Australian National University

College of Engineering and Computer Science

ENGN8537 Embedded Systems And Real Time Digital Signal Processing

PROJECT REPORT

A Self-Balancing Robot with PID/LQR technique and obstacle avoidance function

---

Submitted by:
Aaromal Girish (u6622411)
Akshay Karthik Thana Sekar (u6622423)
Muraleekrishna Gopinathan (u6528602)

Abstract

Two wheeled robots are an example of unstable system similar to the inverted pendulum in a moving cart. The development in control system have paved the way for the control of the robot keeping it in upright position which constitutes for the self-balancing of the robot. This type of robots have gained a lot of attention in the field of research for its robustness with different terrain more than the two legged robots. The control mechanism is by the wheels as the actuator by perceiving the environment using the sensors and the control algorithm used here are Linear Quadratic Regulator (LQR) and the Proportional-Integral-Differential Control (PID). Incorporating complexity to the self balanced model, the robot can also detect and follow a line with a contrast colour gradient from the floor colour. Once a line is detected the control algorithm ensures the minimisation of the deviation of the robot from the line and keeps the robot aligned with the line. Additionally, the robot stops if an obstacle is detected on its path though the line continues. The signal from the ultrasonic sensor is also handled by the control algorithm for stopping the robot in stable upright position as it is when no line is detected to follow. Since, the actuator for all these functionality is the left and the right motor, the control needs to be prioritised where self-balancing is the highest priority followed by line following and obstacle detection. This report summarizes the setup, configuration and the observation of the control functionality of the two wheeled Terasic self-balancing robot.

Contents

List of Tables

List of Figures

1 Introduction

Line Following is one of the most important aspects of robotics. A Line Following Robot is an autonomous robot which can follow either a black or white line that is drawn on the surface consisting of a contrasting color. It is designed to move automatically and follow a line. In this report, we explain the design and implementation of a self-balanced robot from scratch. The robot uses several sensors to identify the line thus assisting the robot to stay on the track. The integration of various sensors makes its movement precise and robust. The robot is driven by DC gear motors to control the movement of the wheels. The control algorithm is implemented on FPGA in DE10-Nano board using Verilog. The software of the project is embedded on a FPGA realized processor - Nios II. The fabric containing the processor and driver components are connected to external sensors using onboard GPIO Pins. The main objectives identified for the self-balancing Robot with PID (Proportional Integral Derivative) or LQR (Linear Quadratic Regulator) control technique to keep the provided two wheeled robots upright under a certain level of disturbance and obstacle avoidance using the control algorithm with the help of the Ultrasonic Sensor. The derived objective is identified as to follow the fixed line while retaining the self-balance. The goal of the robot, in all, is as follows

1. Maintain balance (stay upright)
2. Move along a pre-determined path
3. Stop at detection of an obstacle

This report explains the conceptualisation, system identification, modelling, control system design, and integration of the several functions of the robot to attain this goal. The Section 2 identifies the existing solutions to the problem and how each solutions differ from each other. In Section 3, we detail the design architecture of the robot and explain the function of each process. The mathematical model to design the control

system is presented in the Section 4 to elucidate the underlying problem. We then proceed with the control system design of the robot to attain each function in the Section 5 and 6. Section 7 details the integration of software and hardware and explains how these functions are implemented on the hardware. Finally, we conclude with notes on how new design is better and possible future additions to the system. The complete code for the project is made available in Github [1].

2 Literature Review

Intensive research is under progress in two wheeled robots stabilization and usage because of their advantages over the two legged robots. self balancing is an important challenge for a two wheeled robot and it is a multidisciplinary field of research. Many research institutes around the world are working with the automatic control theory and modern electronic sensing technology to address this problem. In 2001, a innovation was born as "Segway" which is a faster and a convenient way of short distance travel [2].

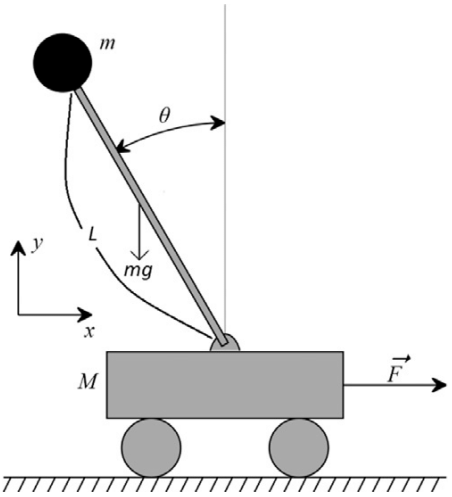


Figure 1: Model of inverted pendulum on a moving cart

Two wheeled self balancing robots are a machines built on a simple physical system - an inverted pendulum (IP) hinged on a cart. An inverted pendulum. is inherently unstable [3]. The non-holonomic cart is moved to keep the pendulum in a dynamic equilibrium. The design and implementation of controller for self balancing has wide variety of applications. From reusable rockets [4] to Segways [5], this principle is being used.

"JOE" was a two-wheeled self balancing robot developed by Swiss Federal Institute of Technology which can handle a maximum speed of 1.5 m/s that is more than the walking speed of people. The DC motor drive

with control algorithm and parallel wheels, "JOE" can make turning to its sides and also to maintain it's self- balance. Adaptive fuzzy control is what they believe to provide interference stability and smooth stability control of "JOE" [6].

In a paper, K. Kankhunthod et al. [7] presented a two-wheeled self-balancing robot by controlling the angular speed of the motor using the gyroscope readings from a sensor. They have implemented on the Raspberry Pi using cascaded stage-two section form two Infinite Impulse Response (IIR) filters and used the control using Fractional-Order Proportional Integral and Derivative (FOPID) controller which is more advantageous and efficient when compared to the conventional PID controllers. MPU-9150 Gyroscope and accelerometer is used to measure the angular position and angular acceleration. FOPID controller stabilizes and increases the transient response with low overshoot and rise time percentage theoretically and in real time FOPID is slower in computation as it is connected to microcontroller or raspberry PI. Kalman filter is used to eliminate noise and data is retrieved by comparing the error covariance between the current and previous state of the system. They have simulated the control system in MATLAB and Simulink using 'FOMCON' toolbox.

In a thesis by H. Hellman, H. Sunnerman [8], they have controlled the robot using Linear Quadratic Regulators (LDR) and used PID controller for validation of results and concluded that the model is not reliable. This model is not reliable as they always had a large error thus large settling time for the impulse response between the real-time implementation and the simulation. The robot can move only forward or backward (one-degree freedom), robot acts a pendulum attached to a cart, linearization for angle deviation is near to upright position. Kalman filter is used as the state estimator in order to the error in angular speed and angular acceleration.

In a thesis by R. C. Ooi [9], the two-wheeled autonomous robot with two control strategies in order to make the robot stand upright provided promising results. The control strategies are Linear Quadratic Regula-

tors (LQR) and pole placement controller. The gain matrices for LQR are obtained from the simulation. Pole are placed arbitrarily which may cause instability of the system. A PID controller tuned with Zeigler-Nichols method is used to control the trajectory of the path on which the robot is traversing. An indirect Kalman filter in fusion with piezo rate gyroscope sensor and a device known as inclinometer is used in order to estimate the tilt angle and derivative of the tilt angle accurately

In a paper [10], the author considered that the self-balancing robot has 3 degree of freedom i.e. x, y, theta. LQR is implemented in order to make the robot to hold upright and only adjusting the motor speed. The regulator gives us the theta value which gets the value from system and the state is estimated by Kalman filter. Current angular position is given by the IMU (Inertial Measurement Unit) and that is used to estimate the variables of the LQR.

In the seminar report [11], the authors have provided the way of embedded control using FPGA. The report deals with the serial and parallel implementation of PID on FPGA. They also glanced at the way of implementation of PID using Distributed Arithmetic (DA). They have also dealt with the recent trends in FPGA and implementation of controllers

3 Design Architecture

The design architecture integrates the sensing modules, the actuators and the control algorithm working under a task prioritizing controller. The interconnections for data transfer with addressing is made with Avalon bus. Avalon MM interconnects are special interconnects that can be used to connect between different components in Qsys. It includes read, readdata, write, writedata, clk, reset and address signals to access slaves and its registers. In this project we plan to isolate the existing Nios based balance car and create a new Vehicle control master in Verilog. For this, a single Avalon bus will interconnect a master with multiple slaves. The software architecture is shown in Figure 1. The vehicle controller, the master, addresses each component based on their base addresses. The MPU 6050 module is interfaced using simpler I2C wishbone master slave protocol. To realise the masterslave addressing, an interconnect/address translator is used. In Qsys, this component is automatically generated when a connection between a master and slave is created. For signals other than the Qsys supported protocols, Conduit interconnects can be used. This interconnect is an aggregate of arbitrary signals which can be used to connect to external components or non-standard internal components. The self-balancing robot system comprises of the sensing unit, Controller unit and the actuator unit. The sensing unit has the Motor encoders, IR sensor array, IMU and the ultrasonic sensor. The DC motor is the integral part of the actuator unit and the controller unit performs the PID, LQR and the priority based task control. These can be seen from the functional block diagram as shown below.

3.1 Functional block diagram

The heart of the architecture is a priority based task controller which applies appropriate control action based on state of the robot. Robot is balanced by applying right PWM to the motors using LQR or PID. If

robot is balanced, it checks if any obstacle is detected in-front, otherwise it follows the line. The current angular position of the robot relative to its vertical axis is provided by the IMU (Gyroscope and Accelerometer) inbuilt in Motor control board and this value are used to estimate the state variables for LQR control. The Proportional Integral Derivative (PID) controller is planned to implement the line following capability to the robot. The Pololu sensor array is placed over the line with a different floor colour and based on the position of the IR detected, the corresponding direction of the bot is determined. PID is the closed loop feedback theory which is used to calculate and correct this error maintaining the position of the bot along the line [12]. The interconnects between the blocks carry the control signals and sensor outputs. The IR sensor array outputs the line deviation signal which is used in error calculations and the PID controller generates a control signal - direction and it is given to the task controller. The IMU triggers the task controller if it is flat with the surface representing the upright position and if there is an angle tilt, the pitch and angular velocity signal flows to the balance controller(LQR) which generates a control signal- Angle and is fed to the task controller. Similarly the functional blocks, motor encoder sends a distance signal to the balance Controller(LQR) for calculating the angle control signal. The Ultrasonic sensor directly sends the stop signal to priority based task controller. The priority based task controller maintains the self balance by providing the PWM signal to the motors and once the robot is in balance and not hindered by an obstacle, the line following task is prioritized to control the motor to move on the line in upright position and continuously sensing any obstacle on its path.

