HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Embedded Project

Hochschule Ravensburg - Weingarten

High Precision Motor Control

Submitted to:	Prof. Dr.-Ing Franz Brümmer Hochschule Ravensburg-Weingarten
Submitted by:	Prithvi Patel (33786) Anubhav Aggarwal (34309) Oseihie Eigbobo (33658)
Submitted on:	01.08.2021

Abstract

The Objective of this project is to control the DC motor with High Precision. This project have been done for the Line following robot. For the motion of the robot, the DC motor has been used. The goal basically is to precisely control the speed of the motor. Thus, this project gives a brief overview of the precise control using control feedback algorithm. For this task, ESP32 is used as the hardware controller and esp-idf framework is used to flash the program into the ESP32. This report contains the information about the approach which has been taken to achieve the goal of this project.

Contents

List of Figures

1 Introduction

In this project, ESP32 is used as a controller to control the line following robot. ESP32 is a controller which can perform as a complete standalone system, which is primarily use to reduce the communication stack overhead on the main application processor. It has a wide range of application. ESP32 is capable of functioning reliably in industrial environments, with an operating temperature ranging from -40°C to 125°C. ESP32 can remove external circuit imperfections and adapt the chnge in external conditions which makes it ideal to use it for motor control applications. [1]

It is pretty common to use Arduino IDE to program the ESP32. But the compilation of the Arduino Libraries takes a huge amount of memory and later, the main application doesn't have enough memory to execute. Thus, ESP-IDF has been used in order to work with project. ESP-IDF is Espressif's Official IoT Development framework for the ESP32, ESP32-S and ESP32-C series of SoCs. It provies a self-sufficient SDK for any generic application development on these platforms, using programming languages such as C and C++. ESP-IDF is freely available on GitHub and majority of the components in the ESP-IDF are available in source code form as examples. Also, ESP-IDF comes with an extensive documentation for its software components not only at the usage level but also at the design level. It ultimately takes up far too less memory compared to arduino libraries. [1]

At first, the wiring connections needed to be done with the schematics provided. These schematics can been seen in the images below.

