FreeRTOS PID Controller

ECE 544: Project 3
Spring 2022

Alex Beaulier, beaulier@pdx.edu
Erik Fox, erfox@pdx.edu

Introduction:

The objective of this project is to gain experience with closed loop control, multi-tasking, synchronization and IPC using FreeRTOS in a complex microprocessor based system.

Main Objectives or outcomes of the Project:

The main outcome is to have a closed loop feedback of a small dc motor with user inputs to control the target speed and compensation mechanism to reach the desired speed. The system shall use FreeRTOS to respond to the user, varying loads and adjust a PWM cycle to the motor maintaining speed.

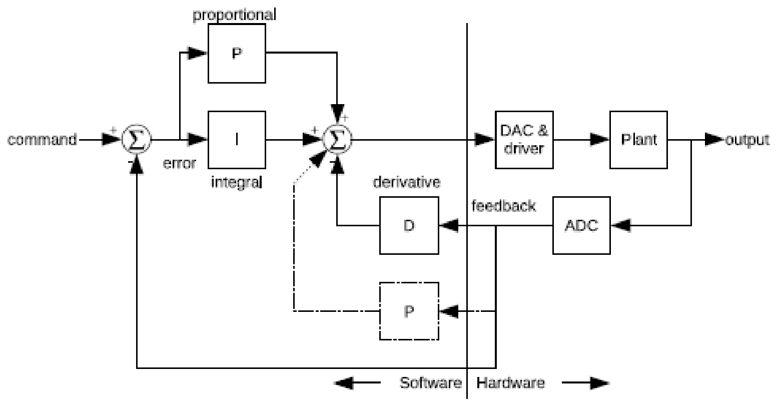


Figure 1: *Example PID Control diagram.*

Description of the project:

The team will create the system using a PMOD ENC, PMOD HB3, PMOD OLEDRGB, NexysA7 including SSEG display, pushbuttons, switches, and LEDs. The OLED displays the PID values, actual motor speed, target speed and selected KPID modifying parameter. The ENC is used for increasing and decreasing motor speed along with a switch for direction. The Nexys switches are used for activating the WDT to reset and other switches for scaling the KPID and ENC increase as well as selecting which KPID to modify. The pushbuttons up or down increase the value of the KPID parameters and the center resets the system to 0. The SSEG displays the target RPM. UART is used for debugging.

Division of Labor

The project was broken into two main sections. The first section involved the hardware development of the HB3 peripheral, including developing a custom module to drive a PWM signal, a custom module to package sensor data from the DC motor encoder, and drivers for interfacing to this peripheral, verifying that the peripheral and its drivers work as expected, building the embedded system in Vivado, and wiring the hardware system. The second section was the software section of the project, which included development of the main program, conversion to FreeRTOS from a superloop, upgrading functions to a new PMOD ENC driver, using existing projects to develop various functions and tasks to support FreeRTOS and debugging of the system and reporting hardware fixes, and testing for a fully functional PID controlled motor. Alex took the software development and Erik handled the hardware implementation. Both members debugged the software together.

Theory of Operation - Software

PID Background:

PID control, proportional, integral and derivative terms are three key terms needed to implement the controller. The proportional term is the error between the current and target multiplied by a constant kp. The integral term is long term added compensation to a control loop combining old error with the new error multiplied by constant ki.This is used after kp for critically dampening the response after a stabilized degrading overshoot is found. Lastly the derivative, kd is used for future prediction and was not utilized in our system but is available. The derivative term calculates future error.

Project 3 implemented a PID controller for the DC motor. The input to the controller was a desired speed from the user scaled linearly from 0-255(PWM) duty cycle to the hardware over the full scale range of 0-MaxRPM. The PID controller calculates each term and generates an output for the system to reach the new desired RPM. The RPM output was converted back into a PWM signal and sent to hardware. Limits were put in place for overshoot of the integral and the max pwm limit of 255.

