Microprocessor Systems and Control Systems Lab Project Report

Submitted by:

Mehrunisa Ashraf	2017-EE-2
Ayesha Sana	2017-EE-8
Haris Rafique	2017-EE-24
Shahkar Ul Hassan	2017-EE-36

Supervised by: Sir Umer Shahid
Sir Ali Shafique

Department of Electrical Engineering

University of Engineering and Technology Lahore

Microprocessor Systems and Control Systems Lab Project Report

Submitted to the faculty of the Electrical Engineering Department
of the University of Engineering and Technology Lahore
in partial fulfillment of the requirements for the Degree of

Bachelor of Science

in

Electrical Engineering.

Internal Examiner	External Examiner

Director
Undergraduate Studies

Department of Electrical Engineering

University of Engineering and Technology Lahore

Declaration

I declare that the work contained in this thesis is my own, except where explicitly stated otherwise. In addition this work has not been submitted to obtain another degree or professional qualification.

Signed: ____________________

Date: ____________________

Acknowledgments

We were successfully able to make this project under the supervision of qualified professors and competent instructors who guided us throughout the composition of this project.

Sir K.M.Hasan
Sir Umer Shahid
Sir Ali Shafique

Contents

List of Figures

List of Tables

Abbreviations

PID	**P**roportional **I**ntegral **D**ifferential
LFR	**L**ine **F**ollowing **R**obot
GPIO	**G**eneral **P**urpose **I**nput **O**utput
PWM	**P**ulse **W**idth **M**odulation

Abstract

The line follower robot is a machine which can detect and follow the line drawn. The path is predefined and is visible like a black line on a white surface. It has a line following module attached to it which is responsible to sense the input from the track (which is the black/white line). As a result, the data is transmitted to the processor by specific transition buses. The microprocessor decides what to do in response to an interrupt (from the track) and is assisted by the PID controller One of the prime objectives of this project is to investigate the application of the Proportional Derivative Integral (PID for short) controller in our daily lives. In this scenario, we have decided to implement such a controller to construct a line-following robot, which has its sole purpose of following a custom-made track of a distinct colour (taking black here with a white layout) without any disturbances or unexpected results.

Chapter 1

Introduction

The essence of this project is to investigate the application of the Proportional Derivative Integral (PID for short) controller in our daily lives. The presence of a PID controller in a closed loop system has many benefits, such as giving an output that is close to our set point (desired output) that we have defined earlier, as well as inducing stability in the system. In this scenario, we have decided to implement such a controller to construct a line-following robot, which has its sole purpose of following a custom-made track of a distinct colour (taking black here) without any disturbances or unexpected results. Apart from implementing a PID controller, an interrupt-based protocol (techincally known as input/output synchronization) is also present and is embedded in the Stellaris TIVA C series (LM4F120H5QR) microcontroller which is essential to make any tricky turns or avoid any obstacles which may or may not be present on the track. How this algorithm is implemented in the microcontroller will be discussed later on.

The line following robot has a practical tendency of speeding up and spinning out of control on the track (in control terms, we may attain an unsteady output). Therefore, for control engineers, it is imperative to install a PID controller, that is based on the feedback theory. When a closed-loop system for the car is established, the error in the output is subtracted from the set point or desired output of the car (which we provide as the input) and fed into the PID controller. This error is gradually eliminated from the system, changing the robots actions at the same time, which will eventually behave as we expect it to do. However, to achieve these sort of results, we need to do some dirty work regarding the parameters of the PID controller, which is commonly known as tuning of the controller in which we take random values of the proportional, derivative and integral constants (denoted by Kp, Kd and KI) until we attain the values that are suitable for the desired output. The details of this procedure will be described later as well in chapter 6.

Furthermore, synchronization between the processor execution and the peripheral interfaces is required to construct an efficient embedded system. These interfaces play a significant role in realizing the coupling between the processor and the external world. Synchronization may be achieved by having an interrupt-based system, as already mentioned. An interrupt, in simple terms, is the indication of the occurrence of a certain event that may cause the processor to deviate from its current activities or executions. For example, if the robot is running on a straight track and a turn approaches, this turn is the interrupt to which the processor has to respond to. The input is taken from the IR sensors (details discussed further on in this report), and fed into the processor, which causes it to undergo an interrupt service routine, which works to produce the appropriate response to this event, that is, being able to make the turn without going out of track. Other ways of implementing input/output synchronization include continuous polling or periodic polling, but we render them redundant in the scope of this project.

The question may arise: why do we need to make such line-following robots? One surprising application includes being used in making the health-based management system, notably in Indian hospitals, in order to improve monitoring the health of various patients in hospitals, whenever they need treatment or not. Statistics showed earlier that around the world thousands of patients died because of not properly monitoring them because of few numbers of trained medical staff. To eradicate these issues, these management systems have caused a lot of hospitals stop recruiting nurses and medical personnel since 2005 and 2006. In addition, the line-following robot symbolizes real-life transportation from one place to another via a predesigned path (usually by trains). Every autonomous car needs a pre-defined path to travel on, as well the running efficiency needs to be taken into account, that involves speed and position control.

Many line-following robots, in the absence of advanced control design, have to run slowly because they cannot correct the deviation smoothly. Most robots use PIDs can work well, but PID has many restrictions. Thus, there is a need for the LFR to be more advanced. In the real world, for instance, if a car is speeding on the road and a turn is coming, it has to slow down to avoid sliding. However, most LFRs cannot brake or brake well due not only to the limitations of the position control of the robot, but also to that of the speed control.

Chapter 2

Equipments

Here are the descriptions of some of the important components used in the project:

4N35 Optocouplers:
an optocoupler is like a small IC chip that is used to provide electrical isolation between two circuits that are operating at different voltages by the means of a light-sensitive interface. This ultimately prevents loading and voltage spikes in the overall circuit, that can have negative consequences on the electrical devices concerned. Here, we have used optocouplers for the protection of the TIVA microcontroller from the motors of the car which will operate at around 8 volts. The microcontroller, on the other hand will operate at a maximum of 5 volts. Any voltage higher than this threshold can damage it, thus it is essential to use the optocouplers according to its correct configuration. The diagram for the optocoupler is also provided

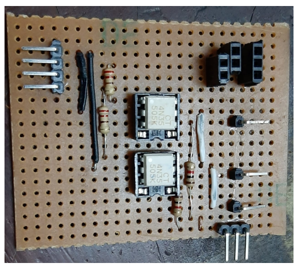


Figure 2.1: Optocoupler

L298H Motor Driver
The motor driver is normally used to draw a high current and step it down into a current that is appropriate for the regular operation of the motors. It has the capability to take in a DC voltage, ranging from 7 to 24 volts. Apart from this, there are 6 enable pins, 2 for conrolling the direction of each motor for the car and 1 each that are called PWM

pins that will be integral for controlling the speed of the motors at different parts of the track.