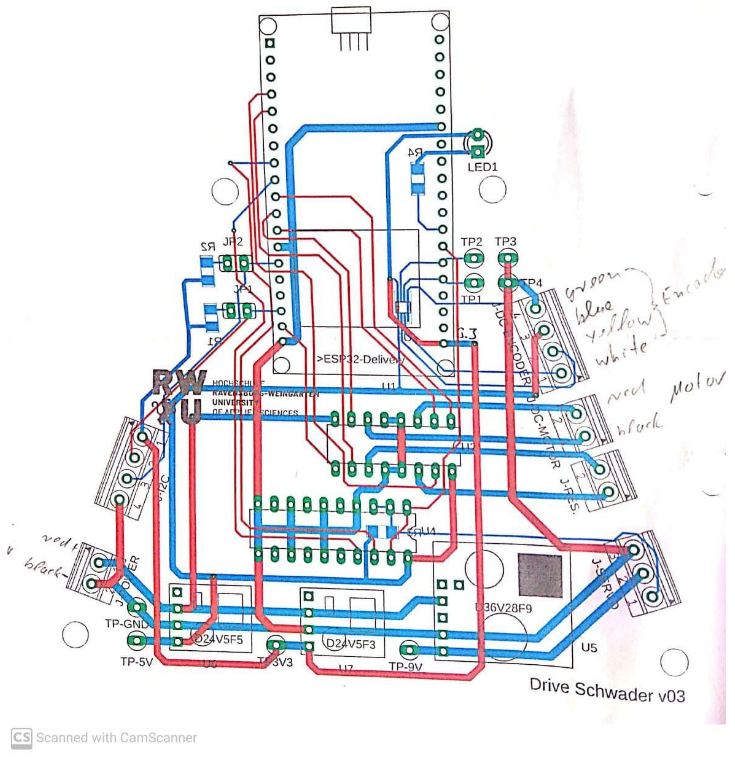


Figure 1: Wiring Connections for the line following robot

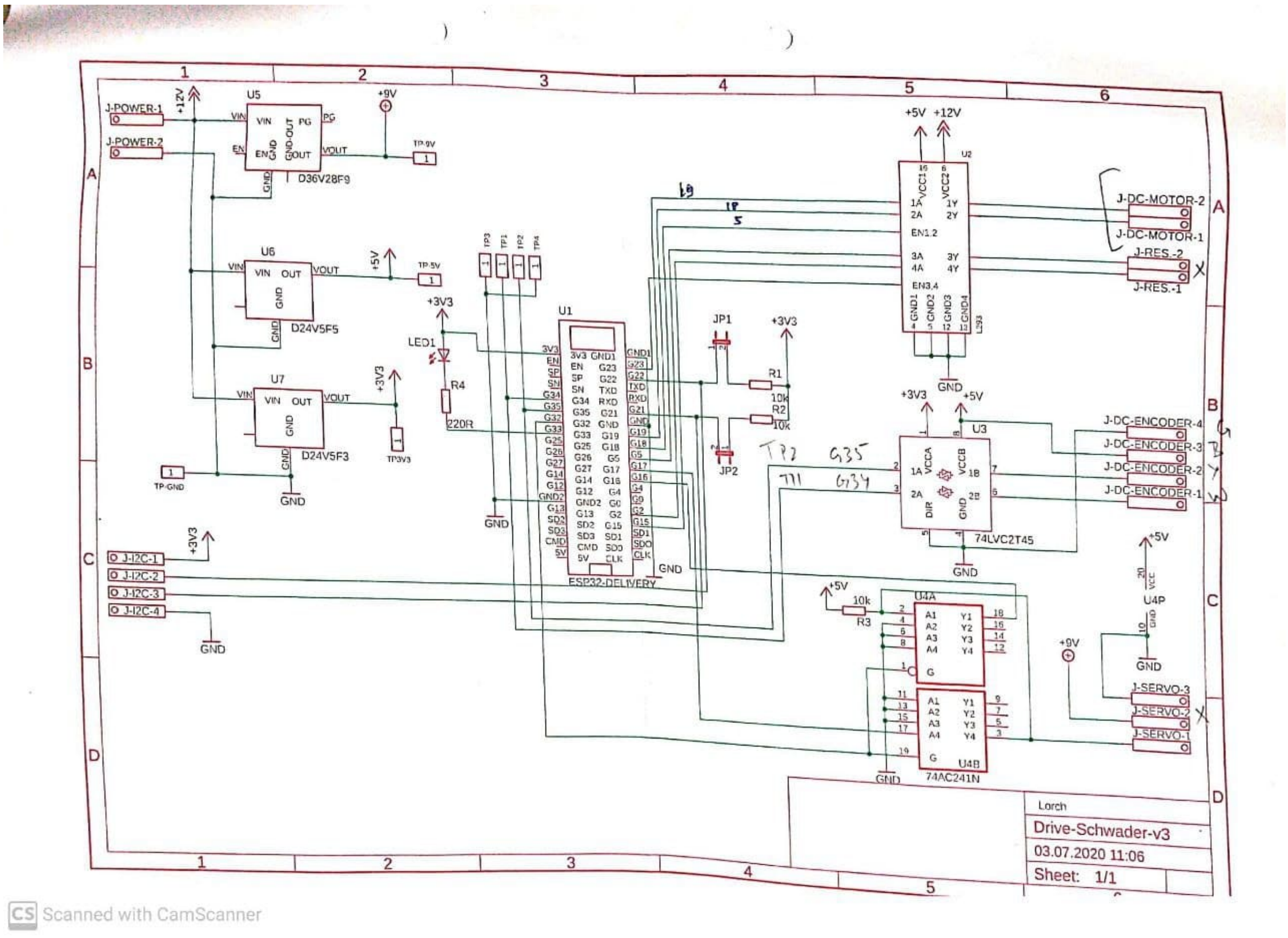


Figure 2: Schematic diagram for the line following robot

The DC motor was provided by the faculty. Through this schematics, it can be found out the connections to the DC motor, the encoders and the blinking LED. The wires were connected accordingly. Also, in order to power up the DC motor, a 12V external power supply is required.

NOTE : This report only covers the description of the final code which can be used for the further development and deployment and doesn't contain smaller sub-steps which has been taken to arrive at the final version of the program.

At first, the program sets the initial duty cycle for the DC motor. Once the initial duty cycle is set, the motor function is executed with initial duty cycle. In figure 3, the motor encoder pin diagram which will be helpful in the explanation for the upcoming chapters.Usually, Encoders are used to get the feedback from the motor. As it can be seen in figure 3, the feedback signals from the encoder which actually gives the feedback information are Quad encoder signal A (Yellow) and signal B (White).

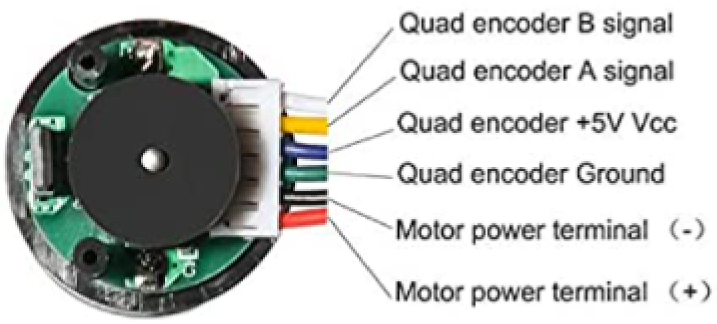


Figure 3: Wiring diagram of the Encoder [2]

These encoder signals A and B are connected to the GPIO 35 and 34 of the ESP32 respectively. Thus, the feedback data is taken back into the controller. After that, the PID feedback control loop is applied which can be seen in figure 4. This is covered in brief in the upcoming chapters.