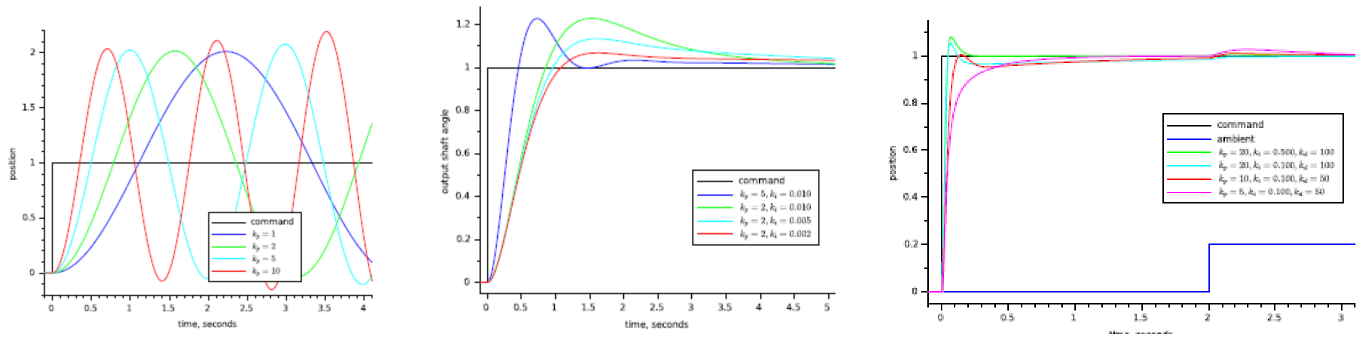


Figure 2: *PID positioning schemes, P, PI, PID. [1]*

FreeRTOS setup:

The FreeRTOS setup for our system uses a preemptive scheduler with priority assignment. This allows the system to run higher priority tasks in the system by context switching and gives a faster response for a “real time” appearance of the system. Main tasks were divided as input, display and PID control. The three threads were run with highest priority given to the display task as it required significant time to operate display panels. The input was given lowest priority as it only required occasional use from the interrupt driven GPIO. The GPIO used a semaphore to flag incoming information to the system. The PID Controller was given second priority to run the motor but was found to be limited in efficiency as the encoder was limited to one second updates. The input thread also passed messages to both threads, one containing the updated KPID variables and desired RPM to the PID Control thread, the other being the variables to the display thread for updating the user of the current input and outputs of the system displayed on the OLED.

The PID Controller read the current RPM for calculations, performed an updated output setpoint which was forwarded to hardware and the updated reading was sent to the OLED display thread as well. All three threads were run in a master thread task and each contained a super while loop for continuous operation. The watchdog timer additionally allowed the entire system to be reset through a switch control which would instantiate this system.

