Learning Objectives:

- Gain experience with microprocessor-based closed loop control
- Gain experience with multi-tasking, synchronization, and inter-process communication (IPC) using FreeRTOS
- Apply the concepts learned in Project #1 to a more complex application
- Flex your hardware-building muscles w/ "real" circuits

This project will be done in teams of two

Project: Microprocessor-based Closed-loop Motor Control

This project is designed to give you practical experience with a common, and often crucial, embedded system function – closed loop control. Closed-loop control involves using feedback from one or more sensors to adjust the input parameters that control the output of the circuit. An example of a control system is automotive cruise control. The driver sets a target speed, engages the cruise control and takes his or her foot off the accelerator. The cruise control does its best to keep the car moving at the target speed (called the *setpoint*) even as the car goes up and down hills. The control circuit for this project is much simpler than cruise control but the same principles apply.

The Project #2 control system regulates the speed of a geared motor. The shaft of the motor is connected to a metal hub. You can apply friction to the hub with your finger to change the load on the motor. The closed loop control system responds to varying loads by adjusting the PWM duty cycle to the motor to maintain the speed. The speed and direction of rotation for the motor are set by a Digilent PmodHB3. The current speed of the motor is reported by the built-in quadrature encoder on the motor.

You can purchase the electronic and electromechanical components you need for this project online (Digi-Key and Pololu). The total cost of all parts is about $55 including the PmodHB3. The PmodHB3 can also be ordered directly from Digilent. The BOM for the project is included in the Project #2 release package.

Motor System

The electromechanical part of this project consists of a 6 VDC motor with an integrated quadrature encoder and a 9.7:1 gearbox. The motor is connected to a Digilent PmodHB3, a 2A H-Bridge capable of driving up to a 12V motor. The PmodHB3 inputs are EN (Enable) and DIR (direction). EN is driven by a PWM signal (0 – 3.3V) to control the speed of the motor. The larger the duty cycle, the higher the average voltage to the motor windings and the faster the motor turns. The DIR signal is driven by a GPIO to set the direction of rotation. The motor windings are best powered from an external power source instead of the Nexys A7. Normally, you could use one of the bench supplies in the lab , but this year has been far from "normal." You could do what others have done and sacrifice one of your old phone chargers by cutting off the connector and wiring the leads to the VM inputs on the PmodHB3 if you don't have happen to have access to a power supply. The ideal phone charger would provide 5VDC @ 1500 ma but the current requirement of the motor should be much less, except maybe for startup. Figure 1 shows the Control Circuit Schematic.

WARNING: Since a small mistake in your circuit can eastly damage the FPGA board and the H-Bridge, you should be careful when powering the DC Motor, PmodHB3 and the FPGA. Always double check all the connections and wires in your design. If you are using a lab-type power supply, make sure to set the right voltage values for the components, and always current limit the circuit.

The output of the encoder will pulse 12 times per revolution per changing edge per encoder (see here: https://www.pololu.com/product/4822 [scroll down to the section *`Using the Encoder`*]. The speed (RPM) of the motor is calculated by counting the number of pulses in a known interval. For example, counting the number of pulses in 1 second and dividing by 12 will give you revolutions per second. Multiply by 60 and you have revolutions per minute. You may find it advantageous to do some simple software filtering to get a "true" reading. For example, you could average several readings together and/or you could toss out "crazy" readings (ex: the motor speed suddenly increased from, let's say 500 RPMs to 10000 RPMs in a second or two).

There needs to be a load that can be varied to stress the control circuit. In this project that load is supplied via a metal hub attached to the drive shaft of the motor. Apply some pressure to the hub and the control program should increase the PWM until the speed reaches the setpoint again. Release the wheel and the program should decrease the PWM to reach the setpoint.

