MASTER OF SCIENCE IN EMBEDDED SYSTEMS DESIGN

Embedded Systems Project

REPORT ON

ANTI-SWAY CONTROL SYSTEM

Under the Guidance of

Prof. Dr.-Ing. Karsten Peter

Submitted by
Nikhil Umesh Dantkale

Submission date: 24, January 2019 **WS - 2018-19**

ACKNOWLEDGEMENT

We are taking this opportunity to express our deepest appreciation to all our respected Professors and fellow colleagues who provided us with the help necessary to complete this project. A special thanks to our project guide **Prof Dr.- Ing Karsten Peter and Prof Dr.- Ing Kai Mueller** whose contribution in stimulating the minds of our group as well as providing us with the constant suggestion throughout the period of work.

Furthermore, we would like to also acknowledge the help rendered by the staff of "**Embedded Systems Design**" in Hochschule Bremerhaven and also show our appreciation for providing us with the required permissions to use the necessary equipment for the project titled, **"Anti-sway control system".**

Lastly, we would like to thank our colleagues for being a part of this journey and being supportive at times of need.

ABSTRACT

The focus of the project is to achieve the position & sway control of cart pendulum system. To achieve this, the mechanical parts of the system has been analyzed, and the non-linear mathematical model has been developed. The obtained non-linear system has been linearized at working point and controllers are designed for the system. The PI and state feedback controllers are designed for position and sway control of cart pendulum system. The system states are obtained from the incremental rotary encoders such as cart linear displacement and pendulum angular displacement and from these derived the angular velocity and linear velocity. The designed controllers are tested with the system model in MATLAB/Simulink. The same controller algorithms are implemented on Microcontroller and tested on cart pendulum system test stand. The experimental results of the implemented controller on real system are also presented.

TABLE OF CONTENTS

Acknowledge **1**
Work Organization **2**
Absract **3**

1 INTRODUCTION **6**
2 PROJECT OVERVIEW **8**
3 MATHEMATICAL MODELLING OF CART PENDULUM SYSTEM **8**
3.1 Non-Linear Model 8
3.2 Linearization of Non-Linear Model 11
3.3 State Space Representation for Linear Model 11
4 MODELLING AND SIMULATION OF CART PENDULUM SYSTEM **13**
4.1 System Modelling 13
4.2 Continuous time controller 18
4.2.1 State feedback controller 18
4.2.2 Proportional (P) Controller 23
4.2.3 PI - Controller 25
4.3 Discrete Time Controller 30
4.4 Fixed Point Controller 33
4.5 S- Function Development 39
4.6 Comparison of Controllers 43
5 SPEED CONTROL OF CART PENDULUM SYSTEM **44**
5.1 Mathematical Modelling of Speed Control Cart Pendulum System 45
5.2 Speed Control Model for Cart Pendulum System 45
5.3 PI Controller 47
5.3.1 Modelling and Simulation of PI Controllers 47
5.3.2 Continuous PI Controllers 48
5.3.3 Discrete PI Controllers 49
5.3.4 PI Controllers in Fixed Point Format 50
5.3.5 PI Controllers in Sensor Fixed Point Format 52
5.3.6 Developing the System Function 54
6 HARDWARE OF CART PENDULUM SYSTEM **57**
6.1 Hardware Description 58
6.1.1 Key features in Spartan 3E 59
6.2 Electrical Interface Circuit 62

6.3 SERVO AMPLIFIER 64
6.4 Power Supply Unit 67
7 HARDWARE DESIGN ON FPGA 68
7.1 User IP Block 68
7.2 Slave Registers 70
7.3 Principle operation of incremental encoder 71
7.3.1 Incremental encoder user IP 72
7.4 Rotary encoder user IP 72
7.5 Digital to Analog convertor (DAC) 73
7.6 DAC Interface Signals 74
7.6.1 DAC input word 75
7.7 SPI Protocol 76
7.7.1 SPI communication user IP for DAC 77
8 SOFTWARE DESIGN 79
8.1 Accessing the position counters 79
8.2 Accessing switches and amplifier 80
8.3 LCD Functions 80
8.4 UART 82
8.5 Accessing DAC 82
8.6 Timer Interrupts 83
9 REFERENCING STAGE 86
9.1 Reference stage state diagram 86
9.1.1 Reference stage C code 88
9.1.2 Velocity PI controller 91
10 STATE MACHINE FOR CP PHASE 92
11 CART PENDULUM SYSTEM GUI 98
11.1 Design 98
11.2 Implementation and Code description 99
11.2.1 FpgaCtrlF.java 99
11.2.2 SerialNetw.java 99
11.3 Input and Output on the GUI 99
11.4 Reference phase 100
11.5 CP phase 100
11.6 Graph Phase 101
11.7 Scroll Pane 101

11.8 Graph Panel 102
12 TESTING AND ANALYSIS 102
12.1 Test case 1 with reference of 0.5 meters 103
12.1.1 Reference position versus cart displacement 103
12.1.2 Angular displacement 104
12.1.3 Manipulating variable (Force) 104
12.2 Test case 2 with external disturbance 105
12.2.1 External disturbance to the pendulum angular displacement 105
12.2.2 Manipulating variable (Force (N)) 106
12.2.3 Reference postion versus cart displacement 106
13 CONCLUSION 107
14 FUTURE WORK 107
15 REFERENCES 108
16 APPENDICES 109
16.1 Appendix – I (Sensor conversions) 109
16.2 Appendix– II (Calculation of friction coefficient on test stand) 111
16.3 Appendix – III (Source Code) 114
16.4 Appendix - IV (HMI CODE) 132

1 INTRODUCTION

The non-inverted pendulum is the classical control problem of an inherently unstable system. During the previous two decades, the endeavor to enhance the handling efficiency of containers at ports have pulled vigorous research in two directions: one is the fast movement of containers between a container ship and trucks, and the other is an automated container handling procedure in the yard. When transferring a container, the swaying phenomenon of the container at the end of flexible ropes makes it difficult to positioning at exact loactaion. This is the same in industries that uses overhead crane to shift heavy objects from one bay to another bay. Below figure shows one such overhead crane used in industries.

Figure 1: Industrial Overhead Crane

The payload which is to be shifted is suspended on the hook block at the bottom. The crane is then moved to shift the payload to the desired position. But the objective here is to move the payload with minimum oscillations so that it doesn't cause any injury to the people on the floor and at the same time doesn't cause any damage to the payload. Thus, it becomes necessary to design a control strategy such that crane moves with minimum oscillation of the suspended mass. This formulates the objective of the project.

A system similar to the one shown in figure 1 can be easily developed by a cart and a pendulum attached to it. In that case, the pendulum with some suspended mass is analogous to the payload and the cart is analogous to the crane. Now, it is physically impossible to arrest the free oscillations of the pendulum without the application of an external force. Thus, the presence of an external force is inevitable.

The project setup similar to the one described above is shown in the figure 2. It involves a simple cart

which runs along the belt driven by motor. Hence motor supplies the external force required for the movement of cart. The pendulum is attached to the center of the cart such that it is free to move without any force restricting its motion. The pendulum is free to move in the same plane as the cart.

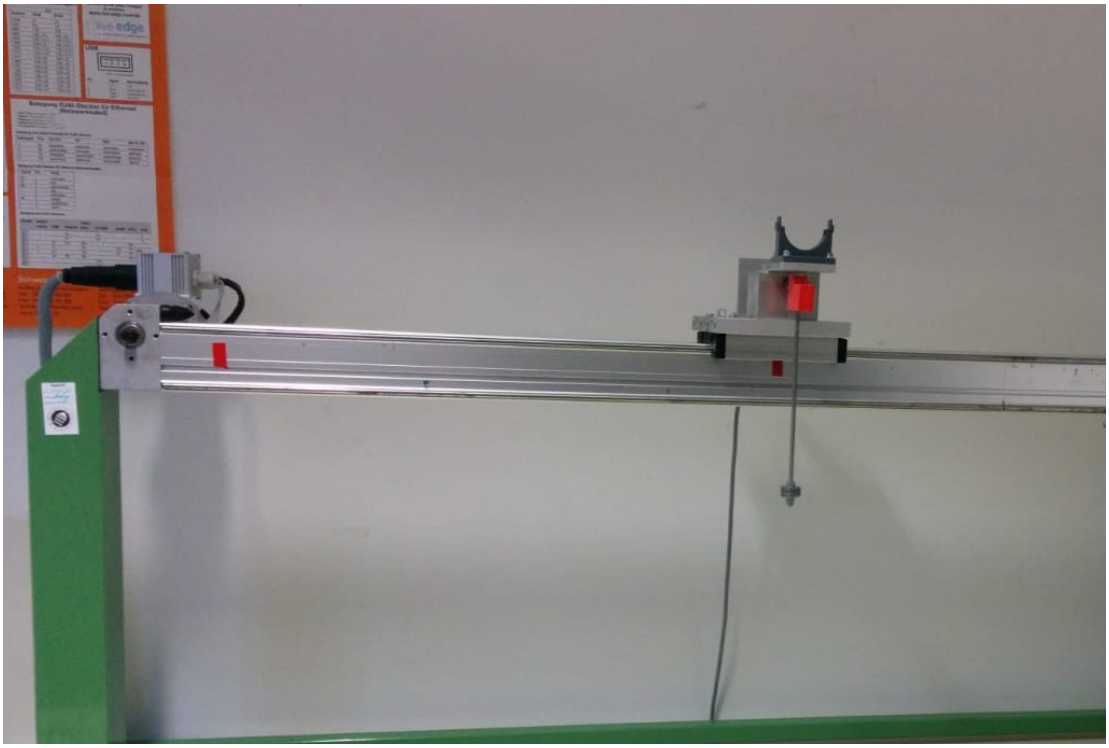


Figure 2: Cart-Pendulum Test Stand

The control strategies adopted here are PI Controller and State feedback control to control the cart position and as well as sway of pendulum. This requires information of the states of the system. To calculate the state of the system two incremental type encoders are provided. One gives the information about the position of the cart on the belt and the other gives the information about the angle of the pendulum. Besides, this limit switches are also provided on either side of the belt to detect the end of the travel span.

With all this set up, the objective is to reach any given reference position on the belt along with damping the oscillations of the pendulum. A step by step approach is carried out to achieve the desired control objectives.

2 PROJECT OVERVIEW

The project is designed with two phases to achieve the desired objectives. The overview of the project is shown in below figure 3.

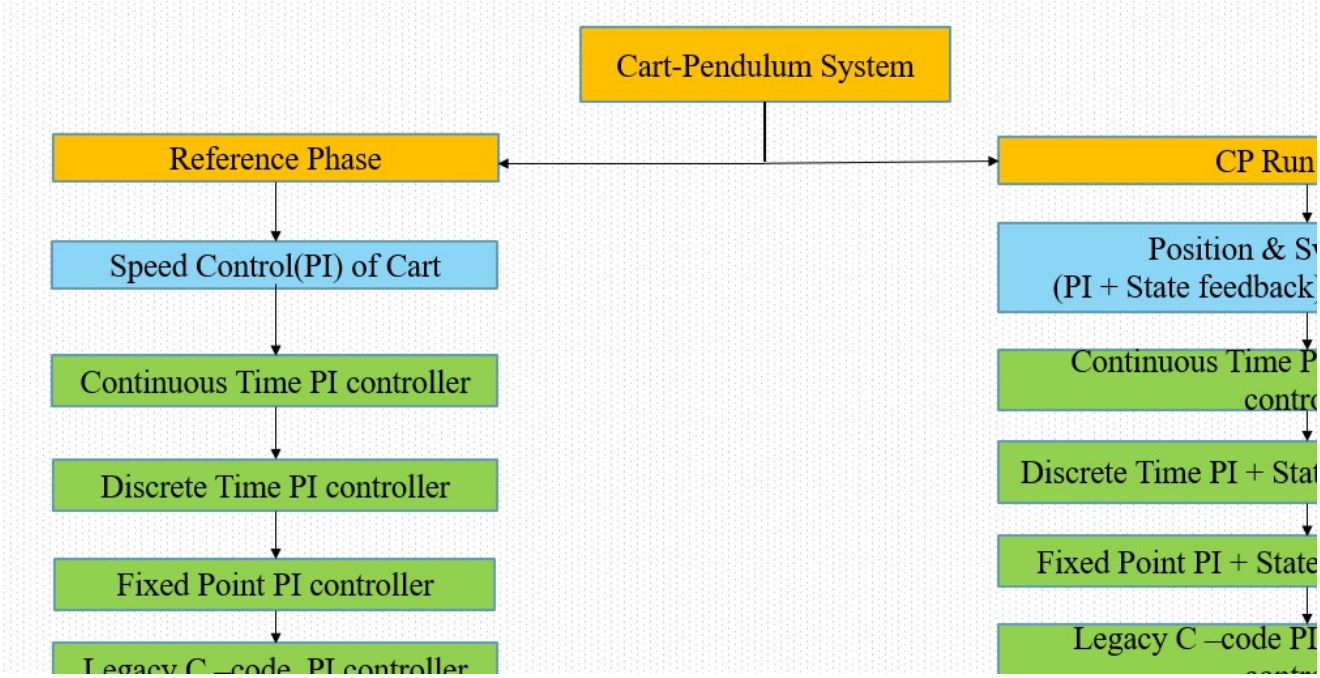


Figure 3: Project overview

3 MATHEMATICAL MODELLING OF CART PENDULUM SYSTEM

Mathematical modelling is a significant tool to gain a clear understanding of the system. In many of the cases, system control algorithms were developed by considering the Newtonian mechanics or the Lagrangian equations. The Lagrangian euations are used to construct the mathematical model of cart pendulum system. But the system model is a nonlinear mathematical model. Therefore, it is converted into a simple linear model using state space control method. This classical approach is popular due to its applicability on most of the control algorithms.

3.1 Non-Linear Model

The cart-pendulum system has two degrees of freedom (2-DoF), namely horizontal displacement of cart and rotation of pendulum around the pivot. A two-dimensional representation of the system can be inferred from the figure 4.

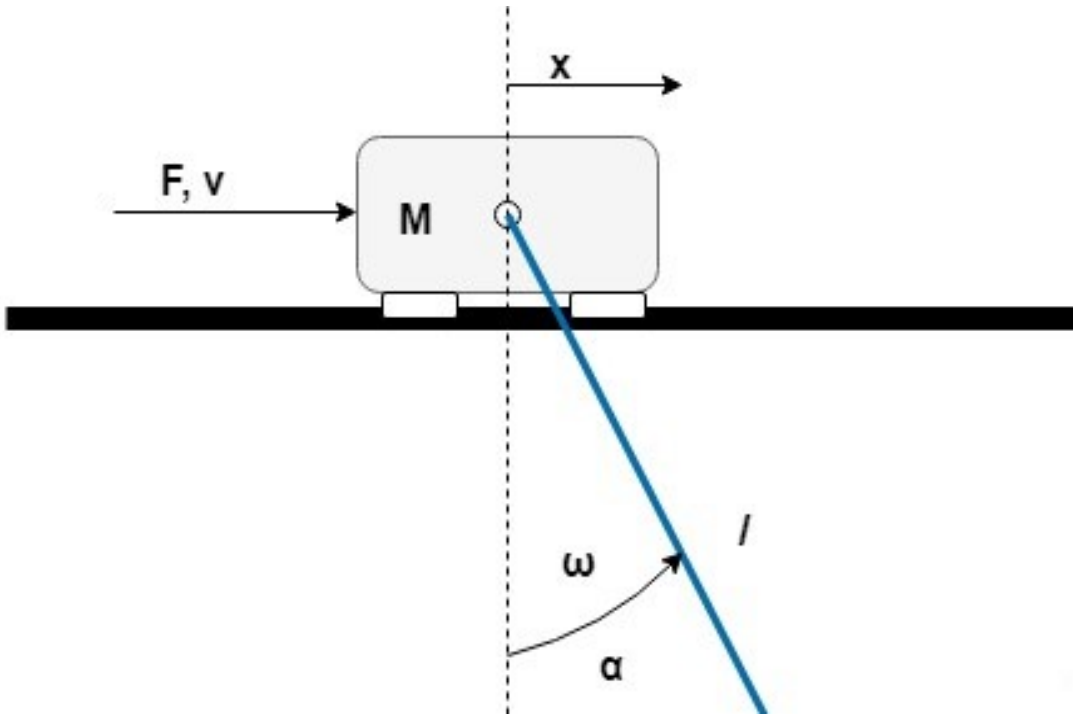


Figure 4: Cart Pendulum System Free Body Diagram

The parameters of the cart pendulum systems are as follows:

$x =$ Linear displacement of the cart(m),
$v =$ Linear velocity of the cart(m/s),
$\alpha =$ Angular displacement of the pendulum(rad),
$\omega =$ Angular velocity of the pendulum(rad/s),
$m =$ Mass of the pendulum(kg),
$M =$ Mass of the cart(kg),
$l =$ Length of the rod(m),
$g =$ Acceleration due to gravity, $9.80655\ (m/s^2)$
$F =$ External force applied to the cart(N).

The assumptions are made on test stand as follows:

- Friction between cart and sledge is neglected
- Friction at pivot of pendulum is neglected
- Mass of the pendulum rod considered in pendulum mass (m)

The equations of motion are derived using Lagrange's method. It is an indirect method and requires judicious selection of the so-called generalized co-ordinates. The criteria for generalized coordinates are successfully fulfilled by the x and α. It is defined by the following formula

$$\frac{\partial}{\partial t}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = Q_i \qquad (1)$$

The vector “q” defines the states of the model. The states represent each physical unit, or in other words the degrees of freedom of the system. The states of the system are the direction x and the angle α.

Generalized coordinates $q_i = \begin{pmatrix} x \\ \alpha \end{pmatrix}$

The vector “Q” contains all non-conservative forces that act on the system. It must be kept in mind that the forces that are present in “Q” act on a certain state, defined in “q”. Therefore “Q” is defined as follows:

$$Q_i = \begin{pmatrix} F \\ 0 \end{pmatrix}$$

The Lagrangian energy (L) for any system is given by,

$$L = T - V \tag{2}$$

Where $T =$ sum of all the kinetic energy of the system and

$V =$ Potential energy of the system

The total kinetic energy (K.E) of the system becomes

$$T = K.E_{Cart} + K.E_{Pendulum}$$

$$T = \frac{1}{2}M \cdot \dot{x}^2 + \frac{1}{2}m \cdot \dot{x}^2 + m \cdot \dot{x} \cdot \dot{\alpha} \cdot l \cdot \cos\alpha + \frac{1}{2}\mathrm{m} \cdot l^2 \cdot \dot{\alpha}^2 \tag{3}$$

The potential energy of the system only contains of the pendulum, because the cart has no degree of freedom in y-direction, and therefore cannot have potential energy. As result, the equation is:

$$V = m \cdot g \cdot l \cdot (1 - cos\alpha) \tag{4}$$

Substitute equations (3) and (4) in equation (2), the Lagrangian energy becomes

$$L = \frac{1}{2}M \cdot \dot{x}^2 + \frac{1}{2}m \cdot \dot{x}^2 + m \cdot \dot{x} \cdot \dot{\alpha} \cdot l \cdot \cos\alpha + \frac{1}{2}\mathrm{m} \cdot l^2 \cdot \dot{\alpha}^2 - m \cdot g \cdot l + m \cdot g \cdot l \cdot cos\alpha \tag{5}$$

The Lagrange equation (1) can now be written with our two generalized co-ordinates as follows,

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\alpha}}\right) - \frac{\partial L}{\partial \alpha} = 0 \tag{6}$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = F \tag{7}$$

Substituting equation (5) in equation (6) and after simplification the following equation is obtained

$$\ddot{x} \cdot \cos\alpha + l \cdot \ddot{\alpha} + g \cdot \sin\alpha = 0 \quad (8)$$

Substituting equation (5) in equation (7) and after simplification the following equation is obtained

$$(M+m) \cdot \ddot{x} + m \cdot l \cdot \ddot{\alpha} \cdot \cos\alpha - m \cdot l \cdot \dot{\alpha}^2 \cdot \sin\alpha = F \quad (9)$$

After re-arranging the equations (8) & (9), the following two non-linear equations of motion are obtained.

$$\ddot{x} = \frac{F + m \cdot l \cdot \dot{\alpha}^2 \cdot \sin\alpha + m \cdot g \cdot \sin\alpha \cdot \cos\alpha}{M + m \cdot sin^2\alpha} \quad (10)$$

$$\ddot{\alpha} = \frac{-F \cdot \cos\alpha - m \cdot l \cdot \dot{\alpha}^2 \cdot \sin\alpha \cdot \cos\alpha - g \cdot (M+m) \cdot \sin\alpha}{l \cdot (M + m \cdot sin^2\ \alpha)} \quad (11)$$

3.2 Linearization of Non-Linear Model

The differential equations (10) & (11) represents dynamic behavior of cart pendulum system at any point of time. But the region of operation of the system in practice is very small where it can be assumed to be linear. Since the cart is displaced very close to its origin and the objective is to keep the oscillations minimum, the system is linearized around the following operating point:

$$(x_0, \dot{x}_0, \alpha_0, \dot{\alpha}_0) \rightarrow (0,0,0,0)$$

By using the Taylor series expansion for linearization around the above-mentioned working point, the following approximations are to be considered.

$\alpha = 0$;
$\cos\alpha \sim 1$;
$\sin\alpha \sim \alpha$;
$\dot{\alpha}^2 \sim 0$;

After substitution of above values in non-linear equations (8) & (9), the following linear system of equations are obtained:

$$\ddot{x} + l \cdot \ddot{\alpha} + g \cdot \alpha = 0 \quad (12)$$

$$(M+m) \cdot \ddot{x} + m \cdot l \cdot \ddot{\alpha} = F \quad (13)$$

Further re-arrangement of equation (12) & (13) and substitution will give the following equations:

$$\ddot{x} = \frac{F}{M} + \frac{(m \cdot g)}{M} \cdot \alpha \quad (14)$$

$$\ddot{\alpha} = \frac{-F}{m \cdot l} - \frac{(M+m) \cdot g}{M \cdot l} \cdot \alpha \quad (15)$$

3.3 State Space Representation for Linear Model

The state-space representation of a linear continuous time invariant system with p inputs, q outputs and

n state variables are written in the following form:

$$\dot{x}\,(t)_{nx1} = A_{nxn} \cdot x(t)_{nx1} + B_{nxp} \cdot u(t)_{px1}$$
$$y(t)_{qx1} = C_{qxn} \cdot x(t)_{nx1} + D_{qxp} \cdot u(t)_{px1}$$

$n = 4^{th}$ order of the system
A = system matrix ($4\ x\ 4$);
B = input matrix ($4\ x\ 1$);
C = output matrix ($4\ x\ 4$);
D = feedforward matrix ($4\ x\ 1$);

From equations (14) & (15) the state space representation can be written as follows:

$$\begin{bmatrix}\dot{\omega}\\ \dot{\alpha}\\ \dot{v}\\ \dot{x}\end{bmatrix} = \begin{bmatrix}0 & \frac{-(M+m)\cdot g}{M\cdot l} & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & \frac{m\cdot g}{M} & 0 & 0\\ 0 & 0 & 1 & 0\end{bmatrix} \cdot \begin{bmatrix}\omega\\ \alpha\\ v\\ x\end{bmatrix} + \begin{bmatrix}-(\frac{1}{M\cdot l})\\ 0\\ \frac{1}{M}\\ 0\end{bmatrix} \cdot F$$

$$y = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \cdot \begin{bmatrix}\omega\\ \alpha\\ v\\ x\end{bmatrix} + \begin{bmatrix}0\\ 0\\ 0\\ 0\end{bmatrix} \cdot F$$

It represents the single input multiple output (SIMO) system. The notations in above equations are as follows:

$$\dot{\omega} = \ddot{\alpha}\ ;\ \ \omega = \dot{\alpha}\ ;\ \ \dot{v} = \ddot{x}\ ;\ \ v = \dot{x}$$

Generally, the internal states of the system are either equal to the order of the system or its degree of freedom. Since there are two differential equations each with an order of two, the state space representation has four internal states.

The measured physical parameters of the system on test stand as follows:
$M =$ 4.676 kg
$m =$ 0.098 kg
$l =$ 0.285 m
$g =$ 9.80665 m/s^2

With these above values the state space matrices become as follows:

$$A = \begin{bmatrix}0 & -35.1305 & 0 & 0\\ 1.0000 & 0 & 0 & 0\\ 0 & 0.2055 & 0 & 0\\ 0 & 0 & 1.0000 & 0\end{bmatrix}$$

$$B = \begin{bmatrix} -0.7504 \\ 0 \\ 0.2139 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The matrices A, B, C, D are the complete representation of the linear cart pendulum system. These matrices are used for the modelling of linear system and the equations (10) and (11) are used for modelling of non-linear system.

4 MODELLING AND SIMULATION OF CART PENDULUM SYSTEM

The previous chapter depicts the non-linear and linear models of cart pendulum system. Now time to analyze the system behavior by using MATLAB/Simulink.

4.1 System Modelling

The cart pendulum system is simulated as both, linear system using state space representation and non-linear system with non-linear differential equations. The M-file 1 shows the Matlab script for both non-linear and linear models.

```
%%Cart Pendulum System Parameters%
M = 4.676;          % Mass(kg) of the cart ( 2.03+2.646= 4.676)
m = 0.098;          % mass(kg) of the pendulum (0.098)
l = 0.285;          % length(m) of the pendulum (0.285)
g = 9.80665;        % accceleration due to gravity (m/s^2)
x0 = [0;0;0;0];     % Initial condition of the system
%Linear Model
A = [ 0 -(M+m)*g/(M*l) 0 0 ; 1 0 0 0 ; 0 m*g/M 0 0 ; 0 0 1 0 ];
B = [-1/(M*l); 0; 1/M ;0 ];
C = eye(4);
D = [0; 0; 0; 0];
```

M-File 1: Matlab script for Linear and Non-Linear Models

The figures 5 & 6 represents the non-linear model of cart pendulum system.

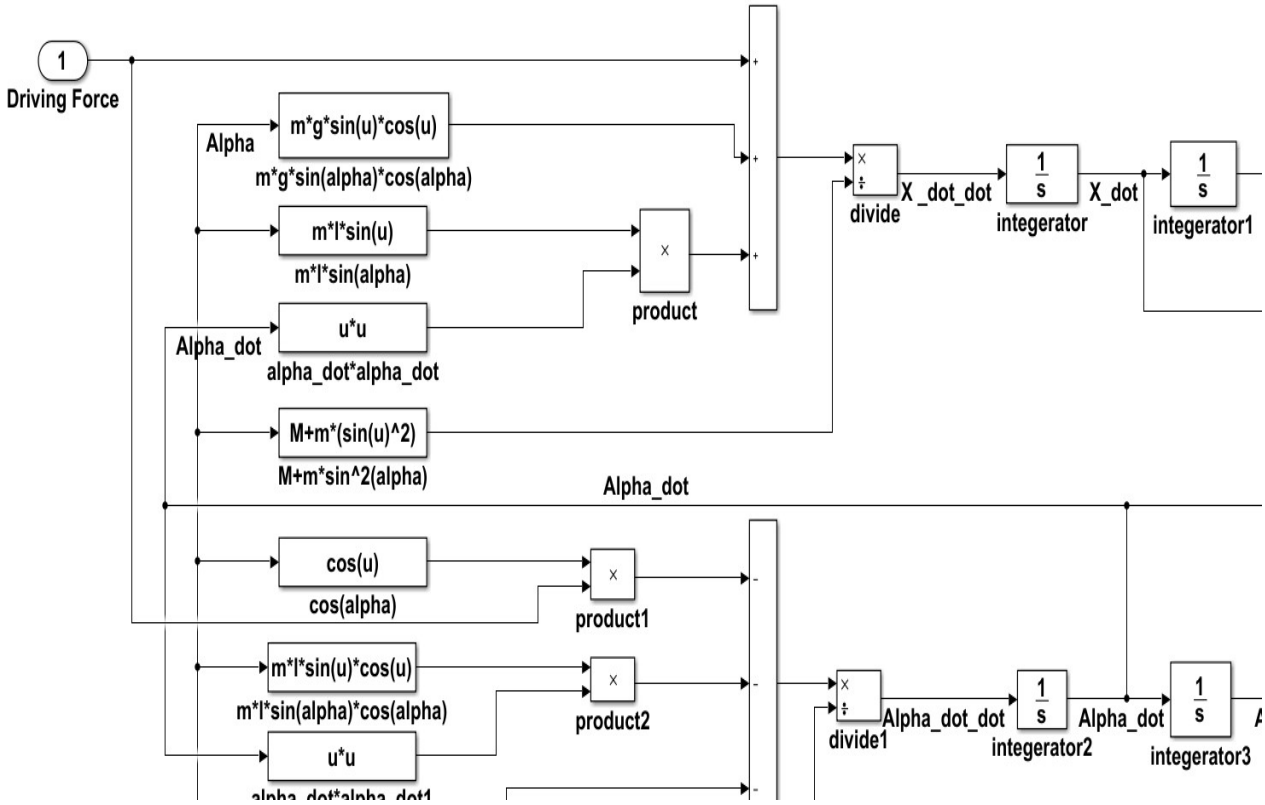


Figure 5: Subsystem for non-linear model

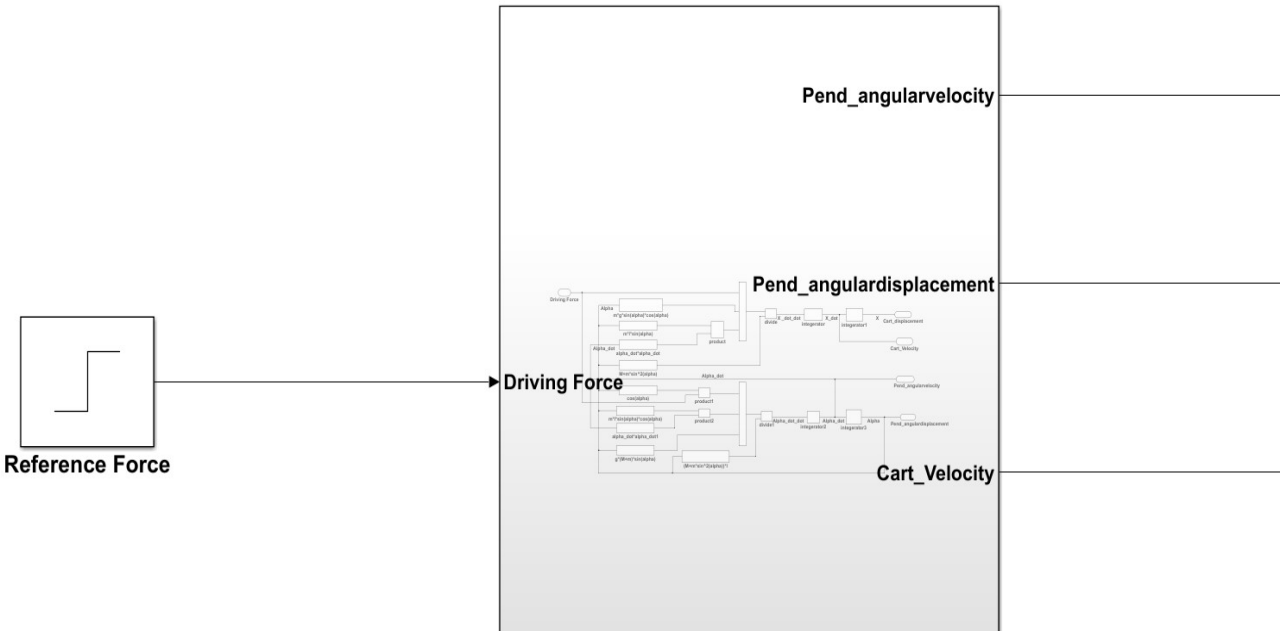


Figure 6: Open loop non-linear model of cart pendulum

The figure (7) shows the model of the linear state space system for cart pendulum. The state space block available in Simulink is used for it. It is configured with the obtained state space matrices A, B, C, D and with an initial condition of $x = 0, \dot{x} = 0, \dot{\alpha} = 0, \alpha = 0$, i.e. the state of rest. Moreover, as seen in both the models the system has '$Force$' as input and output are all the states of the system.

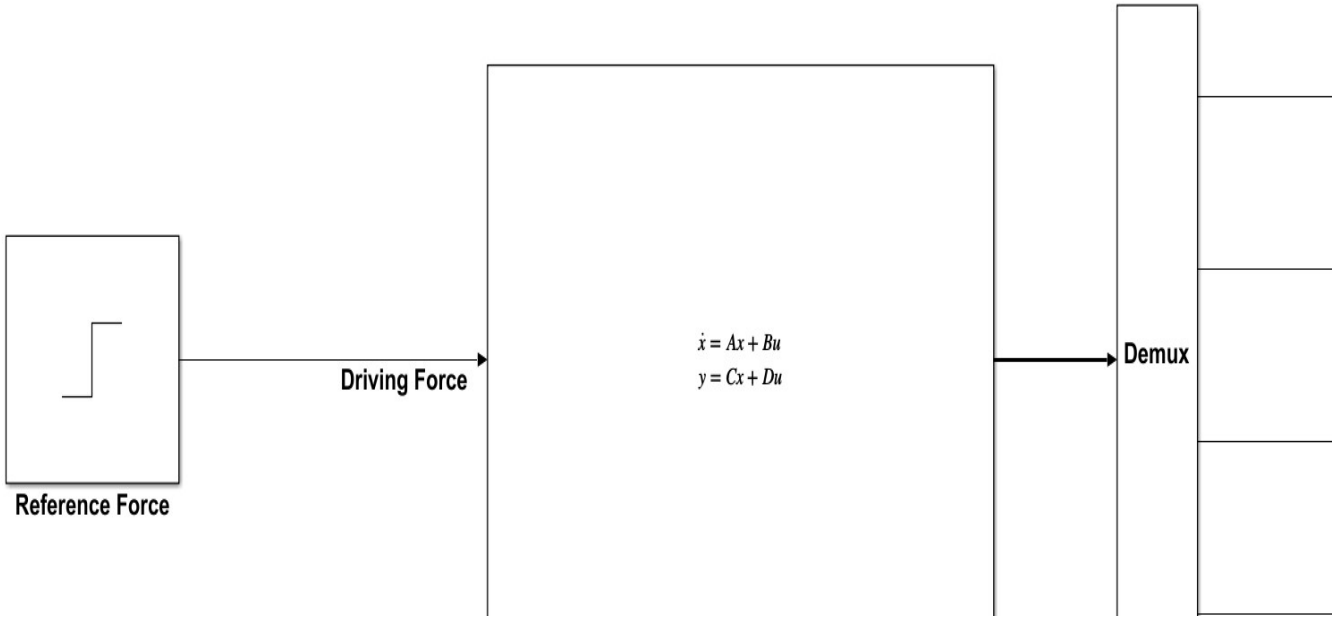


Figure 7: Open loop linear model of cart pendulum

The unit step response (Force = 1 N) of the both linear and non-linear systems are as follows:

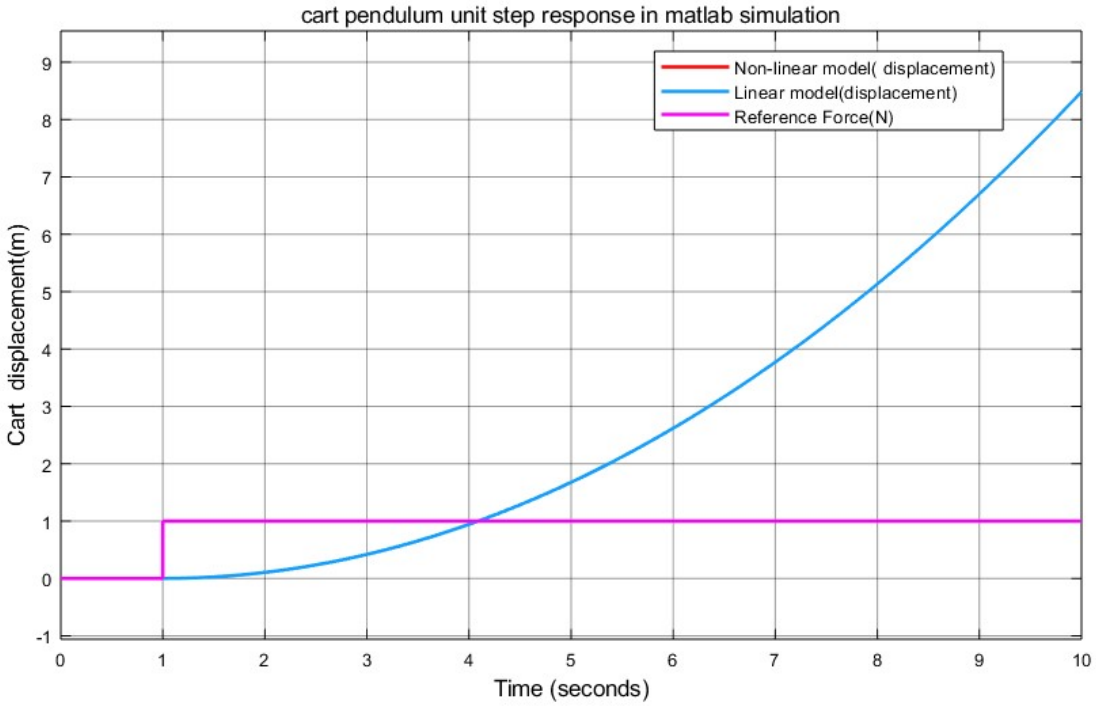


Figure 8: Cart displacement response of non-Linear & Linear models

The cart position is shown in Figure 8. Only one graph can be seen. This indicates that with the step, the linearization is close enough to operation point, otherwise differences could be seen. The cart is accelerated constantly due to constant unit step force, and this results in an exponential increase of position.

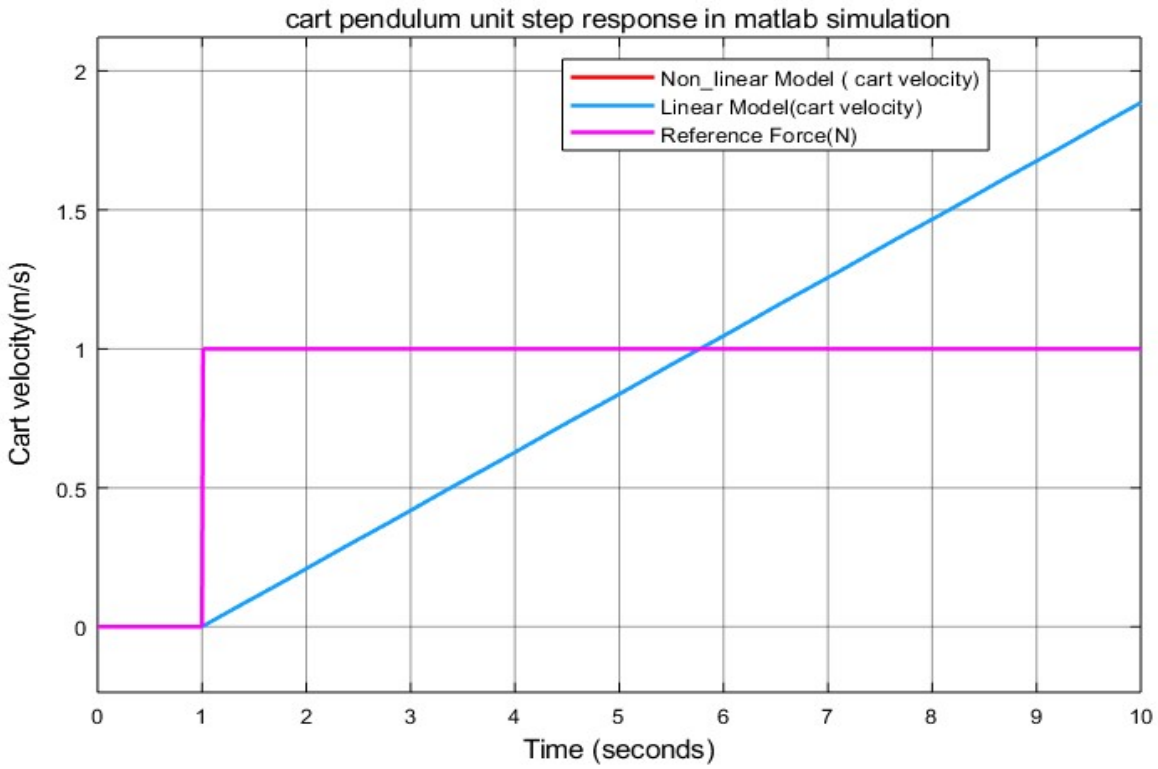


Figure 9: cart velocity response of non-linear & linear models

The cart velocity is shown in Figure 9. Again, only one graph can be seen. This indicates that with the step, the linearization is close enough to operation point. The cart is accelerated constantly due to constant unit step force, and this results in linear increase of cart velocity as expected.

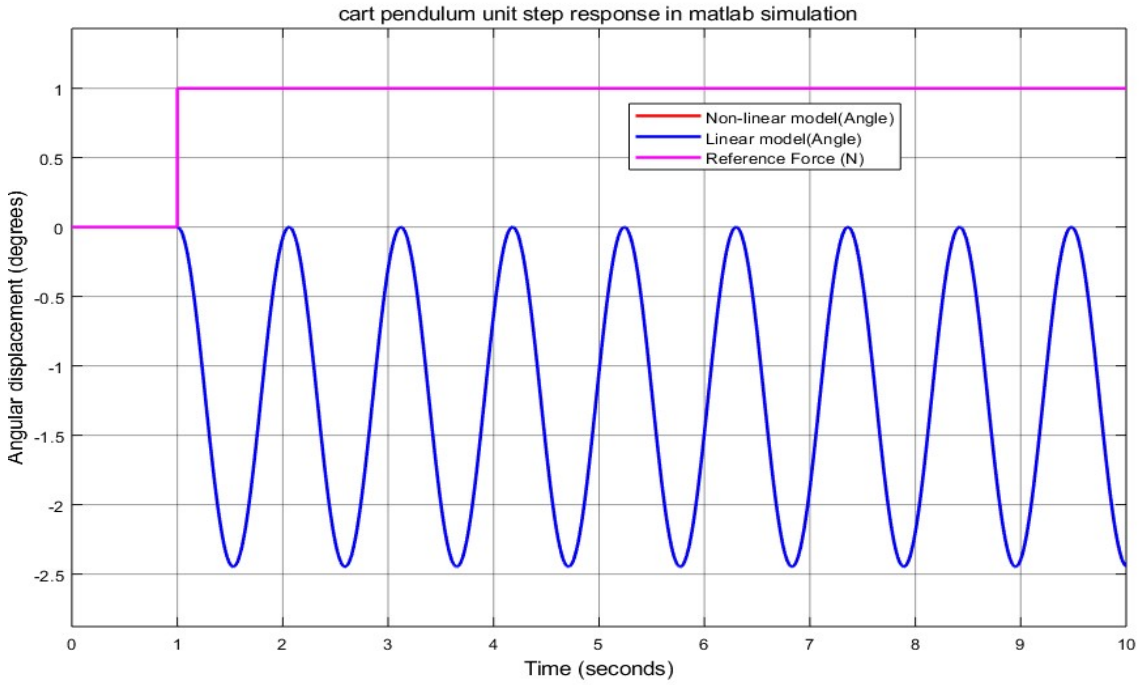


Figure 10: Pendulum angular displacement response of non-linear & linear models

The angle of the pendulum is shown in Figure 10. The range of the pendulum oscillations between 0 to -2.5 degrees, which is small. When the cart is accelerated, the mass of the pendulum tries to keep its current position due to inertia and if the cart is accelerated constantly, the pendulum cannot overcome the cart.

The angular velocity of the pendulum is shown in Figure 11. As the above graphs showed no unexpected events or behavior, the graphs of the angular velocity look like expected and indicate that the models are very close.

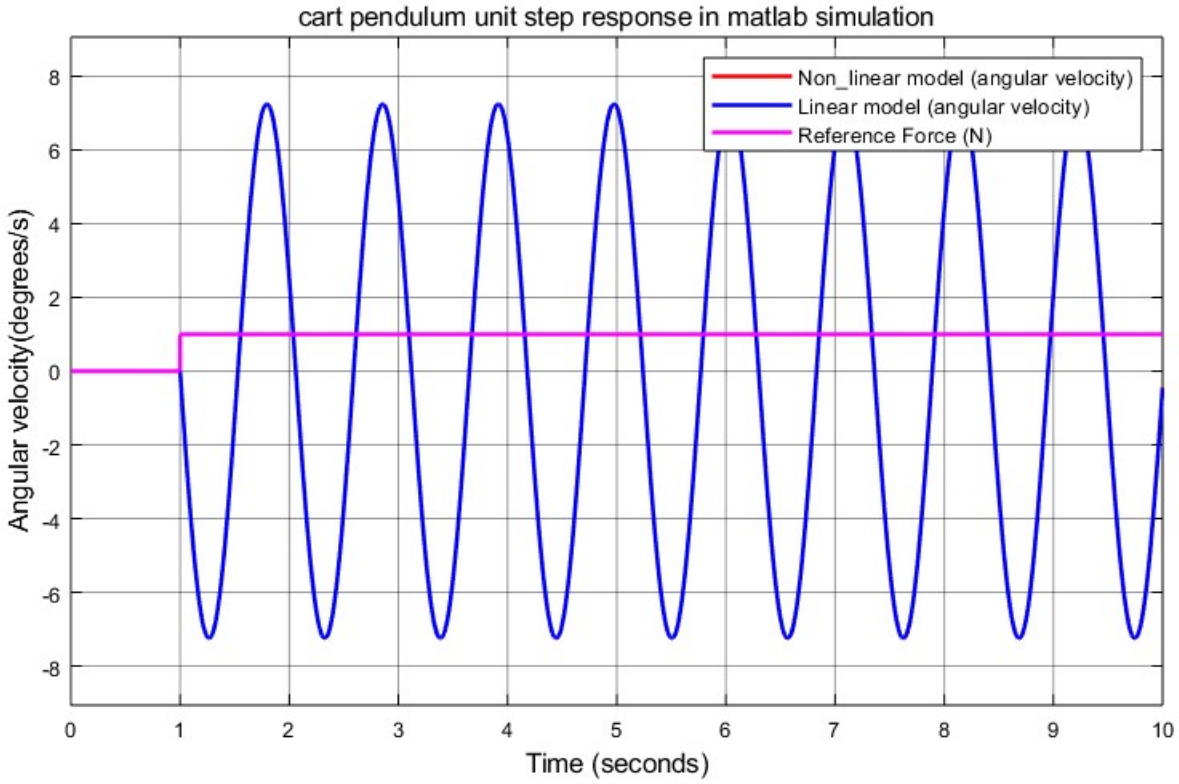


Figure 11: Pendulum angular velocity response of Non-Linear & Linear Models

The above output graphs clearly show that the system is highly unstable. Due to the absence of friction, a constant force gives a constant acceleration. This gives a constantly increasing velocity and therefore exponentially increasing displacement. Besides this, it is also confirmed from the poles of the open loop system or the Eigen values of the system matrix $'A'$ which are as follows:

$p1 = p2 = 0,$
$p3 = 0.0000 + 5.9271i,$
$p4 = 0.0000 - 5.9271i$

The two poles $p3, p4$ are on the imaginary axis (complex conjugate) and the rest of the two poles are on the origin which clearly suggests the instability of the system. Before going to controller design, one needs to make the system stable by using appropriate techniques.

4.2 Continuous time controller

The unstable behavior of the system (plant) is mitigated by implementing a controller. As the information of all states is easily available, the choice of designing a state-feedback controller using state-space methods is made.

4.2.1 State feedback controller

The basic structure of any practical system represented using state space model can be shown by the following figure12.

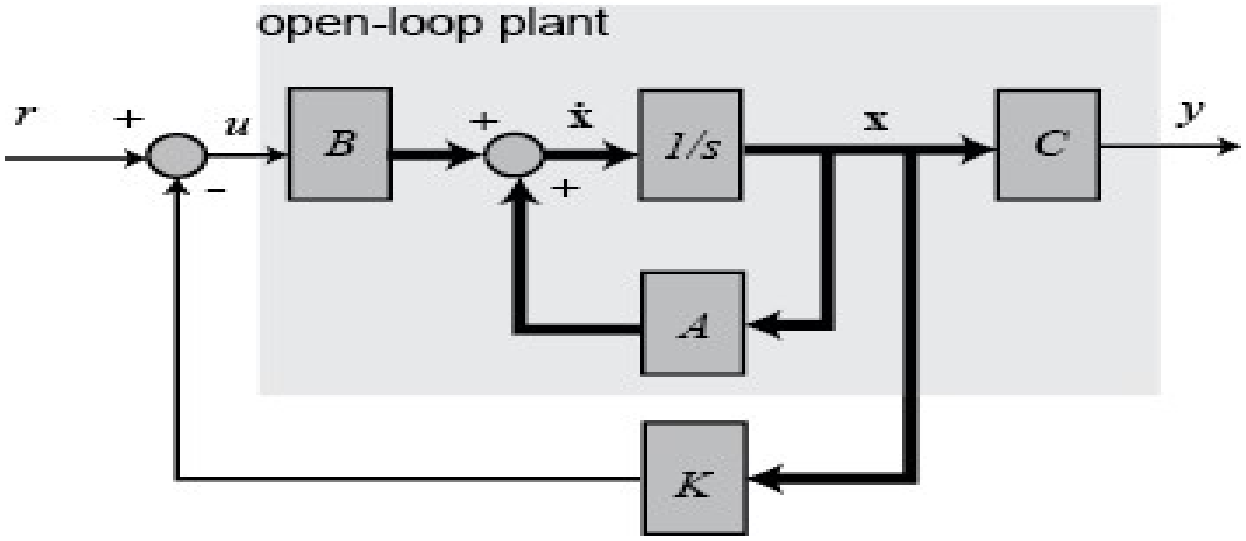


Figure 12: State space model block diagram with state feedback

For cart pendulum system the 'D' matrix is zero as system can not get directly influenced by the input. The A, B and C matrices represents our open loop system. $'r'$ is the reference input to the system and $'u'$ is the manipulating variable (force). The feedback gains for all the states is represented by the gain vector K. With the help of state feedback, the system becomes closed loop.

The state space equations for the closed loop system are

$$\dot{x} = (A - B \cdot K) \cdot x + B \cdot r$$
$$y = C \cdot x$$

This state feedback gain vector affects the poles of the system. If 'K' is chosen such that all the eigen values of the closed loop system '$A - B \cdot K'$ are negative, then the closed loop system becomes stable. The fundamental thing in state feedback is the free placement of poles. State feedback allows to place every single pole at any desired location in the left half of the s-plane while preserving the stability of the system. This method of placement of pole is called Ackerman's formula.

The Ackerman formula is given by

$$k^T = q^T \cdot (\alpha_0 \cdot I + \alpha_1 \cdot A + \cdots + \alpha_{n-1} \cdot A^{n-1} + A^n)$$

Where k^T = state feedback gain vector
q^T = last row of the inverse of controllability matrix
α = coefficients of the desired polynomial obtained by pole placement.
n = order of the system.

The figure 13 shows the state feedback in the linear cart pendulum system. All the poles are placed at (-4) as this pole placement gives the best response. A step input of 1m is given as reference position.

