Adjustable PID Tuner

David Zambrano & Eugenio Sotelo
CDA3631 Section 01

Introduction:

This project is an adjustable PID tuner that allows a user to test different parameter configurations and visualize how it affects the control of a servo motor. The code supports the changing of the Proportional, Integral, and Derivative gain variables used for the simulation via potentiometers. In this way, a user can visually observe how adjusting these gain values affects the response of a control system, in which said control system aims to bring a servo to a desired angle and remain in steady state.

Bill of Materials:

- STM32 Nucleo F446RE.
- Analog Feedback Servo.
- 3x 10KΩ Potentiometers.
- Rocker Switch.
- 2x 1kΩ Resistors.
- 0.1 µF Capacitor.
- 4x M4 X 10mm Button Head screws.
- 4x M4 Nuts.
- Breadboard.
- Breadboard jumper wires.
- PET-G Filament.

Methods:

Breaking down the code into the different sections, we have a few functions that handle the intake of data and some conversion math. Below are the functions that are used in the code:

- **map()** - Maps potentiometer ADC values to appropriate gain values for PID variables. Using the 12-bit ADC, we convert the 0 - 4095 integer range into 0 - 5 float range.
- **read_servo_feedback()** - Adc2 is used to read the analog feedback from the servo motor. This analog feedback provides us with the current position of the servo, read in the 0 - 4095 range. This function starts adc2, reads value on adc2, stops adc2, and returns the read value.
- **adc_to_angle()** - Converts the adc2 analog feedback to the respective servo angle for proper position reading.
- **us_to_counts()** - Converts a time to counts. This is done by using the frequency of the timer.

- **set_servo_angle()** - Checks that angle is within bounds, and converts that angle to a pulse length which the servo expects. Converts the pulse length from time to counts using us_to_counts. We use timer 3 to handle the PWM send to the servo.
- **UART_SEND()** - Passes a string to the COM port via the UART transmit function. Includes mutex acquire and release to prevent priority inversion.

For the main bulk of the code, The FreeRTOS middleware was used. We broke down the two main sections of code into two tasks. The code that handles the changing of the PID gain values via ADC and returning to the 0 angle was placed in a normal priority task. The second task runs the PID loop with the selected gain values, and prints the output, error, and the current servo angle. Regarding the ADC, DMA in conjunction with scanning mode on adc1 was used. Here, we use 3 channels to scan for the gain potentiometer values and use the map() function to scale them accordingly.

