Hochschule **Rosenheim**
University of Applied Sciences

Future Electric Vehicle on LEGO

Master Thesis presented to the faculty of
Electrical Engineering and Information Technology

Hochschule Rosenheim
University of Applied Sciences, Germany.

In partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING
IN ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Author:	Karansingh Savant
Registration Number:	748612
First Supervisor:	Prof. Dr.-Ing. Herbert Thurner
Second supervisor:	Prof. Dr.-Ing. Reiner Schell
Company supervisor:	Dr. Kai Huang(TUM Informatik)

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, April, 2013 **Karan Savant**

Acknowledgement

I would like to express my gratitude to my supervisor Dr. Kai Huang for his guidance, support, encouragement, useful comments and providing me the opportunity to work in this interesting field of research. It is been a great experience working under him in at Technical University Munich. During this thesis work I have gained a lot of knowledge which helped me to grow technically as well as professionally.

I would like to thank Prof.Dr.-Ing. Herbert Thurner for his guidance, kindness and timely support during my thesis work. His quality as a supervisor, useful feedback during my work makes him an ideal person. I would also like to thank Prof. Dr.-Ing. Reiner Schell for his kind support.

Many thanks to my colleagues, administration and IT team at TUM for their support. Last but not least, I would like to thank all my friends especially Sandeep Guidivada for their help, encouragement and support for making my life colorful & to achieve my goals. Above all I would like to express my hearty gratitude to all my family members without whom it would have not been possible.

Thank you all!

Abstract

Control tasks are a part of many mechatronic or embedded systems that can be critical, therefore their performance requires evaluation. The tasks have become increasingly complex and computationally intensive reaching the limits of the available hardware. This raises the need for performance evaluation and the design of a platform to test different possible hardware architectures.

In my thesis, I use a LEGO car speed controller for the design and implementation of an evaluation platform, since it is a classical problem that is excellent for testing and demonstrating different control strategies. The goal of this control task is to control the speed of the Lego car in both directions using PWM waveforms, enable smartphone based control. While traditional approaches implement the control strategies on standard desktop PCs, our approach uses an FPGA. The interest in FPGA-based control is increasing due to the low-cost of FPGAs and their rapid prototyping capabilities. A system on a chip" is designed in the FPGA as the control unit of the inverted pendulum.

First, an NIOS 2 soft-core processor was implemented in FPGA logic. A WIFI interface to the FPGA is enabled using the RN 134 Wi-Fi module so that the car can be controlled via Wi-Fi by a smartphone application. An android application was implemented to control the car using inbuilt sensors. The first strategy I was to control the car without feedback. In the second strategy, Using interrupts, the Back EMF is measured which gives feedback about the car speed used as control parameter.

The experimental results show that FPGAs can be used to implement control tasks and evaluate the performance of different strategies. FPGA based architectures are a promising solution for control problems with intensive computations and problems with a large number of interrupts occurring.

Table of Contents

ACKNOWLEDGEMENT **3**

ABSTRACT **4**

LIST OF FIGURES **7**

LIST OF TABLES **8**

1. INTRODUCTION **9**

1.1 PROBLEM STATEMENT 11
1.2 OUTLINE OF THE THESIS 12

2. BACKGROUND AND RELATED WORK **13**

2.1 RELATED WORK 14
2.2 FPGAS AS PROTOTYPING PLATFORM 14
2.3 CONTROL THEORY 17
2.3.1 PID Controller *18*
2.3.2 PV Controller *18*
2.4 LEGO MINDSTROM NXT KIT 18
2.5 ROVING NETWORKS RN134 WI-FI MODULE 19
2.6 MOTOR BACK EMF(ELECTROMOTIVE FORCE) MEASUREMENT 21

3. CONCEPT WORK: AUTOSAR ON FPGA **25**

3.1 AUTOMOTIVE OPEN SYSTEM ARCHITECTURE (AUTOSAR) 26
3.2 ISO26262 (SAFETY STANDARD) 27
3.3 AUTOSAR ON FPGA 28
3.3.1 System Architecture *29*
3.3.2 ECU design on FPGA-based static hardware *29*

4. FPGA BASED IMPLEMENTATION **35**

4.1 FPGA PLATFORM USED 36
4.2 HARDWARE ARCHITECTURE 37
4.2.1 System Design *37*
4.2.2 Level Shifting circuitry *44*
4.2.3 Driver Circuit-L298 ..... *45*
4.2.4 System Wiring and Setup *45*
4.2 CONTROL SOFTWARE 48
4.2.1 Software Architecture *48*
4.2.2 Implementation of the control software *48*

5. SMARTPHONE BASED CONTROL **56**

5.1 ANDROID 57
5.1.1 Introduction *57*
5.1.2 Sensors in Android *58*
5.1.3 Sensor Framework Android *59*

5.1.4 Position Sensor Android[18] *69*
5.3 WIFI IN ANDROID 73
5.4 INTERFACE WITH THE WI-FI MODULE 74
5.5 ECAR REMOTE APPLICATION 75
6. CONCLUSION AND FUTURE SCOPE 80
BIBLIOGRAPHY 84

List of Figures

Figure 2.1: FPGA design flow [3] 16
Figure 2.2: A feedback loop to control a dynamic system 17
Figure 2.3: Lego Car 19
Figure 2.4: RN 131 19
Figure 2.5: Block Diagram RN 131 20
Figure 2.6: Back EMF(Electromotive force) Measurement[2] 22
Figure 3.1: The AUTOSAR layer-based model, from MCU to application layer [15] 26
Figure 3.2: Porting of the AUTOSAR ECU architecture to an FPGA platform[17] 30
Figure 3.3: Block diagram of an automotive ECU deployed in an FPGA[17] 31
Figure 3.4: Safety Path 33
Figure 4.1: Components of the Sensor less System 37
Figure 4.2: The components in the SOPC Builder 38
Figure 4.3: Connecting the components in the SOPC Builder 39
Figure 4.4: System Design for controlling the ecar 42
Figure 4.5: Level Shifting Circuit 44
Figure 4.6: Motor Driver Circuit 45
Figure 4.7: A block diagram of the final setup 46
Figure 4.8: Flow chart for the main loop 49
Figure 4.9: Timing requirements for the ADC 51
Figure 4.10: PID Controller 53
Figure 4.11: PID Response Graph 54
Figure 5.1: The current Android 57
Figure 5.2: Android Coordinate System 70
Figure 5.3: Interface 74
Figure 5.4: Flowchart for Startup and Exit: 75
Figure 5.5: Flowchart for Managing Remote Functions: 76
Figure 5.6: Application Screen Shots: 78
Figure 6.1: New Configuration, Driving Modes 82
Figure 6.2: Multiprocessor Architecture 83

List of Tables

TABLE 4.1: FPGA PIN NAMES.....47

TABLE 5.1: SENSOR TYPES SUPPORTED BY THE ANDROID PLATFORM. 59

TABLE 5.2: SENSOR AVAILABILITY BY PLATFORM. 62

Chapter

1. Introduction

In modern buildings the light goes on the moment you enter the hallway. As soon as no motion can be detected, the system assumes that the hallway is empty and turns off the light, thus saving a substantial amount of energy. This is one example of many control systems which are nowadays a part of our daily life and are responsible for regulating a lot of things. Another good example is a thermal system. The only thing we need to do, is define the desired temperature in the system and it then automatically decides whether heating or cooling is necessary without interference from us. We daily make use of other control systems in our activities as well.

Driving a car for an example: Control software regulates the motor, cruise control maintains the speed of the car at a chosen speed, the frequency of the windscreen wiper automatically adapts to the density of the rain drops and when the car ride ends, the car is parked automatically. Although different in function, the control systems have the same building blocks: Each system contains a set of sensors that are periodically evaluated by a control unit, which controls the actuators in the system. The sensors measure the state of the system and give feedback to the control unit. Based upon the feedback, the control unit calculates the reaction of the system and drives the actuators correspondingly. Control tasks are usually integrated in mechatronic or embedded systems, which often involve hardware/software interactions and can be safety critical. They are becoming increasingly complex and require sophisticated tools to develop them. While model-based development has been very common for developing application functionality, tools that take into account the hardware as well are gaining interest.

Embedded control systems are found in a wide range of applications such as consumer electronics, medical equipment, robotics, automotive products, and industrial processes. Embedded control systems typically use microprocessors, microcontrollers or Digital Signal Processors (DSPs) for their implementation. For such systems, control algorithms are implemented as software programs that execute on a fixed architecture hardware processor. The processor itself is connected to various peripherals such as memories, Analog to Digital converters, and other I/O devices.

Alternatively, FPGAs are increasingly becoming popular as implementation platforms on which the control algorithms can be implemented by programming reconfigurable hardware logic resources of the device. FPGAs have characteristics that make them suitable for realizing hardware implementations of algorithms and systems. They offer excellent features such as computational parallelism, reconfigurable customization, and rapid-prototyping. Recently, there has been a growing interest in developing FPGA-based control systems. In this application, the high switching frequency (ranging from 20 to 40 kHz) required fast computation of the control law, effectively ruling out software-based implementations on microprocessors or DSPs. In addition to control algorithms, FPGAs can also be used to implement various other components of the control system. For example, Zhao, Kim, Larson, & Voyles (2005) used a FPGA to implement a control system-on-a-chip for a small scale robot. While microcontrollers and DSPs have had the advantage of lower device cost over FPGAs, the gap in cost is narrowing with advancing technology, making FPGAs more and more attractive devices. Moreover, FPGA-based implementations may reduce overall cost of a system since multiple components of the design can be implemented as a system-on-a programmable-chip. In some cases, FPGA-based implementations may give higher levels of performance for a design as compared to other implementations. For such cases, FPGAs may be the only choice for implementation because

microcontrollers and DSPs would not meet the performance requirements, and Application Specific Integrated Circuits (ASICs) may have too high development costs.

1.1 Problem statement

The goal of this thesis is to design and implement an evaluation platform using an FPGA for a controlling a LEGO car using an Android application. The main difference of this approach compared to existing research in this field, is the use of an FPGA for the implementation. Most research projects use a standard desktop PC to control the systems. Control tasks typically contain sensors and actuators - as mentioned above - that need to be interfaced with the platform before the evaluation can be performed. Since an FPGA is used as the platform, interfacing requires the development of circuits that connect the FPGA with sensors and actuators.

The FPGA with the interface circuits form the foundation for the evaluation platform. A processor and memory can then be implemented. The control tasks are then implemented in software which is the reason why the design of software architecture is necessary and included in this work. For this firstly, an NIOS 2 soft-core processor was implemented in FPGA logic. A WIFI interface to the FPGA is enabled using the RN 134 Wi-Fi module so that the car can be controlled via Wi-Fi by a smartphone application. An android application was implemented to control the car using inbuilt sensors. The first strategy was to control the car without feedback. In the second strategy, Using interrupts, the Back EMF is measured which gives feedback about the car speed used as control parameter.

The detailed tasks are as follows:

- Interfacing the Lego car with Wi-Fi and controlling it remotely via Smart-Phone/Tab
  - ✓ Create a Single NIOS core with required peripherals like UART, JTAG debug interface, Memory on the Altera Cyclone 4 Development board.
  - ✓ Build the application program using ALTERA NIOS II 11.0 Software Build Tool for Eclipse.
  - ✓ Investigate how the Application Program can comply with the AUTOSAR and ISO23636 (Safety Standard) Standard.
  - ✓ Follow a Model based development method.
  - ✓ Configure the Wi-Fi Module as an Access point Device and interface the FPGA with the Wi-Fi Module.
  - ✓ Control the Vehicle using an open source Android based Wi-Fi application.
- Re-engineer the Lego Technic 9398 into 4-wheel independent steering/driving
  - ✓ Create a Motor Control IP for the Altera NIOS 2.
  - ✓ Investigate the possible control algorithms.
  - ✓ Investigate the implementation based on automotive standards like AUTOSAR.
  - ✓ Implement and test the developed application.
- Develop an Android Application
  - ✓ Implement Control Based on the Android device inbuilt sensors
  - ✓ Use the Orientation Sensor to control the eCar speed.
  - ✓ Communicate with Wi-Fi module.

1.2 Outline of the thesis

The thesis is structured as follows:

CHAPTER 2: BACKGROUND AND RELATED WORK

In chapter 2 a brief description of the related work in that field is described highlighting the difference of the approach used in this thesis. FPGAs will then be introduced as prototyping platforms. The advantages of using FPGAs will be discussed and the design flow illustrated. An introduction to the control theory basics and motor back EMF measurement is given. Finally some details about the Wi-Fi module and the LEGO car used are given.

CHAPTER 3: CONCEPT WORK: AUTOSAR ON FPGA

In this chapter important automotive standards like the AUTOSAR and ISO26262 safety standard are overviewed. The concept work presented in this chapter is regarding implementation of AUTOSAR on FPGA based platforms.

CHAPTER 4: FPGA BASED IMPLEMENTATION

Chapter 4 explains the essential steps to design back EMF based control of DC motor. After developing the level-shifting circuits to interface the FPGA, the hardware system can be designed. The interrupt handling, which is part of the hardware system, plays an important role in the success of the control software. Therefore, different interrupt controllers and interrupt service routines are also included in this chapter. Following the description of the hardware architecture is a description of the software architecture. Finally, the implementation of the controller is described.

CHAPTER 5: SMARTPHONE BASED CONTROL

To design an Android WiFi remote application to control the motion of the lego car using Wifi. The motion of the Lego Car is controlled using the inbuilt orientation sensor of an android device.

CHAPTER 6: CONCLUSION AND FUTURE WORK

Chapter 6 summarizes the work performed and discusses the results. A brief outlook on possible future work is portrayed at the end of the chapter.

Chapter
2. Background and Related Work

2.1 Related Work

Currently great efforts are taken to move from ordinary cars with combustion engine towards fully electric cars. In industry this way leads over cars with hybrid engines. Thereby a lot of the restrictions of combustion engines are inherited and the electric engine is only used to replace the ordinary engine. So this approach cannot benefit from the new features of electric engines. On the other side all the mechanical safety mechanisms are working as before to ensure necessary functionality in situations of system blackouts. [1]

Within the eCar project, an experimental platform is built to demonstrate the new possibilities with respect to the state of the art in computer science. The goal is to develop an innovative architecture for electric cars fully steered by x-by-wire. Further project goals are centralization of control functions to minimize the number of ECUs, an innovative system for redundancy management, and unification of used bus protocols.

2.2 FPGAs as Prototyping Platform

In the past years FPGAs have been increasingly used to implement digital logic. An FPGA is an integrated circuit that can be configured by the user for a specific application. It contains logic blocks that can be programmed and reconfigurable interconnects to wire the blocks together. In recent years the trend has been going a step further. The logic blocks and interconnects are now combined with microprocessors forming a complete "system on a programmable chip". Alternatively, soft processor cores can be implemented within the FPGA logic providing more flexibility.

The market for FPGAs is increasing, thus leading to an increase in the design starts with FPGAs. Historically, FPGAs were less energy efficient and implemented less functionality than their counterparts, the Application Specific Integrated Circuits (ASICs). Rising costs for integrated circuits and a longer development time resulting in a longer time to market have made FPGAs a better alternative to ASICs for developing and evaluation purposes. Another important advantage of FPGAs is the ability to re-program them making them suitable for prototyping.

Commercial prototyping boards are available bundling FPGAs with memory and I/O interfaces. This makes the use of FPGAs even more attractive because everything is available on one chip. Following list summarizes the advantages of FPGAs:

- Rapid Prototyping
- Shorter time-to-market
- Lower costs
- Re-configurable hardware provides flexibility

The problem with FPGAs is that the internal signals are not visible and thus hard to debug. Bugs in the design that are related to internal signals are very hard to find and thus are time consuming. Different FPGA families exist offering the users general architectures as well as application specific solutions. The scope of applications of FPGAs is wide. Application fields include digital signal processing, automotive, medical imaging, computer vision, bioinformatics, computer hardware emulation and prototyping. Other areas are also growing.

Typically hardware description languages (HDLs) or schematic designs are used to define the behavior of the FPGA. Schematic designs visualize the systems but are not suitable for large structures because the effort of drawing every piece is too high. After simulating the design and verifying the functionality, electronic design automation tools can be used to generate technology-mapped net lists. These are then fitted to the actual FPGA using place-and-route processes that are usually performed by the FPGA Company's proprietary software. Results can be validated by analyzing timing and simulating the design. Other verification methods are also possible. Once the design and validation of the system are complete, the generated file can be used to configure the FPGA. A typical design flow is illustrated in figure 2.1

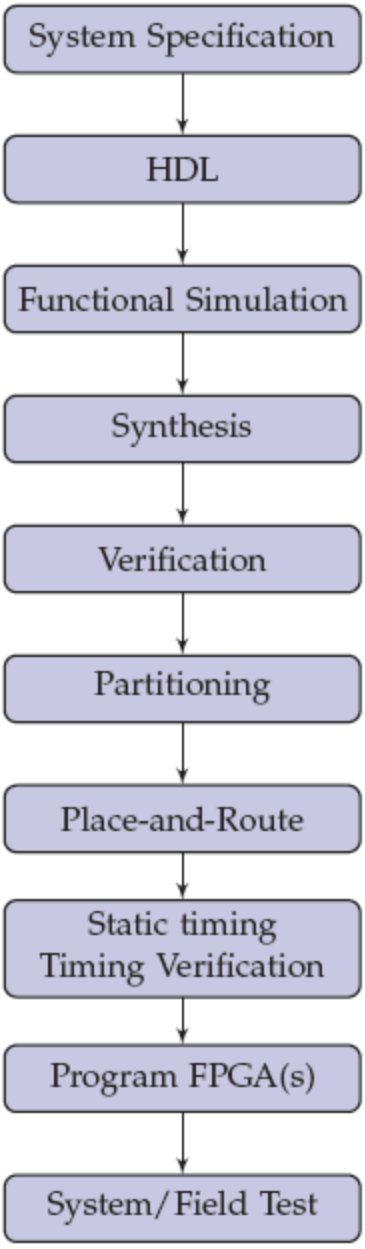


Figure 2.1: FPGA design flow [3]

Nowadays, tools for FPGA configuration that automatically generate HDL files exist. These tools offer libraries and pre-made components that can directly be used to implement systems and different functionalities. An example of such a tool is the SOPC-Builder from Altera. Using these tools makes designs less error-prone because the components are already extensively tested. Another advantage is time. These tools save on design time and testing time and can therefore further reduce time-to-market. Different designs can then be developed using the same hardware, thus reducing costs. Using the same hardware also allows for a better comparison

since different designs use the same resources. Performance differences can then be related to changes in the design excluding possible variations due to hardware changes.

2.3 Control Theory

The benchmarks used in this thesis are all control tasks. To understand how such tasks are fulfilled a brief introduction on control theory is necessary. Control theory deals with the behavior of dynamic systems. It can be divided into three main steps:

- Modeling problems: The goal of this step is to find a correct mathematical model for a real system. The system can be derived from mechanics, electrical engineering, mathematical physics, biology and many more.
- Analysis problems: Here, properties of the system such as controllability, observability and stability among others are analyzed.
- Synthesis problems: In this step the feedback controller which stabilizes and optimizes the performance of the closed-loop system is constructed. Robustness issues are studied in this step as well.

The classical control theory includes feedback in the system. In closed-loop systems feedback is used to control the outputs of the dynamic system. A typical feedback loop is illustrated in figure. Such a loop typically controls one or more outputs of the system. The controller calculates the input of the system from a reference and the state of the system. The difference is the error that is used to calculate the reaction of the system. The input of the system specifies how the system should react to reach a desired state. The current state of the system is determined from sensors placed in the system which form the feedback and close the loop.

