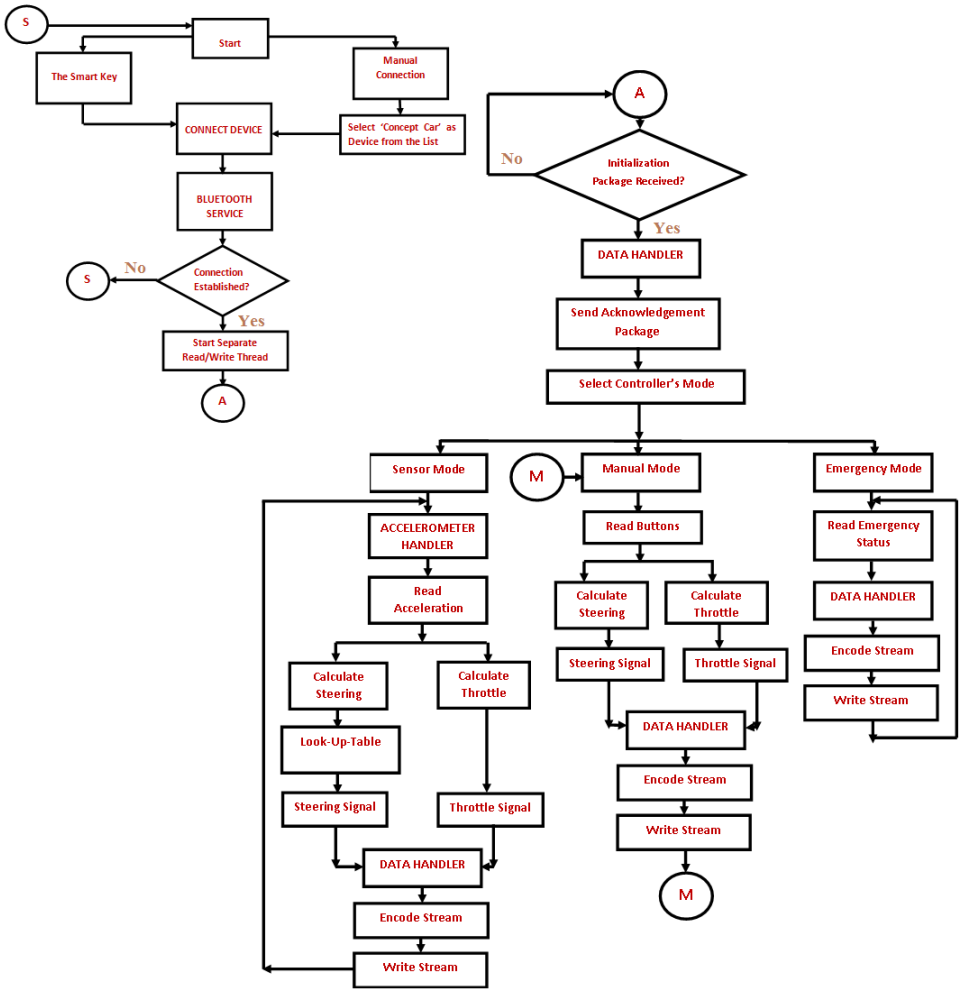


Figure 5.19: The Smart Controller's Flow Diagram at the Smartphone Side

Likewise the Smart Monitor, it also initiates with the same connection mechanism. Upon success it creates a new read/write thread using the *Bluetooth Service* class. After a successful start of the new thread, an initialization package is awaited. As soon as the package is received, the *Data Handler*

class extracts the number of frames field (number of messages enclosed inside each package), and sends back an acknowledgment package. The initialization/acknowledgment packages are sent as discussed in Section 5.3. With this the user is now allowed to choose any of the three available modes, as discussed (see Figure 5.15). Each mode is expected to generate three messages namely steering, throttle and emergency, with different source of controls. Starting with the Sensor Mode, it uses the accelerometer to generate the corresponding steering and throttle messages. Similarly, using Manual Mode, these signals are generated using the arrow buttons. Both of these modes include the emergency status from the menu button. Finally, the Emergency mode generates the emergency signal from the button, along with the idle values of the steering and throttle. Once these signals are generated independent of the mode, the *Data Handler* class encodes the stream, and sends it using the Bluetooth interface. Since this app only accepts the data enclosed inside a standard data package, therefore it ignores any corrupt form of data.