Color	Function
Red	motor power (connects to one motor terminal)
Black	motor power (connects to the other motor terminal)
Green	encoder GND
Blue	encoder Vcc (3.5 V to 20 V)
Yellow	encoder A output
White	encoder B output

The meaning of the wires of DC Motor and the Encoder

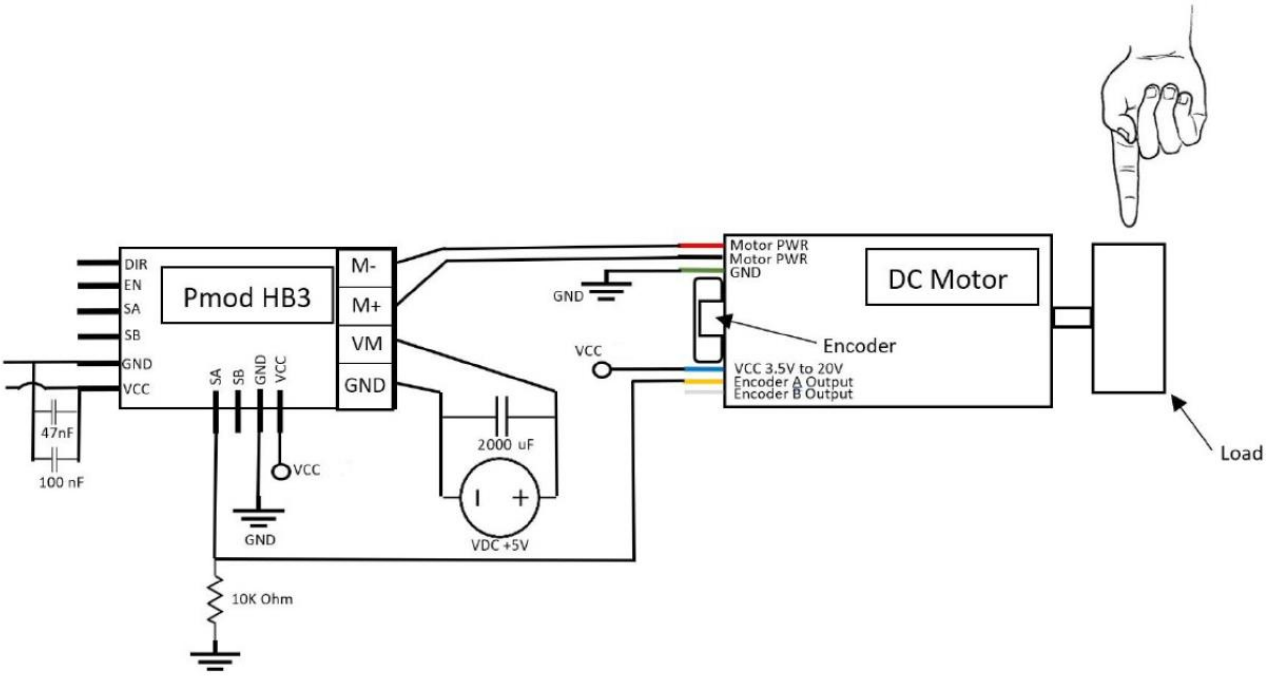

Figure 1. Control Circuit Schematic

Application

You will develop an application in C that implements a full PID controller for the drive motor. The application will display information about the system and produce the control signal for the drive motor. The user will tune the control circuit by using the BtnU (top button) and BtnD (bottom button) to slowly increase (BtnU) or decrease (BtnD) the PID control constants (K_p, K_i, or K_d). We recommend that you use switches 3 and 2 to select which parameter is being modified by the buttons. You can use 3 LEDs, one for each constant, to indicate which control constant you are modifying.

Nexys A7 Device Mapping

- *PmodOLEDrgb:* The OLED display should display the PID values and the actual motor speed. Further innovations are welcome.

- *PmodENC:*
  - Rotary Encoder – The rotary encoder knob is used to select the motor speed. Turning the knob clockwise increases the motor speed. Turning the knob counter clockwise decreases motor speed.

  - Switch – The switch on the PmodENC changes the direction of the motor. Looking at the PmodENC with the switch on the top, with the switch in the left position, the motor turns counterclockwise, with the switch in the right position, the motor turns clockwise. Make sure that the direction is not changed while the motor is running (i.e. set the speed to 0 RPM, change the direction, and then increase the speed).