2 Deeper Insight to the Project

This chapter contains a bit deeper insight of the ESP32. It covers the description of the units and modules which had been used in this project. There were two major units which have been used namely, MCPWM and PCNT units.

2.1 MCPWM unit

MCPWM is an acronym for Motor Control - Pulse Width Modulation unit. As the name suggests, this unit is specifically used for PWM duty cycle control for the motor. ESP32 have two units of MCPWM in it. And each unit has three pairs of PWM outputs. [5]

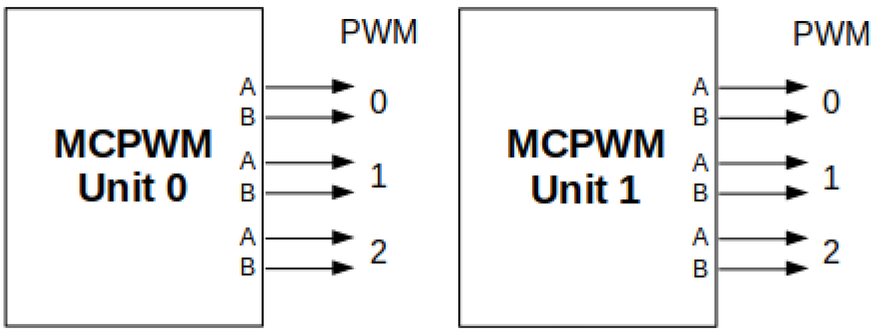


Figure 4: MCPWM units overview [5]

Also there are three timers: 0, 1 and 2. The PWM pairs shown in figure 4 can be clocked with one of the three times. For this project, timer 0 has been used. However, the same timer can also be used for more than one pairs of PWM signals. The configuration of these units are dependent on the type of motor being used, especially the number of inputs and outputs which are required and their sequence to drive the motor. Here is an example from the official documentation of Espressif to the control the brushed DC motor. The PWM pair of signals of the MCPWM unit is connected to H-Bridge to switch the polarization of the voltage applied to the motor and provides sufficient amount of current to drive the motor. [5]

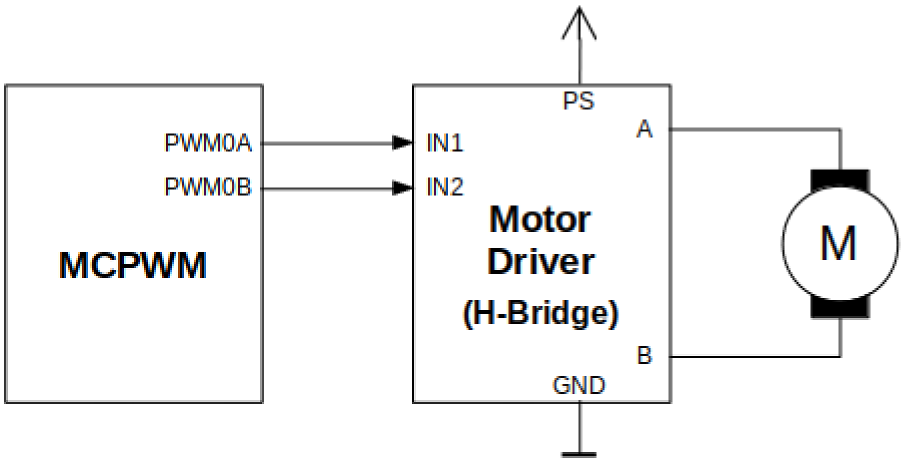


Figure 5: Brushed DC motor configuration example [5]

In order to configure the MCPWM unit, selection of the MCPWM unit that is used to drive the motor is required. It is done with mcpwm_unit_t. After that, the two GPIOs as PWM output needs to be selected and it is done with mcpwm_gpio_init(). These two output signals are using to give instructions to the motor to rotate clockwise or anti-clockwise. Selection of the timer to sequential Here is the configuration that has been used for the project. [5]

```
// Initializing the GPIOs
#define GPIO_PWM0A_OUT 19    //Set GPIO 15 as PWM0A
#define GPIO_PWM0B_OUT 18    //Set GPIO 16 as PWM0B

static void mcpwm_example_gpio_initialize(void){
  printf("initialize mcpwm gpio...\n");
  mcpwm_gpio_init(MCPWM_UNIT_0, MCPWM0A, GPIO_PWM0A_OUT);
  mcpwm_gpio_init(MCPWM_UNIT_0, MCPWM0B, GPIO_PWM0B_OUT);
}
```