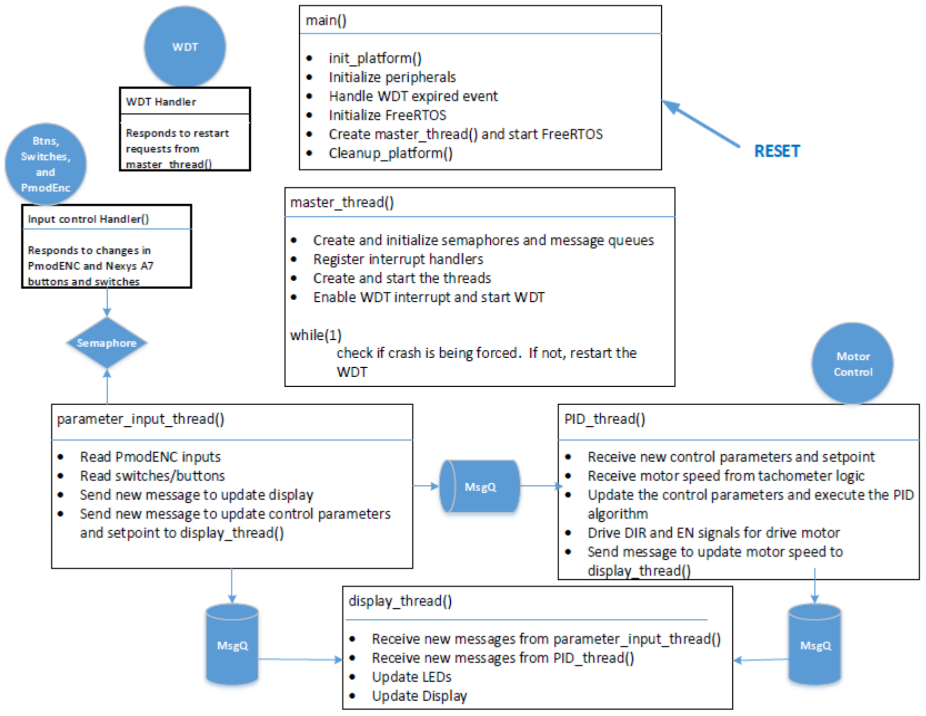

WDT
WDT Handler
Responds to restart requests from master_thread()
main()
• init_platform()
• Initialize peripherals
• Handle WDT expired event
• Initialize FreeRTOS
• Create master_thread() and start FreeRTOS
• Cleanup_platform()
RESET
Btns, Switches, and PmodEnc
Input control Handler()
Responds to changes in PmodENC and Nexys A7 buttons and switches
master_thread()
• Create and initialize semaphores and message queues
• Register interrupt handlers
• Create and start the threads
• Enable WDT interrupt and start WDT
while(1)
check if crash is being forced. If not, restart the WDT
Semaphore
Motor Control
parameter_input_thread()
• Read PmodENC inputs
• Read switches/buttons
• Send new message to update display
• Send new message to update control parameters and setpoint to display_thread()
MsgQ
PID_thread()
• Receive new control parameters and setpoint
• Receive motor speed from tachometer logic
• Update the control parameters and execute the PID algorithm
• Drive DIR and EN signals for drive motor
• Send message to update motor speed to display_thread()
MsgQ
display_thread()
• Receive new messages from parameter_input_thread()
• Receive new messages from PID_thread()
• Update LEDs
• Update Display
MsgQ