- *Nexys A7 Slide Switches:*

  - Switches[15] – Can be used to force a crash (a WDT failure) in your application

  - Switches[14:6] – not assigned

  - Switches[5:4] – control the amount the selected control constant is incremented or decremented with each press of the BtnU or BtnD buttons. Pick whatever constants you need for your algorithm but one suggestion is to implement the following:
    - 00: If both switches are open, change the $K_p/K_i/K_d$ value by ±1 on each button press
    - 01: If Switch[4] is closed, change the $K_p/K_i/K_d$ value by ±5 on each button press
    - 1x: If Switch[5] is closed, regardless of Switch[4], change the $K_p/K_i/K_d$ value by ±10 on each button press.

  An alternative would be to implement an algorithm that increased the value/button press as a function of time. Say for the first 3 or 4 seconds the button is pressed the parameter changes by 1. On the 5th through 10th second without the button being released it changes by 5. After that it changes by 10. May digital clocks set the alarm time in a similar way. It's SMOP (a small matter of programming).

  - Switches[3:2] – Used to select which of the three PID parameters (K_p, K_i, K_d) to change.

  - Switches[1:0]
    - 00: If both switches are open, the Rotary Encoder changes the motor speed by ±1
    - 01: If Switch[0] is closed, change the motor speed by ±5
    - 1x: If Switch[1] is closed, regardless of Switch[0], change the motor speed by ±10

  An alternative would be to implement an algorithm that increased the setpoint by a larger increment if the rotary encoder is spun faster. I implemented this algorithm a number of years ago and it was not terribly hard to do so. SMOP!

- *Nexys A7 Pushbuttons:*
  - BtnC – sets desired motor speed to 0 and turns off the PWM signal to the motor. The control constants needs to be set to a non-zero value like 1.

  - BtnU – Increment the selected PID control constant

  - BtnD – Decrement the selected PID control constant

- *Nexys A7 LEDs:*
  - LED[15:3] – not assigned

  - LED[2:0] – Display which PID parameters are being used. For example PD control would light the P LED and the D LED. PID would light all 3 LEDs. A constant of 0 for a parameter would indicate that it is not being used in the control algorithm.

- *Seven Segment Display:*
  - [7:4] – (Nexys A7 only) Digits and decimal points should be off.
  - [3:0] – Display the desired RPM (as set by the encoder).

Calculating the motor speed and control constants

An important design decision is how to calculate the desired and actual motor speed and the control constants K_p, K_i, K_d. The motor speed can be set by an 8-bit unsigned (0-255) PWM signal. That 8-bit value should scale over the entire motor speed range as much as possible. You will need convert the pulses from the integrated quadrature encoder into a comparable speed measure that corresponds to the PWM value. For example, if the maximum motor speed is 6000 RPM (PWM = 255) then the half-speed (PWM = 127) should be 3000 RPM.

The control constants, say K_p, can also be represented by an 8-bit quantity. What happens when you multiply the "error" signal (the difference between desired and actual speed) by K_p? Multiplying two 8-bit numbers gives you a 16-bit number. To convert back to 8-bit, you have to decide how many bits to throw out to get an appropriate resulting value.

Since the error signal can be positive (motor is turning slower than the setpoint) or negative (motor is turning faster than the setpoint) it is possible for the sum of the desired speed and (K_p * error) to overflow or underflow. Since the command to the motor can only be positive (i.e. you can't set PWM to a value < 0) you should "clamp" the PWM value to 0 (min) or 255 (max). You also need to be careful of how to *handle maximum and minimum speeds with respect to the error signal. Ultimately, the magnitude of the* error signal will determine how you scale your control and speed representation. You will have to experiment with what is the best magnitude of the correction signal, error, and K_p.

The maximum value of 255 is only a suggestion. Feel free to experiment with other values. You may want to consider increasing the precision of the motor speed changes by adding more bits to the PWM duty cycle (ex: 0-511) register. It is possible that the speed will not be linear over the entire PWM range. (i.e. the motor may not start moving until the duty cycle is 20%).