And with the initial duty cycle, separate functions were made to drive the motor in forward, backward or none(stop) direction. mcpwm_set_signal_low() or mcpwm_set_signal_high() is used to drive the particular signal in low or high respectively. This can also make the motor to rotate with maximum speed or to stop it. mcpwm_set_duty() is used to vary the PWM's duty cycle in percentages. with mcpwm_set_duty_type(), one can alter the phase of the PWM signals. Here are the functions that has been used to drive the motor. [5]

```
/*
  Motor moves in forward direction , with duty cycle = duty %
*/
static void brushed_motor_forward(
mcpwm_unit_t mcpwm_num,
mcpwm_timer_t timer_num , float duty_cycle){

  /*printf("Turing motor forward... \n");*/
  mcpwm_set_signal_low(mcpwm_num,
  timer_num, MCPWM_OPR_B);

  mcpwm_set_duty(mcpwm_num, timer_num,
  MCPWM_OPR_A, duty_cycle);

  mcpwm_set_duty_type(mcpwm_num, timer_num,
  MCPWM_OPR_A, MCPWM_DUTY_MODE_0); // call this each time, if
  operator was previously in low/high state
}

/*
  Motor moves in backward direction , with duty cycle = duty %
*/
static void brushed_motor_backward(
mcpwm_unit_t mcpwm_num,
mcpwm_timer_t timer_num ,
 float duty_cycle){

  /*printf("Turing motor backward... \n");*/
  mcpwm_set_signal_low(mcpwm_num, timer_num, MCPWM_OPR_A);
  mcpwm_set_duty(mcpwm_num, timer_num,
  MCPWM_OPR_B, duty_cycle);

  mcpwm_set_duty_type(mcpwm_num, timer_num,
  MCPWM_OPR_B, MCPWM_DUTY_MODE_0); // call this each time,
  if operator was previously in low/high state
}

static void brushed_motor_stop(
mcpwm_unit_t mcpwm_num,
 mcpwm_timer_t timer_num){
  //printf("Motor Stopped \n");
  mcpwm_set_signal_low(mcpwm_num, timer_num, MCPWM_OPR_A);
  mcpwm_set_signal_low(mcpwm_num, timer_num, MCPWM_OPR_B);
}
```

After that, the PWM configuration is initialzied and created a task which with basically driving the motors.

```
static void mcpwm_execute(void *arg){
mcpwm_config_t pwm_config;
pwm_config.frequency = 1000;  // frequency = 1000 Hz
pwm_config.cmpr_a = 0;  // duty cycle for PWMxA
pwm_config.cmpr_b = 0;  // duty cycle for PWMxB
pwm_config.counter_mode = MCPWM_UP_DOWN_COUNTER;
pwm_config.duty_mode = MCPWM_DUTY_MODE_0;
mcpwm_init(MCPWM_UNIT_0, MCPWM_TIMER_0, &pwm_config);

while(1){
        brushed_motor_forward(MCPWM_UNIT_0,
        MCPWM_TIMER_0, abs(init_duty));
        vTaskDelay(1000/portTICK_RATE_MS);

        brushed_motor_backward(MCPWM_UNIT_0,
        MCPWM_TIMER_0, abs(init_duty));
        vTaskDelay(1000/portTICK_RATE_MS);

        brushed_motor_stop(MCPWM_UNIT_0,
        MCPWM_TIMER_0);
        vTaskDelay(1000/portTICK_RATE_MS);

}
}

void app_main(void)
{
  printf("Testing brushed motor... \n");
  xTaskCreate(mcpwm_execute,
  "mcpwm_execute", 4096, NULL, 5, NULL);
}
```

This is how the user-defined functions are used to drive the motor. But still, the motor is limited in control. When the motor is driven with 100 percent duty cycle, it is not really known whether the motor is over clocked or under clocked. Thus, it is required to take the feedback from the encoder. This is done through Pulse Counter Unit (PCNT).

2.2 PCNT unit

The PCNT (Pulse Counter) module has been built to count the number of rising or falling edges or both. Each PCNT unit has a 16-bit signed counter register and two channels that can be configured to either increment or decrement the counter. The signal edges can be detected through each of the channel's input signal. Also the control input can be detected which can be used to enable or disable the input signal. The input ports have filters which are optional and can be used to remove unwanted glitches in the input signal. In order to configure the PCNT, firstly one needs to know that the PCNT unit has 8 independent counting units, between 0 and 7. Each of the independent channels are numbered as 0 and 1.

```
#define PCNT_H_LIM_VAL 20000
#define PCNT_L_LIM_VAL -20000
#define PCNT_THRESH1_VAL 5
#define PCNT_THRESH0_VAL -5
#define PCNT_INPUT_SIG_IO     34
#define PCNT_INPUT_CTRL_IO    35
#define LEDC_OUTPUT_IO        33

 pcnt_config_t pcnt_config = {
     .pulse_gpio_num = PCNT_INPUT_SIG_IO,
     .ctrl_gpio_num = PCNT_INPUT_CTRL_IO,
     .channel = PCNT_CHANNEL_0,
     .unit = unit,
     .pos_mode = PCNT_COUNT_DIS,
     .neg_mode = PCNT_COUNT_INC,
     .lctrl_mode = PCNT_MODE_REVERSE,
     .hctrl_mode = PCNT_MODE_KEEP,
     .counter_h_lim = PCNT_H_LIM_VAL,
     .counter_l_lim = PCNT_L_LIM_VAL,
   };
```

