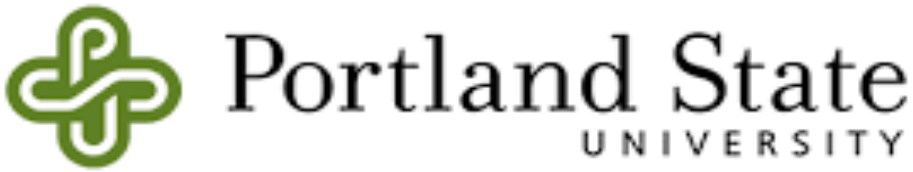


ECE – 544

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Embedded System Design

Project-2 Closed loop Control

Robert Holt
holtrob@pdx.edu

Deepen Parmar
parmar@pdx.edu

Date: 20th May 2020

Table of Contents

Overview

This project explores the idea of closed loop control on a PWM controlled plant motor with encoder feedback that can be transformed into a rotational velocity term. This project explores methods for measuring motor speed in a matter that has a sufficiently high sampling rate (samples faster than the inertial time constants of the motor itself) while attempting to minimize noise in the measurement of the motor speed. The motor is controlled and measured via an integrated Vivado IP which encapsulates the functionality and allows for simple integration in a Vivado block diagram.

Control of the motor is performed by a PID control algorithm with gains tuned empirically according to the well known Ziegler-Nichols method. Specifically, a simple Python application was created which utilized the serial interface to read in CSV formatted time series data realtime to visualize the oscillations and characterize the terms required by the Ziegler-Nichols method. The PID controller steps at 50Hz, which in retrospect is likely too fast as speed updates are only occuring at 16Hz. Regardless, reasonable performance seems to have been achieved.

The display features fixed point display of the PID gain values on the PmodOLEDrgb, which was required due to the range of the PID gains (less than 1.0 for all gains, and much less than 1.0 for a reasonable tuning of Kd). Additionally actual motor speed is displayed on both the upper seven segments (NexysA7 board) and on the PmodOLEDrgb. Three LEDs indicate which gains are in use in the PID controller (non-zero).

Several DIP switches are used to control the scaling of the PID gains and motor set point. The rotary encoder drives the change in set point of the motor speed according to the position of the corresponding DIP switches. Similarly, the up and down pushbuttons control the change in the selected gain by the selected amount according to the position of the corresponding DIP switches. The center pushbutton sets the motor setpoint to zero and quickly sets the motor to a 0% dutycycle (bypassing the PID control).

Finally, a watchdog timer monitors the state of the individual tasks and assures that each task has run at least once during the watchdog interval, thus verifying that no tasks have crashed or are being starved by the scheduler. The watchdog is set to interrupt on a 1.34s interval, so after 2.6s has elapsed since the last successful check, the system will reset.

Motor Interfacing

Control Interface

Motor control is performed via an 8bit PWM signal from the application through the custom created drivers which accompany the IP. Hardware for implementing PWM is fairly simple, with an 8-bit counter running on a 100MHz clock. On each counter clock edge, the counter is checked to see if it is less than the 8bit duty cycle which is being requested for the motor, if it is, the output remains high, otherwise the output goes low. It was decided that the clock should be quite fast to remove any possibility of the PWM edges "bleeding through" to the observed motor speed. A PWM period too low would result in observable oscillation of the motor speed.
Additionally, the IP was implemented as an AXI peripheral, so it made sense to use the 100MHz clock that drives the AXI interface as well. The customer peripheral has 2 implemented registers, with BADDR + 0 being the register for setting the duty cycle. This is stored in the lowest byte of the register.

In testing it was found that an 8-bit PWM is probably not a high enough resolution for the duty cycle. Frequently very minor oscillation can be seen as the PID controller bounces back and forth between two consecutive duty cycle values (i.e. 134/255 is too low and 135/255 is too high). This kind of hunting by the PID controller was an indication that a 9 or 10-bit PWM would have improved the stability of the system.

Tachometer

The motor controller hardware also needed a method for estimating the speed of the motor with a reasonable update rate and low noise. The shorter the sampling interval, the fewer the counts that can be expected from the motor, and the higher the chance that noise or boundary conditions in the encoder could be amplified into high error in the measurement. This can be illustrated in the extreme where the system wants to measure the motor speed at 1000Hz. In this case and at lower speeds, such at 2000rpm on the input shaft, a 12 ticks per rotation encoder would only count 0.4 ticks per millisecond. This means a motor speed of zero would be read for two consecutive cycles followed by a motor speed of 5000rpm. This is an extreme example, but the issue persists to a lesser degree even in more realistic sampling intervals (less than 1000Hz).