Hardware

Control Circuit

The Project #2 release package contains a Bill of Materials (the BOM) for the suggested control circuit. Building the circuitry is straightforward. Simply connect the various pins of the motor connector to the PmodHB3. See here for the pinout of the motor https://www.pololu.com/product/4822 and the PmodHB3 (https://store.digilentinc.com/pmod-hb3-h-bridge-driver-with-feedback-inputs/).

The Tachometer

The outputs of the integrated quadrature encoder are used to determine the speed of the motor. You could re-purpose your hardware pulse-width detection from Project 1 as a tachometer, but we propose an alternate approach for calculating RPM (**R**evolutions **p**er **M**inute) that may be better suited for this application.

The speed (RPM) of the motor is calculated by counting the number of pulses in a known interval. The higher the rotational speed of the motor, the closer together the pulses will be. In effect you are determining an unknown frequency for a known interval (seconds or minutes). As we discussed earlier in the term, a way to do that is to count pulses for a known amount of time.

The output of the encoder will pulse 12 times per revolution per changing edge per encoder. With that in mind is should be easy to implement a SystemVerilog module that detects a rising or falling edge on the sensor and counts the number of edges in a 1 second interval. Dividing that count by 12 will give you revolutions per second. Multiply by 60 and you have revolutions per minute. That count can be returned to the Microblaze through an I/O register or a GPIO port. You could also calculate RPM in the hardware. The Series 7 FPGA on the Nexys A7 contains a number of hardware multiply blocks that can perform fast multiplications. Clocking the edge detection logic with the 100MHz AXI clock should provide plenty of margin for detecting a rising edge on the encoder signal(s).

You may be able to use an AXI timer in capture mode to perform a similar function or you could use the PWM_Analyzer IP provided by Digilent (`digilentip-library-master\ip\PWM_Analyzer_1.0`) to implement the functionality in your custom peripheral.

The motor for this circuit is intended to robotic use and has an integrated gear box (decreases the speed of the motor while increasing the torque). Even though the external driveshaft rotates at a slower speed than the internal driveshaft, the encoder is connected to the internal driveshaft. Therefore, the speed read by the encoder will not match the rotational speed that you see on the external driveshaft. Adjust your calculations for displaying the motor speed appropriately.

Embedded System Configuration

You can create the embedded system the same way you did for Project #1 by creating the block diagram in the IP Integrator and modifying the top level module and constraints file to accommodate the changes.

The resulting embedded system should have this minimum configuration. You can add additional hardware as you see fit:

- Microblaze, mdm, etc. Configure hardware floating point in the Microblaze. That should keep the program size reasonable and provide fast computation for the control loop.
- Local (BRAM) memory of 128KB for program/data memory.
- Digilent PmodEnc and PmodOLEDrgb IP: Provide access to the PmodENC and PmodOLEDrgb. You can use the same drivers that you used in Project #1.
- Nexys4IO: Same as for Project #1
- PmodHB3 Custom Driver: Created by you.

- UartLite peripheral: Used to send the signals to PC to be plotted on a graph.
- AXI Timer/counter: Used to generate the "systick" for FreeRTOS. Should be configured with both timers because FreeRTOS uses Timer 0 in the AXI Timer/counter as the default "systick" generator and Timer 1 in the AXI Timer/Counter to provide performance statistics. It's easiest to dedicate your AXI Timer/counter 0 peripheral to FreeRTOS (that is the default). Use additional AXI Timer/counters if you must perform other timer-based tasks in your implementation.
- AXI Timebase/Watchdog Timer. Configure the Watchdog timer interval to about 2 or 3 seconds lest you get tired of waiting for disaster to strike.
- AXI Interrupt Controller with the Watchdog timer interrupt and Timer 0 (systick) interrupt.