Figure 3: *FreeRTOS scheme. [1]*

<u>State Diagrams & Circuit Diagrams</u>

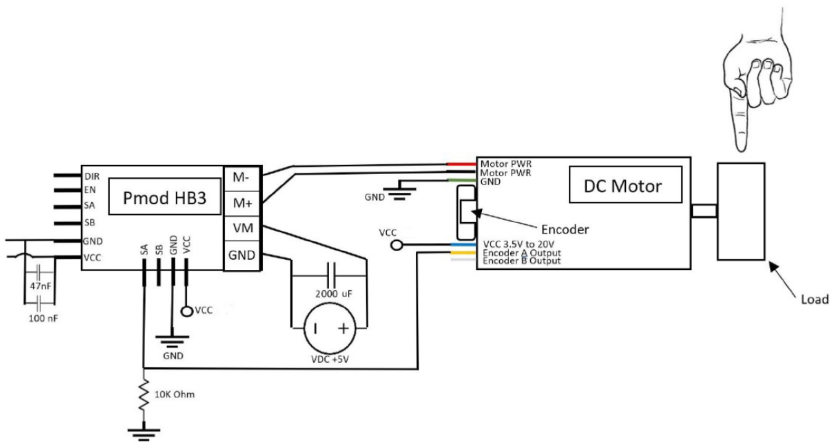
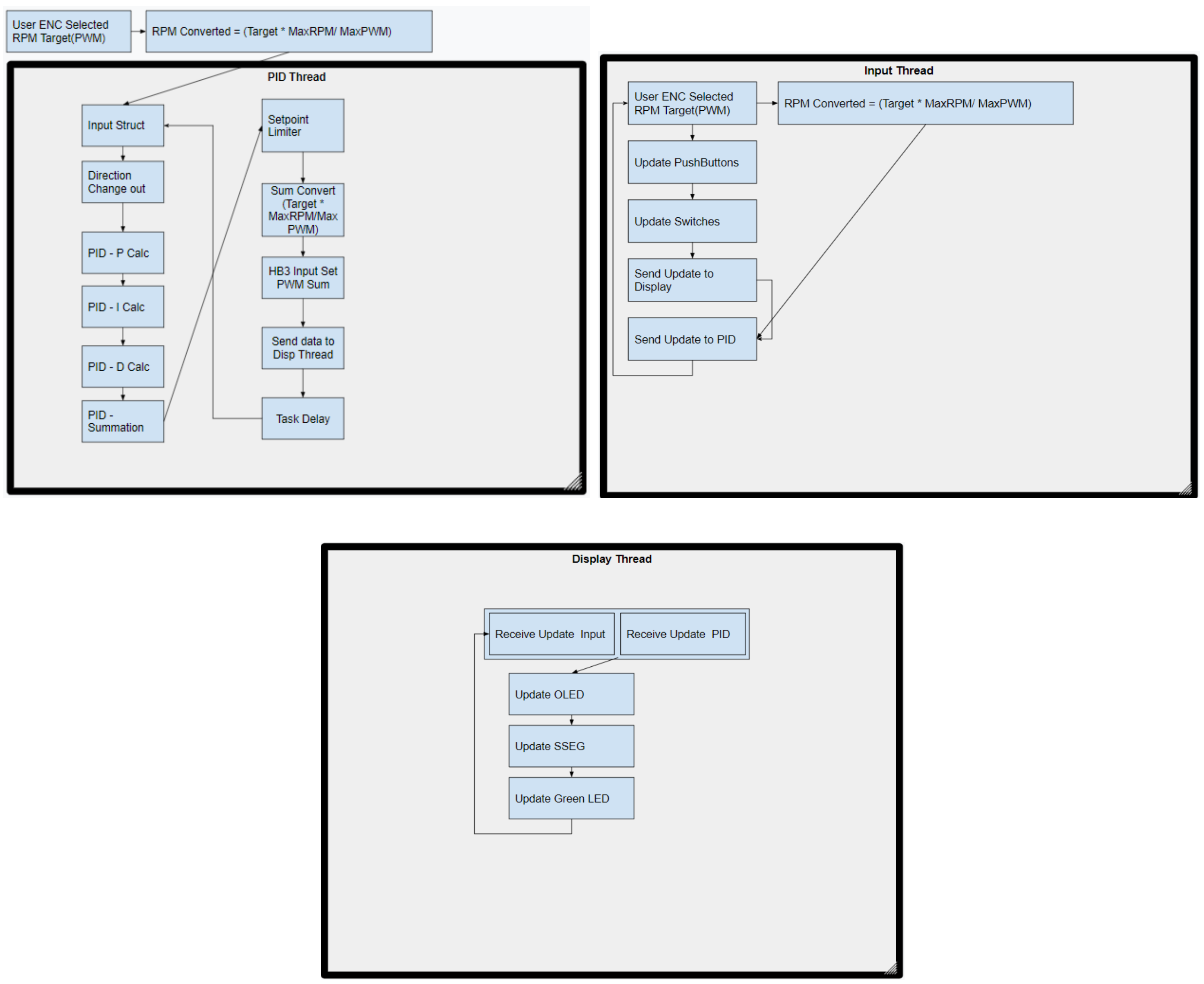


Figure [3.5]: Flowchart of PID controller.



Figures 4: PID, Input and Display Thread state machines.