Conversely, the longer the sampling interval, the slower the controller has to be, and the longer the response time. A very accurate motor speed reading can be achieved by counting ticks for 1s before generating a new speed. This however would mean that either the controller needs to run at 1s intervals, which is a much longer time than the inertial time constants of the motor thus leading to an unfortunately slow response, or the controller runs with old feedback values. This

would lead to integrator windup problems as the controller believes that it needs to try harder and harder to close the error, when in reality the motor is responding, but the measurement is too infrequent.

Our solution was effectively to filter the measurement in hardware. This was implemented by sampling over a 1s timeframe by summing the results of 16 sub-counters each counting for a 1/16s duration. Effectively this meant that we updated the motor speed at 16Hz, but only one counter had new counts from the last 1/16th of a second, with the remaining counts coming from the previous 15/16th second.

In hardware, this was accomplished by a counter that ran for 1/16 of a second, then incremented a selector such that the “buffer” counter wrote it's value to the previously selected subcounter, then restarted. In essence, whenever the selector changed, this triggered the buffer to write into the next subcounter, effectively creating a rolling buffer of values. The actual count representing the motor speed was determined by summing all 16 counters to achieve a total count in the last second metric.

Driver Concepts

The hardware only measures the encoder speed as determined by the input shaft to the gearbox. This means that the speed that the user cares about (the output shaft speed) is not readily available to them if they do a simple register read. Additionally, the measured speed is only available in rotations per second, while many people think in terms of rotations per minute. These types of gaps are well addressed by driver firmware which provides a layer of abstraction that makes the hardware easier to use. The following snippet indicates the interfaces that the driver exposes in it’s header file (initialization function not shown):

```
extern XStatus MotorEncCont_set_direction(bool cw);

extern void MotorEncCont_set_dutycycle(u8 dc);
extern void MotorEncCont_set_dutycycle_pct(u8 dc_pct);

extern u16 MotorEncCont_get_rps_raw();
extern u16  MotorEncCont_get_rpm_raw();

extern u16 MotorEncCont_get_rps();
extern u16 MotorEncCont_get_rpm();

extern void MotorEncCont_set_output_ratio(float ratio);
```

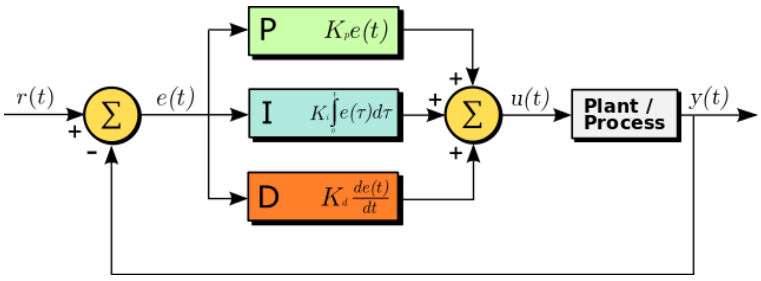
This interface provides an API which allows the duty cycle to be described in a [0-255] scaled range, or in an integer 0-100% format. Additionally, it allows the raw speed, in RPM or RPS to be acquired. If the application sets the output ratio via MotorEncCont_set_output_ratio, the

non-raw calls to get speed will give the result on the output shaft according to the provided ratio. This was a very convenient way to create the application, as the actual application only required that the ratio be set once (by passing 9.71 as the argument), and all further calls were already in reference to the output shaft. The set direction function meant that it was simple to change the direction, and the function enforced a safe change in direction by failing if the motor speed was not low enough. This indicates to the application that the direction will not change and that it needs to manually set the speed to zero and wait before requesting the direction change.

PID Control

Speed Controller Design Concepts

The rationale for a PID controller in this application is that an open loop control, where the user states they want a certain RPM which is then passed through a lookup table or other model is insufficient when considering load on the motor. Without closed loop control, the software has to presume a load on the motor, and thus fails inevitably whenever that load changes (i.e. a finger is applied to the collar on the output shaft). Since the motor has a method of feeding back it's actual state, it becomes possible to enable closed loop control. One simple method of closed loop control is the PID controller. The model for PID control can be seen below:



By Arturo Urquizo - http://commons.wikimedia.org/wiki/File:PID.svg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=17633925

In the above image an input, r(t) represents the target for the controller, which has the same units at y(t), the measurement feedback. When subtracting the output from the process with the set point r(t), and error is determined. This error is the foundation of the algorithm and is then used in between 1 and 3 different terms to determine the input to the process. In our case the setpoint has units of RPM, the output of the process also has units of RPM. Therefore the error has units of RPM. The process input (u(t)) has units of % duty cycle, so the gain Kp effectively has units of % duty cycle / RPM, or just 1/RPM. This Kp gain intuitively is the factor by which is multiplied to the error in order to get the dutycycle.