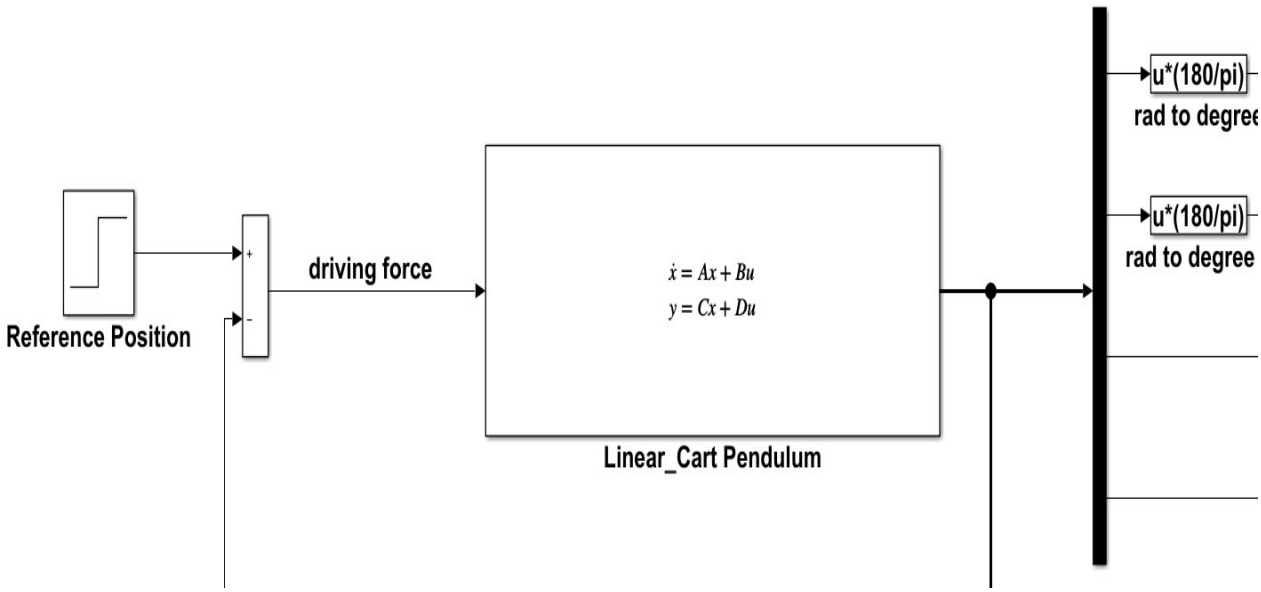


Figure 13: State feedback control without preamplifier

The output of the system is shown in the figure 14. The cart displacement gives the initial transient response and becomes stable at $0.03m$. This is not equal to the desired response of $1m$ as visible below:

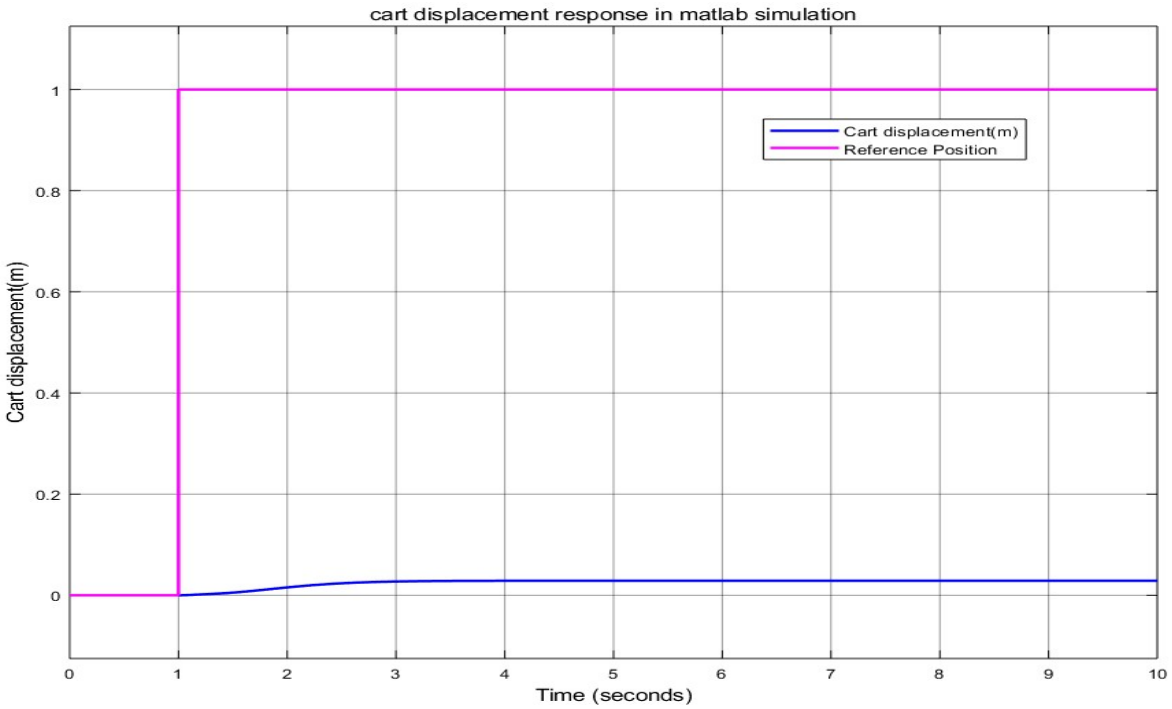


Figure 14: cart displacement with state feedback controller

The state feedback fulfills the requirements of the transient response, but it introduces a steady state

error. This error is calculated by applying the final value theorem for a step input which is given by

$$\lim_{s\to 0} s\cdot Y(s) = C\cdot[B\cdot k - A]^{-1}\cdot B + D$$

This emphasizes on the needs of using a pre-amplifier (p) so that this steady state error is nullified. This pre-amplifier can be calculated as follows.

$$p = \frac{1}{C\cdot[B\cdot k - A]^{-1}\cdot B + D} \qquad (16)$$

By using pre-amplifier, we can reach our final desired state. The introduction of 'p' changes the control structure of the system. This does not affect the closed loop poles or the Eigen values of the closed loop

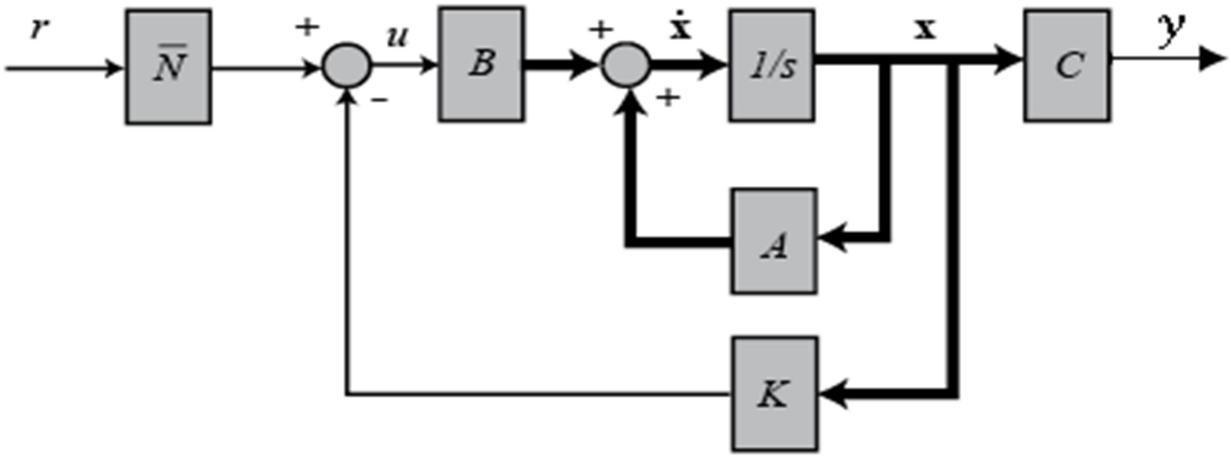


Figure 15: Block diagram of state feedback controller with preamplifier

system. The structure of the proportional controller is as follows:

The constant $\bar{N}$ = pre-amplifier whose value is obtained by using equation (16) as shown above. The below MATLAB script file shows the calculation of state feedback controller along with the preamplifier.

```
% State_feedabck using Ackerman's Formula %
S = [ B (A*B) ((A^2)*B) ((A^3)*B) ]; % controllability Matrix
R=rank(S);      % Rank of the controllability matrix
Sinv = inv(S); % Inverse of controllability Matrix
qt = Sinv(4,1:4); % last row of inverse of controllability matrix
alpha = poly([(-4) (-4) (-4) (-4)]); % the desired pole placement at (-4)
Palpha=((alpha(5)*eye(4))+(alpha(4)*A)+(alpha(3)*(A^2))+(alpha(2)*(A^3))+(A^4));
% Characteristic polynomial
k =(qt*Palpha); % Ackerman formula; 'k' state feedback gain vector
p=1/([0 0 0 1]*(((B*k)-A)^-1)*B); % preamplifier
kadp = k - p*C(4,1:4); % k_corrected due to output feedback
```

M-File 2: Matlab script for state feedback controller with preamplifier

The function $poly$ used in the script file is used to calculate the coefficients of the desired characteristic

polynomial. The output of the function is a row vector with the coefficient of the highest degree as the first element of the row vector. The figure 16 shows the state feedback model along with the pre-amplifier and the displacement of cart as output for reference position of 1 m shown in figure 17.

State feedback gain vector $k = [-11.4078 \quad -71.2036 \quad 34.7887 \quad 34.7887]$
Pre-amplifier gain $p = 34.7887$

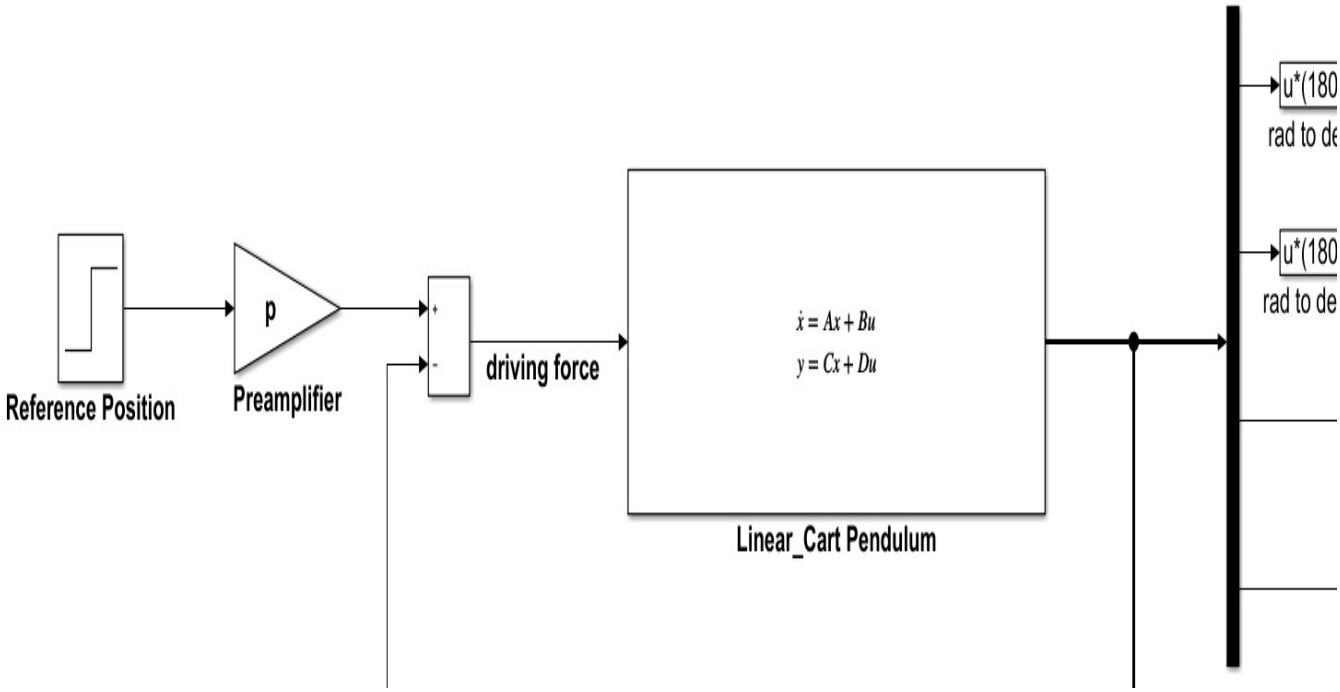


Figure 16: State feedback controller with preamplifier for linear model

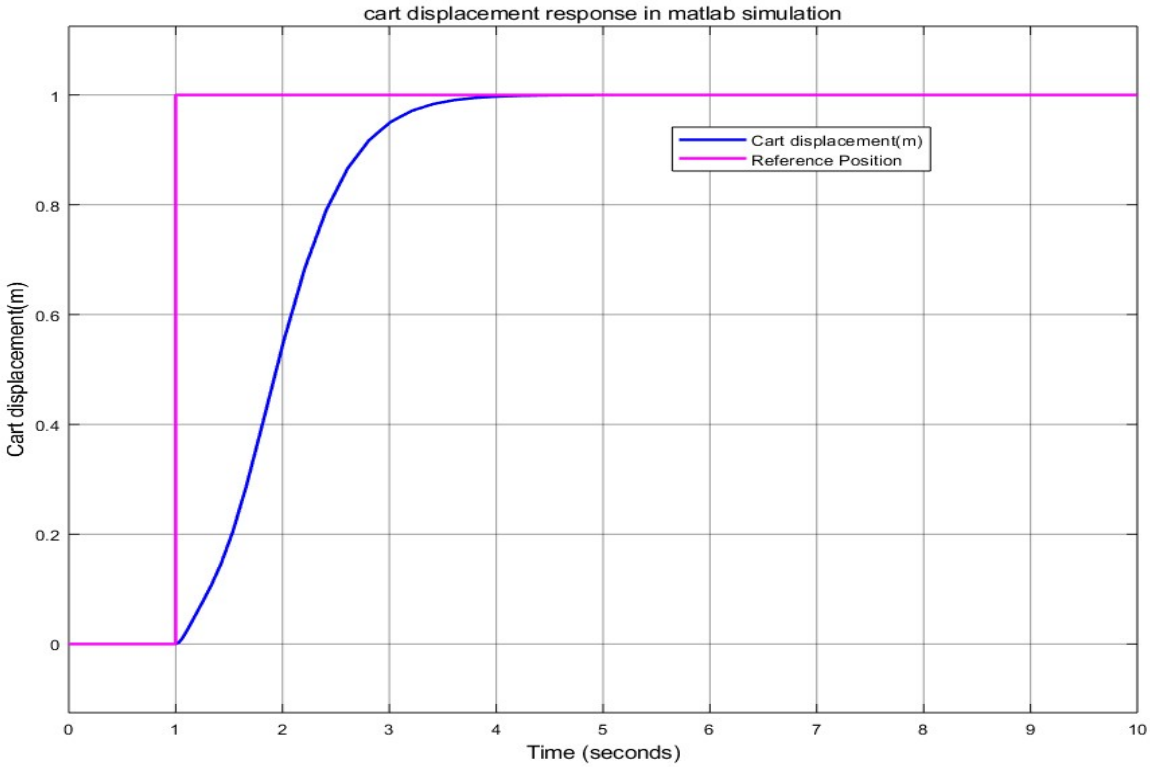


Figure 17: Cart displacement response of state feedback controller with pre-amplifier

4.2.2 Proportional (P) Controller

The state feedback is not transparent to any errors at the output. If there is any disturbance at the output the state feedback controller cannot anticipate and correct it. This induces the need of using output feedback.

The output feedback restores back the steady state accuracy even if there is any step disturbance at the output. The error obtained by calculating the difference between the reference value and the output value is amplified by the calculated proportional gain (p^*) and then given as input to the system. Because of the output feedback, the closed loop poles are affected. In order to compensate for this shift of poles, the state feedback gain vector is modified as follows

$$kadp = k^T - p \cdot C \qquad (17)$$
$$p^* = p$$

$kadp$ is the new state feedback gain vector. This is called P- controller or proportional controller. The last line of the Mfile-2 shows the implementation.

After output feedback adaption, the state feedback gain vector becomes

$$kadp = [-11.4078 \quad -71.2036 \quad 34.7887 \quad 0.0000]$$

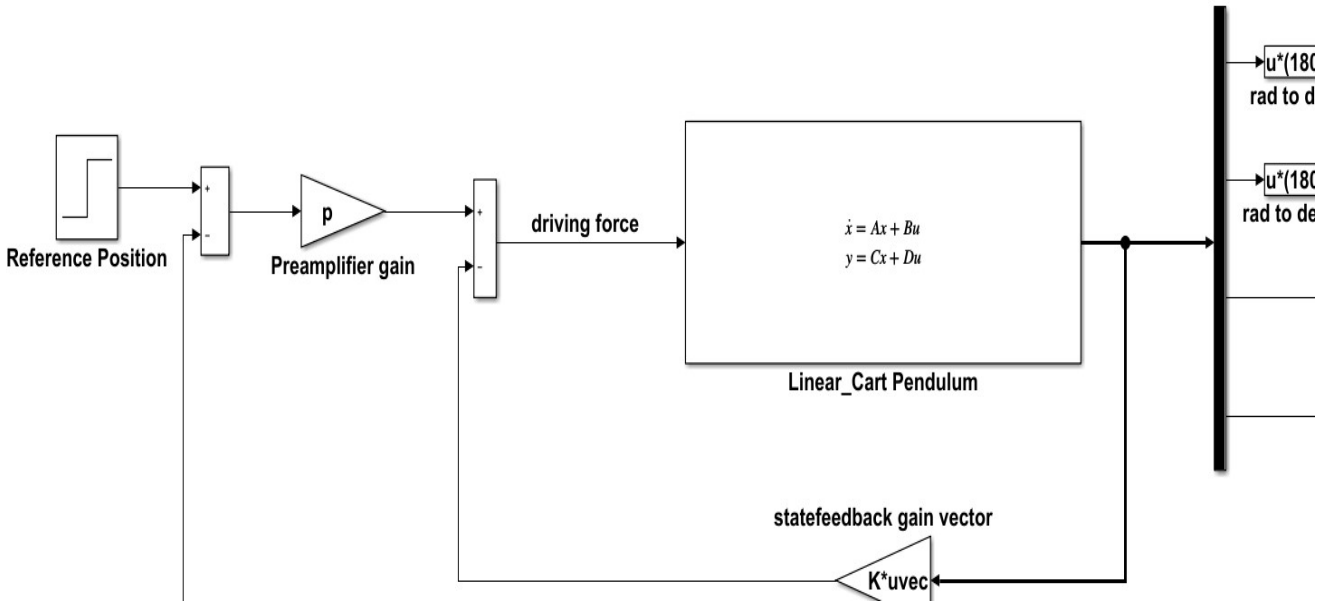


Figure 18: Simulink model for State feedback controller with output feedback

After output feedback adaption, the cart displacement is shown in figure 19. There is no difference in output response due to adaption in state feedback gain vector. But if some disturbance added into the system, the statefeedback controller and output feedback can not make the desired response as shown in figure 20. So, we need to design the controller to achieve the desired response even though disturbance added into the system, for that integral controller is needed.

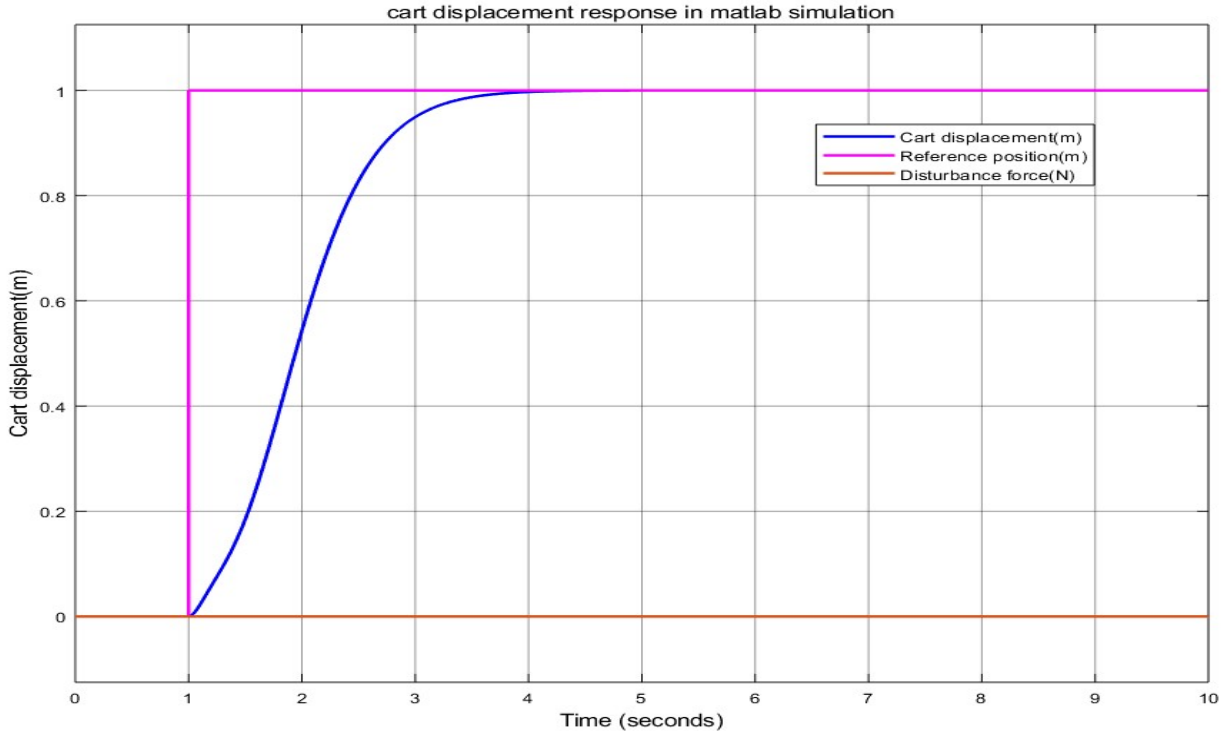


Figure 19: cart displacement for P controller without disturbance force

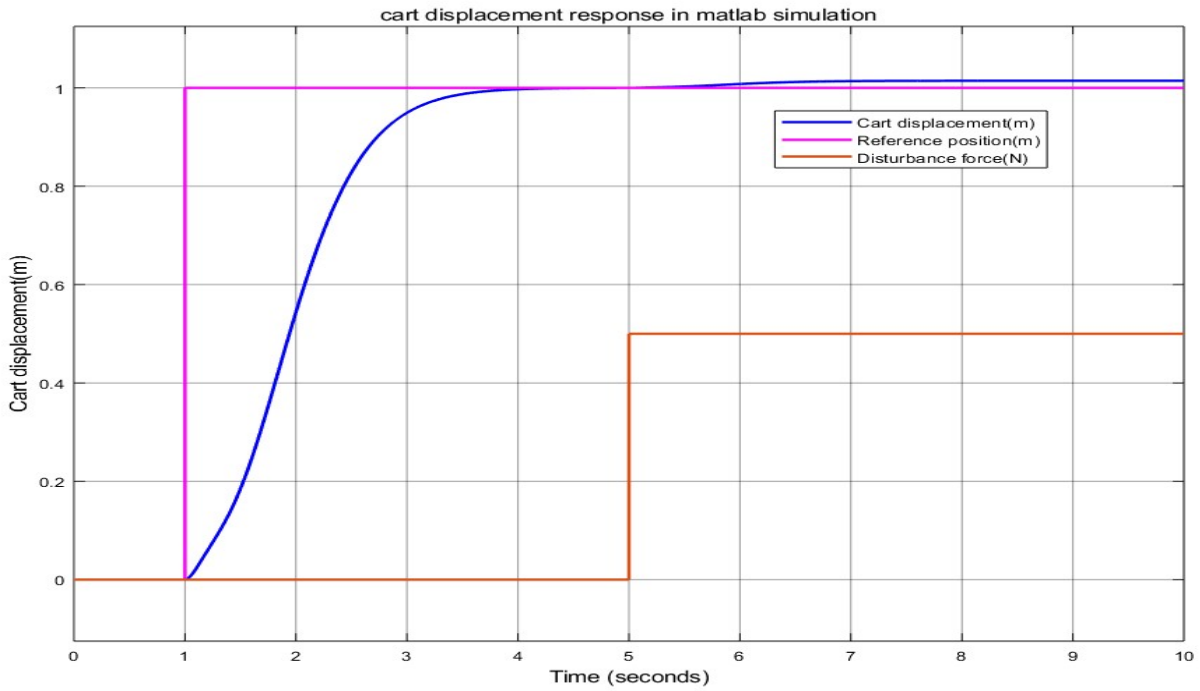


Figure 20: Cart displacement for P controller with disturbance force at t=5 sec

The state feedback controller tries to make the sway of pendulum to zero shown in figure 21, even though disturbance force added to the system at $t = 5\,sec$.

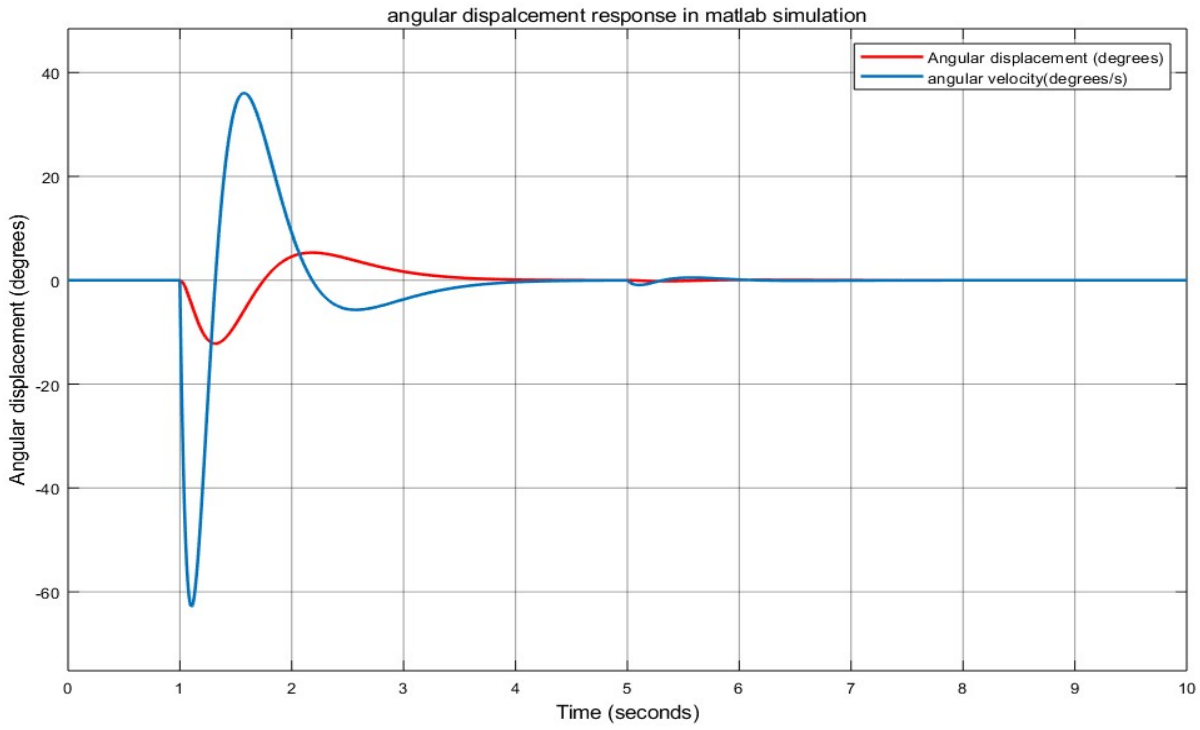


Figure 21: Angular displacement for P controller with disturbance force at t=5 sec

4.2.3 PI - Controller

If there are distortions within the system, then the system needs one more integrator before the input so that it can integrate the error caused by the disturbance and thus restore the steady state accuracy. The error now goes through the integrator before taking any action, the system response becomes comparatively slower. The complete model of such a PI-controller is shown in figure 22.

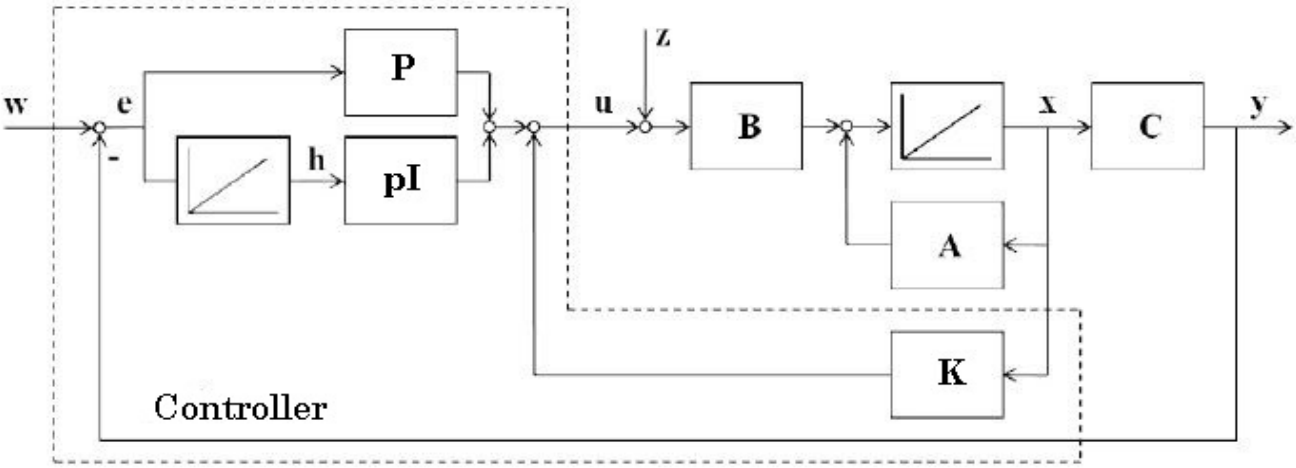


Figure 22: Block diagram of PI-Controller along with State feedback controller

The introduction of the one more integrator increases the overall order of the system. The new state space equation for the extended system is given by

$$\begin{bmatrix} \dot{\boldsymbol{x}} \\ \dot{h} \end{bmatrix} = \begin{bmatrix} A & 0 \\ -\boldsymbol{C} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ h \end{bmatrix} + \begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix} u$$

This results in the calculation of the pI gain with the modified state space matrices of the system. This modified matrix can be given as follows:

$$A' = \begin{bmatrix} A & \mathbf{0} \\ -\boldsymbol{C} & \mathbf{0} \end{bmatrix}$$

$$\boldsymbol{x}' = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{h} \end{bmatrix}$$

$$\boldsymbol{B}' = \begin{bmatrix} \boldsymbol{B} \\ \mathbf{0} \end{bmatrix}$$

$$\boldsymbol{C}' = [\, \boldsymbol{C} \quad \mathbf{0}]$$

With these above extended matrices, the pI gain can be calculated again using the same Ackerman formula but now with increased system order. This gives the new gain matrix where the last element is the $pIgain\ (ki)$. The MATLAB script used for the calculation for the PI-gain is shown in M-file 3.

```
% Adding an Integrator %
```

```
A1=[A [0;0;0;0] ; -C(4,1:4)  0];
B1= [B ; 0];
C1= [0 0 0 1 0];
D1=[0; 0; 0; 0; 0];
S1 = [B1 (A1*B1) ((A1^2)*B1) ((A1^3)*B1) ((A1^4)*B1)];%controllability Matrix
R1=rank(S1); % Rank of controllability matrix
S1inv = inv(S1); % Inverse of controllability Matrix
qt1 = S1inv(5,1:5); % last row of inverse of controllability matrix
alpha1 = poly([(-4) (-4) (-4) (-4) (-4)]); % the desired polynomial
Palpha1        =        ((alpha1(6)*eye(5))        +        (alpha1(5)*A1)        +
(alpha1(4)*(A1^2))+(alpha1(3)*(A1^3)) + (alpha1(2)*(A1^4)) + (A1^5));
k1 = (qt1*Palpha1);% Ackerman formula ;
ki = -k1(5); % the fifth element is the integeral gain
Ctk=[0 0 0 1];
kt_pi= k1(1:4)-p*Ctk; % Kt_Pi adaption due to output feedback
```

M-File 3: Matlab script for PI controller calculation

The complete model of the continuous time PI- controller along with state feedback on the non-Linear cart pendulum (CP) system as simulated in MATLAB is shown in figure 23.

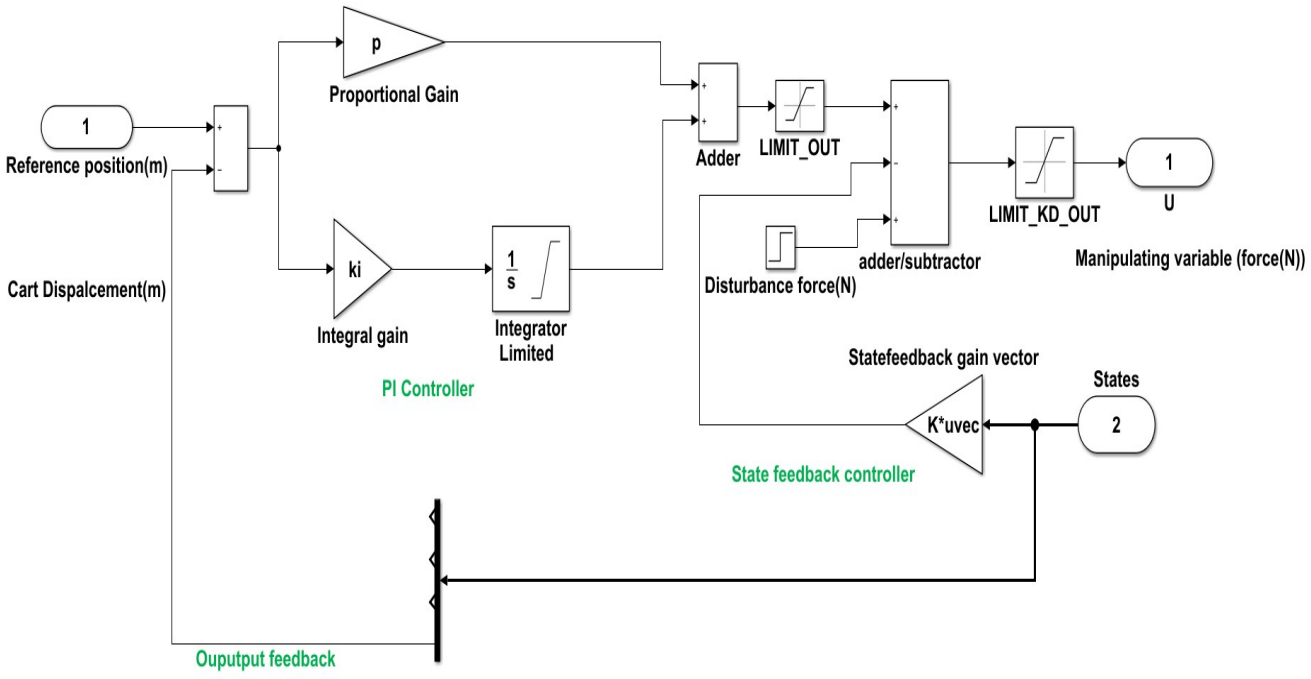


Figure 23: Block diagram of CP with PI Controller along with state feedback controller

The $States$ of the system are demultiplexed by the $Demux$. The cart displacement which is needed for the output feedback is then selected from $Demux$. After adding the PI controller in to the model, the gains are changed as follows,

State feedback gain vector $kt\_pi = [-3.0188 \ -116.8347 \ \ 82.9277 \ \ 139.1549]$

Proportional gain $p = 34.7887$

Integral gain $ki = 139.1549$

So far, we verified all the controller actions on linear system. Now, PI Controller and state feedback controller will verify on non- linear cart pendulum system. In figure 24, graph shows that even disturbance force added into the system at $t = 5\ sec$, still cart displacement reaches steady state accuracy due to integrator.

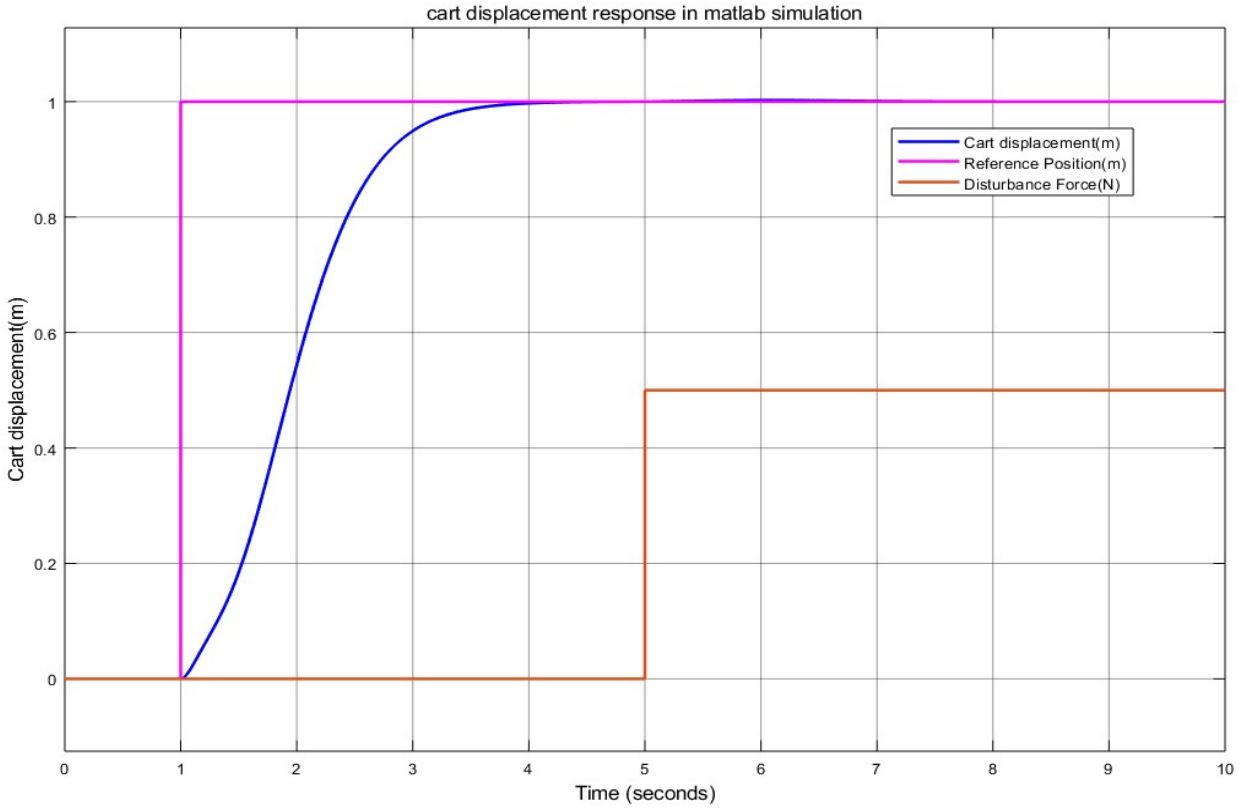


Figure 24: cart displacement for Non-Linear CP model with PI Controller & state feedback controller

In the figures 25, 26 & 27, all the states and manipulating variable are verified and all the states of the system become stable at the end and the cart displacement also reaches the desired steady state accuracy. Our objectives are position and sway control of cart pendulum system and those are achieved with both combination of PI controller and state feedback controller.

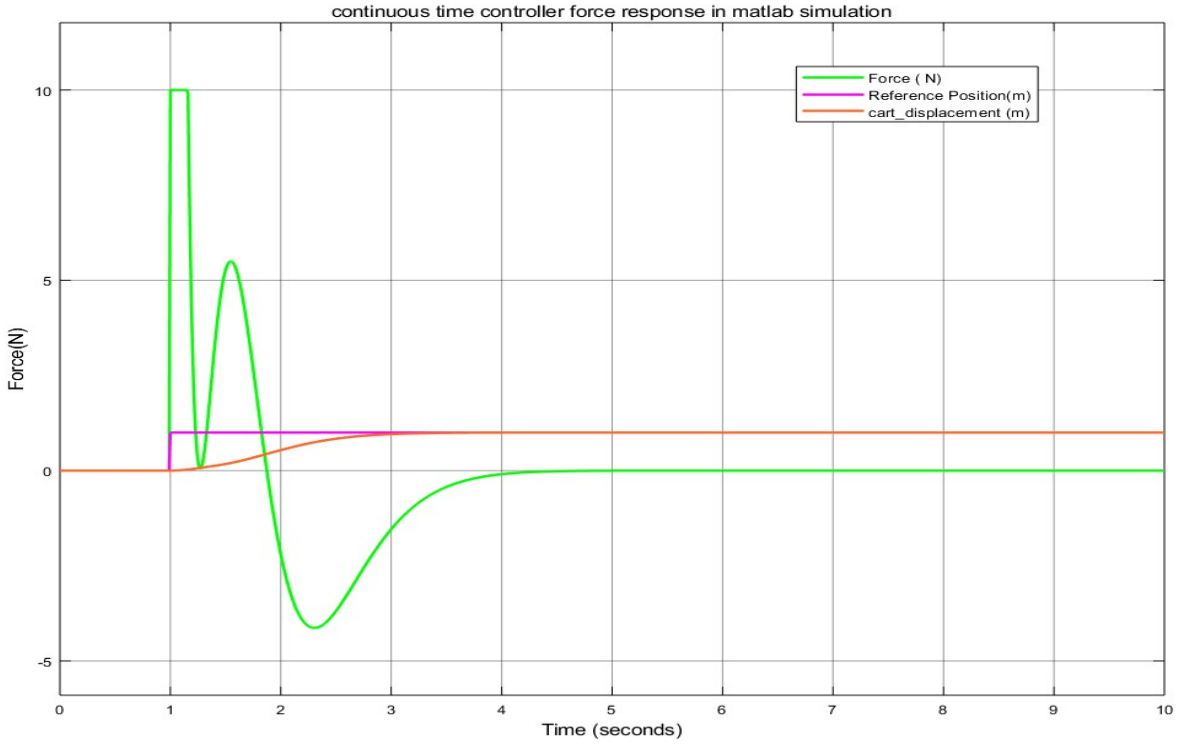


Figure 25: Manipulating variable (Force) response for Non-Linear CP model

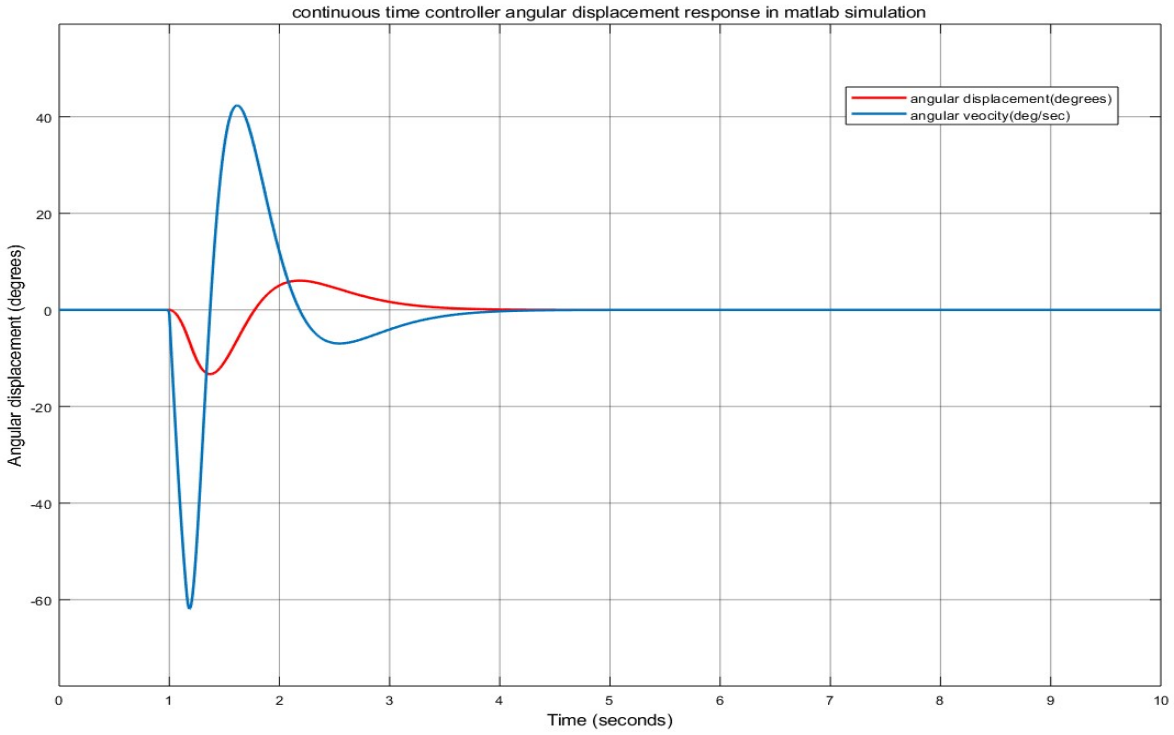


Figure 26: Pendulum angle & Angular velocity responses for Non-linear CP model

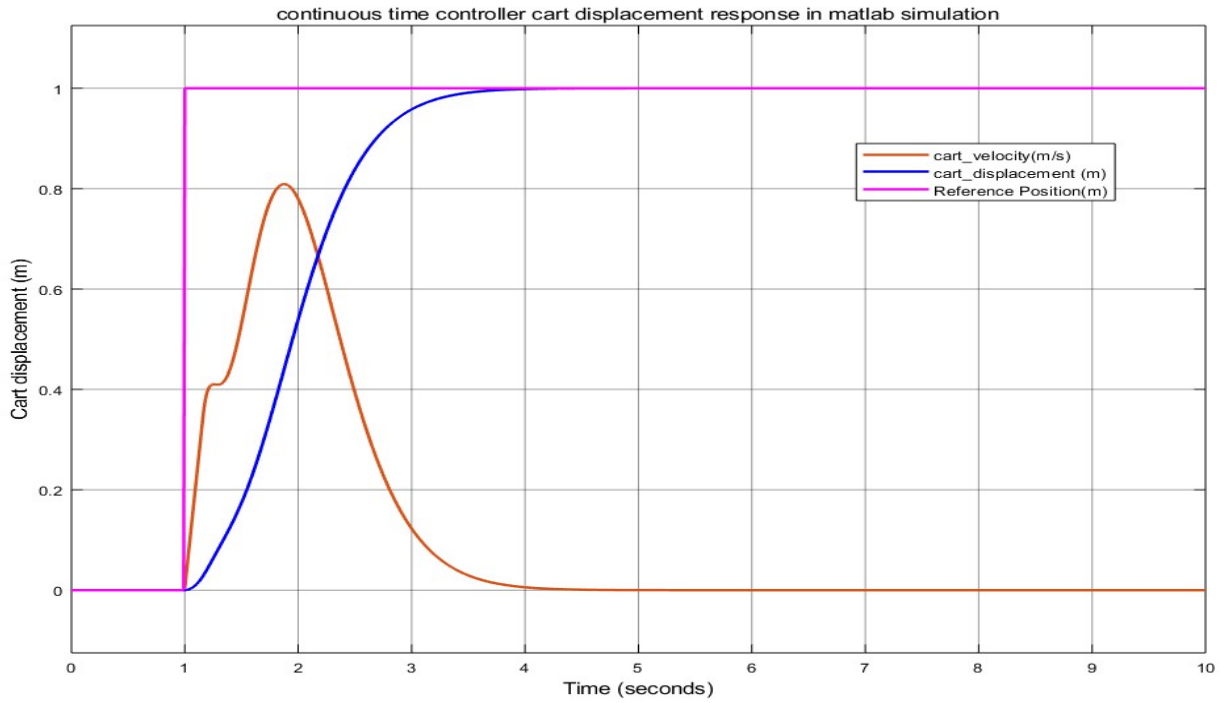


Figure 27: cart displacement & velocity responses for non-linear CP model

4.3 Discrete Time Controller

The controller designed in continuous time is not applicable for implementation in digital systems because digital systems are not in continuous. They are based on sampling time. Hence, if the controller is to be realized into any digital device then the controller must be converted to its discrete form. This is achieved by converting the continuous time system into the discrete form using 'Zero Order Hold' and a suitable sampling time. All the discrete time gains can be calculated by using the same Ackerman formula. The following MATLAB script converts our designed controller in continuous time to its equivalent discrete form. This also requires converting the continuous plant to its discrete form.

```
% Discrete_Time_model implementation %
sys=ss(A1,B1,C1,0);
Ts = 10e-3 ;    % sampling time 10ms (f =100Hz)
sysd=c2d(sys,Ts,'zoh');
[Ad,Bd,Cd,Dd]=ssdata(sysd);
% statefeedback gain vector & integral gain calculation
Spi_d=[Bd Ad*Bd Ad^2*Bd Ad^3*Bd Ad^4*Bd ];
Sinv_d=inv(Spi_d);
qpi_d = Sinv_d(5,1:5);
sys_pi=tf(1,poly([-4,-4,-4,-4,-4]));% desired polynomial
desys_pi=c2d(sys_pi,Ts);  %converting tf to discrete
[num_disc, den_disc]= tfdata(desys_pi,'v');
```

```
%controller statefeedback gain calculation
kt_d=qpi_d*(den_disc(6)*eye(5)+den_disc(5)*Ad+den_disc(4)*Ad^2+den_disc(3)*Ad^3+den_d
isc(2)*Ad^4+Ad^5);
ki_dpi = - kt_d(5);
ki_d =-kt_d(5)*Ts;   % the fifth element is the integeral gain
kt_dg =kt_d(1:4)-p*Ctk; % Kt_dg adaption due to output feedback
```

M-File 4: Matlab script for discrete time controller calculations

The sampling time of the discrete system is $Ts = 10$ ms. As the system is discrete it is also required to convert our desired pole in s − domain to its equivalent poles in z − domain. Since the sampling time is small enough there is not significant difference between the gain values in continuous time and discrete time. The continuous time integrator is converted to its equivalent discrete form as shown in figure 28. The gain values become,

State feedback gain vector $kt\_dg = [-3.2787 \quad -105.1724 \quad 75.8980 \quad 123.3515]$
Proportional gain $p = 34.7887$
Integral gain $ki\_dpi = 125.9915$

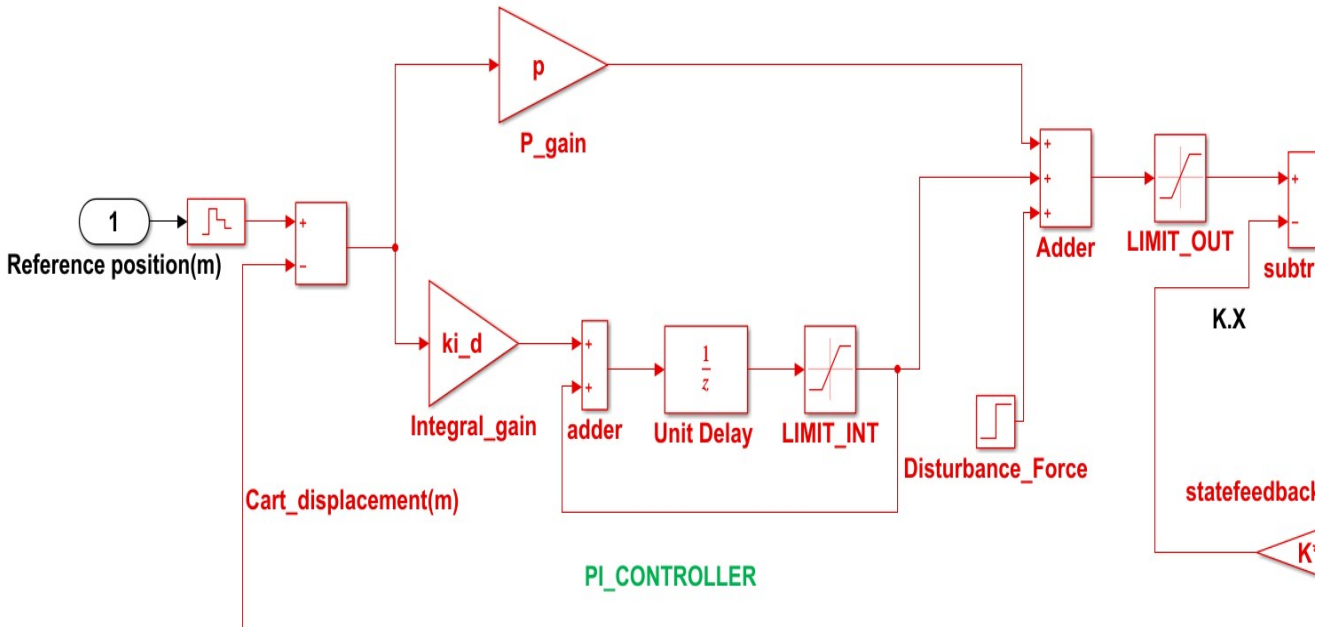


Figure 28: Simulink model for discrete time PI controller with state feedback

In the figures 29, 30 & 31, all the states and manipulating variable are verified with discrete time controller and all the states of the system become stable at the end and the cart displacement also reaches the desired steady state accuracy.