The code above shows the configuration of the unit and channel. GPIO signals for pulse input (signal) and pulse gate input (control) are 34 and 35 respectively. PCNT_COUNT_INC and PCNT_COUNT_DIS are set to their responsible ports to describe how the counter should reach depending on the status of the control signal and how the counting is supposed to be done at positive or neagtive edge of pulse signal. Also, the limit values 20000 and -20000 are defined and are used to trigger the interrupt when the pulse count meets it's limit either in positive or negative.

```
pcnt_unit_config(&pcnt_config);

pcnt_set_filter_value(unit, 100);
pcnt_filter_enable(unit);

pcnt_set_event_value(unit,
PCNT_EVT_THRES_1, PCNT_THRESH1_VAL);

pcnt_event_enable(unit, PCNT_EVT_THRES_1);
pcnt_set_event_value(unit,
PCNT_EVT_THRES_0, PCNT_THRESH0_VAL);

pcnt_event_enable(unit, PCNT_EVT_THRES_0);
pcnt_event_enable(unit, PCNT_EVT_ZERO);
pcnt_event_enable(unit, PCNT_EVT_H_LIM);
pcnt_event_enable(unit, PCNT_EVT_L_LIM);

/* Initialize PCNT's counter */
pcnt_counter_pause(unit);
pcnt_counter_clear(unit);

pcnt_isr_service_install(0);
pcnt_isr_handler_add(unit, pcnt_example_intr_handler, (void *)unit);

/* Everything is set up, now go to counting */
pcnt_counter_resume(unit);
```

Once the setting up is done, it configures the unit and is also using the filter to filter-out the input signal with preset thresholds. Then, it is setting the threshold 0 and 1 values and enabling the events to watch, starting with zero till the maximum and minimum limits. After that, it initializes the PCNT's counter and an interrupt service routine callback handler to read the events triggered by the encoder input signals.

```
res = xQueueReceive(pcnt_evt_queue,
&evt, 1000 / portTICK_PERIOD_MS);
```

With this line of code, it reads the event every one second and through that event, the current pulse counter can be known. Subtracting the current pulse count value with the previous pulse count value, the pulse count value in one second can be known.

$$Pulse\ Count\ per\ sec = Current\ Pulse\ Count - Previous\ Pulse\ Count$$

And thus, the current RPM's value can be known with:

$$RPM_{current} = \frac{Pulse\ Count\ per\ sec \times 60}{Encoder\ Count\ per\ Revolution}$$

Now that, the current RPM value is known, the duty cycle can be found by:

$$Current\ Duty\ Cycle = 100.0 \times \frac{current\ RPM}{Maximum\ RPM}$$

The formula above gives the current duty cycle that has been applied to drive the motor. Given below is the implementation of these formulas as C program:

```
previous_value = Current_value;
Current_value = count;
wheel_pulse_count = Current_value - previous_value;
current_rpm_value =
(float) wheel_pulse_count * 60 / ENC_COUNT_REV;

previous_duty = current_duty;
current_duty = current_rpm_value * 100.0 / (float) MAX_RPM;
```

Note : The event only occurs when the motor is driven. When it is stopped, it is not possible to get that event. In order to tackle this problem, the polling system was implemented when there is no event and with that, the current state of the controller can be known.

2.3 PID Feedback Control

PID is the acronym for Proportional Integral Derivative. It is a controller which is implemented to be used as a feedback system with Proportional, Integral and Derivative Parameters in order to drive the DC motor. In order to control the revolution of the DC motor, first it is required to control the speed of the DC motor and speed of the DC motor is controlled via duty cycle.

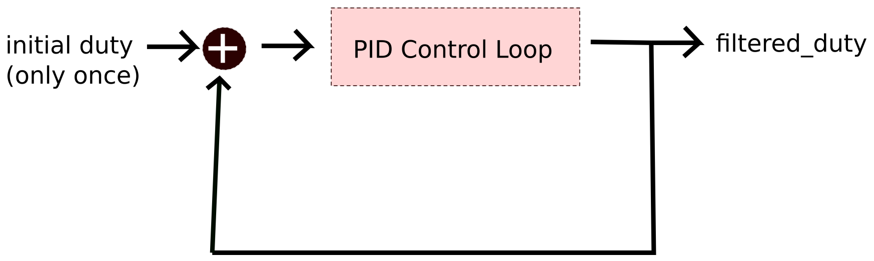


Figure 6: PID feedback control system [5]

It consist of three parameters: Proportional, Integral and Derivative. Depending on the values of these parameters, the PWM signal can be controlled.