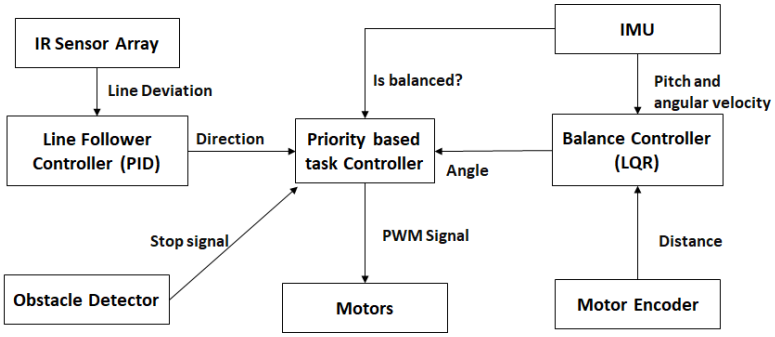


Figure 2: Functional Block Diagram

3.2 Additional structure

Casing for the IR sensor array is 3D printed considering the dimensions of the robot. For the lightness and strength of the structure, Polylactic Acid (PLA) is used. Nylon spacers are used to adjust the IR array spacing to the ground. The modelling is done in the ANSYS workbench 2019 R2. A socket is provided for plugging the wire to the IR sensor module. The length of the holder is 66 mm and the IR case is 75 mm to hold the polulu IR sensor array. The entire structure with the sensor is bolted on the existing battery base of the robot and the whole robot structure is as shown in the appendix Figure 24.

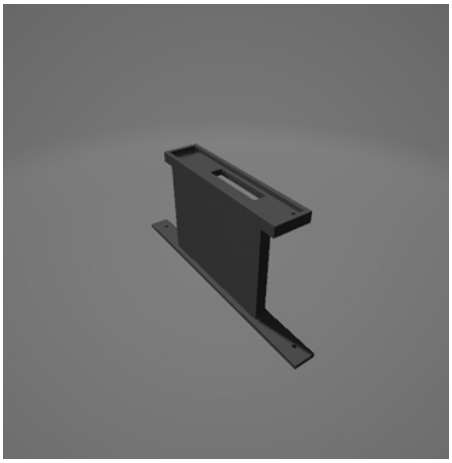


Figure 3: IR Sensor Casing

4 Mathematical modelling

A mathematical model is derived from solving the dynamic equations formed by analyzing the system's internal parameters and their interactions with the external environment. For the given application we use the State space model as shown below with the parameters measured by inspecting the robot. These values are fed to the state space model of the system to get into a model as shown below

4.1 State space form

A two wheeled robot is like an inverted pendulum on a moving cart. By linearising the non-linear model at $x = 0, \theta = 0$, the linear model can be derived as shown,

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta}_p \\ \ddot{\theta}_p \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \dfrac{2k_m k_e \left(M_p l r - I_p - M_P l^2\right)}{Rr^2 A} & \dfrac{M_p^2 g l^2}{A} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \dfrac{2k_m k_e \left(rB - M_p l\right)}{Rr^2 A} & \dfrac{M_p g l \beta}{A} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta_p \\ \dot{\theta}_p \end{bmatrix}$$

$$+ \begin{bmatrix} 0 \\ \dfrac{2k_m \left(I_p + M_p l^2 - M_p l r\right)}{RrA} \\ 0 \\ \dfrac{2k_m \left(M_p l - rB\right)}{RrA} \end{bmatrix} U_a.$$

where the state variables are identified as the θ is the angular displacement, $\dot{\theta}$ is the angular velocity, x is linear displacement and $\dot{x}$ is the linear velocity [13]. The parameters of the robot are identified as

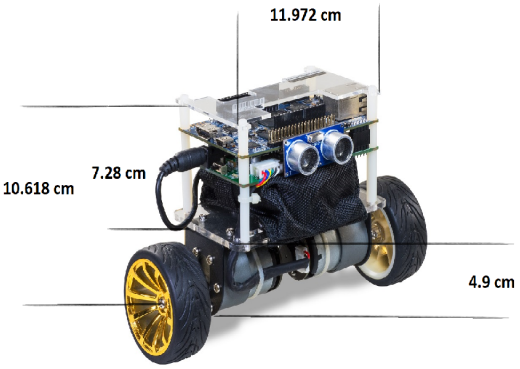
shown in the table below. by substituting the corresponding values on the state space model and simulating the experimental LQR controller gain using MATLAB, an optimized controller gain is identified and is used for the self - balancing of the robot keeping it in the upright position.

Parameter	Value
g	9.81 m/s
r	0.0315 m
M_w	0.0505 kg
M_p	0.750 kg
I_w	0.000085
I_p	0.0028
l	0.062
k_p	0.005814
k_e	0.005498
R	3

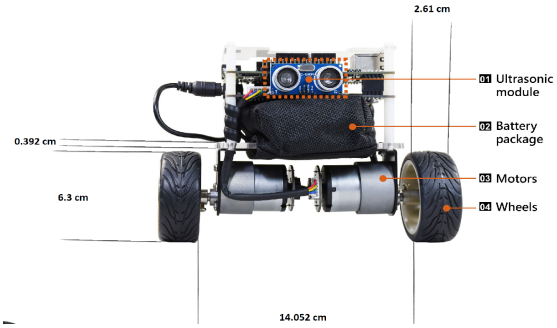
Figure 4: Parameters of the two-wheeled robot

4.2 Robot Model parameters

The two-wheeled self balancing robot by terasic has a standard dimension for its body and the wheels. Yet, for precision, the dimension of the bot was re-calculated and also performed measuring of the parameters of the robot that has an impact on the stability of the robot like the centre of mass, Inertia of the body, Inertia of the battery, weight of the robot etc. These values where used in the system model for simulating the gain of the controller for the self-balancing.



(a) Robot side view



(b) Robot front view

(c) Robot top view

5 Control Theory

5.1 PID control

Real world system seldom behave as designed. The effects of noise, disturbances, and complexity of the system itself will cause the system to drift away from its ideal behaviour. Feedback allows to correct its behaviour in presence of real world perturbations and reach its desired state. The error between actual and desired outputs are related to the controller input. The relation can be proportional, derivative, or integral of the error. The controllers generate control signals to the plant using these relations or its combinations, depending on the plant behaviour. These can be Proportional (P), Proportional-Derivative (PD), Proportional-Integral (PI) and Proportional-Integral-Derivative (PID) controllers used for the purpose. In theory, P control action takes the system to desired state but with some steady state error. Adding integral term will reduce this error but can cause overshoot. The derivative controller will reduce overshoot.

Figure 6: Proportional-Integral-Derivative Controller

The control signal is given by the continuous time equation [14] as in Figure 1,

$$u = K_P e + K_I \int e + K_D \dot{e} \tag{1}$$

where u is the control output, e and $\dot{e}$ the error and its derivative, and K_P, K_I and K_D the controller gains. In s domain,

$$E_1(s) = R(s) - \theta(s) \tag{2}$$

E_1 being the error, R the reference angle and Îÿ the current tilt,

$$E_2(s) = E1(s)K_P + K_I s - K_D(s)\theta(s) \tag{3}$$

Substituting (2) in (3)

$$E2(s) = (R(s) - \theta(s)K_P + K_I s - K_D(s)\theta(s) \tag{4}$$

To implement this in Verilog, the equation needs to be discretised. After discretisation, we obtain [1],

$$u(n) = u(n-1) + K_0 e(n) + K_1 e(n-1) + K_2 e(n-2) \tag{5}$$

where $K_0 = K_P + K_I + K_D, K_1 = -K_P - 2K_D$ and $K_2 = K_D$. This equation avoids integration and differentiation of all past errors.

5.2 LQR control

The self-balancing robot is a non-linear system with three degree of freedom in X direction, Y direction and angle θ. For controlling the angle of deviation from the upright position, LQR is implemented adjusting the motor speed. The equilibrium is attained by Lagrange equation determining the kinetic and potential energies exerted on the bot during its translational and rotational motion [15]. The Linear Quadratic Regulator (LQR) determines the θ control based on the x_e the estimated state from the Kalman filter. The system model is derived as a state space form where, A - system matrix, B - control Matrix and C -output matrix considering D - disturbance which is to be zero. We have linearized the system in the form of $\dot{x} = Ax + Bu$ - next state output to make the non-linear system controllable [14].
For optimising the control of an unstable system, the dynamic system need to be brought back to stability with minimum cost. The cost function is defined by the sum of the deviation of the key measurements of the system like the angular and the positional displacement in the

two-wheeled robot. The LQR algorithm automatically provides a state feedback control to the system keeping it to reduce the deviation of the key parameters. Difficulty in finding the exact value of the weighting factors like Q ad R to support the system of specific key parameters is the challenge involved in this control algorithm.

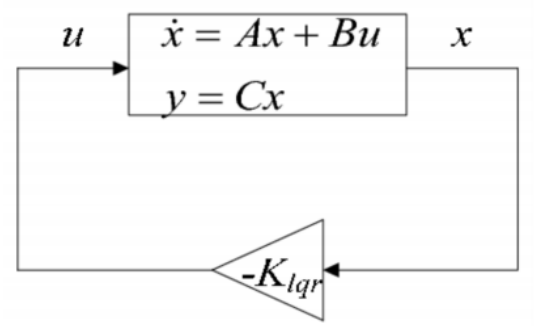


Figure 7: Linear Quadratic Regulator

System stability can be decided based on the number of poles in the negative x-axis. The pole-zero plot in Figure 2 shows that the model has poles in +ve x-axis and thus is unstable [16]. This makes sense as the robot cannot stabilise in all the pitch angles. By applying a controller, we aim to bring all the poles to -ve x-axis and attain $\theta = 0$. We apply LQR controller to the model as to minimise the cost function shown below. MATLAB function lqr can be used to solve for optimal K in $u = -R^{-1}B^{T}Px = -Kx$, solving the The Q and R matrices of the LQR can be used to penalise the cost function [13].

$$J = \int_0^\infty (X^T QX + u^T Ru)dt \tag{6}$$

where Q is proportional to anti-interference ability of the system when increased decreases the response time. This will increase the oscillations and in-turn the energy used by the system. Increasing R will reduce the energy used by the system but increased the response time. The optimal value give the optimal gain K. The selection of Q and R are based on heuristic analysis and experience of the designer. Applying large values

corresponding to x and θ in Q the response time improved and increasing R reduced the energy of the system

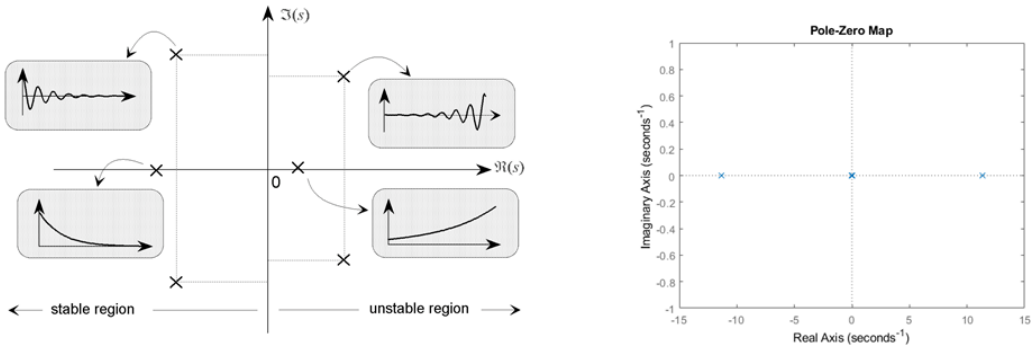


Figure 8: Pole - zero plot Interpretation

The new poles are at -11.45, -11.192729614125167, -0.9297 + 0.9456i and -0.92976 - 0.94565i confirms the model is stable after applying the controller.