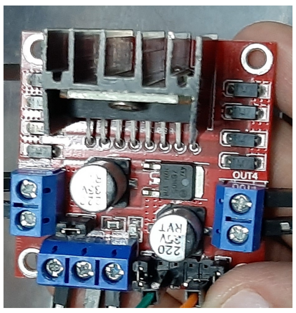


Figure 2.2: L298N

LM2596 DC-DC Buck Converter
This component is used to step down a higher voltage (around 4.5 to 53V) to a lower voltage that can range from 3 to 40 volts. Here, we need a maximum voltage of 8 volts to operate the motors from a higher input of 14.8 volts(which will be provided by the lithium batteries). In addition, this module has a maximum current rating of 3 amperes.

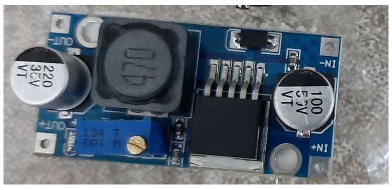


Figure 2.3: LM2596 DC-DC Buck Converter

5 channel TCRT 5000 line following module
This line following module works on the principle of transmitting infrared light from the LED and registering any reflected light on its photo transistor this alters the flow of current between its emitter and collector according to the level of light it receives. As we know that white surfaces reflect light and black surfaces absorb light, the IR light is reflected back to the module from the white surface around the black line. But IR radiation is absorbed completely by black color. Based on this, we can control when the car should stop or keep on going on the track. Specifications of this module include an input voltage, ranging from 3.3 to 5 volts and being able to work up to a distance of 3cm from the surface.

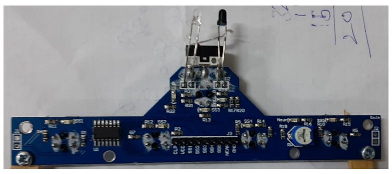


Figure 2.4: TCRT5000 line following module

Pictures of the other common components are attached with this document.

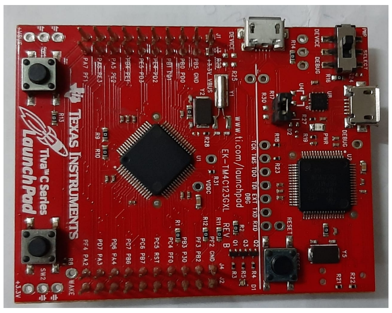


Figure 2.5: Stellaris TIVA microcontroller

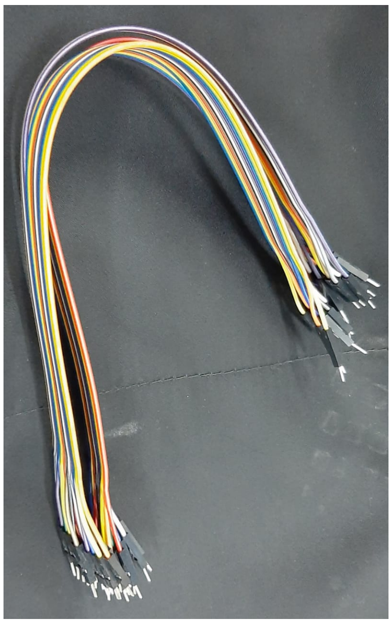


Figure 2.6: Jumper wires that are used to connect the line sensor module with the microcontroller and other parts of the circuit

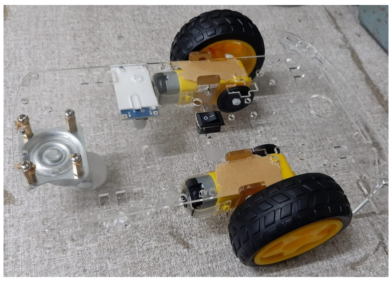


Figure 2.7: Plastic chassis of the robot

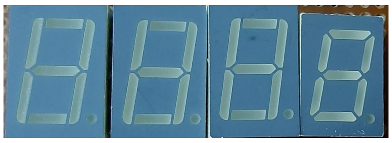


Figure 2.8: Seven Segments that will be used to show the distance covered

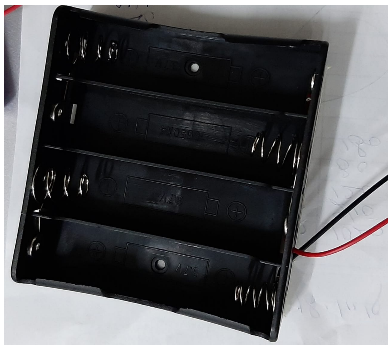


Figure 2.9: Battery holders for the lithium batteries

Figure 2.10: 18650 rechargeable lithium-ion batteries

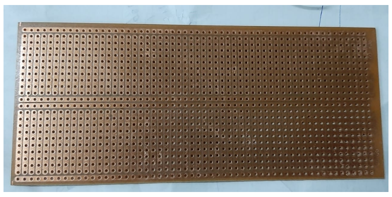


Figure 2.11: Copper veroboard (to solder on integral components of the circuits described earlier)

Chapter 3

Budget

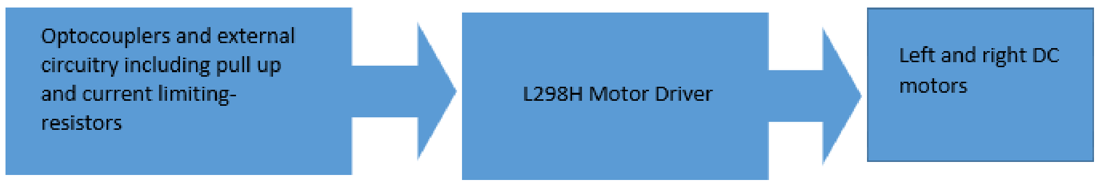
pair of 4N35 Optocouplers	Total Cost: Rs.40
LM2596 DC-DC Buck converter	Total Cost: Rs.95
2 wheel smart robot car chassis kit	Total Cost:Rs.560
Jumper wires	Total Cost: Rs.155
L298H bridge motor driver module	Total Cost:Rs.200
An LM7805 voltage regulator	Total Cost: Rs.12
Copper veroboard (breadboard style)	Total Cost: Rs.30
5 channel TCRT5000 , line tracking module	Total Cost: Rs.850
6 18650 Lithium-ion batteries	Total Cost: Rs.1500
4 battery holder for batteries	Total Cost: Rs.150
2 battery holder for batteries	Total Cost: Rs.90
2 IC Socket Base Bed Jack (Pin 2.54mm)	Total Cost: Rs.6
4 small seven segments	Total Cost: Rs.80
total budget:	Total Cost: Rs.3778

