Maseeh College of Engineering and Computer Science

PORTLAND STATE UNIVERSITY

Winter 2023

ECE544 - Project 2 Design Report

Stephen Weeks, Drew Seidel, Noah Page
Email addresses: dseidel@pdx.edu, stweeks@pdx.edu, nopage@pdx.edu

Maseeh College of Engineering and Computer Science

PORTLAND STATE UNIVERSITY

Introduction

Design Structure

User Interface Control Algorithm and Display

PID Controller Algorithm

Graphing and UART interface

HB3 Custom Hardware and Drivers

Graphs

Challenges

Conclusion

Work Division

Introduction

For this project we were learning and implementing a PID controller to handle a DC brushless motor. The project had three major required components: user interface, display output, and PID control to the DC motor. The design also included a watchdog timer that could be caused to crash with a switch placed on a rotary encoder device as well as a fixed interval timer (FIT) that gave a heart beat on the 7-segment display and was used for when to do UART data transfers for live graphing.

Here we will give a quick overview of functionality, a more detailed description will be given in the design section. For user input there are three ways to interact with the system: push buttons, switches, and a rotary encoder that also has its own push button and switch. The switches provide a way for the user to input the control algorithm they want to use, the step size for the control constants (kp, ki, kd) respectively, and the step size for the encoder RPM selection. The push buttons allow us to toggle between the run or set state, increase or decrease the control constant values, and to select which constant value to choose from. Because we decided to do live graphing, no user interface is required for updating the csv file. The display functionality either shows the control constants or setpoint and current RPM readings depending on if it is in set mode or run mode respectively. When in set mode, it also shows the control algorithm being run (open loop, P, PI, PID, etc) depending on the switch configuration. The encoder functionality allows us to choose the set point by running the washer either clockwise or counterclockwise, reset the PWM to 0 and control constants if the push button is pressed, or crash the watchdog timer if the switch is flipped. Finally, a custom hardware module was designed to run the motor from the HB3 PMOD that is controlled functionally in software and defined by the user input.

The goal of this project was to learn PID control and implement it. To use a watchdog timer to protect against errors or run faults (we chose an interrupt handler approach). To provide an elegant user interface with a mixture of a “super loop” and interrupt capabilities, and to graph PID data from the UART console port.

Design Structure

Hardware

For the design structure we ended up using the following modules/IP: a UARTLite, Nexys4IO, Fixed Interval Timer (FIT), Watchdog Timer (wdt), our own custom HB3 module (full-bridge rectifier module from Digilent), and a Digilent rotary encoder (code named PmodENC544). The majority of the IP in the system was provided by either Roy Kravitz or directly from Xilinx. For our own custom system Noah built our HB3 custom hardware. It consisted of two custom modules, a PWM generation model for the enable pin and a tachometer module that counts ticks every quarter of a second. This allows the drivers to do the mathematical calculations for the RPM for our system. Because we have a gear encoder and our system counts 11 ticks per rotation (per second) and a ~75:1 gear ratio we are able to get ticks per second and then convert for the user by the following equation: ticks * (60/823.13). This truncates the calculation for us and returns it as a uint32_t so there is a variable error of ~2 RPM depending, though this wasn't fully characterized. The user is also able to write to the control register to set the PWM including enabling/disabling the output (similar to project 1). This should allow fast shutoff for future implementations when the user could change the direction of the motor. The other major components for this project were the rotary encoder and watchdog timer. Because we are not using the wdt in window mode the count is set by the width of the timer register that is configured in hardware. We went 29 bits wide which roughly equals 5 seconds of delay. We mostly chose this because of a race condition introduced in the interrupt loop by Stephen before Noah debugged it and found it. It could be set to a quicker time if chosen. The rest of the configuration of the wdt is set up in software and will be discussed later in this section. Finally we have the pmod encoder which is used to set the speed, reset values, and as a kill switch for the system. We opted in using Roy Kravitz's hardware implementation as it provided debouncing to the system already. One key area we ran into was that the washers need to be reversed to their output pins to the FPGA to provide an increased reading when turning clockwise, otherwise setup was very straightforward.

The overall hardware design flow can be seen in the embsys pdf doc that is provided alongside this report. The main hub of the system is the Microblaze processor with its instruction and data cache alongside an interrupt controller. The interrupt controller provided a two based width interrupt connection for the wdt and FIT (the latter was used to provide a heartbeat and UART data processing). The rest of the system: encoder, Nexys4IO, myHB3ip, and UARTlite were all connected through the AXI bus hub to the microblaze. The FIT timer was originally set at 4 Hz tick but was later changed to 2 Hz for better visual processing.