A P only controller is insufficient in our case where maiming speed requires some control effort. When operating in the ideal case, where error is near zero, the P term will generate 0% duty cycle, which then would reduce the motorspeed and causes the error to increase. In effect, a P

only controller will suffer from steady state error when the process input, u(t) is not a delta (or change in) term. If for instance the motor accepted a change in duty cycle at it's input, a P term could be sufficient, but this is not the case in our implementation.

The I term serves to reduce the aforementioned steady state error. The I term has a gain, Ki, multiplied by an integration of the error over time. This means as the error persists over time, the I controller tries harder to ramp up the u(t) such that eventually, when zero error is achieved, the P term generates a value of 0, but the I term is generating the exact duty cycle that is necessary to hold the motor at that speed. A PI controller functions quite well in this application. The effect of this PI controller which compensates for steady state error is that the motor never truely needs to be characterized to map duty cycle to RPM, rather the only empirical tests that need to be performed are in determining if the range of desired RPM values can be achieved with the limits of the process input (duty cycle), u(t).

The D term serves to dampen the response of the PI controller to minimize the overshoot of the setpoint and thus increase the overall stability. When a large error occurs, the P and the I portion of the controller are attempting to get to the setpoint basically as quickly as possible. It is the job of the D term to incorporate the fact that the error is closing quickly and to dial back the response. For instance, if at the start of the control, the error is 200rpm (SP is 200 and measured speed is 0), the PI will increase the value of the duty cycle. In the next step, the D term sees that the error is now 180rpm, and thus determines that the change in error over time is negative (180rpm - 200rpm in 20ms). This negative quantity is then summed with the contributions of the PI controller to effectively reduce the duty cycle they would have intended. As the rate of change in error approaches zero, the D term goes to zero.

The discussed controller is implemented in pid_cont.c and pid_cont.h within our project and allows interfaces to change the gains on the fly or to enable/disable P, I, or D portions of the algorithm. Additionally, a simple “perform_step” function is implemented so that the main application has a very clean interface to get the next duty cycle, requiring only the setpoint and measured values. Finally, integrator windup restriction was put into place to account for the fact that the user can easily request setpoints which are far out of range of our motor, even at 100% duty cycle with no load. In this case, it does not make sense to continually increase the I term, as this means when the setpoint does come down, it can be a very long time with negative error until the integrator “unwinds” and the controller functions properly. Effectively this means if the duty cycle is at its maximum value, then the integration term should not be increased.

Gain Tuning

Our PID controller was tuned empirically via the renowned Ziegler-Nichols method. In this method an empirical approach is took to tuning the system and two values Ku, and Tu are found. The gains Kp, Ki, and Kd are then derived based on simple multiplication rules according to Ziegler and Nichols. The following table shows how Ku and Tu are related to the ideal gains:

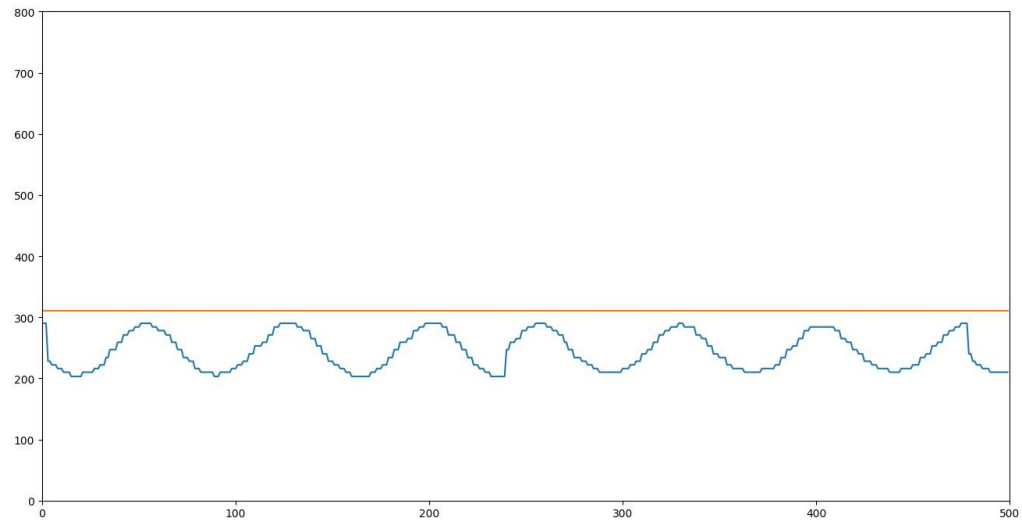
Ziegler–Nichols method[1]