Table 3.1: Budget

Chapter 4

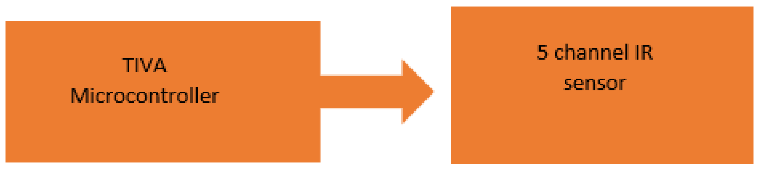
Proposed Approach

4.1 Plant

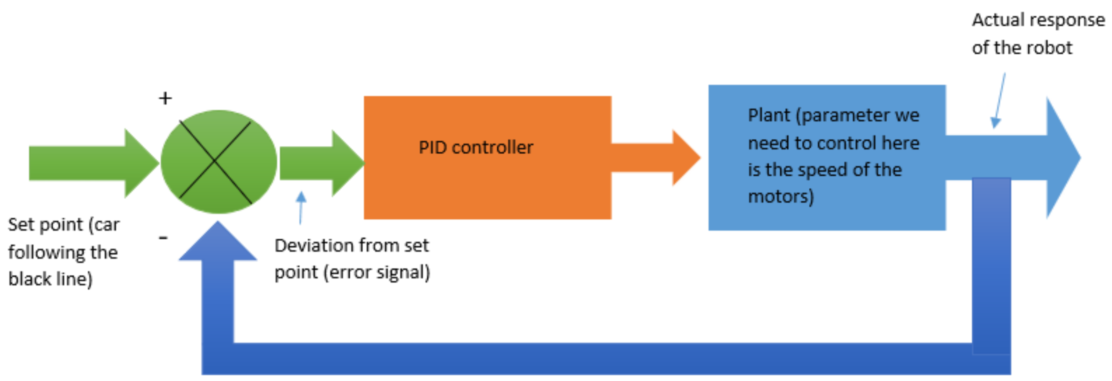


Optocouplers and external circuitry including pull up and current limiting-resistors
L298H Motor Driver
Left and right DC motors

4.2 PID

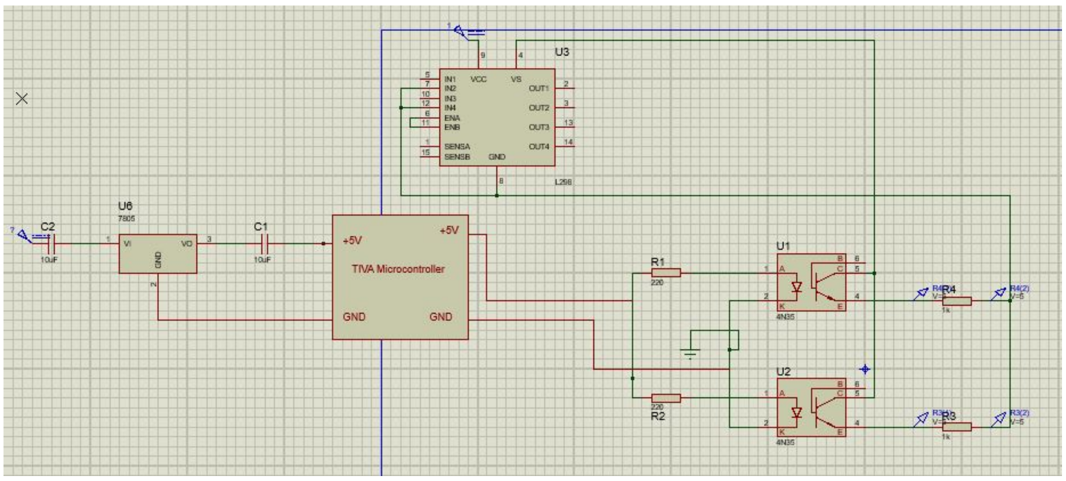


4.3 Block Diagram



Actual response of the robot
+
-
PID controller
Plant (parameter we need to control here is the speed of the motors)
Set point (car following the black line)
Deviation from set point (error signal)

4.4 Circuit Diagram



Chapter 5

Line Following using Conditional Statements

5.1 Description:

The second phase of this project basically involved reading data from the IR sensor array and eliciting the appropriate response from the 2 DC motors as a result. In other words, changing the speed of the DC motors that will cause the direction of the car to change with respect to different turns and edges on the track. To implement this algorithm, we used the IR sensor in tandem with the micro controller and we configured the timer modes of the micro controller that is essential to change the speeds of the motors (i.e the duty cycles are changed via Pulse Width Modulation (PWM)). As far as the programming is concerned, the general pseudo code to control our machine via embedded coding is listed below:

1. First, we decide what Port pins will be utilised on the TIVA board. For this, the datasheet will be consulted (discussed later)

2. Once these pins are decided, the direction, digital enable and alternate functionality registers are configured for that specific port. The timers corresponding to these pins are also configured for each motor, being in periodic mode, with PWM mode set.

Because of the PWM mode causing the timer to behave as a down counter, the duty cycles are changed by loading the **Match** register with any value less than the reload value. For example, the reload value is set to 16,000 for timer 1A. In this instance, if a value of 4000 is loaded to its match register, it will result in a 75 percent duty cycle for the speed of the motors.

5.2 Tiva Microcontroller Configuration:

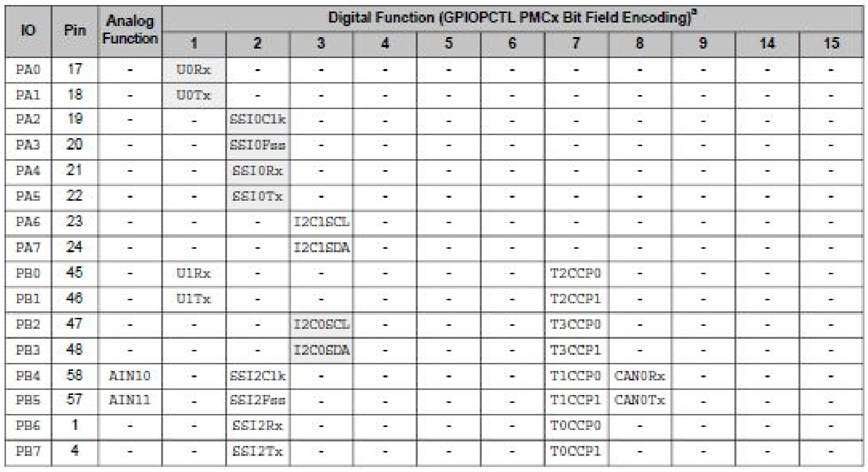
The ports used in the Tiva microcontroller for this purpose are Port F and Port B. We used :

- **Input:** PA2, PA3, PA4, PA5, PA6 (Port A Pins)
- **Output:** PB4, PB5 (Port B Pins)

Timers we Used:

- Timer 1A
- Timer 1B

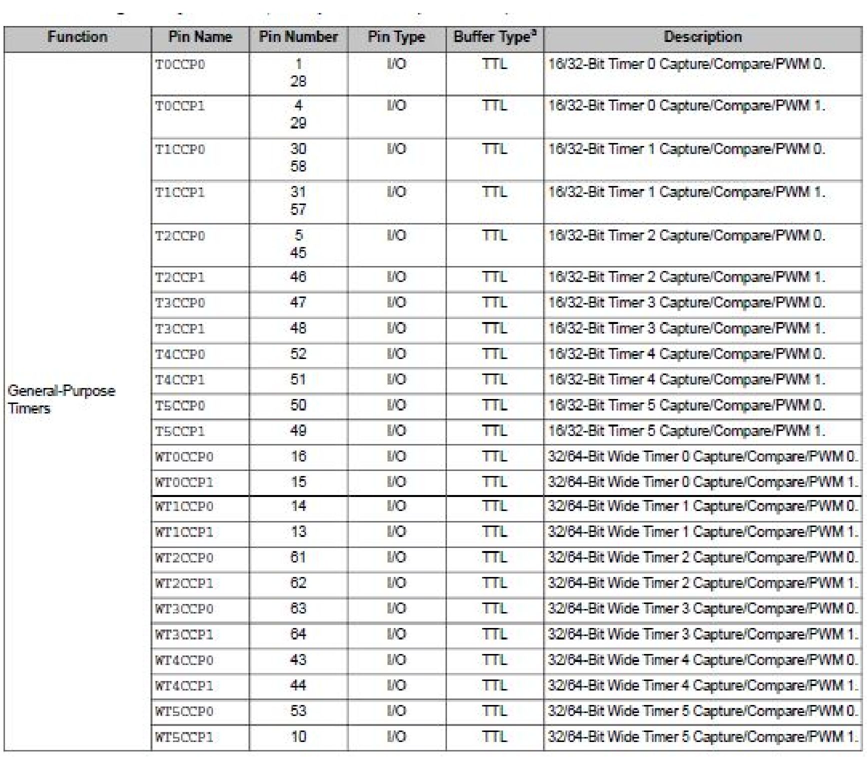
These pins with their respective timers were decided using the designated datasheet for the General Purpose Input/Output (GPIO) pins of the LM4F120H5QR Microcontroller.



IO	Pin	Analog Function	Digital Function (GPIOPCTL PMCx Bit Field Encoding)[a]										
			1	2	3	4	5	6	7	8	9	14	15
PA0	17	-	U0Rx	-	-	-	-	-	-	-	-	-	-
PA1	18	-	U0Tx	-	-	-	-	-	-	-	-	-	-
PA2	19	-	-	SSI0Clk	-	-	-	-	-	-	-	-	-
PA3	20	-	-	SSI0Fss	-	-	-	-	-	-	-	-	-
PA4	21	-	-	SSI0Rx	-	-	-	-	-	-	-	-	-
PA5	22	-	-	SSI0Tx	-	-	-	-	-	-	-	-	-
PA6	23	-	-	-	I2C1SCL	-	-	-	-	-	-	-	-
PA7	24	-	-	-	I2C1SDA	-	-	-	-	-	-	-	-
PB0	45	-	U1Rx	-	-	-	-	-	T2CCP0	-	-	-	-
PB1	46	-	U1Tx	-	-	-	-	-	T2CCP1	-	-	-	-
PB2	47	-	-	-	I2C0SCL	-	-	-	T3CCP0	-	-	-	-
PB3	48	-	-	-	I2C0SDA	-	-	-	T3CCP1	-	-	-	-
PB4	58	AIN10	-	SSI2Clk	-	-	-	-	T1CCP0	CAN0Rx	-	-	-
PB5	57	AIN11	-	SSI2Fss	-	-	-	-	T1CCP1	CAN0Tx	-	-	-
PB6	1	-	-	SSI2Rx	-	-	-	-	T0CCP0	-	-	-	-
PB7	4	-	-	SSI2Tx	-	-	-	-	T0CCP1	-	-	-	-

FIGURE 5.1: GPIO Pins and Alternate Functions

In order to easily interpret the datasheet, we took help from an additonal table, known as the *Signals by Function* table in which we came to know about what terms such as 'T0CCP0' and 'T1CCP0'. The relevant segment of that table is attached with this report.



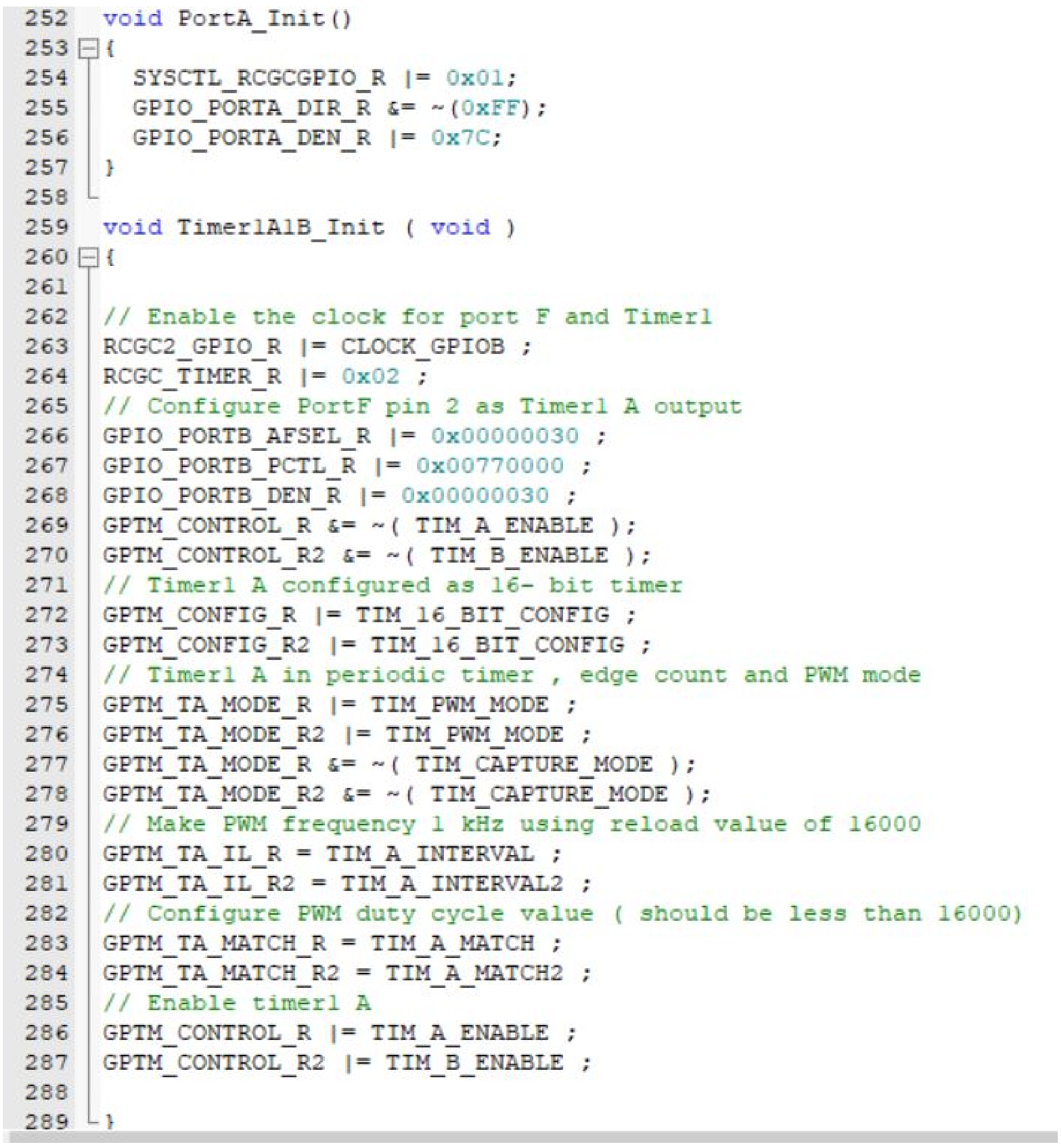
Function	Pin Name	Pin Number	Pin Type	Buffer Type[a]	Description
General-Purpose Timers	T0CCP0	1 28	I/O	TTL	16/32-Bit Timer 0 Capture/Compare/PWM 0.
	T0CCP1	4 29	I/O	TTL	16/32-Bit Timer 0 Capture/Compare/PWM 1.
	T1CCP0	30 58	I/O	TTL	16/32-Bit Timer 1 Capture/Compare/PWM 0.
	T1CCP1	31 57	I/O	TTL	16/32-Bit Timer 1 Capture/Compare/PWM 1.
	T2CCP0	5 45	I/O	TTL	16/32-Bit Timer 2 Capture/Compare/PWM 0.
	T2CCP1	46	I/O	TTL	16/32-Bit Timer 2 Capture/Compare/PWM 1.
	T3CCP0	47	I/O	TTL	16/32-Bit Timer 3 Capture/Compare/PWM 0.
	T3CCP1	48	I/O	TTL	16/32-Bit Timer 3 Capture/Compare/PWM 1.
	T4CCP0	52	I/O	TTL	16/32-Bit Timer 4 Capture/Compare/PWM 0.
	T4CCP1	51	I/O	TTL	16/32-Bit Timer 4 Capture/Compare/PWM 1.
	T5CCP0	50	I/O	TTL	16/32-Bit Timer 5 Capture/Compare/PWM 0.
	T5CCP1	49	I/O	TTL	16/32-Bit Timer 5 Capture/Compare/PWM 1.
	WT0CCP0	16	I/O	TTL	32/64-Bit Wide Timer 0 Capture/Compare/PWM 0.
	WT0CCP1	15	I/O	TTL	32/64-Bit Wide Timer 0 Capture/Compare/PWM 1.
	WT1CCP0	14	I/O	TTL	32/64-Bit Wide Timer 1 Capture/Compare/PWM 0.
	WT1CCP1	13	I/O	TTL	32/64-Bit Wide Timer 1 Capture/Compare/PWM 1.
	WT2CCP0	61	I/O	TTL	32/64-Bit Wide Timer 2 Capture/Compare/PWM 0.
	WT2CCP1	62	I/O	TTL	32/64-Bit Wide Timer 2 Capture/Compare/PWM 1.
	WT3CCP0	63	I/O	TTL	32/64-Bit Wide Timer 3 Capture/Compare/PWM 0.
	WT3CCP1	64	I/O	TTL	32/64-Bit Wide Timer 3 Capture/Compare/PWM 1.
	WT4CCP0	43	I/O	TTL	32/64-Bit Wide Timer 4 Capture/Compare/PWM 0.
	WT4CCP1	44	I/O	TTL	32/64-Bit Wide Timer 4 Capture/Compare/PWM 1.
	WT5CCP0	53	I/O	TTL	32/64-Bit Wide Timer 5 Capture/Compare/PWM 0.
	WT5CCP1	10	I/O	TTL	32/64-Bit Wide Timer 5 Capture/Compare/PWM 1.