The table give below shows the effects of the system response due to PID control gain values.

Gain	Rise Time	Overshoots	Settling Time
K_p	*Decrease*	*Increase*	*Small change*
K_i	*Decrease*	*Increase*	*Increase*
K_d	*Increase*	*Decrease*	*No change*

Table 1: PID parameter's effects on System Response [3]

Using table above and with trial and error, the PID parameter values were found out. With this parameters, the duty cycle of the DC motor has been stabilized and controlled. The C program given below shows the implementation of the PID feedback control and is being executed as a task in ESP32.

```
static void PID_execute(void *args){
  while(1){
    error = init_duty - current_duty;
    Integral_value += KI * error * 10;
    if(Integral_value > 1.0){
      Integral_value = 1.0;
    }
    if(Integral_value < -1.0){
      Integral_value = -1.0;
    }

    Derivative = (current_duty - previous_duty) / 10;

    control_value =
    (KP * error) + Integral_value - (KD * Derivative);

    if(control_value > 1.0){
      Integral_value = 1.0;
    }
    if(control_value < -1.0){
      Integral_value = -1.0;
    }
    filtered_duty =
    (float)(control_value * MAX_CONTROL_VALUE);
    vTaskDelay (2);    // Delay 20 ms = 2 Ticks
  }
}
current_duty =
current_rpm_value * 100.0 / (float) MAX_RPM;
```

Firstly it is important to know the error before solving it. So error is calculated by subtracting the initial driven duty cycle and the current duty cycle which is being read through the encoder. With that error, Integral value is calculated and a limit is set on the integral value that it can only be in between 1.0 and -1.0. Then, the derivative parameter value is calculated. Once we have all the parameter values, the control value is calculated. And then the limit is set to control value and then that value is fed to the DC motor. This can be seen in the Result and Conclusion chapter.

With this, one can control the speed of the motor. Once the speed of the DC motor is under control, moving forward to the actual objective, which is to control the position of the DC motor, is possible.

2.4 The Rotary Encoder

There has been some recent changed with the rotary encoder example and library in the ESP-IDF and it has not been updated in the latest version of the ESP-IDF. This project contains those files and part of that library is used to find the position and direction of the DC motor movement. Previously, one had to do some calculations to get the position and direction of movement of the DC motor but in the updated files, the library itself consist of those functions to get the position and direction of the DC motor movement. The full source code of this project can be found here [6]

Rotary Encoder also uses the PCNT unit to get information out of the encoder.

```
QueueHandle_t event_queue = rotary_encoder_create_queue();
ESP_ERROR_CHECK(rotary_encoder_set_queue(&info, event_queue));

while(1){
  rotary_encoder_event_t event = { 0 };
  if (xQueueReceive(event_queue,
  &event, 1000 / portTICK_PERIOD_MS) ==
  pdTRUE){
       previous_position = current_position;
       current_position = event.state.position;

       ESP_LOGI(TAG, "Event: position %d,
       direction %s, difference %d",
       event.state.position, event.state.direction ?
       (event.state.direction ==
       ROTARY_ENCODER_DIRECTION_CLOCKWISE ?
        "clockwise" : "Anti-Clockwise") : "Not set",
        current_position - previous_position);

      if(((event.state.position)/ENC_COUNT_REV) >=
      revtotake && revtotake > 0){
           printf("stopping motor\n");
           stop_motor = true;
      }
      else if(((event.state.position )/ENC_COUNT_REV) <=
      revtotake && revtotake < 0){
           printf("stopping motor\n");
           stop_motor = true;
      }
  }
}
```

In the code shown able, firstly an event is created to record the data from the encoder as a Queue Handler and set to store the data received through event in the info variable. Then, inside the infinite loop, the previous and current position is recorded and printed on the terminal via ESP_LOGI command.