Software

The software design is broken down into a few major components: user input processing, PID value update (based on user input), PID control loop, software display, and finally UART updates to the python graphing software. Below we will give an overall view of each component and then cover them in more detail in their own sub categories.

The user input is provided in a set of struct data. The original plan for this was to do a greater divorce of concerns between the PID structure and user input structure. Due to some errors in processing the PID structure had to be abandoned for the time being (will be tried again in the final project). The user input consumes the user input data and updates if its previous values have changed, if so it sets a flag in the structure to alert the PID update loop that the system state has changed and processing is required. This allows for a clean handoff in case we would like to separate out modules down the road or use a message queue in FreeRTOS.

For the PID update the functional purpose is to change corresponding constants and values based on user input to the switches, pushbuttons, or encoder functions. This software component also consumes the user input structure and checks if the user input state has changed, if it hasn't it exits to save processing cycles. If it has changed the function breaks down into different states, if the switches have changed it updates the LEDS[6:0] with the new state information and updates the k-constant step values based on switches[2:0], the PID functionality chosen (is this PID control loop, PI loop, etc) based on switches[6:5] and the setpoint step value from switches[4:3]. It updates to the UART port what values have been chosen and their corresponding value. It then checks if the button states have changed, if they have we loop through the different possible states and update the values: if btnR is pushed we loop to through the k-constant choice FSM, if btnL is pushed we toggle data stream mode of the motor through the UART, if btnD was pressed we decrease the currently highlighted state by its set step size, if btnU is pressed we increase the currently highlighted state by its set step size, and finally btnC was pressed we change between set mode and run mode. The process then checks if the encoder updates, if the encoder was turned clockwise the set point is increased by its step value and if it is counterclockwise it decreases. If the pushbutton on the encoder has been pressed then the values are reset to a known value (kp = 1, kd = ki = 0) and then the motor is turned off and the setpoint set to 0 RPM. If the kill switch was flipped on the encoder a variable is set that shuts down the wdt and tells it to roll-over to failure mode. In the final step it sets the user input structure to false so we know it hasn't changed.

The most important function in the system is the PID control functionality. First it checks if the setpoint is set to off mode, if so it merely sets the system to off, otherwise it runs the control loop. In the control loop we set the duty cycle from the setpoint provided by the user. It then converts that to the desired RPM for later display. We used the control loop procedure provided from "PID without a PhD" and added in our own fudge factor for the offset and clamps to make sure that the gain terms are not too large. We also clamp the max value you can write to our duty cycle (1023) so the system can only be driven at 100% DC. The error calculation and integral terms were made to be static so we can maintain state from reading to reading for our measurements.

The display functionality is rather straightforward, it checks if we are in set mode or in run mode. When in set mode we display the k-constants onto the 7-segment display along with the dp(6,3,0) of which one of the three constants is being selected to be modified. If in run mode the system displays the current RPM on displays 5 and 4 and the chosen setpoint is displayed on 1 and 0.

The final major functionality of the system is sending the measured data along with the set parameters over the UART interface to be consumed by the python script. Every second a variable is set by the FIT timer, and if we have enabled the btnL then we will send the new data. We push out the values being used (so if in PD we would only send the kp and kd set values and not ki) along with the set point and current RPM readings of the motor. This is then consumed by a python script that live updates a graph of the current running state.

Two interrupts were used when designing the software: a FIT and wdt. The FIT interrupt blinks dp7 and increments a counter for the UART to send data every second (the FIT timer interrupts every 0.5 seconds). The wdt interrupt resets the timer unless the switch is flipped or another error has occurred, if so it then enters a goodbye state before resetting the system. These were helpful functions in designing the system and allowing for troubleshooting.

That describes the overall main functionality of the embedded system software. It covers all major functionality used for the project and how user input is consumed and translated to motor control.

