

LabVIEW Based Closed Loop DC Motor Speed Control and Performance Analysis Under Variable Load Condition

Pranav Sajeev¹ Dilraj N V² Dr. Saravanakumar R³

^{1,2} School Of Electrical Engineering Vellore Institute Of Technology Vellore, India Dilraj.Nv2024@Vitstudent.Ac.In

³Department Of Control And Automation Vellore Institute Of Technology Vellore, India Rsaravanakumar@Vit.Ac.In

Abstract—This paper describes a low-cost closed-loop speed controller for a 12 V brushed DC motor, written so that the build can be repeated. An ESP32 times the pulses of an optical slot sensor on an 8-slot disc, filters the speed and sends it over USB serial every 20 ms. A LabVIEW VI runs the PID controller and returns a PWM command (0–255) that the ESP32 applies to an L298N driver. All hardware, firmware, VI and test settings are listed, together with the tuning procedure. Mechanical load is applied by hand with a friction pad. We analyse the delay in the loop and show, from the firmware parameters, that the speed filter (about 47 ms of mean lag) dominates the estimated 63 ms loop dead time, while USB-serial timing jitter is only a few milliseconds. In a demonstration run, the PID output stayed between 231 and 248 counts and the mean speed was within about 4 % of the setpoint, but the speed estimate was very noisy because of the hand-cut disc.

Index Terms—DC motor, PID control, LabVIEW, ESP32, variable load, serial communication, timing jitter.

I. INTRODUCTION

A DC motor slows down when it is loaded, and a fixed PWM setting cannot notice this. A speed sensor and a feedback controller can, which is why closed-loop speed control is a standard laboratory exercise. LabVIEW-based PID speed control of DC motors has been reported many times in student and low-cost instrumentation work [1]–[3], and the PID algorithm itself is well established [4]–[6]. We therefore do not claim a new control law.

What is often missing from such reports is the detail that decides whether a second team can repeat the work: the exact sensor and disc, the serial settings, the units of the speed signal, the way the load is applied, and the faults that cost days to find. A second gap is timing. When the controller runs on a PC and reaches the motor through a USB-serial link, the sample interval is not guaranteed, and delay and jitter are known to degrade control loops [7], [8]. Few low-cost reports say how large these effects actually are.

This paper addresses both gaps for an ESP32 –LabVIEW speed controller. Its contributions are: (1) a complete description of the hardware, firmware, VI settings and test conditions, sufficient for replication; (2) a delay budget for the loop, derived from the firmware parameters and serial

arithmetic, which shows where the dead time comes from and how the speed-filter weight trades noise against lag; (3) an analysis of a demonstration run that quantifies the controller

output and the measurement noise; and (4) a record of the faults met during the build and how to diagnose them.

Section II describes the system, Section III the controller and tuning procedure, and Section IV the timing analysis. Section V gives the test method, Section VI the results, and Section VII the discussion and limitations.

II. SYSTEM DESCRIPTION

1. Architecture

Fig. 1 shows the loop. The ESP32 and the L298N form the field side; LabVIEW on a laptop forms the supervisory side. Every 20 ms the ESP32 sends one line of text with the filtered speed. LabVIEW parses it, runs *PID.vi* against the operator setpoint, and returns the command PWM,n, which the ESP32 writes to the driver enable pin. Keeping the PID in LabVIEW allows the gains to be changed while the motor runs, without re-flashing the firmware.

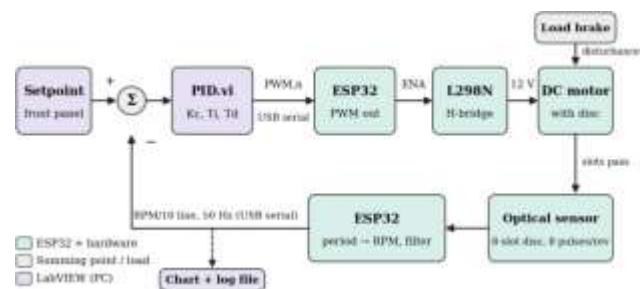


Fig. 1. Block diagram of the closed-loop speed control system.