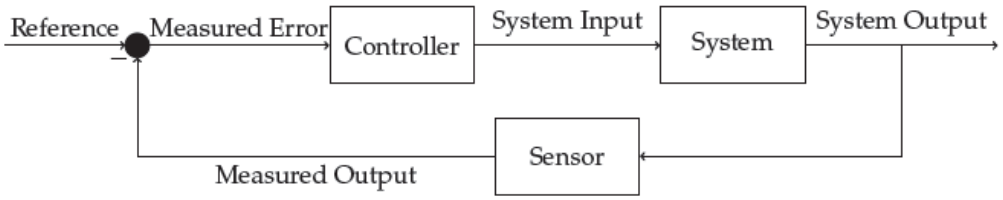


Figure 2.2: A feedback loop to control a dynamic system

The advantages of closed-loop systems are listed below:

- Disturbance rejection: The system is not as much affected by disturbances as an open-loop system
- Guaranteed performance: When the model structure does not perfectly match the real process and the parameters are not accurate, the performance is not affected
- Processes that are unstable can be stabilized using feedback control
- The system is not as sensitive to parameter variations as in open-loop systems

• The performance of the reference tracking is improved

2.3.1 PID Controller

The most used feedback controller is the proportional-integral-derivative (PID) controller. As the name suggests, this controller consists of three separate parameters: A proportional value P, an integral value I and a derivative value D. P depends on the present error, whereas I is derived from the accumulation of past errors and D is a prediction of future errors. Some applications only use one or two of these values and are called correspondingly.

The equation for a PID controller with the output of the system being u(t) is:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t)$$

The proportional gain Kp changes the output proportionally to the current error value e(t). Larger values of the gain typically mean a faster response. The integral gain Ki is proportional to both the magnitude of the error and the duration of the error. A larger integral gain implies that steady state errors are eliminated more quickly, but a larger overshoot is achieved. The derivative gain Kd is proportional to the rate of the change of the error. Larger values decrease the overshoot. However, larger values also slow down the response and can make the system instable.

2.3.2 PV Controller

A proportional-velocity (PV) controller is a special form of the proportional-derivative (PD) controller. Instead of using the rate of the change of the error and multiply it by a derivative gain, the velocity gain is multiplied by the speed of the object to be controlled. Given that x(t) is the position of the object at time t, the equation for the controller is then:

$$u(t) = K_p e(t) + +K_v \frac{d}{dt} x(t)$$

The error function e(t) in this case is the difference between the current position and the reference position.

2.4 Lego Mindstrom NxT Kit

The car used in the thesis is implemented using the Lego Mindstorms kit. This set enables to build and program real-life robotic solutions. Includes the programmable NXT Brick, providing on-brick programming and data logging, three interactive servo motors, ultrasonic, sound, light and two touch sensors, a rechargeable battery, connecting cables, and full-colour building instructions

Key Features:

- Developing solutions, selecting, building, testing and evaluating
- Brainstorm to find creative alternative solutions
- Learn to communicate, share ideas and work together
- Hands-on experience with sensors, motors and intelligent units

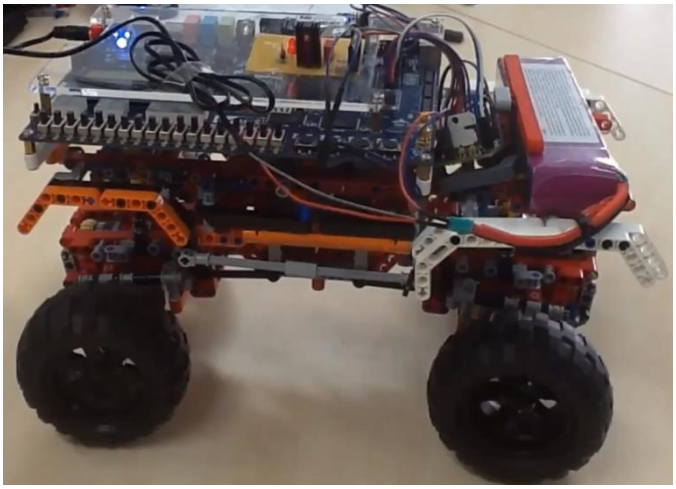


Figure 2.3: Lego Car

2.5 Roving Networks RN134 Wi-Fi Module

Figure 2.4: RN 131

In order to facilitate wireless control of the Lego Car the RN-131 wireless module is used .The RN-131 module is a standalone, embedded wireless 802.11 b/g networking module. With its small form factor and extremely low power consumption, the RN-131 is perfect for mobile wireless applications such as asset monitoring, GPS tracking, and battery sensors. The WiFly module incorporates a 2.4-GHz radio, processor, TCP/IP stack, real-time clock, crypto accelerator, power management, and analog sensor interfaces as shown in Figure. The module is preloaded with software to simplify integration and minimize application development. In the

simplest configuration, the hardware requires only four connections (PWR, TX, RX, and GND) to create a wireless data connection. Additionally, the sensor interface provides temperature, audio, motion, acceleration, and other analog data without requiring additional hardware. The module is programmed and controlled with a simple ASCII command language. Once the module is set up, it can scan to find an access point, associate, authenticate, and connect over any Wi-Fi network.[12]

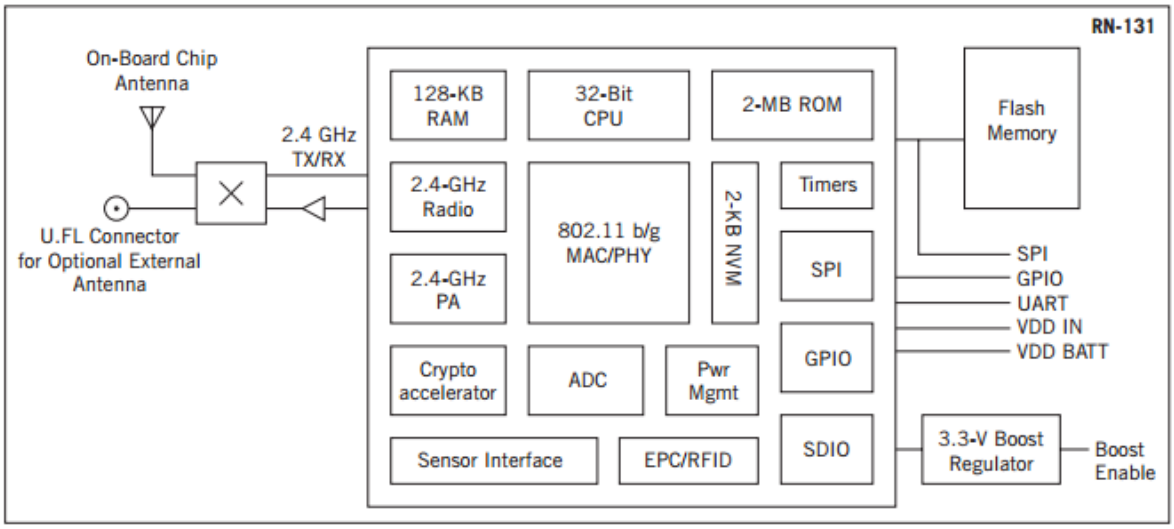


Figure 2.5: Block Diagram RN 131

Some important features of the module that make it a suitable choice for the application are as follows:

- Host data rate up to 1 Mbps for the UART
- Intelligent, built-in power management with programmable wakeup
- Real-time clock for time stamping, auto-sleep, and auto-wakeup
- Configuration over UART using simple ASCII commands
- Remote configuration over WiFi using Telnet
- Secure WiFi authentication using WEP-128, WPA-PSK (TKIP), or WPA2-PSK (AES)
- Built-in networking applications—DHCP, UDP, DNS, ARP, ICMP, TCP, HTTP client, and FTP client
- 802.11 power saving and roaming functions
- On board ECOS -OS, TCP/IP stacks
- 8-Mbit flash memory and 128-KB RAM
- 10 general-purpose digital I/O pins
- 8 analog sensor interfaces

2.6 Motor Back EMF(Electromotive Force) Measurement

Back-EMF refers to using the voltage generated by a spinning motor (EMF) to conclude the speed of the motor's rotation. This can be used in motion control algorithms to modulate the velocity or to compute the angular distance the motor has traveled over time. This article attempts to describe this form of motion control feedback in more detail. Typically a motor takes power in the form of voltage and current and converts the energy into mechanical energy in the form of rotation. With a generator, this process is simply reversed. A generator takes mechanical energy and converts it into both electrical energy with a voltage and current. Most motors can be generators by just spinning the motor and looking for a voltage/current on the motor windings. [2]

When doing Back-EMF measurements for motion control, this fact that a motor can also be a generator is exploited. The motor is run almost continually as a motor with current being supplied to turn the windings. Occasionally, and for a very short period of time, the process is reversed. The windings are allowed to float and the inertia in the motor keeps it spinning while a measurement of the voltage from the spinning motor/generator is taken.
The voltage observed when the motor is spinning is directly proportional to the speed the motor is running. This fact can be used to "peek" at the motor's speed with no optical encoders or other forms of active feedback.

The Process
Reading the velocity from a motor using Back-EMF requires two alternating steps. First, the motor is run for some period of time by providing current to the windings. This current can be supplied as a constant voltage or a PWM motor input to vary the speed. The second step is to remove the current from the windings and "float" them. This means that there is no active circuit between the windings and any other source/sink. This allows the inertia in the motor and mechanical system to spin the motor long enough to measure the voltage produced by the motor. Typically these two steps are alternated roughly 50Hz (times per second).

The time required for the motor to flip from a motor to a generator depends on the inherent capacitance or stored charge in the induction of the motor windings. This time is typically in the order of a millisecond or two, depending on the conditions. Watching the process on a scope can give you some idea of the timing.
Below we walk through some simulated scope pictures to show these various steps and introduce some terminology used in the Application to control a motor using Back-EMF(Electromotive force).

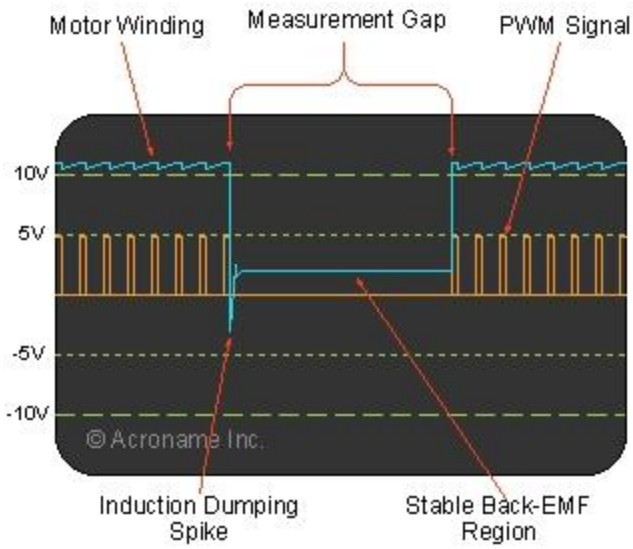
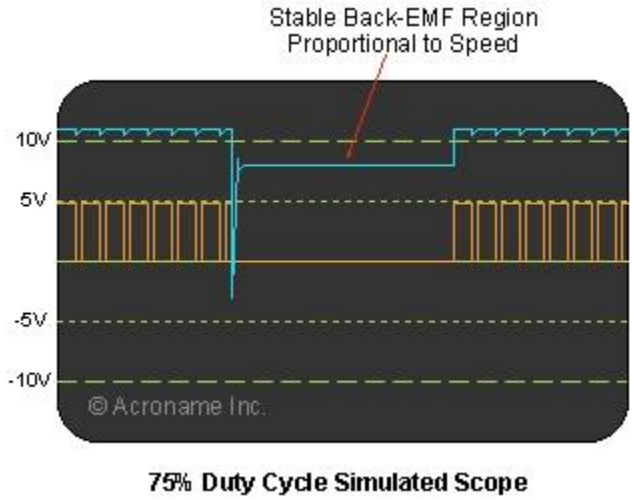
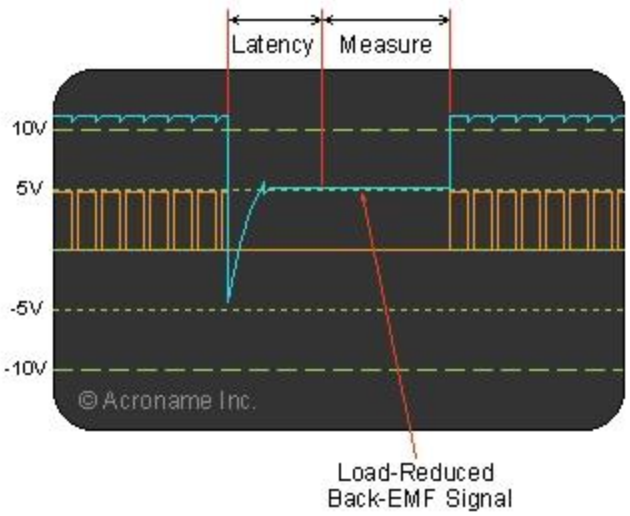


Figure 2.6: Back EMF(Electromotive force) Measurement[2]

Above we see how a scope might look when running a motor at 1/4 speed. Notice how the Back-EMF rises up to roughly 1/4 of the PWM maximum voltage and stabilizes. If the Back-EMF measurement gap is too long, the motor will begin to slow down and the feedback will drop accordingly.



Next, we see what the scope might look like when running the motor at 3/4 speed. Notice how most everything remains the same but now the stable Back-EMF signal is roughly 3/4 of the maximum motor winding voltage.



75% Duty Cycle with High Load Simulated Scope

Finally, we see what the 3/4 speed case might look like when the motor is under load. Since the motor has a great deal of current developing a large induction in the windings when it is under load, the inductive spike is bigger and the stable Back-EMF region takes longer to achieve as this larger induction must be dumped from the windings first before the Back-EMF region stabilizes. Proper tuning of the latency before taking the measurement is important to minimize the measurement gap while allowing enough time for stable Back-EMF measurement.

Back-EMF Measurement Circuits

The Back-EMF measurement circuit can be a bit challenging. The circuit needs to handle the (possibly large) voltages of the motor and convert them into a range that the A/D inputs of a microcontroller can handle. In addition, the windings of the motor invert when the motor direction changes so the circuit needs to both adjust the voltage range and create an input offset so that the neutral (not spinning) voltage output of the measurement circuit centers around a known value. If you are only running the motor in one direction, this circuit can be as simple as a resistor ladder to scale the voltage to the A/D range. If the motor is running bi-directionally, a more sophisticated circuit is required. There are probably as many ways to measure the voltage in a Back-EMF circuit as there are potential motor, direction, and voltage combinations. The key is to ensure that the measurement is passive so it doesn't affect the motor and that it is fast so that the motor can spend most of the time running.

Limitations

Back-EMF velocity measurement is novel and great if you don't have an encoder on your motor. Good quadrature encoders like those used in the Garcia robot will typically outperform the Back-EMF measurement method described here. In addition, encoders can give absolute position information whereas Back-EMF can only report velocity. Position must be computed through integrating the velocity over time which has significant limitations. One other limitation of Back-EMF velocity control is that it effectively reduces the maximal duty cycle that can be

obtained from the motor as it requires the motor to be turned off at times during the operation to facilitate the measurements.

Chapter
3. Concept Work: AUTOSAR on FPGA

The automotive industry is already designing electronics with two crucial standards in mind: AUTOSAR, for handling embedded-system complexity by means of appropriate software and hardware architectures, and the ISO 26262, which governs functional safety. Relevant technical concerns adopted from ISO 26262 and released in AUTOSAR include the detection and handling of safety issues like hardware faults at runtime; abnormal timing and the broken logical order of execution of applications; and data corruption, among others.

3.1 Automotive Open System Architecture (AUTOSAR)

The standardization of the ECU system architecture that AUTOSAR promotes is unavoidable if automakers are to manage the increasing functional complexity of vehicles in a cost-efficient way. It enables the high-level integration of functions distributed in ECUs and the reuse of software components. The main goal of AUTOSAR is to define a uniform ECU architecture that decouples the hardware from the software. In this way, AUTOSAR boosts the reuse of software by defining interfaces that are hardware independent. In other words, a software component written in accordance with the AUTOSAR standard should work on any vendor's microcontroller, provided it has been properly integrated into an AUTOSAR-compliant runtime environment.[14]

This feature delivers increased flexibility to the automaker. Car manufacturers can exchange equivalent versions of the same software module developed by different suppliers throughout their vehicle platforms in a transparent way, thanks to inherent plug-and-play characteristics, and without causing side effects in the behavior of the rest of the functions within the vehicle. In the end, hardware and software are highly independent of each other. This decoupling occurs by means of abstracted layers interconnected through standard software APIs. Figure 3.1 shows the functional-layers breakdown that AUTOSAR defines.

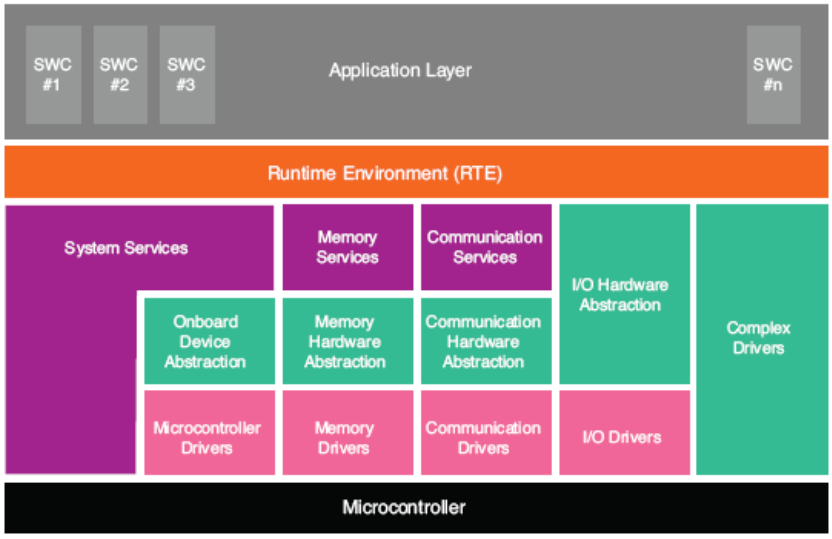


Figure 3.1: The AUTOSAR layer-based model, from MCU to application layer [15]

At the bottom, in black, is the hardware or physical layer, consisting of the MCU itself—that is, the CPU and some standard peripherals attached to it. Above the microcontroller, the basic

software (BSW) is decomposed in three layers: microcontroller abstraction layer (MCAL) in pink, ECU abstraction layer (ECUAL) and complex drivers in green, and services layer (SRV) in purple. These three layers are organized, in turn, into several columns or stacks (memory, communication, input/output and so on).

Close to the hardware components is the microcontroller abstraction layer. As its name suggests, this layer abstracts the MCU. The goal is to have a hardware-independent API that handles the hardware peripherals present in the microcontroller. Next up, the ECU abstraction layer abstracts the other smart devices placed in the ECU board, typically in contact with the MCU (for example, system voltage regulator, smart switching controllers, configurable communication transceivers and the like). Next, the third layer is the services layer. This layer is almost hardware independent and its role is to handle the different types of background services needed. Examples are network services, NVRAM handling or management of the system watchdog. With these three layers, AUTOSAR defines a set of basic software functions that sustain, under a specific hardware platform, all the functionality conceived from higher levels of abstraction to the automotive ECU.

The fourth layer, the runtime environment (RTE), provides communication services to the application software. It is composed of a set of signals (sender/receiver ports) and services (client/server ports) accessible from both the upper layer of the BSW and the application layer (APP). The RTE abstracts the application from the basic software and it clearly delimits the line of the software-stacked architecture that separates the generic and exchangeable software code (APP) from the particular and hardware-dependent code (BSW). In other words, the RTE makes it possible to isolate the software application from the hardware platform; therefore, all software modules running above the RTE are platform-independent.

Above the RTE, the software architecture style changes from layered to component-based through the application layer. The functionality is mainly encapsulated in software components (SWCs). Hence, standardization of the interfaces for AUTOSAR software components is a central element to support scalability and transferability of functions across ECUs of different vehicle platforms. The standard clearly specifies the APIs and the features of these modules, except for the complex drivers. The SWCs communicate with other modules (inter- or intra-ECU) only via the runtime environment.

As ECUs continue to integrate ever more functionality, FPGA devices can be a sensible alternative to single- or multicore MCUs. This overview of the different AUTOSAR layers may hint at the benefits that designers can extract from this architecture when deploying it in programmable logic. Let’s take a closer look at how our design could potentially implement a solution based on custom static hardware (flash- or SRAM-based FPGA technology), and then extend this approach to a runtime reconfigurable hardware implementation (SRAM-based partially reconfigurable FPGA).