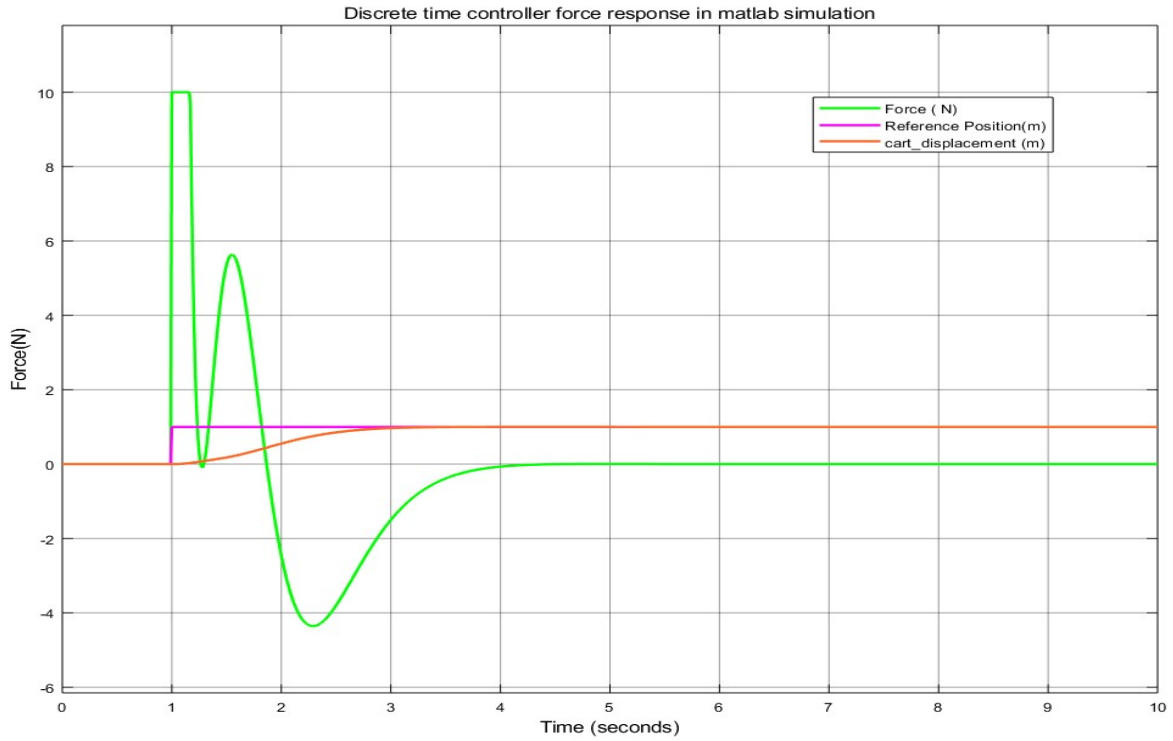


Figure 29: Manipulating variable (Force) response for discrete time controller

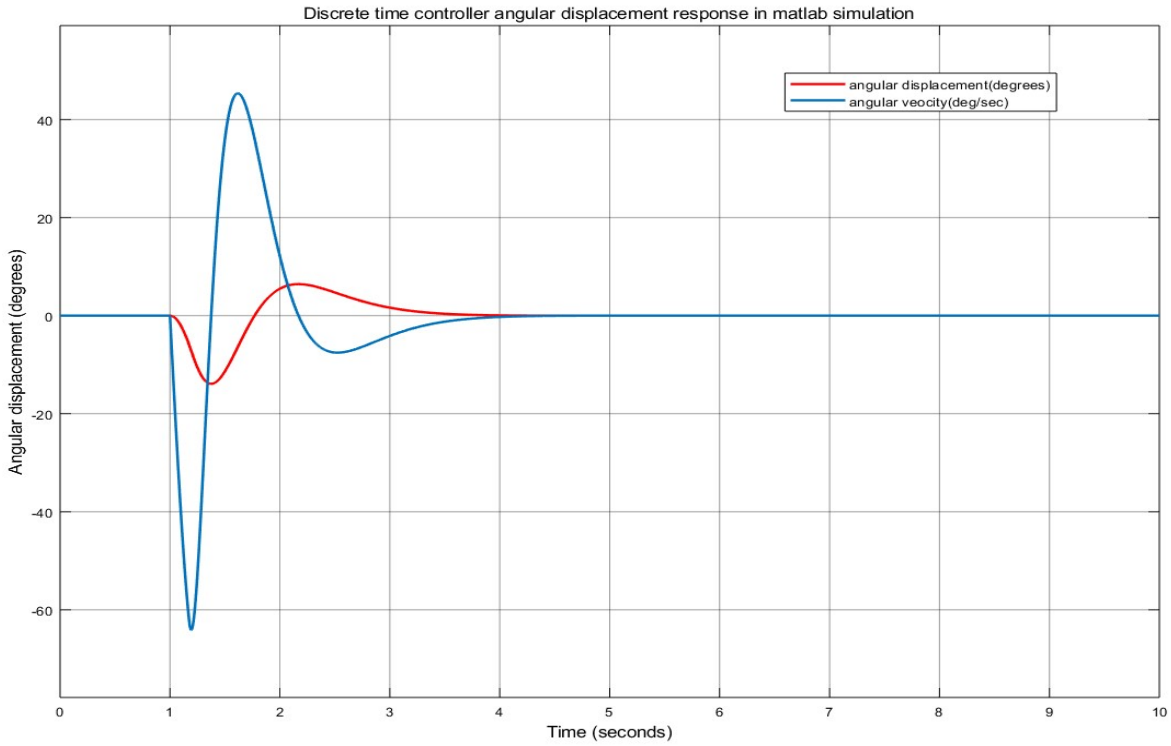


Figure 30: Pendulum angle & angular velocity response for discrete time controller

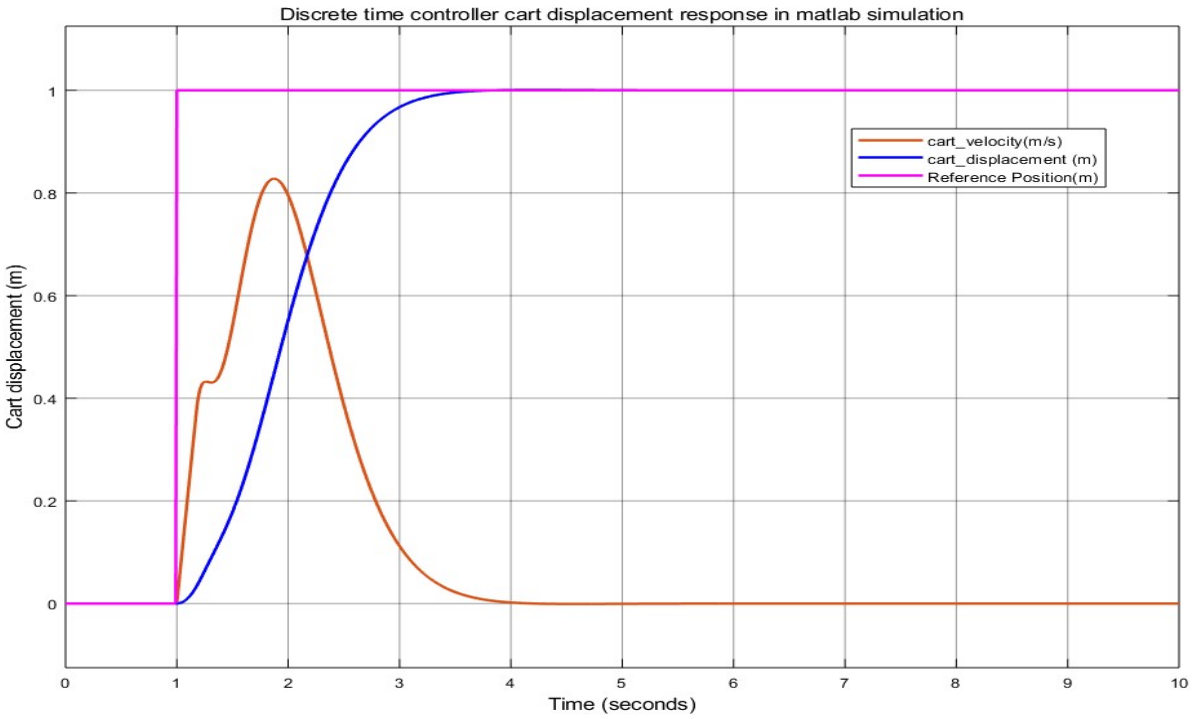


Figure 31: Cart displacement & velocity responses for discrete time controller

4.4 Fixed Point Controller

The calculations of the manipulating variable of the system in the continuous and the discrete time controller is done with double precision can be seen in the respective controller model. This calculation requires a lot of hardware and takes significant amount of time for processing in software. Hence, when it comes to practical implementation such systems fail to fulfill the real time requirement. This emphasizes on the need of a fixed-point model.

Moreover, the designed continuous controller and its equivalent discrete form contain calculation in physical standards of the states. But the real time system does not work on these quantities as the sensors used to get the information about the state of the system generally provide data in digital form such as the incremental encoders used in the setup of the cart pendulum system. These incremental encoders give output in integer format and this output is used as input to FPGA. Hence this integer format must be converted to its physical value to carry out the controller calculations. A better idea would be to include this proportional conversion factors in the respective gain values itself. This saves the time of the controller from conversion from integer to physical value and vice versa. Therefore, the fixed-point format for the gain values are decided after accounting for the sensor conversion factors. This type of controller model is called *sensor fixed point model* as it also accounts for the sensor conversion factors. The following MATLAB script file shows the sensor constants involved in reading all the states and the DAC conversion factor to convert counts into the physical force.

```
%% Sensor conversion factors from physical to integer

F_i = 22.0532;        % to convert DAC_values to voltage(1N = 'F_const'DAC counts)
F_0 = 1;

x_i_T = 57238;        % displacement of cart (1m = 'Disp_const' counts)
x_i = 28619;          % Overflow for x_i_A value in int_16.0,so x_i=x_i_T/2;
x_0 = 1;

vel_i = x_i*Ts         % velocity of cart per 10ms (Unit: m/10ms)
vel_0 = 1;

alpha_i = 4096;    % angular_displacment of pendulum (2*pi = 'Angle_const' counts)
alpha_0 = (2*pi);

omega_i = alpha_i*Ts ;  % angular_velocity per 10ms (Unit: rad/10ms)
omega_0     = 1;
```

M-File 5: Matlab script for sensor conversion factors from physical to integer format

The calculation of above factors is based upon the resolution of the sensor (for detailed information refer Appendix (I)). Even if the sensor is changed in the later state there is no impact on the model, only the above conversion factors needs to be altered. This makes the design process very easy. The actual resolution of the encoder mounted on the servo motor is $\mathbf{57238}$ counts for 1m distance. But the length of the belt is approximately $\mathbf{1.35}\boldsymbol{m}$ i.e. $\mathbf{67747}$ counts. This causes an overflow for a 16 - bit integer. Hence, the solution is to drop the last bit of the displacement. This essentially makes the distance half, which is representable by a 16-bit integer. Since the system is linear, the gains are doubled to compensate for this change.

This conversion factors are now included in the gains before converting it to the appropriate fixed-point format. The gains as calculated in the discrete controller model are used. With this design, the controller now accepts all the inputs only in the integer format. The PI controller has the reference displacement as input and the output of the controller is force. Therefore, the PI controller gains are scaled by displacement constant ($\boldsymbol{x\_i}$) and the Force constant ($\boldsymbol{F\_i}$). The state feedback controller has all states as input. This means the sensor conversion factors for all the states needs to be used. Also, the output of the state feedback controller is Force. Hence, the force scaling has to be accounted for calculating the fixed-point state feedback gain. This is evident in the M-file 6.

```
%% Accounting for Sensor conversion factors in Gain values
% State feedback gain vector
KD = [(F_i/omega_i) (F_i*alpha_0/alpha_i) F_i/vel_i  F_i/x_i].*(kt_dg)% int_16.12
KP_D = (x_0/x_i)* p *(F_i/F_0)  % discrete Proportional Gain ( int_16.15)
KI_D = (x_0/x_i)* ki_d *(F_i/F_0) % discrete Integral Gain   ( int_16.15)
%% Fractional bits
KD_SHIFT = 12;
```

```
KP_SHIFT = 15;
KI_SHIFT = 15;
%% Limiters
limit = 2750;
LIMIT = 10*(F_i/F_0);      % force limited to 10 N
LIMIT_INT = round(limit * 2^KI_SHIFT);     % for integrator output
LIMIT_OUT = round(limit * 2^KP_SHIFT);     % for PI controller output
LIMIT_KD_OUT = round(LIMIT * 2^KD_SHIFT); % for state feedback controller output
```

M-File 6: Gain and Limit calculations for Sensor Fixed point model

The actuation force in the practical world is always limited to certain value. Therefore, the output has to be restricted up to the maximum permissible limit of the controller. In the cart pendulum setup, the limit is imposed by the DAC output which is only in the range 0 - 4095 counts. But on test stand, the DAC driver program implemented to take the both positive and negative counts. This means ideally the upper cap on the force in both direction positive as well as negative is

$$F_{max} = \frac{2047}{F\_i} = 92.8209\ N.$$

As the output force does not become exactly zero at '2047', the positive and negative span of the physical force output is not the same. The output force is limited to $10\ N$. The integrator output is limited to '2750' because higher integrator gain may cause the real non-linear system to become unstable. After consideration of all the sensor conversion factors, the newly calculated gain values & fixed-point formats are listed in table 1. There is also a compromise on the precision of the values because of limited fractional digits.

S.No.	Parameters	Physical value	Fixed point format	Precision
1	Proportional gain (KP_D)	0.0268	$int\_16.15$	$3.051757813 \cdot 10^{-5}$
2	Integral gain (KI_D)	$9.7086 \cdot 10^{-4}$	$int\_16.15$	$3.051757813 \cdot 10^{-5}$
3	State feedback gain vector (KD)	$[-1.765 \quad -3.5579 \quad 5.8485 \quad 0.0951]$	$int\_16.12$	$2.44140625 \cdot 10^{-4}$

Table 1: Gain values for sensor fixed point controller

The controller is to be implemented on 32-bit *MicroBlaze* softcore processor which has all registers of

32-bit. Hence, the integer format is limited to 32-bit as it offers efficiency in carrying out mathematical operations. The figure 32 provides a subsystem view of the conversion of states from physical to fixed point format.

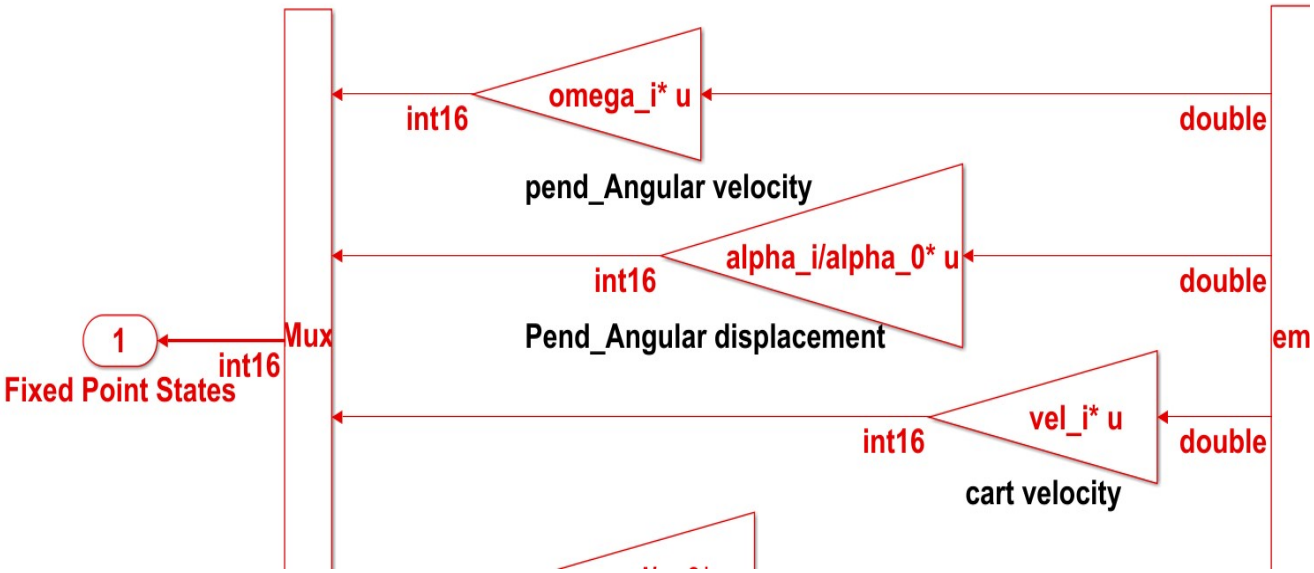


Figure 32: Subsystem for Physical states to fixed point states

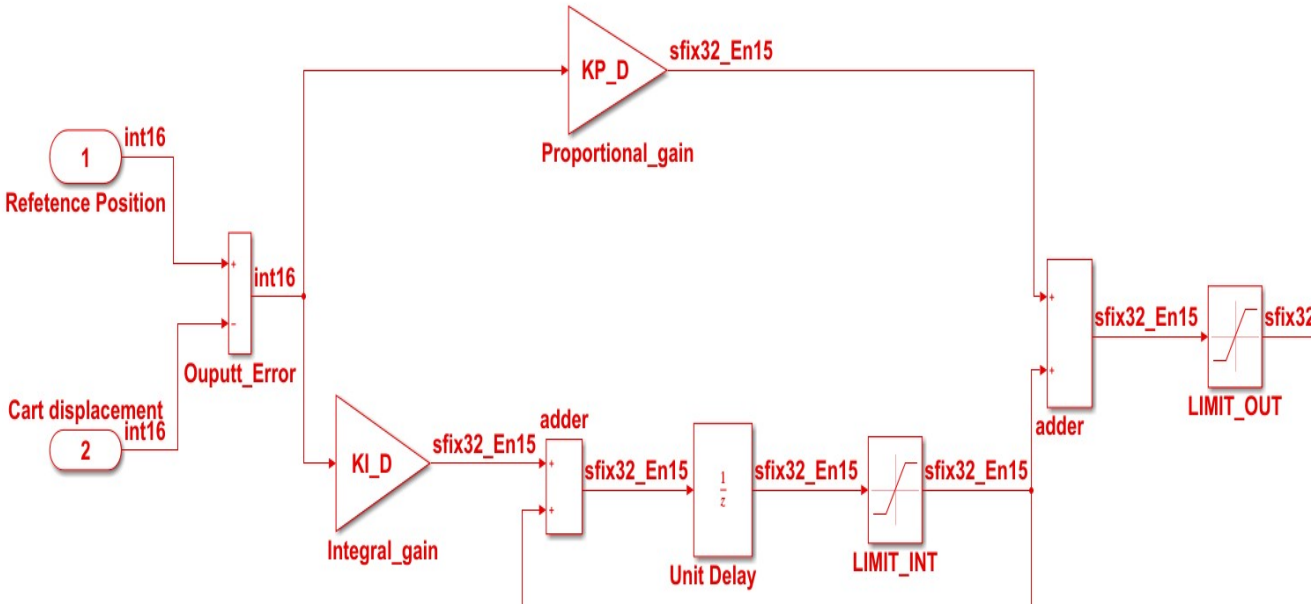


Figure 33: Fixed Point PI Controller

The down-conversion blocks are used to make the similar fractional bits to carry out the arithmetic operations. The output of the PI_ controller is also limited and down converted to match the fixed-point format of the state feedback controller is shown in figure 33. The complete fixed-point controller with the PI-controller and fixed-point states as the subsystem is shown in the figure 34.

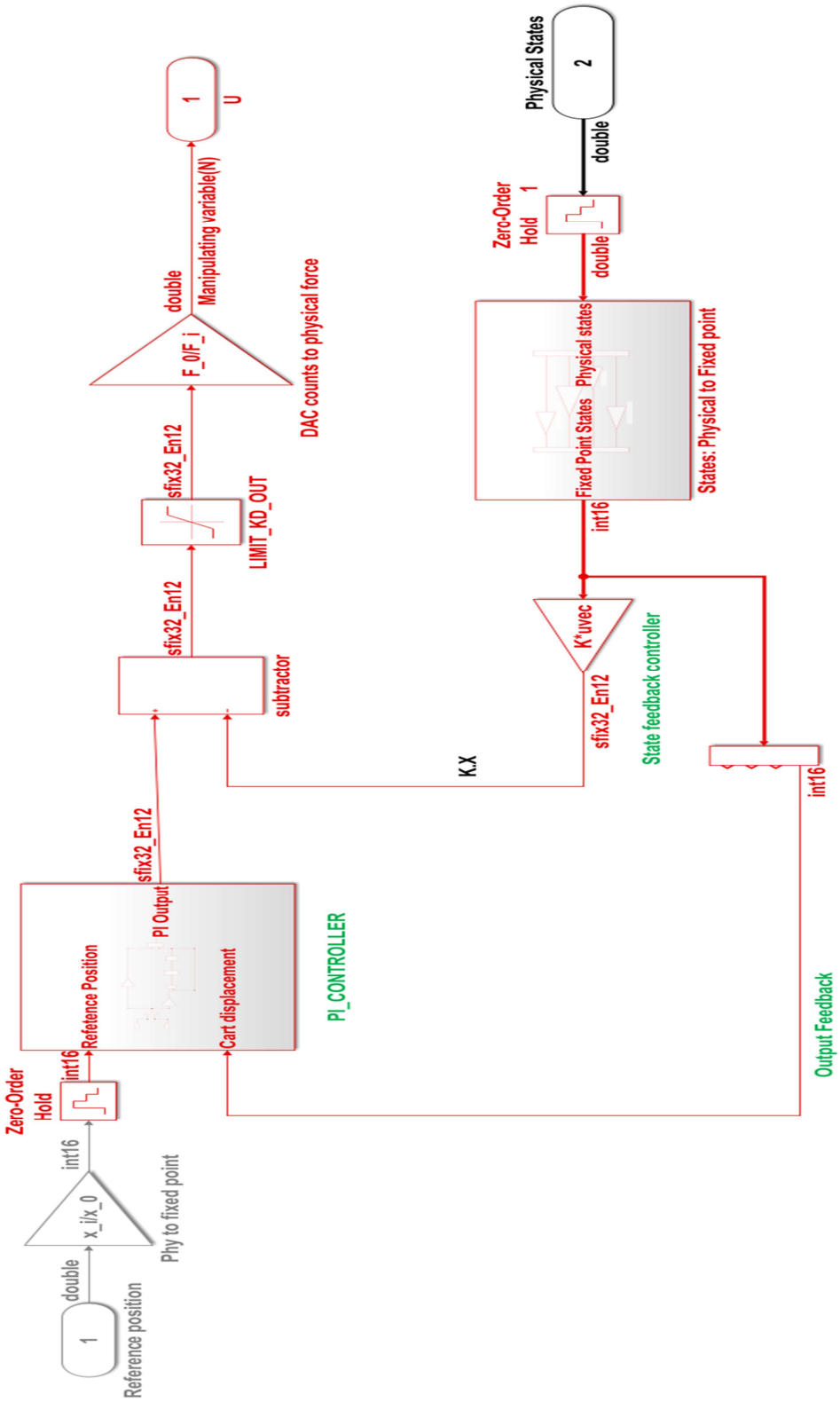


Figure 34 : Simulink model for Sensor Fixed Point Controller

In the figures 35, 36 & 37, all the states and manipulating variable are verified with fixed point controller and all the states of the system become stable at the end and the cart displacement also reaches the permissible steady state accuracy (0.02%). The manipulating variable (force) is oscillating at zero force (see figure 35) due to precision losses in fixed point conversions.

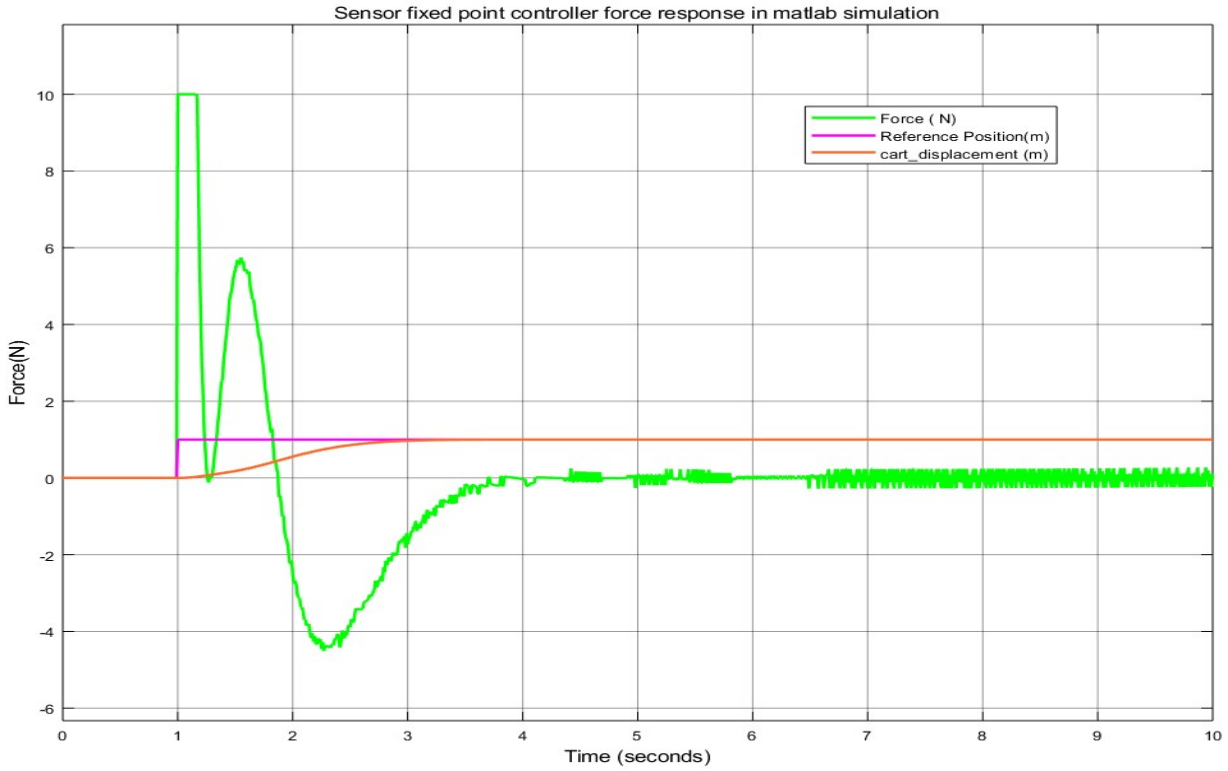


Figure 35: Manipulating variable (force) response for sensor fixed point controller

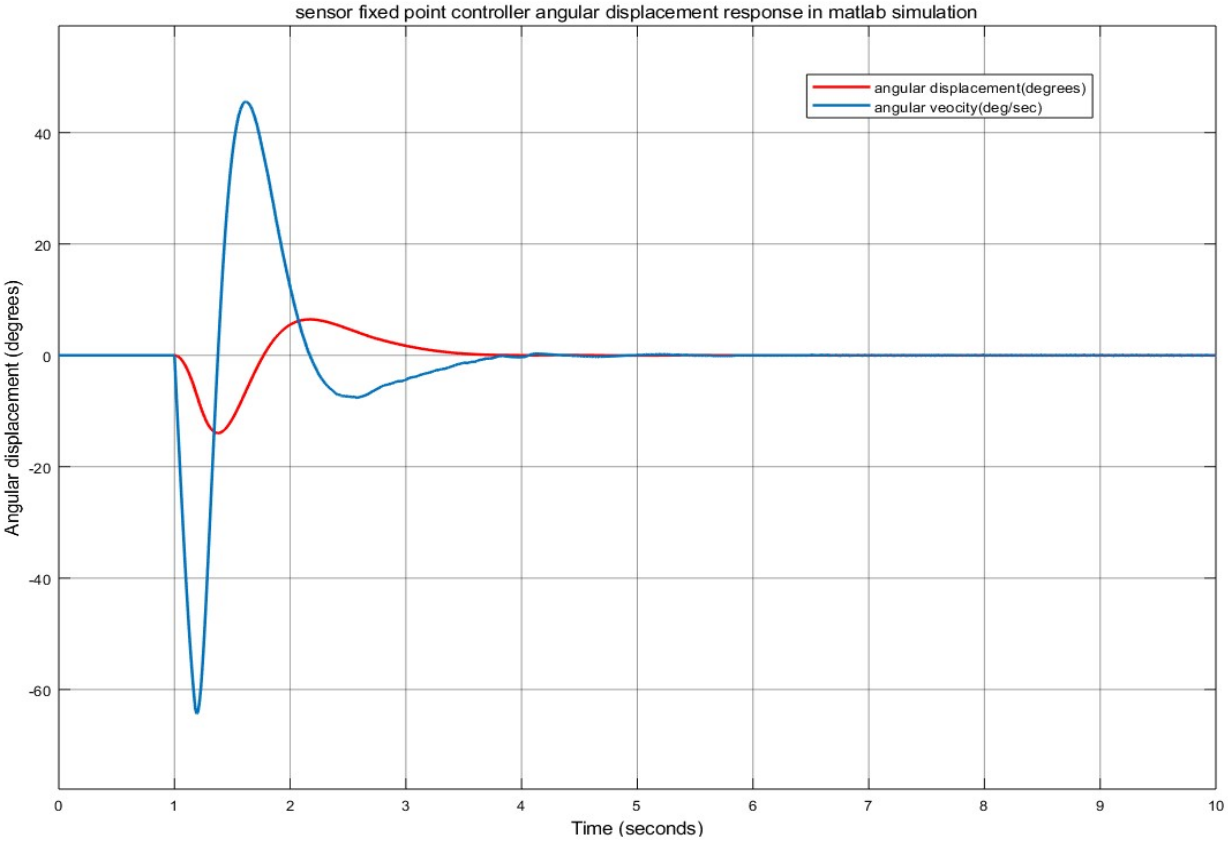


Figure 36: Angular velocity & angle responses for sensor fixed point Controller

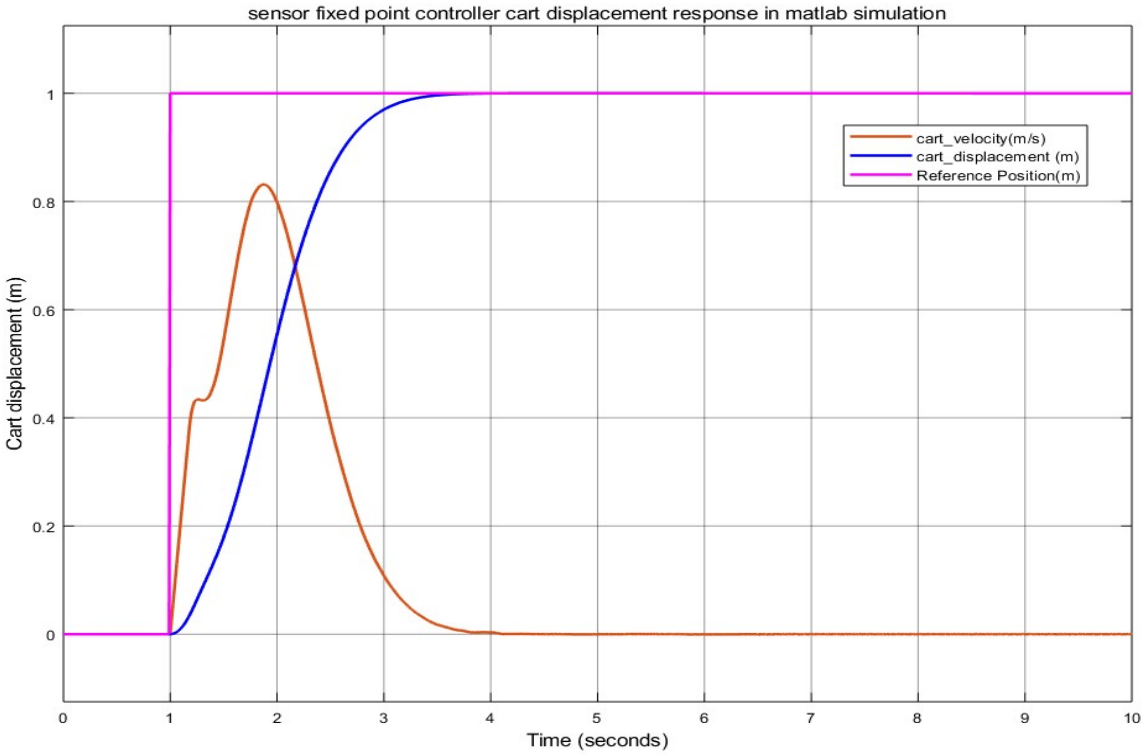


Figure 37: Cart displacement & velocity responses for Sensor fixed point controller

4.5 S- Function Development

The S-function (System Function) is used with which the developed C-code can be tested onto a model. It is basically used to test the overflow conditions & controller algorithms before implementing on hardware. It takes the source file and header file associated with it as input and generates a block of C-code as output which can be simulated with the model for testing and verification. This S-function is generated by the legacy code tool.

The purpose of the S-function is to call a simple legacy function during simulation. We must specify the S-function name in C file of S-function. In this project '*ex_sfun_pos_contr.c*' file is created for legacy function calls and to access input/ output parameters of the system. In MATLAB, sfunlegacy.m file creates legacy mex function for cart pendulum control. When this file ran, MATLAB reads the commands and executes them exactly as any controller/processor. The MATLAB script file used for the legacy code block generation is shown below:

```
% create legacy mex function for cart pendulum control
def = legacy_code('initialize');
def.SourceFiles = {'pos_contr.c','stfbk_control.c', 'pi_control.c'};
def.HeaderFiles = {'stfbk_control.h', 'xil_types.h', 'pi_control.h'};
def.SFunctionName = 'ex_sfun_pos_contr';
def.OutputFcnSpec = 'int16 y1 = pos_contr(int8 u1, int16 u2, int16 u3, int16 u4,
int16 u5, int16 u6)';
```

```
legacy_code('sfcn_cmex_generate', def)
legacy_code('compile', def)
legacy_code('slblock_generate', def)
disp('That`s all folks.')
```

M-File 7: Matlab script for S-block generation & controller run

When the above script is executed in MATLAB without any errors, it generates the S-function block (shown figure 38) which now behaves like any normal Simulink block.

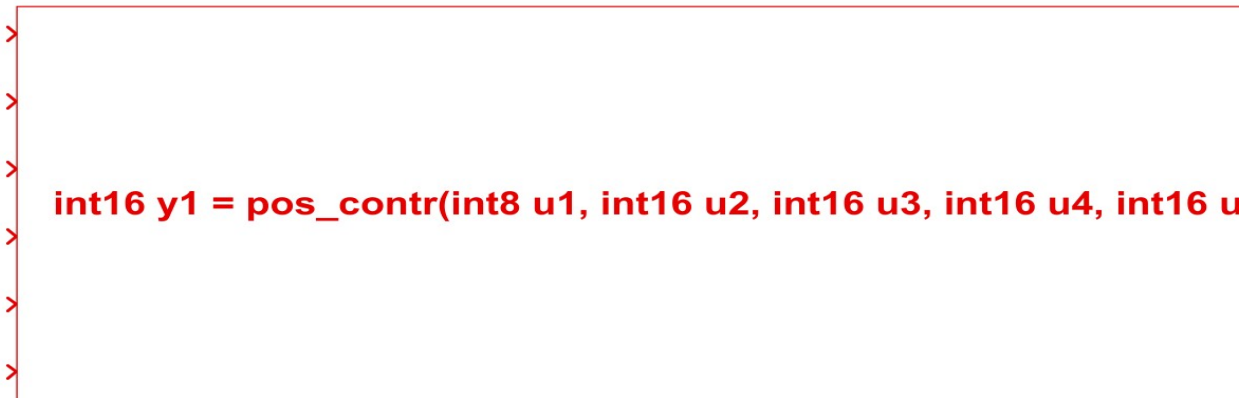


Figure 38: Simulink S-function Block

The '$y1$' is the output of the function which is the force in integer format. The function has six inputs out of which the last four are the states of the system, '$u1$' is reset signal and '$u2$' is the reference. The top-level function performed by this block can be explained by the C-file 1.

```
#include "xil_types.h"
#include "pi_control.h"
#include "stfbk_control.h"
#include "pos_contr.h"
s16 pos_contr(s8 reset, s16 ref_val, s16 state1, s16 state2, s16 state3, s16 state4)
{
      s16 uu=0;
      if (reset > 0 )
      {
            CpContrInit();   // Controller initialization with gain & limit values
            uu = 0;
      }
      else
      {
        uu = CpContrCalc(ref_val, state1, state2, state3, state4);
      }
      return uu ;
}
```

C-File 1: pos_contr.c file

This function takes the states and reference value as input from the Simulink and carries out the calculation of the PI-controller and state feedback controller in fixed point formats. Thus, all the inputs to the system must be converted to their fixed-point format before feeding as input to the block. The output similarly is also a fixed-point number which needs to be converted to its physical value before feeding it to the plant.

The other source files and header files are listed in detail in the Appendix (III). The complete CP legacy-code controller as a Simulink model is shown in figure 39.

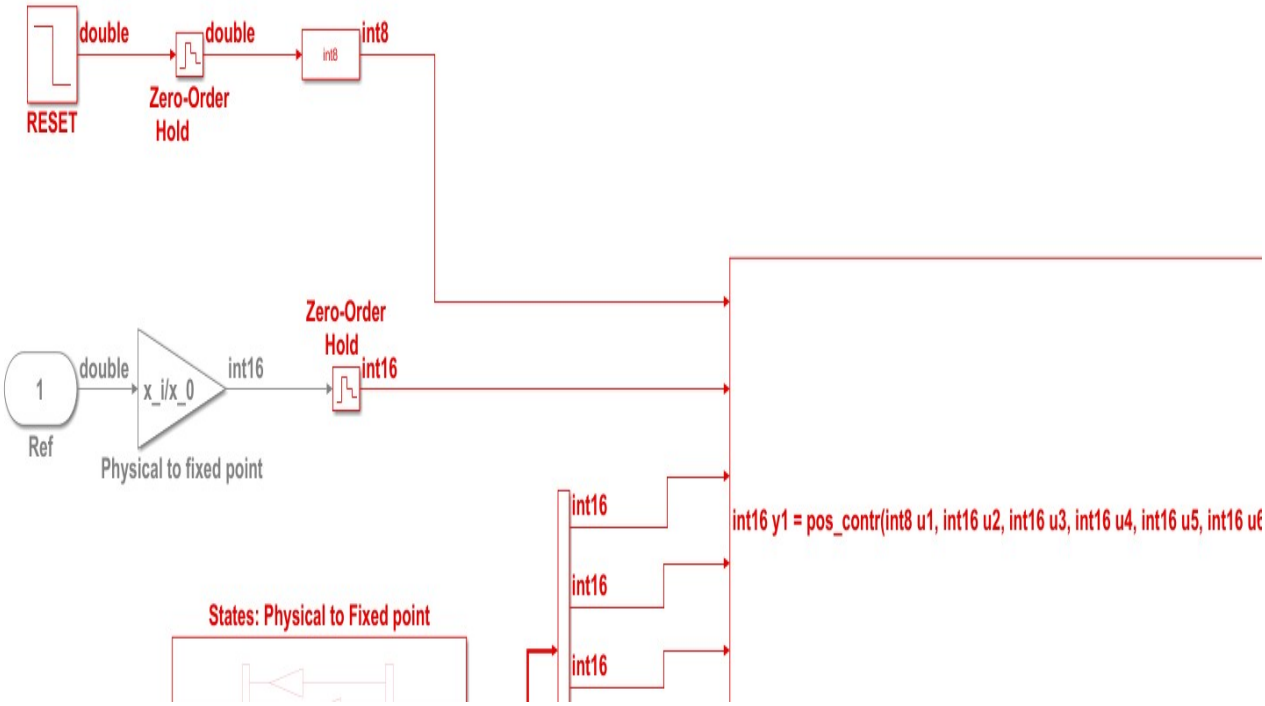


Figure 39: Simulink model for CP Legacy Code Controller

The gain 'F_0/F_i' at the end is the constant factor to convert the force in fixed point format to its equivalent physical value. The Zero order hold blocks are placed so that the working of the controller exactly as in the real time environment can be simulated and verified for its output. The final output using the legacy code will be seen in figures 40, 41 & 42.

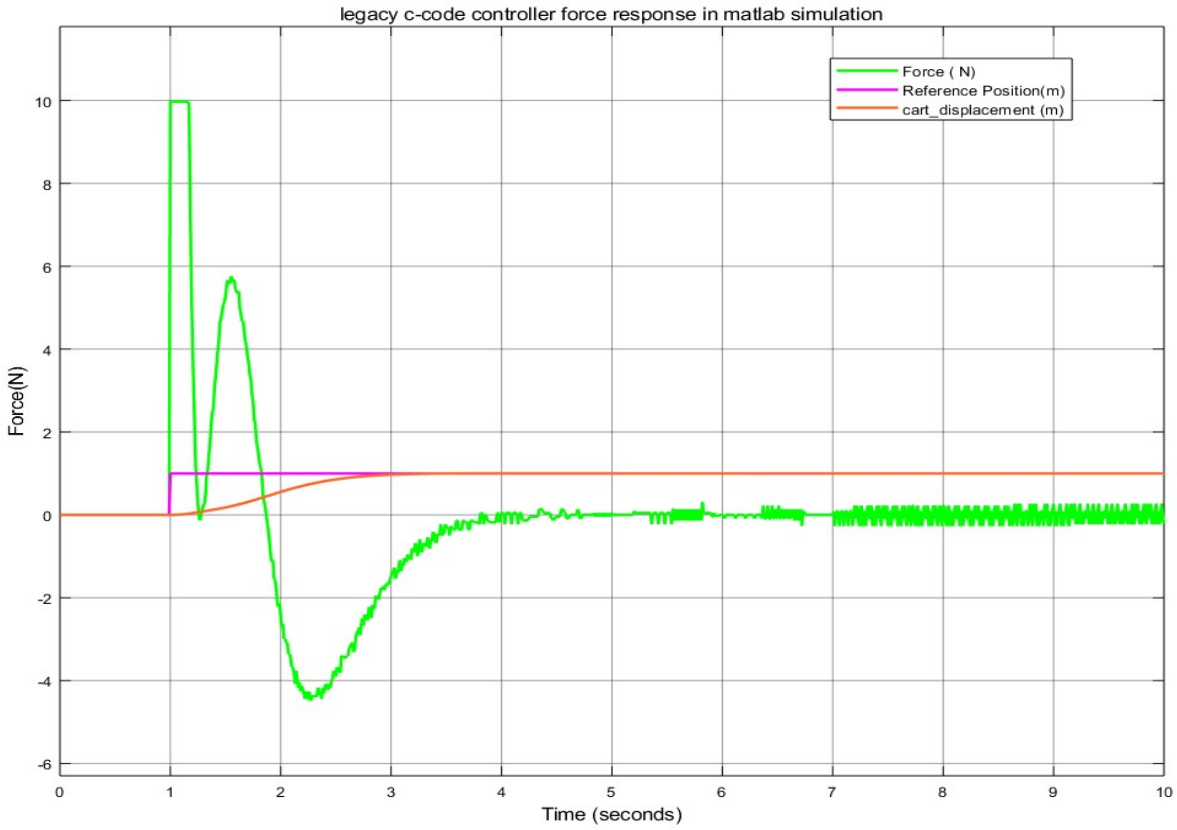


Figure 40: Force response with C-code in legacy tool

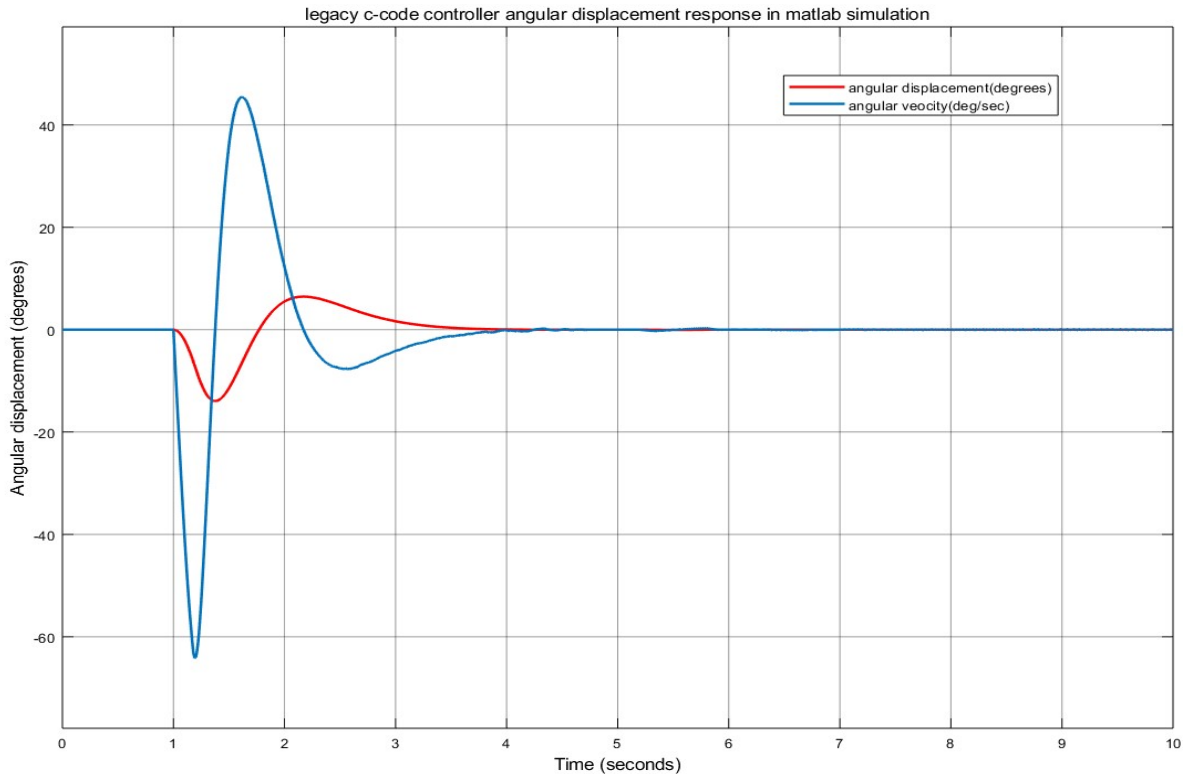


Figure 41: Angular velocity & Angle response with C-code in Legacy tool

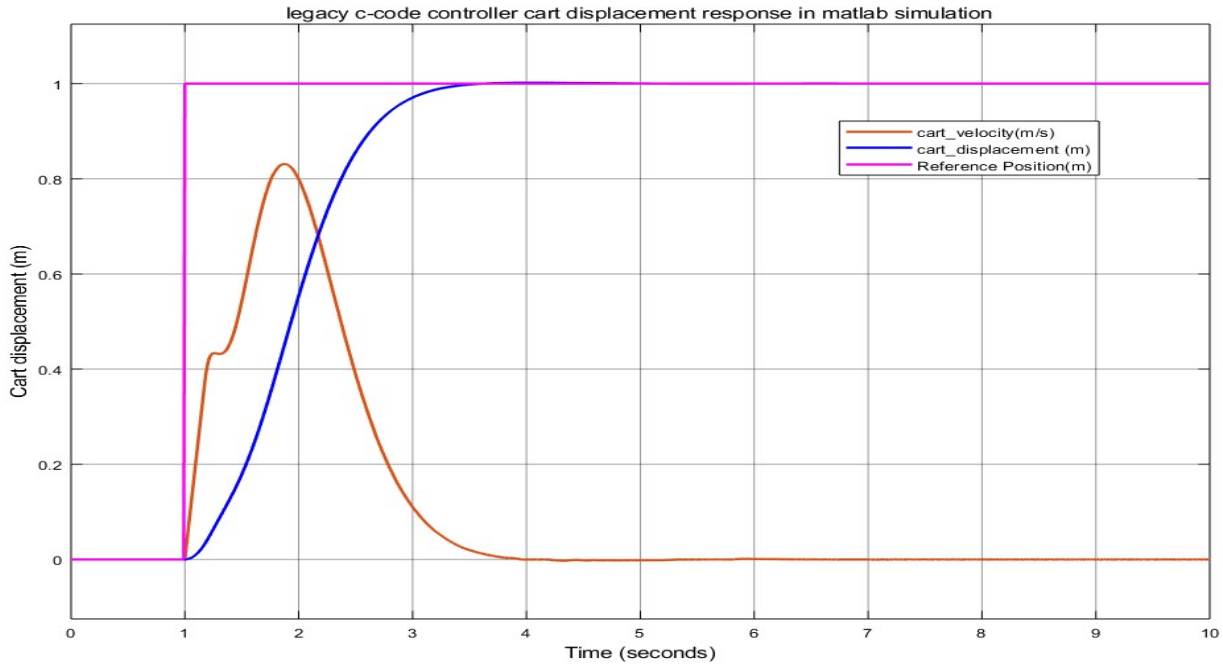


Figure 42: Cart displacement & velocity response with c-code in legacy tool

4.6 Comparison of Controllers

Although the performance of all controllers are acceptable, but their behavior is not the same. The objective here is to displace the cart to match the reference value as accurately as possible. The control of cart displacement by using different controllers on the non-linear cart pendulum system is compared in the figure 43 & 44.

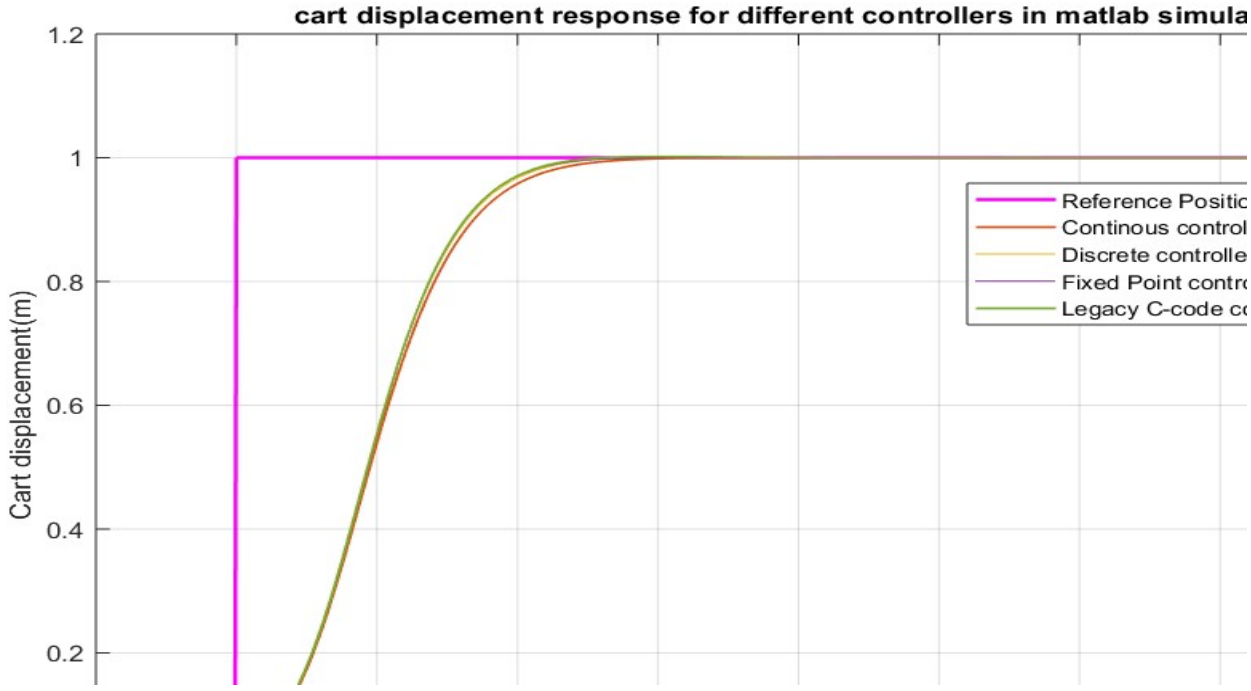


Figure 43: Cart displacement response with different controllers

The best response can be achieved as shown in figure 43, by the fixed-point controller and Legacy (C-code) controller. Both the controller responses match exactly and hence only the C-Code graph is visible

in the plot. The cart displacement reaches the reference position within 4 seconds.