System Overview

The system overview will provide a quick guide to the working functionality of the system from the users perspective. When a user starts the system, it will always start in set mode. The switches will be set to values that will cause a new read (unless for some reason the user has turned on all the switches). It will then parse through the state and set the new switch values and print them to the LED. The user can control the PID functionality via switches [2:0], the setpoint step size via switches [4:3] and the k-constant step size with switches [6:5]. These can be modified in either set mode or run mode by the user. The user can then use the right button to select between which constant to increase or decrease via the up and down buttons respectively. This will be displayed on the 7 segment display where the kp constant is displayed on segments 7 and 6, the ki constant is displayed on segments 4 and 3, and finally the kd constant is displayed on segments 1 and 0. When switching the right button the current constant to be selected and modified is given by the dp value highlighting on the lower display corresponding to the constant value chosen. If the user wants to reset values, they can push the button knob on the rotary encoder module. Once the user is happy with their settings they press the center button to switch over to run mode. From here they can turn the rotary dial clockwise to increase the set point in RPM, or they may want to decrease it by turning it counterclockwise. In run mode the system will try to maintain this value and will update it on the upper 7-segment display of the current measured RPM, where the lower 7-segment display will show the selected setpoint for the system. Dp 7 will blink on and off at a 2 Hz speed as a system heartbeat allowing the user to visually see the system is healthy. The 16th (top most) LED will turn on when the watchdog timer first rolls over, giving a visual cue that the system is guarded. Finally, if the user would like, they can flip the switch on the encoder causing a wdt crash that will print bye bye across the 7-segment display and turn on all the LEDs before resetting the system. At any time when running the system, if the user has configured and setup the python script, they can press the left button to start live graphing to display a graph plot and update a .csv file. They can also deactivate the script in the same manner.

User Interface Control Algorithm and Display

For coherency reasons, it was important to modularly break up the user interface into three sections: user input, system update, and system display. For user input a C structure approach was chosen. The structure would poll if there had been any change in the current state of the buttons or switches on the Nexys A7 board through the Nexys4IO drivers. The previous states were maintained by the structure itself and would be updated with the new states as well as a flag that indicated that the system state had been modified. The struct was also responsible for reading and maintaining the state of the PMOD Encoder, using the PmodENC544 drivers to update if the encoder rotary count or switch/button states had changed. It would follow the same procedure of updating and then setting an update flag to true if things had changed.

The system would then hand off this gpio struct to the system update interface. The system update had to maintain its own state of the previous buttons so it can more granularly sift through the changes. This design approach was chosen because we want to be updating state changes to the controller as quickly as possible with little overhead. The worst case scenario is that all possible user IO has changed and needs to be updated. The system will then go through and check what actually has been changed: first it checks the switches and if they have changed it either updates the appropriate step sizes and or the new controller algorithm. It then will check which button is pressed. This algorithm should be optimized in the future to only handle one button press but at its current state checks each possible state. If a button has been pressed it then applies the appropriate logic: if up was pressed it increments the current constant that is selected and if down is pressed it decrements the current constant that is selected, all by the selected step size. If right is pressed it cycles through the software FSM to select the next constant. If the center is pressed it flips the set/run mode flag. And if the left button is pressed it toggles the live graphing constant. Finally, the system will check the encoder, first to see if the rotary count has changed and apply the appropriate step algorithm based on the change. It then checks if the encoder button was pressed, if so it resets the constants all to zero except for kp which it sets to 1 and sets the motor to off. Finally, it checks if the switch has been flipped, if so it updates a global constant that is consumed by the wdt API interrupt handler to set it in kill mode. The algorithm finishes by resetting the gpio structure change value flag so that the system will not update until the next polled change.

The final process of the user interface was updating the 7-segment displays to give user feedback to the system state changes. These changes displayed differed depending on if the system was in run mode or set mode. If the user had configured the system in set mode then it displays the three constants across the 7-segment display along with the corresponding dp value to the constant being changed. When in run mode the upper 7-segment display would show the measured RPM from the custom HB3 hardware drivers and the lower would show the setpoint that is controlled and configured by the encoder device.

PID Controller Algorithm

To configure the PID controller, one of the main challenges was performing a duty cycle to RPM characterization for our motor. Since the custom IP delivered us a value in rpm, we needed a way to characterize, and potentially limit our duty cycle range to enhance performance and predictability.