3.2 ISO26262 (Safety Standard)

ISO 26262 is a Functional Safety standard, titled "Road vehicles -- Functional safety".[16]

Functional safety features form an integral part of each product development phase, ranging from the specification, to design, implementation, integration, verification, validation, and production release. The standard ISO 26262 is an adaptation of the Functional Safety standard IEC 61508 for Automotive Electric/Electronic Systems. ISO 26262 defines functional safety for automotive equipment applicable throughout the lifecycle of all automotive electronic and electrical safety-related systems.
The first edition, published on 11 November 2011, is intended to be applied to electrical and/or electronic systems installed in "series production passenger cars" with a maximum gross weight of 3500kg. It aims to address possible hazards caused by the malfunctioning behavior of electronic and electrical systems.

The standard consists of 9 normative parts and a guideline for the ISO 26262 as the 10th part.
Like its parent standard IEC 61508, ISO 26262 is risk based safety standard, where the risk of hazardous operational situations are qualitatively assessed and safety measures are defined to avoid or control systematic failures and to detect or control random hardware failures, or mitigate their effects.

- Provides an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases.
- Covers functional safety aspects of the entire development process (including such activities as requirements specification, design, implementation, integration, verification, validation, and configuration).
- Provides an automotive-specific risk-based approach for determining risk classes (Automotive Safety Integrity Levels, ASILs).
- Uses ASILs for specifying the item's necessary safety requirements for achieving an acceptable residual risk.
- Provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety is being achieved.

3.3 AUTOSAR on FPGA

This a pioneering approach to designing an automotive ECU using a programmable FPGA device rather than an MCU-based platform as the foundation for an ECU that conforms to both the AUTOSAR and ISO 26262 standards. This approach explores key features such as parallelism, customization, flexibility, redundancy and versatility of the reconfigurable hardware. After designing the concept, we hope to implement it in a prototype. For this purpose, the Altera Cyclone® V SOC is an excellent candidate. Altera SoCs integrate an ARM-based hard processor system (HPS) consisting of processor, peripherals, and memory interfaces with the FPGA fabric using a high-bandwidth interconnect backbone. The Cyclone® V SoCs [19] reduce system power, system cost, and board size while increasing system performance by integrating discrete processor, FPGA, and digital signal processing (DSP) functions into a single, user customizable ARM-based system on a chip (SoC). Altera SoCs provide the ultimate combination of hardened intellectual property (IP) for performance and power savings, with the flexibility of programmable logic. This FPGA platform meets the needed requirements and also features on-

chip communication controllers commonly used in the vehicle networks, like CAN and Ethernet.[17]

3.3.1 System Architecture

The AUTOSAR and ISO 26262 directives are mainly driven from a software development perspective and oriented toward computing platforms based on microcontroller units. However, the introduction of hardware/software co-design and reconfigurable computing techniques can bring some advantages in this arena. While standard MCUs are often the hardware platform of choice in automotive ECUs, the decreasing cost of new FPGAs, along with the fact that some of them harbor hard-core processors inside, makes these devices a solid solution for a massive deployment in this market. Moreover, the automotive trend of continually incorporating new embedded functionality points to a need for parallel computing architectures. That is particularly true in the infotainment sector today, where high-speed digital signal processing is opening doors to FPGA technology. Programmable logic suppliers like Xilinx,Altera and EDA tool vendors such as MathWorks show clear interest in this field.

Aiming to bring all the advantages of reconfigurable hardware to automotive applications, we describe the potential of this technology through a use case focused on one of the most important ECUs found in the automobile computing network concerning deployment of end-user functions: a body controller module. This ECU, also called a body domain controller, is responsible for synthesizing and controlling the main electronic body functions found in the vehicle, such as the windshield wiper/washer, lighting, window lift, engine start/stop, exterior mirrors and central locking. Our goal was to design an AUTOSAR-compliant ECU system equipped with safety-critical functions on an FPGA platform.

3.3.2 ECU design on FPGA-based static hardware

The AUTOSAR architecture fits well in an embedded system composed of a CPU, memory and programmable logic. The ECU platform requires one CPU or host processor to manage the application and process the different functions distributed in software components through the APP layer. At the same time, the MCU layer and part of the basic software layer can be synthesized in hardware in the programmable logic fabric. Hence, in addition to implementing standard peripherals attached to the CPU, other custom peripherals and coprocessors can coexist in hardware and be totally or partially managed in software.

Dedicated coprocessors or core processors are suitable from the point of view of functional safety too, since they can implement functionality with inherent freedom of interference in hardware, bringing a high level of flexibility—and even redundancy when required—to the system design. Also, the intermediate RTE layer can be synthesized in RAM blocks distributed along the FPGA or in flip-flops embedded in the logic cells of the device, as well as external memory. Moreover, it’s easy to design the RTE signal interfaces to allow both read and write operations (via single-port memories) or to restrict the architecture to either read or write transactions only (by means of single dual-port memories with two independent read and write ports) as a protective measure against interference, like the counterpart sender and receiver software ports that AUTOSAR defines.

Figure shows the proposed porting of the MCU-based AUTOSAR ECU architecture into an Extensible Processing Platform (EPP) or FPGA device, keeping a clear system partitioning in layers. Below the RTE layer are the operating system (OS), memory stack, communication stack, I/O stack and so on. Above it are the software components, which implement the applications and communicate with the RTE through AUTOSAR interfaces.

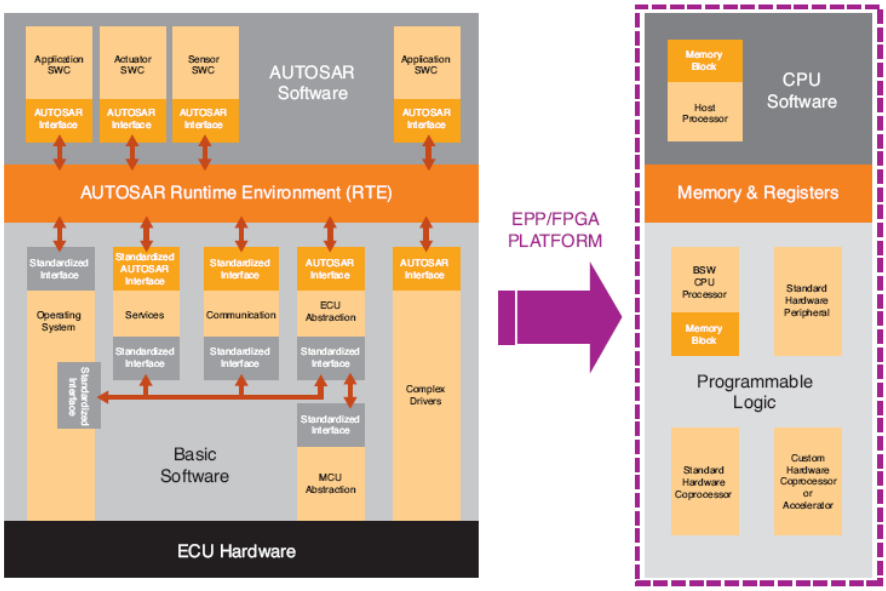


Figure 3.2: Porting of the AUTOSAR ECU architecture to an FPGA platform[17]

Due to the inherent complexity of the AUTOSAR architecture, its deployment demands powerful embedded computing platforms. Today, the typical ECU implementation is based on a 32-bit single-core processor on an MCU platform. However, more and more, a single core is not going to be enough to deliver all the computational power demanded. Yet the use of multicore CPUs can degrade performance if they share the program/data memory through a multiprocessor bus with arbitration mechanisms, often necessitating a highly complex solution.

Instead, a new alternative based on programmable logic and composed of only one single-core processor playing the role of host CPU but surrounded by more-intelligent peripherals, coprocessors or even slave processors can be used. All these computing units can be instantiated in the FPGA fabric as new soft-core processors, such as the Altera NIOS II, which run their own code from dedicated RAM blocks of the FPGA (separate soft-core processors with dedicated program memories), or as made-to-measure hardware accelerators. In both cases, the topology is one host CPU with smart peripherals that offload some of the CPU's tasks, reducing the complexity of the system. Thus, the host CPU manages the whole APP layer in software while the custom peripherals take charge of the BSW layer and run independently of each other, in parallel and autonomously. Moreover, it's a good idea to design these peripherals to make the software execution of the host CPU more linear—that is, without creating excessive interrupts coming from the peripherals to request the CPU's attention via interrupt service routines. Figure

3 shows the block diagram of the system and its component breakdown into functional units synthesized in one FPGA device.

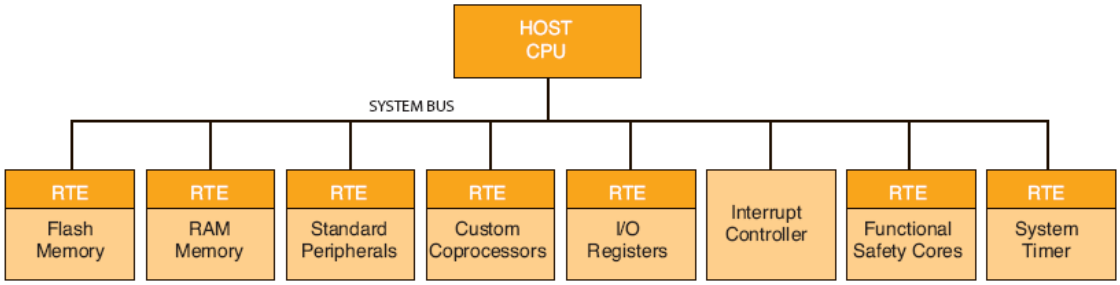


Figure 3.3: Block diagram of an automotive ECU deployed in an FPGA[17]

This approach can attain system performance comparable to that of a multiprocessor platform but with the level of simplicity (regarding software development and maintenance) of a single-core processor. This trade-off is possible by using dedicated hardware to build more powerful and autonomous custom coprocessors that work in parallel with the host processor.

Conceptually, to simplify the idea, it is possible to split the architecture of these systems into two main layers—the high layer and the low layer—separated by the RTE interface. The high layer corresponds to the application layer of AUTOSAR, composed of software components that manage the end-user functions in the vehicle. The low layer comprises the hardware and the basic software up to the RTE link. The application layer can represent, in relative figures, around 90 percent of the high-level functionality within the vehicle, and all this source code—above the RTE—is reusable.

At the same time, the low layer comprises all those features that grant flexibility and versatility to the high layer. That is, the low layer performs the customization of all that reusable functionality in a particular hardware platform. As such, the high layer is essentially a set of software functions that implement the control of some vehicle loads, sensors and actuators by means of algorithms implemented in finite state machines (FSMs). These algorithms are executed cyclically by a CPU and scheduled in software tasks that the OS controls.

The low layer is also responsible for implementing the drivers of all the standard peripherals attached to the CPU—for example, A/D converter, PWM controller, timer or memory controller—to make the abstraction of the high layer feasible. This low layer involves the management of events that need to be served in real time. In this regard, programmable logic can bring some added value. The idea is to reach a host CPU able to process the application as a simple sequence of software functions not influenced by external events typically derived from hardware, but reading or writing RTE signals periodically to evolve the FSMs accordingly. The low layer would hide these hardware events and manage them, preprocessing them and updating certain signals in the RTE or performing certain actions in real time as a result, following its specific tasks scheduling.

Attaching custom hardware controllers to the system CPU minimizes the need for shared resources, if such controllers can work in an autonomous way. From an OS point of view, this helps to reduce the system complexity (avoiding arbitration, latencies, retry mechanisms and the like).

Another advantage is that dedicated hardware can more simply implement certain functionality that is typically performed in software through multithreading, since concurrency is a feature more inherent to hardware than to software. Furthermore, the flexible hardware can be used to reduce execution time by hardwiring computationally intensive parts of the algorithm by means of parallel and pipelined hardware implementations instead of sequential software approaches on Von Neumann machines.You can reduce the software complexity of an automotive ECU by granting a major level of intelligence to the peripherals and hardware coprocessors synthesized in the MCU and BSW layers, freeing up CPU time.

In parallel with the growing complexity of the ECU platforms, the number of I/O lines the system demands is also increasing. In this regard, an FPGA brings a clear advantage over a microcontroller since it typically offers far more user pins. This point is often relevant in MCU-based ECUs, because they need to extend the MCU inputs and outputs with external chips that perform the parallel-serial data conversion, like digital shift registers or analog multiplexers. An FPGA lets you skip all these satellite components, reducing thus the bill of materials as well as the PCB dimensions of the electronic board.

State-of-the-art FPGA devices already incorporate analog-to-digital converters. This feature is interesting in automotive design since many ECUs make use of analog signals (for example, battery voltage) to implement part of the needed functionality. The presence of ADCs in programmable logic devices opens new application fields for FPGAs.

Like MCUs, FPGAs offer remote update capability. However, it is important to note that in this case, the bit stream downloaded into the FPGA relates not only to software code but also to hardware circuitry. This means that, once the product is in production, it is still possible to change the hardware design by means of system updates or upgrades. The automotive industry appreciates such flexibility, since it also enables bug fixes (in this case, both hardware and software) after the product launch.

In any ECU that embeds a function qualified as safety-relevant under ISO 26262, the hardware and software involved in this implementation must fulfill a certain level of protection depending on how it is categorized. From the software point of view, it is necessary to demonstrate freedom from interference—that is, the non-safety-relevant code running in an ECU must not corrupt the operation of any code inside the same ECU classified as safety-relevant. This isolation is necessary to guarantee correct execution of safety-related and non-safety-related functions running side-by-side on the same processor. Often, it’s easier to manage these measures more flexibly in programmable logic than in MCUs.

Regarding memory protection strategies oriented to functional safety, it is necessary to guarantee write access to certain safety-related signals only from authorized safety software components. In the context of MCU devices, memory partitioning provides a fault-containment technique to

separate software applications from each other to avoid any data corruption among them. Programmable logic will likely make it possible to implement a more efficient self-protection mechanism. It is possible to manage the RTE buffer related to safety signals through dedicated single dual-port memories so that the data is written from the write port and read from the read port. In this way, it is possible to implement dedicated hardware controllers that put different restrictions on writing or reading those signals from the software side. The same approach can be implemented with registers.

The possibility of introducing custom hardware solutions in the ECU system is a big advantage of the FPGA approach, especially for safety-related features. In this case, with regard to I/O pins and GPIO controllers, the pin out involved in safety functions can be grouped in made-to-measure I/O ports that are accessed exclusively by safety components inside the ECU, separated from the remaining pins of the device. This is a good way to decouple the safety-critical pins from the non-safety-critical pins of the system, ensuring freedom from interference by design. Any access to non-safety pins cannot corrupt the status of the safety pins, which are managed by safety-relevant code only. This idea is depicted in Figure.

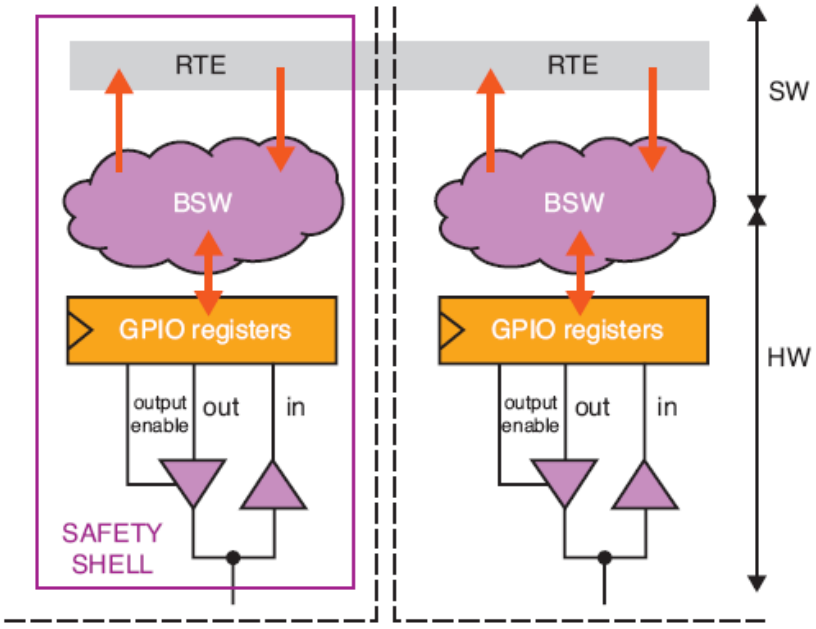


Figure 3.4: Safety Path

Furthermore, it's also possible to tailor the size of each GPIO port to the needs of the application or the software component that handles it, skipping the step of converting a GPIO port into a physical resource that different applications share, as happens with MCU ports. In this way, in an FPGA each application managed by a different SWC (for example, window lift, wiper, exterior mirror, etc.) can have its specific port mapped in specific registers within the system memory map. In MCU platforms this is often not possible, since the ports have a fixed size (typically 8,

16 or 32 bits wide) and are addressed in a word-wise mode, not bit-wise. Therefore, this control register becomes a shared resource that several SWCs access along the program execution. [17]

We can extend the same strategy used for the GPIO controller to other standard peripherals. In this manner, the function partitioning and isolation that AUTOSAR promotes at the high layer with SWCs can extend also to resources of lower layers by means of programmable hardware. Such a technique is impossible with frozen hardware solutions based on standard MCU devices.

The same decoupling strategy we described for MCU standard peripherals can be applied to all the channels or data paths of a safety function. This feature is particularly interesting as a way of implementing highly categorized safety goals—organized by ASIL level in ISO 26262 (see sidebar)—decomposing them in redundant partitions of a lower ASIL level so that each of these parts, performed in duplicate, is then implemented according to its new level. This safety strategy based on redundancy is another argument for choosing programmable logic, which makes it possible to instantiate several identical and independent processing engines multiple times in the same device. Moreover, the fulfillment of a certain ASIL level is always clearer and easier to prove with architectural approaches (hardware) than by abstract software, especially features like freedom from interference, where design failures like a stack overflow or an incorrect handling of a data pointer in C programming language could introduce unexpected safety integrity problems to the system.

Another design advantage derived from the flexibility of programmable logic and applicable to functional safety is the possibility of implementing triple-modular redundancy (TMR) strategies. This is a commonly known method for single-event upset (SEU) mitigation in aerospace applications. Such a mitigation scheme has three identical logic circuits that perform the same task in parallel, with the corresponding outputs compared through a majority-voter circuit. Hardware offers a very efficient way to implement this strategy.

Also, in a market where cost and power consumption are huge concerns, some programmable logic devices, such as the Altera Cyclone V, support several features to lower the overall system power consumption—some of them inherited from MCU devices. Features like the processing system power-on-only mode, sleep mode and peripheral independent clock domains can significantly reduce dynamic power consumption of the device during idle periods.

Certain programmable logic devices come with a hard-core processor placed in the fabric, enabling designers to initially develop the whole system functionality in software, as they typically would do for an MCU-based platform, and then progressively add more hardware in the design, porting certain parts to programmable logic resources. This methodology enables the designer to build different versions of a solution and realize the advantages of synthesizing some functions in custom hardware with respect to a purely software-based approach.

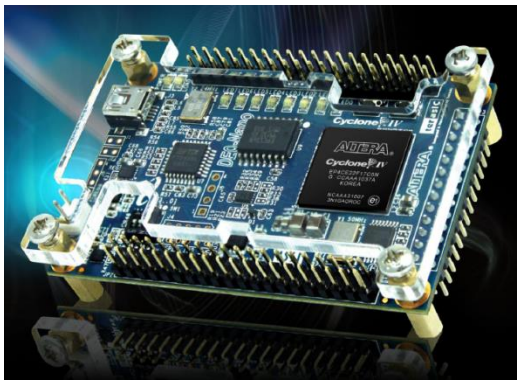
Chapter
4. FPGA Based Implementation

4.1 FPGA Platform Used

The Cyclone IV DE0 Nano Development Board from Altera is used to design the evaluation platform. The DE0-Nano board introduces a compact-sized FPGA development platform suited for prototyping circuit designs such as robots and "portable" projects. The board is designed to be used in the simplest possible implementation targeting the Cyclone IV device up to 22,320 LEs. The DE0-Nano has a collection of interfaces including two external GPIO headers to extend designs beyond the DE0-Nano board, on-board memory devices including SDRAM and EEPROM for larger data storage and frame buffering, as well as general user peripheral with LEDs and push-buttons. [5]

The advantages of the DE0-Nano board include its size and weight, as well as its ability to be reconfigured without carrying superfluous hardware, setting itself apart from other general purpose development boards. In addition, for mobile designs where portable power is crucial, the DE0-Nano provides designers with three power scheme options including a USB mini-AB port, 2-pin external power header and two DC 5V pins. For the analysis and synthesis of the design the Quartus II software also provided by Altera is used. It enables compiling the designs, performing timing analysis, simulating the hardware, placing the pins and routing the design.