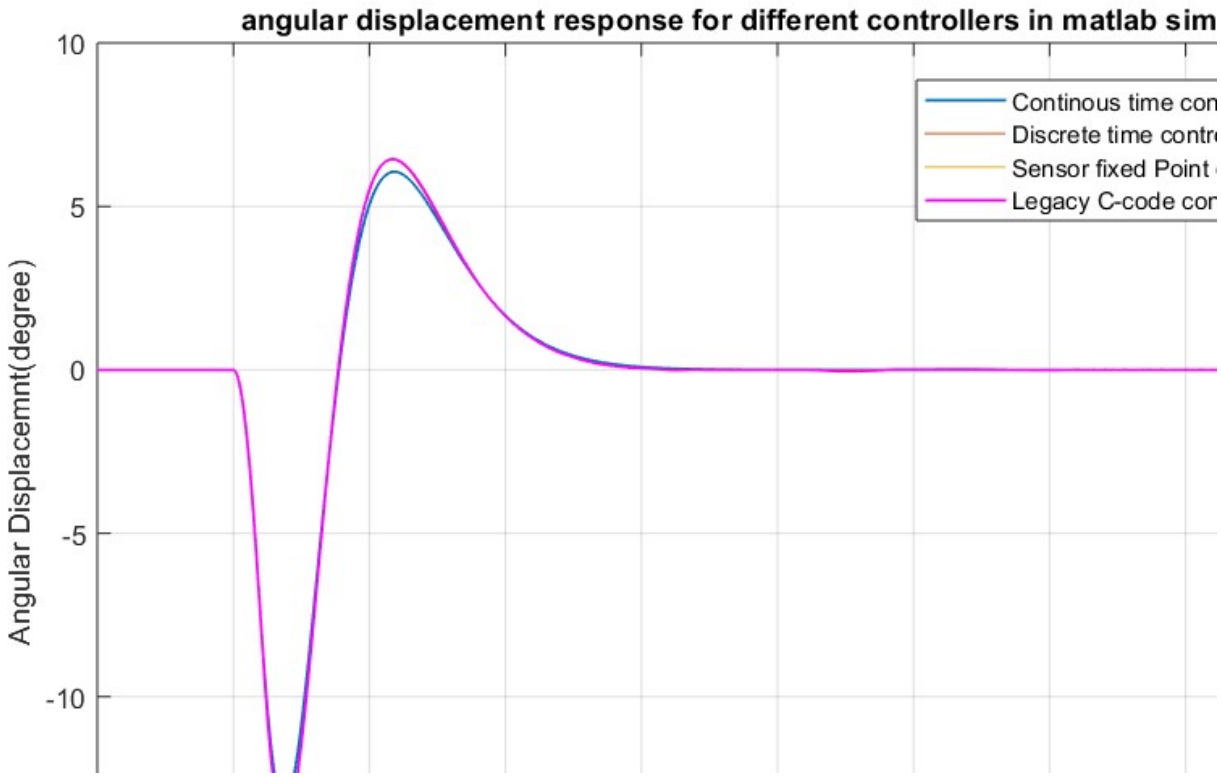


Figure 44: Angular displacement response with different controllers

The best response can be achieved as shown in figure 43, by the fixed-point controller and Legacy (C-code) controller. Both the controller responses match exactly and hence only the C-Code graph is visible in the plot. The damping of oscillations achieved within 4 seconds.

Lastly, both the fixed point and Legacy(c-code) controller have a very small (0.02%) steady state error. This steady state error is caused by the limit of the representation. If the control error is very small and the system does not have enough fractional bits to represent that error, then it is rounded to absolute zero. Hence, the controller does not realize any error and it will not perform any control action. The PI and state feedback controllers are designed, and simulation results are analyzed. The same controllers are implemented on real-time hardware.

5 SPEED CONTROL OF CART PENDULUM SYSTEM

The cart pendulum test stand has limit switches on both sides of the sledge. In reference phase, the cart

has to move with constant speed to detect the limit switches. So, there is a need of speed control of cart to make the reference phase precisely. In order to evaluate the speed control of the cart pendulum system a simple physical model is designed by neglecting the mass of a pendulum. A mathematical model is calculated by applying Newton's equation of motion.

5.1 Mathematical Modelling of Speed Control Cart Pendulum System

A simple free-body diagram of a cart pendulum is drawn as showed in the figure 45. The direction of the arrow shows the direction that the force is acting upon an object. According to the Newton's second law, sum of the forces are equal to zero.

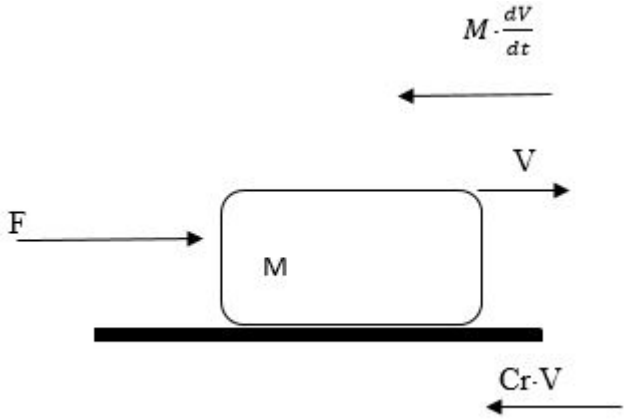


Figure45: Cart moving on conveyor Belt with friction

The parameters of the cart pendulum systems are as follows:
M =Mass of the cart (Kg)
Cr = Friction coefficient (Kg/s)
V = Velocity of the cart (m/s)
From the Newton's equations of motion,

$$F - Cr \cdot V - M \cdot \frac{dV}{dt} = 0$$

$$M \cdot \frac{dV}{dt} = F - Cr \cdot V$$

By converting the above equation from time domain to Laplace domain, the equation will be as follows.

$$M \cdot s \cdot V = F - Cr \cdot V$$

$$M \cdot s \cdot V + Cr \cdot V = F$$

$$V = \frac{1}{M \cdot s + Cr} \cdot F$$

Therefore, the Transfer function is

$$G = \frac{1}{M \cdot s + Cr} \quad (18)$$

5.2 Speed Control Model for Cart Pendulum System

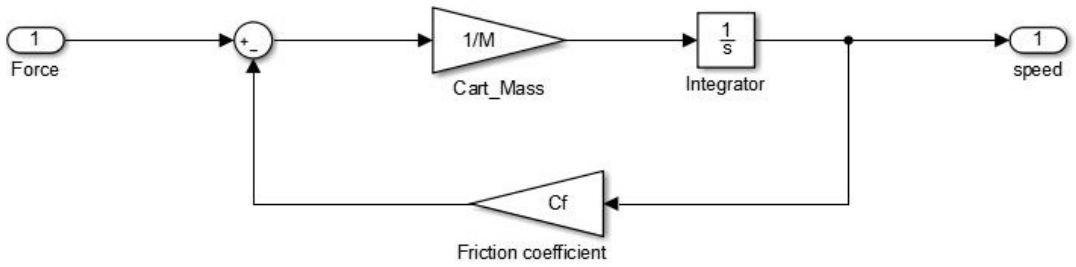


Figure46: Simulink model of Speed Control of cart pendulum

The unit step (Force = 1 N) is given as input to the cart pendulum system and observed the acceleration and velocity as outputs.

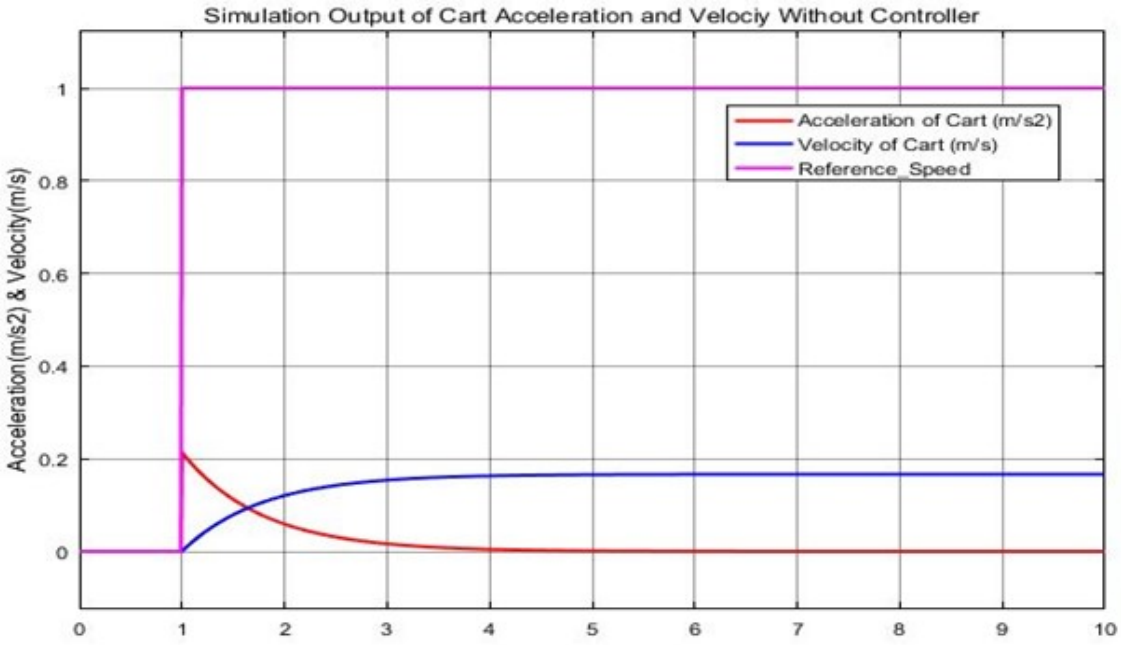


Figure47: Simulation Output of Cart Acceleration and Velocity without Controller

The above output clearly shows that velocity is reaching to 0.166, but it is not reaching the desired response. A constant force gives a constant acceleration. This gives linearly increasing velocity and therefore exponentially increasing displacement. To overcome this and to obtain constant velocity, we are going to design a PI controller to achieve the desired constant velocity of the cart. The PI Controller is as follows,

$$K = Kp + \frac{Ki}{s}$$

$$K = \frac{Kp \cdot s + Ki}{s} \qquad (19)$$

By multiplying both equation (18) and (19), we will get

$$K \cdot G = \frac{\left(\frac{Kp}{Ki}\right) \cdot s + 1}{\frac{s}{Ki}} \cdot \left(\frac{1}{Cr}\right) \cdot \frac{1}{\left(\frac{M \cdot s}{Cr}\right) + 1}$$

$$\text{If} \quad \frac{Kp}{Ki} = \frac{M}{Cr} \qquad \Rightarrow \quad K \cdot G = Ki/(Cr \cdot s)$$

$$Ki = Kp \cdot \left(\frac{Cr}{M}\right) \tag{20}$$

5.3 PI Controller

PI controller will eliminate forced oscillations and steady state error. The proportional gain is obtained by trial and error method, Integral gain is calculated by considering coefficient of friction (Cr) as 6.0 kg/s (detailed in appendix-II) from the equation (20).

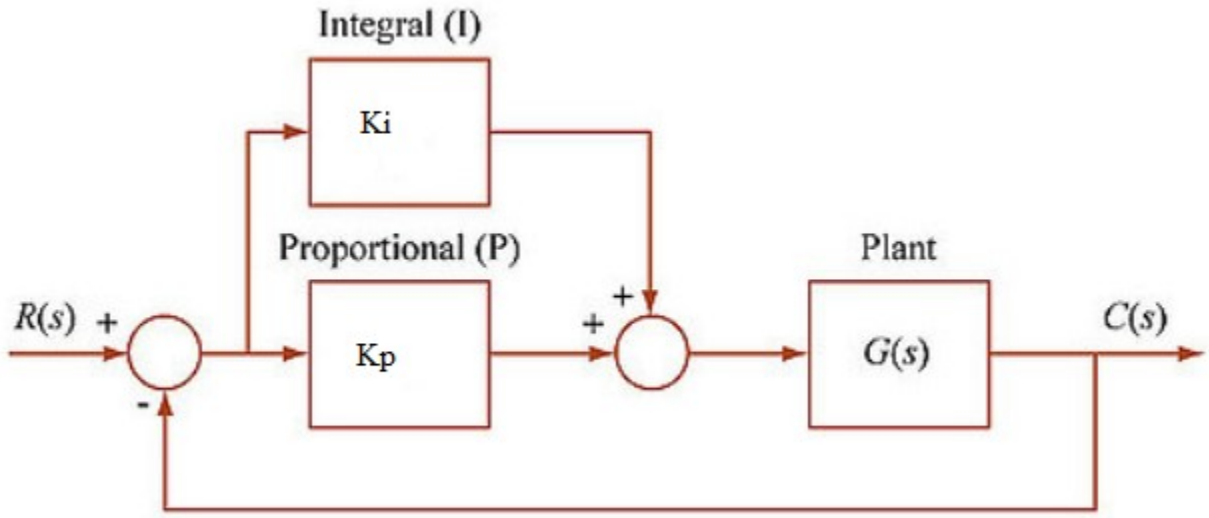


Figure48: Block diagram of PI Controller

5.3.1 Modelling and Simulation of PI Controllers

The complete model of PI- controllers are designed along with output feedback of the cart velocity. Although the performance of all controllers are acceptable but their behavior is not the same. All the controllers are operated using the multiport switch block in Simulink.

The speed control of cart using different controllers on the cart pendulum system shown in the figure 49.

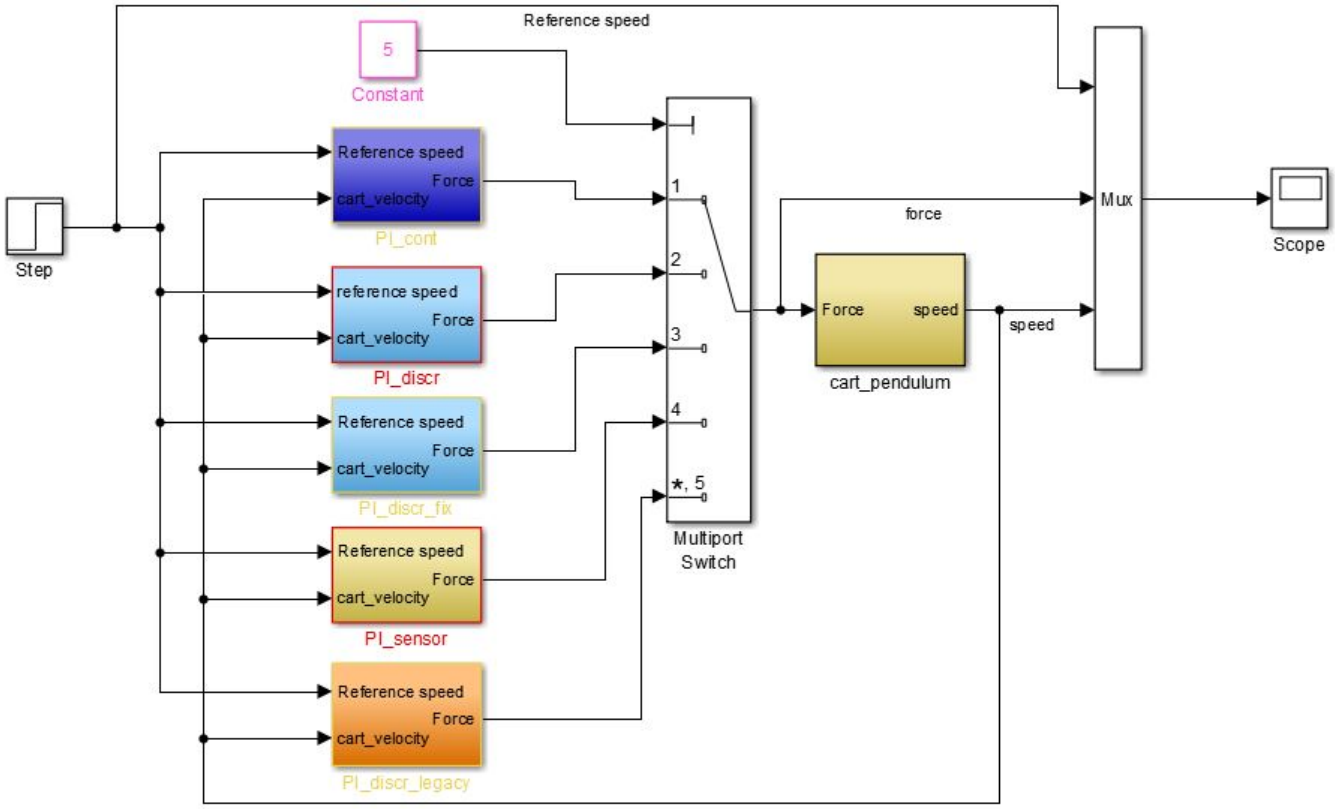


Figure49: Simulink model of the PI Controllers with Plant Model

The objective here is to achieve the constant speed of cart. Also, we check for all the controllers whether it is matching (i.e. continuous, discrete, fixed-point, sensor fixed point, CP legacy) the same output or not. From the test stand,

The mass of the cart (M) = $4.676\ kg$

By selecting, Proportional gain $KP\ = 3$, then

$$\mathrm{Ki} = \mathrm{Kp} \cdot \left(\frac{\mathrm{Cr}}{\mathrm{m}}\right)\ = 3.8494$$

5.3.2 Continuous PI Controllers

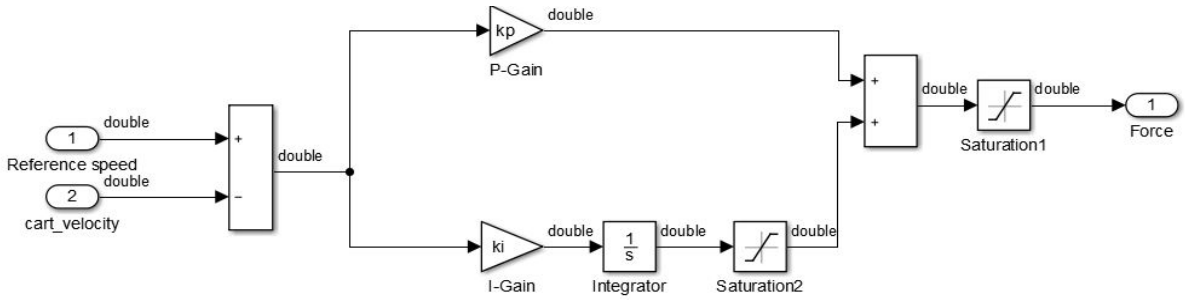


Figure50: Simulink model of Continuous PI Controller

In the above continuous Pi control, we used two saturation blocks by limiting the integral gain and the output in the range of -7 and $+7$ to limit the output force to the cart. The following is MATLAB Script file for the PI Continuous and discrete controllers

```
% cp_speedc_init
% Script for the Pi_contineous and Discrete
disp('cppi_cdc_init...')
Ts = 10e-3;          % sampling time (10 ms)
M = 4.676;           % Mass of the cart
Cf = 6.0;            % kinetic friction coefficient (Cf ~= 6.0)
kp = 3 ;             % P-gain
ki = (kp*Cf)/M;      % integral gain
kp_d = kp;           % Discrete_P-gain
ki_d = ki * Ts;      % Discrete integral gain
```

M-File 8: Matlab script for Pi_contineous and Discrete

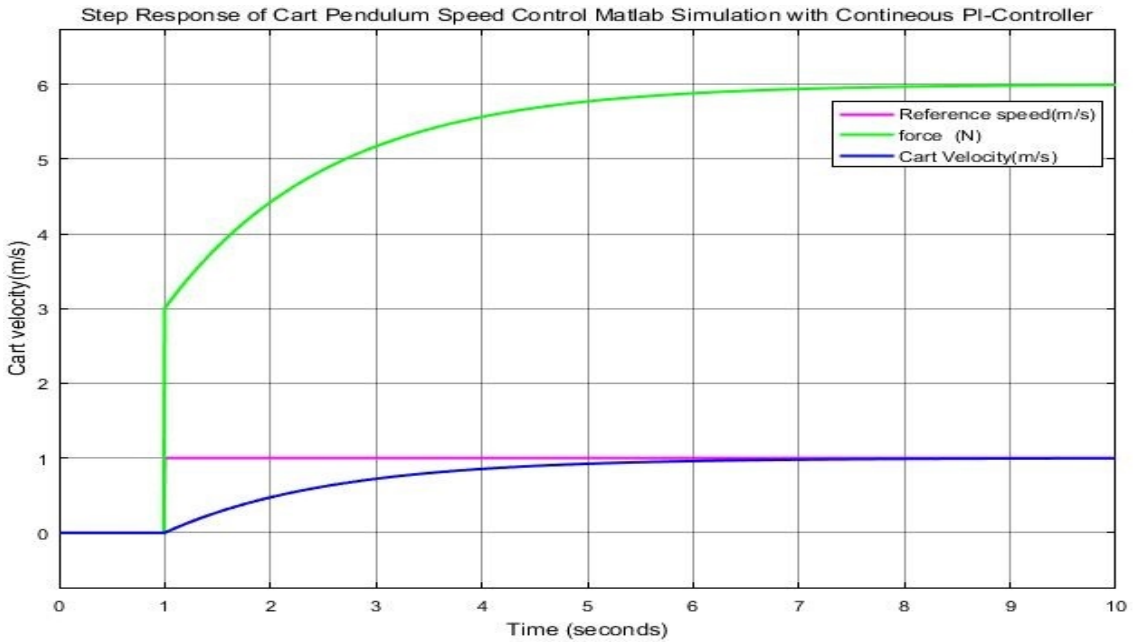


Figure51: Step response of Continuous PI-controller of cart pendulum in Simulink

It is evident from the above graph that the cart velocity reaches reference constant speed by applying the force of $6\ N$ to the cart.

5.3.3 Discrete PI Controllers

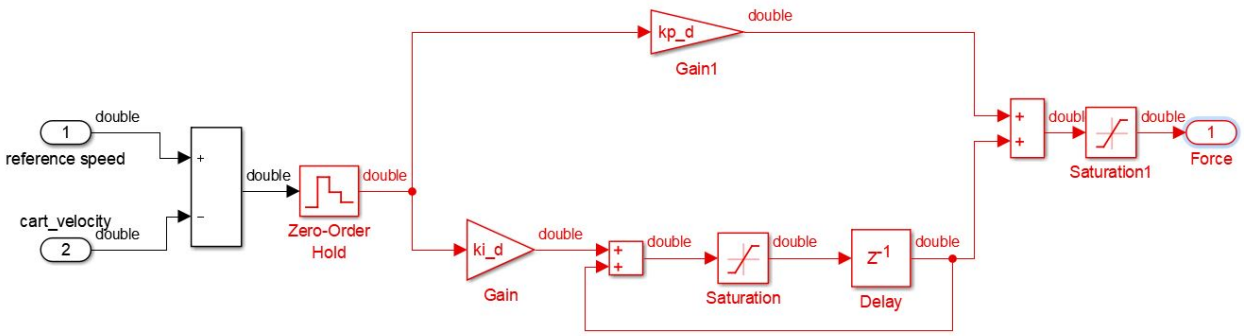


Figure52: Simulink model of Discrete PI Controller

In order to execute the discrete PI controller, change the switch to 2 in the constant block shown in the Simulink model. We used the saturation limits (+7 &-7) as similar to the continuous PI controller. The output of the controller is shown in figure 53.

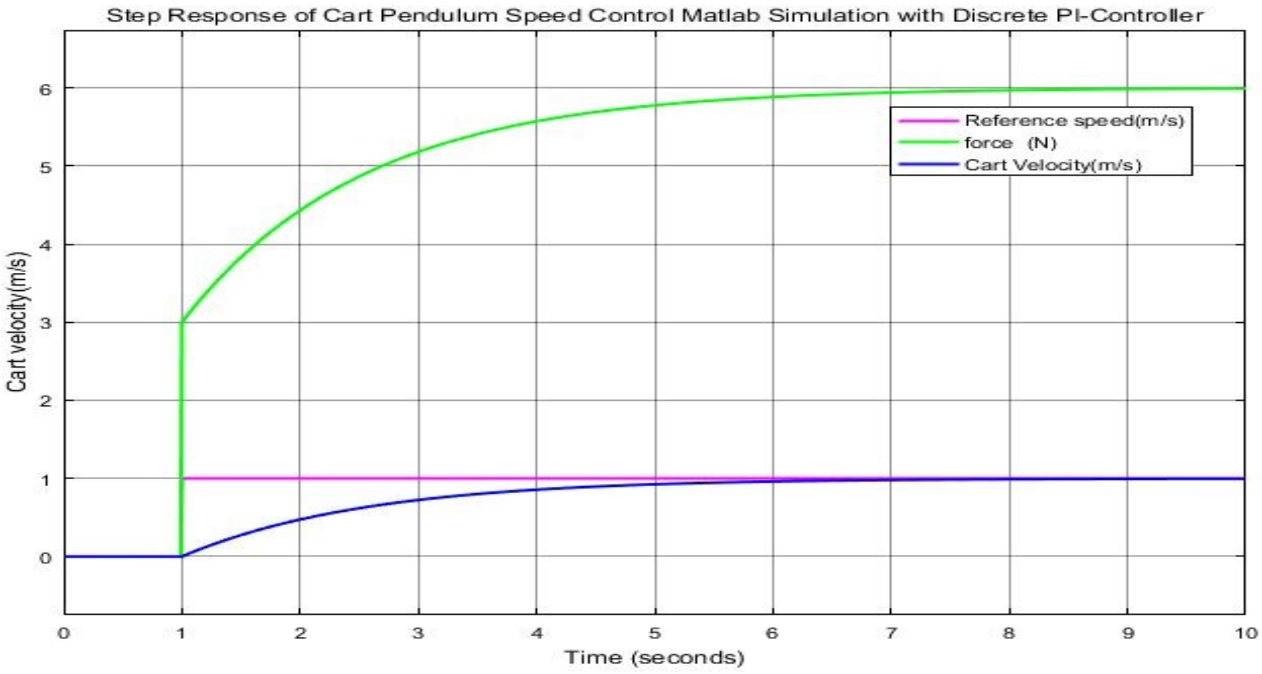


Figure53: Step response of the Discrete PI - controller of the cart pendulum in Simulink

From the above graphs, the output of the discrete time controller and continuous time controller are similar.

5.3.4 PI Controllers in Fixed Point Format

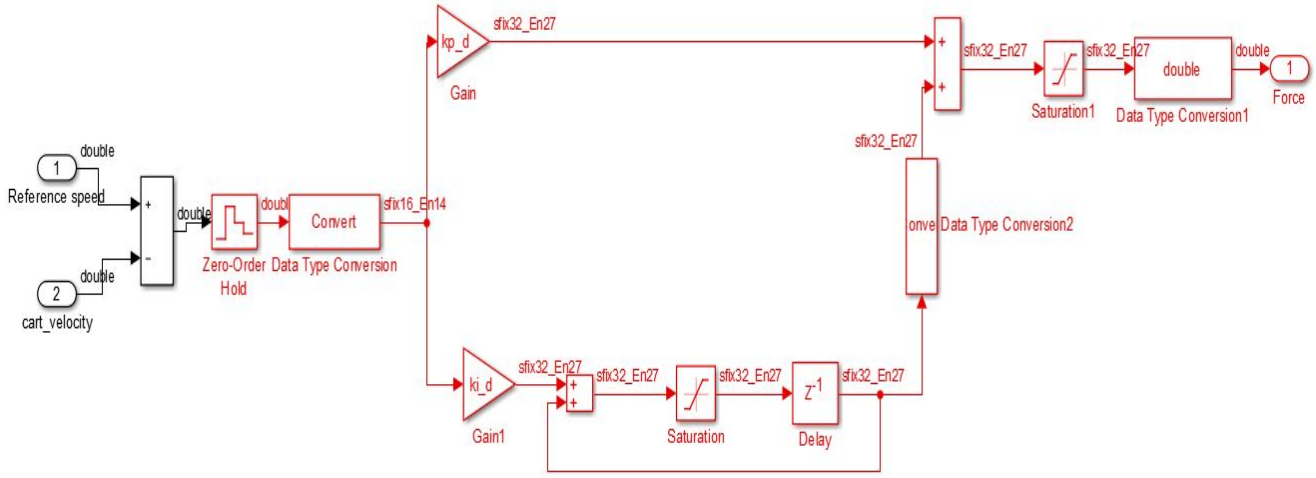


Figure54: Simulink model of fixed-point PI Controller

The calculation of the manipulating variable (Force) of the system in the continuous and the discrete time controller is done with double precision. This calculation requires a lot of hardware and takes a

significant amount of time in software and cost effective. Hence, when it comes to practical implementation such systems fail to fulfil the real time requirement. This emphasizes on the need of a fixed-point calculation.

On the other hand, it is also important to take care the data formats and the physical units required for computation and comparison later. The data formats for this section becomes important due to gain values for the system control, and conversions are necessary since MATLAB models work with floating point numbers.

Since input and output data 16-bits is sufficient to provide this for input and output values. In addition, the result of a 16 bits multiplication fits into 32 bits without any overflow. The data format in fixed point (for the real time system) is selected to int_16.13 for the proportional gain Kp, for the integral gain Ki is int_16.13. Thus, the selected gain values will not exceed the format specified. Due to fixed-point arithmetic the gain values must be shifted the same number of times as its precision. The following is MATLAB Script file for the Fixed-Point PI controller.

```
% Script for the Fixed_point_Pi_controllor
KP_SHIFT = 13;                          % int_16.13
KI_SHIFT = 13 ;                         % int_16.13
KP_D = round(kp_d * 2^KP_SHIFT);
KI_D = round(ki_d * 2^KI_SHIFT);
fprintf('----- cut below ------\n');
fprintf('#define KP_SHIFT  %2d       // int 16.%2d\n', KP_SHIFT, KP_SHIFT);
fprintf('#define KI_SHIFT  %2d       // int 16.%2d\n', KI_SHIFT, KI_SHIFT);
fprintf('#define KP_D  %d\n', KP_D);
fprintf('#define KI_D  %d\n', KI_D);
fprintf('----- cut above ------\n');
disp('That`s all.')
```

M-File 9: Matlab script for Fixed_point_Pi_controllor

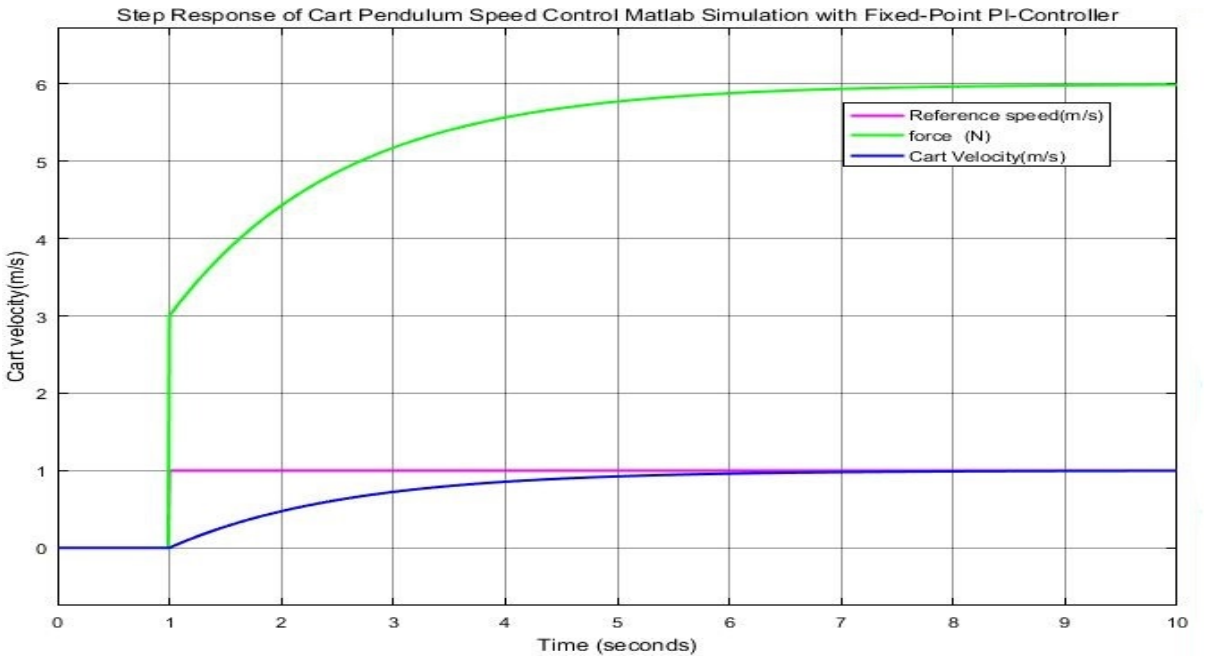


Figure66: Step response of Fixed-point PI-controller of cart pendulum in Simulink

As evident from the output graph of fixed-point model matches output of continuous time controller and discrete time controllers.

5.3.5 PI Controllers in Sensor Fixed Point Format

The designed continuous controller and its equivalent discrete and fixed-point form contain calculation in physical standards. But the real time system does not work on these quantities, as we use sensors to get the information about the state of the system generally provide data in digital form such as the incremental encoders used in the setup of the cart pendulum system.

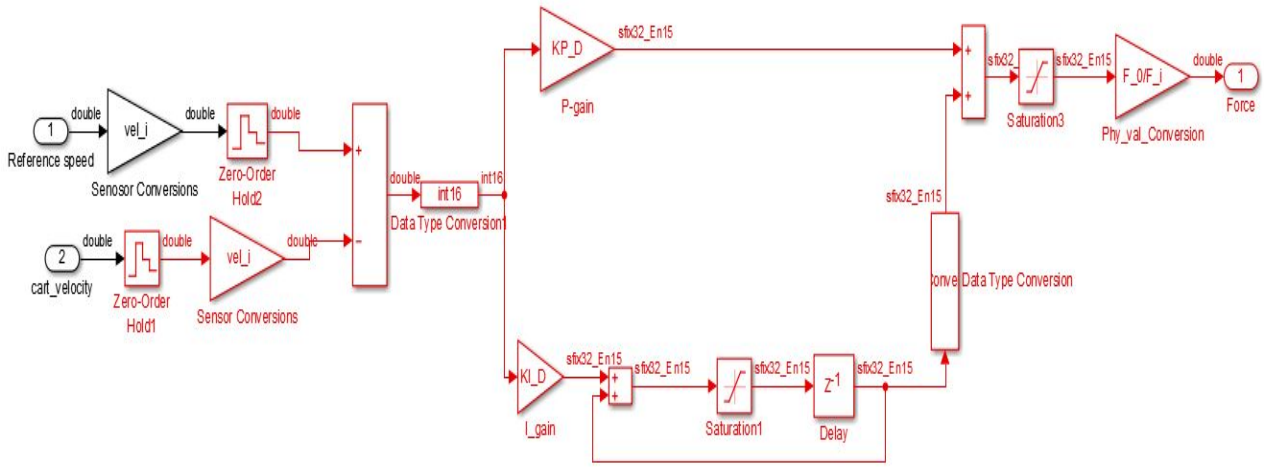


Figure67: Simulink model of Sensor fixed point PI Controller

These incremental encoders give output in integer format and this output is used as input to FPGA.

Hence this integer format must be converted to its physical value to carry out the controller calculations.

A better idea would be to include this proportional conversion factors in the respective gain values itself. This saves the time of the controller to convert integer to physical value and vice versa. The following MATLAB script file shows the sensor constants and the DAC conversion factor to convert counts into the physical force.

```
%% The sensor constants
F_i = 22.0532;% to convert DAC_values to voltage(1N = 'F_const'DAC counts)
F_0 = 1;
x_i_A = 57238;% displacemet of cart (1m = 'Disp_const' counts)
x_i = 28619;  % Overflow for x_i_A value in int_16,so x_i=x_i_A/2;
x_0 = 1;
vel_i = x_i*Ts;% velocity of cart per 10ms (Unit: m/10ms)
vel_0 = 1
alpha_i = 4096;% angular_displacment of pendulum (2*pi = 'Angle_const' counts)
alpha_0 = (2*pi);
omega_i = alpha_i*Ts ;% angular_velocity per 10ms (Unit: rad/10ms)
%% Accounting for Sensor conversion factor in Gain values
KP_D = (vel_0/vel_i)* kp_d *(F_i/F_0) % Proportional Gain
KI_D = (vel_0/vel_i)* ki_d *(F_i/F_0) % Discrete Integrator Gain
KP_SHIFT = 15;
KI_SHIFT = 15;
limit = 7;        % for integrator
LIMIT_INT = round(limit * 2^KI_SHIFT);
LIMIT_OUT = round(limit * 2^KP_SHIFT);
```

M-File 10: Matlab script sensor constants

Therefore, the fixed-point format for the gain values are decided after accounting for the sensor conversion factors. This type of controller model is called sensor fixed point model as it also accounts for the sensor conversion factors.

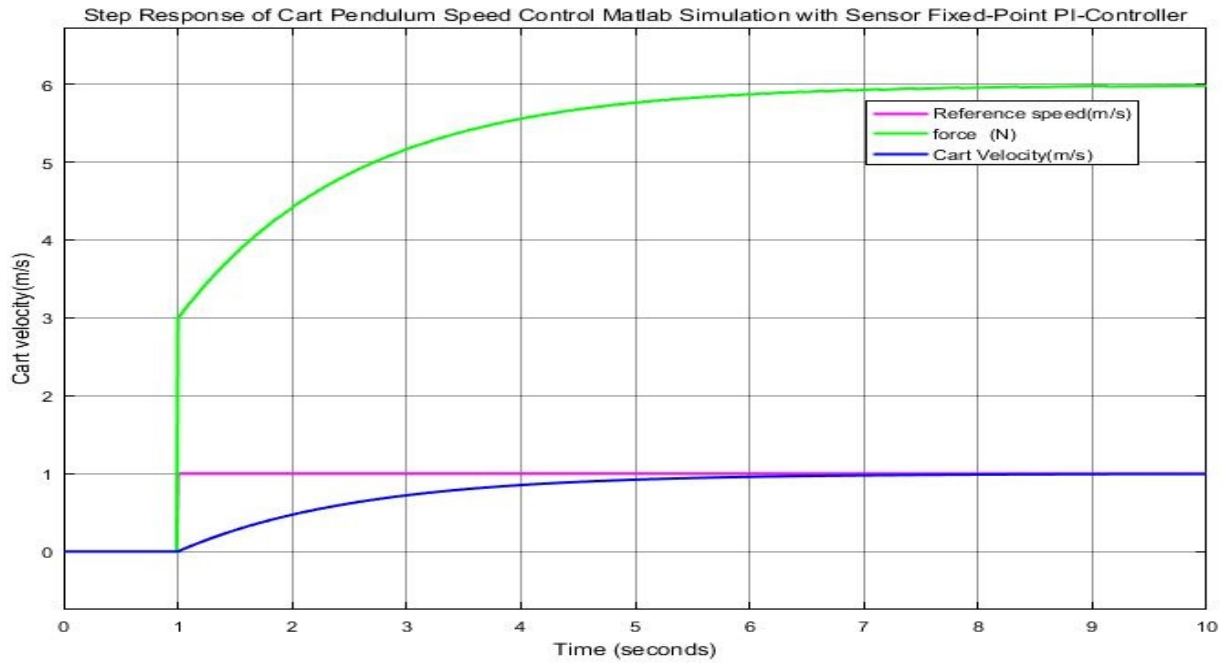


Figure68: Step response of Sensor Fixed-point PI-controller of cart pendulum in Simulink

From the graph, the response of Sensor Fixed-point PI-controller is quite good, and we have a little variation due to precession loss.

5.3.6 Developing the System Function

The S-function (System Function) is used to create the simulation block with the developed C-code in order to test the code in the model. It takes the source file and header file associated with it as input and generates a block of C-code as output which can be simulated with the model for testing and verification. This S-function is generated by the legacy code tool.

The MATLAB script file used for the legacy code block generation is shown below:

```
% create legacy mex function for cart control
def = legacy_code('initialize');
def.SourceFiles = {'spctrl.c', 'pi_control.c'};
def.HeaderFiles = {'spctrl.h', 'xil_types.h', 'pi_control.h'};
def.SFunctionName = 'ex_sfun_spctrl';
s16 spsctrl(s16 reset, s16 ee);
def.OutputFcnSpec = 'int16 y1 = spctrl(int16 u1, int16 u2)';
legacy_code('sfcn_cmex_generate', def)
legacy_code('compile', def)
legacy_code('slblock_generate', def) % generate simulation block
disp('That`s all folks.')
```

M-File 11: Matlab script generating Legacy block for Speed control

When the above script is executed in MATLAB it will generates the following simulation block.

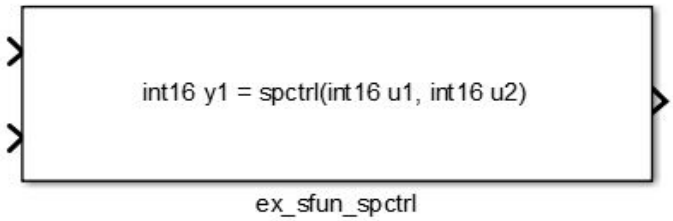


Figure69: Simulink legacy block

The '$y1$' is the output of the function which is the force in integer format. The function has six inputs out of which the last four are the states of the system, '$u1$' is reset signal and '$u2$' is the manipulating variable. The top-level function performed by this block can be explained by the C-file.

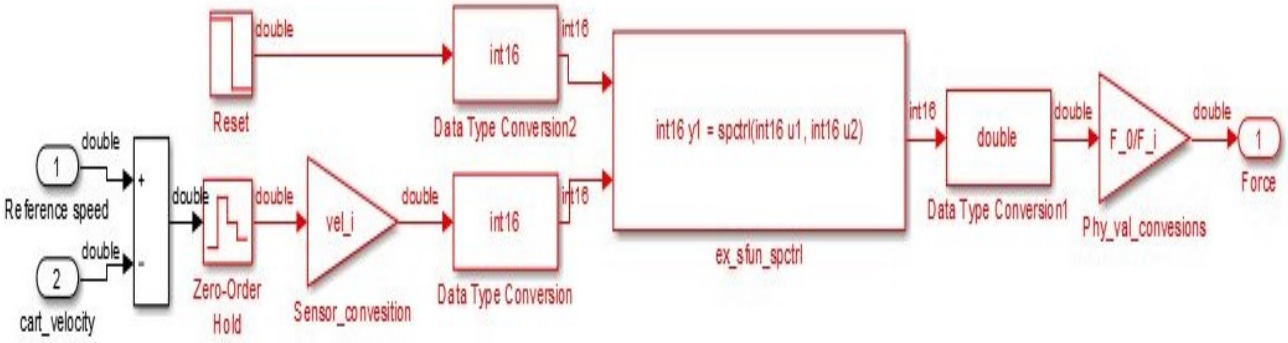


Figure70: Simulink model of PI Controller CP legacy

The gain 'vel_i' is the sensor conversion factor to convert the Force in fixed point format to pass into the C-code. At the end again we are converting to its equivalent physical value and pass to the plant. The Zero order hold blocks are placed so that the working of the controller exactly as in the real time environment can be simulated and verified for its output. The final output using the legacy code is given by the following graph.

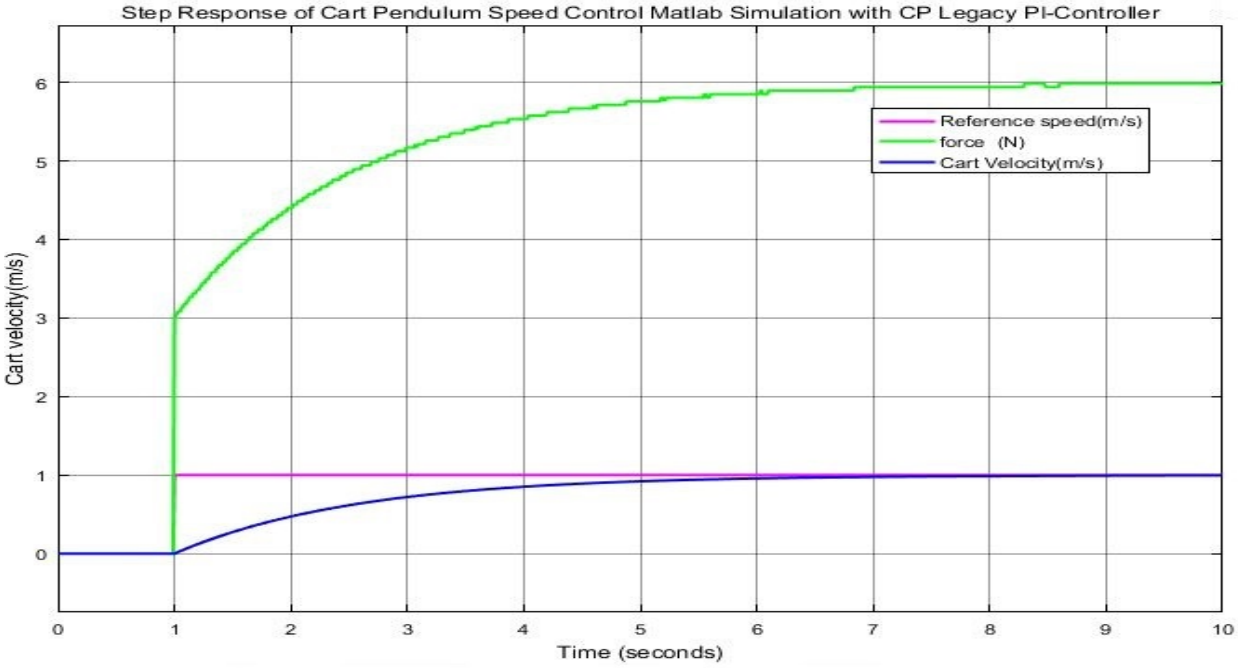


Figure71: Step response of Legacy PI-controller of cart pendulum in Simulink

The legacy c-code output graphs are matching with the sensor fixed point controller. As evident from the outputs from all the controllers are similar.

The designed speed controller (PI) has been tested on test stand in reference phase. The cart moved with constant velicity and detected the limit switches timely. The simuation results are verified on test stand.

6 HARDWARE OF CART PENDULUM SYSTEM

The simulated controllers discussed in chapter 4 &5 are being implemented in hardware. The hardware has to be analyzed before implementation. The hardware being the most important part in any embedded systems, it is important to know how each of the hardware component interact with other part (either software or hardware) to form a real time application.

The figure 72, shows the system components & its interfaces on cart pendulum system.

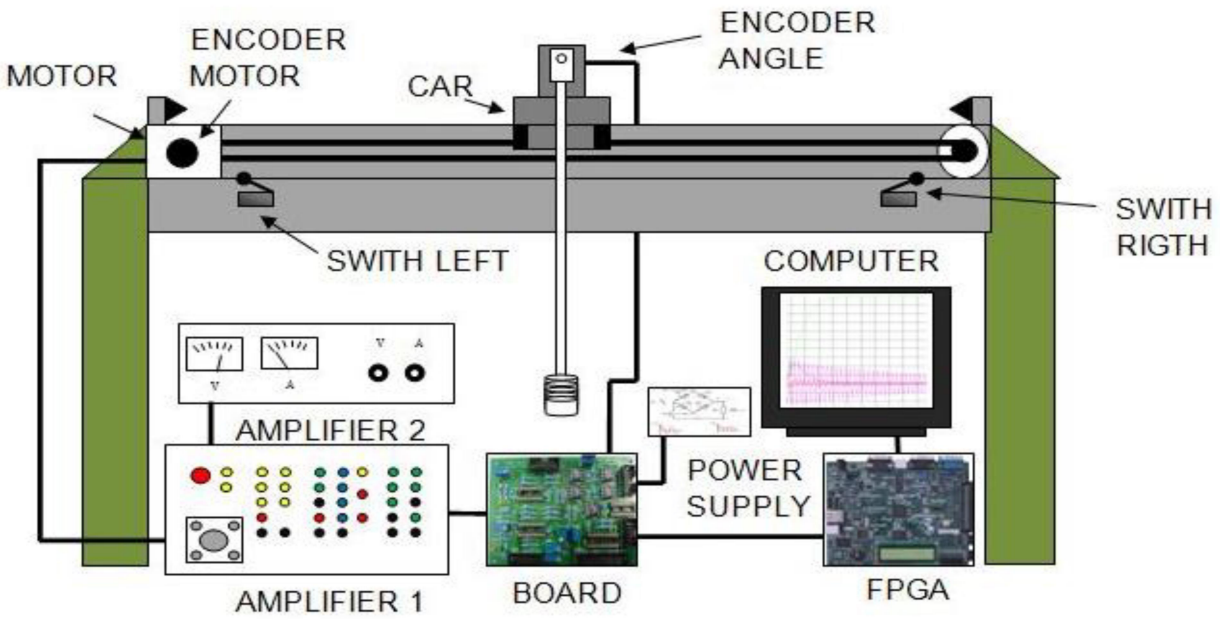


Figure72: Hardware Circuit Connection

- ## Field Programmable Gate Arrays (FPGA)

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application.

In cart pendulum system, Spartan 3E FPGA is used as main hardware component. The Spartan™-3 family of Field-Programmable Gate Arrays is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. The eight-member family offers densities ranging from 50,000 to five million system gates.

- ## Encoder

A rotary encoder, also called a shaft encoder, is an electro-mechanical device that converts the angular position or motion of a shaft or axle to analog or digital output signals.There are two main types of rotary encoder: absolute and incremental.The output of an incremental encoder provides information

about the *motion* of the shaft, which typically is processed elsewhere into information such as position, speed and distance.

- **Amplifier**

An amplifier is an electronic device or circuit which is used to increase the magnitude of the signal applied to its input

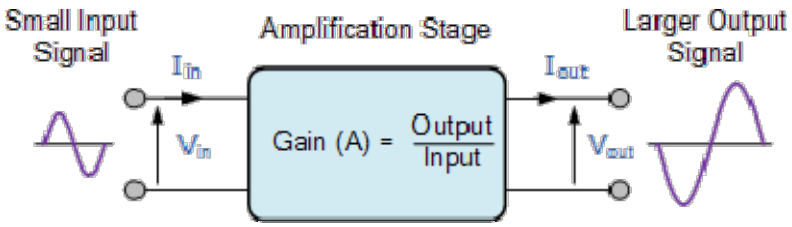


Figure73: Amplifier circuit

- **Cart**

The moving element in the system, which has the tendency to move with a pendulum mounted on it in this case a pendulum with a small mass. The cart system is only a piece of mass which is coupled with the transmission belt engages to the motor shaft.

- **Electrical Drive (DC servo motor)**

The control of the system requires an element which generates the movement of the cart to reach the desired position, for this it needs the introduction of a drive that controls the position of the cart.

6.1 Hardware Description

Following Hardware are being used in the Cart Pendulum Set up.