To do this, the current plotter that we have included was modified, and used to plot set duty cycle versus RPM, running in an open loop format, without interference. The captured relationship between the set duty cycle, and the set RPM is the following:

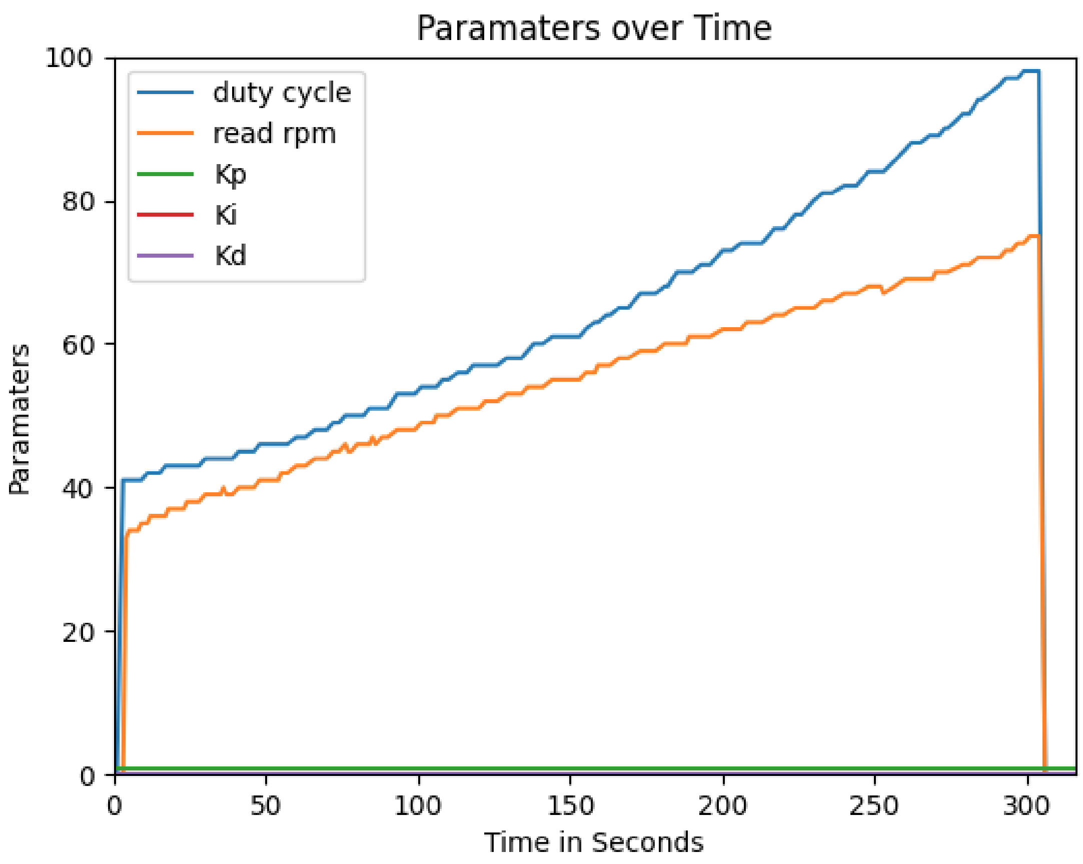
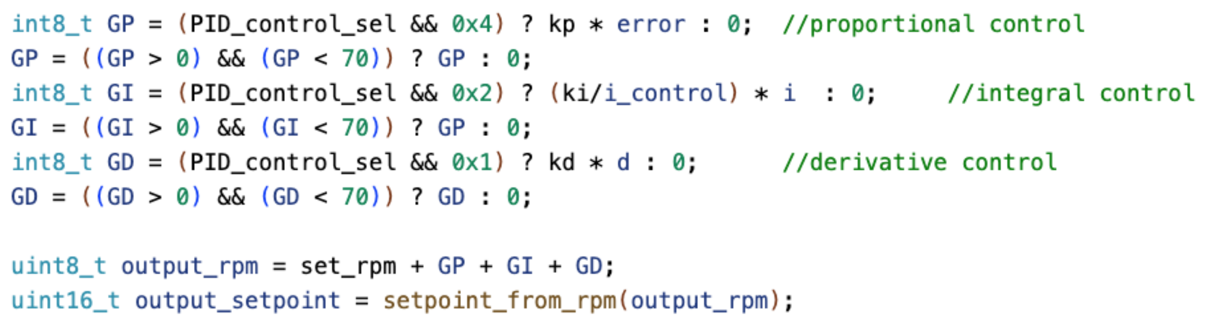


Figure 1: Duty cycle to rpm characterization

In the above graph where duty cycle (as a percentage) and RPM, we derive a linear relationship between the range of roughly 45% to to 60% duty cycle. Beyond this, it is apparent the effect on the motor of a step in the duty cycle decreases.