The key features of the board:
Featured device
Altera Cyclone® IV EP4CE22F17C6N FPGA
Altera serial configuration – EPCS16(16Mbits)
2 debounced pushbuttons
4-position DIP switch
A/D Converter
NS ADC128S022, 8-Channel, 12-bit A/D Converter



In chapter 3 the main parts of the experiment have been described. To implement the controller on the FPGA the setup has to be extended by some parts. The work done in this thesis includes developing the necessary circuits to interface the hardware described in chapter 3 with the FPGA. The FPGA functions as a control unit. A part of the work also includes the development of the hardware system that is downloaded to the FPGA to fulfill the control task. After developing a hardware system, software architecture can be designed to implement the control in software. To achieve the goal of this chapter, the FPGA, which is the control unit, and the Motor Controller circuitry are added to the basic system, resulting in the system in figure 4.1. The Sensor less controller only uses the Back EMF signals of the DC motors to control the speed of the Lego Car.

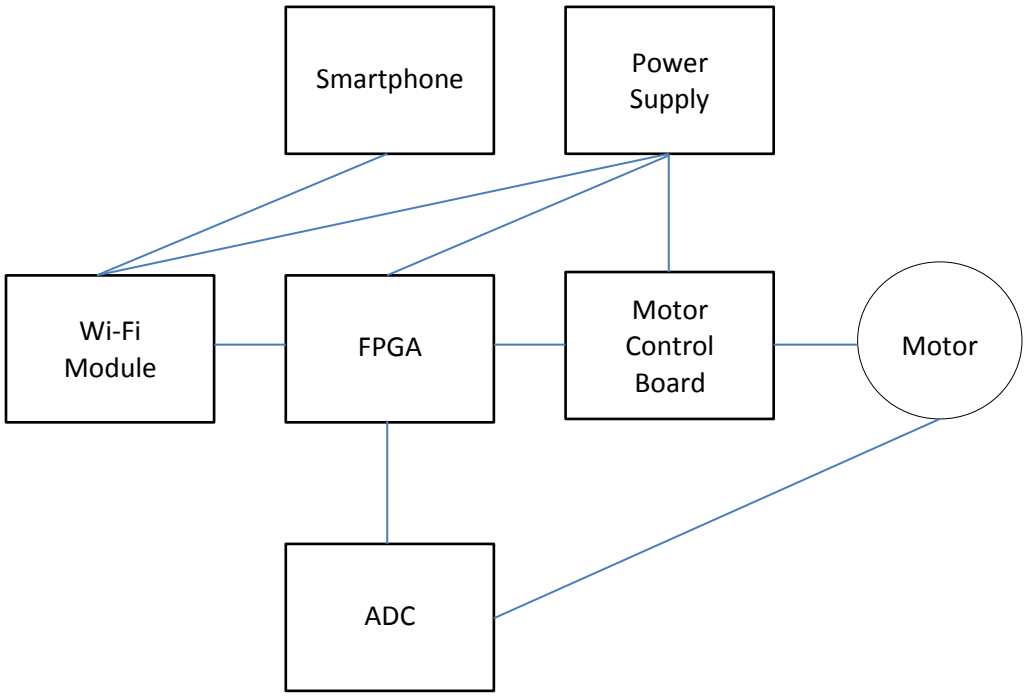


Figure 4.1: Components of the Sensor less System

The goal is to drive the ecar with the right speed and direction by using the signals from the smartphone and calculate the speed by measuring the BEMF of the DC motors. By generating interrupts that are handled by the FPGA the Back EMF signals can be measured for speed measurement. The FPGA has to be configured with a system that can handle interrupts and generate the right motor signals to be able to control the ecar.

This chapter is divided into two parts: Hardware and Software. In the hardware part the system designed is explained. A brief description of each component of the system and its function is given. This part also includes the interrupt handling and the interfaces of the system. The software part explains the interrupt service routines done in software and the control loop. The implementation of the controller in software is also a part of this section.

4.2 Hardware Architecture

Before the controller can be implemented in software the necessary hardware system has to be designed. The design can then be downloaded to the FPGA and can be used to execute the controller software. The FPGA also has to be connected to the experiment. This section describes the components needed to build the system and how the FPGA is connected to the system.

4.2.1 System Design

SOPC Builder

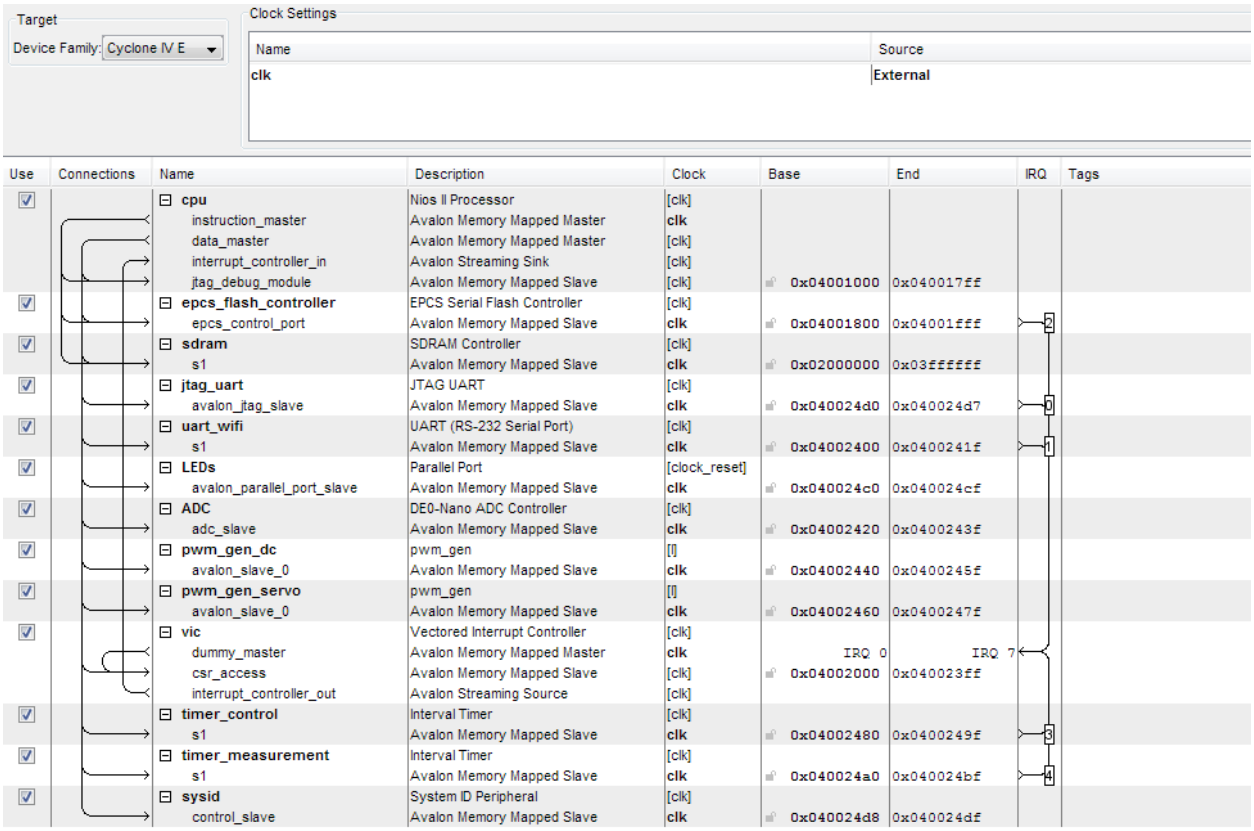


Figure 4.2: The components in the SOPC Builder

The design of the system is done using the SOPC Builder from Altera. This is the software that automates connecting soft-hardware components to build a complete system that can run on any FPGA. There are libraries of pre-made components that can be used as well as user-defined components. The Avalon Bus is used to connect the components. When generating the system bus arbitration, bus width matching and clock domain crossing are handled automatically by the SOPC Builder. Adding components to the system is done using the SOPC Builder GUI.

After generating the system in the SOPC Builder, the system can be connected to the outside world in the Quartus software by connecting the inputs and outputs of the system to the FPGA pins. A screenshot of the SOPC Builder with the final system is illustrated in figure 4.2. All the components of the system are added and connected as desired. An example of some connections is shown in figure 4.3.

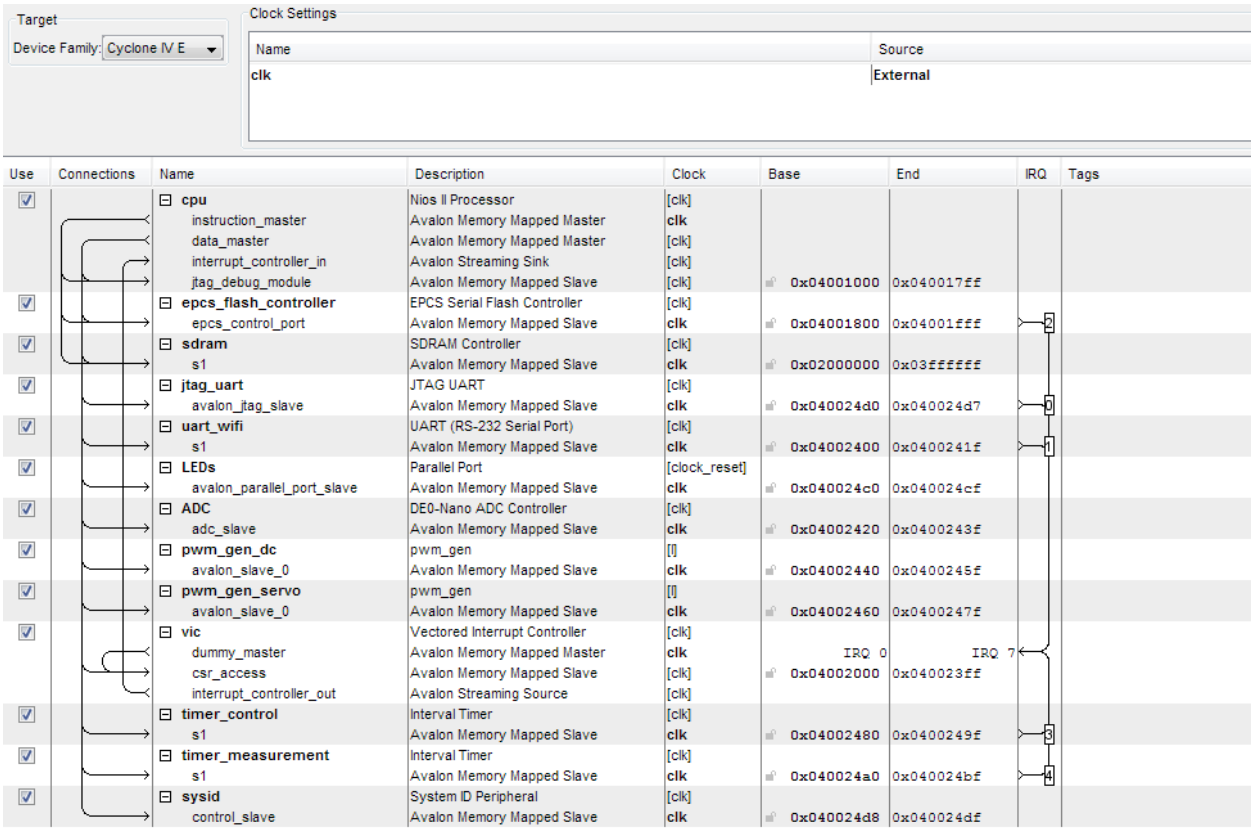


Figure 4.3: Connecting the components in the SOPC Builder

The components are now described briefly.

Nios II processor

A Nios II processor is integrated in the design to be able to run the software in C on the hardware design. Altera offers three different processors: Nios II/e, Nios II/s and Nios II/f. Each of these processors is designed for different applications. While the Nios II/e is designed to achieve the smallest size, the Nios II/f offers the most configuration options and is designed for fast performance. The Nios II/s is the standard core. The different configurations allow for fine-tuning for performance. Considering that size is not an issue in the design and that performance is more important and necessary to handle interrupts and control the pendulum, the Nios II/f core is the better choice. Embedded multipliers and a hardware divide were also included to achieve a greater performance. The reset vector is placed in low-latency on-chip memory. The exception vector is also placed in low latency memory but tightly-coupled memory is used to allow for fast interrupt handling.

The design has a 32MB SDRAM and 2Kb I2C EEPROM. The core is configured to work with an external interrupt controller with seven shadow registers to reduce interrupt response time.

Floating Point Hardware is added to support the calculations of the controller.

Vectored Interrupt Controller
The vectored interrupt controller (VIC) is an external interrupt controller implemented by Altera. It is connected to the Nios II processor and provides higher performance for interrupt handling compared to the internal controller. Interrupts are prioritized in hardware and information about the interrupt with highest priority is sent to the processor. In this design the VIC supports eight interrupts and four bits are used to represent the interrupt level of each interrupt.[6]

On-chip Memory
Memory is needed to store the software. Cyclone III offers on-chip memory blocks that can be used either as RAM or as ROM. These are fast access memory blocks and thus have a low latency. A 2 Kbyte readable and writable (RAM) memory block is included in the design to store the data. The data width is set to 32 bits to match the Nios II processor. The read latency is one cycle because the memory block uses synchronous, pipelined Avalon memory mapped slaves.

JTAG UART
To enable communication between the FPGA and the host PC a JTAG UART is needed. It sends serial character streams and allows for character I/Os and is therefore useful for debugging the controller and the system. Altera provides a JTAG terminal that can decode the JTAG data stream and display the characters sent on screen.

System ID Peripheral
The system ID core provides the system with a unique ID. It is a small read-only device that is used to verify that the executable program was compiled for the actual hardware currently configured on the FPGA. This ensures that the software will run correctly and that the user is warned if the program and the targeted hardware do not match. The core has two registers, one for the ID and one for the time stamp with the generation time.

Interval Timer Core
To compute the values for the controller every millisecond a timer is needed. Altera offers an Interval Timer Core to implement timers. The timers are either 32-bit or 64-bit counters and can either count down once or continuously. They also offer the option to generate interrupts when the timer reaches zero. The system for the pendulum control requires three timers: a system clock timer, a timer for the encoders and a cycle timer.

System Clock Timer
The system clock timer is a 32-bit counter with a timeout period of 1ms. It is a full-featured timer and thus can be started and stopped at processor control.

PWM Control IP

This is a custom IP PWM generator which conforms to the Altera Avalon specification. The IP core can be configured by software on Nios II to generate PWM with different duty cycles and frequencies.

DE0 Nano ADC Controller

The DE0-Nano ADC Controller IP Core manages and controls the signals between the ADC and FPGA, and provides the user with the converted values. The core is usable in both hardware-only and software-controlled versions. It reads each of the input channels of the ADC in ascending order once per update cycle, storing the acquired values locally. Once the update cycle is complete, the new values are available for access. It also provides a number of customizations to the user to control its operation.[4]

The ADC Controller core defines the number of channels in use as a parameter, NUM_CH, which is set by the user when the core is instantiated. Since the core operates by sampling all used channels in series, reducing the number of used channels will reduce the total amount of time required to refresh the values.

The core also allows specification of the SCLK frequency. The user can enter a desired value in the allowed range of 0.8 to 3.2 MHz. Exact matching of the desired SCLK value is not guaranteed, as SCLK is derived as an integer factor of the system clock. Typically, the mismatch will be less than a 5% difference between the desired and implemented value.

PIO Core

The parallel input/output core is used to connect the I/O ports to either on-chip logic or I/O ports of external devices that are connected to the FPGA. It is a memory-mapped interface that connects a slave to general-purpose I/O ports. Each PIO core can support up to 32 I/O ports. It can also be configured to generate interrupt requests (IRQs). The design includes five PIO cores: two encoder PIO cores, the UPM PIO, the button PIO and the LED PIO.

Resulting System

The resulting system is shown in figure 4.4

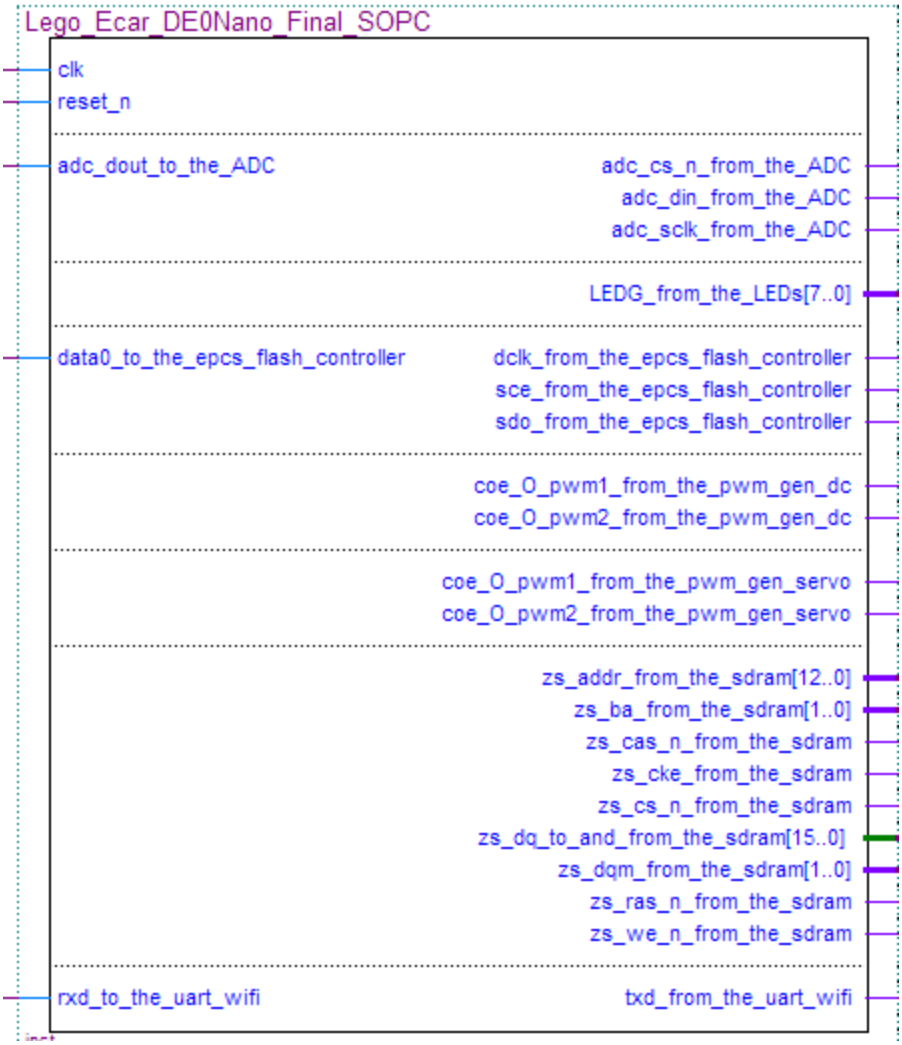


Figure 4.4: System Design for controlling the ecar

The main functions of the controller are as follows:

- To control the Wi-Fi communication via the Wi-Fi control Loop.
- To recognize the messages sent by the android application.
- To control the DC motors using PWM.
- To measure the Back EMF of the DC motors in order to facilitate a closed loop control.