5.3 Data Acquisition and Linearization

The LQR and the PID are the control algorithms that are acting upon the same actuator (DC motors). Based on the priority of self-balancing over the line following, we are planning to setup self-balancing as the highest priority interrupt deciding the control over the actuator. The body angle and the wheel positions are adjusted as the response of the control and the result being stabilising at the vertical position [17]. For the line following capability of robot, we have used the PID control with the logic that the bot being at the centre with the line will have a zero error and based on the deviation from the set value of the robot, the error is identified and a proportional, differential and integral control is applied in reducing the error back to zero. For integral control, we are considering the current and previous two errors and the differential control is using the deviation of the error each time. Based on the reference, the continuous time dependent PID controller output can be discretised and can be expressed in velocity from of PID algorithm as below

$$u(n) = u(n-1) + \Delta u(n) = u(n-1) + ko * e(n) + k1 * e(n-1) + k2 * e(n-2) \quad (7)$$

$$e(n) = p + (-pd) \quad (8)$$

$$po = ko * e(n) \quad (9)$$

$$p1 = k1 * e(n-1) \quad (10)$$

$$p2 = k2 * e(n-2) \quad (11)$$

$$s1 = po + p1 \quad (12)$$

$$s2 = p2 + u(n-1) \quad (13)$$

$$u(n) = s1 + s2 \quad (14)$$

The above equations can be implemented both in parallel and serial designs.[14] The obstacle detection is performed using the ultrasonic sensor module and it has the next priority in control over the same actuator (Motor) after LQR and the calculation of the distance of the obstacle is determined by the distance equation which is sound speed X time from echo out to echo back in which is the counter value. For this reporting week, we have a basic implementation of ultrasound sensor and distance perception. HC-SR04 can measure up to range from 2 cm - 400 cm. [18] Hence, the boundary for our robot is also less than 400 cm.

We have also done various research on addressing in Verilog for connecting one Master to multiple slaves. We have done a research on Avalon interface and retrieval of readings of MPU6500 using i2c protocol. But we are still researching and studying on the Avalon interface, the ways of MPU6500 and the ways of accessing HPS.[19] The various detailed documents are provided by the course convenor so that could help us in understanding the working of the above-mentioned concepts

In this project the hardware is already provided to us. As stated above, the objective is to integrate the self balancing function with the

autonomous line following function. The FPGA will implement an LQR controller based on the state given by Kalman filter. The state is based on the angular speed of the robot from Accelerometer ($\dot{\theta}$) and amount of tilt from Gyroscope (θ) readings. Output of the controller is the required angular speed of the motor to correct the tilt. The error between required angle (90deg) and the actual tilt (θ) is the error to be corrected. The line detection will also be implemented on the FPGA using PID controller. The IR reïňĆuctance sensor (Polulu QTR 8RC [20]) is used to identify if the robot is following the line. The direction is corrected based on the direction of error. The obstacle avoidance function will use HC-SR04 Ultrasonic sensor to detect the object in front of the robot while following the line. This distance value is monitored by the FPGA and robot is manoeuvred based on a set obstacle avoidance algorithm.

5.3.1 MPU - 6500 Six - Axis (Gyro + Accelerometer) MEMS

The MPU-6500 is a Micro-Electro Mechanical system that combines the 3 axis accelerometer and a 3 axis gyroscope. It has a dedicated I2C sensor bus for communicating the Logic unit and the sensing unit. The gyroscope is working under the force of gravity as shown in the figure below and has a digitally programmable low-pass filter with 16 bit ADC. FIFO offers 4096-bit serial interface which reduces traffic and reduce consumption. I2C sensor dedicated MPU-6500 can accept input from other external I2C devices. Gyroscope sensor has an adjustable range of ± 250, ± 500, ± 1000 to $\pm 2000^{\circ} sec$ and low noise to 0.01 dps $\sqrt{2} Hz$. The accelerometer is programmable and has a range of $\pm 2g$, $\pm 4g$, $\pm 8g$ and $\pm 16g$. Both sensors were calibrated at the factory. [21]

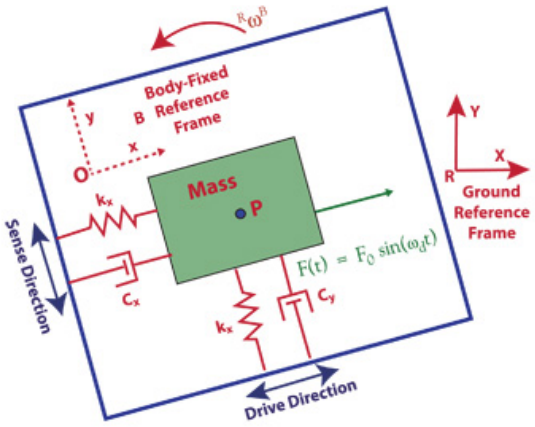


Figure 9: Three axis Gyroscope

5.3.2 Motor Encoder Sensor

An rotary encoder is an electro-mechanical device that produces an electrical signal for the mechanical motion based on the Hall effect sensor. There are two hall effect sensor in the terasic two-wheeled robot. The left and the right motors are controlled by the motor encoders connecting to the GPIO of the DE10 nano via the motor driver board. The position of the robot is tracked by these sensors and used for generating the control signal in stabilising the robot with closed loop feedback control.

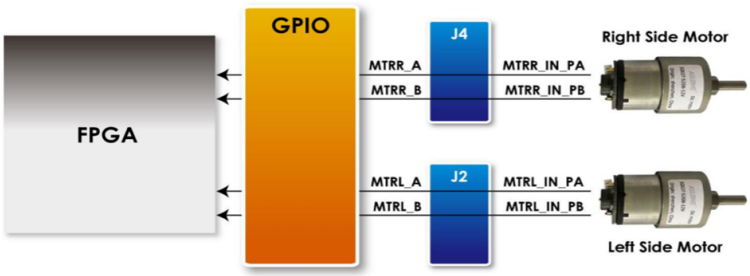


Figure 10: Motor encoder sensor

5.3.3 POLOLU QTR-8RC Reflectance Sensor Array

The sensor consists of 8 IR LED/photo-transistor pairs mounted at 9.525 mm apart from each other[22]. They are connected in series to reduce the current consumption and each of the sensor provides a separate digital I/O-measurable output. A bypass of one stage is done to enable operation at 3.3 V. The LED current is approximately 20-25 mA making the total board consumption to under 100 mA. These output from the sensor is linearized from the mid-value to determine the left or right control of the robot.

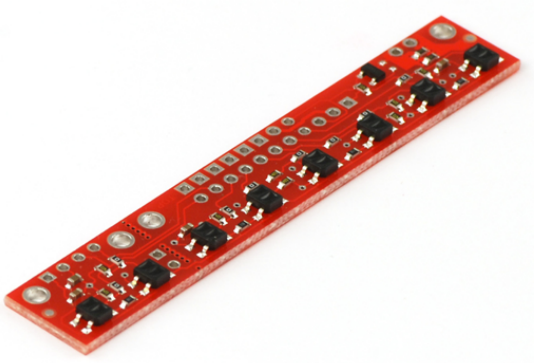


Figure 11: POLOLU QTR-8RC IR ARRAY

5.3.3.1 Linearization Considering the reflection of the infrared being received by the receiver to represent logic 1, the 8 bit binary output from the IR array received at the GPIO pin is linearized as shown in the code provided at the appendix figure 27. So if the robot is aligned with the line in its centre, the linearised output received by the controller would be zero and as the robot deviates from the line, based on the left or right side motion of the robot, the error becomes positive or negative and this value determines the control signal of the controller that is fed to the motors to minimise this deviation and keep the robot aligned with the line. The below figure shows the representation of the IR array facing the line with the linearized weight output.

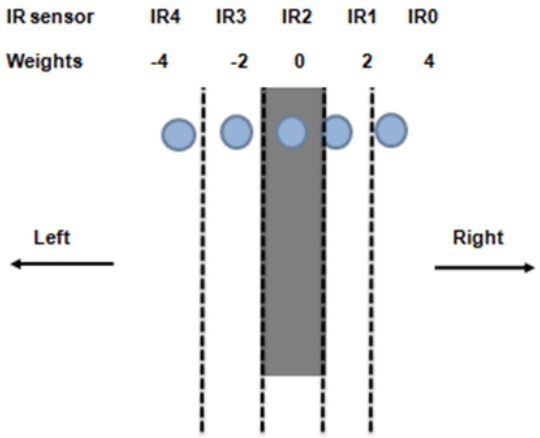


Figure 12: IR array detection on the line

5.3.4 Ultrasonic Sensor

We are provided with the HCSR04 ultrasonic sensor which has a maximum range of 4 meter distance. It works on the principle of trigger-echo ranging. The working is shown in the Figure 13. The sensor generates and sends a signal when the trigger is held for 10 μs. The echo pin is brought to high once the signal is acknowledged by the sensor. Only at the end of 10 μs, a timer is started. The echo signal value falls to LOW which prompts that the sent signal is received. The counter is stopped which gives us the distance between the object(reflected sensor) and the sensor. This is repeated for every 40μs post the trigger. The timing waveform is as shown in the Figure 14.