With this in mind, functions could now be created such that the PID control algorithm could all be done in RPM. As the project specification had quite large limits on PID control k constants, could have used a number of tuning methods (as much of this depends on the environment, motor selection, etc.)

We decided not modify the P, I, and D constants to another range, but instead, note that for the most part the upper ranges are not necessary, and that the lower constants as presented in literature provide enough closed loop control.



```
int8_t GP = (PID_control_sel && 0x4) ? kp * error : 0;  //proportional control
GP = ((GP > 0) && (GP < 70)) ? GP : 0;
int8_t GI = (PID_control_sel && 0x2) ? (ki/i_control) * i  : 0;     //integral control
GI = ((GI > 0) && (GI < 70)) ? GP : 0;
int8_t GD = (PID_control_sel && 0x1) ? kd * d : 0;      //derivative control
GD = ((GD > 0) && (GD < 70)) ? GD : 0;

uint8_t output_rpm = set_rpm + GP + GI + GD;
uint16_t output_setpoint = setpoint_from_rpm(output_rpm);
```

Figure 2: Updating output from PID constants

With tuning via testing, we found this relationship to give desirable results in the noted ranges (k, i, and p lower than 10), even in high pressure conditions.

Graphing and UART interface

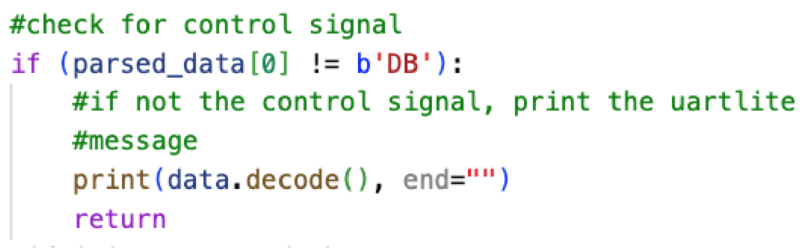
In order to effectively utilize the uartlite module regardless of whether stdin and stdout was directed at the mdm module, or uartlite, the uartlite was initialized, and the uartlite functions were customized and used directly.

With the uartlite set up, the design choice was that graphing could work when other system messages were printing (whether they were printed to the same uartlite instance or not), so a one way parsing control protocol was utilized to interface between the output data and the python script.

The specific implementation of works in the following way:

1. Button left will turn on or off (the opposite of the previous state) printing parameters (set rpm, read rpm, Kp, Ki, Kd) to the console.
2. The python script is running and monitoring the USB port
3. Python script parses lines that begin with the control message: ‘DB’
4. Python script live parses, graphs, and updates CSV file

This allows the graph to be “live stopped” and “live started” by pressing the left button on the Nexys A7 while the script is running.



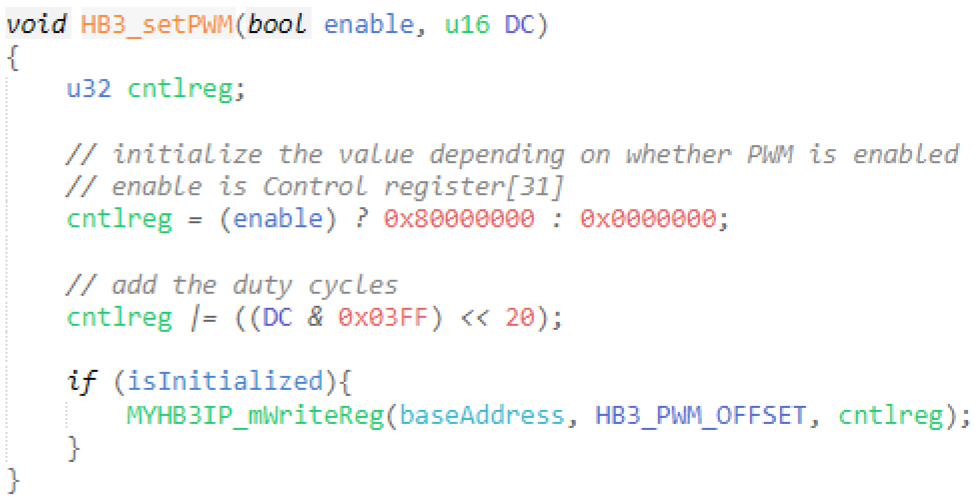
```
#check for control signal
if (parsed_data[0] != b'DB'):
    #if not the control signal, print the uartlite
    #message
    print(data.decode(), end="")
    return
```

Figure 3: Checking for control signal in python parser

HB3 Custom Hardware and Drivers

One of our initial approaches to the project was to use the rgbPWM_r2 and PWM_Analyzer IP from the first project to test that we could set the motor to a specific duty cycle and read the duty cycle back to display. Once we were confident the system worked, we extracted the parts we needed and implemented them in our custom AXI4 Peripheral IP by using the Vivado Create and Package New IP Wizard and following along with the provided YouTube videos.