The inputs and outputs are connected to the general purpose I/O pins of the FPGA. These pins are then mapped to the PIOs of the SOPC-system which generate an interrupt that is handled by the Nios II processor accordingly. The Nios II exception processing is configured depending on the chosen hardware interrupt controller. The hardware interrupt controller can either be internal or external.

Internal Interrupt Controller
When the Internal Interrupt Controller is chosen the Nios II exception handling is implemented in a classic RISC fashion. This means that all exceptions are handled by code residing at a single location: The exception address. It is a very simple interrupt controller with non-vectored hardware. As soon as an interrupt request occurs the controller transfers control to the exception address. Which IRQ is asserted is indicated by the hardware and individual interrupts can be masked by the software.

External Interrupt Controller
When the External Interrupt Controller is used the Nios II processor works with a separate external interrupt controller component. This can either be a custom component provided by the user or the vectored interrupt controller offered by Altera. With External Interrupt Controllers hardware and software interrupts are handled separately and hardware interrupts have their own vectors and funnels. Each interrupt can have its own handler or different interrupts can share a handler. The software exceptions are handled the same way as with Internal Interrupt Controllers. The advantage of an External Interrupt Controller is that exception handling can be customized and is therefore optimal for the implemented application. Another advantage is the shorter response time when a hardware interrupt occurs. The response time includes the exception latency that is the time required to respond to an interrupt, and the time required by the hardware abstraction layer to do some overhead tasks. The overhead tasks can be a context save, an RTOS context switch or a dispatch handler. The context saves the registers on the stack. In case an RTOS is implemented the context switch calls some context-switching functions. The dispatch handler determines the cause of the interrupt and transfers control to the corresponding handler or interrupt service routine.

Thus, the response time denotes the time elapsed between the physical event and the system's specific response to that event. Since External Interrupt Controllers can be vectored, the dispatch is handled by hardware and therefore reduces the response time substantially. External interrupt controllers have no effect though on software interrupts. External Interrupt controllers can also be combined with Shadow Register Sets which make it unnecessary to save registers on the stack, thus further reduce the response time.

Interrupt Handling for the eCar
To ensure an optimally functioning controller the response time for each interrupt has to be small enough to not miss the next Back EMF measurement cycle. Therefore, an external interrupt controller is more suitable. For this design the vectored interrupt controller offered by Altera with seven shadow register sets is sufficient. To further improve the software performance of the interrupt service routine, fast memory is used. Tightly-coupled memory is used for this purpose and the ISRs are placed in it. Additionally a separate exception stack is used. The fastest Nios II processor is used to also improve hardware performance.

4.2.2 Level Shifting circuitry

The FPGA is capable of handling I/O voltages that vary from 1:2V to 3:3V but mainly uses I/O voltages of 2:5V. Since the rest of the circuit includes devices like D/A conversion and encoder inputs that require a 5V level and the FPGA cannot handle 5V logic, level shifting is necessary. For translating the inputs from 5V to 2:5V a simple voltage divider with resistors is enough. A drawback of that would have been the electric power consumption.

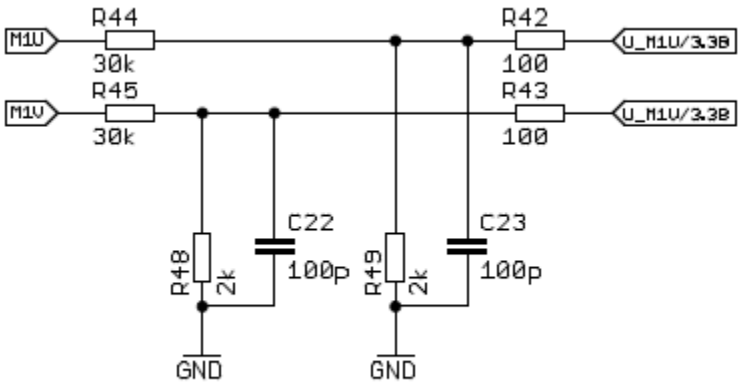


Figure 4.5: Level Shifting Circuit

4.2.3 Driver Circuit-L298

The L298 is an integrated monolithic circuit in a 15- lead Multi watt and PowerSO20 packages. It is a high voltage, high current dual full-bridge driver designed to accept standard TTL logic levels and drive inductive loads such as relays, solenoids, DC and stepping motors. Two enable inputs are provided to enable or disable the device independently of the input signals. The emitters of the lower transistors of each bridge are connected together and the corresponding external terminal can be used for the connection of an external sensing resistor. An additional supply input is provided so that the logic works at a lower voltage.

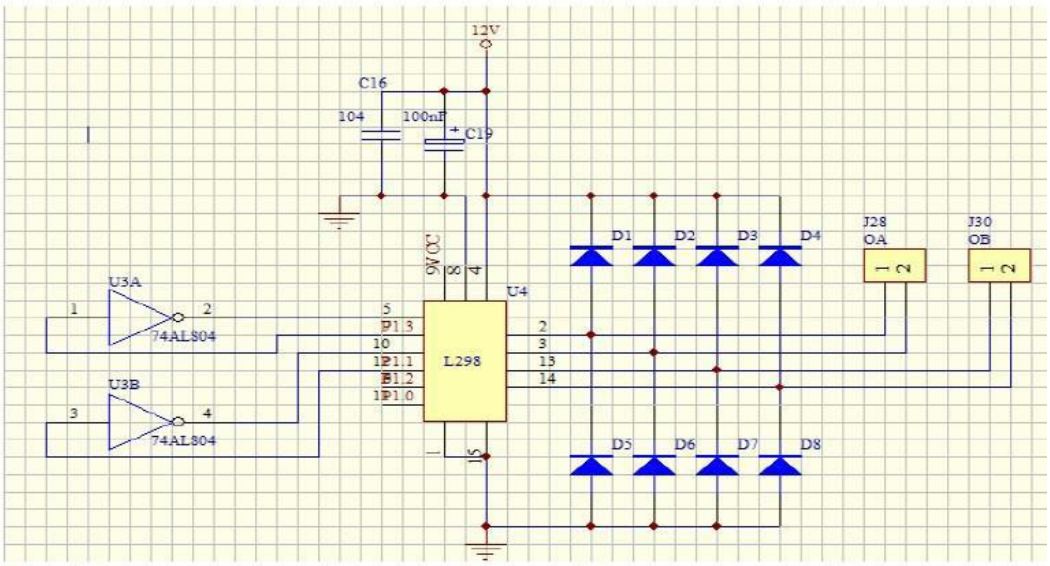


Figure 4.6: Motor Driver Circuit

4.2.4 System Wiring and Setup

The final setup of the experiment consists of a power supply unit, the DE0 NANO FPGA board, the RN131 Wi-Fi Module, the Motor Driver Circuit, the Voltage conversion circuit and the connections to the motors. The connections between the FPGA board and the aforementioned components are shown in the wiring diagram. All connections are on the GPIO 0 of the DE0 Nano board. The back EMF is measured using the on board ADC in the DE0 Nano Board. A voltage conversion circuit is used to convert the back EMF signals to the FPGA levels. The Wi-Fi module is connected to the FPGA board using simple UART.

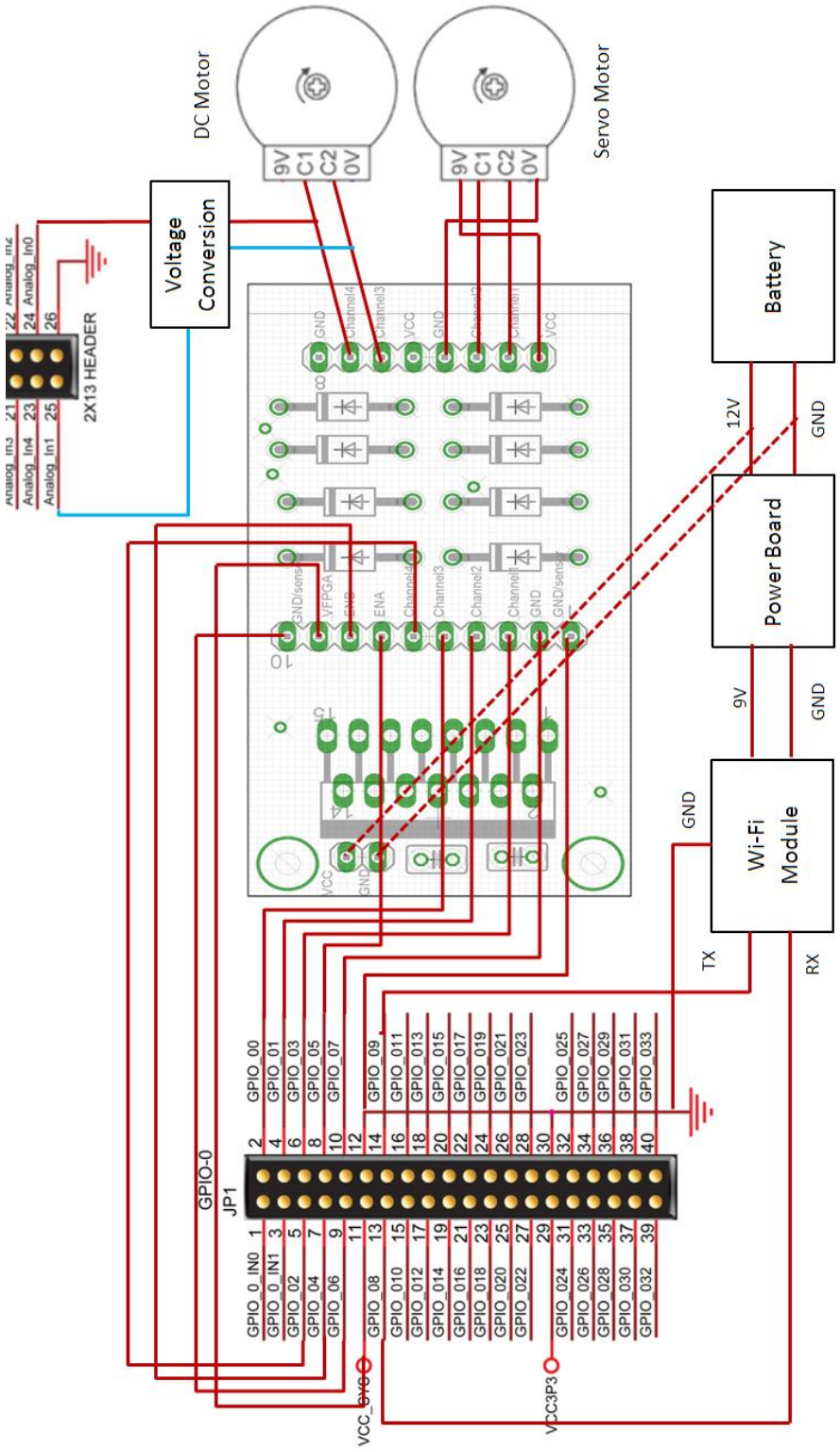


Figure 4.7: A block diagram of the final setup

DE0 Board

NC	1	2	PWM0_DC
NC	3	4	PWM0_Servo
PWM1_DC	5	6	PWM1_Servo
ENA	7	8	ENB
GND	9	10	GND
VFPGA	11	12	GND
TX_Wifi	13	14	RX_Wifi
	GPIO 0		

PWM0_DC	PIN_D3
PWM0_Servo	PIN_C3
PWM1_Servo	PIN_A3
ENB	PIN_B4
GND	PIN_B5
GND	
RX_Wifi	PIN_D5
PWM1_DC	PIN_A2
ENA	PIN_B3
GND	PIN_A4
VFPGA	
TX_Wifi	PIN_A5

Table 4.1: FPGA Pin Names

4.2 Control Software

4.2.1 Software Architecture

After setting up the hardware and configuring the desired system the software can be designed. The requirements for the software are as follows:
• Initialize variables.
• Initialize the ADC.
• Initialize the Wi-Fi Receiver.
• Initialize the Motor PWM Control.
• Enable all.
• Run the control loop every millisecond.
Since the computations are interrupt-based, handling interrupts is also an important aspect of the software architecture.

Interrupt Service Routines
The software must handle three interrupts: Interrupts of the Wi-Fi Module, interrupts of the back EMF Measurement timer and interrupts of the control loop timer. The interrupts are prioritized in hardware. When two interrupts occur at the same time the one with higher priority is handled first. The smaller the value of the priority the higher the priority is. The priorities are set in the SOPC-system described in section 4.1.1 as follows:
• Interrupt of the Wi-Fi Module has priority 0.
• Interrupt of the back EMF Measurement timer has priority 1.
• Interrupt of the control loop has priority 2.

For each of these interrupts an interrupt service routine (ISR) has to be implemented. ISRs are software routines that are invoked by hardware in response to the interrupt. They examine the interrupt and decide how to handle it. After completion of the ISR the processor has to return to its pre-interrupt state. Since the ISR diverts the processors normal execution flow it has to perform very fast to quickly return to the normal execution flow and to avoid slowing down the operation.

The control loop timer has a very simple ISR. In the ISR a flag is set indicating to the main loop that the control software should be executed. Then the status register of the timer is cleared. The ISR of the Wi-Fi module sets the flag so that the receiver loop is checked. The ISR for the back EMF Measurement timer enables the ADC and measures the back EMF of the DC motor which is used for closed loop control.

Control Loop
The task of the control loop is to determine which voltage should be set to drive the ecar in the right direction with the right speed. The flow of the loop is illustrated in figure 4.14 . Depending on the PID controller the values are calculated.

4.2.2 Implementation of the control software

The Nios II Embedded Design Suite (EDS) is the development platform used to write the control program. This is an Eclipse based development toolkit that works for all Nios II processors and

includes proprietary and open-source tools such as the GNU C/C++ tool chain for creating Nios II programs. The advantage of using the Nios II EDS is that it automatically generates the board support package that contains the Altera hardware abstraction layer (HAL) and the device drivers. This insulates the programmer from the underlying hardware.

The Hardware Abstraction Layer

The HAL provides an interface to the peripherals of the system and offers an API with which the access can take place. The HAL is automatically constructed for the hardware thanks to the tight integration between the SOPC Builder and the Nios II software development tools. Changes to the hardware in the SOPC Builder automatically propagate to the HAL configuration making the design less error-prone. When designing a program, the HAL is based on a specific SOPC system in the beginning. In our case the HAL is based on the system described in section 4.1.1 .
The HAL then offers a variety of services stated below:

- Providing familiar C standard library functions
- The HAL API offers an interface to HAL services such as device access, interrupt handling and alarm facilities
- Initialization tasks for the processor and the runtime environment
- Instantiating and Initializing of each device in the system

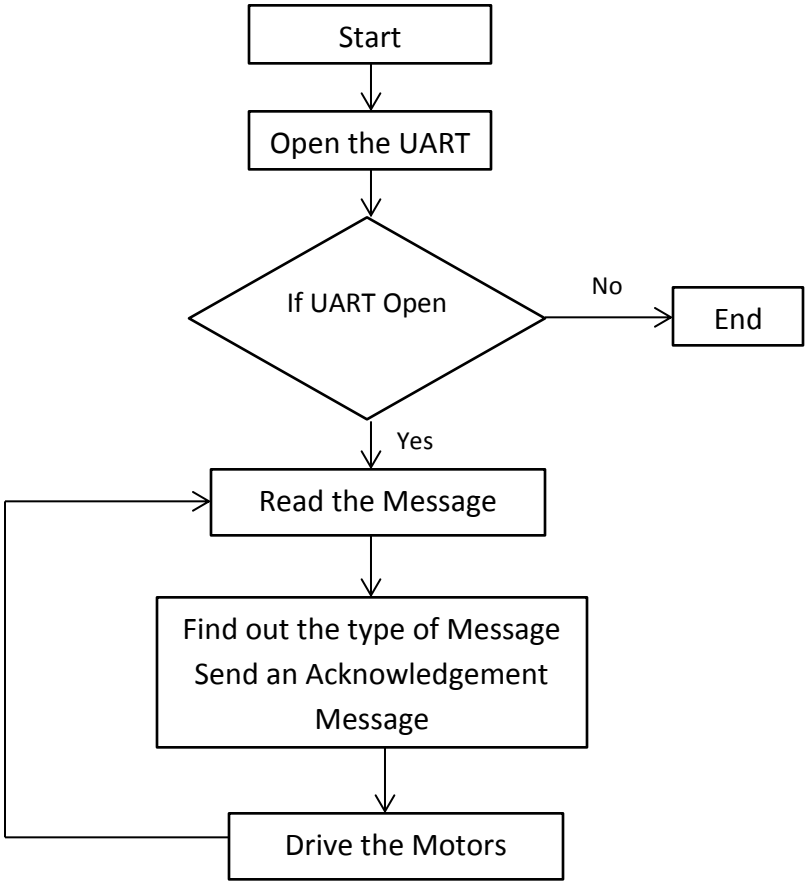


Figure 4.8: Flow chart for the main loop

Using the PWM IP Core Module
The motor_setting.c file provides motor setting function to the user. By calling the motor_setting() function, parameters of PWM waveform such as phase shift, duty cycle, period and enable signal are configured to control the motor.

Arguments of function motor_setting() is introduced here:
- phase: range from 0 to value of period
- duty cycle: set to number=period*percentage of duty cycle
- period: according to the freq of cpu,
- normally the period should be set to the value
- that makes the freq of pwm waveform to be 15 k
- e.x for 50MHz, value should be 3333(0xD05)
- enable:'0' represents off, '1' is on, lease significant bit is for channel 1, second bit is for channel 2, e.x for channel 1 on and channel 2 off, enable=0x1

Accessing and Controlling the ADC
The HAL Driver for the ADC offers five functions for accessing and controlling the ADC. To use these functions, the program must include the statement:
#include "altera_up_avalon_de0_nano_adc.h"

The first step when using the ADC with HAL is to create a device pointer to the ADC. HAL device drivers feature a different variable type for each device; for the ADC controller, the type "alt_up_de0_nano_adc_dev" is used. After creating the pointer, the value is assigned using the alt_up_de0_nano_adc_open_dev (...) function. This function takes in the name of the device and locates it within the system, and returns a pointer to the adc controller. If the default system is used, the string "/dev/ADC" should be used; otherwise, replace ADC with the name of the component as defined in the SOPC builder. The result of this function should be assigned to the device pointer created for the ADC.

Once initialized, the following function is used to read the ADC values. Definition prototype and detailed description for the HAL function are shown below.

alt_up_de0_nano_adc_open_dev
Prototype: alt_up_de0_nano_adc_dev* alt_up_de0_nano_adc_open_dev(const char *name)
Include: <altera_up_avalon_de0_nano_adc.h>
Parameters: name – the ADC Controller name. For example, if the ADC controller name in SOPC is "ADC", then name should be "/dev/ADC"
Returns: The corresponding device structure or NULL if the device is not found.
Description: Open the ADC controller device specified by name.

Timing and Signal Requirements

The ADC128S002 operates on a 16-cycle operational frame, as shown in Figure. The user is required to provide the SCLK, *CS*, and DIN signals to the ADC, and to capture the DOUT signal as it is transmitted.

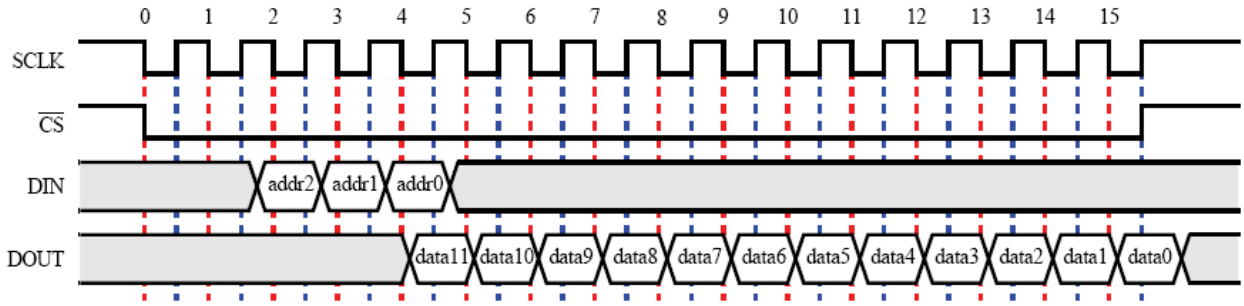


Figure 4.9: Timing requirements for the ADC

The DOUT signal provides the 12-bit converted value of the selected channel. On power-up, channel 0 is selected by default, while subsequent reads will use the address provided in the previous operational frame. The data bits are transmitted in descending order, such that the highest-order bit is delivered first. It is captured by the user on the rising edge of SCLK.