C Code Skeleton Layout

1. Initialize all peripherals, platform, watchdog and interrupts.
2. If Watchdog reset from main is expired, restart it
3. Generate the master thread
    a. Create three threads with priorities and queue
    b. Display Thread
        i. Receives data from both threads
        ii. Updates OLED
        iii. Updates SSEG
        iv. Updates Green LEDs
    c. Input Thread
        i. Reads Rot ENC
        ii. Reads Push buttons

        iii. Reads Switches
        iv. Send data to other threads
    d. PID Thread
        i. Receive input thread data
        ii. Update direction
        iii. Calculate RPM PID
        iv. Send target to hardware
        v. Send current read RPM to display
4. Start the task scheduler
5. Endless loop unless watchdog reset
    a. If watchdog reset, check based on switch, if so, restart system after 3 sec

Theory of Operation - Hardware

Hardware: HB3 Peripheral

The HB3 peripheral is a piece of custom IP that controls a DC motor and reads revolution data returning from that motor through a PMod -bridge. This peripheral utilizes an AXI interface to communicate with the microcontroller, and includes two custom modules. The first custom module is a PWM generator. The PWM generator takes as an input a value between 0 and 255. It will output an asserted signal for that value's number of clock cycles then de-assert for the remaining clock cycles for a period of 255 clocks. The second custom module, the tachometer, contains two counters. The first counter counts 100 million clock cycles, or one second for a system with a 100 MHz clock, before restarting. The second counter increments whenever the input signal it receives from the motor's encoder has a positive edge. This counter drives the output signal to the microcontroller whenever the one second counter resets, then it resets as well.

To interface with the HB3 we have created a series of driver functions. The tachometer has two driver functions. The first reads the AXI register that the tachometer module writes to and then returns that value. The second calls the first function then multiplies that value by 60 and divides by twelve to obtain the motors RPM. The PWM generator has three functions. The first reads the axi register that the microcontroller uses to drive the PWM generator module. The second pwm generator driver function updates the axi register with a new PWM value, without changing the direction. The third driver function changes the direction of the motor. This is achieved by first turning off the motor, then changing the direction, and then restoring the previous PWM value.

Hardware: Embsys

The embsys for this project contains:

- Microblaze microcontroller
- 544 Pmod Encoder
- Pmod OLEDrgb display
- The Nexys4IO
- Our Pmod H-Bridge AXI Peripheral custom IP
- AXI UartLite Peripheral

- AXI Timer
- AXI Watchdog Timer
- AXI Interrupt controller
- Two AXI GPIO peripherals
    - Output: Single-Port to drive LED
    - Input: Dual-Port to read pushbuttons and switches

Hardware Challenges:

- Parameter bug: When developing the custom peripherals I used parameters to indicate the number of clock cycles we wanted to pass before sending tachometer data to axi registers. I discovered that the time period I had chosen to count encoder sensor data was too small, so I needed to adjust this parameter. I ended up changing it in the highest level module where I included this parameter. This did not end up solving the problem because there was a higher level module where this change did not propagate. Vivado created a file where it auto generated a parameter for the lower level modules however did not update it when I updated the IP. I needed to hand change the xilinx generated file to get this problem resolved.
- PWM on/off bug: there was a bug in my PWM generator module that prevented us from setting the motor fully on or fully off. For fully-on there would be a 1 cycle clock pulse where the enable was deasserted, and for fully-off there would be a 1 cycle where the enable was asserted. The issue with fully off was especially important because we risk a short circuit in the H-Bridge if we cannot fully deassert enable when changing directions. This issue was found in verification and was fixed with minor adjustments to the RTL.
- 1/1000th of a second bug: When I initially planned my Tachometer module I wanted to sample the data every 1000th of a second. However at the max RPM of the motor 6200 that would mean there would be (6200/60/1000 *12) = 1.24 sensor pulses possible in a reading at the maximum speed. This leaves no room for any speed variation and is therefore unhelpful for our purposes. We ended up fixing this by sampling every second like was suggested in the project report.
- Make file bug: There is a known bug with custom axi peripheral ip in vivado 2020.2 where the makefile causes a failure when you try to build the project in vitis. I found a solution for this problem in a Xilinx community post, however Prof. Roy Kravitz shared a better fix to the makefile.