Code Snippets

Let's now present some of the important methods of the library implemented in different classes, responsible for data handling and data streaming for the Smart Controller:

The encode method that receives the tuple (id, data), encodes it into a stream in accordance with the standard data package, is implemented in the Data Handler class.

```
byte[] encodePackage(int[] dat, short[] id, int frsz) {
  byte[] txPackage = new byte[24];
  byte i,k;
  byte j = 0;
  int x = 0;
  byte y= 0;
  for(i = 0; i<framesize; i++, j+=8, y++) {
    txPackage[x] = SOF;
    x = x + 1;
    for(k = 0; k<2; k++) {
      txPackage[x] =
       (byte) (id[y] >> SHIFTCOUNT[k]);
      x = x + 1;
        }
    for(k = 0; k<4; k++) {
      txPackage[x] =
```

```
17          (byte) (data[y] >> SHIFTCOUNT[k]);
18        x = x + 1;
19          }
20          txPackage[x] = EOF;
21          x = x + 1;
22    }
23    return txPackage;
24 }
```

In line

- 7) this for loop is the most outer loop that runs frame wise
- 10) this for loop is responsible for encoding the id in the stream
- 15) this for loop is responsible for encoding the data in the stream

The method responsible for writing the stream once encoded using encodePackage method, is write method, implemented in the Bluetooth Service class as:

```
1  /**
2 * Write to the connected OutStream.
3 * @param buffer  The bytes to write
4 */
5 public void write(byte[] buffer) {
6    try {
7      mmOutStream.write(buffer);
8      // Share the sent message back to the UI Activity
9      mHandler.obtainMessage(ConceptCar.MESSAGE_WRITE,
10                    -1, -1, buffer).sendToTarget();
11   } catch (IOException e) {
12   Log.e(TAG, "Exception during write", e);
13   }
14 }
```

In line

- 7) writes the package over the Bluetooth interface
- 9) informs the handler in the main activity after the stream is successfully written

5.5.3 Analyzing the Implementation at the Wireless ECU Side

This section elaborates the implementation of the Smart Controller by presenting the flow diagram and the code examples of the app, at the ECU side. Let's start with discussing the flow diagram of the app, as shown in Figure 5.20:

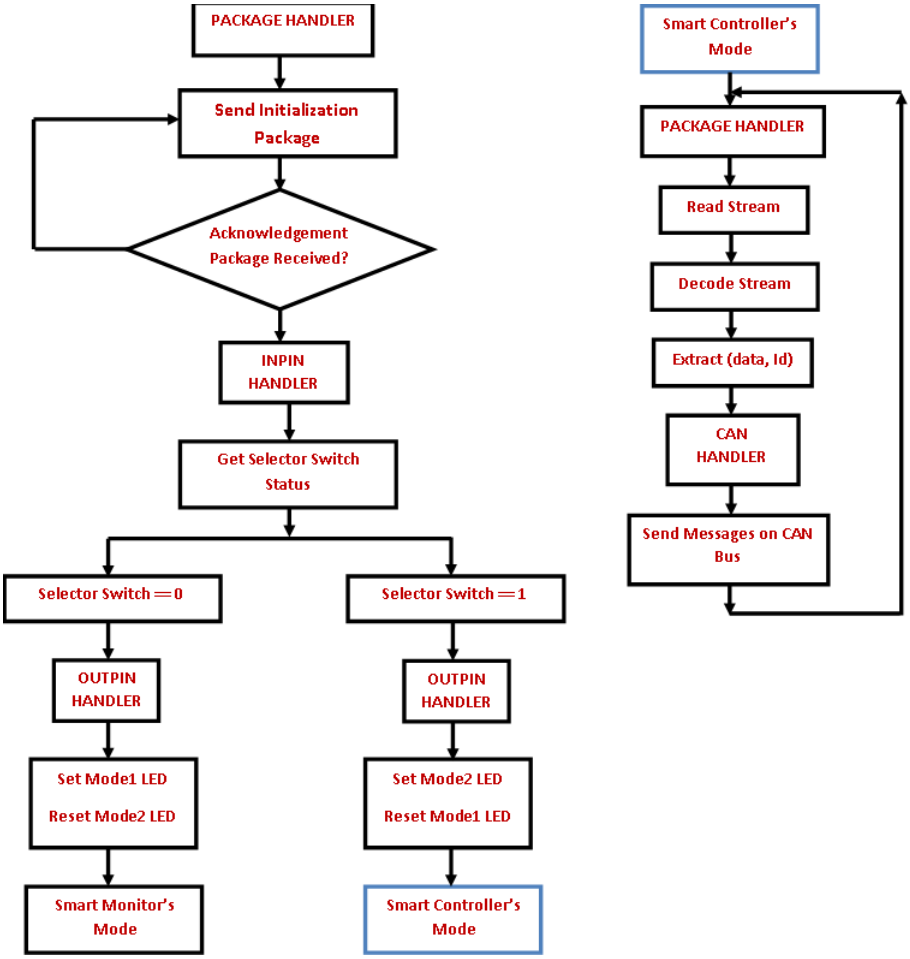


Figure 5.20: The Smart Controller's Flow Diagram at the ECU Side

The app starts with the *Package Handler class* sending the initialization package, until the acknowledgment package is received. As soon as it receives the acknowledgment package, extracts the *number of frames* field, and the *Inpin Handler class* gets the status of the selector button. This status is also sent to the CAN bus using the *CAN Handler class*. The initializa-

tion/acknowledgment packages are sent as discussed in Section 5.3. If the selector switch provides a logic 1, the app starts executing the Smart Controller's mode. Using the *Package Handler* class, it continuously reads the stream, decodes it with respect to the number of frames, and extracts the tuple (data, id). Finally, the *CAN Handler* class places the data with its unique id on the CAN bus. It continues this procedure until the mode changes or the ECU gets reset. Since this app only accepts the data enclosed inside a standard data package, therefore it ignores any corrupt form of data.

Code Snippets

Let's now present some of the important methods of the library implemented in different classes, responsible for data handling and data streaming for the Smart Controller:

The PACKAGE_receive method implemented in the Package Handler class, is responsible for receiving the stream over the Bluetooth interface, and extracting the tuple (data, id) from it as:

```
uint8_t PACKAGE_receive(uint16_t *id, uint32_t *msg,
    uint8_t msgCount){
  currentByte = CC_UART1_Receive();
  for(i = 0; i<msgCount; i++) {
    if(SOF == currentByte)
    {
      currentByte = UART1_Receive();
      while(currentByte != EOF){
        data[index] = currentByte;
        index = index + 1;
        currentByte = UART1_Receive();
      }
      *(id++) = (uint16_t) ((uint16_t) data[ID_BYTE0] |
                (uint16_t) data[ID_BYTE1] << 8);
      *(msg++) =
       (uint32_t) ((uint32_t) (data[DATA_BYTE0])|
       (uint32_t) (data[DATA_BYTE1] << 8)|
       (uint32_t) (data[DATA_BYTE2] << 16)|
       (uint32_t) (data[DATA_BYTE3] << 24));
      index = 0;
      if(i+1 != msgCount)
        currentByte = UART1_Receive();
    }
```

```
23      else return 0;
24    }
25    return i;
26  }
```

In line

- 3) this for loop keeps track of the total number of frames
- 4) this if statement tests for the start of the package
- 7) this while loop deals with the current frame under reception
- 12) decodes the id from the currently received frame
- 14) decodes the data from the currently received frame
- 20) this if statements checks if the all the desired frames are received

The PACKAGE_receive method internally uses the UART Handler's UART_receive method for reading the stream over the Bluetooth interface as:

```
1  uint8_t UART1_Receive (void)
2  {
3
4    /* Wait for data to be received */
5    while ( ! (UCSR1A & (1<<RXC1)));
6
7    /* Get and return received data from buffer */
8    return UDR1;
9
10 }
```

In line

- 5) wait until the first byte is received
- 8) read the byte before the next byte is received

This method, as implemented in the UART Handler class, receives only one byte from the Bluetooth interface. It is used in a loop by the PACKAGE_receive method for streaming in data according to the standard data package.