The DIN signal is used to select the channel to be converted in the following frame. It is delivered in descending order, and is captured by the ADC on the positive edges of SCLK. In order to avoid potential race conditions, the user should generate DIN on the negative edges of SCLK. *CS* should be lowered on the first falling edge of SCLK, and raised on the last rising edge of an operational frame. The SCLK frequency is limited to a range of 0.8 to 3.2 MHz in which the ADC will function correctly.

Initializing the interrupts

Then each of the encoders registers its interrupts. This is done as follows:

- Enable interrupts by setting the corresponding value in the IRQ mask register of the encoder driver
- Clear the edge capture register of the encoder driver
- Register the interrupt handler

HAL functions are used to achieve this result. To write the IRQ mask register the HAL macro IOWR ALTERA AVALON PIO IRQ MASK is used with the encoder base value and the value corresponding to channel A of the encoder as arguments. To address the edge capture register IOWR ALTERA AVALON PIO EDGE CAP is used. This also takes the encoder base as the first argument and the value to be written to the register as the second argument of the macro. For registering interrupt handlers the HAL function alt ic isr register is used. This function specifies the interrupt controller to use, the IRQ number and the function that is called when the interrupt is accepted.

To measure the velocity of the cart movement and the velocity of the pendulum movement the encoder timer has to be initialized and the ISR registered. Registering the ISR is done with the alt ic isr register like for I/Os. Additionally the control register of the timer has to be initialized by setting the corresponding flags. The HAL API provides the IOWR ALTERA AVALON TIMER CONTROL function for this purpose. With ALTERA AVALON TIMER CONTROL START MSK as a flag in the function the timer can be started. The flag ALTERA AVALON TIMER CONTROL ITO MSK then generates an interrupt when the timer times out.

Initializing the control loop timer
The control loop timer is initialized in the same way as the encoder timer.

Processing the ADC data
As mentioned in section 4.2.1 the ADC data is processed in the ISR. The processing in the ISR only returns the back EMF voltage value thus further calculations are necessary to retrieve the the speed of the ecar. The measured back EMF value is converted to the pwm value using conversion factors. This pwm value is directly proportional to speed so can be used as the current value for the PID controller. The ADC timer times out every 10 ms.

To convert the back EMF value into PWM or vice versa the following factors are used.

#define MOTOR_BEMF_AT_MAX_NOMINAL_PWM
#define MAX_NOMINAL_PWM

So,

$$pwm = \frac{\text{MAX\_NOMINAL\_PWM} * \text{bemf}}{\text{MOTOR\_BEMF\_AT\_MAX\_NOMINAL\_PWM}}$$

$$bemf = \frac{\text{MOTOR\_BEMF\_AT\_MAX\_NOMINAL\_PWM} * \text{pwm}}{\text{MAX\_NOMINAL\_PWM}}$$

This calculated value of pwm is given as the measured value to the PID controller which is compared with the set value of pwm to control the speed.

PID control

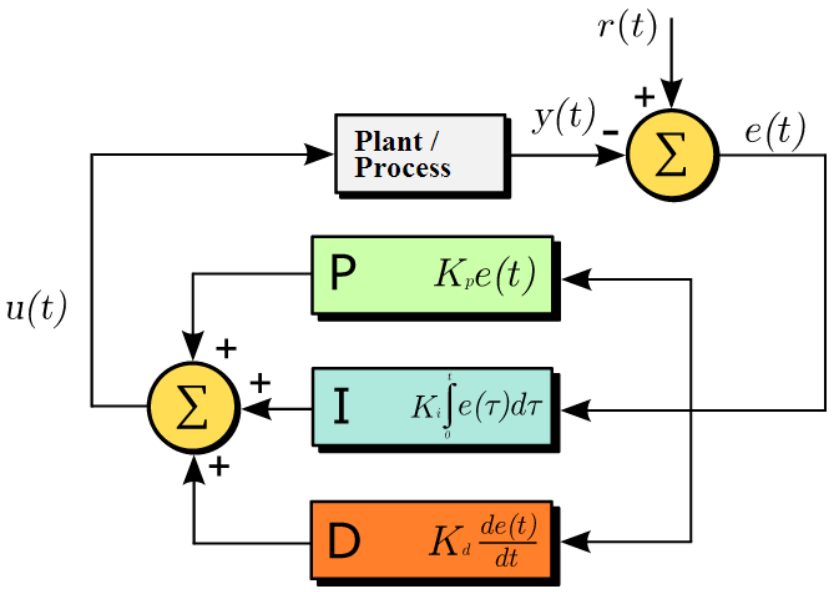


Figure 4.10: PID Controller

The PID controller was implemented on the FPGA using NIOS. A PID controller provides compensation to an existing system by trying to minimize the error between the desired output and actual output. It does this by adjusting the process inputs. A PID controller consists of three forms of compensation, namely, Proportional, Integral and Derivative. The advantages of using a PID controller are many fold. Combining the three forms of compensations we are able to stabilize a potentially unstable system, minimize steady state error and increase system speed respectively.[13]

To implement this on an FPGA, we perform the calculations in discrete time. We read the back EMF at regular time intervals and compared them with the set point. The error value was then fed to the controller that responded accordingly to try to eliminate the error.

Here is a simple software loop that implements a PID algorithm:

```
previous_error = 0
integral = 0
start:
  error = set_speed - measured_speed
  integral = integral + error*dt
  derivative = (error - previous_error)/dt
  output = Kp*error + Ki*integral + Kd*derivative
  previous_error = error
  wait(dt)
```

```
goto start
```

Here two variables that will be maintained within the loop are initialized to zero, and then the loop begins. The current error is calculated by subtracting the measured value of the speed of DC Motor (the process variable or PV) from the current set point (SP) value of the speed. Then, integral and derivative values are calculated and these and the error are combined with three preset gain terms – the proportional gain, the integral gain and the derivative gain – to derive an output value. The output value is converted to PWM value to control the DC motor. The current error is stored elsewhere for re-use in the next differentiation, the program waits a predetermined time, and the loop begins again, reading in new values for the PV and the set point and calculating a new value for the error.

The parameters of the controller are determined by experiments. The following parameters achieved the desired result:
Ki = 0.0168
Kd = 0
Kp = 0.30

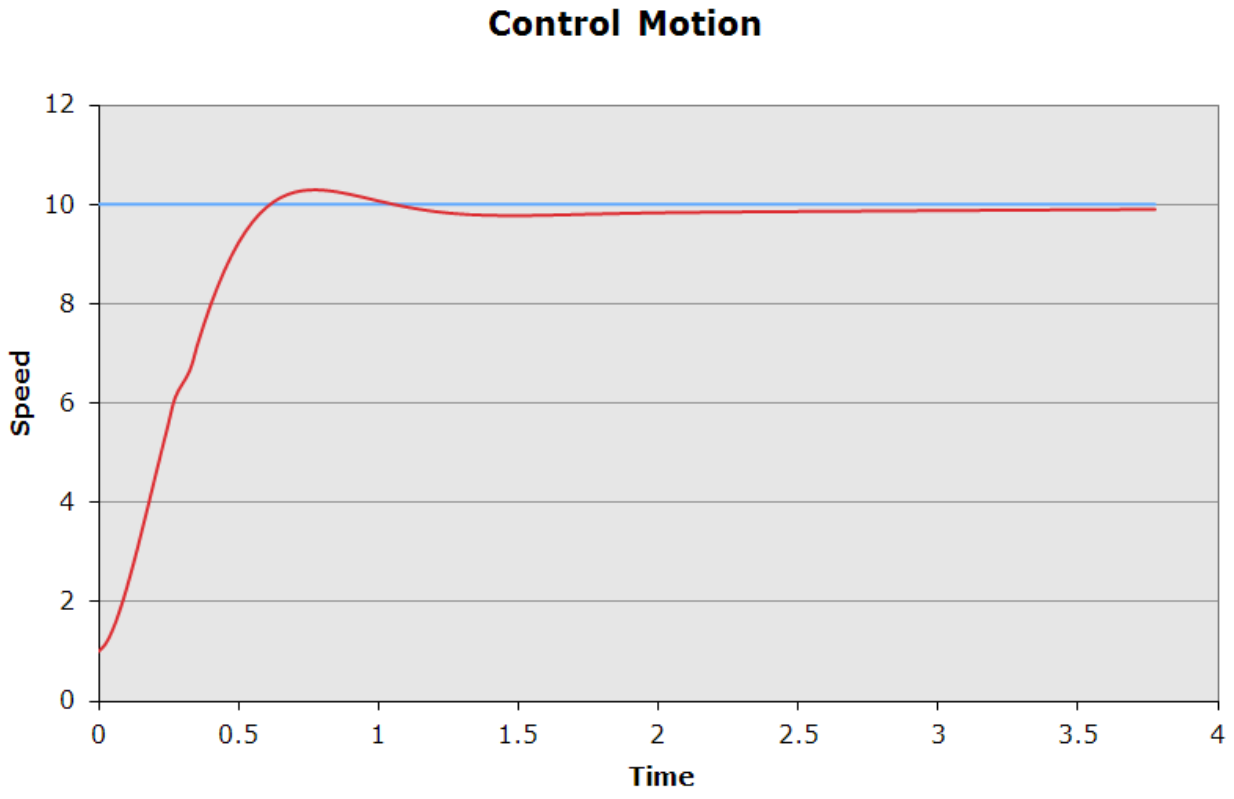


Figure 4.11: PID Response Graph

Processing the Wi-Fi data

The message structure is used for communication over UART is:
Message ID||MSG1||MSG2||MSG3||Data||
* Start of message
* End of message
|| Delimiter (To separate different parts of the message)

The Wi-Fi receiver loop separates the different parts of the message and stores it in an array.
The android application sends the data in form of an angle to the FPGA. This data has to be converted to a PWM value in order to control the car. The maximum and minimum angle values received from the application are as follows:
Maximum angle for Speed: +50 Degrees
Minimum angle for Speed: -50 Degrees
Maximum angle for Turn: +50 Degrees
Minimum angle for Turn: -50 Degrees

To convert the Angle value into PWM or vice versa the following factor is used.

#define *PWM_ANGLE_CONVERSION_FACTOR*

The following algorithm is used to convert the angle values into PWM values.

```
If (Angle > 0)
{
        Set Motor in Forward direction
}
If (Angle < 0)
{
        Angle = -Angle
        Set Motor in Reverse direction
}
pwm = Angle * PWM_ANGLE_CONVERSION_FACTOR
if(pwm > MAX_NOMINAL_PWM)
{
        pwm = MAX_NOMINAL_PWM
}
Set the PWM using the motor_control function.
```

Chapter
5. Smartphone Based Control

5.1 Android

To design a Android Wifi remote application to control the motion of the lego car using Wifi. The motion of the Lego Car is controlled using the inbuilt orientation sensor of an android device. [18]

5.1.1 Introduction

Android, as a system, is a Java-based operating system that runs on the Linux 2.6 kernel. The system is very lightweight and full featured. Figure shows the unmodified Android home screen.

Figure 5.1: The current Android

Android applications are developed using Java and can be ported rather easily to the new platform. Other features of Android include an accelerated 3-D graphics engine (based on hardware support), database support powered by SQLite, and an integrated web browser. If you are familiar with Java programming or are an OOP developer of any sort, you are likely used to programmatic user interface (UI) development—that is, UI placement which is handled directly within the program code. Android, while recognizing and allowing for programmatic UI development, also supports the newer, XML-based UI layout. XML UI layout is a fairly new concept to the average desktop developer.

One of the more exciting and compelling features of Android is that, because of its architecture, third-party applications—including those that are "home grown"—are executed with the same system priority as those that are bundled with the core system. This is a major departure from most systems, which give embedded system apps a greater execution priority than the thread priority available to apps created by third-party developers. Also, each application is executed within its own thread using a very lightweight virtual machine.

Aside from the very generous SDK and the well-formed libraries that are available to us to develop with, the most exciting feature for Android developers is that we now have access to anything the operating system has access to. In other words, if you want to create an application that dials the phone, you have access to the phone's dialer; if you want to create an application that utilizes the phone's internal GPS (if equipped), you have access to it. The potential for developers to create dynamic and intriguing applications is now wide open.

On top of all the features that are available from the Android side of the equation, Google has thrown in some very tantalizing features of its own. Developers of Android applications will be able to tie their applications into existing Google offerings such as Google Maps and the omnipresent Google Search. Suppose you want to write an application that pulls up a Google map of where an incoming call is emanating from, or you want to be able to store common search results with your contacts; the doors of possibility have been flung wide open with Android.

5.1.2 Sensors in Android

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device. For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dew point, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

The Android platform supports three broad categories of sensors:

Motion sensors
These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.

Environmental sensors
These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.

Position sensors
These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

The sensors available on the device can be accessed to acquire raw sensor data by using the Android sensor framework. The sensor framework provides several classes and interfaces that help you perform a wide variety of sensor-related tasks. For example, you can use the sensor framework to do the following:

- Determine which sensors are available on a device.
- Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
- Register and unregister sensor event listeners that monitor sensor changes.

5.1.3 Sensor Framework Android

The Android sensor framework lets you access many types of sensors. Some of these sensors are hardware-based and some are software-based. Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change. Software-based sensors are not physical devices, although they mimic hardware-based sensors. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called virtual sensors or synthetic sensors. The linear acceleration sensor and the gravity sensor are examples of software-based sensors. Table summarizes the sensors that are supported by the Android platform.

Few Android-powered devices have every type of sensor. For example, most handset devices and tablets have an accelerometer and a magnetometer, but fewer devices have barometers or thermometers. Also, a device can have more than one sensor of a given type. For example, a device can have two gravity sensors, each one having a different range.

Table 5.1: Sensor types supported by the Android platform.

Sensor	Type	Description	Common Uses
TYPE_ACCELEROMETER	Hardware	Measures the acceleration force in m/s^2that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.	Motion detection (shake, tilt, etc.).
TYPE_AMBIENT_TEMPERATURE	Hardware	Measures the ambient room temperature in degrees Celsius (°C). See note below.	Monitoring air temperatures.
TYPE_GRAVITY	Software or Hardware	Measures the force of gravity in m/s^2that is applied to a device on all three physical axes (x, y, z).	Motion detection (shake, tilt, etc.).
TYPE_GYROSCOP	Hardware	Measures a device's rate of rotation in rad/s around each of the three	Rotation detection (spin,

E		physical axes (x, y, and z).	turn, etc.).
TYPE_LIGHT	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
TYPE_LINEAR_ACCELERATION	Software or Hardware	Measures the acceleration force in m/s^2that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.	Monitoring acceleration along a single axis.
TYPE_MAGNETIC_FIELD	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT.	Creating a compass.
TYPE_ORIENTATION	Software	Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with thegetRotationMatrix() method.	Determining device position.
TYPE_PRESSURE	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
TYPE_PROXIMITY	Hardware	Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	Phone position during a call.
TYPE_RELATIVE_HUMIDITY	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative

			humidity.
TYPE_ROTATION_VECTOR	Software or Hardware	Measures the orientation of a device by providing the three elements of the device's rotation vector.	Motion detection and rotation detection.
TYPE_TEMPERATURE	Hardware	Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with theTYPE_AMBIENT_TEMPERATURE sensor in API Level 14	Monitoring temperatures.

Sensor Framework
You can access these sensors and acquire raw sensor data by using the Android sensor framework. The sensor framework is part of the android.hardware package and includes the following classes and interfaces:

SensorManager
You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

Sensor
You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

SensorEvent
The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

SensorEventListener
You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

In a typical application you use these sensor-related APIs to perform two basic tasks:
Identifying sensors and sensor capabilities

Identifying sensors and sensor capabilities at runtime is useful if your application has features that rely on specific sensor types or capabilities. For example, you may want to identify all of the sensors that are present on a device and disable any application features that rely on sensors that are not present. Likewise, you may want to identify all of the sensors of a given type so you can choose the sensor implementation that has the optimum performance for your application.

Monitor sensor events
Monitoring sensor events is how you acquire raw sensor data. A sensor event occurs every time a sensor detects a change in the parameters it is measuring. A sensor event provides you with four pieces of information: the name of the sensor that triggered the event, the timestamp for the event, the accuracy of the event, and the raw sensor data that triggered the event.

Sensor Availability
While sensor availability varies from device to device, it can also vary between Android versions. This is because the Android sensors have been introduced over the course of several platform releases. For example, many sensors were introduced in Android 1.5 (API Level 3), but some were not implemented and were not available for use until Android 2.3 (API Level 9). Likewise, several sensors were introduced in Android 2.3 (API Level 9) and Android 4.0 (API Level 14). Two sensors have been deprecated and replaced by newer, better sensors.

Table summarizes the availability of each sensor on a platform-by-platform basis. Only four platforms are listed because those are the platforms that involved sensor changes. Sensors that are listed as deprecated are still available on subsequent platforms (provided the sensor is present on a device), which is in line with Android's forward compatibility policy.

Table 5.2: Sensor availability by platform.

Sensor	Android 4.0 (API Level 14)	Android 2.3 (API Level 9)	Android 2.2 (API Level 8)	Android 1.5 (API Level 3)
TYPE_ACCELEROMETER	Yes	Yes	Yes	Yes
TYPE_AMBIENT_TEMPERATURE	Yes	n/a	n/a	n/a
TYPE_GRAVITY	Yes	Yes	n/a	n/a
TYPE_GYROSCOPE	Yes	Yes	n/a1	n/a1
TYPE_LIGHT	Yes	Yes	Yes	Yes

TYPE_LINEAR_ACCELERATION	Yes	Yes	n/a	n/a
TYPE_MAGNETIC_FIELD	Yes	Yes	Yes	Yes
TYPE_ORIENTATION	Yes2	Yes2	Yes2	Yes
TYPE_PRESSURE	Yes	Yes	n/a1	n/a1
TYPE_PROXIMITY	Yes	Yes	Yes	Yes
TYPE_RELATIVE_HUMIDITY	Yes	n/a	n/a	n/a
TYPE_ROTATION_VECTOR	Yes	Yes	n/a	n/a
TYPE_TEMPERATURE	Yes2	Yes	Yes	Yes

[1] This sensor type was added in Android 1.5 (API Level 3), but it was not available for use until Android 2.3 (API Level 9).

[2] This sensor is available, but it has been deprecated.

Identifying Sensors and Sensor Capabilities

The Android sensor framework provides several methods that make it easy for you to determine at runtime which sensors are on a device. The API also provides methods that let you determine the capabilities of each sensor, such as its maximum range, its resolution, and its power requirements.

To identify the sensors that are on a device you first need to get a reference to the sensor service. To do this, you create an instance of the SensorManager class by calling the getSystemService() method and passing in the SENSOR_SERVICE argument. For example:

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

Next, you can get a listing of every sensor on a device by calling the getSensorList() method and using the TYPE_ALL constant. For example:

```
List<Sensor> deviceSensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
```

If you want to list all of the sensors of a given type, you could use another constant instead of TYPE_ALL such as TYPE_GYROSCOPE, TYPE_LINEAR_ACCELERATION, or TYPE_GRAVITY.

You can also determine whether a specific type of sensor exists on a device by using the getDefaultSensor()method and passing in the type constant for a specific sensor. If a device has more than one sensor of a given type, one of the sensors must be designated as the default sensor. If a default sensor does not exist for a given type of sensor, the method call returns null, which means the device, does not have that type of sensor. For example, the following code checks whether there's a magnetometer on a device:

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null){
  // Success! There's a magnetometer.
  }
else {
  // Failure! No magnetometer.
  }
```

Note: Android does not require device manufacturers to build any particular types of sensors into their Android-powered devices, so devices can have a wide range of sensor configurations.

In addition to listing the sensors that are on a device, you can use the public methods of the Sensor class to determine the capabilities and attributes of individual sensors. This is useful if you want your application to behave differently based on which sensors or sensor capabilities are available on a device. For example, you can use the getResolution() and getMaximumRange() methods to obtain a sensor's resolution and maximum range of measurement. You can also use the getPower() method to obtain a sensor's power requirements.

Two of the public methods are particularly useful if you want to optimize your application for different manufacturer's sensors or different versions of a sensor. For example, if your application needs to monitor user gestures such as tilt and shake, you could create one set of data filtering rules and optimizations for newer devices that have a specific vendor's gravity sensor, and another set of data filtering rules and optimizations for devices that do not have a gravity sensor and have only an accelerometer. The following code sample shows you how you can use the getVendor() and getVersion() methods to do this. In this sample, we're looking for a gravity sensor that lists Google Inc. as the vendor and has a version number of 3. If that particular sensor is not present on the device, we try to use the accelerometer.