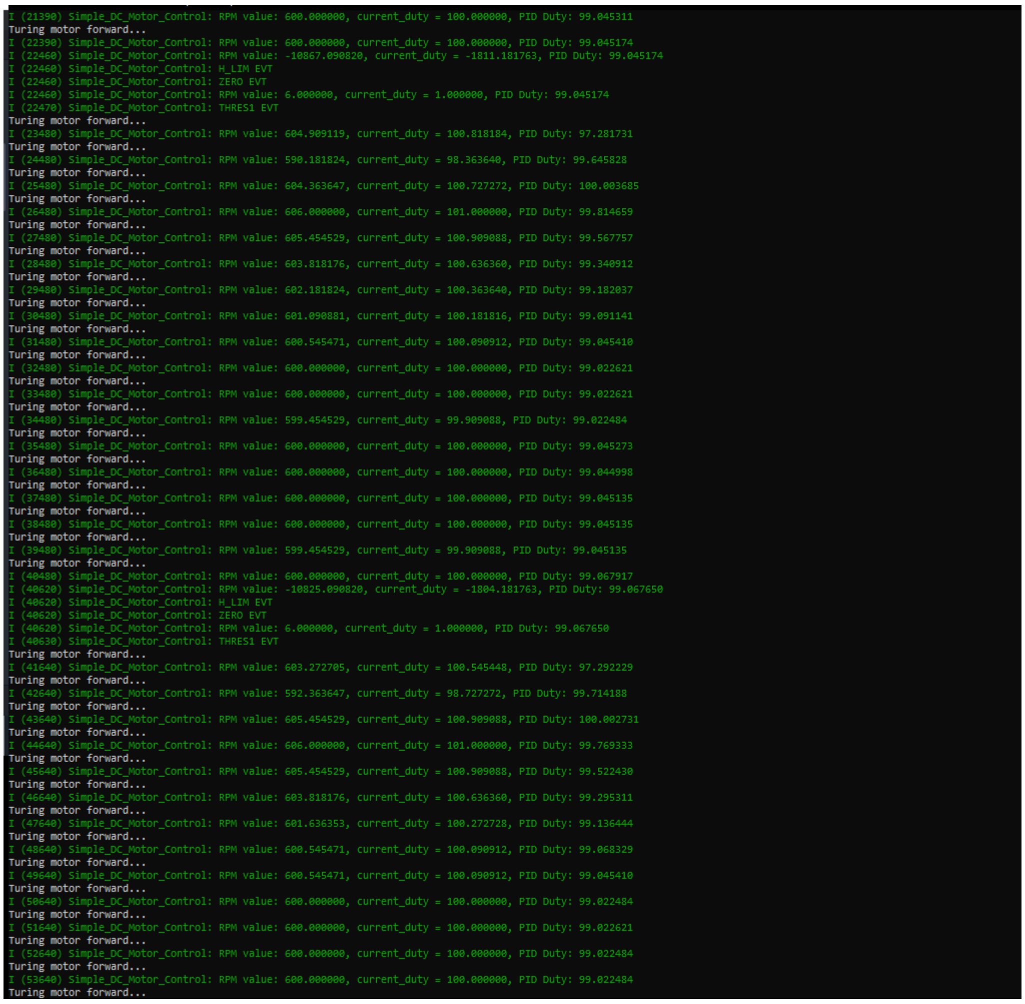
Note: It takes 110 position changes in one revolution of the dc motor. This differs from motor to motor. This value has been achieved via manual testing.

The current position is divided by encoder count per revolution value, which is 110 for the motor used in this project, and is compared with the revolution to take value. Revolution to take variable is for the user to set how many revolutions user needs to take. If the number is revolutions are achieved, the program stops the motor.

This is how the revolution to be taken is achieved.