A CD/DVD is attached at the end of the report for complete set of code.

5.5.4 Results

As discussed, the Sensor Mode of the Smart Controller generates linear steering movement as a function of the acceleration along y-axis. The Sensor Mode interface indicates three important servo positions (extreme left, idle and extreme right), as the acceleration reaches the corresponding values (as shown in Figure 5.21). At an acceleration of -8m/s^2, the servo angle reaches the extreme left position (-90 degrees). Similarly, at 0m/s^2, the servo angle reaches the idle position (0 degrees). Finally at 8m/s^2, the servo angle reaches the extreme right position (90 degrees).

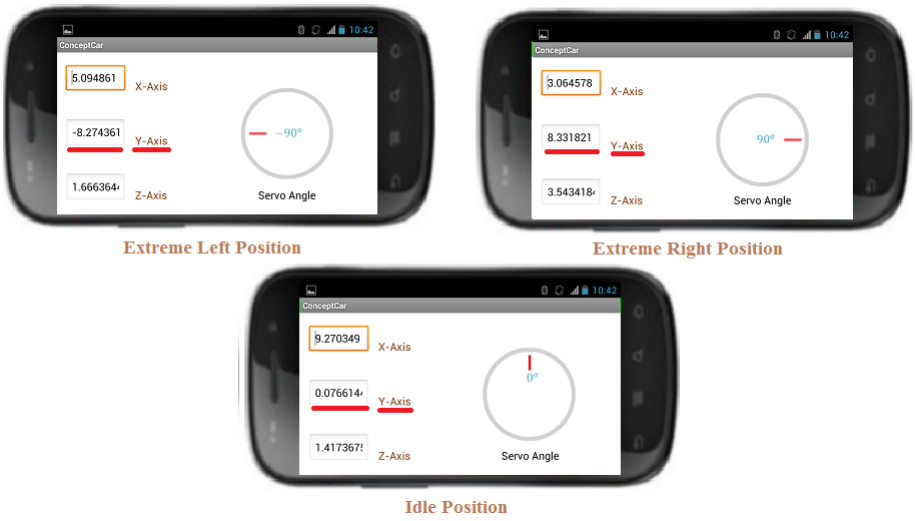


Figure 5.21: The Sensor Mode Results

The Emergency Mode is subjected to present the same behavior as provided by the separate SVS radio transmission system. Unlike this radio transmission system, the Emergency Mode, by default starts up with the normal mode. This is indicated by a green emergency button on the interface, as shown in Figure 5.22. As soon as the user invokes this button, it forces the emergency stop, and indicates it by highlighting the button with red color.