```
private SensorManager mSensorManager;
private Sensor mSensor;


...
```

```
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

if (mSensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY) != null){
  List<Sensor> gravSensors = mSensorManager.getSensorList(Sensor.TYPE_GRAVITY);
  for(int i=0; i<gravSensors.size(); i++) {
    if ((gravSensors.get(i).getVendor().contains("Google Inc.")) &&
       (gravSensors.get(i).getVersion() == 3)){
      // Use the version 3 gravity sensor.
      mSensor = gravSensors.get(i);
    }
  }
}
else{
  // Use the accelerometer.
  if (mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER) != null){
    mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
  }
  else{
    // Sorry, there are no accelerometers on your device.
    // You can't play this game.
  }
}
```

Another useful method is the getMinDelay() method, which returns the minimum time interval (in microseconds) a sensor can use to sense data. Any sensor that returns a non-zero value for the getMinDelay()method is a streaming sensor. Streaming sensors sense data at regular intervals and were introduced in Android 2.3 (API Level 9). If a sensor returns zero when you call the getMinDelay() method, it means the sensor is not a streaming sensor because it reports data only when there is a change in the parameters it is sensing.

The getMinDelay() method is useful because it lets you determine the maximum rate at which a sensor can acquire data. If certain features in your application require high data acquisition rates or a streaming sensor, you can use this method to determine whether a sensor meets those requirements and then enable or disable the relevant features in your application accordingly.

Caution: A sensor's maximum data acquisition rate is not necessarily the rate at which the sensor framework delivers sensor data to your application. The sensor framework reports data

through sensor events, and several factors influence the rate at which your application receives sensor events. For more information, see Monitoring Sensor Events.

Monitoring Sensor Events

To monitor raw sensor data you need to implement two callback methods that are exposed through theSensorEventListener interface: onAccuracyChanged() and onSensorChanged(). The Android system calls these methods whenever the following occurs:

A sensor's accuracy changes. In this case the system invokes the onAccuracyChanged() method, providing you with a reference to theSensor object that changed and the new accuracy of the sensor. Accuracy is represented by one of four status constants:
SENSOR_STATUS_ACCURACY_LOW, SENSOR_STATUS_ACCURACY_MEDIUM,SENSOR_STATUS_ACCURACY_HIGH, or SENSOR_STATUS_UNRELIABLE.

A sensor reports a new value.
In this case the system invokes the onSensorChanged() method, providing you with a SensorEvent object. A SensorEvent object contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded.

The following code shows how to use the onSensorChanged() method to monitor data from the light sensor. This example displays the raw sensor data in a TextView that is defined in the main.xml file as sensor_data.

```
public class SensorActivity extends Activity implements SensorEventListener {
  private SensorManager mSensorManager;
  private Sensor mLight;

  @Override
  public final void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
  }

  @Override
  public final void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Do something here if sensor accuracy changes.
```

```
 }

 @Override
 public final void onSensorChanged(SensorEvent event) {
  // The light sensor returns a single value.
  // Many sensors return 3 values, one for each axis.
  float lux = event.values[0];
  // Do something with this sensor value.
 }

 @Override
 protected void onResume() {
  super.onResume();
  mSensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
 }

 @Override
 protected void onPause() {
  super.onPause();
  mSensorManager.unregisterListener(this);
 }
}
```

In this example, the default data delay (SENSOR_DELAY_NORMAL) is specified when the registerListener()method is invoked. The data delay (or sampling rate) controls the interval at which sensor events are sent to your application via the onSensorChanged() callback method. The default data delay is suitable for monitoring typical screen orientation changes and uses a delay of 200,000 microseconds. You can specify other data delays, such as SENSOR_DELAY_GAME (20,000 microsecond delay), SENSOR_DELAY_UI (60,000 microsecond delay), or SENSOR_DELAY_FASTEST (0 microsecond delay). As of Android 3.0 (API Level 11) you can also specify the delay as an absolute value (in microseconds).

The delay that you specify is only a suggested delay. The Android system and other applications can alter this delay. As a best practice, you should specify the largest delay that you can because the system typically uses a smaller delay than the one you specify (that is, you should choose the slowest sampling rate that still meets the needs of your application). Using a larger delay imposes a lower load on the processor and therefore uses less power.

There is no public method for determining the rate at which the sensor framework is sending sensor events to your application; however, you can use the timestamps that are associated with each sensor event to calculate the sampling rate over several events. You should not have to change the sampling rate (delay) once you set it. If for some reason you do need to change the delay, you will have to unregister and reregister the sensor listener.

It's also important to note that this example uses the onResume() and onPause() callback methods to register and unregister the sensor event listener. As a best practice you should always disable sensors you don't need, especially when your activity is paused. Failing to do so can drain the battery in just a few hours because some sensors have substantial power requirements and can use up battery power quickly. The system will not disable sensors automatically when the screen turns off.

Handling Different Sensor Configurations

Android does not specify a standard sensor configuration for devices, which means device manufacturers, can incorporate any sensor configuration that they want into their Android-powered devices. As a result, devices can include a variety of sensors in a wide range of configurations. For example, the Motorola Xoom has a pressure sensor, but the Samsung Nexus S does not. Likewise, the Xoom and Nexus S have gyroscopes, but the HTC Nexus One does not. If your application relies on a specific type of sensor, you have to ensure that the sensor is present on a device so your app can run successfully. You have two options for ensuring that a given sensor is present on a device:

- Detect sensors at runtime and enable or disable application features as appropriate.
- Use Google Play filters to target devices with specific sensor configurations.

Each option is discussed in the following sections.

Detecting sensors at runtime

If your application uses a specific type of sensor, but doesn't rely on it, you can use the sensor framework to detect the sensor at runtime and then disable or enable application features as appropriate. For example, a navigation application might use the temperature sensor, pressure sensor, GPS sensor, and geomagnetic field sensor to display the temperature, barometric pressure, location, and compass bearing. If a device doesn't have a pressure sensor, you can use the sensor framework to detect the absence of the pressure sensor at runtime and then disable the portion of your application's UI that displays pressure. For example, the following code checks whether there's a pressure sensor on a device:

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (mSensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE) != null){
// Success! There's a pressure sensor.
}
else {
```

```
// Failure! No pressure sensor.
}
```

5.1.4 Position Sensor Android[18]

The Android platform provides two sensors that let you determine the position of a device: the geomagnetic field sensor and the orientation sensor. The Android platform also provides a sensor that lets you determine how close the face of a device is to an object (known as the proximity sensor). The geomagnetic field sensor and the proximity sensor are hardware-based. Most handset and tablet manufacturers include a geomagnetic field sensor. Likewise, handset manufacturers usually include a proximity sensor to determine when a handset is being held close to a user's face (for example, during a phone call). The orientation sensor is software-based and derives its data from the accelerometer and the geomagnetic field sensor.

Position sensors are useful for determining a device's physical position in the world's frame of reference. For example, you can use the geomagnetic field sensor in combination with the accelerometer to determine a device's position relative to the magnetic North Pole. You can also use the orientation sensor (or similar sensor-based orientation methods) to determine a device's position in your application's frame of reference. Position sensors are not typically used to monitor device movement or motion, such as shake, tilt, or thrust (for more information, see Motion Sensors).

The geomagnetic field sensor and orientation sensor return multi-dimensional arrays of sensor values for each SensorEvent. For example, the orientation sensor provides geomagnetic field strength values for each of the three coordinate axes during a single sensor event. Likewise, the orientation sensor provides azimuth (yaw), pitch, and roll values during a single sensor event. For more information about the coordinate systems that are used by sensors, see Sensor Coordinate Systems. The proximity sensor provides a single value for each sensor event. Table summarizes the position sensors that are supported on the Android platform.

Definition of the coordinate system used by the Sensor Event API.

The coordinate-system is defined relative to the screen of the phone in its default orientation. The axes are not swapped when the device's screen orientation changes. The X axis is horizontal and points to the right, the Y axis is vertical and points up and the Z axis points towards the outside of the front face of the screen. In this system, coordinates behind the screen have negative Z values.

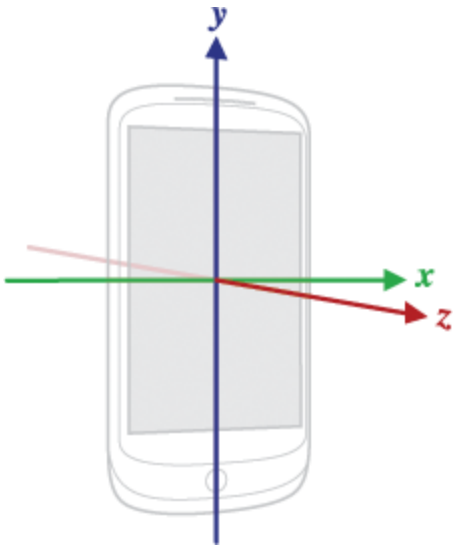


Figure 5.2: Android Coordinate System

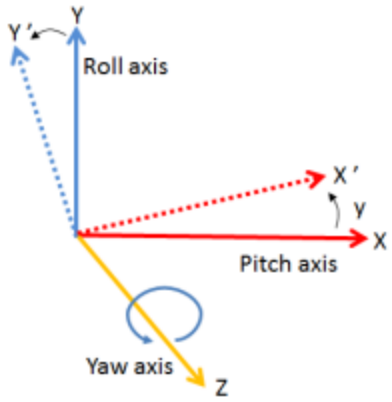
Using the Orientation Sensor

The orientation sensor lets you monitor the position of a device relative to the earth's frame of reference (specifically, magnetic north). The following code shows you how to get an instance of the default orientation sensor:

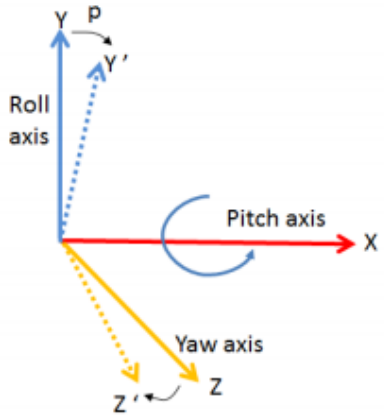
```
private SensorManager mSensorManager;

private Sensor mSensor;

...

mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
```

The orientation sensor derives its data by using a device's geomagnetic field sensor in combination with a device's accelerometer. Using these two hardware sensors, an orientation sensor provides data for the following three dimensions:

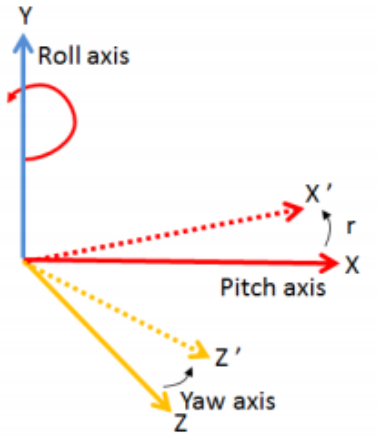
Azimuth (degrees of rotation around the z axis). This is the angle between magnetic north and the device's y axis. For example, if the device's y axis is aligned with magnetic north this value is 0, and if the device's y axis is pointing south this value is 180. Likewise, when the y axis is pointing east this value is 90 and when it is pointing west this value is 270.

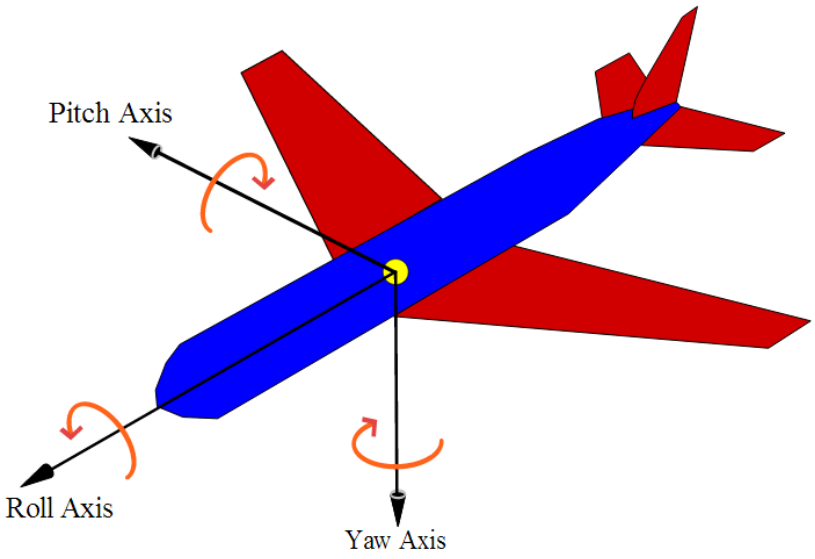


Pitch (degrees of rotation around the x axis). This value is positive when the positive z axis rotates toward the positive y axis, and it is negative when the positive z axis rotates toward the negative y axis. The range of values is 180 degrees to -180 degrees.



Roll (degrees of rotation around the y axis). This value is positive when the positive z axis rotates toward the positive x axis, and it is negative when the positive z axis rotates toward the negative x axis. The range of values is 90 degrees to -90 degrees.





Note: This definition is different from yaw, pitch, and roll used in aviation, where the X axis is along the long side of the plane (tail to nose). Also, for historical reasons the roll angle is positive in the clockwise direction (mathematically speaking, it should be positive in the counter-clockwise direction).

The orientation sensor derives its data by processing the raw sensor data from the accelerometer and the geomagnetic field sensor. Because of the heavy processing that is involved, the accuracy and precision of the orientation sensor is diminished (specifically, this sensor is only reliable when the roll component is 0). As a result, the orientation sensor was deprecated in Android 2.2 (API level 8). Instead of using raw data from the orientation sensor, we recommend that you use the getRotationMatrix() method in conjunction with the getOrientation() method to compute orientation values. You can also use the remapCoordinateSystem() method to translate the orientation values to your application's frame of reference.

You do not usually need to perform any data processing or filtering of the raw data that you obtain from an orientation sensor, other than translating the sensor's coordinate system to your application's frame of reference. The Accelerometer Play sample shows you how to translate acceleration sensor data into another frame of reference; the technique is similar to the one you might use with the orientation sensor.

5.3 WiFi in Android

The **android.net.wifi** package provides classes to manage Wi-Fi functionality on the device. The Wi-Fi APIs provide a means by which applications can communicate with the lower-level wireless stack that provides Wi-Fi network access. Almost all information from the device supplicant is available, including the connected network's link speed, IP address, negotiation state, and more, plus information about other networks that are available. Some other API features include the ability to scan, add, save, terminate and initiate Wi-Fi connections.

Some APIs may require the following user permissions:

- ACCESS_WIFI_STATE
- CHANGE_WIFI_STATE
- CHANGE_WIFI_MULTICAST_STATE

Note: Not all Android-powered devices provide Wi-Fi functionality. If your application uses Wi-Fi, declare so with a <uses-feature> element in the manifest file:

```
<manifest ...>
    <uses-feature android:name="android.hardware.wifi" />
    ...
</manifest>
```