1) FPGA: Spartan 3E Board
2) Electrical Interface Circuit
3) Violin Servo Amplifier
4) Incremental encoders (i.e. one for the cart position and other for the pendulum angle)
5) Power supply

1) Spartan 3E Board

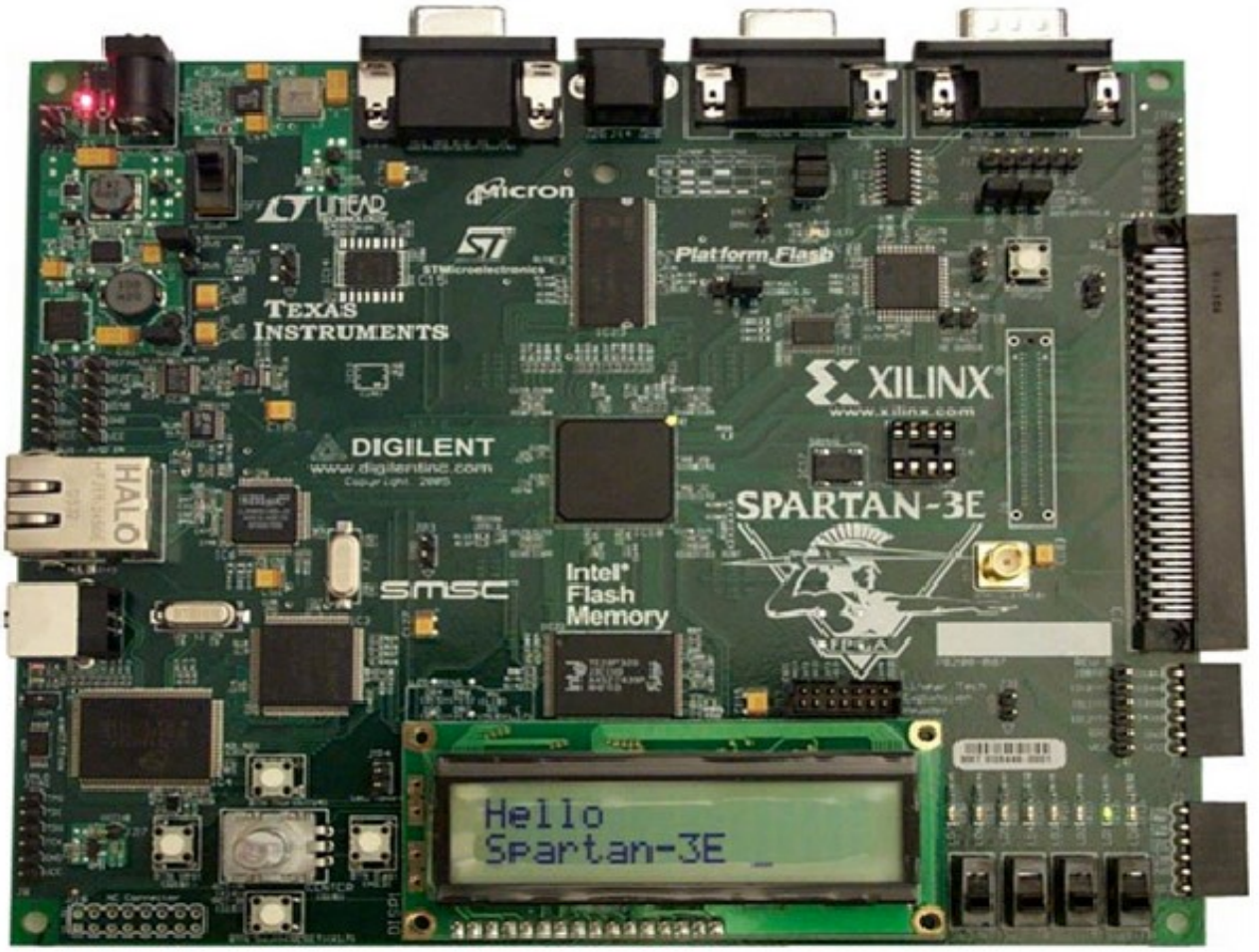


Figure74: Spartan 3E Board

The Spartan-3E Starter Kit board highlights the unique features of the Spartan-3E FPGA family and provides a convenient development board for embedded processing applications. The Spartan-3E Starter Board provides a powerful and highly advanced self-contained development platform for designs targeting the Spartan-3E FPGA from Xilinx. It features a 500K gate Spartan-3E FPGA with a 32-bit RISC processor and DDR interfaces.

6.1.1 Key features in Spartan 3E

- Xilinx XC3S500E Spartan-3E FPGA
  - Up to 232 user-I/O pins
  - 320-pin FBGA package
  - Over 10,000 logic cells
- Xilinx 4 Mbit Platform Flash configuration PROM
- Xilinx 64-macrocell XC2C64A Cool Runner CPLD
- 64 Mega Byte (512 Mbit) of DDR SDRAM, x16 data interface, 100+ MHz
- 16 Mega Byte (128 Mbit) of parallel NOR Flash (Intel Strata Flash)
  - FPGA configuration storage

    - MicroBlaze code storage/shadowing
- 16 Megabits of SPI serial Flash (STMicro)
    - FPGA configuration storage
    - MicroBlaze code shadowing
- 2-line, 16-character LCD screen
- PS/2 mouse or keyboard port
- VGA display port
- 10/100 Ethernet PHY (requires Ethernet MAC in FPGA)
- Two 9-pin RS-232 ports (DTE- and DCE-style)
- On-board USB-based FPGA/CPLD download/debug interface
- 50 MHz clock oscillator
- SHA-1 1-wire serial EEPROM for bitstream copy protection
- Hirose FX2 expansion connector
- Three Digilent 6-pin expansion connectors
- Four-output, SPI-based Digital-to-Analog Converter (DAC)
- Two-input, SPI-based Analog-to-Digital Converter (ADC) with programmable-gain pre-amplifier
- ChipScope™ SoftTouch debugging port
- Rotary-encoder with push-button shaft
- Eight discrete LEDs
- Four slide switches
- Four push-button switches
- SMA clock input
- 8-pin DIP socket for auxiliary clock oscillator

The above provided features had made Spartan 3E board most popular among the Spartan family. From the above features we are using the basic thing for development of Anti-sway control system.

- **Microblaze**

The MicroBlaze is a virtual microprocessor that is built by combining blocks of code called cores inside a Xilinx Field Programmable Gate Array (FPGA).

The MicroBlaze processor is a 32-bit Harvard Reduced Instruction Set Computer (RISC) architecture optimized for implementation in Xilinx FPGAs with separate 32-bit instruction and data buses running at full speed to execute programs and access data from both on-chip and external memory at the same time. The backbone of the architecture is a single-issue, 3-stage pipeline with 32 general-purpose registers and an Arithmetic Logic Unit (ALU), a shift unit, and two levels of interrupt. This basic design can then be configured with more advanced features to tailor to the exact needs of the target embedded application such as: barrel shifter, divider, multiplier, single precision floating-point unit (FPU), instruction and data caches, exception handling, debug logic, Fast Simplex Link (FSL) interfaces

and others.

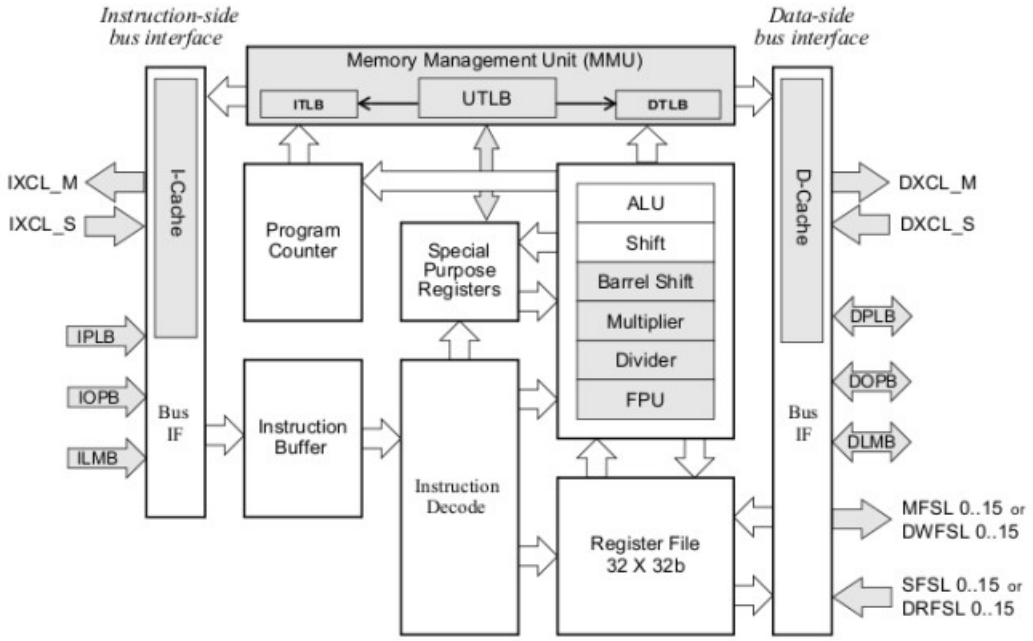


Figure75: Block Diagram of MicroBlaze

The blocks with white background are the backbone of the MicroBlaze architecture while the items shaded gray are optional features available depending on the exact needs of the target embedded application. Because MicroBlaze is a soft-core microprocessor, any optional features not used will not be implemented and will not take up any of the FPGAs resources.

The configuration and the attachment of user logic is carried out in Xilinx EDK (Embedded Design Kit).The user logic is usually attached to a bus system which is called PLB (Processor Local Bus), copyrighted by IBM SOFTWARE. The integration of the controller for Cart Pendulum system is carried out by writing software for the MicroBlaze platform.

- **Digital to Analog Converter (DAC)**

The Spartan®-3E FPGA Starter Kit board uses SPI-compatible, four-channel, serial Digital-to-Analog Converter (DAC). The DAC device is a Linear Technology LTC2624 quad DAC with 12-bit unsigned resolution. The four outputs from the DAC appear on the J5 header, which uses the Digilent 6-pin Peripheral Module format. The DAC and the header are located immediately above the Ethernet RJ-45 connector.

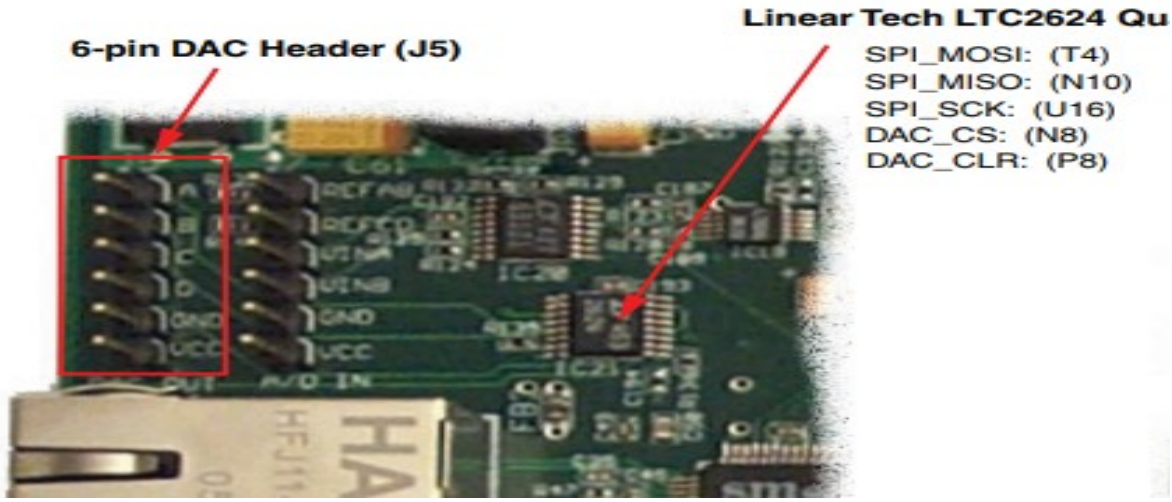


Figure 76: Linear Technology LTC2624

- **Pin and voltage levels for SPI**

SPI_MOSI----------> T4 ----------> FPGA----------> DAC (Serial data: Master Output, Slave Input)
DAC_CS ----------> N8 ----------> FPGA ----------> DAC (Active-Low chip-select. Digital-to-analog Conversion starts when signal returns high.)
SPI_SCK ----------> U16 ----------> FPGA ----------> DAC (Clock)
DAC_CLR ----------> P8 ----------> FPGA----------> DAC (Asynchronous, active-Low reset input)
SPI_MISO ----------> N10 ----------> FPGA----------> DAC (Serial data: Master Input, Slave Output)

6.2 Electrical Interface Circuit

The interface circuit used here has two important roles to be played in our project.
It has two important sides are the input of the signals coming on from the different devices supply, and another one is the output to the actuators.

- **OP470P-(Op-Amp)**

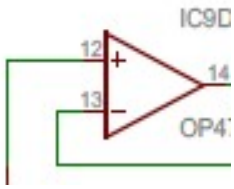


Figure77: OP470P operational Amplifier

The output from the DAC is unipolar rail-to-rail but we need a bipolar analog input. This OP470P circuit above is used to provide gain and bias to convert from unipolar $0 - 2.5V$ output of DAC to bipolar Input to servo amplifier to $\pm 3.75V$.

- **74HC125N**

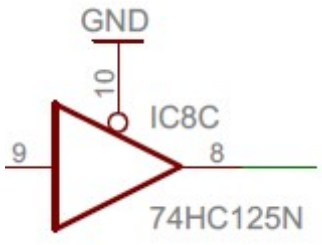


Figure78: Tri-State Buffer

The 74HC125 has four independent buffer and this independent buffers gates has with 3-state outputs. Each buffer has a separate enable pin that if driven with a high logic level places the corresponding output in the high impedance state. The device is designed for operation with a power supply range of 2.0V to 6.0V.

The basic concept of the third state, high impedance (Hi-Z), is to effectively remove the device's influence from the rest of the circuit. If more than one device is electrically connected to another device, putting an output into the Hi-Z state is often used to prevent short circuits, or one device driving high (logical 1) against another device driving low (logical 0).

- **HCPL-9030A**

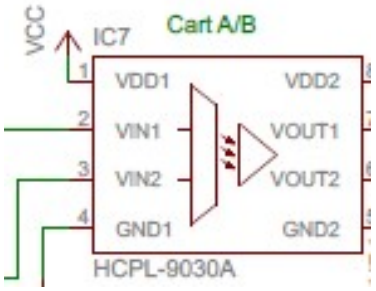


Figure 79: HCPL-9030A

The HCPL-9030A combines voltage level conversion and digital isolation in the same device. Different supply voltages can be connected to both sides of device. The logic signals are converted from one voltage to another without transfer of input side voltage spikes to the output side.Interface signals should not connect directly to the FPGA. In some cases voltage levels do not match so that a level conversion is required. The standard logic for interface is 5V, modern logic operates at 3.3V or below.

- **The MAX3095C:**

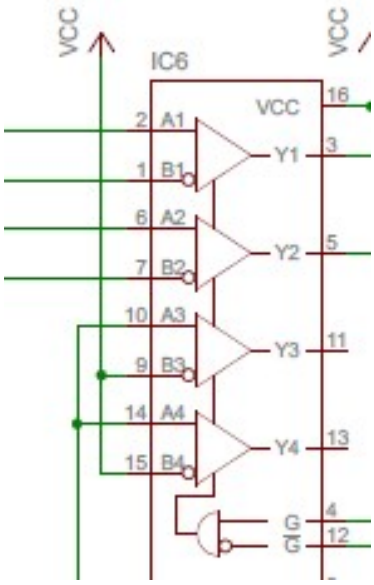


Figure80: MAX-9030C

The MAX3095/MAX3096 are rugged, low-power, quad, RS-422/RS-485 receivers with electrostatic discharge (ESD) protection.

The position and angle encoders send differential voltage to avoid errors in transmitting signals in a noisy environment. We have max3095c device that performs differential to single-ended signal conversion. Since this device operates from Vcc= 5V, a level conversion. The voltage of the signals is then converted to the 3.3 V logic voltage of the FPGA.

The Electrical Interface board transmits the analog control signal of the FPGA to the servo amplifier, which controls the servomotor. The analog logic voltage of the FPGA is in the range of $0\ V$ to $3.3\ V$. This signal is converted to a voltage of $-3.75\ V\ to\ +3.75\ V$.

A voltage of $0\ V$ at the DAC thus corresponds to $-3.75\ V$ at the servo amplifier, while $1.25\ V$ at the FPGA corresponds to 0 V at the servo amplifier. The VIOLIN servo amplifier has a power supply for the motor of $\pm 15\ A$.

6.3 SERVO AMPLIFIER

From the block diagram it shows the basic function of the servo amplifier:

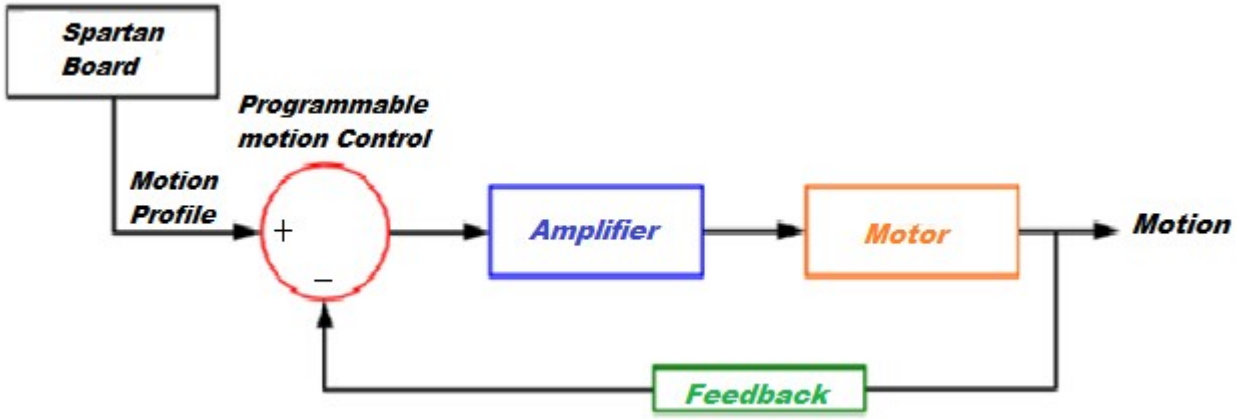


Figure81: Block diagram of Amplifier

1) **Input**: The digital outputs received from the Spartan board through digital isolator into the power required for the servo motor to move the Cart Pendulum system.
2) **Controller:** The controller is responsible for calculating the path or trajectory required and sending low-voltage command signals to the Amplifier/drive.
3) **Amplifier/Drive:** The Amplifier then takes the control signal from the controller and amplifies it to deliver a specific amount of voltage and current to the motor achieve the required motion.
4) **Motor:** As the amplified current required to produce the torque is supplied from the amplifier now the electrical energy is converted into mechanical energy for rotation.
5) **Feedback:** the feedback device evaluates the relation of the control input to the actual position of the mechanism or control shaft. By understanding the relationship between the actual value and "wanted value" of the carts position, the controller is able to send a signal to the drive for corrective action in the motor.

Additionally, it also has signals for turning the amplifier on/off and for knowing the status of the amplifier. These signals are used for turning the servo motor on and off irrespective of the voltage drop across the servo motor.

The common mode being the torque-mode amplifier. It converts the command signal from the controller into a specific amount of current to the motor. Since current is directly proportional to torque(shown in below T-I curve), the drive is controlling the amount of torque that the motor produces. In a linear motor, where current is proportional to force, the drive is directly controlling the force output of the motor.

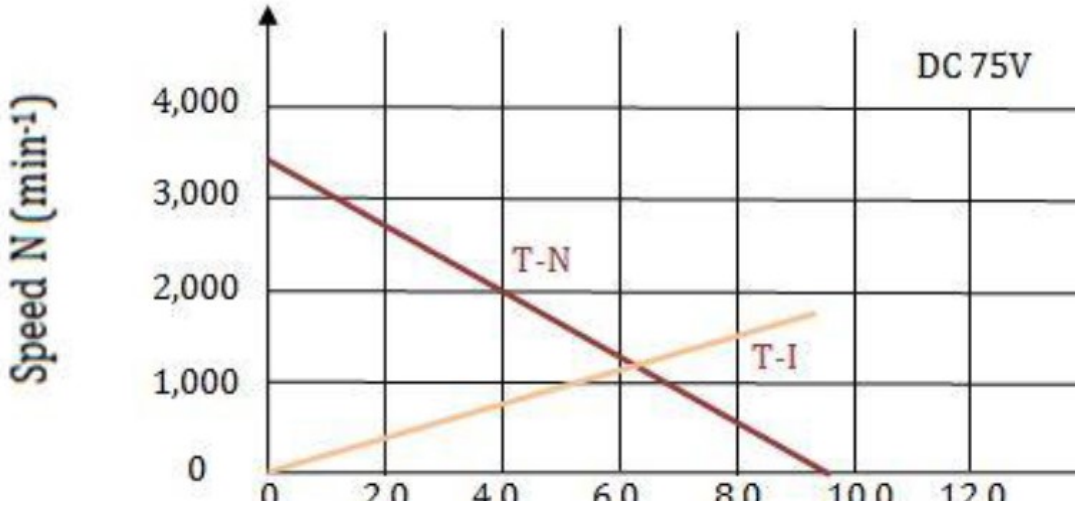


Figure: T – I Characteristics curve

MOTOR AND ENCODER

Technical Data

Motor MV 300	
Rated power [W]	300
Nominal / maximum speed [min-1]	2500/4000
Nominal / peak torque [Nm]	1.20/3.60
Rated / peak current [A]	5.1/15.3
Torque constant [Nm/ A]	0.268
Motor Encoder	
Power supply	5VDC(+-5%)/ 200mA
Resolution	1000 Incr./Rev
Signal output	Rectangle(max 70 kHz), RS422
Output voltage	V_low -0V, V_high +2.5V
Pendulum encoder	
Power supply	4.75 VDC to 5.5 VDC/ max 70mA
Resolution	4096 Incr./Rev
Signal output	TTL(max 200 kHz), RS422

Table2: Motor and Encoder Data Sheet

6.4 Power Supply Unit

The main function of a power supply is to convert AC voltage in DC voltage. It has the following component:

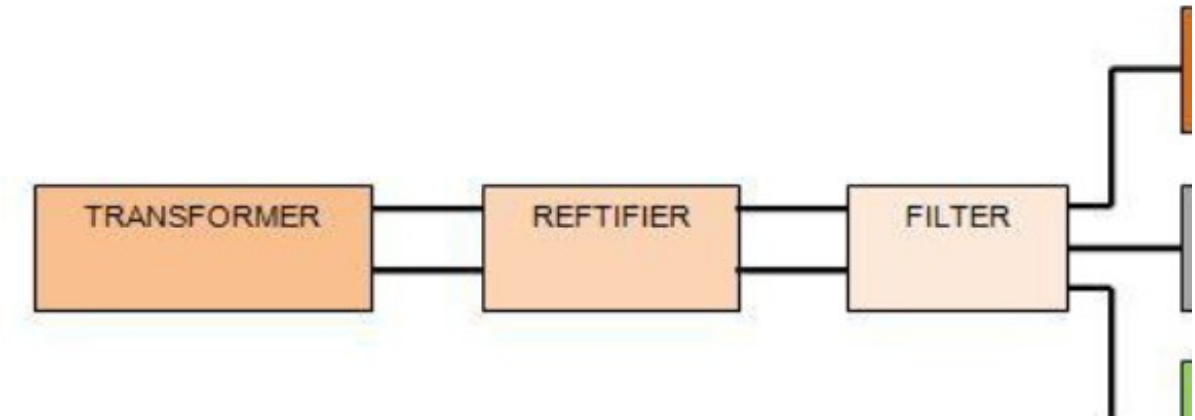


Figure82: Block Diagram of power supply unit

- **Transformer**

The input voltage is transformed into secondary voltage, this transformer has input of $220\ V$ and output of $15V$.

- **Diode rectifier**

The output from the transformer which is an alternating voltage is rectified to a direct voltage. In our case, it used the rectifier is called the rectifier bridge.

The function is very simple, the diodes only when its anode voltage is greater than its cathode voltage. The rectifier bridge is a combination of four simple diodes. With this diode bridge what we get is the rectification of the sine wave in a double wave. As we can see, although we have not yet acheived a continuous signal, this achievement was reached through the following phases.

- **Filter**

The purpose of the filter is when the signal of the wave reaches the Vmax and starts to decrease; the filter has to maintain the value of the output as constant as possible, meaning that we need an element which provides the system voltage between two maximums peaks of each peak wave.

- **Regulator**

A regulator is an electrical circuit that is responsible for reduce the ripple and to provide an output voltage of the exact voltage is needed for the output. The voltage signal enter from the network, it transformed from $220\ V\ to\ 5\ V$ then it is rectified from a sinusoidal signal to a double wave, that signal is filtered and approximately to a constant output with some ripple, the signal is the input for the regulator which makes it constant in a desired value and reduce the ripple from input to output near 1000 times.

7 HARDWARE DESIGN ON FPGA

The system is built with the Spartan-3E FPGA development kit. The major components available on Spartan board are Xilinx XC3S500E Spartan-3E FPGA, SPI-based Digital-to-Analog Converter (DAC), 2-line, 16-character LCD screen, Rotary-encoder with push-button shaft. In Cart Pendulum system the interfacing of available components is given by hardware configuration on FPGA. FPGA is configured with writing the VHDL code for each component in terms of Intellectual property (IP). Later, the designed logic is integrated with the existing IP with the Xilinx MicroBlaze™ CPU.

The hardware generation with Softcore CPU and User IP is designed with help of Xilinx ISE software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. In the cart pendulum system, the following user IP are created for interfacing of sensors and actuators.

1. **rotenc_8.vhd:** Interfacing of the rotory knob rotation detection present on hardware kit.
2. **iencspd.vhd:** Interfacing of the incremental encoder as sensor for cart position & pendulum angle.
3. **Spis24.vhd:** Interfacing of the onboard Digital to Analog (DAC) converter with FPGA pins.

After generation of hardware on the FPGA, the device is ready for software implementation which uses the MicroBlaze CPU to run it on. To communicate with the data available at User IP hardware with the MicroBlaze CPU, internally slave registers are used. The hardware is configured with memory mapped in the microcontroller address space the drivers can directly access the interface registers via it’s decoded memory registers.

7.1 User IP Block

For interacting with outside world of hardware device we need to create the interface between hardware and sensor, actuators. that is done by creating an IP block as shown in figure 83.

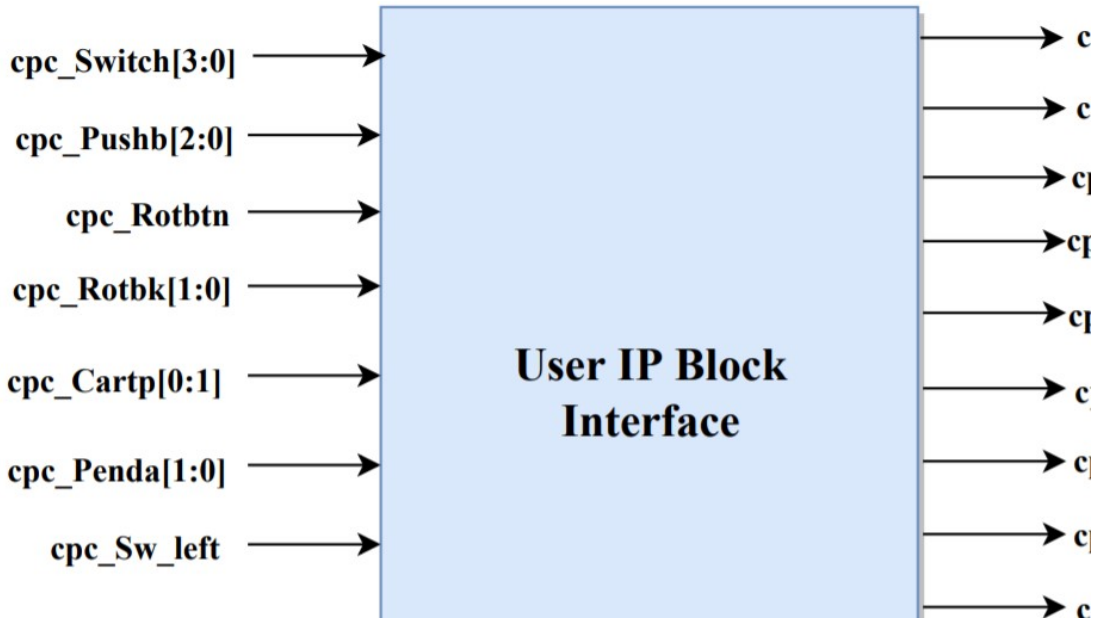


Figure83: User IP Block

The input signals of the IP block are shown on the left-hand side and the output signals are shown on the right-hand side. A total of ten slave registers are created in the FPGA. This register is assigned with specific hardware either with input or output.

Input Signals

- **cpc_Switch:** It is vector signal, consists of four different signals with same name. It is used to read the status of the switches on the FPGA board.
- **cpc_Pushb:** It is implemented to read the three push button on the development kit.
- **cpc_Rotbtn:** It is push button attached with rotary button.
- **cpc_Rotbk:** The rotary button knob works on the principle same as the incremental encoders. It has two shaft encoders ROT_A and ROT_B. VHDL code is written further which gives the rotations as counts.
- **cpc_Cartp:** The encoder attached to motor outputs two continuous signal. Based on the signal received a VHDL file is implemented with which the FPGA converts to 20-bit digital value.
- **cpc_Penda:** The encoder used to record the pendulum angle, based on signal received by FPGA, it converts it to 16-bit digital value.
- **cpc_Sw_left:** To read the status of the left limit switch on the belt.
- **cpc_Sw_right:** To read the status of the right limit switch on the belt.
- **cpc_Amp_ok:** To check if the servo amplifier is powered on or off.

Output Signals

- **cpc_Leds:** To write the value to the LEDs on the board.
- **cpc_Amp_ena:** To enable and disable the servo amplifier.
- **cpc_Lcd:** To perform read/write operations on the LCD. Here the LCD is used in the 4-bit mode.
- **cpc_SPI_MOSI:** The Master Out Slave In line to send the data serially to the DAC.
- **cpc_SPI_SCLK:** The clock signal for the SPI communication.
- **cpc_DAC_CS:** Active-Low Chip select, to start SPI communication.
- **cpc_DAC_CLR:** Asynchronous, active – Low reset input.

7.2 Slave Registers

The following table shows the elaborated description on the configuration of slave registers.

Slave Register	Bit Location	Signal Caption
slv_reg0	28 to 31	cpcu_Switch [3:0]
slv_reg0	25 to 27	cpcu_Pushb [2:0]
slv_reg0	24 to 31	cpcu_Leds [7:0]
slv_reg1	23	cpcu_Rotbtn
slv_reg1	24 to 31	rotpos_i
slv_reg1	25 to 31	cpcu_lcd [0 :6]
slv_reg2	8 to 31	SPI_DAC
slv_reg3	12 to 31	pos20_i
slv_reg4	16 to 31	pos16_i
slv_reg5	29	cpcu_Sw_left
slv_reg5	30	cpcu_Sw_right
slv_reg5	31	cpcu_Amp_ok
slv_reg5	5	cpcu_Amp_ena

Table 3: Slave Registers and their Accessing Address

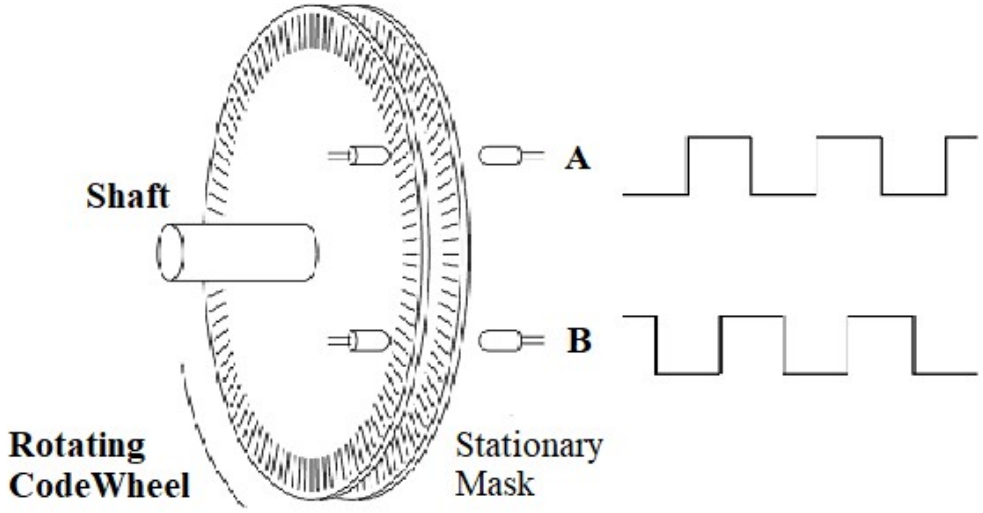
The following code shows the signals read from the hardware and stored in slave register to get

accessed for MicroBlaze CPU.

```
case slv_reg_read_sel is
  when "1000000000" => slv_ip2bus_data <= slv_reg0(0 to 24) & pushb_r & switch_r
  when "0100000000" => slv_ip2bus_data <= slv_reg1(0 to 22) & rotbtn_r & rotpos_
  when "0010000000" => slv_ip2bus_data <= slv_reg2(0 to 30) & spidone_i;
  when "0001000000" => slv_ip2bus_data <= "000000000000" & pos20_i;
  when "0000100000" => slv_ip2bus_data <= "0000000000000000" & pos16_i;
  when "0000010000" => slv_ip2bus_data
                       <= "00000000000000000000000000000000" & cpcu_Sw_left & cpcu_Sw_
  when "0000001000" => slv_ip2bus_data <= "000000000000" & imp20_i;
  when "0000000100" => slv_ip2bus_data <= "000000000000" & timer_lin;
  when "0000000010" => slv_ip2bus_data <= "0000000000000000" & imp16_i;
```

H-file 1 : hardware read logic

7.3 Principle operation of incremental encoder



The incremental encoder is used as one of the sensors for cart pendulum system. It transmits the different pulses to recognize the rotation & direction of object loaded on sensor shaft. The incremental encoders work on a non-contact magnetic scanning principle. A diametric magnetized magnet is mounted in the stainless-steel shaft with its backlash-free bearings. If the shaft is rotated, the magnet and the magnetic field rotate with it. This change in the magnetic field is detected and processed by a sensor chip. The evaluation enables signals to be generated that are 90° phase-shifted as well as a zero pulse. The downstream electronics conditions these into high-precision signals and amplifies them into industrially usable square-wave pulses in TTL plus their inverted signals. A incremental encoder has six output signals A, $\bar{A}, B, \bar{B}$, which issue square waves in quadrature when the encoder shaft rotates. The square wave frequency indicates the speed of shaft rotation, whereas the A and B pulses phase relationship indicates the direction of rotation. The inverted version of A and B signals also transmitted to avoid the noise affection while transmitting it to destination.

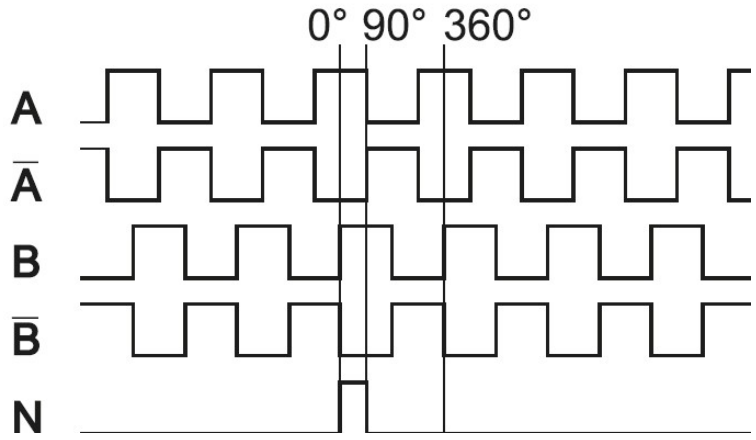


Figure84: Indicates output pulse signals from encoder.

7.3.1 Incremental encoder user IP

The VHDL file consists of the user logic of the Incremental encoder for cart displacement and pendulum angle. The Incremental encoder sensor is embedded in to the DC servo motor for cart displacement measurement and second Incremental encoder is mounted on the cart for pendulum angle measurement. To calculate the linear displacement of cart, the rotational pulses are converted in to displacement with this user logic IP. The following are some important signals interfaces are available for processor and incremental encoder sensor with their description below.

1. **tra & trb:** Each 1-bit signal is used for calculation of magnitude and direction
2. **iclear:** is an input signal to clear the existing latched position of encoder.
3. **sumreg:** is output data indicates the position and angle.
4. **ireset:** provides the input to reset the hardware logic.

```
entity iencspd is
      generic(IECTR_size : integer := 16);
    Port ( iclk : in  STD_LOGIC;
           ireset : in  STD_LOGIC;
           iclear : in  STD_LOGIC;
           tra : in  STD_LOGIC;
           trb : in  STD_LOGIC;
           latchp : in  STD_LOGIC;
           sumreg : out  STD_LOGIC_VECTOR (IECTR_size
           impreg : out  STD_LOGIC_VECTOR (IECTR_size
```

H-file 2: entity for incremental encoder

The above user IP logic is used for measurement of cart displacement and pendulum angle, since both incremental encoder sensors work on same principle.

7.4 Rotary encoder user IP

The VHDL file consists of the user logic for accessing the rotary knob present on spartan board. The application of user logic is used to manually control the position of cart from rotary knob. The following signal interfaces available for processor and rotary knob with their description below.

1. **inc_a & inc_b:** Each 1-bit input signal used to control direction of cart.
2. **clr_pos:** is input signal to clear the existing latched data from rotary knob.
3. **inc_pos:** is output 8-bit data indicates the rotary counts measured by rotary knob.
4. **inc_reset:** provides the input to reset the hardware logic.

```
entity rotenc_8 is
    Port ( inc_a : in  STD_LOGIC;
           inc_b : in  STD_LOGIC;
           inc_latch : in  STD_LOGIC;
           clr_pos : in  STD_LOGIC;
           inc_pos : out  STD_LOGIC_VECTOR (7 
           inc_clk : in  STD_LOGIC;
           inc_reset : in  STD_LOGIC);
```

H-file 3: entity for encoder

7.5 Digital to Analog convertor (DAC)

The Spartan-3E FPGA Starter Kit board includes an SPI-compatible, four-channel, serial Digital-to-Analog Converter (DAC). The DAC device used on Spartan 3E is a Linear Technology LTC2624 quad DAC with 12-bit unsigned resolution. In cart pendulum system the output of controller is in 12-bit digital data which further needs to be converted in to analog voltage. The IC present on Spartan-3E board receives serial input digital data over SPI communication. Therefore, it is necessary to create hardware user logic for SPI protocol to transmit 12-bit digital data to DAC IC.

The DAC present on Spartan board has 4 channels with selected reference voltage. Hence it is possible to get output from all 4 DAC channels simultaneously, However the cart pendulum system uses only 1 DAC channel (DAC-C) with reference voltage 2.5 volts. The voltage equation for DAC outputs C is following.

$$V_{out} = \frac{D[11:0]}{4096} \times (2.5\,V\ \pm 5\%$$

The DAC counts and corresponding voltages on test stand as shown in table 2.

DAC Counts (From user view)	**DAC Counts** (Input to DAC component)	**DAC Voltage (V)** (Input from FPGA to Interfacing board)	**Voltage input at Servo Amplifier (V)**	**Current (A)**
-2048	0	0	-3.75	-15
0	2048	+1.25	0	0
+2047	4095	+2.5	+3.75	+15

Table 4: DAC Counts from Test stand

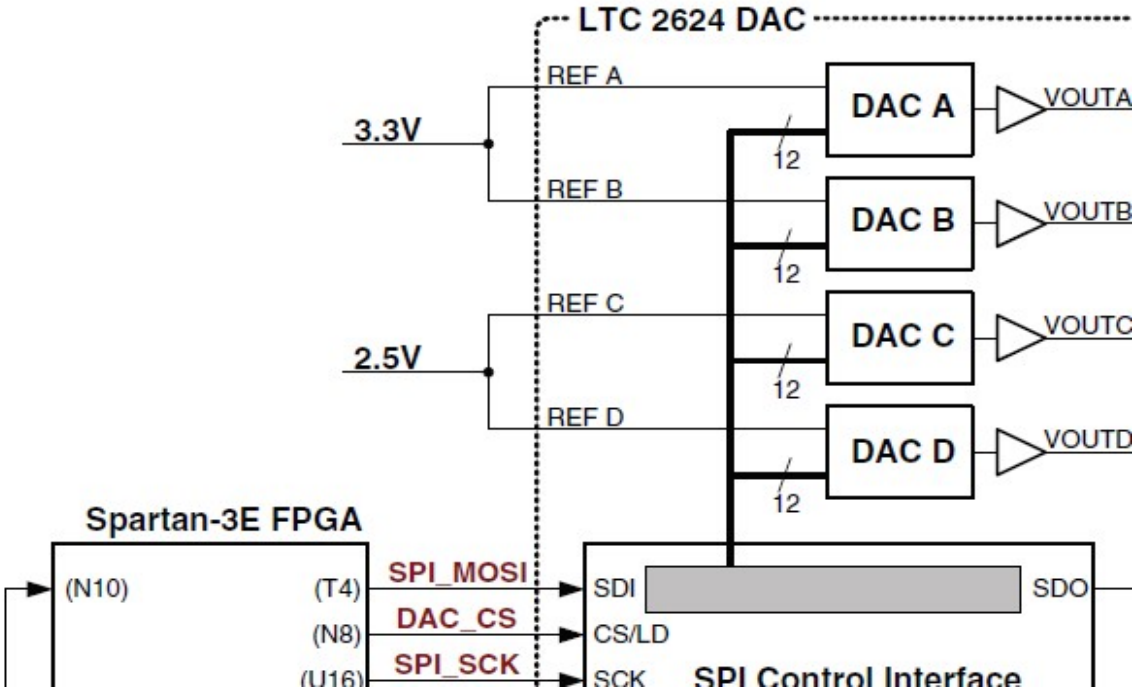


Figure85: Schematic of DAC

7.6 DAC Interface Signals

Following table shows the interface signals between the FPGA and the DAC. The SPI_MOSI, SPI_MISO, and SPI_SCK signals are shared with other devices on the SPI bus. The DAC_CS signal is the active-Low slave select input to the DAC. The DAC_CLR signal is the active-Low, asynchronous reset input to the DAC.

Signal Abbrevation	Direction of Data flow	Description of signal connection
SPI_MOSI	FPGA→DAC	Master Out Slave In used to transmit data from Master to Slave Device.
DAC_CS	FPGA→DAC	Active low chip select signal, SPI communication starts when signal becomes low.
SPI_SCK	FPGA→DAC	SPI clock signal shared between master and slave device.
DAC_CLR	FPGA→DAC	Asynchronous, active-Low reset input
SPI_MISO	DAC→ $FPGA$	Master In Slave Out used to transmit data from Slave to Master Device.

Table 5: Interfacing signals in SPI communication

SPI_MOSI signal is Master out slave in signal, that transmits data from master node to slave device. SPI_CLK is the Clock generated for SPI communication in the FPGA hardware. The SPI clock helps to synchronize the serial transmission between the master & slave devices. SPI_MISO is Master in Slave out signal, that receives the data from slave into the master device.

7.6.1 DAC input word

The DAC input consists of 20-bit information. The data is present in 15 to 4 (D11 to D0) MSB-to-LSB. The DAC channels can be selected by using bits 19 to 16 (A3-A0) called address bits. To be more control on DAC output channels there are 4 bits 23 to 20 (C3-C0) appended with DAC data called command bits. The following table shows the detailed view on address & command bits.

The DAC-C channel is selected by sending data (0010) MSB-LSB on address bits of SPI data. Similarly, all DAC channels can be selected by writing (1111) MSB-LSB in to address bits of SPI data.

Address (MSB-LSB)				DAC Channel
A3	A2	A1	A0	
0	0	0	0	DAC A
0	0	0	1	DAC B
0	0	1	0	DAC C
0	0	1	1	DAC D
1	1	1	1	All DACs

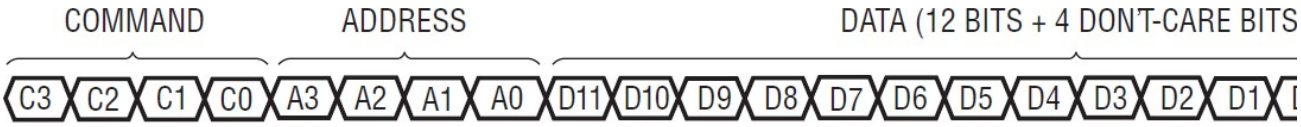
Table 6: DAC channel and Address

The commands are selected to choose operating conditions of the DAC. Each command serves different purpose of the DAC operation. The following table shows the description on the command bits selected by transmission. The first four commands in the table consist of write and update operations. A write operation loads a 16-bit data word from the 32-bit shift register into the input register of the selected DAC, n. An update operation copies the data word from the input register to the DAC register. Once copied into the DAC register, the data word becomes the active 12-bit input code and is converted to an

analog voltage at the DAC output. The update operation also powers up the selected DAC if it had been in power-down mode.

Command (MSB-LSB)				Command Description
C3	C2	C1	C0	
0	0	0	0	Write to input register n
0	0	0	1	Update (Power Up) DAC Register n
0	0	1	0	Write to Input Register n, Update (Power Up) All n
0	0	1	1	Write to and Update (Power Up) n
1	1	0	0	Power Down n
1	1	1	1	No Operation

Table 7: DAC Command



The input word (LTC2624) of the DAC is shown in the following figure. In cart pendulum system the data from SPI master is transmitted serially with 24 bits to slave device. First 4 bits are used as don't care bits & bits from (MSB-LSB) (D11-D0) is used for digital data calculated in PI controller. The last 8 bits are used to select the DAC & command bits.

7.7 SPI Protocol

The Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. The SPI devices communicate in full duplex mode using a master-slave architecture with a single master. The master device originates the frame for reading and writing. The SPI bus can operate with a single master device and with one or more slave devices. The bus is fully static and supports clocks rate up to the maximum of 50 MHz. The slave device is selected by driving the $\overline{CS}$ select signal Low, the FPGA transmits data on the SPI_MOSI signal, MSB first.

LTC2624 DAC uses clock polarity(CPOL- 0) & clock phase(CPHASE- 0) mode. Once the SPI_CS signal goes to low, on the first rising edge of SCK the LTC2624 DAC captures its data from SDI pin of LTC2624 in other words, SPI_MOSI pin of FPGA. Then on the falling edge of SCK, data present on SDI i.e. SPI_MOSI of FPGA is changed with next bit. After transmitting all 24 data bits FPGA completes the SPI bus transaction by returning the $\overline{CS}$ slave select signal high.

The following figure shows the timing diagram of SPI Communication with respect to DAC LTC2624.

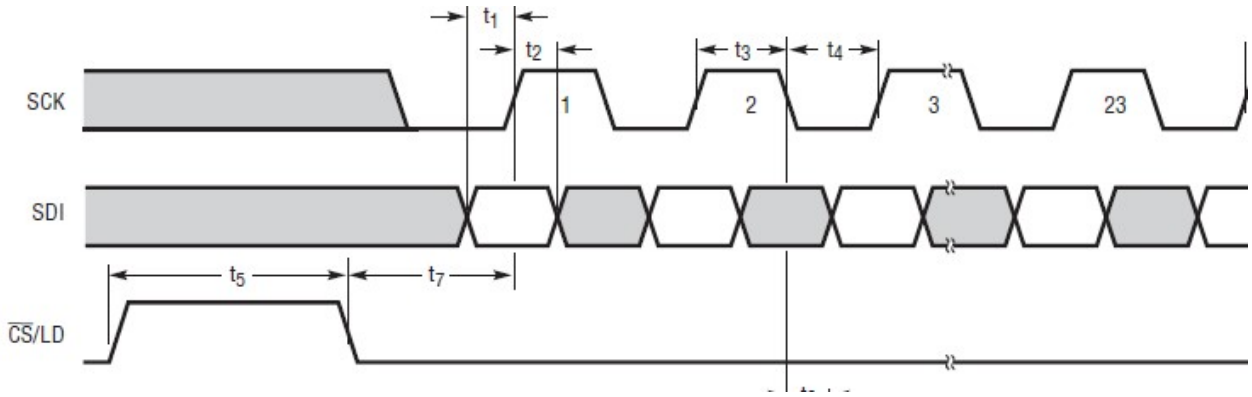


Figure86: Signals of SPI communication

$t_1 = 4ns, t_2 = 4ns, t_3 = 9ns, t_4 = 9ns, t_5 = 10ns, t_6 = 7ns, t_7 = 7ns, t_8 = 20ns, t_9 = 20ns, t_{10} = 7ns.$

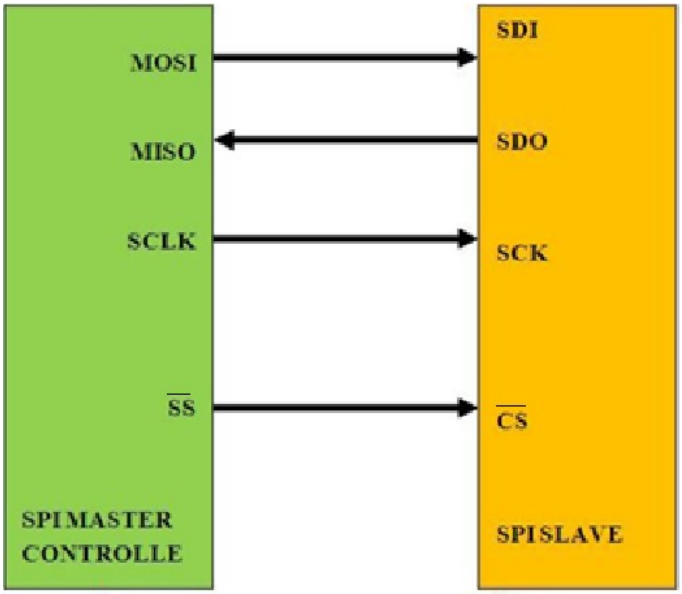


Figure87: SPI Master and Slave Communication Signals

7.7.1 SPI communication user IP for DAC

The VHDL file contains the design of SPI protocol communication. The SPI signals are connected internally from FPGA pins to on board DAC LTC2624 present on the Spartan 3E board. The SPI user IP on FPGA design acts the master device, however the DAC device acts as slave in SPI communication. The design generates its own SPI clock since it is master & clock output is given to other slave devices. The output of cart pendulum system is 24-bit digital data which will be stored in SPI user IP from slave register & transmitted to DAC by SPI serial communication. The d_done bit signal is used to monitor the SPI transmission is completed or not. If the SPI is successfully transmitted the 24 bits the d_done bit becomes logic 1 to indicate SPI is transmitted successfully & ready for next byte to transfer. The following entity figure shows the interfacing signals used for design SPI user logic IP.

1. **sysclk:** It is main system clock from the FPGA.
2. **sreset:** The 1-bit signal used to reset the SPI state machine.
3. **spistart:** The 1-bit signal used to start the SPI transmission.
4. **spdin:** The 24-bit input signal used to store the data needs to be transmitted over SPI.
5. **sck:** It is SPI clock signal generated from SPI Master.
6. **sdo:** It is 1-bit output serial data signal connected to SDI of DAC device.
7. **spidone:** It is 1-bit output signal to indicate SPI transmission is finished & ready for next transmission.

```
entity spis24 is
    Port ( sysclk : in  STD_LOGIC;
           sreset : in STD_LOGIC;
         spistart : in  STD_LOGIC;
           spdin : in  STD_LOGIC_VECTOR (23
           sck : out STD_LOGIC;
           sdo : out STD_LOGIC;
         spidone : out  STD_LOGIC);
```

8 SOFTWARE DESIGN

The tool used for software development, Xilinx® Software Development Kit (SDK) is an Integrated Development Environment (IDE) for development of embedded software applications targeted towards Xilinx embedded processors. SDK is based on the Eclipse open source standard. SDK features include: Feature-rich C/C++ code editor and compilation environment. By application of SDK the software for cart pendulum controller is written in C language. In later stages, the generated code is dumped on the memory available on Spartan board to execute software functions

The main objective of the software code in the cart pendulum system is to design the controller & interface the sensor, actuator & human machine interface (HMI). The controller initially designed & verified with the C-code in Matlab legacy code tool. The software developed in cart pendulum system is bifurcated with following components. The complete code is written in 'C' programming language with help of Xilinx software development kit (SDK).

1. Interface of basic hardware
2. Perform initial reference stage. (Reference Stage)
3. Perform cart pendulum controlling stage. (Control Phase)

The main Function of software design is carried out in the Xilinx Software Development Kit (SDK) and before implementing the logic in SDK we have additionally designed all controller logic and compiled and verified, in MATLAB SIMULINK using the LEGACY CODE TOOL concept.