There are 2 verilog modules used in the design, HB3 and ticker. The HB3 module is used to set the duty cycle using 10 bits with an output frequency of 2KHz. The PmodHB3 Reference Manual said to use a “high enough frequency” and their graph had 2KHz, so we used a DIVIDE_COUNT of 49, and MAX_COUNT of 1024 parameters to achieve our 2KHz output based on the AXI 100MHz clock. The design uses the slv_reg0 as the control register that a user can write to in order to set the duty cycle, where bit 31 is the enable pin and bits 29:20 are the 10 bit value where the output would be in a high state. The Driver then supplies a HB3_setPWM function that uses a boolean enable and u16 DC input to write to the control register.

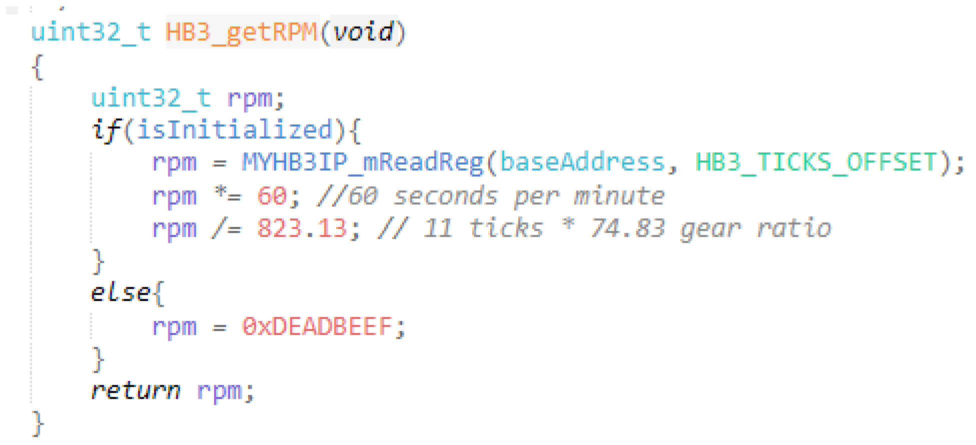


```
void HB3_setPWM(bool enable, u16 DC)
{
    u32 cntlreg;

    // initialize the value depending on whether PWM is enabled
    // enable is Control register[31]
    cntlreg = (enable) ? 0x80000000 : 0x0000000;

    // add the duty cycles
    cntlreg |= ((DC & 0x03FF) << 20);

    if (isInitialized){
        MYHB3IP_mWriteReg(baseAddress, HB3_PWM_OFFSET, cntlreg);
    }
}
```

Figure 4: HB3_setPWM

The ticker module counts the number of positive edges from the motor’s A phase encoder within a 0.25s window, using the 100 MHz AXI clock as input. This value is then multiplied by 4 to give an approximation for ticks/second. An initial design only updated every second, but was changed so that our control loop would have a faster response. The output of ticks per second is at the location of slv_reg1 and the Driver provides 2 functions that will read this register, HB3_getTicks and HB3_getRPM.

The HB3_getTicks returns the number of ticks per second and was provided for testing and debugging. The HB3_getRPM returns the RPM of the motor, a value that is updated every 0.25s and is used by the PID Control Loop. The function reads the ticker register to get the ticks/second, multiplies by 60 to get ticks/minute, and then divides by 823.13 to get the RPM of the outer shaft of the motor based on the motor's 74.83:1 gear ratio with 11 ticks per revolution of the internal motor shaft. The truncated integer value will then be returned to the caller.



```
uint32_t HB3_getRPM(void)
{
    uint32_t rpm;
    if(isInitialized){
        rpm = MYHB3IP_mReadReg(baseAddress, HB3_TICKS_OFFSET);
        rpm *= 60; //60 seconds per minute
        rpm /= 823.13; // 11 ticks * 74.83 gear ratio
    }
    else{
        rpm = 0xDEADBEEF;
    }
    return rpm;
}
```

Figure 5: HB3_getRPM

Graphs

The performance of our PID controller was evaluated over the course of the project. For a complete history of testing, see the logger directory.