5.4 Interface with the Wi-Fi Module

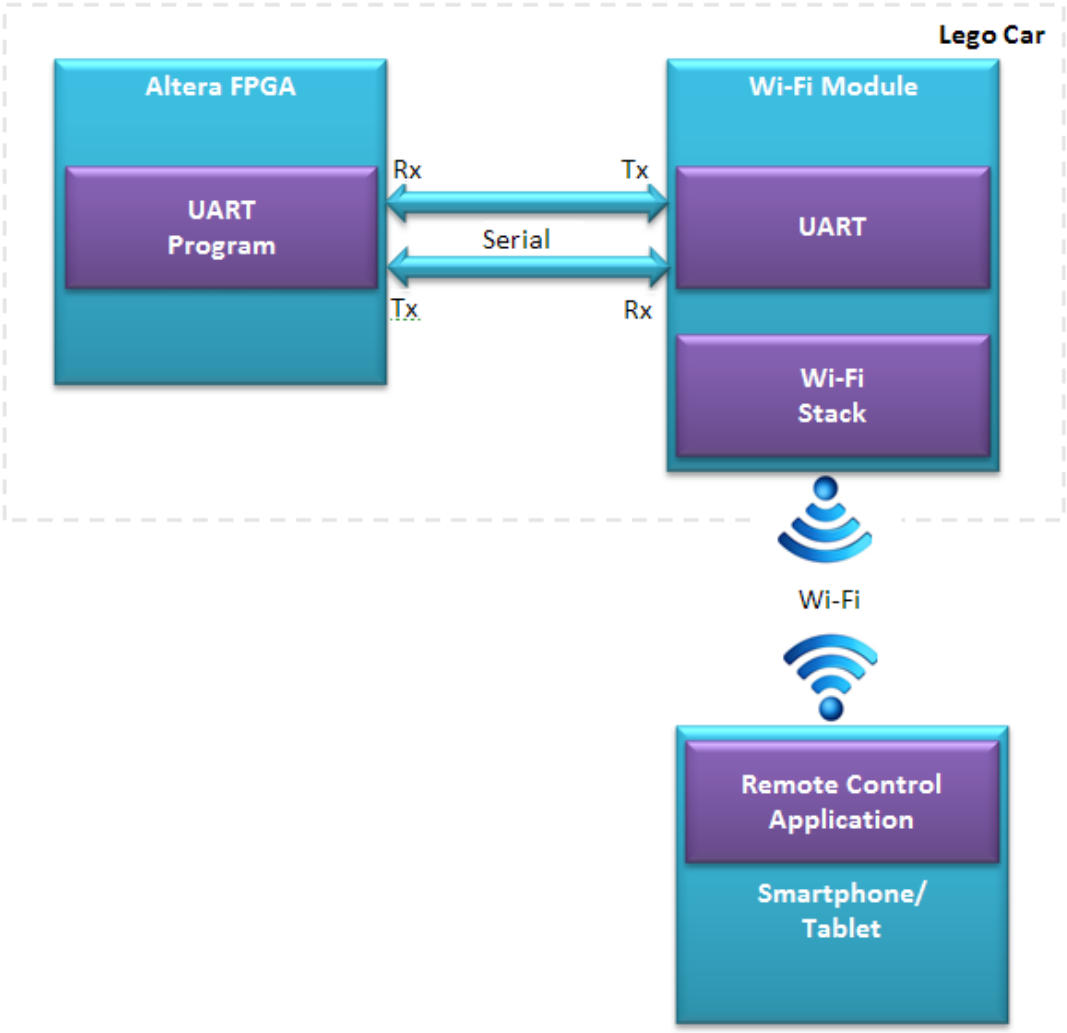


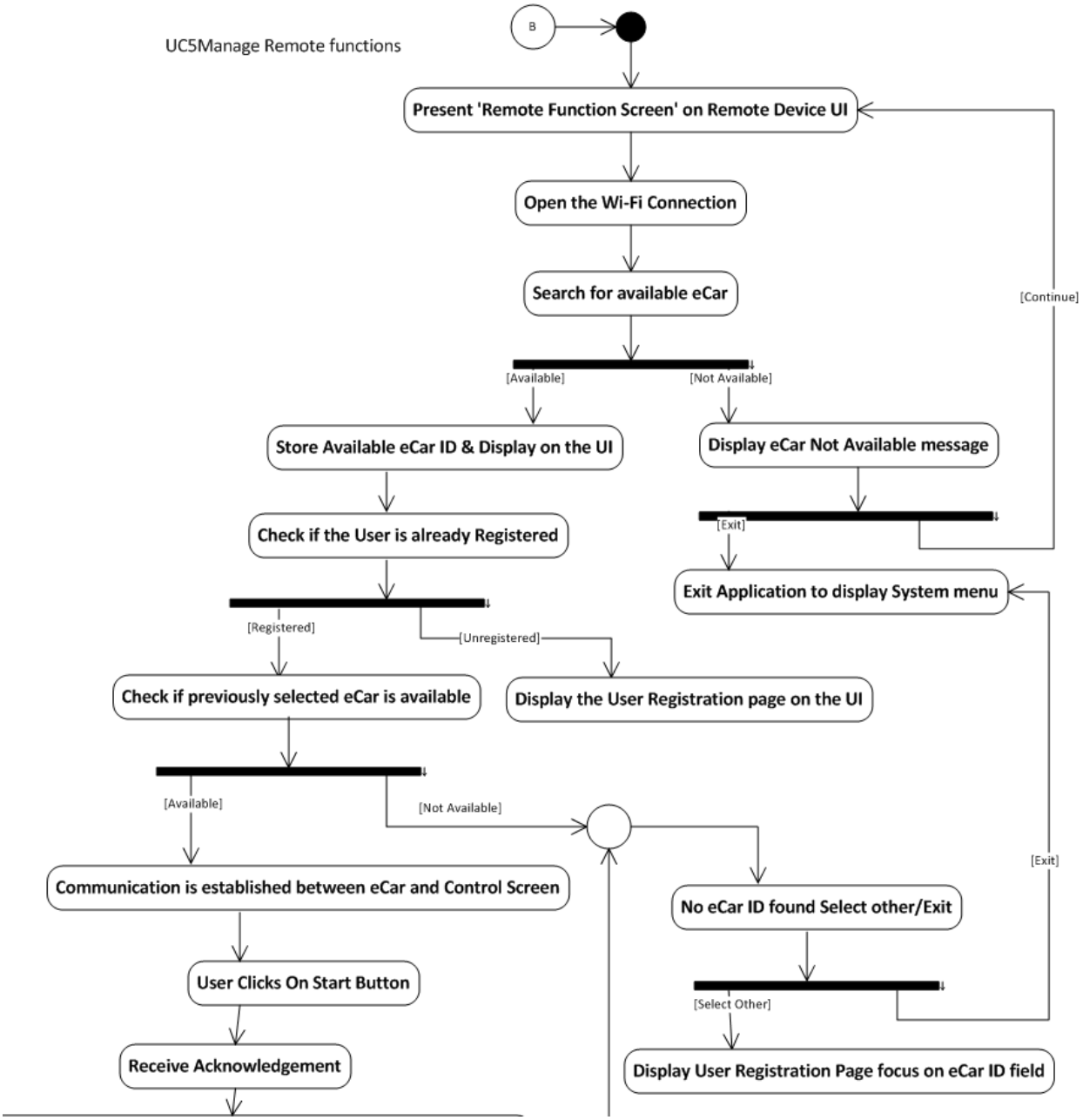
Figure 5.3: Interface

The ecar remote application connects to the Lego car using a wifi link. The RN131 module on the ecar acts as an secure access point with a WPA2-PSK security password. The ecar application only scans for access points with AG as the prefix (ex AG001001).

5.5 Ecar Remote Application

Figure 5.4: Flowchart for Startup and Exit:

Figure 5.5: Flowchart for Managing Remote Functions:



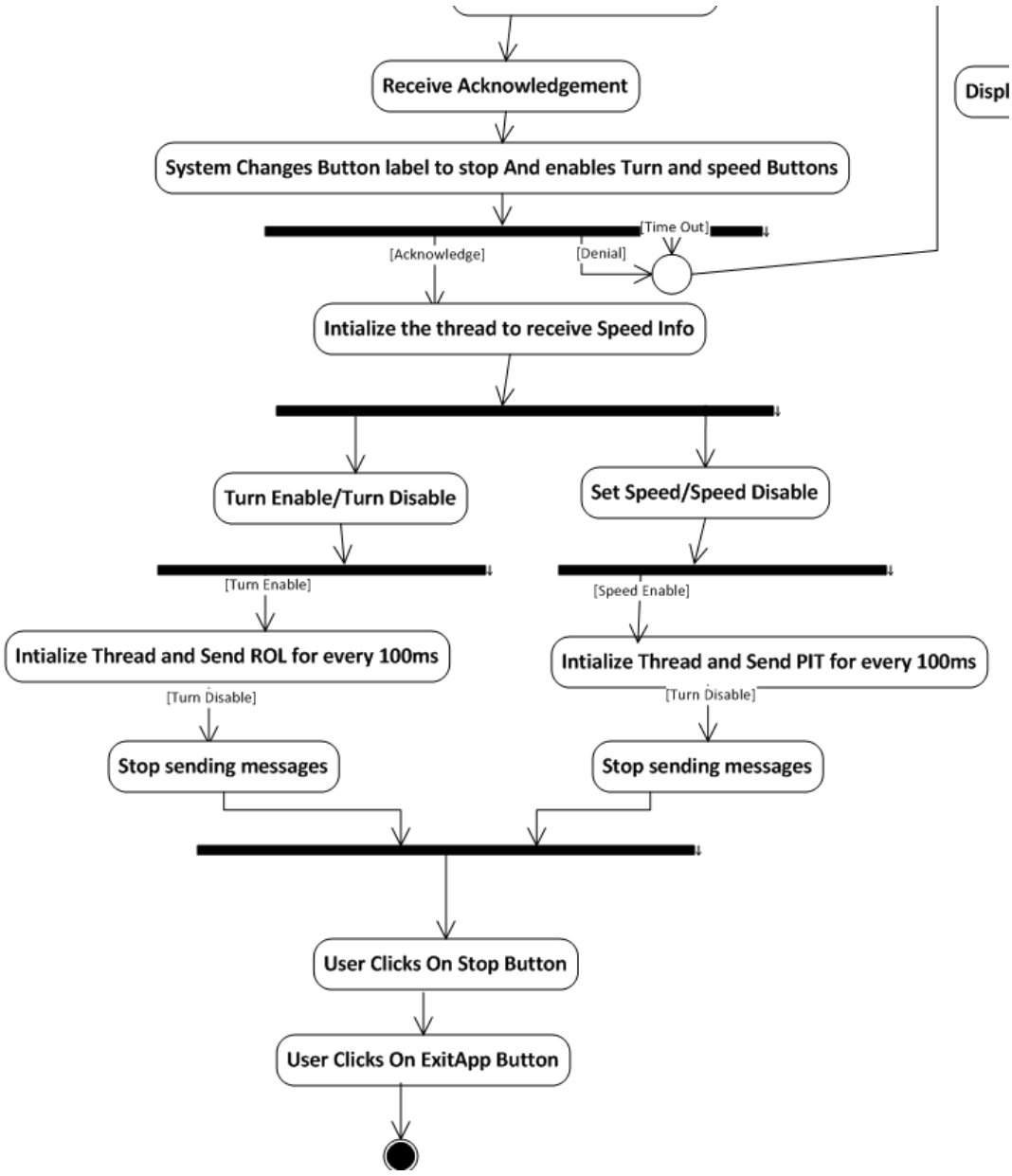
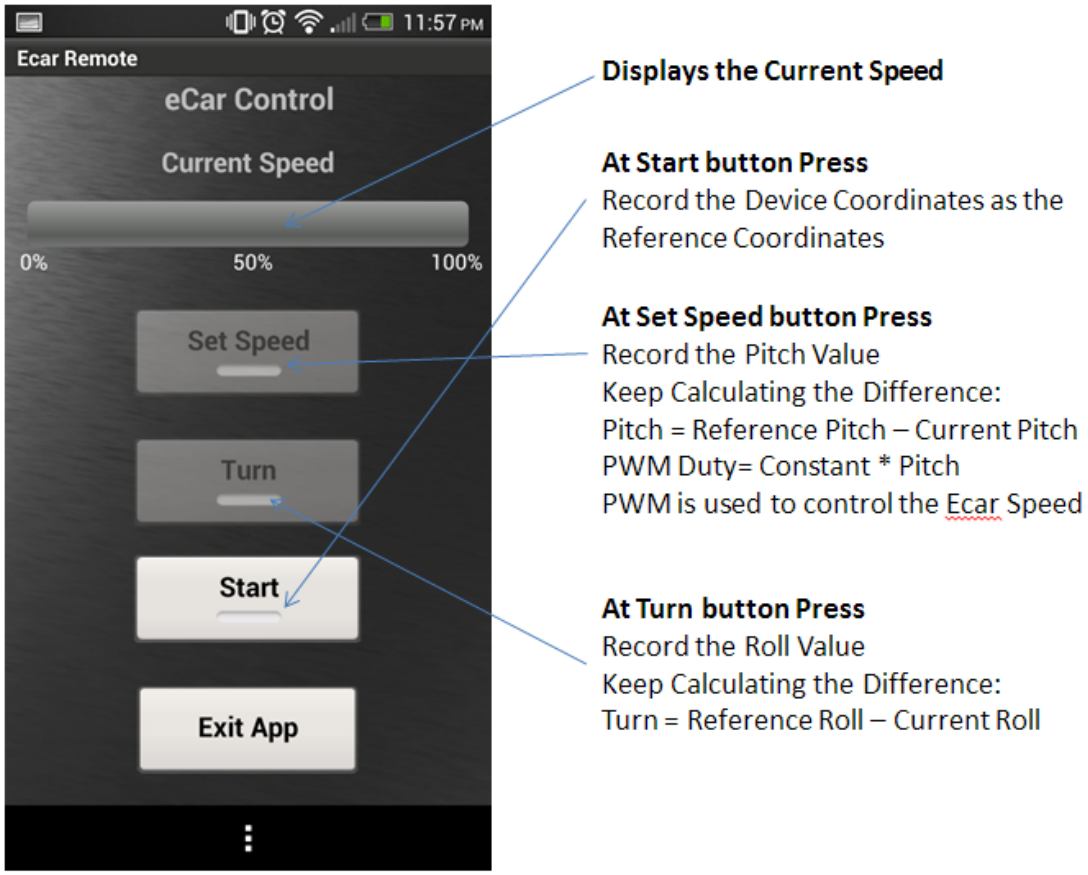

Receive Acknowledgement
Displ
System Changes Button label to stop And enables Turn and speed Buttons
[Time Out]
[Acknowledge]
[Denial]
Intialize the thread to receive Speed Info
Turn Enable/Turn Disable
Set Speed/Speed Disable
[Turn Enable]
[Speed Enable]
Intialize Thread and Send ROL for every 100ms
Intialize Thread and Send PIT for every 100ms
[Turn Disable]
[Turn Disable]
Stop sending messages
Stop sending messages
User Clicks On Stop Button
User Clicks On ExitApp Button

Figure 5.6: Application Screen Shots:




11:52 PM
Ecar Remote
searching device
11:51 PM
Ecar Remote
Language
English
First Name
Karan
Last Name
Savant
Email ID
:aransavant@gmail.com
eCar ID
AG219001
Password
Register
Speed Increase
11:57 PM
Ecar Remote
eCar Control
Current Speed
0%
50%
100%
Set Speed
Turn
Start
Exit App
htc
Left
Right
Speed

Chapter
6. Conclusion and Future Scope

The objective of the thesis was to design and implement and analyze different real-time control of a LEGO car. To be flexible and allow for adjustments in the platform depending on the task to be evaluated, the platform is designed on an FPGA. The reasons for this decision are the low cost of the FPGA, the ability to do rapid prototyping and the possibility to re-configure and re-program the device as often as necessary. The FPGA used in this work comes bundled with memory and I/O pins and provides the possibility to attach daughter cards.

After setting up the foundation, a first benchmark was implemented. The control task to be was to control the speed of the DC motor. To implement such a control, hardware architecture for the control unit was designed and configured in the FPGA. The main components of the hardware are a processor, memory, input and output interfaces. To evaluate the state of the system as a feedback, sensors are an integral part of every control system. Their value is either determined through polling or interrupts. To avoid busy-waiting, interrupts were used to control the LEGO car making the interrupt handling a crucial part of the hardware design.

Two interrupt controllers were examined while designing the hardware architecture: An internal interrupt controller and an external. The external interrupt controller turned out to be the better choice because its response time is much lower. Control systems have actuators to accomplish the control task. PID controller was implemented for closed loop control. Choosing the right parameters of proportional, integral and derivative components for the controller succeeded in controlling the car. An android application was implemented to control the car using inbuilt sensors. The orientation sensor of the android OS was used.

Future Work

Using the new configuration for the LEGO car which It has Four DC and Four Servo Lego Motors the various driving modes can be implemented. In the default mode, steer angles of front axis and rear axis are controlled normally. Whereas the above mode is realized at higher velocities, driving in strongly related slow mode is characterized by a steering of the two different axes being controlled vice versa (positive and negative angles). At this, the proportion of the angle of the rear axis and the angle of the front axis can be predetermined arbitrarily such that the car is driving around a common pivot point. This mode decreases turn radius on the one hand and improves maneuverability at low speed on the other hand. The parallel mode is used when all the wheels should be oriented to the same direction, which allows laterally movement into parking lots or lane change. As soon as all the wheels are turned in a way that they are arranged along a common circle, thus being able to rotate the car around the center of the circle, the rotational mode is activated. The parking brake mode is performed when all the wheels are arranged like a cross to prevent the car from inadvertent rolling because of the lack of stopping brakes. During the emergency brake mode the wheels are pair wisely slightly oriented like a V-shape, similar to a plough, in order to be able to stop in the case of failure of the engine brake

due to the non-existence of any mechanical brakes. As a matter of course, this mode has to be carefully parameterized before execution, otherwise the car could get out of control.[1]

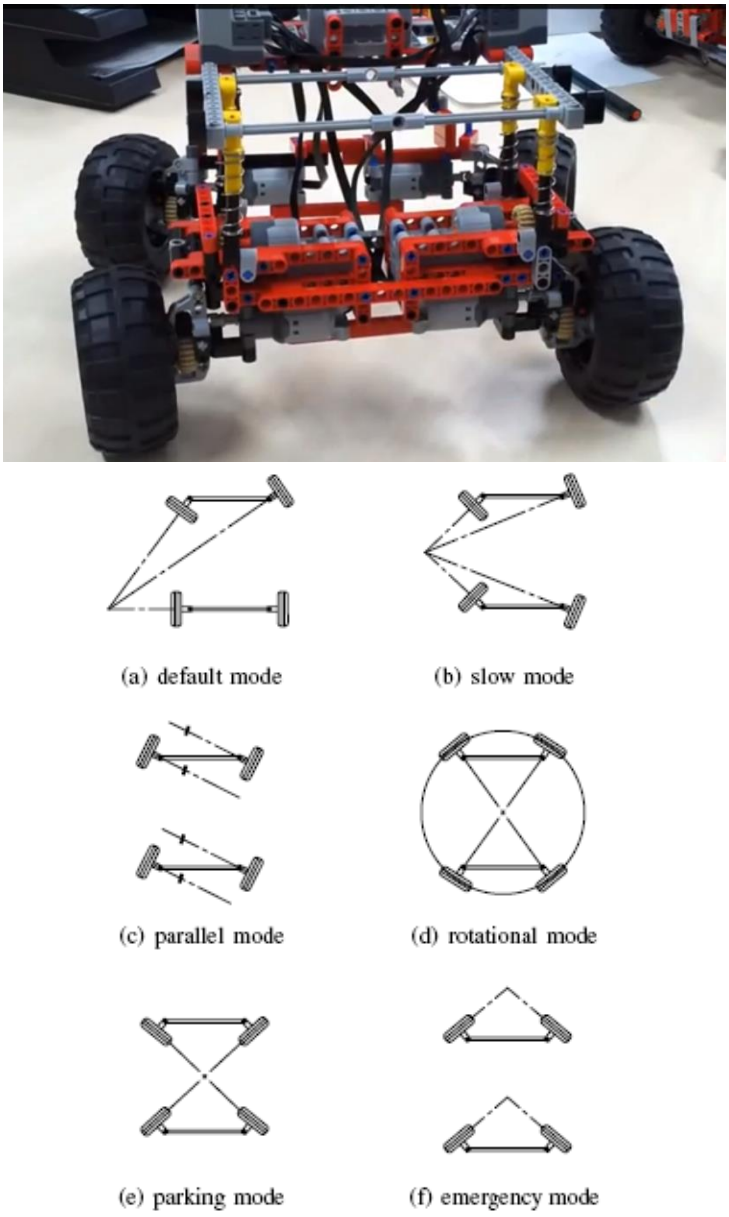


Figure 6.1: New Configuration, Driving Modes

Building upon the idea of adding more resources, the evaluation platform can be further expanded to form a complete multi-core system with two or more processors and a shared memory for communication. A possible architecture is illustrated in figure 6.1. Having a multi-core system as an evaluation platform offers more opportunities for evaluations.

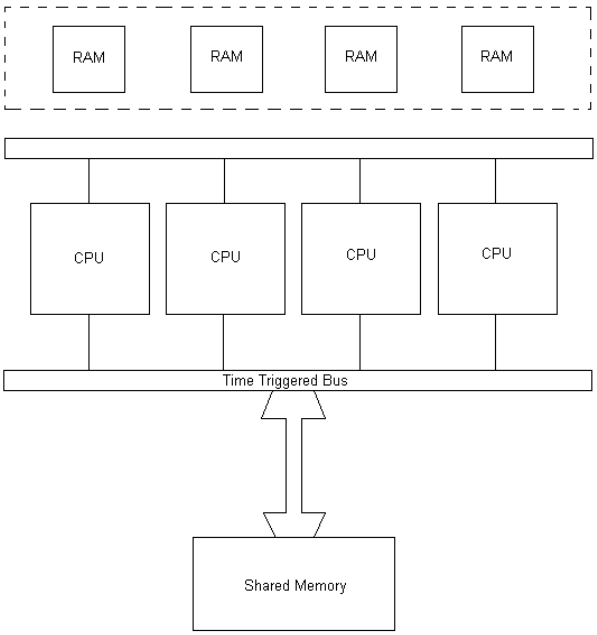


Figure 6.2: Multiprocessor Architecture

Beside real-time control tasks that can run on one core, more complex tasks that can be divided into independent subtasks can also be evaluated on this platform. As an example the camera-based control of the LEGO car is a candidate for such a platform. Using a fast camera, one core can implement various image processing based control functions.

Bibliography

[1] Martin Eder and Alois Knoll “Design of an Experimental Platform for an X-by-wire Car with Four-wheel Steering “ IEEE2010

[2] Back EMF (Electromotive Force) Measurement http://www.acroname.com/robotics/info/articles/back-emf/back-emf.html

[3] Altera Corporation. FPGAs Use in Verification, 2007.

[4] Altera Corporation. Using the DE0-Nano ADC Controller.

[5] Altera Corporation. Cyclone IV Development Kit User Guide

[6] Altera Corporation. AN595: Vectored Interrupt Controller Usage and Application,November 2009.

[7] Altera Corporation. Nios II Software Developer’s Handbook, chapter 9. November 2009.

[8] Altera Corporation. Nios II Software Developer’s Handbook, chapter 8. November 2009.

[9] Altera Corporation. Nios II Software Developer’s Handbook, chapter 5. November 2009.

[10] Altera Corporation. Avalon Interface Specifications, August 2010.

[11] Altera Corporation. Embedded Peripherals IP User Guide, December 2010.

[12] Roving Networks RN134 Wifi Module:http://www.rovingnetworks.com/products/RN134

[13] PID Control: http://en.wikipedia.org/wiki/PID_controller

[14] AUTOSAR. [Online] http://www.autosar.org/.

[15] AUTOSAR**.** *AUTOSAR_EXP_LayeredSoftwareArchitecture.* 2011

[16] ISO26262: http://en.wikipedia.org/wiki/ISO_26262

[17] FPGA Based Automotive ECU Design: http://issuu.com/xcelljournal/docs/xcell_journal_issue_78/20?e=2232228/2668194

[18] Android: http://www.android.com/

[19] Altera SOC Cyclone V: http://www.altera.com/devices/fpga/cyclone-v-fpgas/cyv-index.jsp?ln=devices_processor&l3=SoCs-Cyclone%20V%20(SE,%20SX,%20ST)