- (optional if you choose to use the recommended tasking model and/or you want to run the FreeRTOS starter program we provided) An AXI GPIO configured with one 16-bit output port for the LEDs. If you'd rather use Nexys4 IO to control the LEDs you will have to make changes to the starter program.
- (optional if you choose to use the recommended tasking model and/or you want to run the FreeRTOS starter program we provided) A second AXI GPIO configured with two ports. One port (input only) for the pushbuttons and the second port for the switches. This AXI GPIO peripheral should be configured with interrupts and added as a third interrupt source to the interrupt controller. You may also choose to connect the switches and buttons inputs to Nexys4IO, as well, if you want to use the Nexys4IO driver and API. The starter program and tasking model are based on the buttons and switches being interrupt-driven and Nexys4 IO does not have that capability.

  Hint: You can use an xlconcat block to combine the individual pushbuttons into a bus that can be connected to the GPIO input in your block diagram. Alternatively you can bring both the LEDs GPIO and the buttons/switches GPIO to the top level and connect them there (much the same way that you may have connected your Hardware PW Detect module output to a GPIO.

Embedded System Drivers and Link Map

Your Project #2 application will use the drivers and FreeRTOS provided by Xilinx. The following additional drivers and files should be included in your software platform:

- PmodEnc and PmodOLEDrgb, and Nexys4IO drivers as used in Project #1.
- Your driver for the PmodHB3 peripheral.

A few brief words on using FreeRTOS:

- The application will be targeted to FreeRTOS rather than the Standalone OS we used for Project #1.
- The system can run on a FreeRTOS kernel on the Nexys A7. While creating a new platform in VITIS you will have to select FreeRTOS rather than the Standalone OS when you create the board support package and use the FreeRTOS API to develop your application.
- You will need to configure FreeRTOS to use a smaller heap (the default is 32K) or your application may not fit in 128KB of BRAM.

- We have included a Getting Started FreeRTOS app in the project release. The application uses two Tasks and one Queue, One interrupt generated GPIO switch, one GPIO LED to blink an LED (how do you spell “overkill”) using a GPIO interrupt.

The project release includes a possible tasking model for the project. You can implement that model or invent one of your own. If you modify the tasking model be sure to describe and document it in your design report.

Project 2 Tasks Summary

Download the Project #2 release package

Download the Project #2 release package from the course website.

Select a partner and join a Project #2 team

You will work in teams of two for Projects #2. We have set D2L up for self-enrollment. We will assign teams for any students who have not enrolled in a team within a few days of project assignment. To self-enroll in a group:

- One member of your team should claim an unused group (Group #) and add himself/herself to the group
- Same person should email, text, or whatever the other member(s) of the team with the group number
- Other member(s) of the team should self-enroll in the same group
- When a group is full we will rename the group to match the team members

While there are many ways to split the work on this project one suggestion is that one partner creates the embedded system and builds the control circuit hardware while the other partner creates the control application and peripheral. While you may collaborate with other teams, all of the work you submit must be your own and those you collaborated with should be named in your report.

Mount the motor, build the control circuit and connect it to your FPGA development board.

The Project #2 release package includes the schematic and BOM for the control circuit, including the motor. You may be able to use the PWM and pulse-width detection circuitry from Project #1 to test your control circuit by connecting the PWM output from your Project 1 to the motor. This allows you to start operating the motor without control feedback. Make sure that the direction is set to a constant logic level before the PWM output is applied to the EN signal of the PmodHB3. In fact, you can characterize the control circuit by sweeping the PWM value from 0 to 255 and observing the output of the encoder on an oscilloscope or logic analyzer. Doing so will provide the minimum and maximum speed of the motor and the offset needed to start the motor rotating. Adding some delay before changing the duty cycle will give the circuit a chance to settle.

Create your PmodHB3 custom peripheral

This three-part lectures should give you the idea on how to create custom IP.