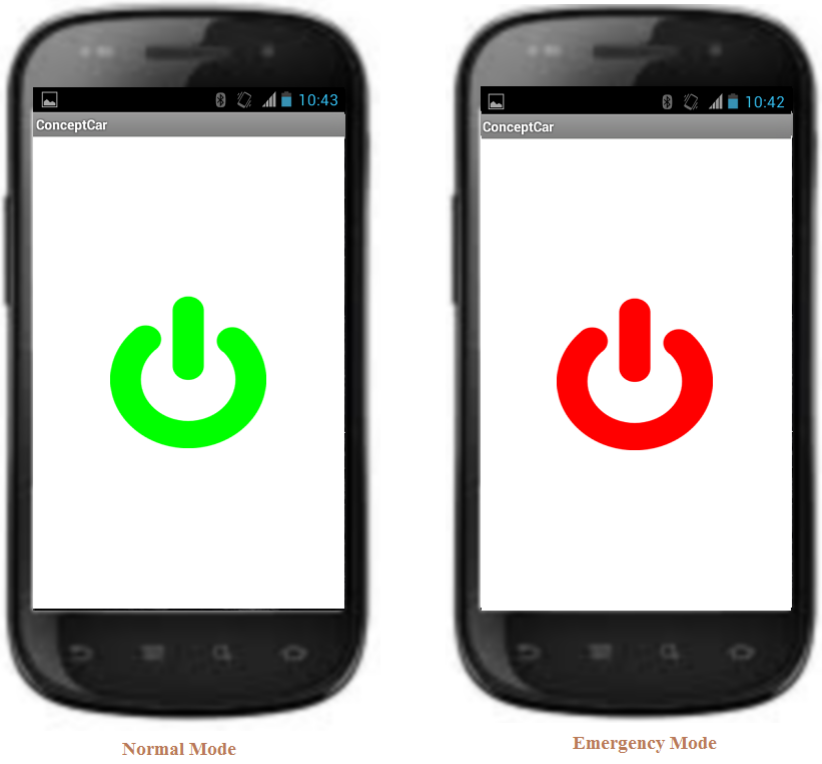


Figure 5.22: The Emergency Mode Results

5.6 Conclusion

This chapter starts off with recalling the problems and the devised solutions. The applications designed to test and verify the devised solution are presented. The first application, the Smart Monitor for monitoring the car internals lively in comparison with the already existing monitoring modes is presented. The improvement achieved in monitoring the car internals is explained. The idea of how these features are developed is elaborated by presenting the design, flow diagram and code examples of the Smart Monitor. Finally, the results obtained are presented. Similarly, the second application, the Smart Controller as an alternative to radio transmitter system has been elaborated.

Chapter 6

Summary and Future Recommendations

6.1 Summary

The Concept Car is designed and developed with the idea of a modern car architecture, so as to; study, design, implement, test and deploy modern future car features. Although the Concept Car does not incorporate as many ECUs a modern car carries (up to hundred), it still shares many attributes of a modern car, such as a centralized CAN bus architecture. Since it is a research platform with an objective of discovering future car features, testing applications before deployment is a very important aspect. For testing purposes, firstly, a mean of lively monitoring the car internals while the car is driven was desired. Secondly, a driving tool was desired so as to move the car inside a lab or room, for few meters. The idea of lively monitoring the car parameters and driving the car for shorter range of distances has lead to the concept of *Designing, Developing and Integrating a Wireless Communication Unit in the Concept Car*. With the natural architecture of the Concept Car, consisting of separate ECUs responsible for their distinctive tasks, and the CAN bus as the centralized interaction medium for the ECUs, the solution became quite obvious as to add a new ECU capable of wireless communication. With the selection of a user end device for monitoring and controlling, there were many choices such as PDAs, Laptops, PCs etc. An Android based smartphone, being used very commonly for development purposes, was an easier selection amongst competitors.

6.2 Future Recommendations

The Concept Car offering a wide variety of applications to be deployed, meanwhile also provides a space for bringing up improvements within the current existing platform. The current work while targeting the problems under consideration, and implementing the devised solutions, ends up with some noticeable recommendations as:

- The Wireless ECU as presented in this work, incorporates a reserved socket for introducing another wireless technology within this ECU. Likewise the Bluetooth module used in this work, the WiFi devices are also commonly available with the serial interface, and are widely used for embedded development. Additionally, WiFi technology is one of the most common features available almost with all the smartphones. For these reasons, a WiFi technology can be easily adapted to the current version of the Wireless ECU, and is therefore strongly recommended.

- The EmergencyBoard responsible for introducing the galvanic isolation within the system, is one of the most important ECUs of the lot. The current version of this ECU does not incorporate the CAN feature. This is the only ECU not capable of interacting with the other ECUs via CAN bus. It only accepts the input from the radio receiver and the ActorBoard, and bypasses it to the SensorBoards and the actuators respectively. With the advent of the Wireless ECU, after this work, the need of making the EmergencyBoard blessed with the CAN interface, becomes immensely important. Therefore, revising the current version of this ECU is strongly recommended.

- The Wireless ECU and the smartphone software tool are designed and developed with the generic implementation of the software libraries. This allows the user to design and develop any application using the same hierarchy of class libraries. The Smart Monitor and the Smart Controller are implemented for targeting the specific tasks. For instance, the Smart Monitor only comes into play when the car is driven from the radio transmission system. Similarly, the Smart Controller is aimed to drive the car independent of the Smart Monitor. For future work, it is recommended to use the same class of libraries for featuring an application capable of performing both tasks simultaneously, on the same user interface.