2. Hardware and test conditions

Table I lists the components and the conditions that apply to every test. Photographs are shown in Fig. 2 and the wiring in Fig. 3 and Table II.

TABLE I Components and test conditions

Item	Specification
Controller board	ESP32-WROOM-32 DevKit (30-pin), Arduino core 3.x
Motor driver	L298N module, channel A; ENA jumper removed, IN1 high, IN2 low
Motor	12 V brushed DC, about 1000 rpm rated
Speed sensor	Slot-type photo-interrupter module, digital output (FC-03 LM393)

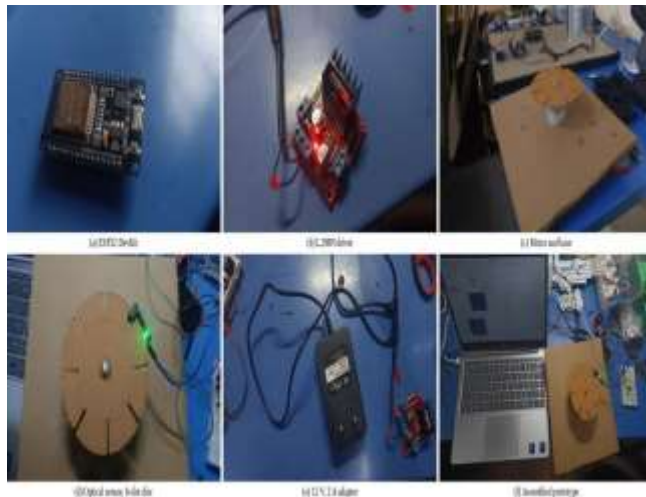


Fig. 2. Hardware used: (a) ESP32 DevKit, (b) L298N driver, (c) motor and base, (d) optical sensor and 8-slot disc, (e) 12 V adapter, (f) assembled prototype with the laptop.

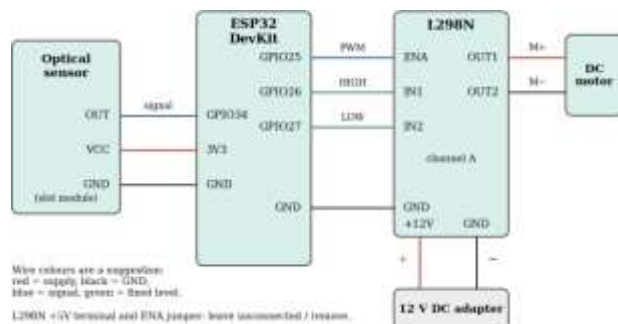


Fig. 3. Wiring diagram of the ESP32, L298N, optical sensor, motor and adapter.

TABLE II Pin and terminal connections

From	To	Purpose
GPIO 34	Sensor OUT	Pulse input
GPIO 25	L298N ENA	PWM speed
GPIO 26 / 27	L298N IN1 / IN2	Held high / low
3V3, GND	Sensor VCC, GND	Sensor supply
ESP32 GND	L298N GND	Common ground
OUT1 / OUT2	Motor	Motor drive
+12 V / GND	Adapter + / -	Motor supply

3. Firmware

The ESP32 firmware has three jobs. A falling-edge interrupt on GPIO 34 stores the time between successive slots. Every 20 ms the main loop converts that period to speed with $n = 60 \times 10^6 / (N \cdot T_{\mu s})$ (1)

where $N = 8$ is the number of slots and $T_{\mu s}$ the slot period in microseconds. The speed is smoothed by the first-order filter $y[k] = \alpha x[k] + (1 - \alpha) y[k-1]$, $\alpha = 0.3$ (2)

divided by 10, and printed as one line of text. Finally, any received line beginning with PWM, is limited to 0–255 and written to the LEDC channel. Table III gives the firmware parameters. Pulses closer than 1 ms are rejected as glitches, which sets an upper limit of $60 \times 1000/8 = 7500$ rpm. If no pulse arrives for 300 ms the speed is reported as zero, so speeds below about 25 rpm read as stopped.