- Part 1 - https://www.youtube.com/watch?v=BEQXV3eAZNs
- Part 2 - https://www.youtube.com/watch?v=tDmu_FnZuGs
- Part 3 - https://www.youtube.com/watch?v=qgnTkZ-InK8

There are several ways to create your custom PmodHB3 IP block:

- (Recommended) Write SystemVerilog code to implement the PWM and tachometer functionality.
- Use the Digilent PWM generation and analysis IP blocks in your custom peripheral. Since these peripherals come with software drivers and connect to the AXI bus like any other IP block and you want a single peripheral you may have to extract parts of the IP blocks and leave the redundant AXI-bus related functionality out of the peripheral.
- Include an AXI Timer instance and the `pwm_tmrctr` library for generating the PWM signal needed to control the speed of the drive motor and a GPIO to control the DIR signal of the PmodHB3 H-Bridge. Create a new RPM detector or modify your hardware pulse-width detect circuit from Project #1.
- Combinations of the above. We are not overly concerned about how you generate the PmodHB3 peripheral, just whether it is configured as a custom IP block and meets the needs of the project.

Build the embedded system and top-level module for the project

Build your embedded system using the IP Integrator and instantiate it in your top-level module. Connect the embedded system ports to the top-level ports and make any necessary changes to the constraints file and the top-level module generated by the IP Integrator. Your embedded system should include all the peripherals your application needs, including your custom peripheral. Synthesize, implement, generate bitstream and export your hardware to the VITIS.

Implement the control loop software and interface

This application is more complex than the application in the first project. You may implement the user interface as described in this write-up or you can innovate and develop a user interface of your own design. The important thing is to implement the PID control algorithm in a way that can be quickly and effectively demonstrated and that gives you a platform to experiment with different closed loop control algorithms and control constants.

Integrate and characterize the motor system

Once you have built the control circuit and at least a partial application (perhaps with the Standalone OS) you are ready to integrate the system and characterize the circuit if you have access to test equipment to measure RPM. Two effective methods to characterize the circuit are:

- Sweep the duty cycle to the drive motor from 0 to 99% and measure the RPM's. Insert a small delay between duty cycle changes to allow the circuit to settle. You can use the data points you collect to establish a relationship between RPM and duty cycle – a necessary step towards putting RPM's and duty cycle into the same frame of reference.
- Apply a step (0% to 99%) function to the drive motor and record the RPM change over time. Data from this exercise will help establish the min and max motor speeds and the average rate of change of the motor speed.

Complete and integrate the control application and tune the control system

As part of your experimentation you should determine the value of $K_p/K_i/K_d$ that gets the motor speed up to the setpoint quickly and then stabilizes on, or as near to, the setpoint as is possible. Try different control algorithms (P, PD, PI, PID, etc.) and constants and graph the most promising results.

Add the watchdog timer functionality to your application

The Xilinx AXI Timebase Watchdog timer peripheral serves as both a timebase (a simple counter like the FIT timer) and a watchdog timer (WDT). A requirement of this project is to include watchdog timer functionality in the application. The application and watchdog timer should operate with the following characteristics:

- The application should provide some kind of recovery action. This could be as simple as displaying a message and waiting in an infinite loop for a reset or a power cycle, or it could include provisions to restart the application. Your application can use the `XWdtTb_IsWdtExpired()` API function to determine whether the reset was caused by a WDT timeout or a system reset.

- The application should have the ability to force a WDT crash. This is most easily accomplished by preventing the calls to the `XWdtTb_RestartWdt()` from occurring when the "Force Crash" switch is on. You can use the leftmost slide switch (`SW[15]`) for the "Force Crash" switch.

The application should check that it is, in fact, running rationally. A simple way to do this is to periodically set a "system running" flag in the master thread and to restart the WDT every time through the main loop. A more elegant way is to use a WDT interrupt handler to read and clear the flag and restart the WDT whenever the handler is triggered (typically the first WDT overflow). The WDT IP block will force a CPU reset when it expires the second time (the first generates an interrupt but not a reset). To do provide hardware support for this connect `WDT_Reset` output from the WDT to the `Aux_Reset_In` input on the `proc_sys_reset` block.