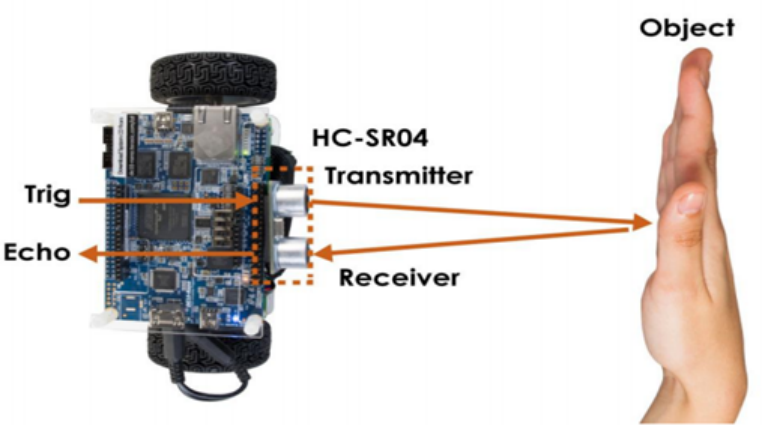


Figure 13: Ultrasonic Sensor Detection

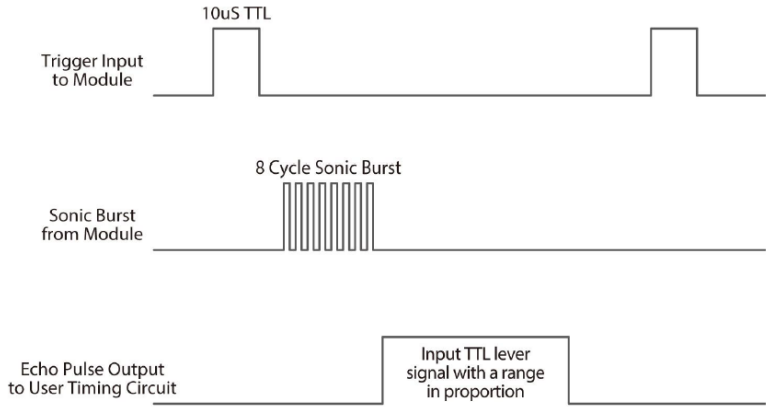


Figure 14: Timing Waveform of the ultrasonic module

5.4 Control system flow

The control system flow for maintaining the balance in the robot and being aligned with the line is as shown in the flowchart in figure 16. Once the program begins, the tilt of the robot from its upright position is been constantly measured and once the robot deviates from vertical angular position zero, the condition to check if the robot is upright fails and the error is taken as the input signal for the LQR controller that generates a control signal to be fed back to the system where the actuator (motors)

are controlled to rotate in a direction that minimises the deviation in the angular position bring the system back to its upright position. Retaining the robot in upright position, the IR sensor array constantly checks for the deviation in the position of the robot from the line it is placed on. If the line is aligned at the centre of the robot (IR sensor array), the linearized output from the sensor is zero and this makes the robot to move forward as per the program control. If the robot deviates from the line, the linear error value being a positive or negative value input to the PID controller that decides the control signal to be fed to the motors and this adds up to the control signal from the LQR for self-balancing and corrects the position of the robot on the line minimising the deviation and also maintaining the self-balance.

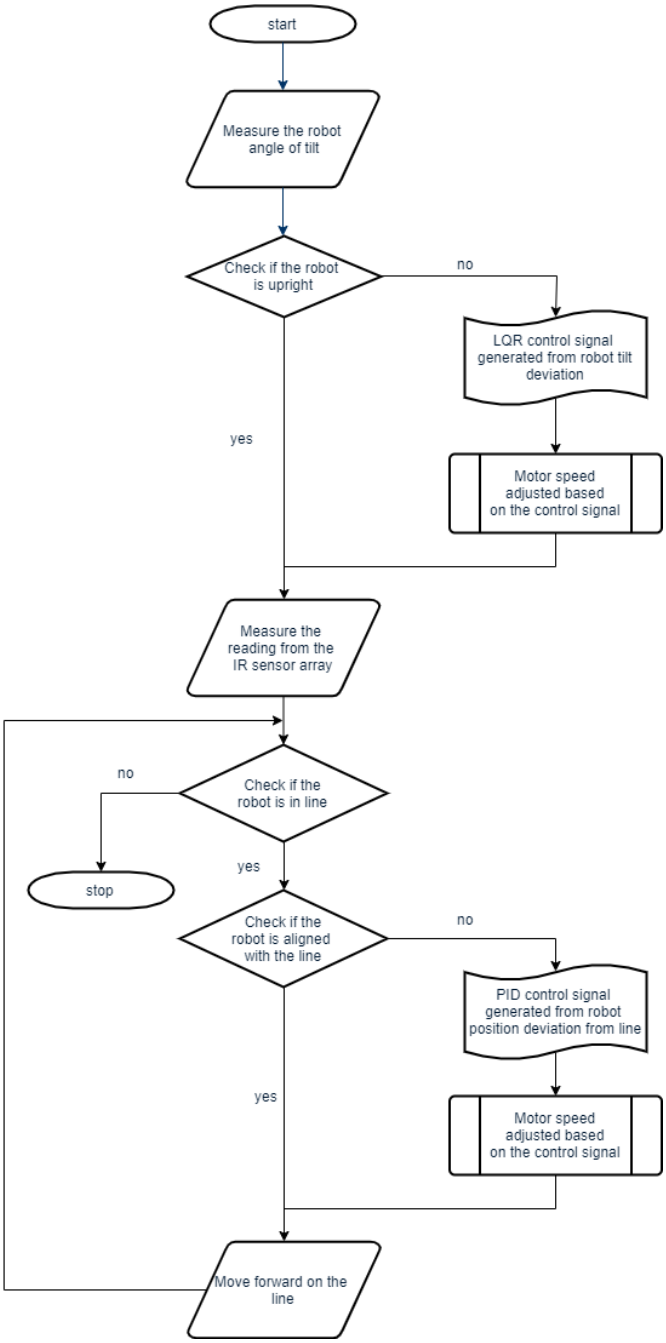


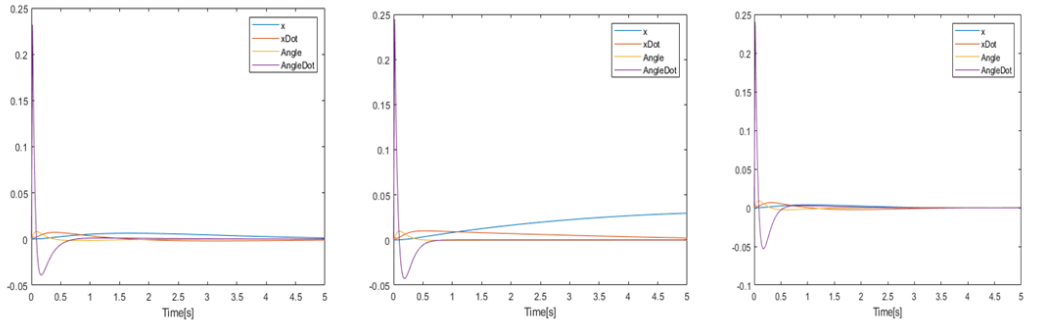
Figure 15: Control system flowchart

6 Controller Tuning

6.1 LQR Controller

The impulse response of the key parameters of the system i.e the positional displacement - x, positional velocity - $\dot{x}$, angular displacement - θ and the angular velocity - $\dot{\theta}$ with respect to time is simulated in MATLAB as shown in the Appendix L and optimised to have a shorter settling time for quicker control of the robot in the upright position. Initially the simulation was performed with Q and R weighting factors set to an identity matrix and the response in the leftmost shows the settling of the key parameters after a time period of above 5 ms. Hence, the tuning of the parameters are needed. By increasing the value of R, it is identified that the system came to equilibrium with a change in its initial position as shown in the plot in the middle. Hence, on further modifying the Q and R accordingly, the impulse response is optimised to settle earlier with a better performance as shown in the rightmost plot. The corresponding gain values are identified as below:

$$K_x = 0.18, K_{\dot{x}} = 0.79, K_\theta = 17.30, K_{\dot{\theta}} = 1.49$$



Initial (Slow) Response | Large distance deviation | Response with optimal Q and R

Figure 16: System with LQR control simulation response

6.2 PID controller

Proportional Integral and Derivative (PID) controller is used to implement smooth line following capability of the self-balancing robot. In order to minimize the error i.e. deviation of the robot from the line, we planned to introduce PID controller.We used trial and error method to fine tune the controller. These are the steps we used to optimize the controller.

1. Only Proportional controller is introduced to the system which resulted in minimization of the error. The proportional gain K_p was set to a higher value.

2. When considerable overshoot was observed we introduced Integral term to reduce the overshoot. The initial value was taken as 10% of K_p is added to the Proportional controller.

3. Although the overshoot has reduced, the integral term introduced oscillations in the system. Derivative term was added to damp the oscillation where the derivative gain K_d is taken as ten times the value of Proportional gain K_p.

The optimised gain K_p, K_i, K_d values of the PID controllers are 7, 0.000005, 27 respectively.

7 Hardware and Software integration

7.1 Hardware

Our Terasic self-balancing robot subsumes two main control boards. One of them is DE10-Nano Board and the other is the Motor Driver Board [23]. DE10-Nano Board is responsible for the logic controls such as self-balancing, line following, and ultrasonic obstacle avoidance. Motor board takes care of the control signals of the motors and motor control with the help of a motor driver chip. The motor board is responsible for controlling accelerometer, gyroscope and ultrasonic encoder data and sends it to DE10-Nano board [17]. We will discuss each board briefly in the following sections.

7.1.1 DE10-Nano Board

DE10-Nano board consists of a Cyclone V SoC FPGA, a Hard Processor System (HPS), set of peripherals and programmable logic [17]. It contains ARM processor. This facilitates to run the code in Linux Operating System(OS) too. Figure 17 shows the detailed components of Cyclone V SoC [23]. The components of the DE10-Nano board are given below [24].