3 Result and Conclusion

This chapter comprises of the results which has been recorded through out the project.



```
I (21390) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.045311
Turing motor forward...
I (22390) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.045174
I (22460) Simple_DC_Motor_Control: RPM value: -10867.090820, current_duty = -1811.181763, PID Duty: 99.045174
I (22460) Simple_DC_Motor_Control: H_LIM EVT
I (22460) Simple_DC_Motor_Control: ZERO EVT
I (22460) Simple_DC_Motor_Control: RPM value: 6.000000, current_duty = 1.000000, PID Duty: 99.045174
I (22470) Simple_DC_Motor_Control: THRES1 EVT
Turing motor forward...
I (23480) Simple_DC_Motor_Control: RPM value: 604.909119, current_duty = 100.818184, PID Duty: 97.281731
Turing motor forward...
I (24480) Simple_DC_Motor_Control: RPM value: 590.181824, current_duty = 98.363640, PID Duty: 99.645828
Turing motor forward...
I (25480) Simple_DC_Motor_Control: RPM value: 604.363647, current_duty = 100.727272, PID Duty: 100.003685
Turing motor forward...
I (26480) Simple_DC_Motor_Control: RPM value: 606.000000, current_duty = 101.000000, PID Duty: 99.814659
Turing motor forward...
I (27480) Simple_DC_Motor_Control: RPM value: 605.454529, current_duty = 100.909088, PID Duty: 99.567757
Turing motor forward...
I (28480) Simple_DC_Motor_Control: RPM value: 603.818176, current_duty = 100.636360, PID Duty: 99.340912
Turing motor forward...
I (29480) Simple_DC_Motor_Control: RPM value: 602.181824, current_duty = 100.363640, PID Duty: 99.182037
Turing motor forward...
I (30480) Simple_DC_Motor_Control: RPM value: 601.090881, current_duty = 100.181816, PID Duty: 99.091141
Turing motor forward...
I (31480) Simple_DC_Motor_Control: RPM value: 600.545471, current_duty = 100.090912, PID Duty: 99.045410
Turing motor forward...
I (32480) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.022621
Turing motor forward...
I (33480) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.022621
Turing motor forward...
I (34480) Simple_DC_Motor_Control: RPM value: 599.454529, current_duty = 99.909088, PID Duty: 99.022484
Turing motor forward...
I (35480) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.045273
Turing motor forward...
I (36480) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.044998
Turing motor forward...
I (37480) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.045135
Turing motor forward...
I (38480) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.045135
Turing motor forward...
I (39480) Simple_DC_Motor_Control: RPM value: 599.454529, current_duty = 99.909088, PID Duty: 99.045135
Turing motor forward...
I (40480) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.067917
I (40620) Simple_DC_Motor_Control: RPM value: -10825.090820, current_duty = -1804.181763, PID Duty: 99.067650
I (40620) Simple_DC_Motor_Control: H_LIM EVT
I (40620) Simple_DC_Motor_Control: ZERO EVT
I (40620) Simple_DC_Motor_Control: RPM value: 6.000000, current_duty = 1.000000, PID Duty: 99.067650
I (40630) Simple_DC_Motor_Control: THRES1 EVT
Turing motor forward...
I (41640) Simple_DC_Motor_Control: RPM value: 603.272705, current_duty = 100.545448, PID Duty: 97.292229
Turing motor forward...
I (42640) Simple_DC_Motor_Control: RPM value: 592.363647, current_duty = 98.727272, PID Duty: 99.714188
Turing motor forward...
I (43640) Simple_DC_Motor_Control: RPM value: 605.454529, current_duty = 100.909088, PID Duty: 100.002731
Turing motor forward...
I (44640) Simple_DC_Motor_Control: RPM value: 606.000000, current_duty = 101.000000, PID Duty: 99.769333
Turing motor forward...
I (45640) Simple_DC_Motor_Control: RPM value: 605.454529, current_duty = 100.909088, PID Duty: 99.522430
Turing motor forward...
I (46640) Simple_DC_Motor_Control: RPM value: 603.818176, current_duty = 100.636360, PID Duty: 99.295311
Turing motor forward...
I (47640) Simple_DC_Motor_Control: RPM value: 601.636353, current_duty = 100.272728, PID Duty: 99.136444
Turing motor forward...
I (48640) Simple_DC_Motor_Control: RPM value: 600.545471, current_duty = 100.090912, PID Duty: 99.068329
Turing motor forward...
I (49640) Simple_DC_Motor_Control: RPM value: 600.545471, current_duty = 100.090912, PID Duty: 99.045410
Turing motor forward...
I (50640) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.022484
Turing motor forward...
I (51640) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.022621
Turing motor forward...
I (52640) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.022484
Turing motor forward...
I (53640) Simple_DC_Motor_Control: RPM value: 600.000000, current_duty = 100.000000, PID Duty: 99.022484
Turing motor forward...
```

Figure 7: PID feedback control result

As it can be seen in the image above, the values of the RPM and current duty cycle are synchronized quite well. Current RPM and Current duty are the calculated values which are being read through the encoder and filtered value is the one which is coming out of the PID feedback control loop and is driving the DC motor.

Now then, through the code, the value can be set for how many revolutions the DC motor has to take and can be set by user. As the speed is controlled, so does the torque and thus the control over the position is accurate. Unfortunately, during this project the H-Bridge to control the motor was damaged. The reason was that in the code, both of the H-Bridge were running simultaneously to drive the motor and that overloaded the H-Bridge and it was damaged. Unfortunately, this report does not contain the result of the final object since we couldn't test it further. The motor showed the exact same location for DC motor after 10 revolution in the forward direction. Unfortunately, it was not possible to test it for backward direction but we are pretty much sure it is supposed to work for backward direction as well.

The code is available at Github repository. [6]

References

[1] *ESP32 Wifi and Bluetooth MCU*
`https://www.espressif.com`
[Accessed on 01/08/2021]

[2] *Amazon Website Description for Motor*
`https://www.amazon.com/25GA370-Encoder-Metal-Gearmotor-150RPM/dp/B07GNFYGYQ`
[Accessed on 01/08/2021]

[3] *Application Note Position and Speed Control of a DC Motor using Analog PID Controller*
`https://www.dialog-semiconductor.com/sites/default/files/an-cm-250_position_and_speed_control_of_a_dc_motor_using_analog_pid_controller.pdf`
[Accessed on 01/08/2021]

[4] *ESP-IDF github*
`https://github.com/espressif/esp-idf`
[Accessed on 01/08/2021]

[5] *Espressif Documentation*
`https://www.espressif.com/en/support/documents/technical-documents`
[Accessed on 01/08/2021]

[6] Prithvi Patel, *Github repository*
`https://github.com/Prithvipatel007/HighPrecisionMotorControl`
[Accessed on 01/08/2021]