Bibliography

[1] Atmel Corporation Devices. http://www.atmel.com

[2] AT91SAM7A2 Board. http://www.keil.com

[3] SVS Website. http://www.svs-funk.com

[4] Atmel Corporation Devices. http://www.alldatasheet.com

[5] PWM Introduction. http://www.acroname.com

[6] LEGO MINDSTORMS. http://mindstorms.lego.com

[7] Chih-Yang Chen, Tzuu-Hseng S. Li, Kai-Chuin Lim. Design and Implementation of Intelligent Driving Controller for Car-Like Mobile Robot. ICSSE 2010, Taipei, Taiwan.

[8] Heikki Rissanen, J.Mahonen, Keijo Haataja, Markus Johansson, Pekka Toivanen. Designing and Implementing an Intelligent Bluetooth-Enabled Robot Car. University of Kuopio, Finland.

[9] Yeong Che Fai, Shamsudin H.M. Amin, Norsheila nt Fisal, J. Abu Bakar. Bluetooth Enabled Mobile Robot. IEEE ICIT'02, Bangkok, THAILAND.

[10] Smartphone enabled Legos. http://www.kleekbots.com

[11] GPRS Introduction. http://www.etsi.org

[12] Xue Huixia, Gao Lin, Wang Lu, Li Wenbin, Yang Kai. Monorail Car's Wireless Control System based on Smartphone Platform. 2011 Third International Conference on Measuring Technology and Mechatronics Automation

[13] Wang Shaokun, Xiao Xiao, Zhao Hongwei. The Wireless Remote Control Car System Based On ARM9. 2011 Third International Conference on Measuring Technology and Mechatronics Automation.

[14] Autonomous Labs. http://www.autonomos.inf.fu-berlin.de

[15] NFC Standard. http://www.nearfieldcommunication.org

[16] WiFi definition. http://www.webopedia.com

[17] Bluetooth Standard. http://www.bluetooth.com

[18] NFC Technical Specifications. http://www.nfc-forum.org

[19] IEEE 802.11. http://standards.ieee.org

[20] The RS232 Standard. http://www.camiresearch.com

[21] Bluetooth Specs. http://www.bluetooth.com

[22] Bluetooth Specs. http://www.bluetooth.org

[23] Smartcard Basics. http://www.smartcardbasics.com

[24] C.Y. Leong, K. C. Ong, K. K. Tan*, O.P. GAN. Near Field Communication and Bluetooth Bridge System for Mobile Commerce. 2006 IEEE International Conference on Industrial Informatics.

[25] Roving Networks Webpage. http://www.rovingnetworks.com

[26] UART definition. http://www.pcmag.com

[27] USB Homepage. http://www.usb.org

[28] PCA82C250 DataSheet. http://www.nxp.com

[29] MAX1837 DataSheet. http://www.maximintegrated.com

[30] GNU Compiler Collection. http://gcc.gnu.org/

[31] AVRISP Programmer. http://www.atmel.com/tools

[32] AVRDUDE Homepage. http://www.nongnu.org

[33] Android Introduction. http://developer.android.com

[34] Eclipse Homepage. http://www.eclipse.org

[35] Android Development. http://developer.android.com

[36] Cadsoft's Website. http://www.cadsoft.de

[37] Maxim IC's. http://www.datasheets.maximintegrated.com

[38] RS232 to USB Adapter. http://www.pollin.de

[39] TTL Level. Balch, Mark (2003). Complete Digital Design: A Comprehensive Guide To Digital Electronics And Computer System Architecture

[40] Serial Port Complete: Programming and Circuits for Rs-232 and Rs-485 Links and Networks By Jan Axelson, Lakeview Research

[41] HyperTerminal Website. http://www.hilgraeve.com

[42] PuTTY Website. http://www.putty.org

[43] TeraTerm Website. http://www.ayera.com

[44] Configuring Serial Terminal Emulation Programs. http://www.actel.com

[45] SPP profile. http://www.palowireless.com

[46] Federal Standard 1037C. http://www.its.bldrdoc.gov

[47] BlueTerm Android App. https://play.google.com

[48] CAN Adapter. http://www.mhs-elektronik.2de