FPGA Device

- Intel Cyclone V SE 5CSEBA6U23I7 device (110K LEs)
- Serial configuration device - EPCS64
- USB-Blaster II onboard for programming; JTAG Mode
- HDMI TX, compatible with DVI 1.0 and HDCP v1.4
- 2 push-buttons
- 4 slide switches
- 8 green user LEDs
- Three 50MHz clock sources from the clock generator

- Two 40-pin expansion headers
- One Arduino expansion header (Uno R3 compatibility), can be connected with Arduino shields
- One 10-pin Analog input expansion header (shared with Arduino Analog input) A/D converter, 4-pin SPI interface with FPGA

Hard Processor System (HPS)

- 800MHz Dual-core ARM Cortex-A9 processor
- 1GB DDR3 SDRAM (32-bit data bus)
- 1 Gigabit Ethernet PHY with RJ45 connector
- USB OTG Port, USB Micro-AB connector
- Micro SD card socket
- Accelerometer (I2C interface + interrupt)
- UART to USB, USB Mini-B connector
- Warm reset button and cold reset button
- One user button and one user LED
- LTC 2x7 expansion header

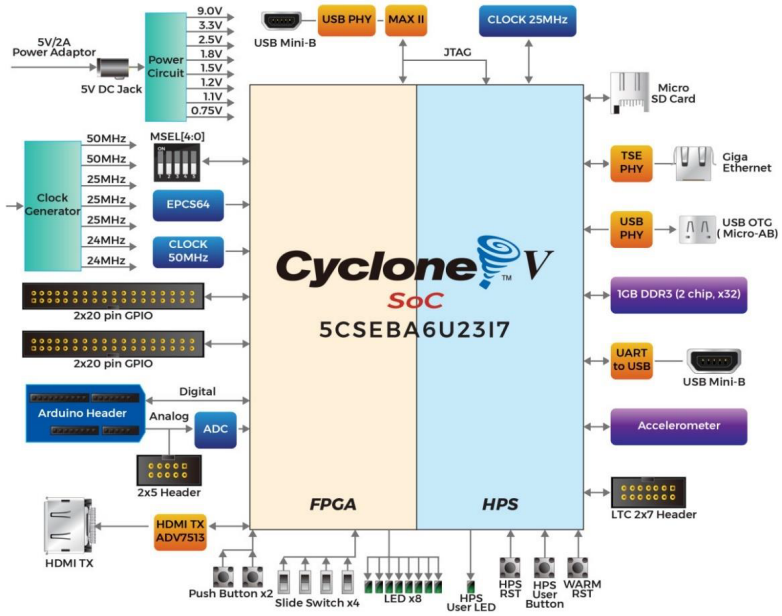


Figure 17: Cyclone V SOC chip [23]

NIOS II processor can be realized on this board. So, we planned to make use of NIOS II for control logic such as self-balancing, turning and speed control of the left and right motors, line following, and ultrasonic obstacle distance calculation. The block diagram of NIOS II control is shown in Figure 18 [23]. DE10-Nano board is connected to the motor driver board through GPIO pins.

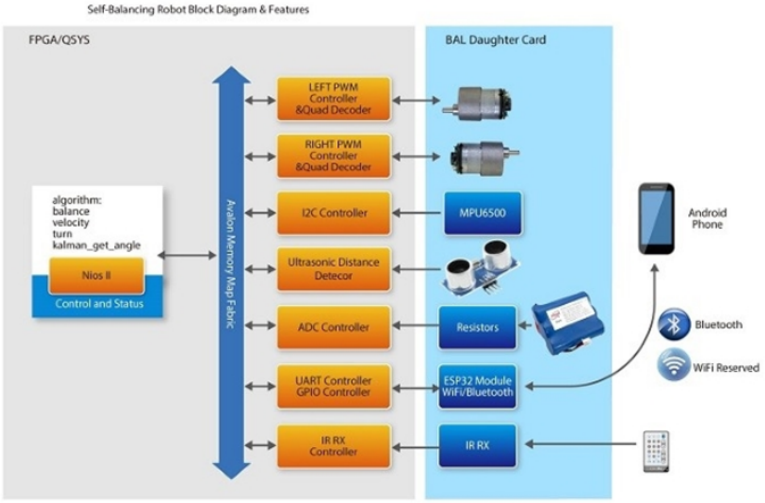


Figure 18: DE 10 Nano board control flow [23]

7.1.2 Motor Control Board

The motor driver board deals with the control signals for the motors and control of the motors which are connected to the wheels of the robot [24]. The motor driver board consists of the following components [24].

- DC Motor Driver and Connectors
- Wi-Fi/Bluetooth module
- 2X20 GPIO Connector to the DE10-Nano Board
- IR Receiver
- ADC Power Monitor
- Ultrasonic Connector
- 12V Power Input
- 5V Power Output to FPGA Board
- Six-Axis (Gyro+Accelerometer) MEMS Motion Tracking

Signals from DE10-Nano board are received by the motor driver which makes the motor to drive at the speed decided by the logic in the NIOS II [17]. Inertial Measurement Unit (IMU) is also included in this motor driver board. Ultrasonic sensor is connected to motor driver board through the Ultrasonic connector and Hall-effect sensor is attached to the motors and they are connected to the motor driver board [17]. This motor driver board also includes the IR remote control receiver module [24]. The different sensors and its description are shown in table 1.

Sensors	Description
MPU-6500 IMU (Accelerometer and gyroscope)	It is present in motor driver board. Used to measure the tilt angle of the robot during self-balancing.
Ultrasonic sensor (HCSR04)	Connected through Ultrasonic connector. Used to detect obstacle and measure the distance between sensor and the obstacle.
Motor Encoder (Hall-effect sensor)	It is connected to the motor and to the motor driver board. Used to determine the position and velocity of the motor.

Table 1: Sensors and Usage

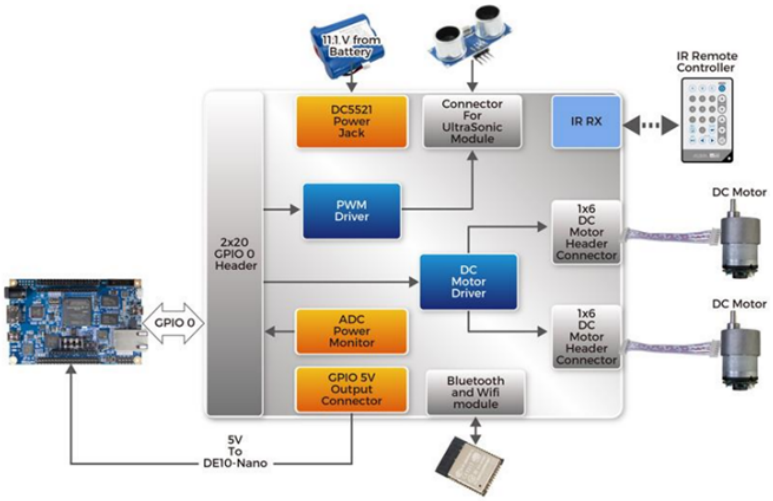


Figure 19: Motor control board integration with hardware [23]

These sensor values from the hardware are used in NIOS II for the self-balancing and obstacle avoiding logic.

7.1.3 Infra-Red Sensor

We are provided with QTR-8RC Reflectance Sensor Array and the working of this sensor has been discussed in the section 5.3.3. This is interfaced with the DE10-Nano board through GPIO 1 pins. The value from the IR sensors are linearized between -7 to +7 with reference as 0. Getting the IR value and those values are linearized using Verilog code in Quartus. These linearized values are utilized in NIOS II processor for line following. The verilog code for linearizing *linearizer* is shown in Appendix 26.

7.2 Software

The data from the hardware, for example, IR array, ultrasonic , IMU values and more are used for the logic controls in which NIOS II is realized in the Cyclone V SoC FPGA. In NIOS II, the following logic are implemented.

1. Self-balancing with LQR controller.
2. Calculating tilt angle from the values of MPU6500.
3. Calculating the distance between Ultrasonic sensor and the obstacle.
4. PWM Velocity of wheel using Motor wheel encoders.
5. Line following with PID controller using the IR array readings.

We have used Quartus for the verilog codes and Eclipse IDE in order to build the NIOS II code.

7.2.1 Balancing Algorithm

In the demo code of the Terasic Self-balancing robot, the self-balancing algorithm was using PD Controller [25]. We have modified the code in such a way that now, the self-balancing of the robot is implemented using

the Linear Quadratic Regulator. This is implemented in C++ using the Eclipse IDE for Nios II Software build tools. This balancing algorithm is governed by the data from the gyroscope and the Kalman filtered angle. The code in the *int balance* of the demo code is replaced with the LQR control as shown in Appendix D.

7.2.2 Turn Algorithm

This is also an existing code in the Terasic Self-balancing robot demo where the turn angle calculated using the z-axis of MPU-6500 and difference in speed of two encoders [25]. The rotational angle is controlled by PD controller. This is the *int turn* function of the demo code. The code snippet is shown in Appendix E.

7.2.3 Speed Algorithm

The speed control algorithm uses PI controller in order to control the speed of the motors. This is the same existing of code of Terasic Self-balancing robot is used where P controls speed deviation and I for the displacement [25]. The C++ code for this speed control is shown in Appendix F.

7.2.4 Line following Algorithm

In this function, the correction value from the PID controller for the respective IR value. The following table 2 shows the movement of the robot based on the correction. The code snippet for the line following function is given in Appendix G.

IR values	Movement
-7 to -1	Robot turns right.
0	Robot moves Forward.
+1 to +7	Robot turns left.
9(other and default value)	Robot pauses.

Table 2: IR values and respective movement of Robot

7.2.5 PID control function

The exported linearized value from the linearizer of verilog code is used in this NIOS II and this is done by the function $GetValue()$ in IRArrayController.cpp. This code snippet is shown in the Appendix H. The linearized infra-red array values are the error of the process. So, as per (1), code in C++ has been written as shown in Appendix I.

7.2.6 Main function

The main function measures the distance using Ultrasonic sensor, gets the values of IR from *IRArrayController.cpp* and this is the input parameter of the line following function *follow_line*. This line following function is called only if the boolean flag $'run'$ is true. The flag $'run'$ is set true only if '2' is pressed in the remote. Once, '2' is pressed, the line following *follow_line* function will be executed. The flag $'run'$ is set False at default or if '5' is pressed in the remote when flag $'run'$ is already set True and this pauses the robot. The code for the main is given in Appendix K.

7.2.7 Interrupt algorithm

As discussed earlier, this project comprises of priority based tasks. When the line is detected, the robot follows the line only after the event of self-balancing, absence of obstacle in front of the robot and '2' is pressed in the remote controller. So, after self-balancing and if there is no obstacle in the front of the robot, the line will be followed by the robot. If there

is an obstacle in the path of the robot, no line found or '5' is pressed on the remote controller, pauses the robot. This prioritization of tasks is done within the Interrupt Service Routine (ISR). This ISR is executed by NIOS II whenever an interrupt occurs. This function also sets only the speed of the robot based on the balance, speed and turn function. The sampling values of these functions are controlled at regular intervals say 10 ms which is as same as the demo code [25]. The code is shown in Appendix J. The connections of NIOS II are summarised in the Figure 20.