The user code is fractioned into three C-files.

- Cpmain.c: holds the main loop of the program, and the interrupt service routine.
- Cpcontrl.c: the operational logic of the state machine and the different controllers is implemented.
- Velcontr.c: the function for controlling the Speed of the cart with different controller is implemented.

For performance reasons, and due to higher precision, all calculations are carried out in fixed point arithmetic. The number formats are discussed in detail in the fixed-point model and Legacy model part of this documentation (chapter 4.4).

8.1 Accessing the position counters

From the slave register table, we can see that the slv_reg3 for Cart_Position_counts 20 bit (31 downto 12) and slv_reg4 for Pendulum_angle_counts 16 bits (31 downto 16). So data of the position counters, can easily be accessed by reading in C program of SDK values from the Registers CI_REG(3) (position of the cart) and CI_REG(4) (position of the Pendulum).

To reset of the counter of the CI_REG(3) and CI_REG(4) PosClear() function is used. Function sets the Register CI_REG(3) to IENC_CLR (clears position counters)which makes the reset of the register by FPGA which in turn sets the CI_REG(3) is set to zero for counting from start.

The position of the cart can hold the register with a variable of 24 Bit. Due to the resolution of the encoder and the length of the beam being 1.35m, a movement of the cart from the left end switch to the right end switch results in a 17 Bit number (detailed in chapter 4.4). In order to avoid an overflow, this number needs to be divided by two (or one times shifted to the right), before being assigned to s16 variable.

8.2 Accessing switches and amplifier

From the slave register table, we can see that Switches and Amplifier are available for access in slv_reg(5) and we can use this registers in SDK as CI_REG(5).

The signals are LEFT_SWITCH & RIGHT_SWITCH (29th and 30th bit of slv_reg(5) respectively)for reading the states of the end switches and similarly AMP_OK (31st bit of slv_reg(5))for checking whether the motor driver is ready for operation and AMP_ENA (5th bit of slv_reg(5)) is to enable the motor driver.
The Following function:

```
u8 ReadLimitSw (void)            // Reads the present condition of Left and right
Limit Switch
{
    u32 Switchdata = 0;

    Switchdata = CI_REG(5);              // read the switches
    Switchdata &= 0x00000006;            // mask the other bits
    Switchdata >>= 1;                    // shift to right
/*
    ** Switchdata    : 0 - No limit
    **               : 1 - Right_limit
    **               : 2 - Left_limit
    **               : 3 - Both_limit
    */

    return (u8)Switchdata;
}
```

C-File 6: Accessing and checking limit switches

8.3 LCD Functions

From the slave register table, we know that to Access the LCD from hardware. We can Access by slv_reg(1) for cpc_LCD[0 :6] 7 bits (31 downto 25). By using this slv_reg(1) we can program the LCD.
The LCD can be used as a nice debugging tool if your serial UART is compromised for some other functionality.

We are using the LCD for displaying the starting strings. Which indicates that the system has configured itself and the downloaded program has started executing itself. The below code snippet shows the Function used for LCD functions.

```
/*
void lcdsleep(unsigned int delay)
static void lcdinitinst(void)
static void lcdwriteinst(unsigned long lcd_inst)
```

```
static void lcdwritedata(unsigned long lcd_data)
static void SiLcdInit()
static void SiLCDPrintString(int lpos, char *line) */
```

lcdsleep() The required delay is provided by the function lcdsleep(). As LCD is a slow device it needs some time to execute the given instructions and to print the received data.
lcdinitinst()**:** This function is used for initialization of the LCD and to enable LCD and to enable data for the LCD.
Lcdwriteinst()**:**This function Reads the data and instructions to LCD.
lcdwritedata (): This function writes the data and instructions to LCD.
SiLcdInit(): This function initializes the LCD with required configuration. The LCD is used in 4-bit configuration here and the corresponding instructions are provided to LCD.
SiLCDPrintString(): Generally statements are required to be printed on the LCD rather than characters. This function SiLCDPrintString() prints the passed data starting from the position as specified by the first argument.

Flow Chart for Main Function:

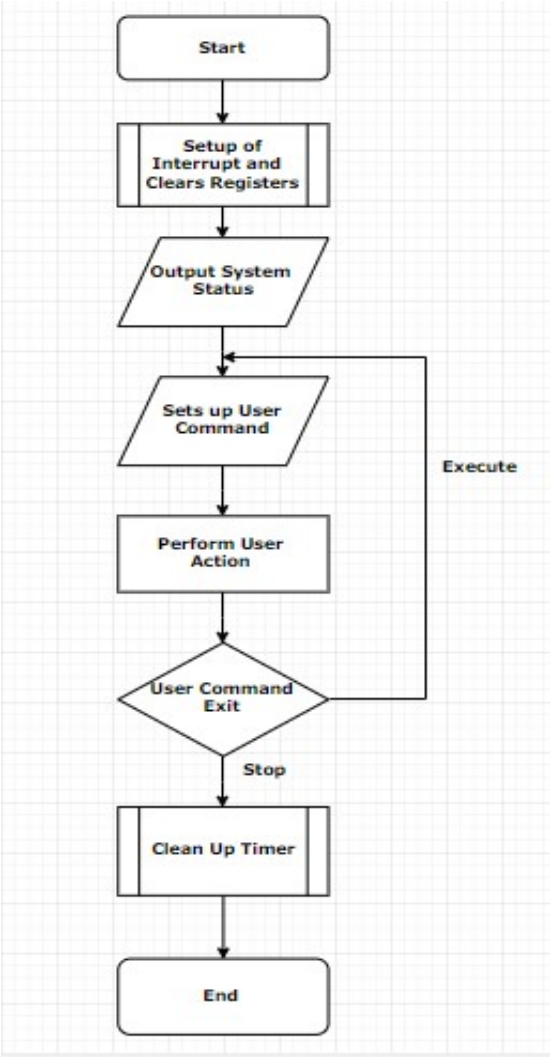


Figure88: Flow Chart for main function

Once the Microcontroller enters the program the main *()* function contained in *Cpmain.c*. It sets the interrupt and initializes all the registers. The main () function runs into an endless loop where it checks for user inputs over RS232 from Java GUI and reacts depending on these inputs. Once the user enters the command "exit" timing critical actions like control actions the function will not be executed and cleans up the system including the interrupt and ends the program.

8.4 UART

This function receives each character serially into the receive buffer of the MicroBlaze. If the receive buffer is not empty, then it copies the receive data into the address the pointer is pointing to.

```
/*
** UART getchar // gives a single character from the  Recieve Buffer
*/
static boolean UARTgetchar(char *ch) {
   char cc = 0;
   boolean  ret_val;

   ret_val = (!XUartLite_IsReceiveEmpty(XPAR_RS232_DCE_BASEADDR));
   if (ret_val) cc = XUartLite_RecvByte(XPAR_RS232_DCE_BASEADDR);
   *ch = cc;
   return ret_val;
}
```

C-File 18: UART Function

8.5 Accessing DAC

The function DACwrite() is used to create the packet of 24-bit data needed to send for DAC. The function *DACwrite() has two arguments "chan" & "ddata",* The *chan* is used with the case statement to select the DAC channel used to be in the system. The cart pendulum system uses the DAC_ADDR_C channel of DAC. The ddata argument is used to pass the data from main function, so that it will be appended in the DAC data packet. The DAC_CMD is type defined with value 0x00300000 which indicates the command "Write to and Update (Power Up) n" is selected. At the end of function DACwrite the whole packet of 24 bit is written in to slave register CI_REG (2) used by SPI user logic IP.

The following code shows how the data along with the address & command of the DAC channel is written to the slave register of DAC.

```
static void DACwrite(int chan, unsigned int ddata) { // To append DAC data, command
                                                        selection of DAC cahnnel.
    int  kc = 0;
    u32  ci_data, xt;

    ci_data = DAC_CS;              // Make chip select active.
    CI_REG(2) = ci_data;           // Update data in the Slave Register of SPI data.
    switch (chan) {          // Select DAC cgannel, Start bit, command bits for DAC.
    case 0:
       ci_data |= SPI_START | DAC_CMD | DAC_ADDR_A | (ddata & 0x0ffff);//Select DAC A
        break;
```

```
    case 1:
       ci_data |= SPI_START | DAC_CMD | DAC_ADDR_B | (ddata & 0x0ffff);//Select DAC B
        break;
    case 2:
       ci_data |= SPI_START | DAC_CMD | DAC_ADDR_C | (ddata & 0x0ffff);//Select DAC C
        break;
    case 3:
       ci_data |= SPI_START | DAC_CMD | DAC_ADDR_D | (ddata & 0x0ffff);//Select DAC D
    }
    CI_REG(2) = ci_data;            // Update data in the Slave Register of SPI data.
    CI_REG(2) = ci_data & (~SPI_START);               // Make SPI start bit to zero.
    do {
        xt = CI_REG(2);
        kc++;
    } while ((!(xt & SPI_DONE)) && (kc < 100));            // Check for SPI done bit.
    CI_REG(2) = 0;  // Clear the SPI data from slave register once it is transmitted.
}
static void DACclear() { // Function to clear all DACs.
CI_REG(2) = DAC_CLR;
CI_REG(2) = DAC_CLR;
CI_REG(2) = 0;
}
```

C-File 7: DAC write function

8.6 Timer Interrupts

The controller for the Cart Pendulum system is a discrete and has been designed to work with a sampling time of 10ms. This requirement in microcontroller is met by using a precise Timer Interrupt. The timer in MicroBlaze is configured for this purpose. The clock frequency of the MicroBlaze is 50Mhz. With no prescaler, the number required to be loaded in the timer to achieve this delay can be calculated based on this clock.

Timer count value calculation, For cart pendulum system, we are choosing a sampling time of $Ts = 10ms$.

$$\begin{aligned} Timer\ value &= clock\ frquency \cdot sampling \\ &= (50 \cdot 10^{6}) \cdot (10 \cdot 10^{-3}) \\ &= 500000 \text{ counts} \end{aligned}$$

The following function CiSetupInterrupt( ) is called from main( ) program to initialize the time interrupt with 10ms sampling time & enable the interrupt. The functions used inside CiSetupInterrupt () is defined in the library "xtmrctr_l.h" file.

```
/* initialize timer, interrupt controller and register handler */
static void CiSetupInterrupt() {
    // disable everything in timer
    XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 0);             // [1]
    // set the load register
    XTmrCtr_SetLoadReg(XPAR_XPS_TIMER_0_BASEADDR, 0, CI_COUNTER_10MS);        // [2]
    // load counter
    XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,                 // [3]
```

```
            XTC_CSR_LOAD_MASK);
    // setup AXI interrupt
    XIntc_RegisterHandler(XPAR_INTC_0_BASEADDR,XPAR_INTC_0_TMRCTR_0_VEC_ID,
                          CITimerInterrupt,(void *)XPAR_XPS_TIMER_0_BASEADDR);// [4]
    XIntc_MasterEnable(XPAR_INTC_0_BASEADDR);                                 // [5]
    XIntc_EnableIntr(XPAR_INTC_0_BASEADDR,XPAR_XPS_TIMER_0_INTERRUPT_MASK);   // [6]
    // start counter
    XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,                 // [7]
            XTC_CSR_ENABLE_TMR_MASK
          | XTC_CSR_AUTO_RELOAD_MASK
          | XTC_CSR_DOWN_COUNT_MASK
          | XTC_CSR_ENABLE_INT_MASK);
    microblaze_enable_interrupts();                                           // [8]
}
```

C-File 8: Interrupt function

Description of functions used for timer interrupt generation are below,

[1] Disable timer interrupt completely to prevent ny signals from timer.

[2] Provide timer count value for the timer load register. Since the timer will count down to zero the sampling time is < timer value> / <clock frequency> as calculated above.

[3] the count register is loaded with value calculated from [2].

[4] Register the handler (the interrupt service routine code that should be called when interrupt occurs).

[5] Enable interrupt controller. [pass the interrupt to MicroBlaze].

[6] Enable the timer interrupt.

[7] Start the timer with required parameters.

[8] This enables the interrupts on MicroBlaze.

Before closing the application of cart pendulum system, it is necessary to disable timer interrupt and make exit of the application. The function static void CiCleanupInterrupt**()**is called at during the exit procedure of application to disable the time interrupts.

```
/* cleanup timer interrupt stuff */
static void CiCleanupInterrupt() {
    microblaze_disable_interrupts();            // [1]
    XIntc_DisableIntr(XPAR_INTC_0_BASEADDR,
            XPAR_XPS_TIMER_0_INTERRUPT_MASK);   // [2]
    XIntc_MasterDisable(XPAR_INTC_0_BASEADDR);  // [3]
    // disable everything in timer
```

```
    XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 0); // [4]
}
```

C-File 9: cleanup timer interrupt

Description of functions used for timer interrupt generation are below,

[1] Disable interrupts on MicroBlaze.

[2] disable the interrupts for the timer on interrupt controller.

[3] Disable all interrupts on interrupt controller.

[4] Stop timer and disable interrupt signal generation.

9 REFERENCING STAGE

A state machine is a concept used in designing computer programs or digital logic. There are two types of state machines: finite and infinite state machines. The former is comprised of a finite number of states, transitions, and actions that can be modeled with flow graphs, where the path of logic can be detected when conditions are met. The latter is not practically used. A state machine is any device storing the status of something at a given time. The status changes based on inputs, providing the resulting output for the implemented changes. A finite state machine has finite internal memory. Input symbols are read in a sequence producing an output feature in the form of a user interface. State machines are represented using state diagrams. The output of a state machine is a function of the input and the current state. State machines play a significant role in areas such as electrical engineering, linguistics, computer science, philosophy, biology, mathematics, and logic. They are best used in the modeling of application behavior, software engineering, design of hardware digital systems, network protocols, compilers, and the study of computation and languages. Hence, we have implemented state machine to achieve the objectives of Anti sway control system.

The Cart Pendulum system when powered up can be at any position on the belt. If the control mode is started directly upon initialization of the system, then cart will migrate to the incorrect position as it doesn't have the correct reference. Upon start, it will consider its current position as zero reference and with respect to this it will make its displacement which is false. The correct zero reference is the center of the belt according to the design. Thus, at start a referencing phase is performed to bring the cart at the center of the belt.

The state diagram for the referencing phase is shown in the figure 89. In this phase the cart is moved left and right to calculate the length of the belt between the two limit switches. And after calculation the cart is moved at the midpoint of the computed distance which becomes zero reference for Cart pendulum control stage. To detect the belt end positions on the test stand two limit switches are installed on the plant. The rotary encoder sensor installed within the motor is needed to count the cart position on the belt & by using reference stage algorithm cart will be stopped at center position to start with controlling stage of pendulum.

9.1 Reference stage state diagram

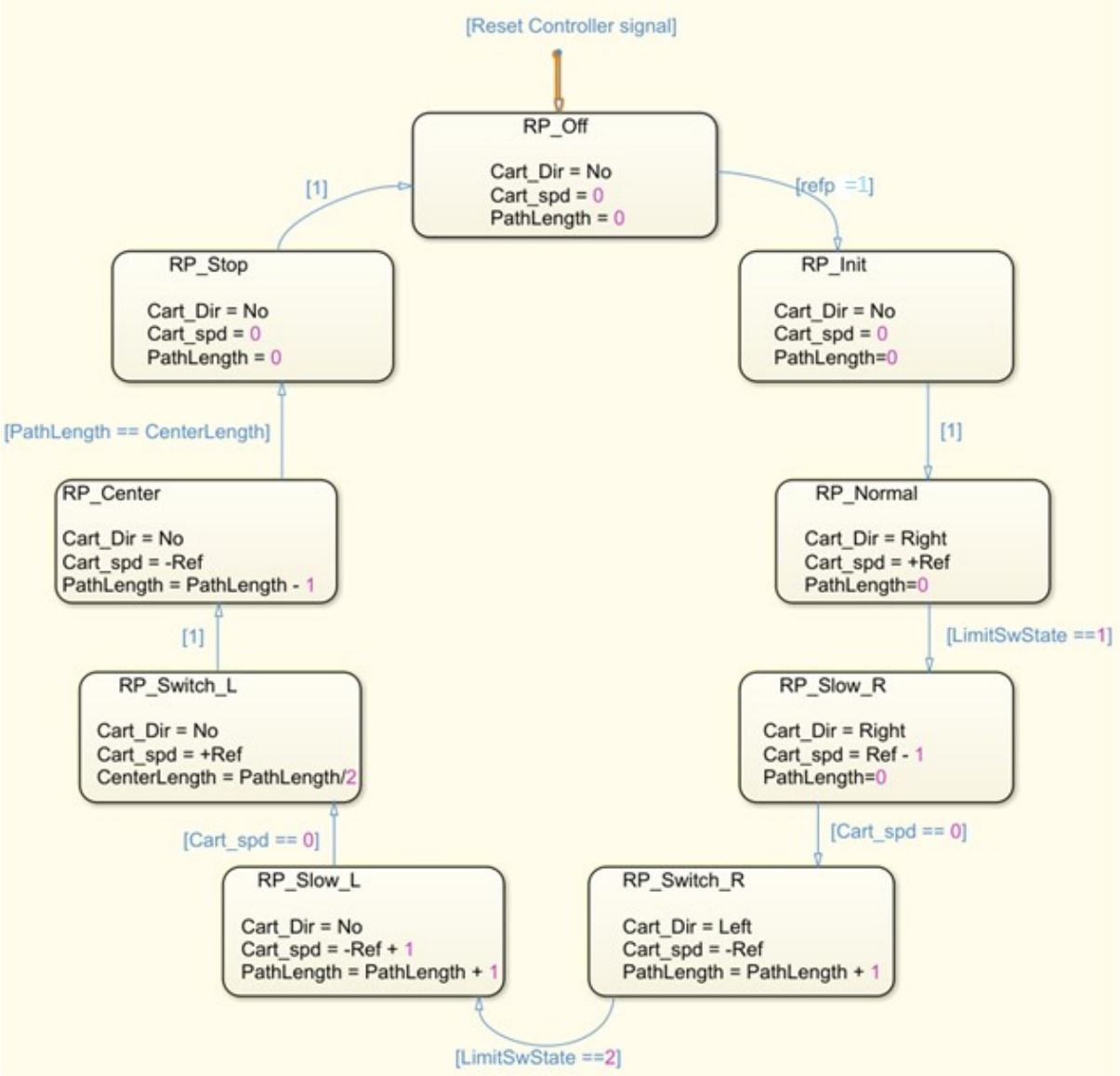


Figure89: Reference Phase State Machine

The state diagram shows the 9 states required to implement the reference stage algorithm. According to same state diagram the C code is implemented within the cart pendulum controller.

When command referencing is given by the GUI, the first step is to load the reference velocity within the PI controller & disable the control stage of pendulum. The c code for reading the limit switches is coded with function u8 ReadLimitSw(void), which returns the limit switch detected on the plant. The function reads the limit switch values from the slave register CI_REG(5) & calculates which switch is detected on plant. The return value 1 indicates the Right limit switch is pressed & value 2 indicates left written switch is pressed. The value 3 gives both limit switch is presses is unlikely not possible unless any human presses both switches manually.

The following C code explains about the initialization of the reference stage algorithm. Firstly, it will reset every data within every variable to clear the path length value & switch detects value. Later it detects the limit switch data to make sure cart is not at extreme position of the belt. If any one of limit

switch is detected, then system recognizes the cart is at end position of the belt & starts algorithm with next state.

For example: Suppose the cart is at right end of belt indicates limit switch data has value 1 in variable LimitSwData. Hence the reference stage algorithm starts with the state RP_Switch_R that refers the limit switch right is already detected. The function also initializes the parameters for velocity control.

```
// Checks the initial condition of Limit_Switch and Initializes the reference phase
s8 RefPhaseInit(void)
{
    s8 LimitSwData = 0;                          // Initilisation of all data to zero.
    PathLength = 0;
    SwitchDetects = 0;
    vel_incr = 0;

    LimitSwData = ReadLimitSw();                 // Function read limit switches.

    switch(LimitSwData){

       case 0 : Ref_State = RP_Normal; break;    // cart is at unknown position.
       case 1 : Ref_State = RP_Switch_R; break; // cart is at the right limit switch.
       case 2 : Ref_State = RP_Normal; break;    // cart is at the left limit switch.
       case 3 : Ref_State = RP_Stop; break;
       default : Ref_State = RP_Stop; break;
    }
    if(Ref_State)
    {
        VelCtrlInit(KP_D,KI_D,LIMIT_INT,LIMIT_OUT); // initialize the velocity PI
                                                    controller
    }

    return Ref_State ;
}
```

C-File 10 Initializing the Reference phase function

9.1.1 Reference stage C code

The reference stage is called at each sample time of 10ms by application of timer interrupt. Once the timer interrupt occurs, the RP_Init is executed to initialize the parameters for velocity controller. Next, state RP_Normal is called which processes normal running of the cart where the velocity is increased incrementally till its maximum reference velocity provided by user. Once the right switch is detected the control reaches to RP_Slow_R state to slow down the cart. Further, the cart direction will be changed with state RP_Switch_R. The calculation of distance starts only after the cart has detected Right limit switch. It calculates the distance by summing the velocity at each sampling time on the basis of the formula:

$$S = \sum_{0}^{k} v_k \cdot t$$

This discrete integration continues until the cart detects the other end of the belt. The program jumps to RP_Slow_L when any Left switch is detected during movement of the cart. Then, the whole path length

is divided with 2 to compute the half length of cart in the RP_Switch_L state. Lastly, the state RP_Center is executed to make cart position at the center of the belt. The care has taken during slowing down the cart when any of limit switches is detected by state RP_Slow_L & RP_Slow_R which provides smooth operation of cart positioning.

The RP_Complete marks the end of the entire referencing Phase. Since the cart is almost at the center, this phase slows down the cart until its reference velocity becomes zero. This brings the cart to halt. This completes the referencing procedure and hence it ends by sending short information to the user that the referencing phase is completed.

A velocity controller which is a basic PI-controller is implemented to slowly increase the velocity of the cart up to the provided reference velocity in either direction. It ensures the constant speed of cart to detect the limit switches without any delay. The reference velocity is varied as a ramp function with constant slope until the maximum velocity is reached. After that the velocity remains constant. According to the mode of operations the reference velocity is increased or decreased with different slopes. For instance, when the limit switch is detected the reference velocity is slowed down rapidly to instantly bring the cart to halt. The function *VelContr()* is implemented to carry out this task as shown below in C-file.

```
// reference stage algorithm implementation.

LimitSwState = ReadLimitSw();       // get the states of both the limit switches
GetStates();                        // Get the controller states i.e. Int_States[k]

        if(Ref_State != RP_Off){  // Check Reference state off condition.
            switch (Ref_State){   // to select state in reference stage algorithm
                case RP_Init:     // Initilisation stage
                RefPhaseInit();   // call reference stage initialization function
                break;
                case RP_Normal:   // Normal state
                    if(LimitSwState){          // Detech limit switch
                        if(LimitSwState == 1){ // Detech Right limit switch
                            Ref_State = RP_Slow_R; // change state to make cart slow
                        }
                    }
                    else{
                    Ref_State = RP_Normal;
                    }
                break;
                case RP_Slow_R:              // Cart slow at right side of belt.
                    Pos_Incr = Pos_Incr >> 1; // Make division to stop cart slowly.
                    if(Pos_Incr == 0){        // Check for zero cart velocity.
                        Ref_State = RP_Switch_R; // Change to right switch detected
                    }
                    else{
                        Ref_State = RP_Slow_R;
                    }
                    break;
                case RP_Switch_R: //State to change direction of cart at right switch
                    if(LimitSwState == 2){     // check left switch detection
                        Ref_State = RP_Slow_L; // switch to make slow movement of
```

```
                                                cart at left.
                    }
                    else{
                        Ref_State = RP_Switch_R;
                    }
                break;
                case RP_Slow_L:
                    Neg_Incr = Neg_Incr && 0x7FFF; // Mask the MSB for Negative sign.
                    Neg_Incr  =  Neg_Incr >> 1; // Make division to stop cart slowly.
                    if(Neg_Incr == 0){              // Check for zero force.
                      Ref_State = RP_Switch_L;      // Change to left switch detected.
                    }
                    else{
                        Ref_State = RP_Slow_L;
                    }
                    break;
                case RP_Switch_L: //State to change direction of cart at left switch
                     Ref_State = RP_Center;
                break;
                case RP_Center:                 // State to make cart in center postion
                if( (PathLength >= (CenterLength-NR_ORIGIN)) && (PathLength <=
(CenterLength+NR_ORIGIN)) ){
                    Ref_State = RP_Stop;      // Check for cart is at center or not.
                }
                break;
                case RP_Stop:                 //  Stop state for reference stage.
                    Int_States[3] = 0;        // Make displacement state to value zero.
                    Ref_State = RP_Off;       // Switch to reference phase off state.
                break;
                default :
                    Ref_State = RP_Stop;      // Make off state in default condition.
                break;
             }

            VelContr(Ref_State); // Control the velocity as required by the stage of
                                 operation
            }
```

C-File 11: Reference stage algorithm implementation in C code.

```
// To compute the velocity of cart for left and right movement with PI controller

void VelContr(u8 control_state) // Function to control the speed & direction of cart.
{
    if (control_state != RP_Off) {
        if(control_state == RP_Normal){
            vel_incr = PosRef(ref_vel); // Increase cart velocity slowly towards
                                        right direction.
        }
        else if(control_state == RP_Switch_R){
            PathLength +=Int_States[2]; // start counting of displacement of cart
                                        position.
            vel_incr = NegRef(ref_vel); // Increase cart velocity slowly to left
                                        direction.
        }
        else if(control_state == RP_Slow_R){
            vel_incr = Pos_Incr;         // make slow stopping of cart at right side.
        }
```

```
        else if(control_state == RP_Slow_L){
            vel_incr = Neg_Incr;          // make slow stopping of cart at left side.
        }
        else if(control_state == RP_Switch_L){
            CenterLength = PathLength>>1; // divide the measured length of belt for
                                             center postion.
        }
        else if(control_state == RP_Center){
            PathLength +=Int_States[2]; // Measure the displacment of cart to make at
                                           center position.
            vel_incr = PosRef(ref_vel); // Increase cart velocity slowly to right
                                           direction.
        }
        else if(control_state == RP_Stop){
            vel_incr = 0;               // Make velocity of cart to zero.
        }
        uu = VelContrCalc(vel_incr, Int_States[2]);// Compute the velocity of cart
                                                      from PI controller.
    }
    else{
        uu = 0;              // Make the cart not to move.
    }
}

DACwrite(3,(uu+ZERO_VOLT)<<4);    // Write the computed force values to DAC


static s16 PosRef(s16 ref_vel){ // Function to increase cart velocity slowly in right
                                   direction
        if(Pos_Incr<ref_vel)
        {
            Pos_Incr = Pos_Incr + 2; // Increment velocity of cart slowly to right
                                        direction.
        }
    return Pos_Incr;
}

 static s16 NegRef(s16 ref_vel){ // Function to increase cart velocity slowly in left
                                    direction
        if(Neg_Incr>(-ref_vel))
        {
            Neg_Incr = Neg_Incr - 2;// Increment velocity of cart slowly to left
                                       direction.
        }
    return Neg_Incr;
}
```

C-File 12: Reference stage function for velocity control of cart.

The required force needed to move cart is stored in terms of digital values in variable “uu”. The DACwrite() function used to append DAC controlled bits with data avalaible in “uu”.

9.1.2 Velocity PI controller

The *VelContrCalc()* function calculates the required input force based on the error between the reference velocity and current velocity of the cart. It is a normal discrete fixed-point PI-controller written as a piece of C-code. The following code refers to the PI controller implemented for the velocity control.

```
//PI control for velocity control
s16 VelContrCalc(s16 reff, s16 out_fbb)
{
    s32 u_ctr;
    s16 ctr_e;

    ctr_e = reff - out_fbb ;                    // Calculate the error

    u_ctr = RefPhase.kp * ctr_e + ( RefPhase.xi >> (KI_SHIFT - KP_SHIFT));

    if (u_ctr > RefPhase.limit_out )
    {
        u_ctr = RefPhase.limit_out            // limiter for PI controller
    }
    else if (u_ctr < -RefPhase.limit_out )
    {

        u_ctr = -RefPhase.limit_out;         // limiter for PI controller
    }

    u_ctr >>= KP_SHIFT;

    RefPhase.xi += (RefPhase.ki * ctr_e);     // Integrator within PI controller

    if ( RefPhase.xi > RefPhase.limit_int )   // limiter for integrator
    {
        RefPhase.xi = RefPhase.limit_int;
    }
    else if ( RefPhase.xi < -RefPhase.limit_int ) // limiter for integrator

    {
        RefPhase.xi = -RefPhase.limit_int;
    }

    return (s16)u_ctr;                          // Return velocity from PI control
}
```

C-File 13: PI control for velocity control

10 STATE MACHINE FOR CP PHASE

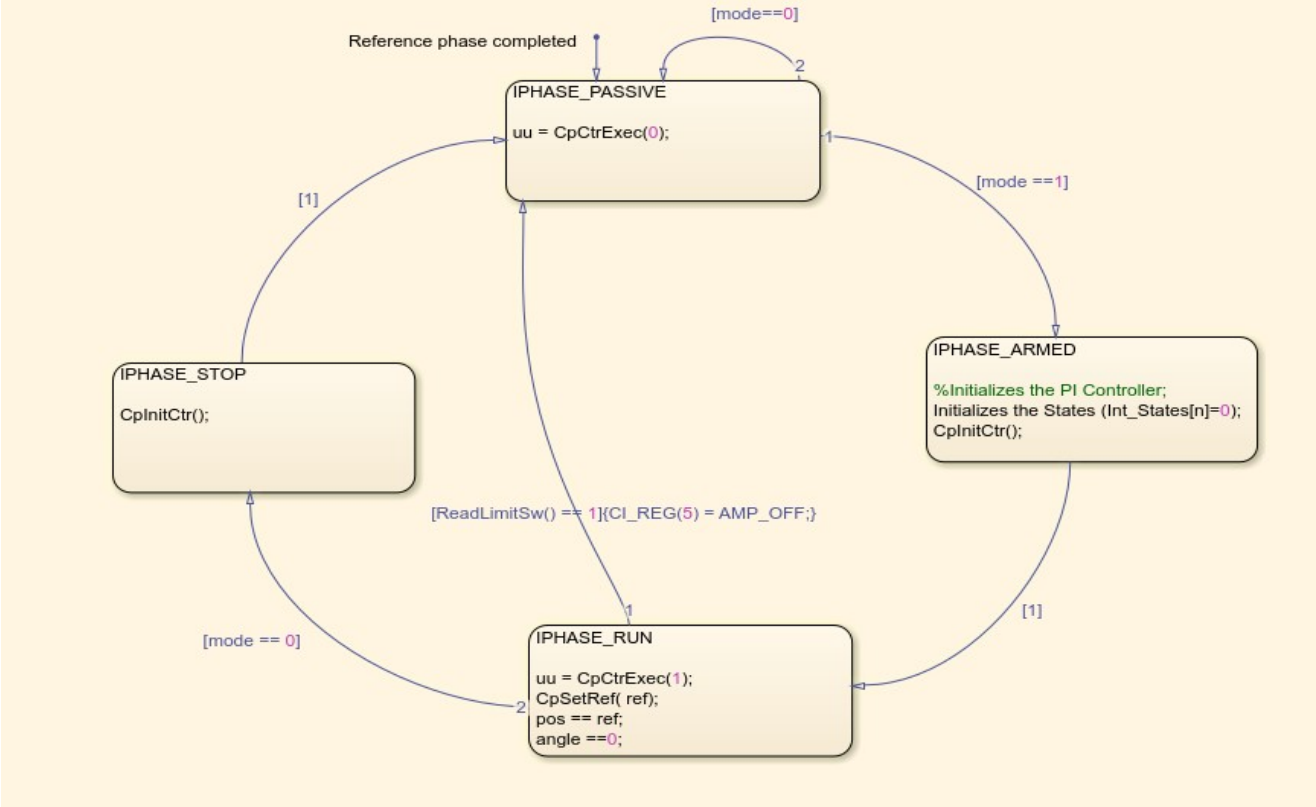


Figure 90: State Machine for CP phase

The main objective of anti-sway control system is to control the swinging of the pendulum and move the cart to desired position without (or minimum) swinging of pendulum. To get these objectives, controller need to read all the states of the system and computing the controller calculations for the manipulating variable (force). The state machine shown in figure 90 will discuss all about the functions written for CP phase.

The state machine for CP_Phase is activated. As per reference phase algorithm, once reference phase is completed the next state moved to passive state of CP phase. So, the default condition for Cp_phase in IPHASE_PASSIVE. In IPHASE_PASSIVE we have the following action the output for the System is Zero (i.e. the force uu=CpCtrExec (0)).

The CP phase controller gets into action, when it will get “mode on “command from GUI. When the mode is on enter to IPHASE_ARMED state. IN IPHASE_ARMED state we will have all the required data like states of the system made (Int_States ==0) and Initialization of Controller with gain values.

The controller needs output feedback to calculate the error and to make the appropriate control action. The output feedback is received from the incremental encoder values for every sampling time.
The following function reads the state:

```
void GetStates(void)                    //  To calculate current velocity of cart
```

```
{
    PosLatch();                    // Latch the position and angle encoder values
    PastPosCount = PresPosCount;
    PresPosCount = CI_REG(3);
    Int_States[2] = (s16)(PresPosCount - PastPosCount);     // Cart velocity
    Int_States[3] += (Int_States[2]);                  // Cart Displacement/2 add if
problem with cp phase
    PastPendCount = PresPendCount;
    PresPendCount = CI_REG(4);
    Int_States[0] = (s16)(PastPendCount - PresPendCount);   // pend_vel
    Int_States[1] += Int_States[0];                         // pend_angle
}
```

C-File 13: To reads the state of the System

During the next execution of the ISR, velocities can be calculated as the difference between current position and last position. The sensor values are stored in a four field s16 array called Int_States[k].

So, now we are in IPHASE_ARMED state with all the state of the system required at this point initialize the controller for system. The entire code concerning the controller is implemented in file 'cpcontrol.c' and this file also contains several functions to log the systems states and controller outputs, and also hosts the code of the state machine that controls the different modes of operation of the program.

This function call initializes the PI controller:

```
//PI control Initializations
void CpContrInit(void)
{
    Cp_PiInit(CP_KP_FIX, CP_KI_FIX, CP_LIMIT_INT, CP_LIMIT_OUT);   // Initialize
                                                                    the PI controller
}
```

C-File 14: PI control Initializations

This function in turn calls the Initialize the PI controller. The value of the PI controller is saved in 'cpcontrl.h' file.

```
#define N_STATES 4
#define CP_KP_FIX 878           // int_16.15        All poles at (-4)
#define CP_KI_FIX 32            // int_16.15
#define CP_KP_SHIFT  15
#define CP_KI_SHIFT  15
#define CP_LIMIT_INT   98304000 //int_32.15(integral value limited to 3000*2^15)
#define CP_LIMIT_OUT   98304000 //int_32.15(PI controller output value limited
                                   to 3000 *2^15)
#define KD_SHIFT 12             // int_16.12
#define LIMIT_POS  903299  //int_32.12 format(limiting the force to 10*F_i/F_0*2^12)
#define LIMIT_NEG   -903299   // int_32.12
```

```
typedef struct {
    s16 kp;
    s16 ki;
    s32 limit_out;
    s32 limit_int;
    s32 xi;
} PIObj_struct;
```

C-File 15: cpcontrl.h

After Initialization of the controller, the user can give the reference position from the GUI and that value will read in below function.

```
/*
 *  Set the reference value for the controller
 */
void CpSetRef(s16 ref) {
    pos_ref = ref;
}
```

C-File 16: Function to set the reference position

If the cart is moving under reference phase, the CP run phase goes to passive state. So, cmode ==0.
If cmode ==1, then the system will run under CP phase and will make the appropriate control action.

```
/*Starts the controller for cart-pendulum system ;
   if c_mode = 0  (i.e I_PHASE = IPHASE_PASSIVE) then its turned off
 */
s16 CpCtrExec(int cmode) {

    s16  uu;

    uu = CpContrCalc(pos_ref);  // Calculate the manipulating variable according
                                   to the Reference value
    if (Logger_Active == 1)     // If log is ON
    {
        CpPushRecord(uu);       // Then push the data in the ring buffer
    }

    if (cmode == 0)
    {
        uu = 0;       // If the INTERRUPT_PHASE is '0',then stop the controller
    }
    return uu;
}
```

C-File 16: Controller Action function

Every time the Microblaze receives the string '.cp', it switches the controller to active mode and the controller starts taking the control action execute the control action(to control the swing of the pendulum and to move to desired distance)this below function is called

```
/* State feedback and  PI controller for cart-pendulum system calculation*/
s16 CpContrCalc(s16 refval)
{
    s32 u_pi=0;
```

```
    s32 u_sfb=0;
    u8 k;
    u_pi = Cp_PiCntrlC(refval, Int_States[3]);  // PI_controller calculated
                                                   output in format 32.KP_SHIFT;

    u_pi >>= CP_KP_SHIFT-KD_SHIFT;   // Its necessary to make the fractional
                                        digits same int_32.12

    for (k = 0; k < N_STATES; k++)     // State feedback calculation ( K*X ) ;
    {
        u_sfb += (KD[k] * Int_States[k]);
    }

    u_sfb = u_pi - u_sfb;      // state feedback controller output (int_32.12)

    if (u_sfb > LIMIT_POS )    // Check if the force is within the range
    {
        u_sfb = LIMIT_POS;     // Limiting the state feedback controller output
    }
    else if (u_sfb < LIMIT_NEG )
    {
        u_sfb = LIMIT_NEG;
    }

    u_sfb >>= KD_SHIFT;      // Shifting the result value further 12 bits to fit
                                the value in DAC range (int_32.24)
    return (s16) u_sfb;
}
```

C-File 16: State feedback function

```
/* PI controller for cart-pendulum system calculation;*/
Cp_PiCntrlC():

s32 Cp_PiCntrlC(s16 reff, s16 out_fbb)
{
    s32 u_ctr;
    s16 ctr_e;

    ctr_e = (s16)(reff - out_fbb) ;     // Calculate the error

    u_ctr = CpPhase.kp * ctr_e + (CpPhase.xi >> (CP_KI_SHIFT - CP_KP_SHIFT));
//calculating the PI controller output

    if (u_ctr > CpPhase.limit_out )      // limiting the PI controller output value
    {
        u_ctr = CpPhase.limit_out;
    }
    else if (u_ctr < -CpPhase.limit_out )
    {
        u_ctr = -CpPhase.limit_out;
    }
```

```
    CpPhase.xi += (CpPhase.ki * ctr_e) ;       // calculating the integral values

    if ( CpPhase.xi > CpPhase.limit_int )              // Limit on integral values
    {
       CpPhase.xi = CpPhase.limit_int;
    }
    else if ( CpPhase.xi < -CpPhase.limit_int )
    {
       CpPhase.xi = -CpPhase.limit_int;
    }

    return u_ctr;
}
```

C-File 17: PI control function for Cp_phase

From the above code, the limiters are being introduced in controller algorithm to operate the system in safety region without loosing its stability. The limiter values are listed in table 8.

S. No.	Limiters	Fixed point format	Limit value	Remarks
1	LIMIT_INT	$int\_32.15$	90112000	Limiter on integrator ($2750 \cdot 2^{15}$)
2	LIMIT_OUT	$int\_32.15$	90112000	Limiter on PI output ($2750 \cdot 2^{15}$)
3	LIMIT_POS	$int\_32.12$	903299	Limiter on state feedback controller output ($(10 \cdot \frac{F\_i}{F\_0}) \cdot 2^{15}$)
4	LIMIT_NEG	$int\_32.12$	903299	

Table 8 : Limter values for controller outputs

11 CART PENDULUM SYSTEM GUI

11.1 Design

The major benefit of a GUI is that systems using one are accessible to people of all levels of knowledge, from an absolute beginner to an advanced developer or other tech-savvy individuals. In the cart pendulum system, GUI is designed to perform a velocity and position control to move the cart to a specific position, a string of instructions have to be fed to the processor which in turn activates the corresponding actuator to perform specified command.

Without GUI this task would be very tedious as one has to write list of instructions repeatedly. So, the basic impression behind the development of GUI is to ease the usage by creating an Interface to access and provide commands to the processor to perform certain actions. The interface for the cart pendulum system is shown in the figure 91.

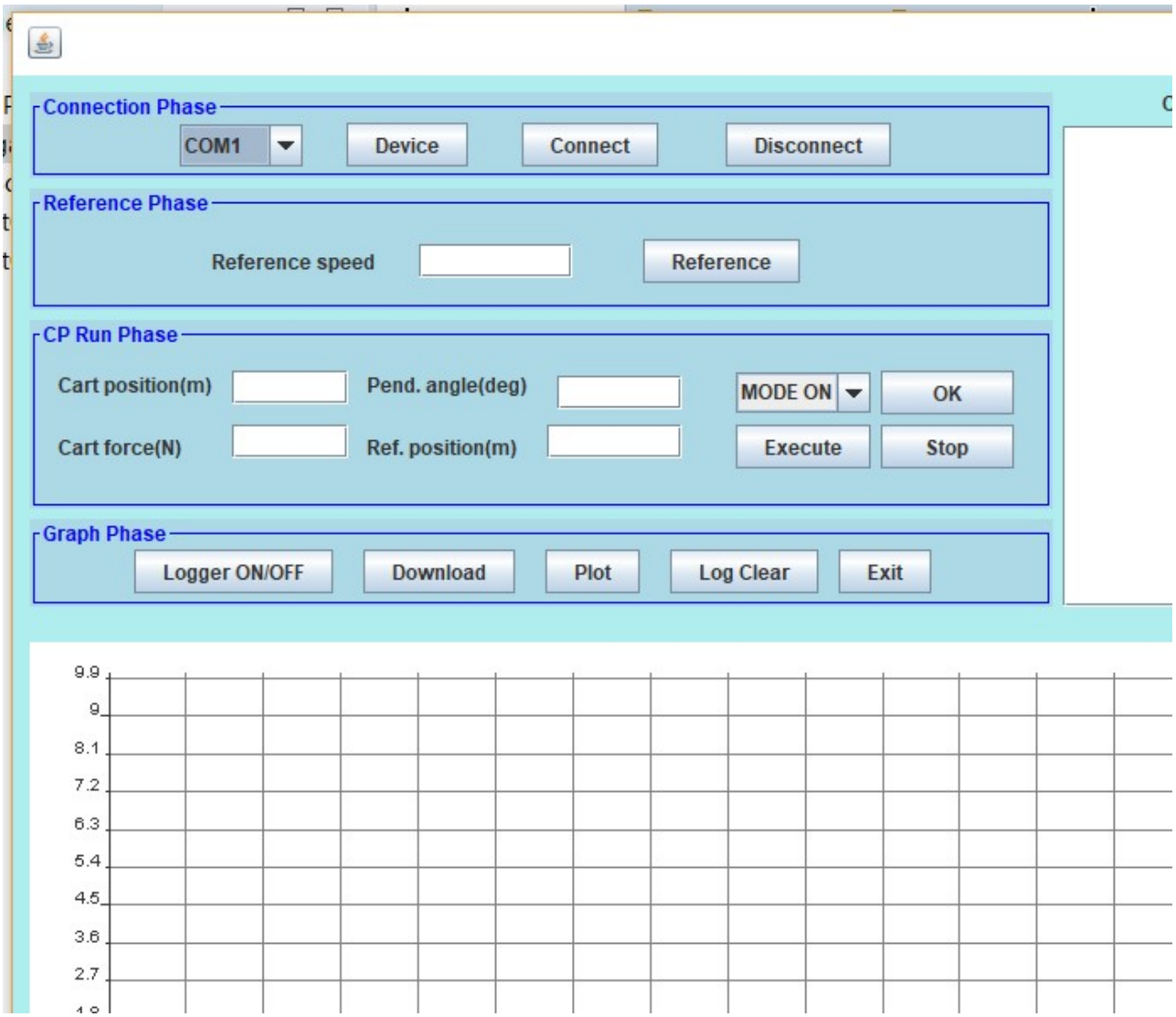


Figure91: Graphical User Interface

GUI is designed and coded using Java to control from the PC. The FPGA is connected by a serial line at

115200 bit/s. Since native support for serial lines was officially dropped for JAVA, the RTXTXcomm(gnu.io) was used to establish serial connections at any desired baud rate.

The User Interface consists of different panels to provide input to the system and display output from the system. It also includes buttons, labels and text boxes. Each button performs different functions on the FPGA. Upon activation of any of the button, the command is sent to the FPGA through the serial communication interface and the FPGA performs the set of instruction stored under it.

11.2 Implementation and Code description

The GUI Java package consists of two classes namely FpgaCtrlF and SerialNetw.

11.2.1 FpgaCtrlF.java

The class FpgaCtrlF deals with the GUI design form designed with Windows Builder Editor and internal and external commands.

11.2.2 SerialNetw.java

Serial communication is the process of sending data one bit at a time, sequentially, over a communication channel or computer bus. Here we are using CommPortIdentfier class to perform the serial connection. CommPortIdentifier is the central class for controlling access to communications ports. It includes methods for:

- Determining the communications ports made available by the driver.
- Opening communications ports for I/O operations.
- Determining port ownership.
- Resolving port ownership contention.
- Managing events that indicate changes in port ownership status.

Any application would first use methods in *CommPortIdentifier* to negotiate with the driver to discover which communication ports are available and then select a port for opening. Here a specific port named *PORT_SERIAL* is checked to make the connection.
Apart from port connectivity, the class also consists of subclasses for receiving and transmitting strings serially to the FPGA.

11.3 Input and Output on the GUI

This section consists of explanation for each phase. As we know that for any serial communication, a COM port has to be selected to send data serially to the FPGA. Here in the first phase, we look for the available devices and establish a connection to it. The following are the buttons with their functionalities:

COM1/COM2 – To select desired COM port.
Device – It lists available COM port or displays information about connected port.
Connect – To connect the PC system to FPGA. E.g. conn COM1.
Disconnect – To disconnect from connected serial port.

Figure 92: Connection Phase panel

11.4 Reference phase

After the connection to the device is successfully established, it is necessary to initialize the system. This is done to make sure the cart is placed at the desired initial position no matter where it was placed before. To do so as the system is powered on, reference mode has to be performed. And this function is implemented with button called "Reference" placed under the panel "Reference Phase".

Figure 93: Reference Phase panel

Upon execution of "Reference" button, the GUI sends the string '*.ref*' to the FPGA. The execution of the CP mode is not allowed before performing the reference mode. The user is restricted to do this by displaying the following message.

11.5 CP phase

This phase of the GUI provides an interface specifically for the CP phase where the control of the system takes place. As shown in the screenshot, it has buttons as discussed below.

- Cart position, Ref. position, Pend. angle, Cart force:

When the CP phase is being performed, states of the Cart Pendulum system are updated in real time in these textboxes. The brackets indicate the units of those physical quantities. The update of the field boxes stops if the CP MODE is OFF. All values sent serially by the processor is in fixed point format. These values are converted at the back end to corresponding physical units.

- Execute:

This button primarily locks the reference value in the 'Ref (m):' field and sends it to the processor in the format '*.ref COUNTS*' where the COUNTS represents the fixed point value of the given reference value in metres.

- MODE ON/OFF:

This is a combo box to select Mode ON/ Mode OFF for the system. The selection of the 'MODE ON/OFF' combo box essentially sends '*.mode on*' or '*.mode off*' strings respectively to the processor. To perform the cart pendulum control, the combo box MODE ON/OFF must be pressed before performing "Execute" placed in the panel "CP Run Phase".

- STOP:

This button sends '.exit' string to the FPGA.

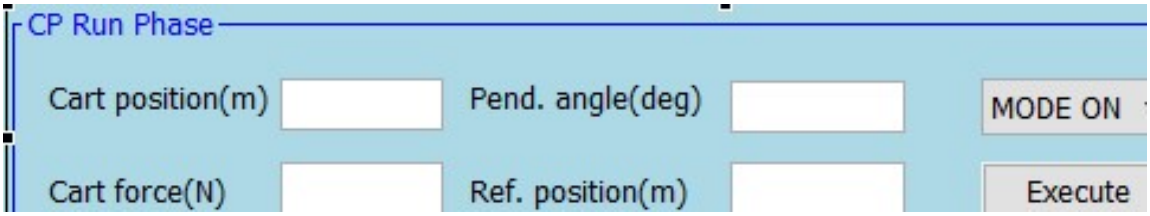


Figure 94: CP Run Phase panel

11.6 Graph Phase

In this panel, following buttons are used to plot the graph. If command starts with ".", followed by command string then such commands are sent serially to the processor by the GUI. The response of the processor then can be seen on the scroll pane of the GUI as text.

- Logger ON/OFF

This button is toggle button, when button is selected the values of cart position, force applied and angle of the pendulum are been recorded by ring buffer implemented on FPGA. This is done to record the data points into the ring buffer. And when the button is deselected, the recording of next data would be stop and ring buffer will hold values it has already recorded.

- Log clear

On execution of this button the data recorded in the ring buffer is cleared i.e. the value of the registers holding the cart position and pendulum angle Force and velocity. This is basically done to flush the previous data from the ring buffer before downloading new data.

- Download

This command loads data from the ring buffer into an array in GUI which is later used for plotting.

- Plot

When plot button is executed graph will be plotted provided data points have been downloaded into the GUI.

- Exit

This button is used to terminate the GUI.

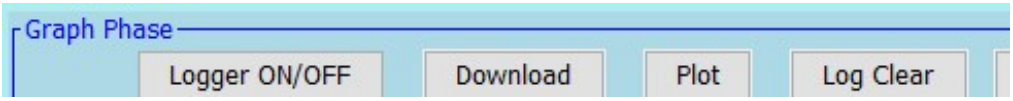


Figure 95: Graph phase panel

11.7 Scroll Pane

The scroll pane acts as a display where the progress of the system is displayed and also error message

will be printed if wrong input is given. The below figure shows the default scroll pane when the GUI is launched to provide user with some help in case the commands are not known.

Figure 96: Scroll pane of the GUI

11.8 Graph Panel

The Graph Panel is provided as an aid to perform analysis on the output of the system. Upon the execution of the command 'PLOT' the GUI takes the data point and plots it. As seen in the below figure only three physical quantities are plotted as they are the most important. Reference position is also plotted to know the steady state error of the system.

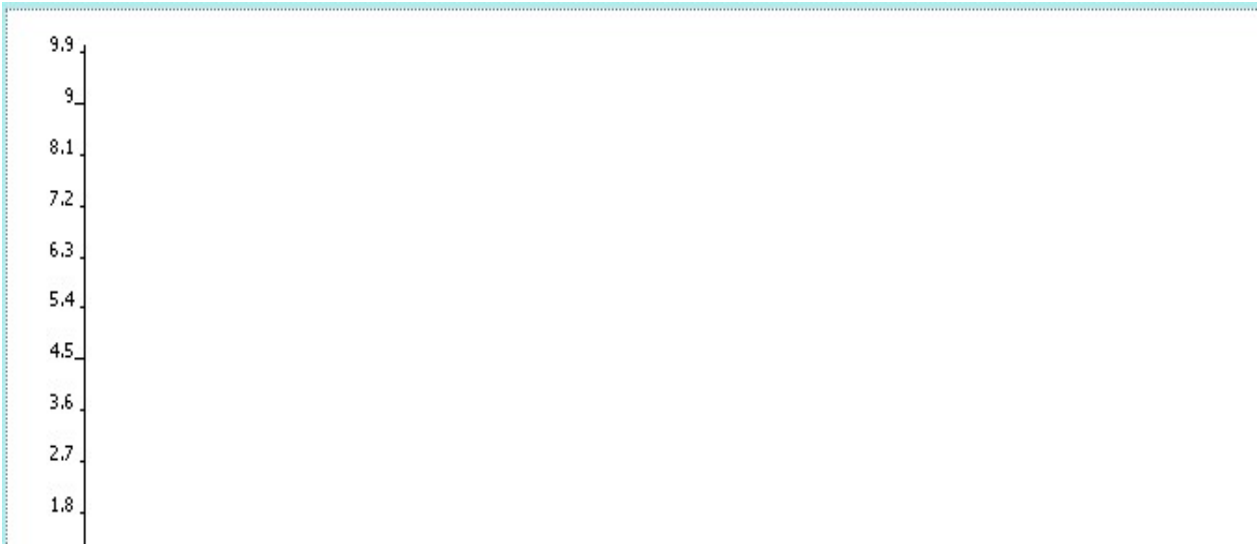


Figure 97: Graph panel

12 TESTING AND ANALYSIS

Testing is used at key checkpoints in the overall process to determine whether objectives are being met

by evaluating the application's behavior, performance, robustness against expected criteria. In this process a set of specially selected inputs are applied to the system and system behavior is observed. Several tests were performed to know the performance capabilities of cart pendulum system in real time.