Discussion and Adjustments To Plan

The project had many minor issues that were corrected. One issue was incorrectly setting the WDT bits based on clocks. The software had issues with various priority settings not allowing other threads to operate at all as well as unresponsive hardware if split evenly. The PID controller was changed multiple times from sweep curve fitting to fitting any inertial load with a linear range assumption. The software also started without any FreeRTOS implemented to test functionality of each peripheral first in a traditional superloop.

One challenging aspect of the project is iteration from software back to hardware. There were small bugs only found during software testing that had to be updated in hardware. Some bugs were also searched for on the software side to be found in hardware later. Another challenge is the significant amount of various debugging and implementations. The team had no experience in watch dogs(or lecture material), PID Control and FreeRTOS which made it difficult to manually test each and implement together with the multiplied complexity. The examples given for these were very limited and some did not function with identical setup code in some cases such as the interrupt handler.

Another aspect is the rotary encoder. The PMOD ENC digilent driver has significant debounce issues. We had to spend time readjusting our hardware for the Roy's driver which eliminated many of the issues with some minor persistent issues.

Final Project Results/Discussion

The project functions correctly up to the limitations of the motor. Graphs were found which replicate the results near expected from the PID protocol with limitation of the encoder feedback rate.

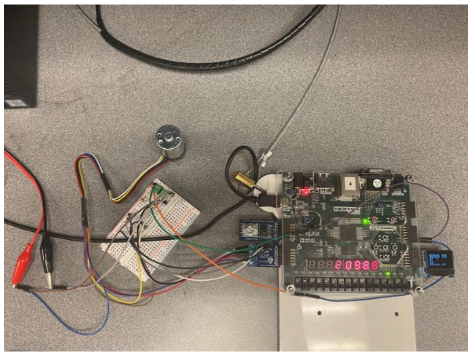


Figure [5]: Pmod physical port setup.

Deliverables below are included in the submission and a video link is provided.

Video Link Here

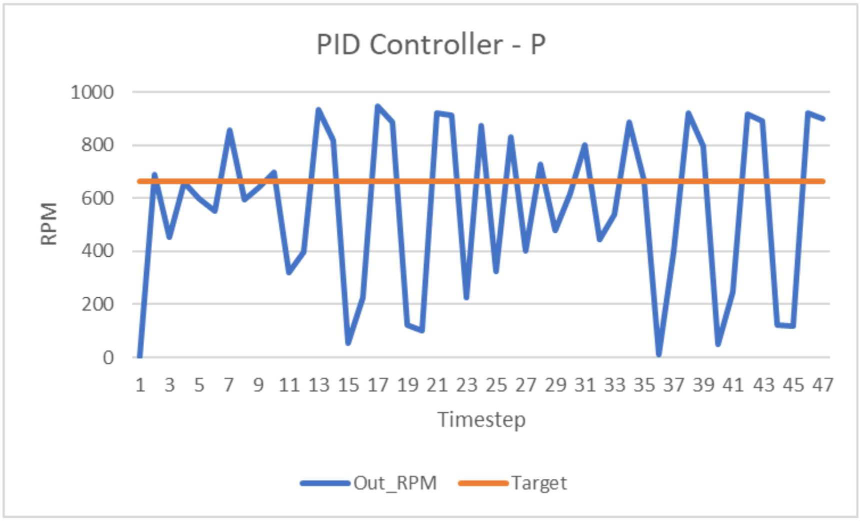


Figure [6]: P Controller Result. KP set to 1.