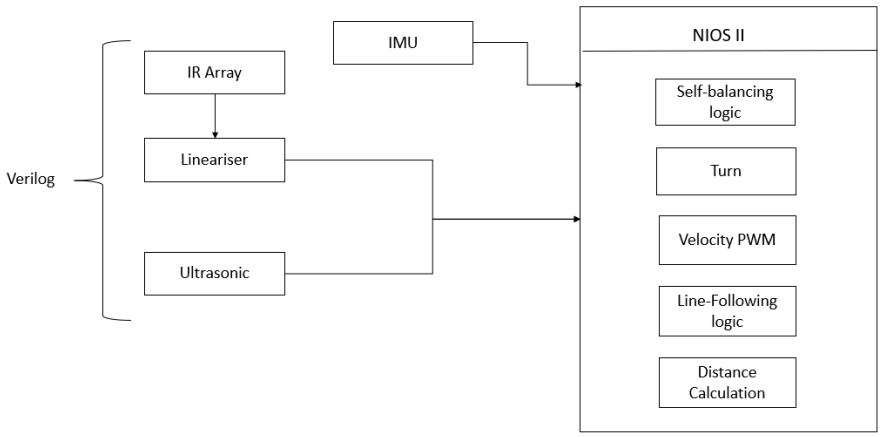


Figure 20: Nios ll connection

7.3 Integration

For the integration of these Hardware verilog code and NIOS II, we are using Qsys tool. Qsys is a system integration tool or platform designer and this helps in connecting the IPs and subsystems [26]. The verilog code of the IR array and the linearizer are instantiated in a top module where the value from the IR array is sent to the linearizer. So, after building the project in Quartus, we have to add a component 'ir_array_ctrl_0' in the Qsys which has the created top module as its implementation. Once, the parameters and signals and interfaces are sorted for the newly created component, the connections to the NIOS II have to be made. So, the data_master of the NIOS II gen controller is connected to the avalon

slave of the newly created component. The IR array values i.e. linearised value will be exported. Once the HDL of Qsys is generated which forms the connection between these Hardware and NIOS II processor. Qsys schematic diagram is shown in Figure 21.

At the end of the generating HDL and building the verilog project, .sopcinfo and .sof files are generated. The generated .sopcinfo file is used in order to generate Board Support Package (BSP) files using Eclipse IDE for NIOS II software build tool. This is necessary in order to build the project in NIOS II. This BSP has libraries which are required by the hardware for the software runtime environment [27]. Once, the BSP is created and the project is built, .elf file is created. This .sof and .elf files are put in the *'demo_batch'* folder in the project and the *test.bat* is executed. The integration of the hardware and software can be depicted as shown in Figure 22 shown below.

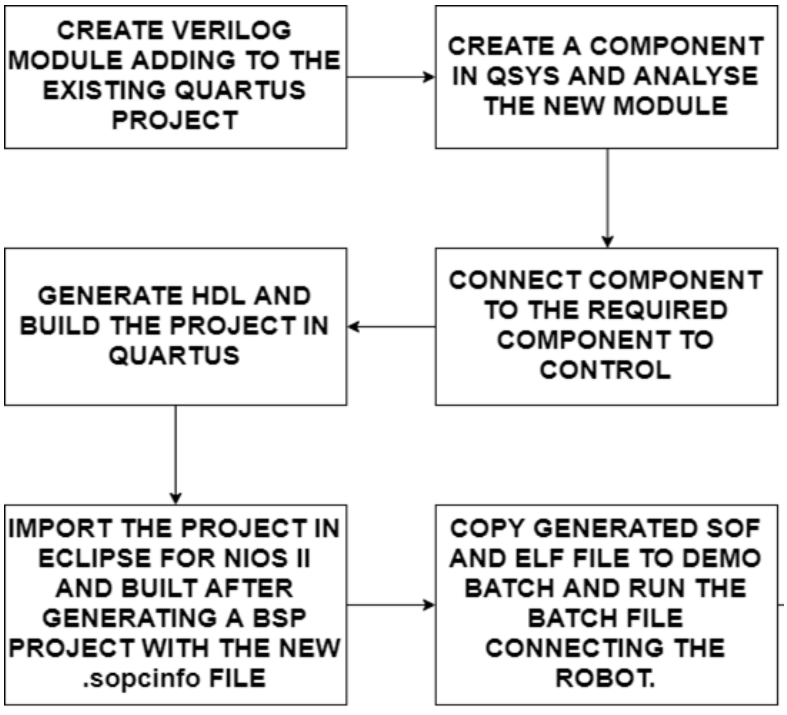


Figure 22: Software design flow

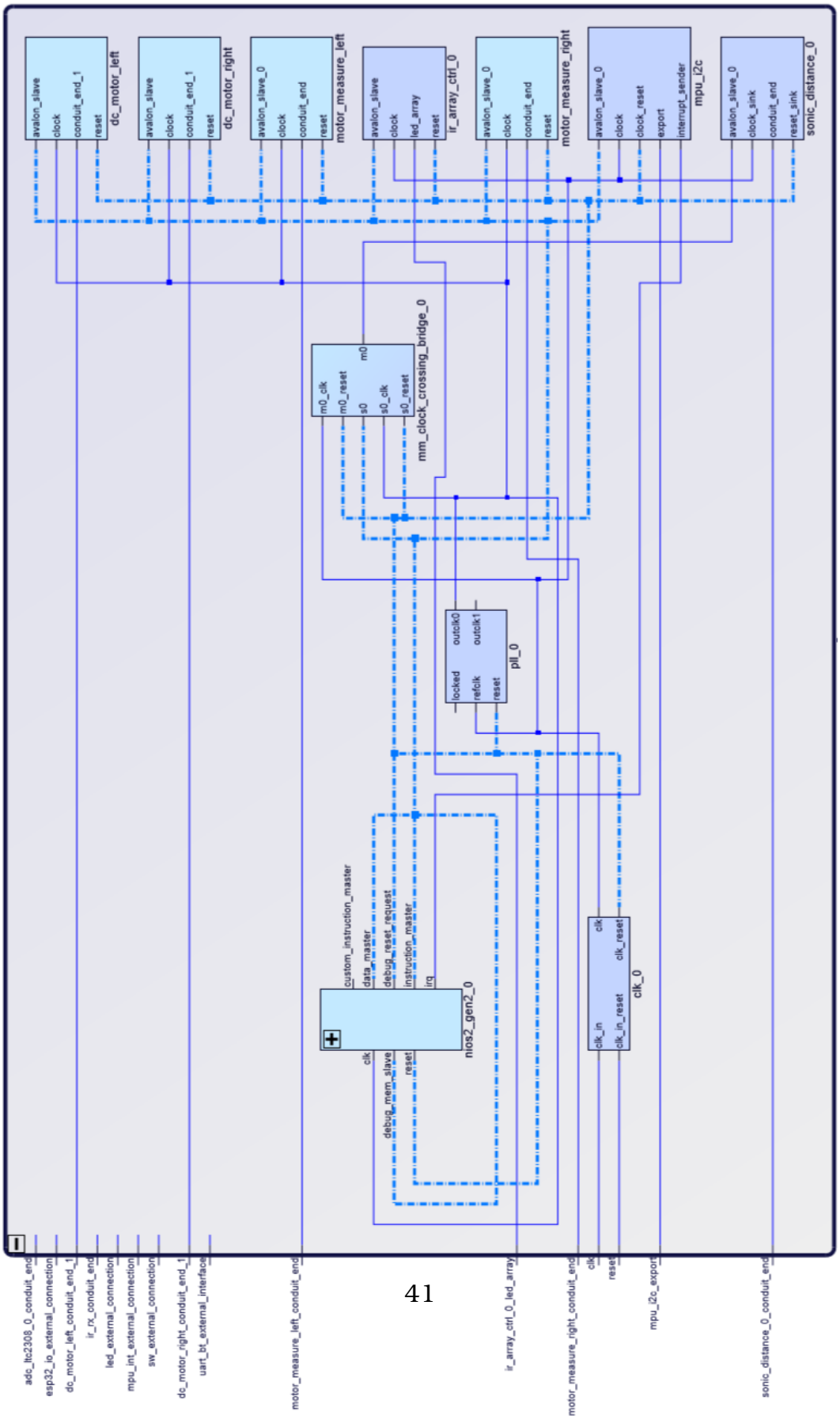


Figure 21: Schematic of connection in Qsys

8 Conclusion

Thus, the objective of this project to design a control scheme for a two wheeled robot and implementing it on a Terasic multi-function robotics platform is achieved. Hence, the two-wheeled robot shall balance itself using a Proportional-Integral (PID) with 2 actuators for motion and input from 6 Dof inertial measurement unit (IMU) is utilised to estimate state of the system. Also, The robot is follows a predefined trajectory marked on the ground (line following) and detects and stops in the line if any obstacle is in its path (obstacle avoidance). The following were the overall project deliverable:

- Interfacing the Pololu IR Array with the DE10 Nano
- Creating modules in Verilog and connecting it in Qsys.
- Programming the Nios ll in C using Eclipse.
- Applying LQR to the control of the balance in the Robot.
- Fine tuning PID in smooth motion of the Robot aligned with the line.
- Modelling a structure to case the IR array attached with the robot.
- Task Prioritising to have control over the motors from multiple control signals.
- Running successful trials and calibrating the performance.
- Workload splitting using Gantt-chart (Appendix B) and management softwares like Trello.
- Preparing and presenting the approach and results.
- Reporting the project for future reference.

As a future expansion, cascading the Controllers i.e implementation of a controller on a system which is already a closed loop controlled system can be possible study to improve the performance of the two-wheeled robots. Also, the obstacle avoidance with a re-routing of the robot's path can be an additional area of interest in this field for achieving a better autonomy.

The project was challenging in finding a way to access the actuator from multiple control masters. Our team had an opportunity to understand the connection of verilog modules with Qsys platform designer and to learn the extensive usage of the Quartus Prime and Eclipse development environments. We also got a platform to practically implement the control algorithm like PID and LQR and visualize their response on a model.

References

[1] M. Gopinathan, A. K. T. Sekar, and A. Girish, *De10nano-balance-car*, https://https://github.com/gmuraleekrishna/DE10Nano-Balance-Car, 2019.

[2] R. Babazadeh, A. Gogani Khiabani, and H. Azmi, "Optimal control of segway personal transporter", Jan. 2016, pp. 18–22. DOI: 10.1109/ICCIAutom.2016.7483129.

[3] K. Sultan, *Inverted Pendulum: Analysis, Design and Implementation*, 1st ed. 2003.

[4] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback control of dynamic systems*. Pearson Prentice Hall, 2006, p. 910, ISBN: 0131499300.

[5] R. Ferrante, "A Robust Control Approach for Rocket Landing", PhD thesis, University of Edinburgh, 2017.

[6] F. Grasser, A. D'Arrigo, S. Colombi, and A. C. Rufer, "Joe: A mobile, inverted pendulum", *IEEE Transactions on Industrial Electronics*, vol. 49, no. 1, pp. 107–114, Feb. 2002. DOI: 10.1109/41.982254.

[7] K. Kankhunthod, V. Kongratana, A. Numsomran, and V. Tipsuwanporn. (2019), [Online]. Available: http://www.iaeng.org/publication/IMECS2019/IMECS2019_pp77-82.pdf.

[8] H. HELLMAN and H. SUNNERMAN. (2015), [Online]. Available: http://www.diva-portal.org/smash/get/diva2:916184/FULLTEXT01.pdf.