FIGURE 5.2: Signals by Function, apart from GPIO

5.3 Code:

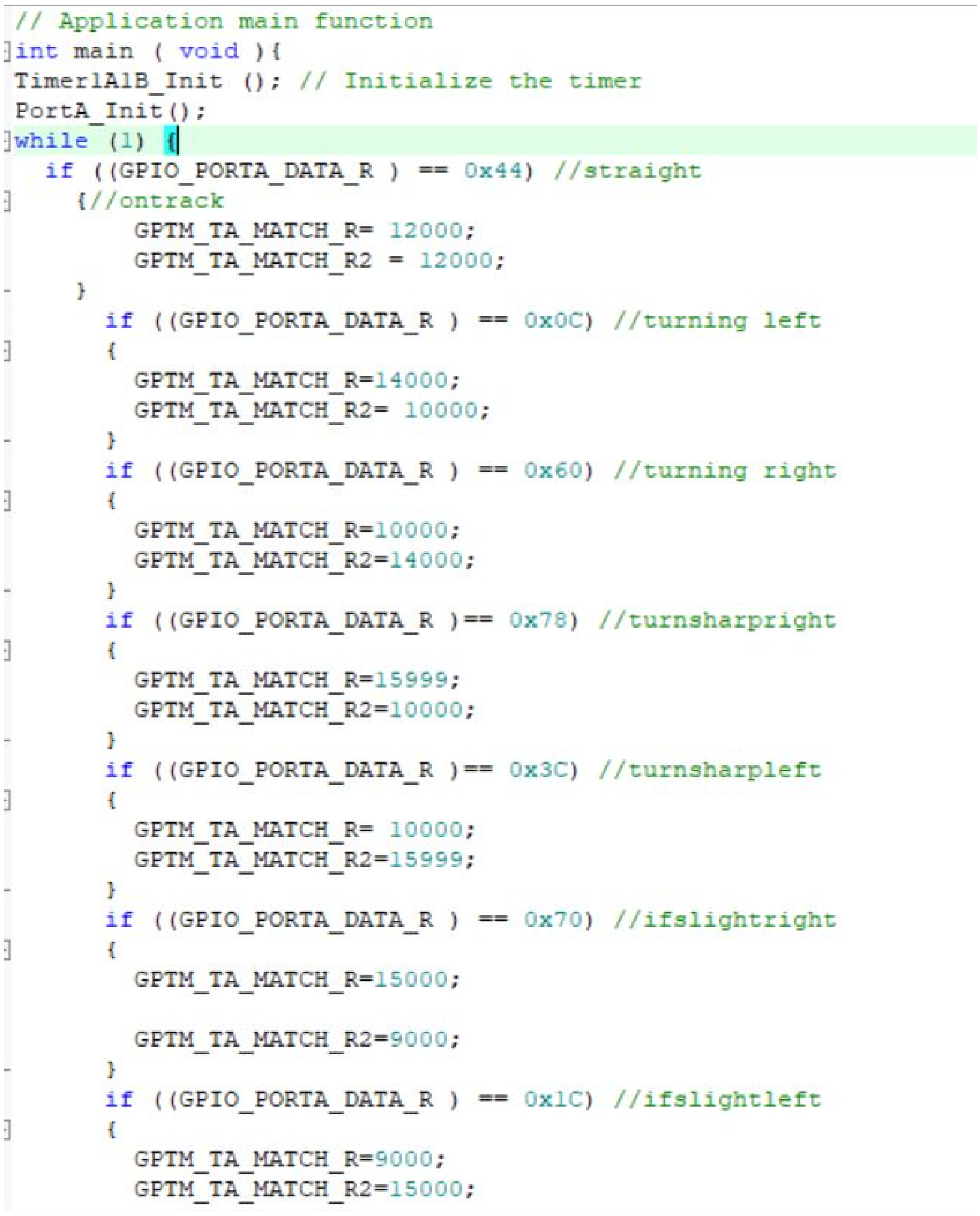
The C-program for this project is described in this section.