Demonstrate your project and submit the deliverables

Once you have your project working prepare a video demo of your project. We will be offering "live" (via Zoom) demo opportunities if you would like to show off your handiwork. Please include a video demo of your project even if you do a "live" demo. Submit your deliverables to the D2L dropbox in a single .zip or .rar file of the form <yournames>_proj2.zip. Only one submission per team is necessary. We will grade the submission with the latest timestamp if there are more than one.

We would like you to use, and we recommend that you use, GitHub and GitHub classroom for this assignment so make a final push of all your deliverables to your GitHub repository for the project when you are finished. Teams in former classes have almost uniformly approved and recommended GitHub as a way to keep team-oriented projects up to date…even more critical during the pandemic and the PSU campus closure because it is difficult to get together in person to work on a project

Deliverables

- A demonstration video of your working project. Be sure to demonstrate the capabilities of your system by varying the control constants and trying different closed loop control methods.
- A five to seven (5 to 7) page design report explaining the operation of your design, most notably your control algorithm and user interface. Please include a few "interesting" graphs showing the results from the control algorithm. List the work done by each team member. Be sure to note any work that you borrowed from somebody else.
- 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…*

Grading

You will be graded on the following:

- 60 pts - The functionality of your demo. Part of this will be how well your demo works, it must meet the spec. Also, we would like you to take initiative and add some bells and whistles. We will look at other "quality" of design issues such as unnecessary display flicker, slow response of the system to buttons and control changes.
- 20 pts - The quality of your design expressed in your C and SystemVerilog source code. Please comment your code to help us understand how it works. The better we understand our code the better grade we can give you.
- 20 pts - The quality of your project report.

Extra Credit Opportunities

Project #2 offers several opportunities to earn extra credit points. Here are some suggestions but we are willing to be amazed and amused:

- Innovate on the user interface –you want to be able to easily "tinker" with the control loop
- Enhance the tachometer functionality to be able to detect direction of rotation in addition to the speed.

References

[1] *Digilent Nexys A7 Board Reference Manual*. Copyright Digilent, Inc.

[2] *Digilent PmodHB3 H-Bridge Motor Control Reference Manual*. Copyright Digilent, Inc.

[3] *Digilent PmodENC™ Reference Manual*. Copyright Digilent, Inc.

[4]*Nexys4IO and PmodEnc Product Guides and Driver User Guides* by Roy Kravitz, 2014

[5]*Getting Started in ECE 544 (Vivado/Nexys A7)* by Roy Kravitz, et. al.

[6]*Digilent PmodOLEDrgb* User Manual (optional)

[7] YouTube videos

Revision History

Rev 1.0	20-Feb-2017 DH	Major Revision to Project #2. This project does closed loop Proportional control to maintain the speed of a small electric motor. Many of the concepts (closed loop control, create/package custom IP, etc.) are borrowed from the previous project concept but the application is different.
Rev 1.1	28-Apr-2017 RK	Changed pulse width detect from high/low count to counting rising edges for a one second interval. This algorithm is better for using a tach pulse from a Hall Effect sensor to calculate RPM. Also changed KP factor selection to provide greater precision using fixed point. Organization, typographical and writing style changes.
Rev 1.2	04-Feb-2018 RK	Major revision to Project #2. Project supports full PID controller and is targeted to FreeRTOS
Rev 1.3	28-Apr-2018 STB, RK	Minor revisions to simplify project.
Rev 2.0	31-Aug-2018 JAG, RK	Major revision to Project #2. Replaced the motor with a geared motor with integrated encoder. Updated the documentation and release package.
Rev 2.1	23-Apr-2019 ME	Updated the documentation. Updated links, BOM, and Datasheets. Added a new control circuit schematic.
Rev 2.2	26-April-2020 YL	Modify spec so it can be implemented in both Nexys A7 and Basys 3. Major revision to Project #2. Updated the control circuit schematic.

Rev 2.3	29-April-2020 RK	Removed references to using Digilent Board files and the Getting Started with FreeRTOS document.
Rev 2.4	04-May-2021 RK	Removed references to Basys3. Updated for Vivado/VITIS. Minor editorial changes.