[9] S. Kashani-Akhavan and R. Beuchat, 2003. [Online]. Available: http://robotics.ee.uwa.edu.au/theses/2003-Balance-Ooi.pdf.

[10] A. Azar, H. Hassan, M. Barakat, M. Saleh, and M. Abdelwahed, "Self-balancing robot modeling and control using two degree of freedom pid controller", in. Jan. 2019, pp. 64–76, ISBN: 978-3-319-99009-5. DOI: 10.1007/978-3-319-99010-1_6.

[11] A. Al-Mahmood and M. Opoku Agyeman, "A study of fpga-based system-on-chip designs for real-time industrial application", *International Journal of Computer Applications*, vol. 163, pp. 9–19, Apr. 2017. DOI: 10.5120/ijca2017913544.

[12] A. Cakan and F. Botsali, "Modeling and lqr control of a wheeled self-balancing robot", Oct. 2016.

[13] J. Fang, "The lqr controller design of two-wheeled self-balancing robot based on the particle swarm optimization algorithm", vol. 4, 2014. DOI: http://dx.doi.org/10.1155/2014/729095.

[14] V. S. Viswanathan, "Embedded Control Using FPGA", Tech. Rep. November, 2005.

[15] N. M. A. Ghani, F. Naim, and T. P. Yon, 2019. [Online]. Available: https://waset.org/publications/6566/two-wheels-balancing-robot-with-line-following-capability.

[16] M. I. O. TECHNOLOGY, *Understanding poles and zeros*, ch. Analysis and Design of Feedback Control Systems.

[17] I. Inc, *Self-balancing robot based on the terasic de10-nano kit*, 2018. [Online]. Available: https://software.intel.com/en-us/articles/self-balancing-robot-based-on-the-terasic-de10-nano-kit.

[18] E. Wings. (2019), [Online]. Available: https://www.electronicwings.com/sensors-modules/ultrasonic-module-hc-sr04.

[19] S. Kashani-Akhavan and R. Beuchat, 2016. [Online]. Available: https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/SoC-FPGA%20Design%20Guide_EPFL.pdf.

[20] Pololu. (2001-2014), [Online]. Available: https://www.pololu.com/docs/0J12.

[21] O. Impulse. (), [Online]. Available: https://www.openimpulse.com/blog/products-page/product-category/mpu6500-accelerometer-giroscope-module/.

[22] Polulu, "QTR-8A and QTR-8RC Reflectance Sensor Array User's Guide", pp. 1–12, 2001. [Online]. Available: http://www.pololu.com/docs/0J12/all.

[23] T. Inc, *Terasic - robotic kits - self-balancing robot*, 2019. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English%5C&CategoryNo=238%5C&No=1096%5C&PartNo=4.

[24] Terasic, "Terasic DE10-Nano Development Kit", Tech. Rep., 2019. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English%7B%5C&%7DCategoryNo=167%7B%5C&%7DNo=1046.

[25] T. Technologies, *Self-balancing robot, user guide*, 2019. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English%5C&CategoryNo=238%5C&No=1096%5C&PartNo=4.

[26] I. Inc, *Introduction to qsys*, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/support/training/course/oqsys1000.html.

[27] ——, *Intel.com*, 2019. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2sw%5C_nii52015.pdf.

Appendices

A Life cycle

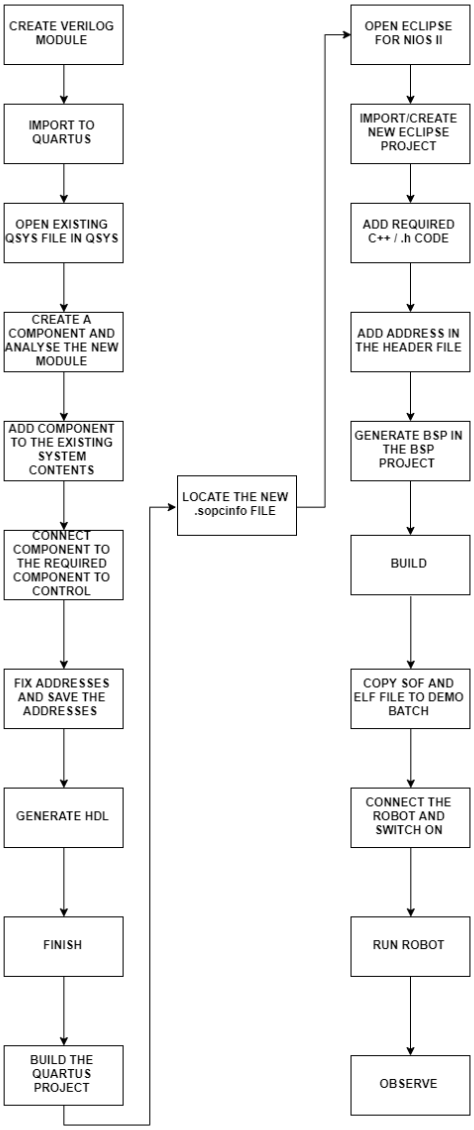


Figure 23: Project Development Life Cycle

B Project development overview

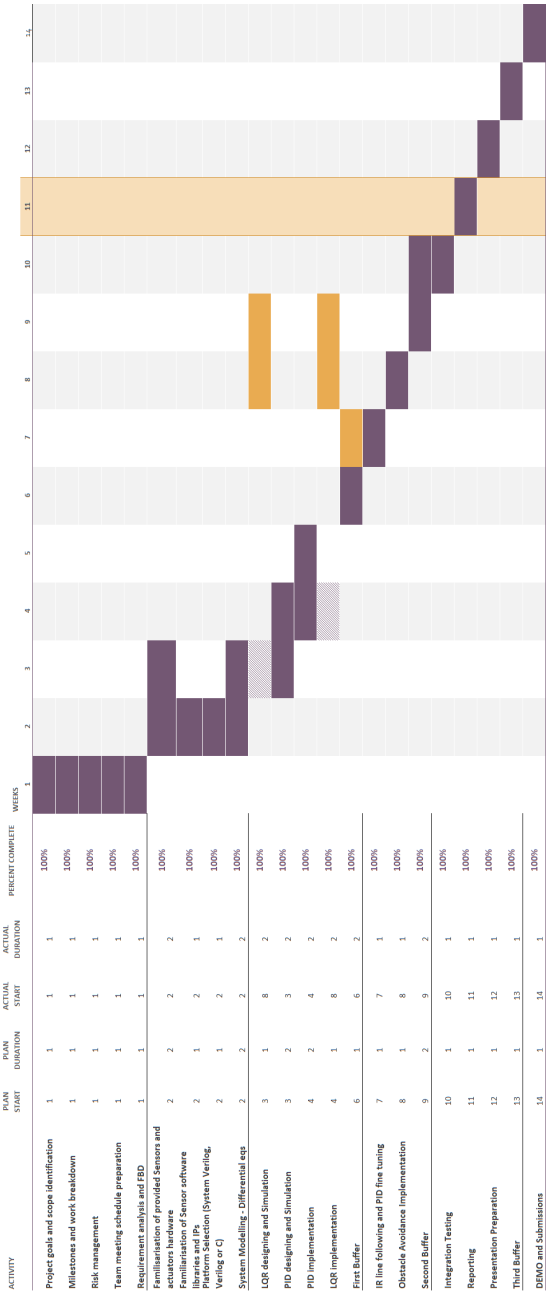


Figure 24: Gantt Chart for workload splitting

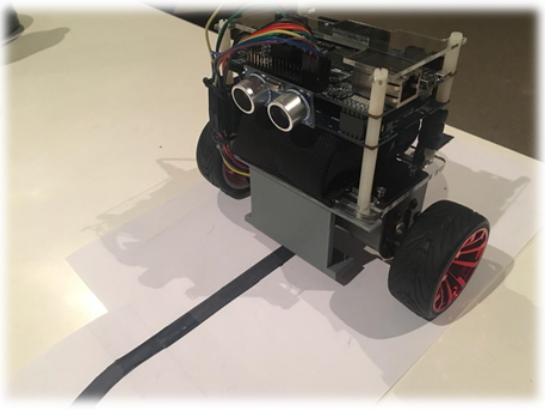


Figure 25: Entire structure with connection and the casing

C Code of the project

```
module linearizer (
    input wire [7:0] process,
    output reg signed [4:0] value
);

    always @(*) begin
        case (process)

            8'b00000001: value <= 5'd-7;
            8'b00000011: value <= 5'd-6;
            8'b00000010: value <= 5'd-5;
            8'b00000110: value <= 5'd-4;
            8'b00000100: value <= 5'd-3;
            8'b00001100: value <= 5'd-2;
            8'b00001000: value <= 5'd-1;
            8'b00011000: value <= 5'd0;
            8'b00010000: value <= 5'd1;
            8'b00110000: value <= 5'd2;
            8'b00100000: value <= 5'd3;
            8'b01100000: value <= 5'd4;
            8'b01000000: value <= 5'd5;
            8'b11000000: value <= 5'd6;
            8'b10000000: value <= 5'd7;

            default: value <= 5'd9;
        endcase
    end
endmodule
```

Figure 26: Linearized IR values

D Balance function

```
/*****************************************************************************
```

```
 This code is modified on the balance function of demo code
 Function     : Upright Closed-loop Control (PD)
 parameter    : The Angle valueThe angular velocity value
 return value : Upright Closed-loop Control PWM
**************************************************************************/
int balance(float Angle, float Gyro) {
static float pi_by_180 = 0.0174, wheel_radius = 0.0315;
static float K_x = 0.18, K_x_dot = 0.79, K_theta = 17.30, K_theta_dot = 1.49;
//static float K_x = 3.16, K_x_dot = 3.81, K_theta = 19.4584, K_theta_dot = 1.55;
float linear_speed, linear_distance;
static float Encoder_Integral, Encoder, Encoder_Least;
Encoder_Least = (-Car.Measure_L + Car.Measure_R) - 0;
Encoder *= 0.8;
Encoder += Encoder_Least * 0.2;
Encoder_Integral += Encoder;
Encoder_Integral = Encoder_Integral - Movement;
if (Encoder_Integral > 8000)
Encoder_Integral = 8000;
if (Encoder_Integral < -8000)
Encoder_Integral = -8000;
if (stop_flag)
Encoder_Integral = 0;
Encoder *= 0.0315;
int balance;
linear_speed = Encoder * wheel_radius;
linear_distance = Encoder_Integral * wheel_radius;
balance = linear_distance * K_x + linear_speed * K_x_dot + K_theta * Angle
+ Gyro * K_theta_dot * pi_by_180;
return balance;
}
```