TABLE III Firmware parameters

Parameter	Value
SLOTS_PER_REV	8
MIN_PULSE_US	1000 (1 ms)
STOP_TIMEOUT_US	300000 (300 ms)
FILTER_ALPHA	0.3
LOOP_PERIOD_MS	20 (50 Hz)
Serial timeout	5 ms
Output scaling	speed ÷ 10

Units. Because of the division by 10, LabVIEW works in scaled units: a reading of 25.0 means about 250 rpm, and the setpoint is entered in the same units. The scaling keeps the speed error of the same order as the 0–255 output range, so a proportional gain of order 0.1 is sensible. In this paper all speeds are in these indicated units unless stated.

Calibration. The nameplate rating of about 1000 rpm is lower than the 2500 rpm that the indicated setpoint of 250 would mean. The absolute scale of the indicated speed is therefore not verified. The calibration check described in

Section V (stopwatch count of disc turns at a fixed PWM)

gives the factor between indicated and true speed. Control performance expressed as a percentage of the setpoint does not depend on this factor.

4. LabVIEW program

The VI (Fig. 4) opens the serial port, waits, flushes, and then runs one While Loop until Stop is pressed. Table IV lists the serial settings. Each block has a reason, given here because several of them were added after a fault.

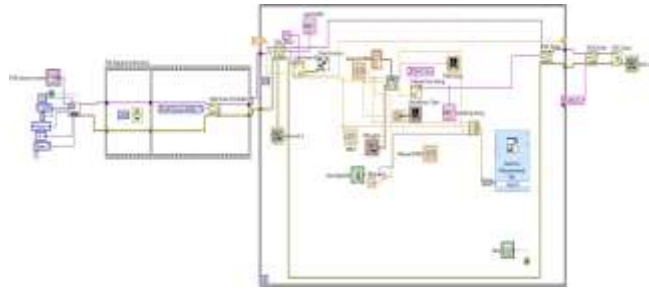


Fig. 4. Block diagram of the LabVIEW VI.

Table Iv Visa Serial Settings

Setting	Value
Resource	COM port from a front-panel list
Baud rate	115200
Data / parity / stop	8 / none / 1.0
Flow control	none
Timeout	500 ms
Termination character	enabled (line feed)
Read length	20 bytes

- **Wait 2000 ms and VISA Flush I/O Buffer.** Opening the port resets many ESP32 boards, which then print boot text. Without the wait and flush, this text reaches the number parser and raises error 85 (scan failed).
- **VISA Read and Scan From String (%f).** The read returns one line, because the termination character ends it. The scan turns the text into a number. Clear Errors.vi stops one unreadable line from halting the loop, and a shift register feeds the previous good value to the scan block, so a failed scan repeats the last speed instead of reporting zero.
- **PID.vi.** The inputs are the setpoint, the measured speed and the gains cluster (K_c , T_i , T_d , with times in minutes). The output range is set to 0–255 so that the VI limits the output and stops the integral from winding up during saturation. The time step is left unwired and measured by the VI.
- **Format Into String and VISA Write.** The string PWM,%d\n builds the command. The backslash code

must be shown in “\ codes” mode so that a real line feed is sent.

- **Charts and logging.** A waveform chart shows the setpoint, speed and a load marker (the Load Applied switch converted to 0/1 and multiplied by 100). A second chart shows the PWM. Build Array and Write To Measurement File save the setpoint, speed, PWM and load flag to an LVM file for every run.
- **Shutdown.** After the loop, a second VISA Write sends PWM,0 and VISA Close releases the port. Runs must be ended with Stop, not the abort button, or the motor is not commanded off and the port may stay locked.

this system. They are estimates from the firmware parameters and the serial arithmetic, not an end-to-end measurement.

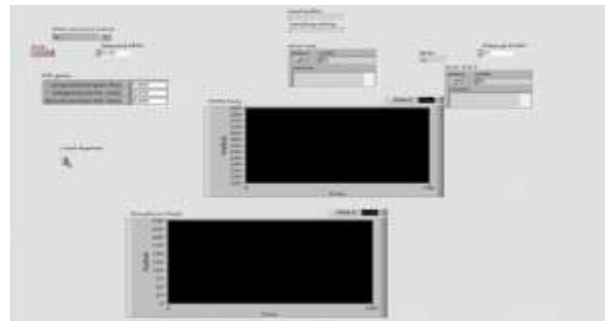


Fig. 5. LabVIEW front panel: port selector, setpoint, PID gains, load switch, indicators and charts.