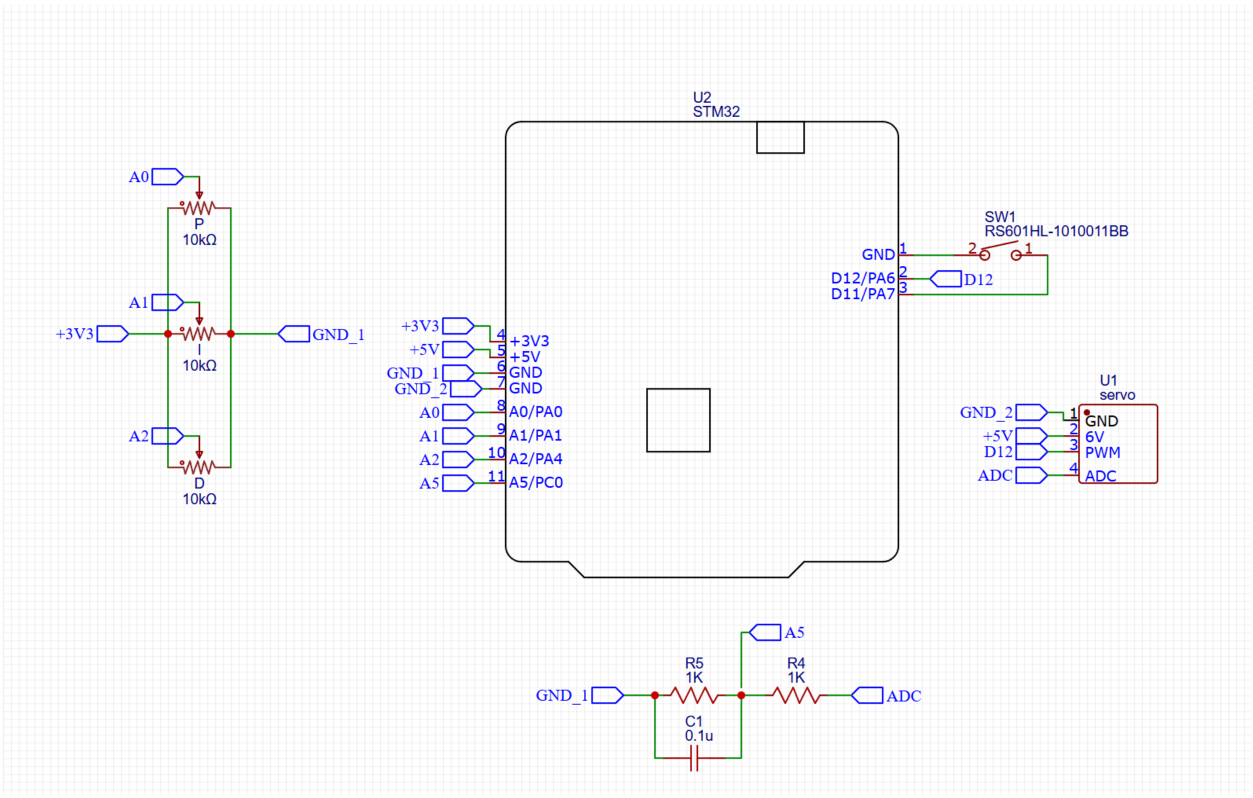

Figure 1: Wiring Diagram

The wiring diagram is very simple. All three potentiometers get 3.3V and GND, with the outputs going to A0-A2. We have a switch between GND and D12, which is our on/off toggle. The servo receives 5V, GND, and a PWM input from D12. Since the analog feedback from the servo can reach up to 5V, we use a ½ voltage divider by using two equal-value resistors to take the analog feedback down to a safe level for the board. The capacitor is in parallel with the second resistor to clean up the signal. These descriptions are visualized in the wiring diagram in Figure 1.

Results & Discussion:

After much struggle along each step of the way, our code reached a complete state. Once the bugs and errors were tackled internally, we found that the PID tuner was working as intended and was adjusting the output to reduce the error to 0 as best as it could. Ultimately, this is a demonstration device, where a user can visually observe how changing the proportional, integral, and derivative gain values affects the output response. For this purpose, the device works exactly as intended. The main issue with the device is the precision of the ADC, in which the read values from the potentiometer tend to vary despite 0 input from the user. This also affects the PID loop itself, where the analog feedback from the servo is not always accurate or precise. Despite this adc inaccuracy, however, the PID loop and its gain values do dramatically affect how well the servo reaches its target.

Conclusion:

This project was a success even with the problems we had at the beginning. We initially struggled with DMA for multiple channels on a single ADC and FreeRTOS now meshing properly. This issue was resolved, and we were able to work on mapping ADC inputs from the potentiometers to the PID variables, as well as the analog feedback from the servo to an angle. From there, debugging was tedious but simple to have our code run smoothly and reliably. Our original design had to be modified, as we ran into a few issues with the code, and we wanted to focus on having a simple but more robust system. While we did struggle a decent amount with the FreeRTOS implementation, this project worked mostly how we intended. We learned a lot about FreeRTOS and how a complete project using an STM32 board would look.

GitHub Repository:

https://github.com/TheDogIsHungry/PIDTuningDeviceSTM32

Acknowledgment:

This project was created as a final project for CDA3631 - Embedded Operating Systems. This course goes over the fundamentals of real-time operating systems, with a focus on FreeRTOS. The instructor for this course is Professor Hoan Ngo - hngo@floridapoly.edu.

References:

https://www.st.com/en/evaluation-tools/nucleo-f446re.html - STM32 Development Board.
https://www.adafruit.com/product/1404 - Analog Feedback Servo.
https://scholar.google.com/citations?user=niMq2ssAAAAJ&hl=en - Professor Profile.