E Turn Function

```
/**************************************************************************
 This is used from the demo code of the self-balancing robot of Terasic
 Function     : Turn Closed-loop Control (PD)
 parameter    : The Z axis gyroscope value
 return value : Turn Closed-loop Control PWM
 **************************************************************************/
int turn(float Gyro) {
float Bias, kp, kd;
float turn_pwm;
if (apply_turn_correction == true) {
kp = 1.0;
kd = 0.02;
} else {
kp = 0.3;
kd = 0.015; // kp=0.3;kd=0.015;
}
Bias = Car.Measure_L + Car.Measure_R;
if (Car.turn_direction & CAR_TURN_LEFT) {
Bias += 110;
} else if (Car.turn_direction & CAR_TURN_RIGHT) {
Bias -= 110;
}
if ((led3 == 0) && (Angle_Balance < 10 && Angle_Balance > -10))
turn_pwm = 0;
else
turn_pwm = kp * Bias - kd * Gyro;
return turn_pwm;
}
```

F Speed Function

```
/**************************************************************************
 This code belongs to demo code of self-balancing robot of Terasic
 Function     : Speed Closed-loop Control (PI)
 parameter    : The encoder values of the wheelthe
 return value : Speed Closed-loop Control PWM
 **************************************************************************/
int speed(void) {
static float Speed, Encoder_Least, Encoder;
static float Encoder_Integral;
float kp = 3, ki = 0.1; // float kp=2.2,ki=0.05;
Encoder_Least = (-Car.Measure_L + Car.Measure_R) - 0;
Encoder *= 0.8;
Encoder += Encoder_Least * 0.2;
Encoder_Integral += Encoder;
Encoder_Integral = Encoder_Integral - Movement;
if (Encoder_Integral > 8000)
Encoder_Integral = 8000;
if (Encoder_Integral < -8000)
Encoder_Integral = -8000;
if (stop_flag)
Encoder_Integral = 0;
Speed = (Encoder * kp + Encoder_Integral * ki);
return Speed;
}
```

G Line Follow function

```
/**************************************************************************
 This code created by us in order to move the robot based on the PID
 correction
 Function     : Line Following based on PID correction
```

```
 parameter    : value = IR value, distance = ultrasonic distance
 return value : No return value. Movement of the robot is evident.
*****************************************************************************/
void follow_line(alt_32 value, int distance) {
float correction = LineFollowerPID.Calc(value);
//printf("Correction = %.3f\n", correction);
if (value == 9 || distance < 15) {
left_correction = 0;
right_correction = 0;
Movement = 0;
Car.Pause();
}

else if (value == 0) {
Movement = -12;
} else if (value > 0 && value < 8) {
Movement = -15;
left_correction = correction;
right_correction = -correction;
}

else if (value < 0) {
Movement = -15;
left_correction = correction;
right_correction = -correction;
} else {
left_correction = 0;
right_correction = 0;
Movement = 0;
Car.Pause();
}
}
```

H IR Controller Function

```
#include <stdio.h>
#include <math.h>
#include "IRArrayController.h"

//#define     speed_read_reg     0x01
#define     value_read_reg   0x00


IRArrayController::IRArrayController(int BaseAddress):
m_BaseAddress(BaseAddress)
{
}

IRArrayController::~ IRArrayController() {
// TODO Auto-generated destructor stub
//delete m_mmap;
}


/*************************************************************************
Function     :get Infra-Red value of the IR array
parameter    :
return value :
*************************************************************************/
signed short IRArrayController::GetValue(void){ //unsigned
signed short value = 0;//unsigned
value = IORD(m_BaseAddress, value_read_reg);
return value;
}
```

I PID Control function

```
/*********************************************************************
Function     :Calculate Control Signal value based on error
parameter    :
return value :
*********************************************************************/
float PID::Calc(short value){
signed int cur_error = value; //signed int
float error_diff = (prev_error - cur_error) * 0.01;
error_integral += cur_error;

//Integral windup
if(error_integral > 2) {
error_integral = 0;
}

//Calculation of control singal
float u = (Kp * cur_error) + (error_integral * Ki) + (Kd * error_diff);
prev_error = cur_error;
return u;
}
```

J Interrupt Service Routine

```
/*********************************************************************
 This code belong to the Interrupt Service Routine of demo code of Terasic
 Self-balancing robot
 Function     : The interrupt function for 10ms
 parameter    :
 return value :
*********************************************************************/
```

```
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT //nios2 91 edition or later
void MPU_INT_ISR(void *contex)
#else //before nios2 91 edition
void MPU_INT_ISR(void * contex, alt_u32 id)
#endif
{
alt_32 static count = 0;
int cnt;
const int nAdjust = 15;
if (!IORD_ALTERA_AVALON_PIO_EDGE_CAP(MPU_INT_BASE)) {
return;
} else {
IOWR_ALTERA_AVALON_PIO_EDGE_CAP(MPU_INT_BASE, 0x00);
IOWR_ALTERA_AVALON_PIO_IRQ_MASK(MPU_INT_BASE, 0x00);
Get_Angle();
Car.meaure_speed();
cnt_ng = abs(Car.Measure_L) + abs(Car.Measure_R);
if (vol < 10.0)
cnt = 100 - nAdjust;
else if (vol >= 10.0 && vol < 10.5)
cnt = 110 - nAdjust;
else if (vol >= 10.5 && vol < 11.0)
cnt = 115 - nAdjust;
else if (vol >= 11.0 && vol < 11.5)
cnt = 120 - nAdjust;
else if (vol >= 11.5 && vol < 12.0)
cnt = 130 - nAdjust;
else if (vol >= 12.0)
cnt = 135 - nAdjust;
if (cnt_ng >= cnt)
ng_count++;
else
ng_count = 0;
```

```
if (ng_count == 30) {
pick_up = true;
ng_count = 0;
}
if (!stop_flag) {
led0 = 0x01;
UnNomal_Check();
}
if (stop_flag) {
if (Restart_Check())
count++;
else {
led0 = 0x00;
led2 = 0x00;
led3 = 0x00;
Car.Stop();
count = 0;
}
if (count == 100) {
led0 = 0x01;
count = 0;
stop_flag = 0;
Car.Start();
}
}
balance_pwm = balance(Angle_Balance, Gyro_Balance);
turn_pwm = turn(Gyro_Turn);
velocity_pwm = speed();
Car.SetSpeed(-balance_pwm - velocity_pwm + turn_pwm + left_correction,
-balance_pwm - velocity_pwm - turn_pwm + right_correction); //update PWM

}
IOWR_ALTERA_AVALON_PIO_IRQ_MASK(MPU_INT_BASE, 0x01);
```

```
}
```

K Main Function

```
/*************************************************************************
 This code is modified version of demo code of Terasic Self-balancing
 robot.
 Function     : The main function
 parameter    : data
 return value :
*************************************************************************/
int main() {
alt_u32 data;
alt_u32 temp;
alt_u8 number;
alt_u16 Data16;
char szADC_Data[10] = { 0 };
int i = 0;
int Command_EPS32, Command_IR, Param;
char szData[10];
alt_32 ir_value;
float enco_integral;
int balance;
//bool while_case = true;
//float vel = 0,bal_vel=0,turn_vel=0;
printf("Hello BAL\r\n");
mpu.initialize();
IR.Enable();
Car.Stop();
Car.Start();
Car.SetSpeed(0, 0);
led1 = 0x00;
```

```
MPU_INT_INIT();
Command_IR = 0;
while (1) {

//power calculation
read_power(0x01, &Data16);
vol = (float) Data16 * 18.0 / 4.7 / 1000.0;
if (vol < 10.5)
led2 = 0x01;
else
led2 = 0x00;

//distance calculation
data = IORD(SONIC_DISTANCE_0_BASE, 0x00);
distance = (float) data * 34.0 / 100000.0;
ir_value = IRArray.GetValue();
printf("IR value :%d\n", ir_value);
if(ir_value == 9) {
led3 == 0x01;
}

if (!IR.IsEmpty()) {
Command_IR = IR.Pop();
if(Command_IR == CIrRx::IR_NUM_2 && !Run) {
Command_IR = IR.Pop();
Car.Pause();
Car.SetSpeed(0, 0);
Run = true;
IR.Clear();
} else if(Command_IR == CIrRx::IR_NUM_5 && Run) {
Run = false;
IR.Clear();
ir_value = 9;
```

```
Car.SetSpeed(0, 0);
}
} else if (Run) {
follow_line(ir_value, distance);
}


}

led = ((led0 & 0x01) << 7) | ((led1 & 0x03) << 5) | ((led2 & 0x01) << 4)
| (led3 & 0x0f);
IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, led);
return 0;
} //end main
```

L LQR controller MATLAB

```
%*************************************************************************
 This code is to calculate the poles, K value of LQR
%*************************************************************************
clc;
clear all;
close all;
g = 9.81;
r = 0.0315;
M_w = 0.050;
M_p = 0.750;
I_w = 0.000085;
I_p = 0.0028;
len = 0.062;
K_m = 0.005814;
```

```
K_e = 0.005498;
R_w = 3;

bet = (2*M_w+(2*I_w/r^2)+M_p);
alp = (I_p*bet+2*M_p*len^2*(M_w+I_w/r^2));


C = [1 0 0 0;
    0 0 1 0];

D = [0;
    0];


B = [     0;
    (2*K_m*(I_p+M_p*len^2-M_p*len*r))/(R_w*r*alp);
    0;
    (2*K_m*(M_p*len-r*bet)/(R_w*r^2*alp))];


A = [0      1              0           0;
    0 (2*K_m*K_e*(M_p*len*r - I_p - M_p*len^2))/(R_w*r^2*alp)
    (M_p^2*g*len^2)/alp   0;
    0      0              0           1;
    0 (2*K_m*K_e*(r*bet - M_p*len))/(R_w*r^2*alp)       M_p*g*len*bet/alp  0];




system_poles = eig(A)
sys = ss(A,B,C,D);
[num,den] = ss2tf(A,B,C,D)
```

```
figure(1);
impulse(sys);
pzmap(zpk(sys));

a = 5000;
b = 1;
c = 100;
d = 1;

Q = [a 0 0 0;
    0 b 0 0;
    0 0 c 0;
    0 0 0 d];

R = 1;%500
K = lqr(sys,Q,R)

A_f = (A-B*K);
B_f = B;
C_f = C;
D_f = D;

sysf = ss(A_f,B_f,C_f,D_f);
figure(2);
impulse(sysf);

T = 0:0.02:30;
U = zeros(size(T));
U(1) = 1;

[Y,X] = lsim(A_f,B_f,C_f,D_f,U,T);
[n, m] = size(X);
```

```
for i = 1:n
    UU(i) = -K*X(i,:)';
end

poles_new = eig(A_f)
figure(3);
plot(T,[X(:,1) X(:,2) X(:,3) X(:,4)]);
legend('x','xDot','Theta','ThetaDot');
xlabel('Time[s]');
title('Impulse response with LQR control');
```