Here GUI is implemented in such a way that the user is not only providing the input to the system but also plotting the behaviour of the system viz. cart position, pendulum angle and the manipulating variable (Force). The following two test cases are performed on the cart pendulum system. The referencing phase is carried out before running the test cases. Both the test cases are performed for the CP phase.

12.1 Test case 1 with reference of 0.5 meters

12.1.1 Reference position versus cart displacement

The graph in the figure 98 shows the reference versus displacement with the reference position of 0.5 meters. The controller action is visible by the S- curve in the graph. The behavior of cart displacement matches to the simulated result. If we observe the beginning portion of cart displacement curve, the reference position started at 0.6 sec, but cart displacement started at 0.7 sec due to static friction between cart and sledge.

The steady state error in the simulation is $-\ 0.3\ mm$ for the reference input provided.

$$steady\ state\ error\ = \frac{(reference\ position - cart\ displacement)}{encoder\ value\ for\ 1\ m} \cdot\ 1000\quad mm$$

$$= \frac{(28619 - 28639)}{57238} \cdot\ 1000$$

$$= -0.3\ mm$$

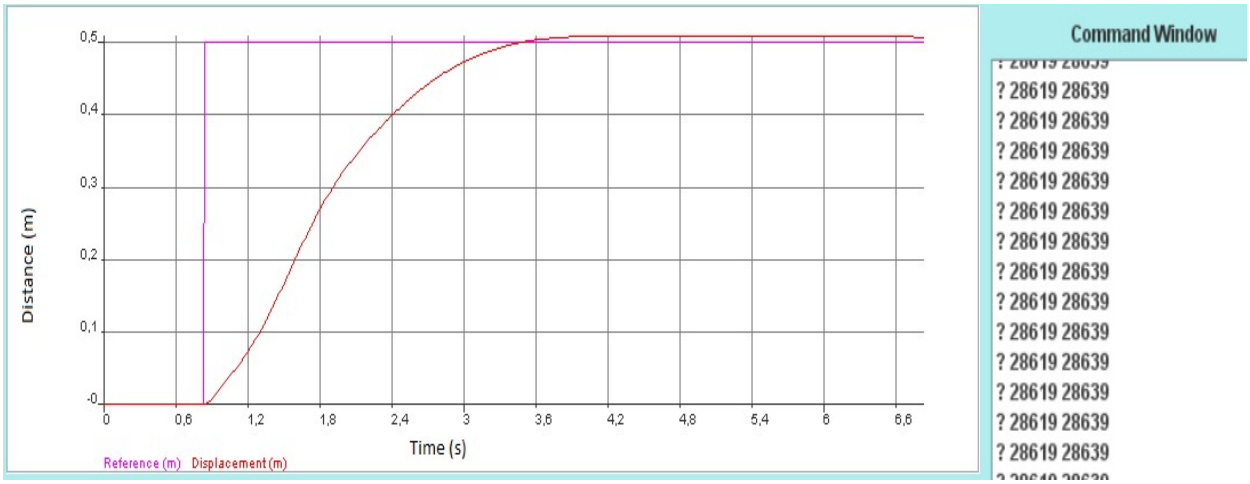


Figure 98: Cart displacement response on test stand

The steady state error(0.03%) is due to the precision loss in gain values, as the control error is very small and the system does not have enough fractional bits to represent that error, then it gets rounded off to absolute zero. Hence, the controller does not realize any error and it will not perform any control

action.

12.1.2 Angular displacement

The figure 99 shows the angular displacement with -6 degrees as the maximum swing of the pendulum. The state feedback controller tries to damp the oscillations while moving the cart from 0 to the desired reference position $(0.5\ m)$ as seen in the figure.99. The behavior of the controller action is same as simulated result.

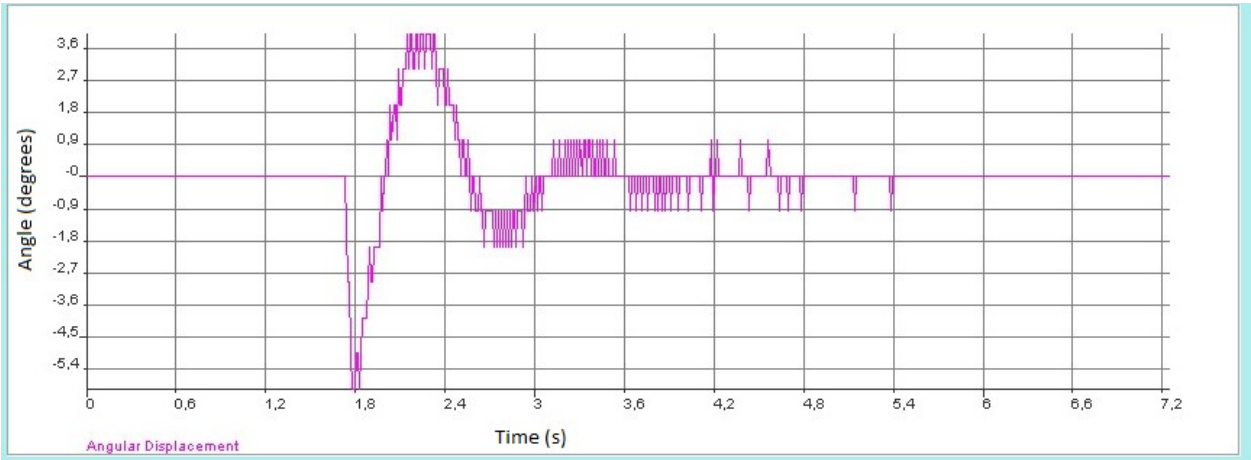


Figure 99: Angular displacement response on test stand

12.1.3 Manipulating variable (Force)

The figure 100 shows the graphical view of the manipulating variable (force). In the time period between 0 to 1.4 s, the force is applied on the cart due to the steady state error obtained in the reference phase. As the graph shows, the force never becomes zero due to the steady state error in the cart displacement. Initially more force $(10\ N)$ is required to overcome the static friction. The behavior of the manipulating variable is same as the simulated results.

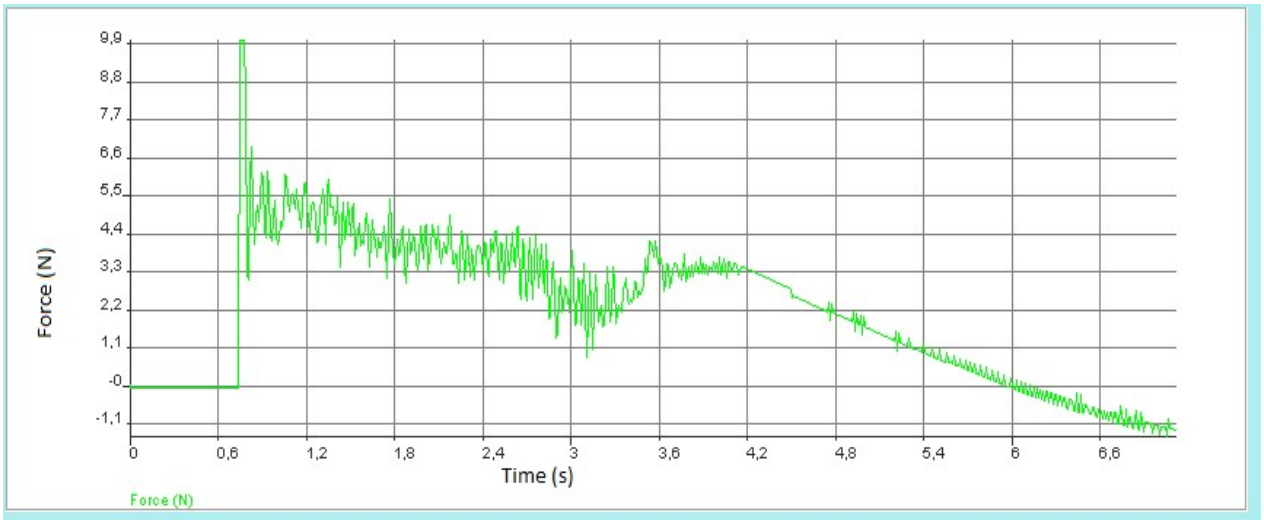


Figure100: Manipulating variable (Force (N)) response on test stand

12.2 Test case 2 with external disturbance

12.2.1 External disturbance to the pendulum angular displacement

In this test case the stability of the pendulum is disturbed by introducing an oscillation to the pendulum. With this test scenario the response of the controller to an external disturbance on the system is verified.

As seen in the figure 101 and figure 102, a disturbance of -45 and -60 degrees are given to the pendulum separately. This disturbs the stability of the system. And hence the State feedback controller tries to correct it. To damp this oscillation the controller causes a displacement of the cart as seen in plot of displacement. As the oscillation of the pendulum comes under control, the controller again tries to match the reference value. The control action is all carried out by the manipulating variable which is shown in the last plots of the figure 103.

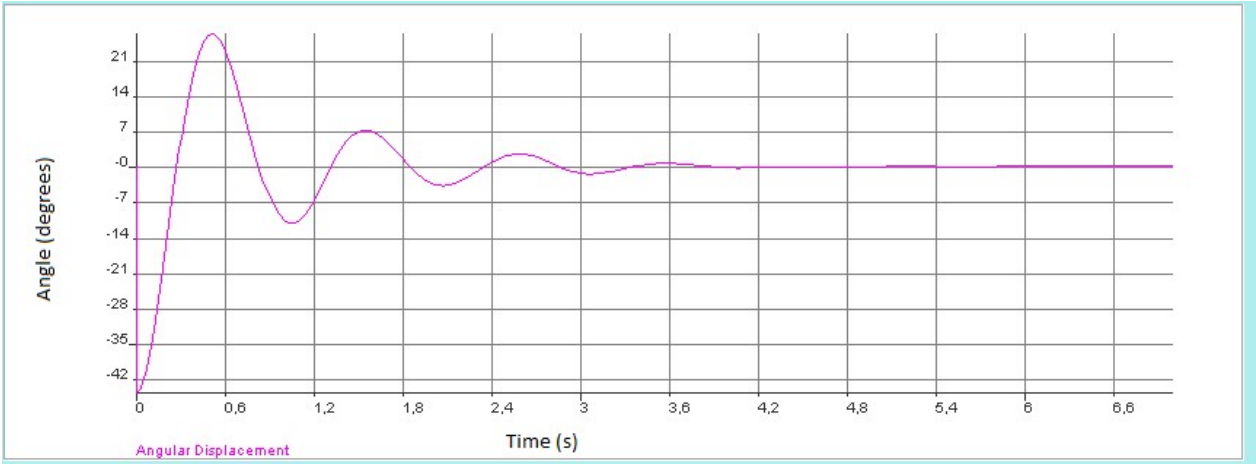


Figure 101: Angular displacement with -45 degrees disturbance on test stand

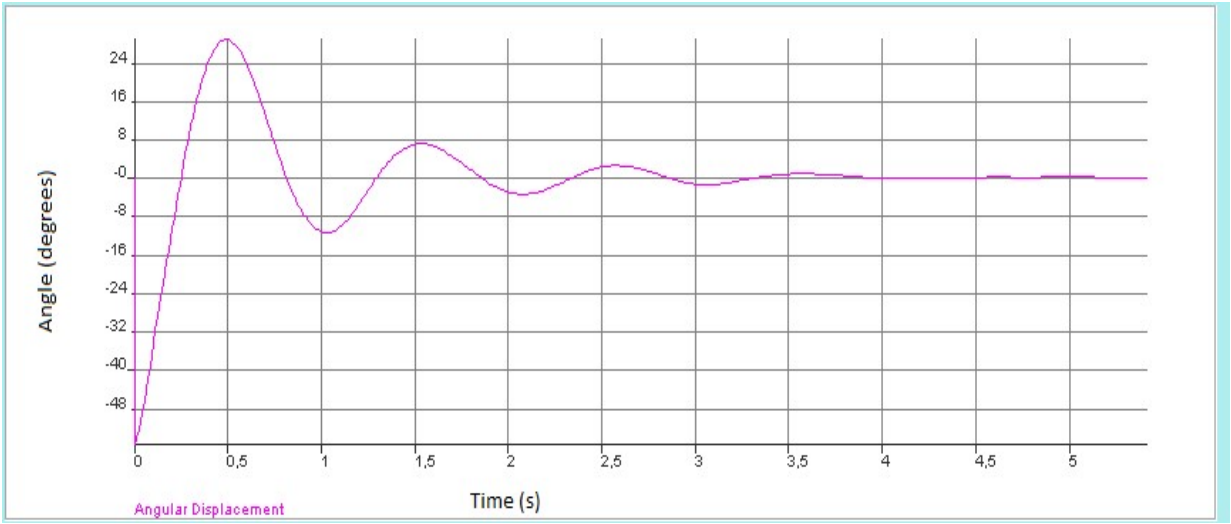


Figure 102: Angular displacement response for -60 degrees disturbance on test stand

12.2.2 Manipulating variable (Force (N))

As the figure 103 depicts the manipulating variable changes according to the controller action, to make position & sway control against the external disturbances.

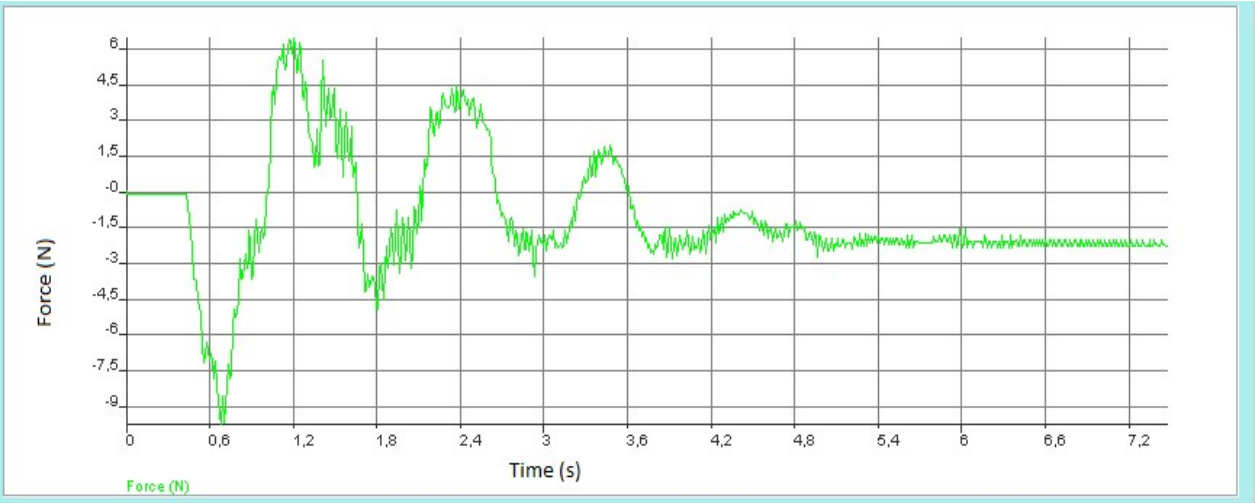


Figure 103: Manipulating variable (force(N)) response with disturbance on test stand

12.2.3 Reference postion versus cart displacement

The figure 104 depicts the position control when a disturbance is added at the reference position of $0.5m$

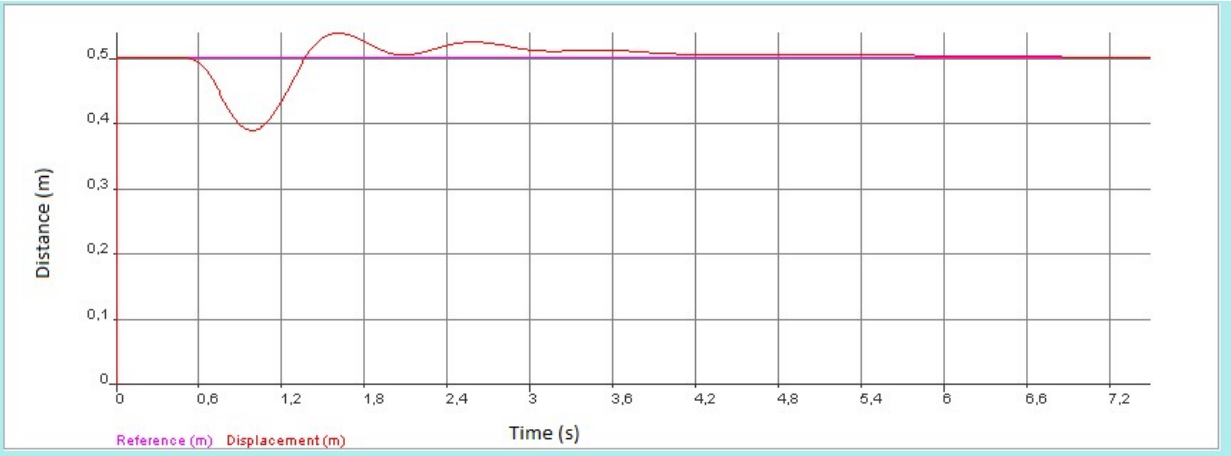


Figure 104: Cart displacement response with disturbance on test stand

13 CONCLUSION

The position & sway control of cart pendulum system involved many aspects of control systems and modern control theories. Many of its theories are applied to make the system controllable and to improve the performance of controller on the physical system in real time.

The solution to this classical problem is obtained by following an engineering approach. The system was first mathematically modelled to achieve the differential equations of the system which gives the complete information about the dynamic behavior of the system. The non-linear model is then linearized to design the controller by using state space control method.

The state feedback controller in combination with PI controller is first tested by simulation and then verified on physical system in real time. The real time performance of the controller matches with our design and agrees with the simulation results. The position & sway control of cart pendulum system are achieved with the designed controller.

There are always some model uncertainties between the modelled system and the physical system. The friction was one such a variable which was not accounted in the simulated model. Despite this it is observed that the controller designed for linear systems has worked on real time systems.

14 FUTURE WORK

- The designed controller (with and without friction) has been tested on test stand. There is no significant difference between the test results due to only consideration of linear friction $(Fc = Cr.V)$ in friction model (detailed in Appendix-II).

- The same project can be extended to achieve the faster control by considering the both static and kinetic friction between the cart and sledge.

15 REFERENCES

[1] Prof. Dr.-Ing K. Mueller, "Vorlesungen Kai Müller," [Online]. Available: http://www1.hs-bremerhaven.de/kmueller/Skript/espro_all.pdf.

[2] Prof. Dr.-Ing K. Mueller, "Vorlesungen Kai Müller," [Online]. Available: http://www1.hs-bremerhaven.de/kmueller/Skript/sysoc_all.pdf

[3] "Lagrangian mechanics - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Lagrangian_mechanics

[4] L. Günter, Theoretische Regelungstechnik 1, Springer publishing house Berlin Heidelberg, 1995.

[5] "Friction and the Inverted Pendulum Stabilization Problem", [online] Availiable: http://www.math.uwaterloo.ca/~sacampbe/preprints/fric2.pdf

[6] "An Inverted Pendulum Demonstrator for Timed Model-Based Design of Embedded Systems", Availiable: https://ieeexplore.ieee.org/document/6197419

[7] "Inverted Pendulum Demonstrator", Available: http://fse.studenttheses.ub.rug.nl/17710/1/final.pdf

[8] "Sparten 3E data sheet ", Available: https://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

[9] "Encoder WDG data sheet", Available: http://www.jinzon.com/pdf/WDG40S.pdf

[10] "Theoretische_Regelungstechnik_I and Theoretische_Regelungstechnik_II", Author:"Ludyk" , Publication by "Springer lehrbuch"

16 APPENDICES

16.1 Appendix – I (Sensor conversions)

The system can understand only physical values (S.I units), which means that it takes force as input and gives output(states) again in physical units. But on test stand, the position & angles are getting in integers from encoders. The DAC can understand only integer values. So, there is a need of DAC conversion factor. The experiment performed on test stand with spring by applying different voltages from the DAC and observed the different forces as shown in table 2.

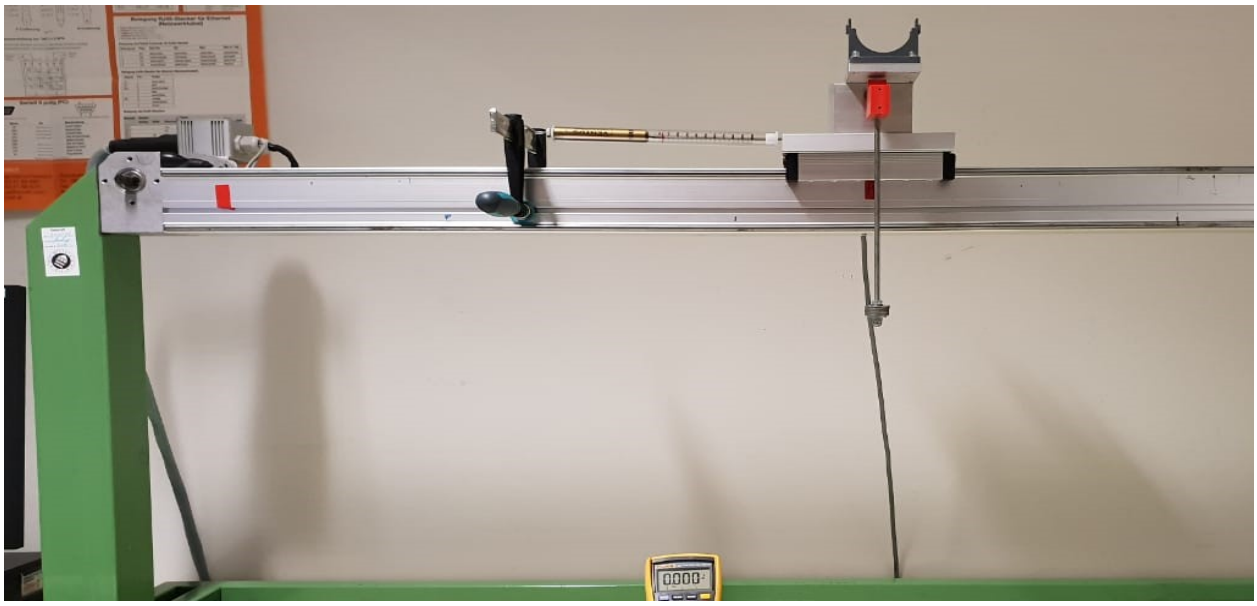


Figure 97: Test setup for DAC counts vs Force measurements

S.No.	DAC Counts (User view)	DAC Actual values (0-4095)	Voltage at Servo Amplifier (V)	Measured with spring Force (N)
1	200	2248	0.45	58.00
2	180	2228	0.412	48.00
3	160	2208	0.375	42.00
4	140	2188	0.336	35.00
5	120	2168	0.3	26.00
6	100	2148	0.262	21.00
7	80	2128	0.225	14.00
8	60	2108	0.187	10.00
9	40	2088	0.15	6.00
10	20	2068	0.113	1.00
11	0	2048	0.075	0.00

Table9: DAC Counts vs Force

The above table 9 values are plotted and observed that DAC counts are directly proportional(linear) to force values.

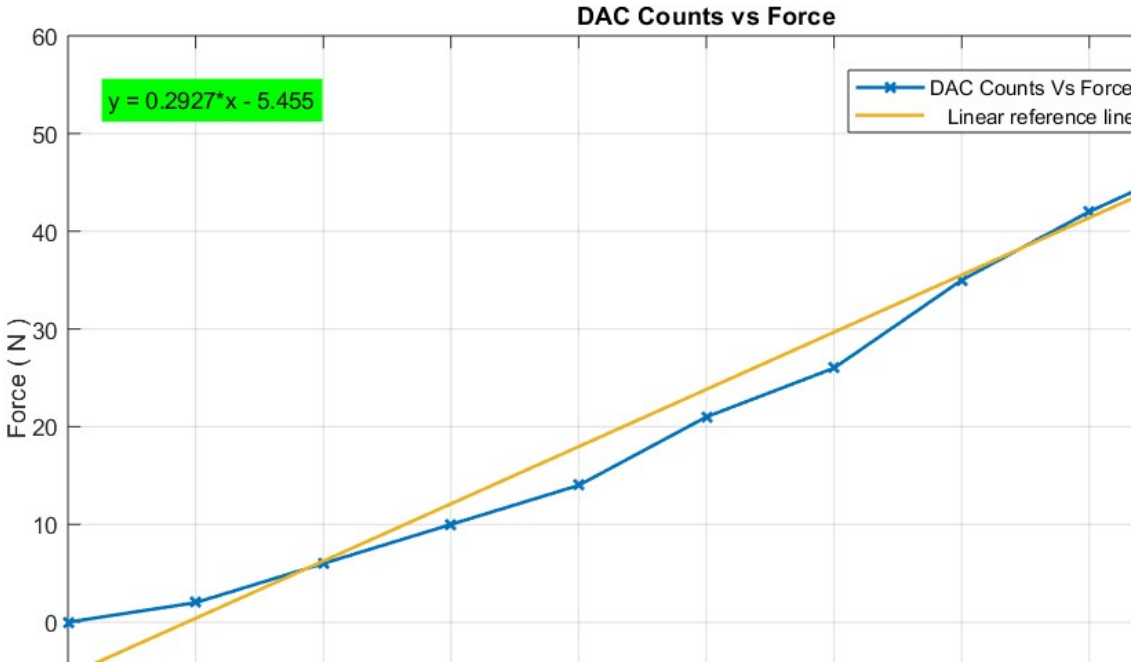


Figure 98: Plot for DAC counts vs Force

The linear equation from the graph,

$$y = 0.2927 \cdot x - 5.455 \quad (21)$$

From the equation (21), the Force $1\ N = \mathbf{22.0532}$ DAC Counts.

S.No.	Measurement	Distance(m)	Position encoder value	Pendulum encoder value
1	Distance between left limit switch & Right limit switch	1.35	-	-
2	Distance between left side cart start position & Right limit switch	1.18	67,747	-
3	Position Encoder value for 1m	1	57,238	-
4	Angle encoder value for 360°	360°	-	4096

Table 10: Test stand Position & Angle encoder values

The position encoder value gives for 1m displacement $= 57238$, it fits in 17-bit, but our inputs are limited to int_16.0 format. So, the displacement is dropped by one bit, which means that position encoder value divided by 2. It can measure only 0.5 meters displacement $= 28619$. The idea of full-

length sledge utilization, the reference position brings down to middle position and from that are moving left side $0.5\ m$ and right side $0.5\ m$.

The sensor conversion for cart displacement (m),

$$x\_i = 28619$$
$$x\_0 = 1$$

The sensor conversion for cart velocity (m/10ms),

$$v\_i = x\_i \cdot Ts$$
$$v\_0 = 1$$

Here, sampling time $Ts = 10ms$.

The sensor conversion for pendulum Angular displacement (radians),

$$alpha\_i = 4096$$
$$alpha\_0 = (2 \cdot pi)$$

The sensor conversion for pendulum Angular velocity (radians/10ms),

$$omega\_i = alpha\_i \cdot Ts$$
$$omega\_0 = 1$$

Here, Sampling time $Ts = 10ms$

16.2 Appendix– II (Calculation of friction coefficient on test stand)

We have taken force values for different velocities on the test stand and plotted graph in MATLAB showed below, later find the slope of the graph which gives the coefficient of friction $(6.171\ kg/s)$ of the test stand.

Sl.no	Cart speed in m/s (from GUI)	Force interms of DAC counts (X)	Force in N (X/F_I)
1	0.07	135	6.14
2	0.14	150	6.82
3	0.21	160	7.27
4	0.28	165	7.5
5	0.35	176	8
6	0.42	185	8.41

Table: Calculation of friction coefficient of test stand

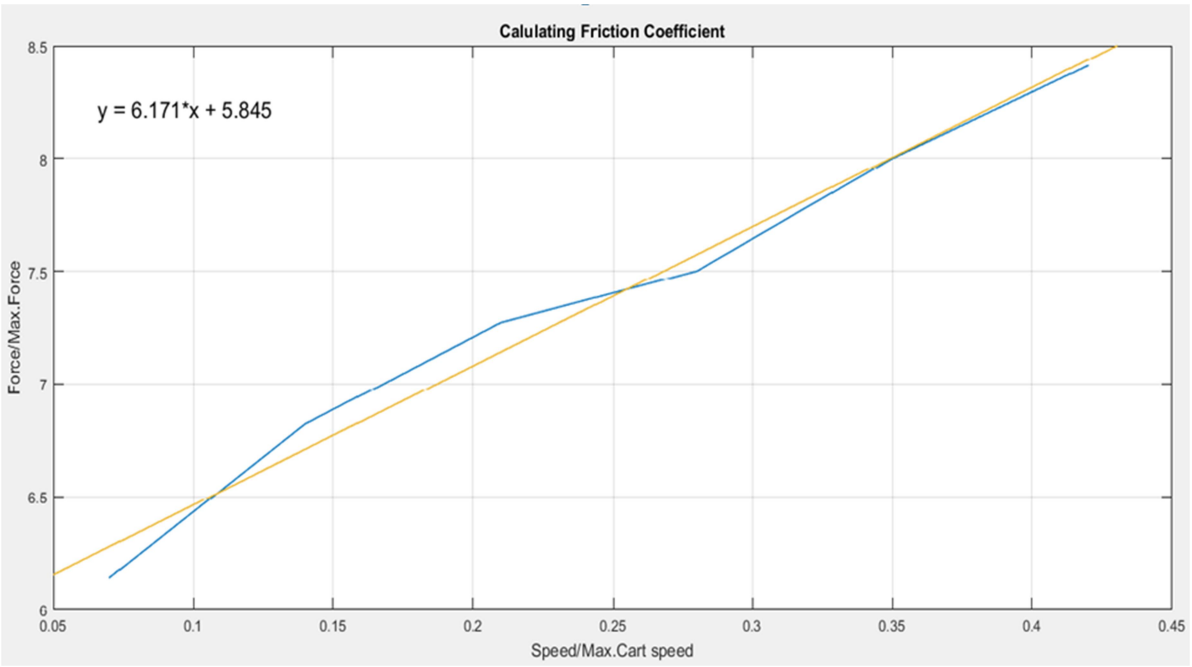


Figure: Plot for friction coefficient Speed vs Force

Mathematical modelling of cart pendulum system with Friction :

The cart-pendulum system has two degrees of freedom (2-DoF), namely horizontal displacement of cart and rotation of pendulum around the pivot. The 'F' is external force and 'Fc' is linear frictional force.

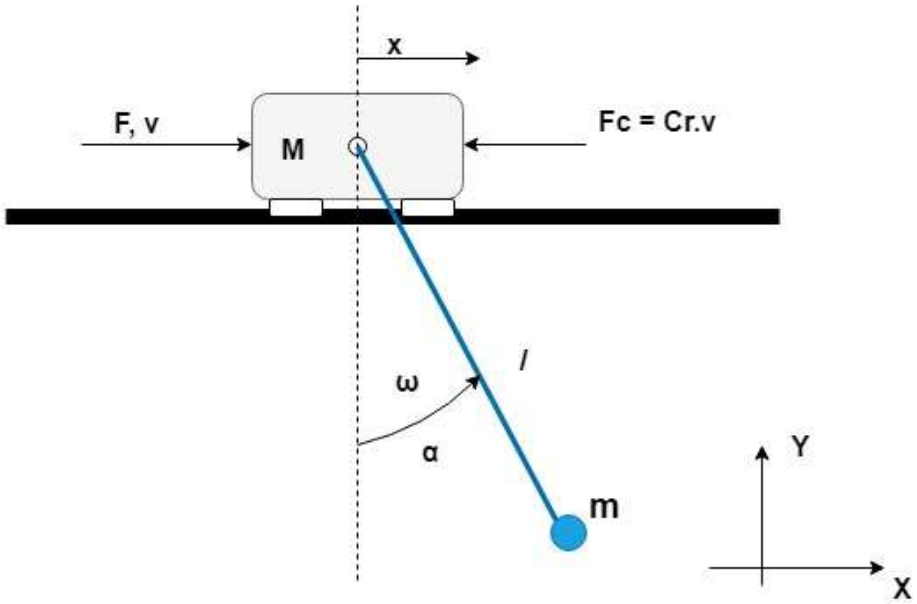


Figure 45: Cart pendulum system with Friction

Refer to chapter (3), the LaGrange's equations for general-coordinates (α,x) becomes

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\alpha}}\right) - \frac{\partial L}{\partial \alpha} = 0 \tag{22}$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = F\ - Fc \tag{23}$$

After solving equations (22) & (23), the non-linear equations are

$$\ddot{x} = \frac{F + m \cdot l \cdot \dot{\alpha}^2 \cdot \sin\alpha + m \cdot g \cdot \sin\alpha \cdot \cos\alpha - Cr \cdot \dot{x}}{M + m \cdot sin^2\alpha}$$

$$\ddot{\alpha} = \frac{-F \cdot \cos\alpha - m \cdot l \cdot \dot{\alpha}^2\ \cdot \sin\alpha \cdot \cos\alpha - g \cdot (M+m) \cdot \sin\alpha + Cr \cdot l \cdot \dot{x} \cdot cos\alpha}{l \cdot (M + m \cdot sin^2\ \alpha)}$$

The state space representation for linear model with friction is as follows:

$$\begin{bmatrix} \dot{\omega} \\ \dot{\alpha} \\ \dot{v} \\ \dot{x} \end{bmatrix} = \begin{bmatrix} 0 & \frac{-(M+m)\cdot g}{(M\cdot l)} & \frac{Cr}{(M\cdot l)} & 0 \\ 1 & 0 & 0 & 0 \\ 0 & \frac{(m\cdot g)}{M} & \frac{-Cr}{M} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \omega \\ \alpha \\ v \\ x \end{bmatrix} + \begin{bmatrix} -(\frac{1}{M\cdot l}) \\ 0 \\ \frac{1}{M} \\ 0 \end{bmatrix} \cdot F$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \omega \\ \alpha \\ v \\ x \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \cdot F$$

16.3 Appendix – III (Source Code)

- **Cpmain.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "xparameters.h"
#include "xil_printf.h"
#include "xil_cache.h"
#include "xbasic_types.h"
#include "xparameters.h"
#include "xtmrctr_l.h"
#include "xintc_l.h"
#include "velcontr.h"
#include "cpcontrl.h"

#ifndef boolean
typedef unsigned int boolean;
#endif

// uartlite peripheral control functions
#include "xuartlite_l.h"

#define LCD_INIT_DELAY  1000    // delay after LCD init
#define LCD_INST_DELAY   500    // delay after LCD instructions
#define LCD_DATA_DELAY   250    // delay after LCD data
// LCD Register Bits
#define  LCD_RW  0x010
#define  LCD_RS  0x020
#define  LCD_E   0x040

// DAC register bits (register 02)
#define  DAC_CS      0x80000000     // select DAC (active '1'!)
#define  DAC_CLR     0x40000000     // clear all DACS (0V, active '1'!)
#define  SPI_START   0x20000000     // start bit for SPI transmission
#define  SPI_DONE    0x00000001     // done bit for SPI transmission
#define  DAC_CMD     0x00300000     // update DAC immediately
#define  DAC_ADDR_A  0x00000000
#define  DAC_ADDR_B  0x00010000
#define  DAC_ADDR_C  0x00020000
#define  DAC_ADDR_D  0x00030000

// incremental encoder bits (register 03)
#define  IENC_LATCH    0x01     // latch position counters
#define  IENC_CLR      0x02     // clears position counters

// Accessing the Base Address
#define CI_REG(k)   (*(volatile u32*)(XPAR_CPINTERF_0_BASEADDR+4*k))

#define CI_COUNTER_15MS  500000     // counter reaches zero after 10ms

// buffer size for command and output
#define XBUF_SIZE 256
// Interrupt phases
#define IPHASE_PASSIVE 0
#define IPHASE_ARMED 1
#define IPHASE_RUN 2
#define IPHASE_STOP 3
```

```
// ring buffer transfer parameter
#define NLINES_TRANSMIT  100

static volatile unsigned int Lcd_NDelay = 26;
static volatile unsigned int Do_Lcd = 0;
static volatile u32 Intr_Count = 0;
static volatile int Serial_Count = 0;
static volatile int SProt_Count = 0;
static volatile int Intr_Phase = IPHASE_PASSIVE;
static volatile u32 Global_Led = 0;
static DataSet_type  Data_Set;


//*************************************************************************//
//                  VARIABLES : REFEERENCE MODE                            *//
//*************************************************************************//

#define AMP_EN 0x01                             // to enable  servo amplifier
#define AMP_OFF 0                               // to disable servo amplifier
#define SWRIGHT 1                               // right switch detected
#define SWLEFT 2                                // left switch detected.
#define ZERO_VOLT 2010                          // Value for 0 volt to motor input

// Reference States
#define RP_Off 0      // OFF state
#define RP_Init 1     // Initialization state
#define RP_Normal  2  // Reference Normal state
#define RP_Switch_R 3 // Right side switch detect
#define RP_Switch_L 4 // Left side switch detect
#define RP_Center 5 // Center position of cart state
#define RP_Stop 6   // Stop state of reference stage
#define RP_Slow_R 7 // Cart slow down at right switch detect
#define RP_Slow_L 8 // Cart slow down at left switch detect

#define POS_VEL 1       // velocity factor for right direction
#define NEG_VEL -1      // velocity factor for left direction
#define NR_ORIGIN 100   // Tolerence value at middle postion of cart
#define ALMOST_ZERO 20


u8 SwitchDetects =0;            // to store the detection of switch
u8 LimitSwState =0 ;            // Gives the state of the limit Switches
u8 Ref_State =0;                // gives the mode of the reference phase
s16 ref_vel =0 , vel_incr =0;   // reference velocity & velocity increment
s16 Pos_Incr = 0;               // used to slow down cart from positive speed
s16 Neg_Incr = 0;               // used to slow down cart from negative speed
s32 PathLength = 0;             // variable to store distance in counts between the
limit switches during reference mode
s32 CenterLength = 0;
u8 RefPhaseDone = 0;

//*************************************************************************//
//                  VARIABLES: CP_CONTROLLER                               *//
//*************************************************************************//

#define ALL_STATES 2

u32 PastPosCount =0, PresPosCount = 0;              // To store the
```

```
cart_position_encoder counts
u16 PastPendCount =0, PresPendCount = 0;            // To store the
pendulum_angle_encoder counts
static s16 uu=0;                                    // Force value
static s16 pos_ref=0;
volatile u8 PrintInfo=0;                            // to print the counts

static void DACwrite(int chan, unsigned int ddata); // declaration of function
                                                       dacwrite
static void PosClear();                             // Clears the position counter
                                                       of cart and Pendulum
static void PosLatch();                             // Latches the current count
                                                       in the hardware
static u8 ReadLimitSw (void);                       // Gives the present state of
                                                       the Limit Switches
static void DirCompute(void);                       // Path length and Direction
                                                        Computation in the
                                                        referencing Phase
void GetStates(void);                               // Gives the current velocity
                                                       of the cart
void VelContr(u8 control_mode);                     // To control the reference
                                                       velocity of the cart
                                                       in Reference Phase


extern s16 Int_States[N_STATES] ;                   // States of the Cart_Pendulum
                                                       System [0]=pend_velocity
                                                       [1]=pend_angle
                                                       [2]=cart_velocity
                                                       [3]=cart_position
/*
** -----------------------------------------------------
** timer interrupt handler (associated with timer 0)
** -----------------------------------------------------
*/
void CITimerInterrupt(void * baseaddr_p)
{
    u32  tcsr;
    static int led_toggle = 0;

    tcsr = XTmrCtr_GetControlStatusReg(baseaddr_p, 0);
    if (tcsr & XTC_CSR_INT_OCCURED_MASK) {
     LimitSwState = ReadLimitSw();   // get the states of both the limit switches
     GetStates();                    // Get the controller states i.e. Int_States[k]

        if(Ref_State != RP_Off){  // Check Reference state off condition.
            switch (Ref_State){   // to select state in reference stage algorithm
                case RP_Init:     // Initilisation stage
                RefPhaseInit();   // call reference stage initialization function
                break;
                case RP_Normal:   // Normal state
                    if(LimitSwState){          // Detech limit switch
                        if(LimitSwState == 1){ // Detech Right limit switch
                            Ref_State = RP_Slow_R; // change state to make cart slow
                        }
                    }
                    else{
                    Ref_State = RP_Normal;
                    }
                break;
                case RP_Slow_R:              // Cart slow at right side of belt.
```

```
                    Pos_Incr = Pos_Incr >> 1; // Make division to stop cart slowly.
                    if(Pos_Incr == 0){        // Check for zero cart velocity.
                        Ref_State = RP_Switch_R; // Change to right switch detected
                    }
                    else{
                        Ref_State = RP_Slow_R;
                    }
                    break;
                case RP_Switch_R: //State to change direction of cart at right switch
                    if(LimitSwState == 2){     // check left switch detection
                        Ref_State = RP_Slow_L; // switch to make slow movement of
                                               cart at left.
                    }
                    else{
                        Ref_State = RP_Switch_R;
                    }
                break;
                case RP_Slow_L:
                    Neg_Incr = Neg_Incr && 0x7FFF; // Mask the MSB for Negative sign.
                    Neg_Incr  =  Neg_Incr >> 1; // Make division to stop cart slowly.
                    if(Neg_Incr == 0){             // Check for zero force.
                      Ref_State = RP_Switch_L;     // Change to left switch detected.
                    }
                    else{
                        Ref_State = RP_Slow_L;
                    }
                    break;
                case RP_Switch_L: //State to change direction of cart at left switch
                     Ref_State = RP_Center;
                break;
                case RP_Center:               // State to make cart in center postion
                if( (PathLength >= (CenterLength-NR_ORIGIN)) && (PathLength <=
(CenterLength+NR_ORIGIN)) ){
                    Ref_State = RP_Stop;     // Check for cart is at center or not.
                }
                break;
                case RP_Stop:               //  Stop state for reference stage.
                    Int_States[3] = 0;      // Make displacement state to value zero.
                    Ref_State = RP_Off;     // Switch to reference phase off state.
                break;
                default :
                    Ref_State = RP_Stop;    // Make off state in default condition.
                break;
             }

            VelContr(Ref_State); // Control the velocity as required by the stage of
                                 operation
            }
        else {
            switch (Intr_Phase) {
                    case IPHASE_PASSIVE:
                        uu = CpCtrExec(0);
                        break;
                    case IPHASE_ARMED:
                        CpContrInit();
                        Intr_Phase = IPHASE_RUN;
                        break;
                    case IPHASE_RUN:

                        if(ReadLimitSw() == 0)
```

```
                        {
                            uu = CpCtrExec(1);
                            PrintInfo = ALL_STATES ;
                        }
                        else
                        {
                            vel_incr =0;
                            Intr_Phase = IPHASE_PASSIVE;
                            CI_REG(5) = AMP_OFF;
                        }
                        break;
                    case IPHASE_STOP:
                        Intr_Phase = IPHASE_PASSIVE;
                        break;
                    default:
                        Intr_Phase = IPHASE_PASSIVE;
                        break;
                }
        }

        DACwrite(3,(uu+ZERO_VOLT)<<4);    // Write the computed_force_value to DAC
    }

    if (Intr_Count >= 16) {
        Intr_Count = 0;
        if (led_toggle == 0) {
            Global_Led |= 1;
            led_toggle = 1;
        } else {
            Global_Led &= 0xfffffffe;
            led_toggle = 0;
        }
    } else Intr_Count++;

    CI_REG(0) = Global_Led;
    XTmrCtr_SetControlStatusReg(baseaddr_p, 0, tcsr);

}

/*
** initialize timer, interrupt controller and register handler
*/
static void CiSetupInterrupt() {
    // disable everything in timer
    XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 0);
    // set the load register
    XTmrCtr_SetLoadReg(XPAR_XPS_TIMER_0_BASEADDR, 0, CI_COUNTER_15MS);
    // load counter
    XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,
            XTC_CSR_LOAD_MASK);
    // setup AXI interrupt
    XIntc_RegisterHandler(XPAR_INTC_0_BASEADDR,
            XPAR_INTC_0_TMRCTR_0_VEC_ID,
            CITimerInterrupt,
            (void *)XPAR_XPS_TIMER_0_BASEADDR);
    XIntc_MasterEnable(XPAR_INTC_0_BASEADDR);
    XIntc_EnableIntr(XPAR_INTC_0_BASEADDR,
            XPAR_XPS_TIMER_0_INTERRUPT_MASK);
    // start counter
    XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,
```

```
            XTC_CSR_ENABLE_TMR_MASK
          | XTC_CSR_AUTO_RELOAD_MASK
          | XTC_CSR_DOWN_COUNT_MASK
          | XTC_CSR_ENABLE_INT_MASK);
    microblaze_enable_interrupts();
}

/*
** cleanup timer interrupt stuff
*/
static void CiCleanupInterrupt() {
    microblaze_disable_interrupts();
    XIntc_DisableIntr(XPAR_INTC_0_BASEADDR,
            XPAR_XPS_TIMER_0_INTERRUPT_MASK);
    XIntc_MasterDisable(XPAR_INTC_0_BASEADDR);
    // disable everything in timer
    XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 0);
}


/*
** DAC utilities
*/
static void DACclear() {
    CI_REG(2) = DAC_CLR;
    CI_REG(2) = DAC_CLR;
    CI_REG(2) = 0;
}

static void DACwrite(int chan, unsigned int ddata) { // To append DAC data, command
                                                     selection of DAC cahnnel.
    int  kc = 0;
    u32  ci_data, xt;

    ci_data = DAC_CS;              // Make chip select active.
    CI_REG(2) = ci_data;           // Update data in the Slave Register of SPI data.
    switch (chan) {          // Select DAC cgannel, Start bit, command bits for DAC.
    case 0:
       ci_data |= SPI_START | DAC_CMD | DAC_ADDR_A | (ddata & 0x0ffff);//Select DAC A
        break;
    case 1:
       ci_data |= SPI_START | DAC_CMD | DAC_ADDR_B | (ddata & 0x0ffff);//Select DAC B
        break;
    case 2:
       ci_data |= SPI_START | DAC_CMD | DAC_ADDR_C | (ddata & 0x0ffff);//Select DAC C
        break;
    case 3:
       ci_data |= SPI_START | DAC_CMD | DAC_ADDR_D | (ddata & 0x0ffff);//Select DAC D
    }
    CI_REG(2) = ci_data;            // Update data in the Slave Register of SPI data.
    CI_REG(2) = ci_data & (~SPI_START);               // Make SPI start bit to zero.
    do {
        xt = CI_REG(2);
        kc++;
    } while ((!(xt & SPI_DONE)) && (kc < 100));           // Check for SPI done bit.
    CI_REG(2) = 0;  // Clear the SPI data from slave register once it is transmitted.

static void PosClear() {
    CI_REG(3) = IENC_CLR;
```

```
    CI_REG(3) = 0;
}

static void PosLatch() {
    CI_REG(3) = IENC_LATCH;
    CI_REG(3) = 0;
}


/*UART getchar // gives a single character from the  Recieve Buffer*/

static boolean UARTgetchar(char *ch) {
   char cc = 0;
   boolean  ret_val;

   ret_val = (!XUartLite_IsReceiveEmpty(XPAR_RS232_DCE_BASEADDR));
   if (ret_val) cc = XUartLite_RecvByte(XPAR_RS232_DCE_BASEADDR);
   *ch = cc;
   return ret_val;
}

/* LCD stuff */
// Utility routines
void lcdsleep(unsigned int delay) {
unsigned int j, i;

for(i = 0; i < delay; i++)
    for(j = 0; j < Lcd_NDelay; j++);
}

static void lcdinitinst(void)
{
    CI_REG(1) = 0x03;  lcdsleep(1);
    //set enable and data
    CI_REG(1) = LCD_E | 0x03;  lcdsleep(1);
    CI_REG(1) = 0x03;  lcdsleep(LCD_INIT_DELAY);
}

static void lcdwriteinst(unsigned long lcd_inst)
{
    unsigned long xinst;

    xinst = (lcd_inst >> 4) & 0x0f;
    CI_REG(1) = xinst;
    lcdsleep(1);
    CI_REG(1) = LCD_E | xinst;
    lcdsleep(1);
    CI_REG(1) = xinst;
    lcdsleep(1);
    xinst = lcd_inst & 0x0f;
    CI_REG(1) = LCD_E | xinst;
    lcdsleep(1);
    CI_REG(1) = xinst;
    lcdsleep(LCD_INST_DELAY);
}

static void lcdwritedata(unsigned long lcd_data)
{
    unsigned long xdata;
```

```
    xdata = LCD_RS | ((lcd_data >> 4) & 0x0f);
    CI_REG(1) = xdata;  lcdsleep(1);
    CI_REG(1) = LCD_E | xdata;  lcdsleep(1);
    CI_REG(1) = xdata;  lcdsleep(1);

    xdata = LCD_RS | (lcd_data & 0x0f);
    CI_REG(1) = LCD_E | xdata;  lcdsleep(1);
    CI_REG(1) = xdata;  lcdsleep(LCD_DATA_DELAY);
}

static void SiLcdInit()
{
    lcdsleep(15000);    // After VCC>4.5V Wait 10ms to Init Char LCD
    lcdinitinst();
    lcdsleep(4100);     // Wait 4.1ms
    lcdinitinst();
    lcdsleep(100);      // Wait 100us
    lcdinitinst();
    lcdinitinst();

    //Function Set
    lcdwriteinst(0x028);
    //Entry Mode Set: increment, no shift
    lcdwriteinst(0x006);
    //Display: on, no cursor, no blinking
    lcdwriteinst(0x00C);
    //Display Clear
    lcdwriteinst(0x001);
}

static void SiLCDPrintString(int lpos, char *line){
    int i = 0;
    char  ch;

    switch (lpos & 0x03) {
        case 0:
            lcdwriteinst(0x080);
            break;
        case 1:
            lcdwriteinst(0x088);
            break;
        case 2:
            lcdwriteinst(0x0C0);
            break;
        default:
            lcdwriteinst(0x0C8);
    }
    while((ch = line[i++])) lcdwritedata(ch);
}

/*
** REF_PHASE sub routines
*/

u8 ReadLimitSw (void)           // Reads the present condition of Left and right
Limit Switch
{
    u32 Switchdata = 0;

    Switchdata = CI_REG(5);             // read the switches
```

```
    Switchdata &= 0x00000006;           // mask the other bits
    Switchdata >>= 1;                   // shift to right
/*
    ** Switchdata    : 0 - No limit
    **               : 1 - Right_limit
    **               : 2 - Left_limit
    **               : 3 - Both_limit
    */

    return (u8)Switchdata;
}

// Checks the initial condition of Limit_Switch and Initializes the reference phase
s8 RefPhaseInit(void)
{
    s8 LimitSwData = 0;
    SwitchDetects = 0;
    ref_vel = 0;
    vel_incr = POS_VEL;

    LimitSwData = ReadLimitSw();

    switch(LimitSwData){

        case 0 : RefPhase = RP_NORMAL; break;       // move left
        case 1 : RefPhase = RP_LIMIT; break;        // cart is at the right_ limit
        case 2 : RefPhase = RP_LIMIT; break;        // cart is at the left_ limit
        case 3 : RefPhase = 0; break;               // Error
        default : RefPhase = 0; break;
    }

    if(RefPhase)
    {
        VelCtrlInit(KP_D,KI_D,LIMIT_INT,LIMIT_OUT); // initialize the velocity
controller
    }

    return RefPhase ;
}

void GetStates(void)                //  To calculate current velocity of cart
{
    PosLatch();                     // Latch the position and angle encoder values