III. CONTROLLER AND TUNING PROCEDURE

PID.vi uses the ideal form [14]

$u = K_c [e + (1/T_i) \int e dt + T_d de/dt]$ (3) where e is the setpoint minus the indicated speed and u is the PWM count. The integral gain in counts per scaled unit per second is $K_c/(60 T_i)$ when T_i is in minutes. We tuned by hand, in the following order, which follows the usual practical approach [5], [6], [9]:

1. **Proportional only.** Set T_i large and $T_d = 0$. Raise K_c until the speed follows the setpoint quickly but starts to oscillate slightly, then reduce K_c by about one third.
2. **Add integral action.** Reduce T_i slowly until the remaining gap closes without slow oscillation.
3. **Derivative.** Keep $T_d = 0$ unless recovery is sluggish. The speed estimate is noisy, and derivative action amplifies noise.
4. **Freeze the gains.** Use the same gains for all load levels and record them with every run.

The gains used in the demonstration run of Section VI were $K_c = 0.050$, $T_i = 0.050$ min (3 s, so the integral gain is 0.017

counts per scaled unit per second) and $T_d = 0$, i.e. a PI controller. Windup protection is the one built into *PID.vi* through the output range.

IV. TIMING ANALYSIS: USB-SERIAL DELAY AND JITTER

A controller reached through a serial link has two kinds of timing error: a mean delay (dead time), which reduces phase margin, and jitter, which makes the sampling interval vary [7], [8]. For a plant with a time constant τ and total dead time L , PI and PID control work well when L/τ is small, and dead-time compensation is considered only when it is large [5], [10]. Table V lists the contributions to L for

TABLE V Estimated loop dead time contributions

Source	Basis	Estimate (ms)
Sample hold	20 ms report period, mean age	10
Pulse-period lag	half a slot at 2500 rpm (if the scale is true)	1.5
Speed filter	mean lag $(1-\alpha)Ts/\alpha$, $\alpha = 0.3$	46.7
Serial transfers	about 6 + 8 bytes at 115200 baud (87 μ s per byte)	1.2
USB bridge buffering	1–2 USB frames; some drivers up to 16	2
LabVIEW and ESP32 handling	loop execution, command parsing	2
Total		≈ 63

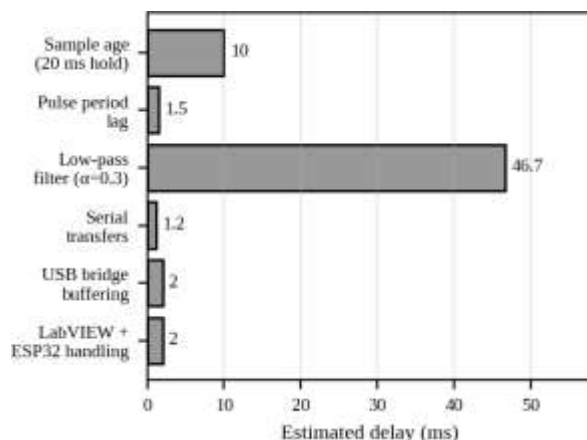


Fig. 6. Estimated dead-time contributions (Table V). The speed filter dominates.

Two results follow. First, the filter is the largest single contribution: about 47 of the 63 ms. The USB -serial path (transfers, bridge, handling) adds about 5 ms, and its jitter, which is a fraction of that plus the operating-system scheduling of the VI, is of the order of a few milliseconds. Second, jitter in the sampling instant matters little here because *PID.vi* measures the actual time step for its integral term, so the integral is not corrupted when an interval is long. As an illustration, a loop with a crossover frequency of 2 rad/s loses $\omega L = 2 \times 0.063 = 0.13$ rad (about 7°) of phase margin from the 63 ms dead time, but a ± 2 ms variation changes this by only 0.004 rad (0.2°).