The relevant registers for the timers, as well as the ports had their addresses initialised at the start of the code as shown below. The ports, also, had their respective timers initialised as well, alongside with interrupt-level configuration for the ports A and B.



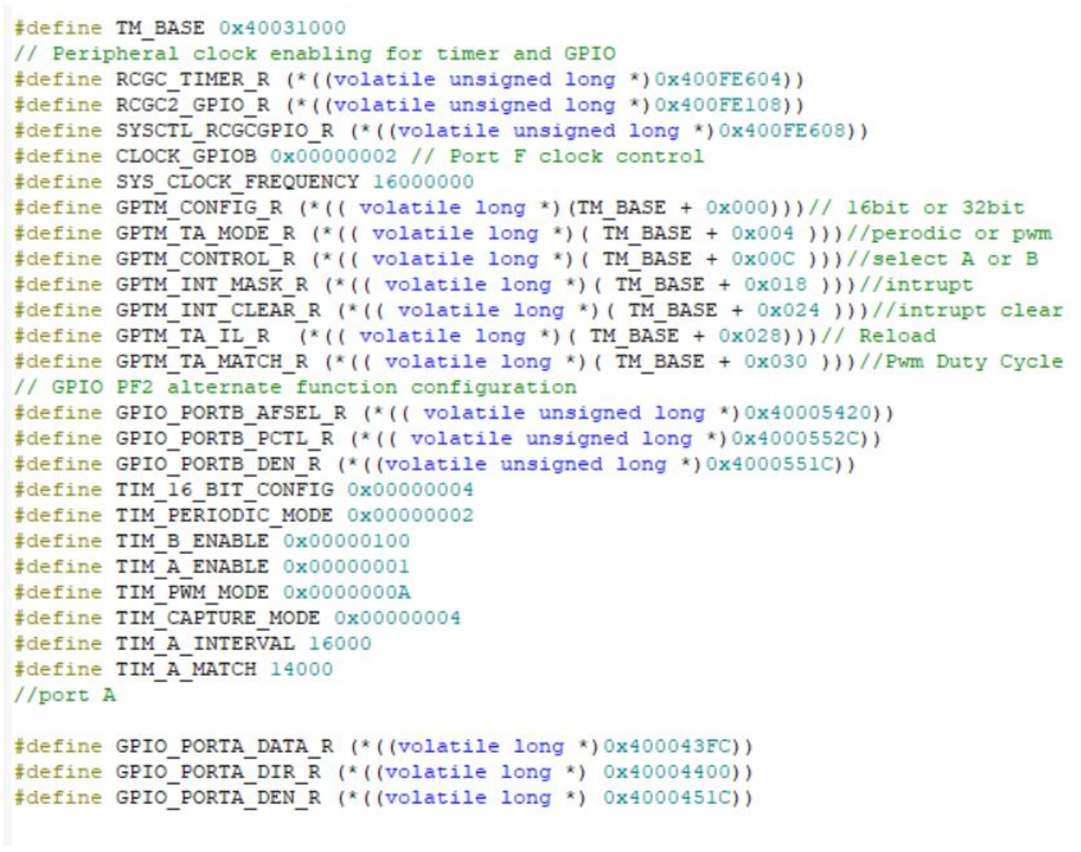
```
void PortA_Init()
{
  SYSCTL_RCGCGPIO_R |= 0x01;
  GPIO_PORTA_DIR_R &= ~(0xFF);
  GPIO_PORTA_DEN_R |= 0x7C;
}

void Timer1A1B_Init ( void )
{

// Enable the clock for port F and Timer1
RCGC2_GPIO_R |= CLOCK_GPIOB ;
RCGC_TIMER_R |= 0x02 ;
// Configure PortF pin 2 as Timer1 A output
GPIO_PORTB_AFSEL_R |= 0x00000030 ;
GPIO_PORTB_PCTL_R |= 0x00770000 ;
GPIO_PORTB_DEN_R |= 0x00000030 ;
GPTM_CONTROL_R &= ~( TIM_A_ENABLE );
GPTM_CONTROL_R2 &= ~( TIM_B_ENABLE );
// Timer1 A configured as 16- bit timer
GPTM_CONFIG_R |= TIM_16_BIT_CONFIG ;
GPTM_CONFIG_R2 |= TIM_16_BIT_CONFIG ;
// Timer1 A in periodic timer , edge count and PWM mode
GPTM_TA_MODE_R |= TIM_PWM_MODE ;
GPTM_TA_MODE_R2 |= TIM_PWM_MODE ;
GPTM_TA_MODE_R &= ~( TIM_CAPTURE_MODE );
GPTM_TA_MODE_R2 &= ~( TIM_CAPTURE_MODE );
// Make PWM frequency 1 kHz using reload value of 16000
GPTM_TA_IL_R = TIM_A_INTERVAL ;
GPTM_TA_IL_R2 = TIM_A_INTERVAL2 ;
// Configure PWM duty cycle value ( should be less than 16000)
GPTM_TA_MATCH_R = TIM_A_MATCH ;
GPTM_TA_MATCH_R2 = TIM_A_MATCH2 ;
// Enable timer1 A
GPTM_CONTROL_R |= TIM_A_ENABLE ;
GPTM_CONTROL_R2 |= TIM_B_ENABLE ;

}
```