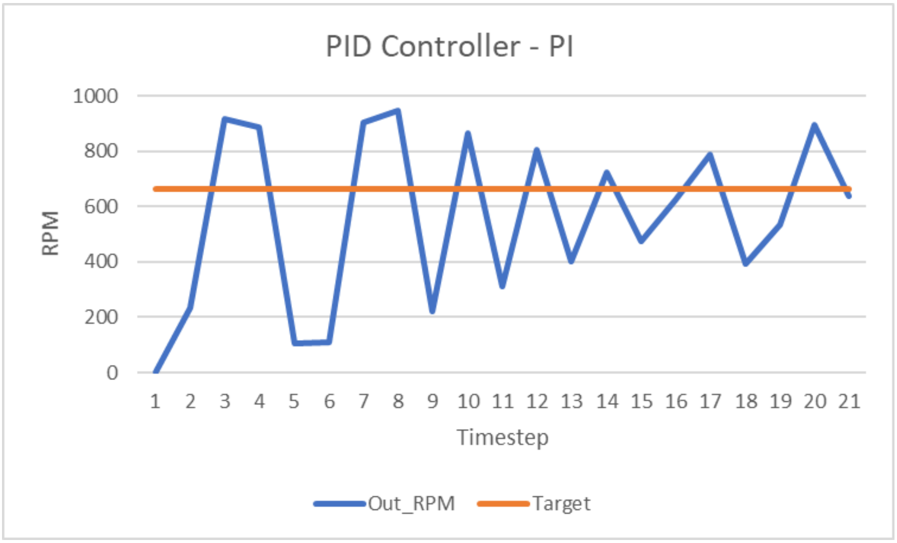


Figure [7]: PI Controller Result. KPI all set to 1.

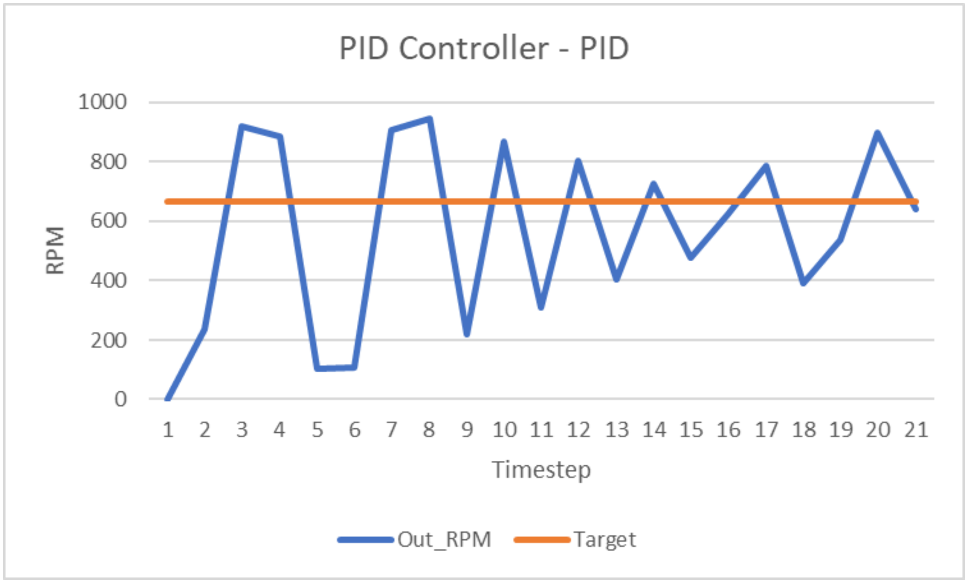


Figure [8]: PID tuned result. KPID all set to 1.

Deliverables

A demonstration video of your working project. Be sure to demonstrate the capabilities of your system by varying the control constants kp and ki.

A 5 to 7 page design report explaining the operation of your design, most notably your control algorithm and user interface.

A flowchart/state diagram is expected including such features as measurement, conversion to RPM, display, utilization of PID control math, conversion of PID loop output to PWM, etc. Please include a few "interesting" graphs showing the results from the control algorithm.

•Source code for your C application(s). Please take ownership of your application. We want to see your program structure and comments, not ours.

• All source files regarding your new motor control and speed measurement system (IP), including the SystemVerilog hardware, and driver code.

• Your constraint and top level Verilog files.

• A schematic for your embedded system. You can generate this from your block design by right clicking in the diagram pane and selecting Save as PDF File…

Appendix

Appendix A - C Code

See Project3.c file within Vitis->App->src directory