The filter weight therefore sets the main trade-off. Fig. 7 shows, for uncorrelated noise, how the lag and the residual noise change with α . Reducing α from 0.3 to 0.1 lowers the noise standard deviation from 0.42 to 0.23 of its raw value, but raises the mean lag from 47 ms to 180 ms. Because the lag adds directly to the dead time, filtering harder is only acceptable if L/τ stays small. The measured jitter of the real loop is reported in Table IX.

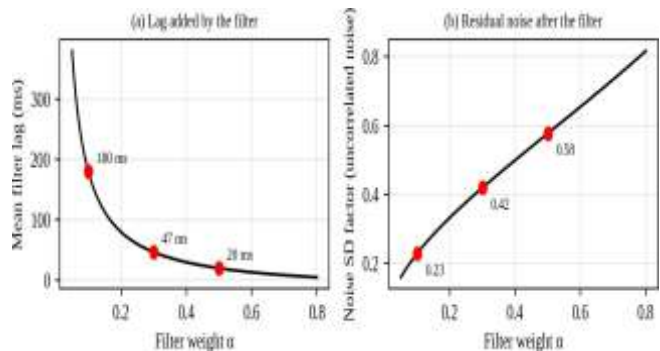


Fig. 7. Effect of the filter weight α at $T_s = 20$ ms: (a) mean lag, (b) residual noise for uncorrelated noise. The marked points are $\alpha = 0.1, 0.3$ and 0.5 .

Comparison of controllers. The load test was repeated

V. TEST METHOD

Calibration. Before any test, the displayed speed was compared with a stopwatch count of disc turns over 10 s at a fixed PWM (PWM,150 typed in the Arduino Serial Monitor) and the conversion between indicated and true speed was noted.

Step response. From rest, with no load, the setpoint was set to a chosen value and the run was logged for 40 s. Rise time is the time from 10 % to 90 % of the setpoint, overshoot is the peak above the setpoint in percent, settling time is the

time after which the speed stays within $\pm 2\%$, and steady-state error is the mean setpoint minus speed over the last 5 s.

Load disturbance. At a constant setpoint, once the speed was steady, a soft friction pad was pressed gently on the rotating shaft while the Load Applied switch was flipped at the same instant. The pressure was repeated at two or three levels, at least three times each, since brush contact and friction vary between runs. The speed dip is the maximum fall below the pre-load speed and the recovery time is the time to return to within $\pm 2\%$ of the setpoint. Because the speed is noisy, dip and recovery are computed on a 0.5 s moving average.

with P-only, PI and PID at the same setpoint and load.

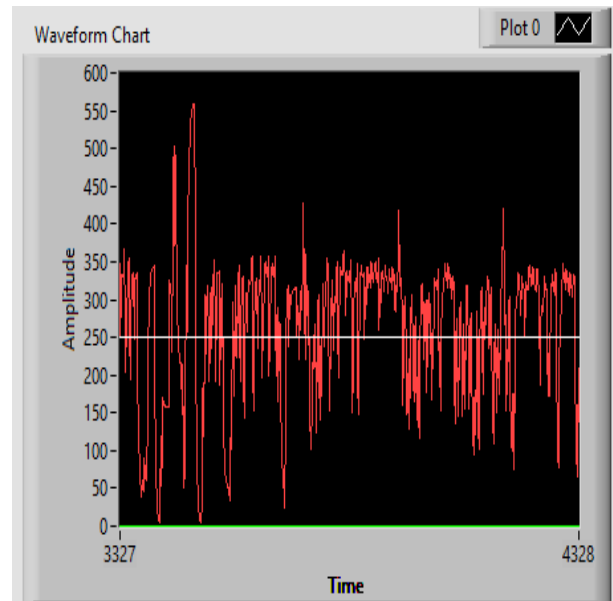
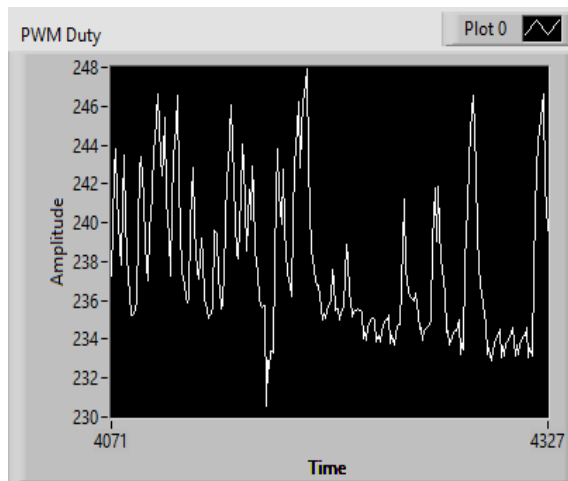
Timing. The loop interval was obtained from the time column of the logged LVM file (differences between successive samples). Mean, standard deviation, minimum, maximum and the fraction of intervals longer than 1.5 times nominal are reported.