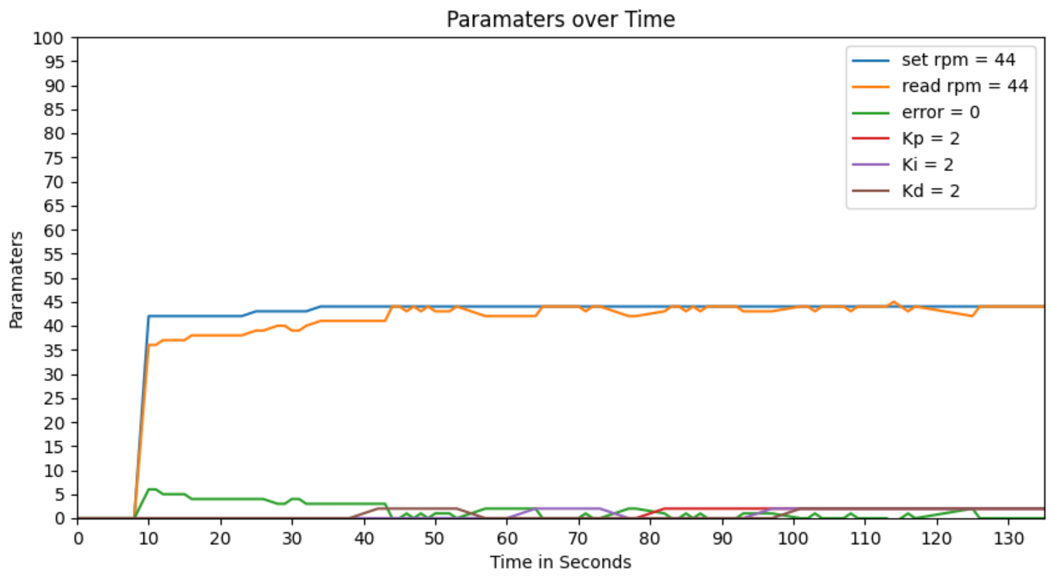


Figure 6: Low pressure testing (early version test)

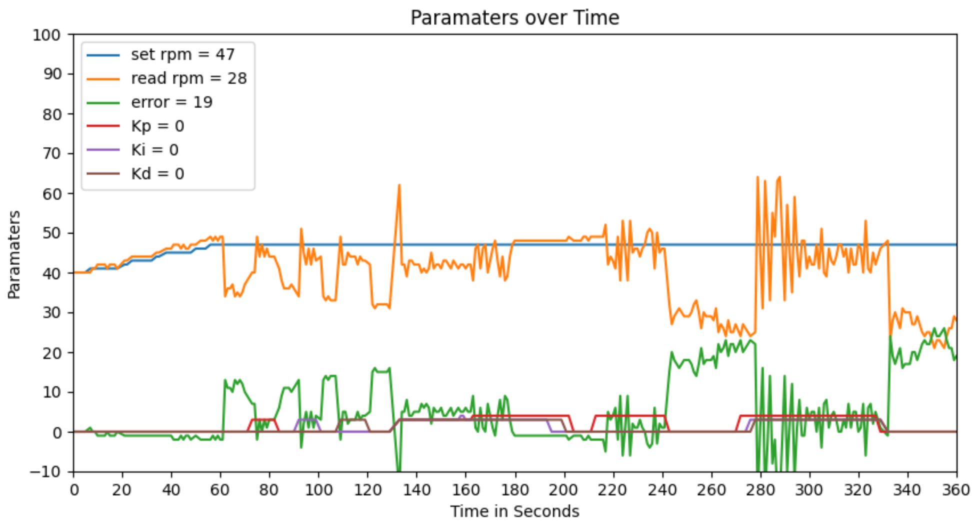


Figure 7: High pressure test (Version 1.0)

Watchdog Timer

The watchdog timer configuration was fairly straightforward. In the system initialization function, the watchdog timer was configured and set up in time based run mode, the timer was then stopped. The handler was registered and configured with the interrupt controller, before being activated. Once the interrupt was activated, the watchdog timer was re-started. Once in enable mode, the interrupt handler would be triggered. On the first trigger the system state initializes by turning on the 16th LED as a visual indicator to the user of the watchdog timer in active mode. The watchdog timer then would restart the timer if the kill switch on the encoder is not thrown. If thrown, and the timer has expired, the system would enter into a state where it prints “bye bye” on the 7-segment display and lights up all the LEDs before the system reset is triggered through hardware configuration.