    PastPosCount = PresPosCount;
    PresPosCount = CI_REG(3);
    Int_States[2] = (s16)(PresPosCount - PastPosCount);     // Cart velocity
    Int_States[3] += (Int_States[2]);  // Cart Displacement/2 add  if problem with
                                          cp phase

    PastPendCount = PresPendCount;
    PresPendCount = CI_REG(4);
    Int_States[0] = (s16)(PastPendCount - PresPendCount);   // pend_velocity
    Int_States[1] += Int_States[0];                         // pend_angle

}

// Path length and Direction Computation in the referencing Phase
void DirCompute(void)
{
```

```
    SwitchDetects++;                // one more time a limit_switch is detected

    if(LimitSwState == SWLEFT){
        vel_incr = POS_VEL;         // change direction of velocity to +X direction
    }
    else if(LimitSwState == SWRIGHT){
        vel_incr = NEG_VEL ;        // change direction of velocity to -X direction
    }
}

// To control the reference velocity of the cart in Reference Phase and compute the
PI_controller calculation

void VelContr(u8 control_mode)
{
    if (control_mode != RP_OFF) {
        if(control_mode == RP_NORMAL){
             ref_vel += vel_incr  ;   // If normal mode then increase the velocity
                                         in the direction
        }
        else if( (control_mode == RP_SLOW) || (control_mode == RP_MIDDLE)){
            ref_vel += (vel_incr<<4) ;     // slow down the cart quickly and
                                              change the direction of the velocity
        }
        else if(control_mode == RP_COMPLETE){
            ref_vel -= (vel_incr<<4) ;   // Increase the cart velocity in the
                                            direction opposite to the current to
                                            bring it to zero
            if(ref_vel > -ALMOST_ZERO && ref_vel < ALMOST_ZERO) {
                ref_vel = 0;
            }
        }

        if(ref_vel > MAX_SPEED){       // Don't let the reference velocity
                                          increase beyond the Maximum; positive as
                                          well as negative

            ref_vel = MAX_SPEED;
        }else if(ref_vel < -MAX_SPEED){
            ref_vel = -MAX_SPEED;
        }


        uu = VelContrCalc(ref_vel, Int_States[2]);      // Compute the output
    }
    else{
        uu = 0;
    }
}

// Here we start...
int main()
{
    u32  x_data;
    int  k;
    //int  Amp_On;
    char cmdchar;
    //unsigned int  dac_data = 0;
    //char nbuf[16];
    char cmdbuf[XBUF_SIZE];
```

```
int  cmdbcnt, ltxcnt;
boolean cmd_valid, exit_cp = FALSE;

Xil_ICacheEnable(); // Enable Instruction Cache
Xil_DCacheEnable(); // Enable Data Cache
print("--- pendulum control V0.0a ---\n");
print(" o init hardware...\n");

// Clear position counters
PosClear();
SiLcdInit();
SiLCDPrintString(0, "Cart-Pend. V1.a");
SiLCDPrintString(2, "ESD |");
SiLCDPrintString(3, "2018");
//Amp_On = 0; // to turn ON the servo-Amp

DACclear();          // Clear all DAC

DACwrite(3,ZERO_VOLT<<4);       //make the motor voltage 0V

x_data =0xFFFFFFFF;
xil_printf("d_x=%d h_x=%x\n",x_data, (unsigned int)x_data);

for (k = 0; k < 8; k++)
{
    x_data = CI_REG(k);
    xil_printf("k=%d x=%x\n", k, (unsigned int)x_data);
}
print(" o set up timer interrupt...\n");
ClearAllStates();                          // All internal states are cleared
CiSetupInterrupt();                        // setting up interrupt
cmdbcnt = 0;  cmd_valid = FALSE;

// main loop starts here
do {
        if(PrintInfo){
            switch(PrintInfo){
                case RP_COMPLETE : xil_printf("Reference Phase Completed\n");
                          break;
                case ALL_STATES  : xil_printf("? %d \n",Int_States[0]);
                          break;

               default: PrintInfo = 0;
                          break;
            }
            PrintInfo = 0;
        }
        // waits for the end of the command i.e \r \n
        if (UARTgetchar(&cmdchar))
        {
            if (cmdchar != '\r') {
                if (cmdchar == '\n') {
                    if (cmdbcnt > 0) {
                        cmd_valid = TRUE;    // evaluate the given user
                                               input string which is now valid

                        cmdbuf[cmdbcnt++] = '\0';       // append a null
                    }
                } else if (cmdbcnt < XBUF_SIZE-2) cmdbuf[cmdbcnt++] = cmdchar;
```

```
// store the user input (from GUI)
                }
                if (cmd_valid) {
                        if (!strcmp(cmdbuf, "exit")){
                         exit_cp = TRUE;
                     }
                        // to exit the program
               else if (!strncmp(cmdbuf, "mode ", 5)) {
                  if (!strcmp(&cmdbuf[5], "on")) {Intr_Phase = IPHASE_ARMED;}
                  else if (!strcmp(&cmdbuf[5], "off")){ Intr_Phase = IPHASE_STOP;}
                            else {
                                xil_printf("I_Phase = %d\n", Intr_Phase);
                            }
                        } else if (!strncmp(cmdbuf, "log ", 4)) {
      // to start and stop logging the data; to show and clear the Data recorded
                            if (!strcmp(&cmdbuf[4], "on"))
                            {
                                CpStartLogger();
                                print("Logger On.\n");
                            }
                            else if (!strcmp(&cmdbuf[4], "off"))
                            {
                                CpStopLogger();
                                print("Logger Off.\n");
                            }
                            else if (!strcmp(&cmdbuf[4], "clr"))
                            {
                                CpClearLogger();
                                print("Log cleared.\n");
                            }
                            else if (!strcmp(&cmdbuf[4], "show"))
                            {
                                print("Logger data:\n");
                                while ((k = CpPopRecord(Data_Set)) != 0)
                                {
                                    xil_printf(" (%d): %d \n", k,Data_Set[2]);//,
                                }
                                print("Logger complete.\n");
                            } else {
                                if ((k = CpPopRecord(Data_Set)) == 0)
                                    print("*** no log data ***\n");
                                else {
                                    xil_printf(" (%d): %d \n", k,Data_Set[2]);//,
                                }
                            }
                        }else if (!strcmp(cmdbuf, "p")) {
                            ltxcnt = 0;
                            do {
                                k = CpPopRecord(Data_Set);
                                xil_printf("k value \n", k);
                                if (k == 0)  {
                                    print("###\n");
                                } else {
                                    ltxcnt++;
                                    if (ltxcnt < NLINES_TRANSMIT) {
                                        xil_printf("$ %d \n", Data_Set[2]);
                                    } else {
                                        xil_printf("~ %d \n", Data_Set[2]);
                                    }
                                }
```

```
                            } while ((k > 0) && (ltxcnt < NLINES_TRANSMIT));
                        }
                        else if (!strncmp(cmdbuf, "refp",4)){
                            PathLength = 0;
                            SwitchDetects= 0;
                            if(RefPhaseInit())
                            {
                                xil_printf("Ref_Phase Started.Pl wait..\n");
                                CI_REG(5) = AMP_EN;     // Enable servo amplifier
                                Intr_Phase = IPHASE_FAIL;
                            }
                         else
                          xil_printf("error: Ref_Phase initialization failed.\n");
                        }
                        else if (!strncmp(cmdbuf, "cp ",3)){

                            k = atoi(&cmdbuf[3]);

                            if ((k >= -30000) && (k <= 30000)){
//  Maximum allowed length 0.5m equal to 28619 encoder value
                                CpSetRef(k);
                                pos_ref =k;
                            }
                            else{
                                print("*** invalid ref value ***\n");
                            }
                        }
                        else if (!strncmp(cmdbuf, "refsp ",6)){

                            k = atoi(&cmdbuf[6]);

                            if ((k >= -286) && (k <= 286))
{               //  Maximum allowed length 0.5m equal to 28619 encoder value
                                CpSetRef(k);
                                pos_ref =k;
                            }
                            else {
                                print("*** invalid ref value ***\n");
                            }
                        }
                        cmdbcnt = 0;  cmd_valid = FALSE;
                    }
                }

    } while (!exit_cp);
    print(" o clean up timer interrupt...\n");
    CiCleanupInterrupt();
    print(" o interrupts off...\n");

    SiLCDPrintString(2, "  Good Bye.  ");
    DACwrite(3,ZERO_VOLT<<4);       //make the motor voltage 0V
    CI_REG(5) = AMP_OFF;            // disable the servo Amplifier
    print("Thank you for using pendctr.\n");

    Xil_DCacheDisable();
    Xil_ICacheDisable();
    return 0;

}
```

- **cpcontrl.c**

```
#include "xbasic types.h"
#include "xil_types.h"
#include "cpcontrl.h"
#include "velcontr.h"
#include "xil_printf.h"

static s16 KD[N_STATES] = { -7231, -14573, 23956, 389 }; // without friction, state
feedback gain vector all poles are at (-4)  ; fixed_point int_16.12
static PIObj_struct CpPhase;
s16 Int_States[N_STATES] = {0, 0, 0, 0};
static s16 pos_ref = 0;
static volatile int Logger_Active = 0;
static int InPointer = 0, OutPointer = 0;
static DataSet_type Logger_Data[NRECORDS];

/*
 *  Data logger for all states of the controller , Input and Force
 *  Prints the data of every row of ring buffer ; the size of ring Buffer is
1000(rows)*6(columns)
 */
void CpStartLogger() {
    Logger_Active = 1;
}
void CpStopLogger() {
    Logger_Active = 0;
}
void CpClearLogger() {
    Logger_Active = 0;
    InPointer = 0;  OutPointer = 0;
}
int CpPopRecord(DataSet_type cp_rec) {
    int  nr;
    nr = InPointer - OutPointer;
    if (nr < 0)
        nr += NRECORDS;
    if (nr != 0) {
        cp_rec[0] = Logger_Data[OutPointer][0];
        cp_rec[1] = Logger_Data[OutPointer][1];
        cp_rec[2] = Logger_Data[OutPointer][2];
        cp_rec[3] = Logger_Data[OutPointer][3];
        cp_rec[4] = Logger_Data[OutPointer][4];
        cp_rec[5] = Logger_Data[OutPointer][5];
        OutPointer = (OutPointer + 1) % NRECORDS;
    }
    return nr;
}
static void CpPushRecord(int uu) {
    Logger_Data[InPointer][0] = (CpInt16)pos_ref;
    Logger_Data[InPointer][1] = (CpInt16)uu;
    Logger_Data[InPointer][2] = (CpInt16)Int_States[0];
    Logger_Data[InPointer][3] = (CpInt16)Int_States[1];
    Logger_Data[InPointer][4] = (CpInt16)Int_States[2];
    Logger_Data[InPointer][5] = (CpInt16)Int_States[3];
    InPointer = (InPointer + 1) % NRECORDS;
    if (InPointer == OutPointer)  OutPointer = (OutPointer + 1) % NRECORDS;
```

```
}

// PI controller for  cart_position
void Cp_PiInit(s16 pgain, s16 igain, s32 limit_i, s32 limit_o){
    CpPhase.kp = pgain;
    CpPhase.ki = igain;
    CpPhase.limit_int = limit_i;
    CpPhase.limit_out = limit_o;
    CpPhase.xi = 0;
    CpSetRef(0);
}
s32 Cp_PiCntrlC(s16 reff, s16 out_fbb){
    s32 u_ctr;
    s16 ctr_e;
    ctr_e = (s16)(reff - out_fbb) ;     // Calculate the error

    u_ctr = CpPhase.kp * ctr_e + (CpPhase.xi >> (CP_KI_SHIFT - CP_KP_SHIFT));
 // calculating the PI controller output
    if (u_ctr > CpPhase.limit_out ) {   // limiting the PI controller output value
        u_ctr = CpPhase.limit_out;
    }
    else if (u_ctr < -CpPhase.limit_out ){
        u_ctr = -CpPhase.limit_out;
    }
    CpPhase.xi += (CpPhase.ki * ctr_e) ;       // calculating the integral values

    if ( CpPhase.xi > CpPhase.limit_int ) {    // Limit on integral values

       CpPhase.xi = CpPhase.limit_int;
    }
    else if ( CpPhase.xi < -CpPhase.limit_int {
       CpPhase.xi = -CpPhase.limit_int;
    }
    return u_ctr;
}
/*
 * Initialize the controller for cart pendulum system
 */
void CpContrInit(void){
// Initialize the PI controller
    Cp_PiInit(CP_KP_FIX, CP_KI_FIX, CP_LIMIT_INT, CP_LIMIT_OUT);
}
/*
 * Initialize the controller for cart pendulum system
 */
void ClearAllStates(void){
    u8 k;

    for (k = 0; k < N_STATES; k++)  Int_States[k] = 0;

    pos_ref = Int_States[3];
}
/*
 *  Controller Calculation ( PI + state_feedback)
 */
s16 CpContrCalc(s16 refval){
    s32 u_pi=0;
    s32 u_sfb=0;
    u8 k;
```

```
    u_pi = Cp_PiCntrlC(refval, Int_States[3]);    // PI_controller calculated
                                                    output in format 32.KP_SHIFT;
    u_pi >>= CP_KP_SHIFT-KD_SHIFT;                // its necessary to make the fra-
                                                    ctional digits same int_32.12

    for (k = 0; k < N_STATES; k++) {             //State feedback calculation( K*X)
        u_sfb += (KD[k] * Int_States[k]);
    }
    u_sfb = u_pi - u_sfb;                       // state feedback controller output
                                                    (int_32.12)

    if (u_sfb > LIMIT_POS ) {                   // Limiting the state feedback
                                                  controller output
        u_sfb = LIMIT_POS;
    }
    else if (u_sfb < LIMIT_NEG ) {

        u_sfb = LIMIT_NEG;
    }
    u_sfb >>= KD_SHIFT;
// Shifting the result value further 12 bits to fit the value in DAC range
(int_32.24)
    return (s16) u_sfb;
}
/*
 *  Set the reference value for the controller
 */
void CpSetRef(s16 ref) {
    pos_ref = ref;
}

/*
 * Starts the controller for cart-pendulum system ; if c_mode = 0  (i.e I_PHASE =
IPHASE_PASSIVE) then its turned off
 */
s16 CpCtrExec(int cmode) {

    s16  uu;

    uu = CpContrCalc(pos_ref);  // Calculate the manipulating variable according
                                   to the Reference value
    if (Logger_Active == 1)     // If log is ON
    {
        CpPushRecord(uu);       // Then push the data in the ring buffer
    }
    if (cmode == 0)
    {
        uu = 0;         // If the INTERRUPT_PHASE is '0', then stop the controller
    }
    return uu;
}
```

- **Cpcontrl.h**

```
/* cpcontrl.c */

#ifndef CPCONTRL_H_
#define CPCONTRL_H_
#define NRECORDS 1000
#define NRSIZE 6

/* ---- from MATLAB: ------- */

#define N_STATES 4
#define CP_KP_FIX 878           // int_16.15        All poles at (-4)
#define CP_KI_FIX 32            // int_16.15
#define CP_KP_SHIFT  15
#define CP_KI_SHIFT  15
#define CP_LIMIT_INT    90112000  //int_32.15(integral value limited to 2750*2^15)
#define CP_LIMIT_OUT    90112000  //int_32.15(PI controller output value limited
                                    to 2750*2^15)

#define KD_SHIFT 12                 // int_16.12
#define LIMIT_POS   903299          // int_32.12 format(limiting the force to
                                       10*(F_i/F_0)*2^12)
#define LIMIT_NEG   -903299         // int_32.12


typedef short int  CpInt16;

typedef CpInt16 DataSet_type[NRSIZE];

void CpStartLogger();
void CpStopLogger();
void CpClearLogger();
int CpPopRecord(DataSet_type cp_rec);
void CpPopCurrent(DataSet_type cp_rec);
void CpSetRef(s16 ref);
s16 CpCtrExec(int cmode);
void CpContrInit(void);
void ClearAllStates(void);


#endif /* CPCONTRL_H_ */
```

- **Velcontr.c**

```
#include "xil_types.h"
#include "velcontr.h"
#include "xil_printf.h"
static PIObj_struct RefPhase;
/*
 *  To initialize the PI_controller for controlling the velocity
 *  of the cart for the reference phase. */
void VelCtrlInit(s16 pgain, s16 igain, s32 limit_i, s32 limit_o)
{
```

```
	RefPhase.kp = pgain;
	RefPhase.ki = igain;
	RefPhase.limit_int = limit_i;
	RefPhase.limit_out = limit_o;
	RefPhase.xi = 0;
}
/*
 * The function calculates the PI_velocity_controller output according
 *  to the output_error */
s16 VelContrCalc(s16 reff, s16 out_fbb)
{
	s32 u_ctr;
	s16 ctr_e;
	ctr_e = reff - out_fbb ;                        // Calculate the error
	u_ctr = RefPhase.kp * ctr_e + ( RefPhase.xi >> (KI_SHIFT - KP_SHIFT));  //
                                                   calculate the PI_ controller output
	if (u_ctr > RefPhase.limit_out )
	{
		u_ctr = RefPhase.limit_out;             // check the PI_controller
                                                    output is within permissible limit
	}
	else if (u_ctr < -RefPhase.limit_out )
	{
		u_ctr = -RefPhase.limit_out;
	}
	u_ctr >>= KP_SHIFT;
	RefPhase.xi += (RefPhase.ki * ctr_e);     // Integral steps calculation
	if ( RefPhase.xi > RefPhase.limit_int )   // check the Integrator output is
                                                    within permissible limit
	{
		RefPhase.xi = RefPhase.limit_int;
	}
	else if ( RefPhase.xi < -RefPhase.limit_int )
	{
		RefPhase.xi = -RefPhase.limit_int;
	}

	return (s16)u_ctr;
}
```

- **Velcontr.h**

```
#ifndef VELCONTR_H_
#define VELCONTR_H_
#define MAX_SPEED 286 // counts for 0.5 m/s converted to 286 counts/10ms
#define KP_D  7575
#define KI_D  97
#define KP_SHIFT  15          // int 16.15
#define KI_SHIFT  15          // int 16.15
#define LIMIT_INT    4423680  // int 32.15  ( limiter on integrator 135*2^15 )
#define LIMIT_OUT    4423680  // int 32.15  (limiter on PI Output  135*2^15 )

s16 VelContrCalc(s16 reff, s16 out_fbb);
void VelCtrlInit(s16 pgain, s16 igain, s32 limit_i, s32 limit_o);
typedef struct {
      s16 kp;
      s16 ki;
      s32 limit_out;
      s32 limit_int;
      s32 xi;
} PIObj_struct;
#endif /* VELCONTR_H_ */
```

16.4 Appendix - IV (HMI CODE)

```
package prjgui;

import info.monitorenter.gui.chart.Chart2D;
import info.monitorenter.gui.chart.IAxis;
import info.monitorenter.gui.chart.ITrace2D;
import info.monitorenter.gui.chart.traces.Trace2DSimple;
import info.monitorenter.gui.chart.views.ChartPanel;
import javax.swing.JMenuItem;
import java.awt.EventQueue;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import java.awt.BorderLayout;
import javax.swing.JTextPane;
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.FocusAdapter;
import java.awt.event.FocusEvent;
import java.beans.PropertyChangeListener;
import javax.swing.SpringLayout;
import javax.swing.JLabel;
```

```
import javax.swing.JOptionPane;
import javax.swing.SwingConstants;
import javax.swing.JTextField;
import javax.swing.Timer;
import javax.swing.text.BadLocationException;
import javax.swing.text.Document;
import javax.swing.text.SimpleAttributeSet;
import javax.swing.text.StyleConstants;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import java.awt.FlowLayout;
import java.awt.Font;
import java.io.*;
import java.util.Locale;
import javax.swing.border.EmptyBorder;
import javax.swing.border.EtchedBorder;
import prjgui.SerialNetw;
import javax.swing.GroupLayout;
import javax.swing.GroupLayout.Alignment;
import javax.swing.JComboBox;
import javax.swing.BorderFactory;
import javax.swing.DefaultComboBoxModel;
import javax.swing.JButton;
import javax.swing.LayoutStyle.ComponentPlacement;
import javax.swing.JToggleButton;
import javax.swing.border.TitledBorder;
import javax.swing.border.MatteBorder;

public class FpgaCtrlF {

    private static final samplingTime = 0.01;   //To multiply the x-axis of chart
                                                with
                                                  designed sampling rate (10 ms)
    private JFrame FG_frame;
    private JTextField cmd_textField;

    /**
     * Launch the application.
     */

    public static void main(String[] args) {
        SerialNetw.initSerialNetw();
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    FpgaCtrlF window = new FpgaCtrlF();
                    window.FG_frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

    /**
     * Create the application.
     */
    public FpgaCtrlF() {
```

```
        initialize();
    }

    /**
     * Initialize the contents of the frame.
     */

    private void initialize() {
        FG_frame = new JFrame();
        FG_frame.getContentPane().setBackground(new Color(175, 238, 238));
        FG_frame.setBounds(100, 100, 880, 753);
        FG_frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        SpringLayout springLayout = new SpringLayout();
        FG_frame.getContentPane().setLayout(springLayout);

        JPanel Cmd_panel = new JPanel();
        springLayout.putConstraint(SpringLayout.WEST,          Cmd_panel,          10,
        SpringLayout.WEST, FG_frame.getContentPane());
        springLayout.putConstraint(SpringLayout.SOUTH,         Cmd_panel,         -10,
        SpringLayout.SOUTH, FG_frame.getContentPane());
        springLayout.putConstraint(SpringLayout.EAST,          Cmd_panel,         -10,
        SpringLayout.EAST, FG_frame.getContentPane());
        Cmd_panel.setBorder(new EtchedBorder(EtchedBorder.LOWERED, null, null));
        FlowLayout flowLayout = (FlowLayout) Cmd_panel.getLayout();
        flowLayout.setHgap(10);
        flowLayout.setAlignment(FlowLayout.LEFT);
        FG_frame.getContentPane().add(Cmd_panel);
        JScrollPane scrollPane = new JScrollPane();
        springLayout.putConstraint(SpringLayout.EAST, scrollPane, -10,
        SpringLayout.EAST, FG_frame.getContentPane());
        JLabel lblNewLabel = new JLabel("cmd>> ");
        lblNewLabel.setHorizontalAlignment(SwingConstants.LEFT);
        Cmd_panel.add(lblNewLabel);
        cmd_textField = new JTextField();
        cmd_textField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                CommandHandler(cmd_textField.getText());
                cmd_textField.setText("");
            }
        });

        /**
     * These are the textFields, buttons, comboBox and other declaration of GUI with
       methods consisting of their functions.
     */

        Cmd_panel.add(cmd_textField);
        cmd_textField.setColumns(30);
        FG_frame.getContentPane().add(scrollPane);
        Gr_panel = new JPanel();
        springLayout.putConstraint(SpringLayout.NORTH, Cmd_panel, 12,
        SpringLayout.SOUTH, Gr_panel);
        springLayout.putConstraint(SpringLayout.EAST, Gr_panel, 0,
        SpringLayout.EAST, Cmd_panel);
        JButton btnClear = new JButton("Clear window");
        Cmd_panel.add(btnClear);
        btnClear.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                CommandHandler("clc");
            }
```

```
});
springLayout.putConstraint(SpringLayout.SOUTH, Gr_panel, -58,
SpringLayout.SOUTH, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.WEST, Gr_panel, 10,
SpringLayout.WEST, FG_frame.getContentPane());
tout_textPane = new JTextPane();
tout_textPane.setEditable(false);
scrollPane.setViewportView(tout_textPane);
FG_frame.getContentPane().add(Gr_panel);
panel = new JPanel();
springLayout.putConstraint(SpringLayout.WEST, panel, 10,
SpringLayout.WEST, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.EAST, panel, -245,
SpringLayout.EAST, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.WEST, scrollPane, 6,
SpringLayout.EAST, panel);
panel.setBackground(new Color(173, 216, 230));
springLayout.putConstraint(SpringLayout.NORTH, panel, 10,
SpringLayout.NORTH, FG_frame.getContentPane());
panel.setBorder(new TitledBorder(new MatteBorder(1, 1, 1, 1,
(Color) new Color(0, 0, 255)), "Connection Phase",
TitledBorder.LEADING, TitledBorder.TOP, null, Color.BLUE));
FG_frame.getContentPane().add(panel);
JComboBox comboBox = new JComboBox();
comboBox.setModel(new DefaultComboBoxModel(new String[] {"COM1",
"COM2"}));
JButton btnDevice = new JButton("Device");
btnDevice.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String dev_name = "com1";
        int stat;
        PrintTxtWin("dev...", 0, true);
        for (int k = 0; k < SerialNetw.getNDev(); k++)
        {
            dev_name = SerialNetw.getDevName(k);
            stat = SerialNetw.getDevStat(dev_name);
            if (stat == 2)
            {
                PrintTxtWin("  " + dev_name + " [connected]", 4,
                true);
            }
            else
            {
                PrintTxtWin("  " + dev_name + " [avail]", 4, true);
            }
        }
    }
});

JButton btnConnect = new JButton("Connect");
btnConnect.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String command1 = "conn ";
        String dev_name1 = "COM1";
        if (SerialNetw.getDevStat(dev_name1) != 1) {
            PrintTxtWin("*** device unavailable", 3, true);
        } else {
            (new SerialNetw()).spConnect(dev_name1);
            PrintTxtWin("OK", 1, true);
        }
```

```
    }
});
JButton btnDisconnect = new JButton("Disconnect");
btnDisconnect.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String dev_name = SerialNetw.getConName();
        if (dev_name != null) {
            SerialNetw.spDisconn(dev_name);
            PrintTxtWin("OK", 1, true);
        } else {
            PrintTxtWin("*** no device connected", 3, true);
        }
    }
});
GroupLayout gl_panel = new GroupLayout(panel);
gl_panel.setHorizontalGroup(
    gl_panel.createParallelGroup(Alignment.LEADING)
        .addGroup(gl_panel.createSequentialGroup()
            .addGap(84)
            .addComponent(comboBox, GroupLayout.PREFERRED_SIZE, 73,
             GroupLayout.PREFERRED_SIZE)
            .addGap(26)
            .addComponent(btnDevice)
            .addGap(32)
            .addComponent(btnConnect)
            .addGap(40)
            .addComponent(btnDisconnect)
            .addContainerGap(64, Short.MAX_VALUE))
);
gl_panel.setVerticalGroup(
    gl_panel.createParallelGroup(Alignment.TRAILING)
        .addGroup(Alignment.LEADING,
         gl_panel.createSequentialGroup()
         .addGroup(gl_panel.createParallelGroup(Alignment.BASELINE)
         .addComponent(comboBox, GroupLayout.PREFERRED_SIZE,
               GroupLayout.DEFAULT_SIZE,GroupLayout.PREFERRED_SIZE)
              .addComponent(btnDevice, GroupLayout.DEFAULT_SIZE,
               GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
              .addComponent(btnConnect, GroupLayout.DEFAULT_SIZE,
               GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
              .addComponent(btnDisconnect,
               GroupLayout.DEFAULT_SIZE,
               GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                .addContainerGap(25, Short.MAX_VALUE))
);
panel.setLayout(gl_panel);

panel_1 = new JPanel();
springLayout.putConstraint(SpringLayout.NORTH, panel_1, 67,
SpringLayout.NORTH, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.WEST, panel_1, 10,
SpringLayout.WEST, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.EAST, panel_1, -6,
SpringLayout.WEST, scrollPane);
springLayout.putConstraint(SpringLayout.SOUTH, panel, -6,
SpringLayout.NORTH, panel_1);
panel_1.setBackground(new Color(173, 216, 230));
panel_1.setBorder(new TitledBorder(new MatteBorder(1, 1, 1, 1,
(Color) new Color(0, 0, 255)),
"Reference Phase", TitledBorder.LEADING, TitledBorder.TOP, null,
```

```
Color.BLUE));
FG_frame.getContentPane().add(panel_1);
JLabel lblReferenceSpeed = new JLabel("Reference speed");
textField = new JTextField();
textField.setColumns(10);
JButton btnReference = new JButton("Reference");
btnReference.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        txtRefPos.setText("0.0");
        txtForce.setText("0.0");
        txtPendulumAngle.setText("0.0");
        txtCartDisp.setText("0.0");
        String refString = "refsp";
        SerialNetw.TxDataCt.sendString(refString.substring(1) +
        "\n");
    }
});
GroupLayout gl_panel_1 = new GroupLayout(panel_1);
gl_panel_1.setHorizontalGroup(
    gl_panel_1.createParallelGroup(Alignment.LEADING)
        .addGroup(gl_panel_1.createSequentialGroup()
            .addGap(103)
            .addComponent(lblReferenceSpeed)
            .addGap(26)
            .addComponent(textField, GroupLayout.PREFERRED_SIZE,
             91, GroupLayout.PREFERRED_SIZE)
            .addGap(42)
            .addComponent(btnReference)
            .addContainerGap(148, Short.MAX_VALUE))
);
gl_panel_1.setVerticalGroup(
    gl_panel_1.createParallelGroup(Alignment.LEADING)
        .addGroup(Alignment.TRAILING,
         gl_panel_1.createSequentialGroup()
            .addContainerGap(14, Short.MAX_VALUE)
             .addGroup(gl_panel_1.createParallelGroup
            (Alignment.BASELINE)
                .addComponent(textField,
           GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
           GroupLayout.PREFERRED_SIZE)
                .addComponent(btnReference)
                .addComponent(lblReferenceSpeed))
             .addContainerGap())
);
panel_1.setLayout(gl_panel_1);
panel_2 = new JPanel();
springLayout.putConstraint(SpringLayout.NORTH, panel_2, 145,
SpringLayout.NORTH, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.WEST, panel_2, 10,
SpringLayout.WEST, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.SOUTH, panel_2, -449,
SpringLayout.SOUTH, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.EAST, panel_2, -6,
SpringLayout.WEST, scrollPane);
springLayout.putConstraint(SpringLayout.SOUTH, panel_1, -6,
SpringLayout.NORTH, panel_2);
panel_2.setBackground(new Color(173, 216, 230));
panel_2.setBorder(new TitledBorder(new MatteBorder(1, 1, 1, 1,
(Color) new Color(0, 0, 255)), "CP Run Phase", TitledBorder.LEADING,
TitledBorder.TOP, null, new Color(0, 0, 255)));
```

```
FG_frame.getContentPane().add(panel_2);
panel_3 = new JPanel();
springLayout.putConstraint(SpringLayout.WEST, panel_3, 10,
SpringLayout.WEST, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.EAST, panel_3, -6,
SpringLayout.WEST, scrollPane);
springLayout.putConstraint(SpringLayout.SOUTH, scrollPane, 0,
SpringLayout.SOUTH, panel_3);
springLayout.putConstraint(SpringLayout.NORTH, Gr_panel, 21,
SpringLayout.SOUTH, panel_3);
springLayout.putConstraint(SpringLayout.SOUTH, panel_3, -391,
SpringLayout.SOUTH, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.NORTH, panel_3, 6,
SpringLayout.SOUTH, panel_2);
panel_3.setBackground(new Color(173, 216, 230));
panel_3.setBorder(new TitledBorder(new MatteBorder(1, 1, 1, 1,
(Color) new Color(0, 0, 255)), "Graph Phase", TitledBorder.LEADING,
TitledBorder.TOP, null, new Color(0, 0, 255)));
JLabel lblCartPositionm = new JLabel("Cart position(m)");
JLabel lblNewLabel_1 = new JLabel("Cart force(N)");
txtCartDisp = new JTextField();
txtCartDisp.setColumns(10);
JLabel lblNewLabel_2 = new JLabel("Pend. angle(deg)");
txtPendulumAngle = new JTextField();
txtPendulumAngle.setColumns(10);
JLabel lblRefPosition = new JLabel("Ref. position(m)");
textField_4 = new JTextField();
textField_4.setColumns(10);
txtForce = new JTextField();
txtForce.setColumns(10);
txtRefPos = new JTextField();
txtRefPos.setColumns(10);
JComboBox comboBox_1 = new JComboBox();
comboBox_1.setModel(new DefaultComboBoxModel(new String[] {"MODE
ON", "MODE OFF"}));
btnOk = new JButton("OK");
btnOk.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(comboBox_1.getSelectedItem().equals("MODE ON") )
        {
            String refString = ".mode on";
            SerialNetw.TxDataCt.sendString(refString.substring(1) +
            "\n");
        }
        else
        {
            String refString = ".mode off";
            SerialNetw.TxDataCt.sendString(refString.substring(1) +
            "\n");
        }
    }
});
btnStop = new JButton("Stop");
btnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String exitString = ".exit";
        SerialNetw.TxDataCt.sendString(exitString.substring(1) +
        "\n");
    }
});
```

```
btnExecute = new JButton("Execute");
btnExecute.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        String CP_command = ".cp " + Ref_int;
        CommandHandler(CP_command);
    }
});
txtRefPos.addFocusListener(new FocusAdapter(){
    @Override
    public void focusLost(FocusEvent arg0)
    {
    Ref_int = Float.parseFloat(txtRefPos.getText());
    Ref_int *= 1/X_PHY;
    }
    });
GroupLayout gl_panel_2 = new GroupLayout(panel_2);
gl_panel_2.setHorizontalGroup(
    gl_panel_2.createParallelGroup(Alignment.LEADING)
        .addGroup(gl_panel_2.createSequentialGroup()
            .addContainerGap()
        .addGroup(gl_panel_2.createParallelGroup(Alignment.LEADING)
                .addComponent(lblCartPositionm)
                .addComponent(lblNewLabel_1))
            .addPreferredGap(ComponentPlacement.RELATED)
       .addGroup(gl_panel_2.createParallelGroup(Alignment.LEADING,
        false)
           .addComponent(txtForce, 0, 0, Short.MAX_VALUE)
           .addComponent(txtCartDisp, GroupLayout.PREFERRED_SIZE,
            69, GroupLayout.PREFERRED_SIZE))
            .addGap(11)
        .addGroup(gl_panel_2.createParallelGroup(Alignment.LEADING,
        false)
        .addGroup(gl_panel_2.createSequentialGroup()
          .addComponent(lblNewLabel_2)
             .addPreferredGap(ComponentPlacement.UNRELATED)
                .addComponent(txtPendulumAngle,
GroupLayout.PREFERRED_SIZE, 74, GroupLayout.PREFERRED_SIZE))
                .addGroup(gl_panel_2.createSequentialGroup()
                    .addComponent(lblRefPosition)
                    .addGap(18)
                    .addComponent(txtRefPos, 0, 0,
                    Short.MAX_VALUE)))
                    .addGap(32)

       .addGroup(gl_panel_2.createParallelGroup(Alignment.TRAILING)
                .addComponent(btnExecute, GroupLayout.DEFAULT_SIZE,
                83, Short.MAX_VALUE)
                .addComponent(comboBox_1, 0, 83, Short.MAX_VALUE))
            .addPreferredGap(ComponentPlacement.RELATED)

       .addGroup(gl_panel_2.createParallelGroup(Alignment.LEADING,
          false)
                .addComponent(btnStop, GroupLayout.DEFAULT_SIZE,
GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .addComponent(btnOk, GroupLayout.DEFAULT_SIZE, 79,
                 Short.MAX_VALUE))
            .addPreferredGap(ComponentPlacement.RELATED)
            .addComponent(textField_4, GroupLayout.PREFERRED_SIZE,
            0, GroupLayout.PREFERRED_SIZE)
```

```
                .addGap(15))
        );
        gl_panel_2.setVerticalGroup(
            gl_panel_2.createParallelGroup(Alignment.LEADING)
                .addGroup(gl_panel_2.createSequentialGroup()
                    .addContainerGap()
                .addGroup(gl_panel_2.createParallelGroup(Alignment.LEADING)
                        .addGroup(gl_panel_2.createSequentialGroup()
                            .addComponent(txtCartDisp,
        GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
        GroupLayout.PREFERRED_SIZE)
                            .addPreferredGap(ComponentPlacement.UNRELATED)
            .addGroup(gl_panel_2.createParallelGroup(Alignment.BASELINE)
                            .addComponent(txtForce,
        GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
        GroupLayout.PREFERRED_SIZE)
                            .addComponent(lblRefPosition)
                            .addComponent(txtRefPos,
        GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
        GroupLayout.PREFERRED_SIZE)
                            .addComponent(btnExecute)
                            .addComponent(btnStop)
                            .addComponent(lblNewLabel_1)))

            .addGroup(gl_panel_2.createParallelGroup(Alignment.LEADING)
                            .addComponent(textField_4,
        GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
        GroupLayout.PREFERRED_SIZE)
                            .addComponent(lblCartPositionm))
                        .addComponent(lblNewLabel_2)

            .addGroup(gl_panel_2.createParallelGroup(Alignment.BASELINE)
                            .addComponent(txtPendulumAngle,
        GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
        GroupLayout.PREFERRED_SIZE)
                            .addComponent(comboBox_1,
        GroupLayout.PREFERRED_SIZE, 24, GroupLayout.PREFERRED_SIZE)
                            .addComponent(btnOk)))
                    .addContainerGap(16, Short.MAX_VALUE))
        );
        panel_2.setLayout(gl_panel_2);
        FG_frame.getContentPane().add(panel_3);
        btnDownload = new JButton("Download");
        btnDownload.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                CommandHandler(".p");
            }
        });
        btnPlot = new JButton("Plot");
        btnPlot.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                CommandHandler("plot");
            }
        });
        btnLogClear = new JButton("Log Clear");
        btnLogClear.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                CommandHandler(".log clr");
            }
        });
```

```
btnExit = new JButton("Exit");
btnExit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String dev_name2 = SerialNetw.getConName();
        if (dev_name2 != null) {
            SerialNetw.spDisconn(dev_name2);
        }
        DispUpdate_Timer.stop();
        System.exit(0);
    }
});
JToggleButton tglbtnLoggerOnoff = new JToggleButton("Logger
ON/OFF");
tglbtnLoggerOnoff.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
            if(tglbtnLoggerOnoff.isSelected())
            {
            CommandHandler(".log on");
            }
            else
            {
            CommandHandler(".log off");
            }
    }
});
GroupLayout gl_panel_3 = new GroupLayout(panel_3);
gl_panel_3.setHorizontalGroup(
    gl_panel_3.createParallelGroup(Alignment.LEADING)
        .addGroup(gl_panel_3.createSequentialGroup()
            .addGap(57)
            .addComponent(tglbtnLoggerOnoff)
            .addGap(18)
            .addComponent(btnDownload)
            .addGap(18)
            .addComponent(btnPlot)
            .addGap(18)
            .addComponent(btnLogClear)
            .addPreferredGap(ComponentPlacement.UNRELATED)
            .addComponent(btnExit)
            .addContainerGap(71, Short.MAX_VALUE))
);
gl_panel_3.setVerticalGroup(
    gl_panel_3.createParallelGroup(Alignment.LEADING)
     .addGroup(gl_panel_3.createSequentialGroup()
     .addGroup(gl_panel_3.createParallelGroup(Alignment.BASELINE)
                .addComponent(tglbtnLoggerOnoff)
                .addComponent(btnDownload)
                .addComponent(btnPlot)
                .addComponent(btnLogClear)
                .addComponent(btnExit))
            .addContainerGap(GroupLayout.DEFAULT_SIZE,
             Short.MAX_VALUE))
);
panel_3.setLayout(gl_panel_3);
lblCommandWindow = new JLabel("Command Window");
springLayout.putConstraint(SpringLayout.SOUTH, lblCommandWindow, -
682, SpringLayout.SOUTH, FG_frame.getContentPane());
springLayout.putConstraint(SpringLayout.NORTH, scrollPane, 6,
```

```
        SpringLayout.SOUTH, lblCommandWindow);
        springLayout.putConstraint(SpringLayout.EAST, lblCommandWindow, -
        73, SpringLayout.EAST, FG_frame.getContentPane());
        FG_frame.getContentPane().add(lblCommandWindow);
        CreateChart();
        DispUpdate_Timer = new Timer(250, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                UpdateDynamic();
                DispUpdate_Timer.restart();
            }
        });
        DispUpdate_Timer.start();
        Downl_Cnt = 0;
        Pb_NValues = 0;
        Pb_Ready = false;
    }

    /**
     * This method is used to specify the commands and its functions to perform the
       actions specified in the GUI
     */

    private void CommandHandler(String cmds) {
        String command = cmds, dev_name, fname;
        int  k, q, stat;
        Writer out = null;

        if (command.equals("clc")) {
            tout_textPane.setText("");
        } else if (command.equals("clg")) {
            ClearChart();
        } else if (command.equals("help")) {
            PrintTxtWin("FPGA Control Help:", 1, true);
            PrintTxtWin("    clc - clear text window", 1, true);
            PrintTxtWin("    clg - clear chart window", 1, true);
            PrintTxtWin("    dev - list avaliable devices", 1, true);
            PrintTxtWin("    conn {comX} - connect", 1, true);
            PrintTxtWin("    disconn - disconnect", 1, true);
            PrintTxtWin("    .{sendstring}", 1, true);
        } else if (command.equals("dev")) {
            PrintTxtWin("dev...", 0, true);
            for (k = 0; k < SerialNetw.getNDev(); k++) {
                dev_name = SerialNetw.getDevName(k);
                stat = SerialNetw.getDevStat(dev_name);
                if (stat == 2) {
                    PrintTxtWin("  " + dev_name + " [connected]", 4, true);
                } else {
                    PrintTxtWin("  " + dev_name + " [avail]", 4, true);
                }
            }
        } else if (command.startsWith("conn ")) {
            dev_name = command.substring(5).toUpperCase();
            if (SerialNetw.getDevStat(dev_name) != 1) {
                PrintTxtWin("*** device unavailable", 3, true);
            } else {
                (new SerialNetw()).spConnect(dev_name);
                PrintTxtWin("OK", 1, true);
            }
        } else if (command.equals("disconn")) {
```

```
        dev_name = SerialNetw.getConName();
        if (dev_name != null) {
            SerialNetw.spDisconn(dev_name);
            PrintTxtWin("OK", 1, true);
        } else {
            PrintTxtWin("*** no device connected", 3, true);
        }
    } else if (command.equals("exit")) {
        dev_name = SerialNetw.getConName();
        if (dev_name != null) {
            SerialNetw.spDisconn(dev_name);
        }
        DispUpdate_Timer.stop();
        System.exit(0);
    } else if (command.equals("downl")) {
        if (SerialNetw.getConName() != null) {
            Downl_Cnt = 0;
            Pb_NValues = 0;
            Pb_Ready = false;
            SerialNetw.TxDataCt.sendString("p\n");
            PrintTxtWin(" => starting download:\n ", 1, false);
        } else {
            PrintTxtWin("*** no connected device", 3, true);
        }
    } else if (command.startsWith("plot")) {
        if (!Pb_Ready) {
            PrintTxtWin("*** no plot data available", 3, true);
        } else if (Pb_NValues < 2) {
            PrintTxtWin("*** no data points", 3, true);
        } else {
            if (command.equals("plot")) {
                ClearChart();
                for (k = 0; k < Pb_NValues; k++) {
                    if (Pb_NValues < NROWS) {
                        for (q = 0; q < NTRACES; q++) {
                            switch(q)
                            {
                                case REF_POS: Plot_Buffer[k][q] *=
                                X_PHY;
                                break;
                                case DISP: Plot_Buffer[k][q] *= X_PHY;
                                break;
                                case FORCE: Plot_Buffer[k][q] *= F_PHY;
                                break;
                                case ANGLE : Plot_Buffer[k][q] *=
                                ALPHA_PHY;
                                break;
                                case ANG_VEL: Plot_Buffer[k][q] *=
                                OMEGA_PHY;
                                break;
                            }
                            mtraces[q].addPoint((double)k * samplingTime,
                            Plot_Buffer[k][q]);
                        }
                    }
                }
            } else {
                if (command.length() < 6) {
                    PrintTxtWin("*** no file name given", 3, true);
                } else {
```

```
                    fname = command.substring(5);
                    PrintTxtWin(" => attemp to write file \"" + fname +
                    "\"", 1, true);
                    try {
                        out = new OutputStreamWriter(new
                              FileOutputStream("\\Temp\\" + fname));
                        for (k = 0; k < Pb_NValues; k++) {
                            out.write(String.format(Locale.ENGLISH, "%f
                            %f \n",Plot_Buffer[k][0],Plot_Buffer[k][1],
                            Plot_Buffer[k][2],Plot_Buffer[k][3]));
                        }
                        out.close();
                    } catch (Exception ex) {
                        PrintTxtWin(ex.toString(), 3, true);
                    }
                }
            }
        }
    } else if (command.startsWith(".")) {
        if (SerialNetw.getConName() != null) {
            SerialNetw.TxDataCt.sendString(command.substring(1) + "\n");
        } else {
            PrintTxtWin("*** no connected device", 3, true);
        }
    } else if (command.length() > 0) {
        PrintTxtWin("*** command???: \"" + command + "\"", 3, true);
    }
}

/**
 * This method is used to display the outputs on the Command window when a user
   enters or clicks any buttons.
 */
private void PrintTxtWin(String twstr, int twstyle, boolean newline) {
    try {
        Document doc = tout_textPane.getStyledDocument();
        StyleConstants.setItalic(TextSet, false);
        StyleConstants.setBold(TextSet, false);
        StyleConstants.setForeground(TextSet, Color.BLACK);
        switch (twstyle) {
            case 0:
                StyleConstants.setBold(TextSet, true);
                StyleConstants.setForeground(TextSet, Color.DARK_GRAY);
                break;
            case 1: StyleConstants.setForeground(TextSet, Color.BLUE);
                break;
            case 2: StyleConstants.setForeground(TextSet, Color.BLACK);
                break;
            case 3: StyleConstants.setForeground(TextSet, Color.RED);
                break;
            case 4: StyleConstants.setForeground(TextSet, Color.GREEN);
                break;
            default:
                doc.remove(0, doc.getLength());
        }
        if (twstyle >= 0) {
            tout_textPane.setCharacterAttributes(TextSet, true);
            if (newline) {
                doc.insertString(doc.getLength(), twstr+"\n", TextSet);
            } else {
```

```
                doc.insertString(doc.getLength(), twstr, TextSet);
            }
        }
    } catch (BadLocationException ex) {
        System.out.println(ex.toString());
    }
}
private void ClearChart()
{
    int  k;
    for (k = 0; k < NTRACES; k++) {
        mtraces[k].removeAllPoints();
        mtraces[k].addPoint(0.0, 0.0);
    }
}

/**
 * This method is used to create a chart and traces in the graph with the
   declaration of grids.
 */

private void CreateChart()
{
    int  k;
    chart = new Chart2D();
    for (k = 0; k < NTRACES; k++) {
        mtraces[k] = new Trace2DSimple();
        chart.addTrace(mtraces[k]);
    }
    mtraces[0].setColor(Color.blue);     mtraces[0].setName("Reference
    Position");
    mtraces[1].setColor(Color.red);
    mtraces[1].setName("Displacement");
    mtraces[2].setColor(Color.green);    mtraces[2].setName("Force");
    mtraces[3].setColor(Color.magenta);  mtraces[3].setName("Angle");
    chart.setPaintLabels(true);
    chart.setGridColor(BLACK);
    Gr_panel.setLayout(new BorderLayout(0, 0));
    Gr_panel.add(chart);
    Gr_panel.setSize(100,200);
    chart.setVisible(true);
    Gr_panel.setVisible(true);
    Gr_panel.repaint();
}
public javax.swing.JMenuItem createChartGridMenu(ChartPanel chartPanel,
        boolean adaptUI2Chart)
{
    return null;
}

/**
 * This method is used to update the real time data on the textfields.
 */

private void UpdateStateData()
{
    String force =
    String.format("%5.5s",String.valueOf(data[1]*F_PHY));
    String Angle =
    String.format("%5.5s",String.valueOf(data[2]*ALPHA_PHY));
```

```
        String disp = String.format("%5.5s",String.valueOf(data[3]*X_PHY));
        txtForce.setText(force);
        txtPendulumAngle.setText(Angle);
        txtCartDisp.setText(disp);
    }

    /**
     * This method is used to receive the data from the FPGA, compare with the
       string for the correct data and store in the buffer.
     */

    private void UpdateDynamic() {
        String  recv_s;
        String  splits[];
        int  k;

        while ((recv_s = SerialNetw.RxDataCt.getstring()) != null) {
            // System.out.println("["+ recv_s + "]");
            if (recv_s.startsWith("$") || recv_s.startsWith("~")) {
                PrintTxtWin("+", 1, false);
                Downl_Cnt++;
                if (Downl_Cnt >= 25) {
                    Downl_Cnt = 0;
                    PrintTxtWin("\n    ", 1, false);
                }
                if (Pb_NValues < NROWS) {
                    splits = recv_s.split(" ");
                    try {
                        for (k = 1; k < splits.length; k++) {
                            Plot_Buffer[Pb_NValues][k-1] =
                            (double)Integer.parseInt(splits[k]);
                        }
                        Pb_NValues++;
                    } catch  (NumberFormatException e) {
                        System.out.println(e.toString());
                    }
                }
                if (recv_s.startsWith("~")) {
                    SerialNetw.TxDataCt.sendString("p\n");
                }
                else if(recv_s.startsWith("?"))
                {
                    splits = recv_s.split(" ");
                    for (k = 1; k < splits.length; k++)
                    {
                        data[k] = (float)Integer.parseInt(splits[k]);
                        UpdateStateData();
                    }
                }
            } else if (recv_s.startsWith("###")) {
                PrintTxtWin("\nDownload finished.", 1, true);
                Pb_Ready = true;
            } else {
                PrintTxtWin(recv_s, 0, true);
                if(recv_s.equals("Reference Phase Completed "))
                {
                    REF_PHASE_DONE = 1;
                }
                else if( REF_PHASE_DONE ==1 && recv_s.equals(" o set up
                        timer interrupt..."))
```

```
                {
                    REF_PHASE_DONE = 0;
                    txtRefPos.setText("0.0");
                    txtForce.setText("0.0");
                    txtPendulumAngle.setText("0.0");
                    txtCartDisp.setText("0.0");
                }
            }
        }
    }

    static int Downl_Cnt;
    static boolean Pb_Ready;
    static final int NROWS = 2048, NCOLS = 4;
    static double Plot_Buffer[][] = new double[NROWS][NCOLS];
    static int Pb_NValues;
    static Chart2D chart;
    static final int NTRACES = 4;
    static ITrace2D mtraces[] = new ITrace2D[4];
    private SimpleAttributeSet TextSet = new SimpleAttributeSet();
    private Timer DispUpdate_Timer;
    private JPanel Gr_panel;
    private JTextPane tout_textPane;
    private JPanel panel;
    private JPanel panel_1;
    private JPanel panel_2;
    private JPanel panel_3;
    private JTextField textField;
    private JTextField txtCartDisp;
    private JTextField txtPendulumAngle;
    private JTextField textField_4;
    private JTextField txtForce;
    private JTextField txtRefPos;
    private JButton btnOk;
    private JButton btnStop;
    private JButton btnExecute;
    private JButton btnDownload;
    private JButton btnPlot;
    private JButton btnLogClear;
    private JButton btnExit;
    private JLabel lblCommandWindow;

    /**
     * These are the initialization of the constants used for the physical to
       integer conversions for inputting to the FPGA.
     */
    static final int REF_POS = 0, FORCE = 4, ANGLE = 2, DISP = 1, CART_VEL = 5;
    static final double X_0 = 1, X_I = 57238, F_0 = 1, F_I= 22.0532;
    static final double PI = 22/7;
    static final double ALPHA_0 = 360, ALPHA_I = 4096;
    static final double VEL_I = 286.19 ,VEL_0 = 1;
    static final double OMEGA_0 = 1, OMEGA_I =40.9600;
    static final double OMEGA_PHY= OMEGA_0/OMEGA_I;
    static final double VEL_PHY = VEL_0/VEL_I;
    static final double ALPHA_PHY = ALPHA_0/ALPHA_I;
    static final double X_PHY = X_0/X_I;
    static final double F_PHY = F_0/F_I;
    static float Ref_int, Ref_prev = 0;
    static int REF_PHASE_DONE =0;
    static float data[] = new float[4];
```

```
}
```