Data processing. All metrics were computed from the logged files with a script (analyze_lvm.py), so that every number in Section VI can be reproduced from the raw data. Load levels are described by the pre-load and post-load PWM, because torque was not measured.

VI. RESULTS

1. Demonstration Run: Controller Output And Speed

Fig. 8 shows the two charts from a closed-loop run at a setpoint of 250 indicated units with $K_c = 0.050$, $T_i = 0.050$ min and $T_d = 0$. The charts were digitised from the screenshot, calibrated with the axis limits and the setpoint line (the 250 line was found at the expected pixel row). The accuracy is about one pixel: 0.08 PWM counts on the PWM chart and 2.4 indicated units on the speed chart. The charts cover different spans (256 and 1000 samples), so the PWM chart corresponds to the last quarter of the speed chart.



a. PWM Duty chart (controller output, 0–255). (b) Waveform chart: measured speed (red) and setpoint 250 (white). Fig. 8. Charts from the demonstration run (separate charts of the front panel).

Controller output. The PWM stayed between 230.6 and 247.9 counts (mean 237.9, standard deviation 3.7), that is 90–97 % of full scale. The output was never at the 255 limit, so the PID was not saturated, but it worked close to the limit. The motor therefore needed nearly full drive for this setpoint, partly because the L298N drops about 2 V. A further load would soon exhaust the range, and the speed would then fall regardless of the gains. For this reason load tests should be done at a setpoint that leaves PWM headroom.

Speed. The mean of the speed trace was about 260 units, 4 % above the setpoint of 250 (median about 282), so the integral action kept the average speed in the region of the setpoint. The instantaneous reading, however, scattered widely: the trace spanned about 2 to 559 units, the standard deviation of the per-column midpoints was about 86 units (about 33 % of the mean), 10 % of the columns reached below 100, and 60 % reached above 300. A DC motor with inertia cannot change speed by hundreds of rpm many times per second, so this scatter is mostly measurement noise.

Consistency check. With $K_c = 0.050$ and a speed scatter of about 86 units, the proportional term alone predicts an output scatter of about 4.3 counts. The observed PWM standard deviation of 3.7 counts agrees with this, which supports the reading of both charts and shows that the controller passes about K_c of the speed noise to the output.

VII. DISCUSSION

Noise sources. The speed estimate uses one slot period at a time. A disc cut by hand from cardboard has unequal slot spacing and widths, and a small wobble. Each error appears directly in the period, and so in the speed, once per revolution and at its harmonics. This is the most likely reason for the scatter in Fig. 8(b), together with stray noise edges. It could be tested by averaging over exactly one revolution (eight slots), which would cancel the pitch errors, and by replacing the disc with a printed or machined one. We have not yet tested either.

Serial parsing. A second possible source of sudden dips is a partial line from VISA Read, which Scan From String converts to a small number. The two causes can be separated by watching the Arduino Serial Monitor at the same speed with LabVIEW closed: if the numbers there are smooth, the fault is in the LabVIEW read; if not, it is in the sensor or firmware.

Effect of USB-serial delay and jitter. The analysis of Section IV shows that, for this system, the USB-serial path contributes only a few milliseconds of the estimated 63 ms dead time, and its jitter changes the phase margin by a fraction of a degree in the illustrative case. The deliberate speed filter matters more. This suggests that effort spent on reducing USB latency would give little benefit, while the filter weight and the noise at its source deserve attention. The conclusion depends on the plant time constant, which should be measured.

Practical faults. Table X lists the faults met and their fixes. Port errors look alarming but come from how Windows and VISA hold the port [15], not from the control program.