Challenges

Stephen:
I faced two major challenges when setting up the system software for the user interface and watchdog timer. For the user interface the biggest challenge was weird behavior when using two structures. Originally, I wanted the main loop to contain all state data. So we would have two structures, a GPIO and PID. The controller module would then consume these two and update as appropriate. However, for some reason, my PID design would lead to very weird and non-deterministic behavior of my user interface. Things would update that should be updating in the button loops and being displayed incorrectly. Therefore, I ended up scrapping the PID struct (at the time I thought the project had a more limited timeline) and just made PID states internal to the module. Looking back and doing some reading, I think I had a stack clash with the fact

that my structures weren't packed and were pointers. So possible boundary issues. The other challenge faced for me was the wdt. We had a weird random crash that would happen undeteremenistically. Turns out, Noah Page found this and solved it, it was because I created a main condition by doing polling in the main loop. Because of this the system might not have updated when interrupted. Therefore, we moved all handling directly into the interrupt handler itself.

Drew:
I faced the following challenges:

While it was tempting to use "xil_printf" with uartlite, I saw a fatal design flaw in doing it this way; that is that if a user wanted to debug the system with xil_printf messages printed to mdm, than that means that if we were to use xil_printf to also send the logger data, then the plotter would be rendered useless.

This is an edge case scenario, but the uartlite should *always* receive the logger data, regardless of whether stdout is the uartlite, or MDM. This means having to format the data correctly, in a parssable way with uartlite functionality, and devising a system for the python parser to only look at the correct messages. This was done using a control signal that is outlined in the Graphing and UART Interface section.

Second, the duty cycle to RPM characterization and PID control algorithm tuning posed an interesting challenge that required a few simple design approaches that produced a quality performing overall system. Had we not clipped the PID controller to a predictable and linear range, the solution would not have been as trivial, and the performance may not have yielded the same quality results. Having the live plotter altered for a duty cycle characterization test was beneficial for this design consideration.

Noah:
When I wrote the custom HB3 AXI4 peripheral, I wasn't sure how to get the automatically generated selftest to work with my ticker's read register location. I modified the AXI.v file in the way the video series showed, and I looked at other IP we had used, but couldn't find a good enough example of how to change the verilog wrapper so that it would pass the selftest. Since I was trying to get the IP out to the group in a timely manner, I ended up just commenting out the R/W block of the selftest as a quick solution.

The wdt seemingly random crashes was something I had noticed early on and although the project would have been presentable, I really wanted to figure out what was going on. I first thought it was an issue with the FIT interrupt since the decimal point heartbeat seemed to have some inconsistent behavior. But after wasting a lot of time trying to remove the FIT timer while preserving the UART graphing functionality, I realized that even without the FIT timer it continued to crash. So I finally started looking into the wdt itself and found that the polling in the main loop was the cause and it was very satisfying to finally figure that out.

Another issue I spent a lot of time trying to figure out was the duty cycle of the motor would give inconsistent speeds. I was first using a cheap variable power supply and noticed inconsistent voltage readings, so I thought that by switching to a spliced USB cable with a wall adapter would be more consistent. But even with that I have seen the RPM output vary by quite a bit even within our linear range we selected. I'm not sure if it is because my motor hasn't been unplugged and that maybe the continuous power has affected its response, but I guess it will remain a mystery to me.

Conclusion

In the end, we ended up creating a PID controller system that can select different controller algorithms and constants to control a feedback loop to a motor. We learned a lot from our design and were able to solve some very interesting problems. The team did a good job of setting up and delegating details. Thanks to Drew putting in extra effort on making tcl script to allow us to easily re-build hardware and pass it back and forth we were able to quickly and easily prototype hardware without difficulties of hand offs. Software development was seamless as the team used asynchronous communication effectively. The goals were met within spec and on time.

Work Division

Stephen Weeks - User input software, display software, FIT and watchdog timer interrupts. Software debug prototype design

Drew Seidel - Vivado revisioning setup, PID control loop, uartlite, live plotter/csv writer, testing and debugging

Noah Page - Initial motor setup, integrate PmodENC into input software, develop HB3 custom IP and integrate into design, testing and debugging software design