FIGURE 5.3: Function that configures timers 1A and 1B (for Port B) and Port A



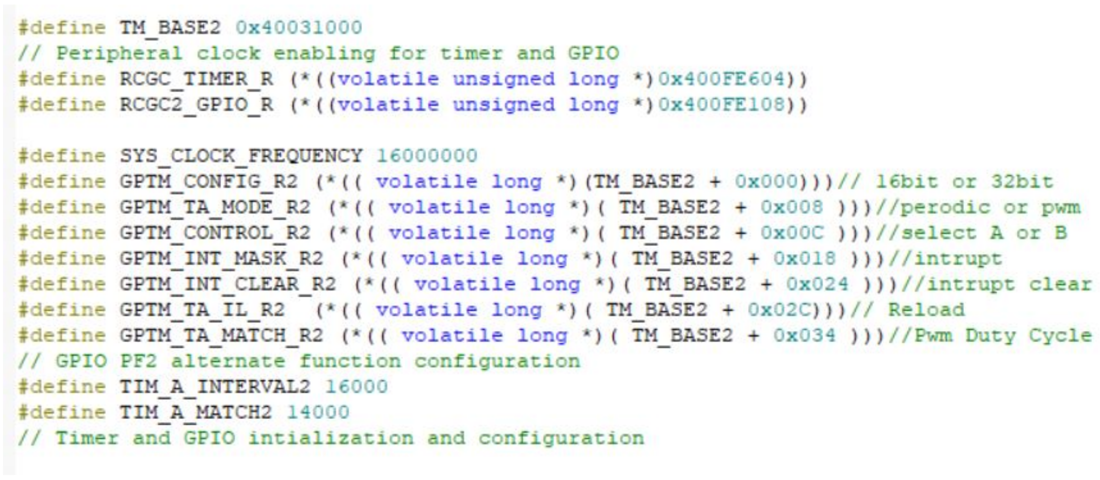
```
// Application main function
int main ( void ){
Timer1A1B_Init (); // Initialize the timer
PortA_Init();
while (1) {
  if ((GPIO_PORTA_DATA_R ) == 0x44) //straight
    {//ontrack
        GPTM_TA_MATCH_R= 12000;
        GPTM_TA_MATCH_R2 = 12000;
    }
      if ((GPIO_PORTA_DATA_R ) == 0x0C) //turning left
      {
        GPTM_TA_MATCH_R=14000;
        GPTM_TA_MATCH_R2= 10000;
      }
      if ((GPIO_PORTA_DATA_R ) == 0x60) //turning right
      {
        GPTM_TA_MATCH_R=10000;
        GPTM_TA_MATCH_R2=14000;
      }
      if ((GPIO_PORTA_DATA_R )== 0x78) //turnsharpright
      {
        GPTM_TA_MATCH_R=15999;
        GPTM_TA_MATCH_R2=10000;
      }
      if ((GPIO_PORTA_DATA_R )== 0x3C) //turnsharpleft
      {
        GPTM_TA_MATCH_R= 10000;
        GPTM_TA_MATCH_R2=15999;
      }
      if ((GPIO_PORTA_DATA_R ) == 0x70) //ifslightright
      {
        GPTM_TA_MATCH_R=15000;

        GPTM_TA_MATCH_R2=9000;
      }
      if ((GPIO_PORTA_DATA_R ) == 0x1C) //ifslightleft
      {
        GPTM_TA_MATCH_R=9000;
        GPTM_TA_MATCH_R2=15000;
```

FIGURE 5.4: Main function calling 'Timer1A1BInit' and 'PortAInit'



```
#define TM_BASE 0x40031000
// Peripheral clock enabling for timer and GPIO
#define RCGC_TIMER_R (*((volatile unsigned long *)0x400FE604))
#define RCGC2_GPIO_R (*((volatile unsigned long *)0x400FE108))
#define SYSCTL_RCGCGPIO_R (*((volatile unsigned long *)0x400FE608))
#define CLOCK_GPIOB 0x00000002 // Port F clock control
#define SYS_CLOCK_FREQUENCY 16000000
#define GPTM_CONFIG_R (*(( volatile long *)(TM_BASE + 0x000)))// 16bit or 32bit
#define GPTM_TA_MODE_R (*(( volatile long *)( TM_BASE + 0x004 )))//perodic or pwm
#define GPTM_CONTROL_R (*(( volatile long *)( TM_BASE + 0x00C )))//select A or B
#define GPTM_INT_MASK_R (*(( volatile long *)( TM_BASE + 0x018 )))//intrupt
#define GPTM_INT_CLEAR_R (*(( volatile long *)( TM_BASE + 0x024 )))//intrupt clear
#define GPTM_TA_IL_R  (*(( volatile long *)( TM_BASE + 0x028)))// Reload
#define GPTM_TA_MATCH_R (*(( volatile long *)( TM_BASE + 0x030 )))//Pwm Duty Cycle
// GPIO PF2 alternate function configuration
#define GPIO_PORTB_AFSEL_R (*(( volatile unsigned long *)0x40005420))
#define GPIO_PORTB_PCTL_R (*(( volatile unsigned long *)0x4000552C))
#define GPIO_PORTB_DEN_R (*((volatile unsigned long *)0x4000551C))
#define TIM_16_BIT_CONFIG 0x00000004
#define TIM_PERIODIC_MODE 0x00000002
#define TIM_B_ENABLE 0x00000100
#define TIM_A_ENABLE 0x00000001
#define TIM_PWM_MODE 0x0000000A
#define TIM_CAPTURE_MODE 0x00000004
#define TIM_A_INTERVAL 16000
#define TIM_A_MATCH 14000
//port A

#define GPIO_PORTA_DATA_R (*((volatile long *)0x400043FC))
#define GPIO_PORTA_DIR_R (*((volatile long *) 0x40004400))
#define GPIO_PORTA_DEN_R (*((volatile long *) 0x4000451C))
```

FIGURE 5.5: Relevant initializations of the ports with alternate functionalities



```
#define TM_BASE2 0x40031000
// Peripheral clock enabling for timer and GPIO
#define RCGC_TIMER_R (*((volatile unsigned long *)0x400FE604))
#define RCGC2_GPIO_R (*((volatile unsigned long *)0x400FE108))

#define SYS_CLOCK_FREQUENCY 16000000
#define GPTM_CONFIG_R2 (*(( volatile long *)(TM_BASE2 + 0x000)))// 16bit or 32bit
#define GPTM_TA_MODE_R2 (*(( volatile long *)( TM_BASE2 + 0x008 )))//perodic or pwm
#define GPTM_CONTROL_R2 (*(( volatile long *)( TM_BASE2 + 0x00C )))//select A or B
#define GPTM_INT_MASK_R2 (*(( volatile long *)( TM_BASE2 + 0x018 )))//intrupt
#define GPTM_INT_CLEAR_R2 (*(( volatile long *)( TM_BASE2 + 0x024 )))//intrupt clear
#define GPTM_TA_IL_R2  (*(( volatile long *)( TM_BASE2 + 0x02C)))// Reload
#define GPTM_TA_MATCH_R2 (*(( volatile long *)( TM_BASE2 + 0x034 )))//Pwm Duty Cycle
// GPIO PF2 alternate function configuration
#define TIM_A_INTERVAL2 16000
#define TIM_A_MATCH2 14000
// Timer and GPIO intialization and configuration
```

FIGURE 5.6: Second segment of the code

Chapter 6

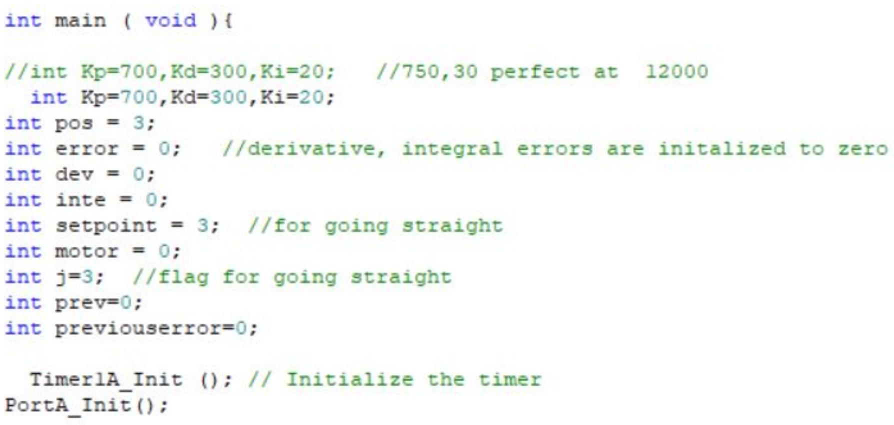
Line following using the PID Controller

The third and final phase of this project included tuning the PID parameters of the car so that it followed the black line on the track with minimal oscillations at a fairly quick speed. The code implemented in the second phase had an obvious drawback, which was that the machine exhibited many oscillations whilst on the track, not resulting in an smooth response. To eliminate this problem, a PID controller was implemented via our knowledge of embedded C programming, albeit a PID controller can be made using entirely hardware components (for example, a proportional controller can be applied with an op-amp with open loop gain). To start off, we take values of 1, 0 and zero for the three PID constants, Kp, Kd and Ki and proceed from there.