Symptom	Cause	Fix
Error 85 at start	Boot text parsed	Wait 2 s, flush, Clear Errors
Error -1073807246	Port busy	Close Arduino IDE and Serial Monitor
Error -1073807194	Connection lost	Replug USB, new cable or port, reselect port
PWM has no effect	ENA jumper fitted	Remove jumper
Motor will not start	L298N dead zone	Use a higher setpoint, raise K_c
PWM spikes	Failed scan gave zero	Shift register holds last value

inertia cannot change speed by hundreds of rpm many times per second, so this scatter is mostly measurement noise.

Limitations. The load is applied by hand, so the load level is not repeatable and is not measured as torque; it is described only through the PWM it requires. The absolute speed scale is unverified until calibrated. A single-channel sensor gives no direction. The motor sits near the top of its PWM range at the demonstration setpoint. Only one motor was tested. Finally, the delay budget of Table V is an estimate, not an end-to-end measurement.

Future work. We plan to compute speed over a full revolution, to use a better disc, to replace the hand load with a second motor acting as a loaded generator, to add a current sensor, and to move the PID to the ESP32 so that USB timing leaves the loop.

VIII. CONCLUSION

We described an ESP32–LabVIEW speed controller in enough detail to be rebuilt, including hardware, firmware, VI settings, tuning and test conditions. A delay analysis showed that the speed filter, not the USB-serial link, dominates the estimated 63 ms loop dead time. In a demonstration run the PID output stayed within 231–248 counts and the mean speed stayed near the setpoint, but the hand-cut disc made the speed estimate very noisy. The main remaining work is a calibrated, repeatable load and a cleaner speed measurement.

Acknowledgment

The authors thank the faculty and laboratory staff of Vellore Institute of Technology, Vellore, for the facilities provided.

REFERENCES

1. P. S. Vikhe, N. Punjabi, and C. B. Kadu, "DC motor speed control using PID controller in LabVIEW," *Int. J. Innovative Sci. Modern Eng.*, vol. 3, no. 3, Feb. 2015.
2. A. A. Hassen, "Real time digital speed control system for DC servo motor using LabVIEW 8.5 package," *Eng. Technol. J.*, vol. 28, no. 6, pp. 1169–1185, 2010.
3. S. Gupta, P. Kumar, V. Gupta, Salim, and A. Kanungo, "Speed and position control of DC motor using LabVIEW," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 9, no. 8, Aug. 2021.
4. K. Ogata, *Modern Control Engineering*, 5th ed. Upper Saddle River, NJ, USA: Pearson, 2010.
5. K. J. Åström and T. Hägglund, *PID Controllers: Theory, Design, and Tuning*, 2nd ed. Research Triangle Park, NC, USA: ISA, 1995.
6. K. H. Ang, G. Chong, and Y. Li, "PID control system analysis, design, and technology," *IEEE Trans. Control Syst. Technol.*, vol. 13, no. 4, pp. 559–576, Jul. 2005.
7. B. Wittenmark, J. Nilsson, and M. Törngren, "Timing problems in real-time control systems," in *Proc. Amer. Control Conf.*, Seattle, WA, USA, 1995, pp. 2000–2004.

8. J. Nilsson, B. Bernhardsson, and B. Wittenmark, "Stochastic analysis and control of real-time systems with random time delays," *Automatica*, vol. 34, no. 1, pp. 57–64, 1998.
9. J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," *Trans. ASME*, vol. 64, pp. 759–768, 1942.
10. J. E. Normey-Rico and E. F. Camacho, *Control of Dead-time Processes*. London, U.K.: Springer, 2007.
11. K. J. Åström and L. Rundqwist, "Integrator windup and how to avoid it," in *Proc. Amer. Control Conf.*, Pittsburgh, PA, USA, 1989, pp. 1693–1698.
12. Espressif Systems, "ESP32 series datasheet," Shanghai, China.
13. [13] STMicroelectronics, "L298: Dual full-bridge driver," datasheet.
14. National Instruments, *LabVIEW PID and Fuzzy Logic Toolkit User Manual*, Austin, TX, USA.
15. Keysight Technologies, "VISA error codes," Keysight IO Libraries Suite help.