The proportional (Kp), derivative (Kd) and integral (Ki) gain are usually functioning in tandem to eliminate the shortcomings of one another. In other words, the proportional controller, working independently,decreases the steady state error but increases the overshoot. The integral controller, together with the proportional controller, eliminates the steady state error completely but increases the overshoot from the set point even more, alongside the settling time of the system (practically causing the system output to oscillate). Therefore, a derivative controller is introduced, decreasing the overshoot and settling time of the system, proving that these three parameters when set are effective in the process than a standalone proportional controller. Note that the rise time in all of them decrease so there is no negative effect of any of the controllers in that respect.

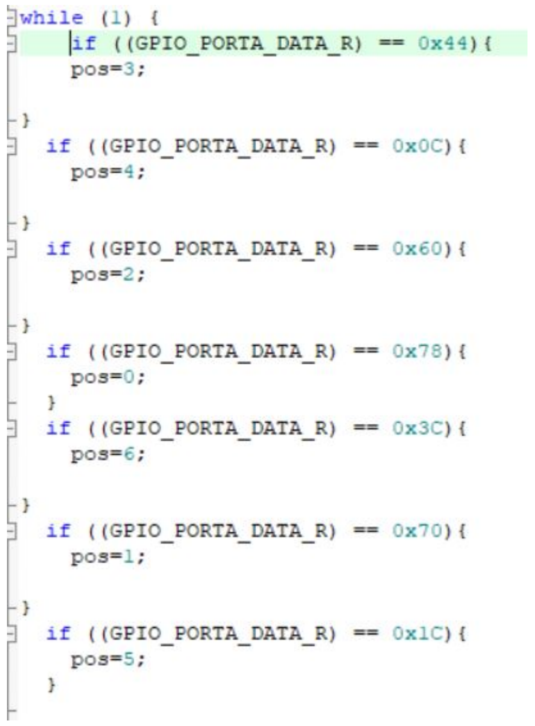
From a coding point of view, the Kp, Kd, and Ki constants are tuned to suit our need i.e the values of them are changed in such a way that will give us the best performance. An important thing to note here that the initialization and configurations of the timers and port will be unchanged from the code of the second phase.

The function which is essential for our robot to run via a PID controller is attached below. For minimal oscillations in the system, the values of the PID parameters turned out to be 700 for Kp, 300 for Kd and 20 for Ki.



```
int main ( void ){

//int Kp=700,Kd=300,Ki=20;   //750,30 perfect at  12000
  int Kp=700,Kd=300,Ki=20;
int pos = 3;
int error = 0;   //derivative, integral errors are initalized to zero
int dev = 0;
int inte = 0;
int setpoint = 3;  //for going straight
int motor = 0;
int j=3;  //flag for going straight
int prev=0;
int previouserror=0;

  Timer1A_Init (); // Initialize the timer
PortA_Init();
```

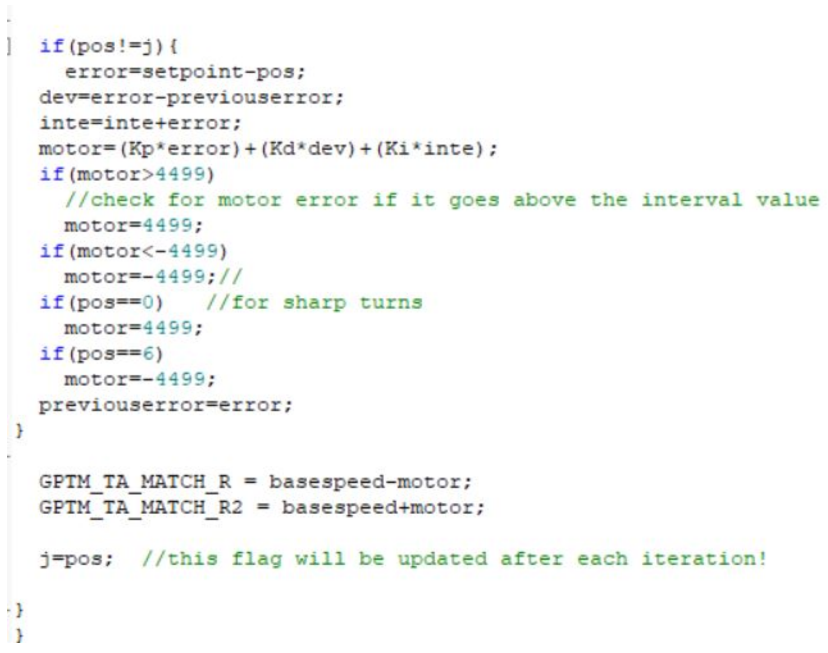
Figure 6.1: Relevant initializations of PID parameters



```
while (1) {
    if ((GPIO_PORTA_DATA_R) == 0x44){
    pos=3;

}
  if ((GPIO_PORTA_DATA_R) == 0x0C){
    pos=4;

}
  if ((GPIO_PORTA_DATA_R) == 0x60){
    pos=2;

}
  if ((GPIO_PORTA_DATA_R) == 0x78){
    pos=0;
  }
  if ((GPIO_PORTA_DATA_R) == 0x3C){
    pos=6;

}
  if ((GPIO_PORTA_DATA_R) == 0x70){
    pos=1;

}
  if ((GPIO_PORTA_DATA_R) == 0x1C){
    pos=5;
  }
```

Figure 6.2: If-else statements for positions

In the above code segment, we have applied checks to the input of port A which is attached to the IR sensor array. When we attain various values of the *GPIOPORTA-DATA* register, we assigned different position variables which would be used later on in calculating the error in the speed at tricky parts of the track. This would ultimately enable the car to course-correct itself.



```
  if(pos!=j){
    error=setpoint-pos;
  dev=error-previouserror;
  inte=inte+error;
  motor=(Kp*error)+(Kd*dev)+(Ki*inte);
  if(motor>4499)
    //check for motor error if it goes above the interval value
    motor=4499;
  if(motor<-4499)
    motor=-4499;//
  if(pos==0)   //for sharp turns
    motor=4499;
  if(pos==6)
    motor=-4499;
  previouserror=error;
}

  GPTM_TA_MATCH_R = basespeed-motor;
  GPTM_TA_MATCH_R2 = basespeed+motor;

  j=pos;  //this flag will be updated after each iteration!

}
}
```

Figure 6.3: Lines that calculate the error in speed

The lines above in the figure, calculate the accumulated error for each iteration (position on the track) and that error is added and subtracted into each match value that is to be given to the timer match registers, corresponding to the two tyres.

https://www.uet.edu.pk/pp/ee/~mtahir/teaching.html