Control Type	K_p	T_i	T_d	K_i	K_d
P	$0.5K_u$	–	–	–	–
PI	$0.45K_u$	$T_u/1.2$	–	$0.54K_u/T_u$	–
PD	$0.8K_u$	–	$T_u/8$	–	$K_uT_u/10$
classic PID[2]	$0.6K_u$	$T_u/2$	$T_u/8$	$1.2K_u/T_u$	$3K_uT_u/40$
Pessen Integral Rule[2]	$7K_u/10$	$2T_u/5$	$3T_u/20$	$1.75K_u/T_u$	$21K_uT_u/200$
some overshoot[2]	$K_u/3$	$T_u/2$	$T_u/3$	$0.666K_u/T_u$	$K_uT_u/9$
no overshoot[2]	$K_u/5$	$T_u/2$	$T_u/3$	$(2/5)K_u/T_u$	$K_uT_u/15$

Ziegler, J.G & Nichols, N. B. (1942). "Optimum settings for automatic controllers" (PDF). Transactions of the ASME. **64**: 759–768.

Ku represents the threshold gain and can be viewed as the value of Kp in a P only controller where oscillations begin to occur continually. For our application, this occured at a gain of Ku = Kp = 0.514. Tu represents the period of the oscillation, which in our application was 1.4s. From here, the ideal values of Kp, Ki, and Kd were determined using the “classic PID” control line in the table above. This meant that we set the initial values in software to Kp = 0.3084, Ki = 0.4406, and Kd = 0.054.

The actual process of tuning was done by printing CSV formatted data to the console, which was then read and monitored realtime via plots in Python. The python code which was used for the tuning utilized the serial port with the pySerial package, and plotted using the FuncAnimation functionality within Matplotlib. An example of the response can be seen below:



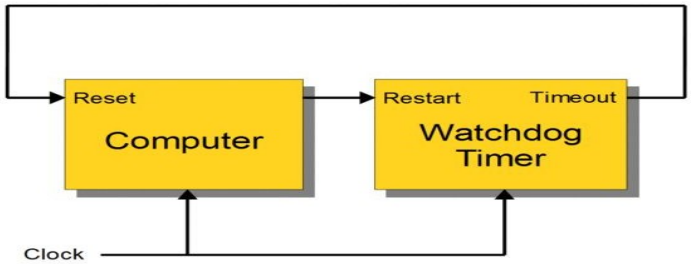
Determining the ultimate gain, Ku, and ultimate time Tu

The above plot shows RPM on the Y axis, and sample number on the X axis. Samples are approximately 20ms apart. Note that at Ku = Kp = 0.514 gain, oscillations can be seen with a

period of around 70 samples (Tu = 1.4s). Please see Appendix 1 for more images of tuned controller response.

Watchdog Timer Functionality

A watchdog timer is a timer that is used to detect and recover from any unexpected behavior in the application. During normal operation, the program regularly resets the watchdog timer to prevent it timing out. If, due to program error, the program got stuck somewhere then it fails to reset the watchdog, the timer will elapse and generate a signal. This signal is used to restart the system to get out of the unexpected behavior and restart the system. To implement the watchdog functionality in our project we need to add the functionality in both Hardware and the software.



https://en.wikipedia.org/wiki/Watchdog_timer

Hardware

In Hardware we included the Watchdog timer IP which is 32-bit peripheral that provides 32-bit free running timebase and watchdog timer. This IP was connected to the microblaze via the AXI Interrupt controller which generates the interrupt whenever the WDT Overflow occurs.

Software

In software we implemented the Watchdog Interrupt Handler which was called every time the interrupt occurs the occurrence of the interrupt was decided by an interval which was configured using the API.

Then in the Handler it checks the state of the sw[15] which was used as an indicator to force crash the application. If the sw[15] is in ON state then the application will stick into the while loop which then restarts the system. Also we set the Flag in each of the three tasks to check the functionality of it and if either of the flags is not set then force crash is initiated else the WatchDog timer will be restart

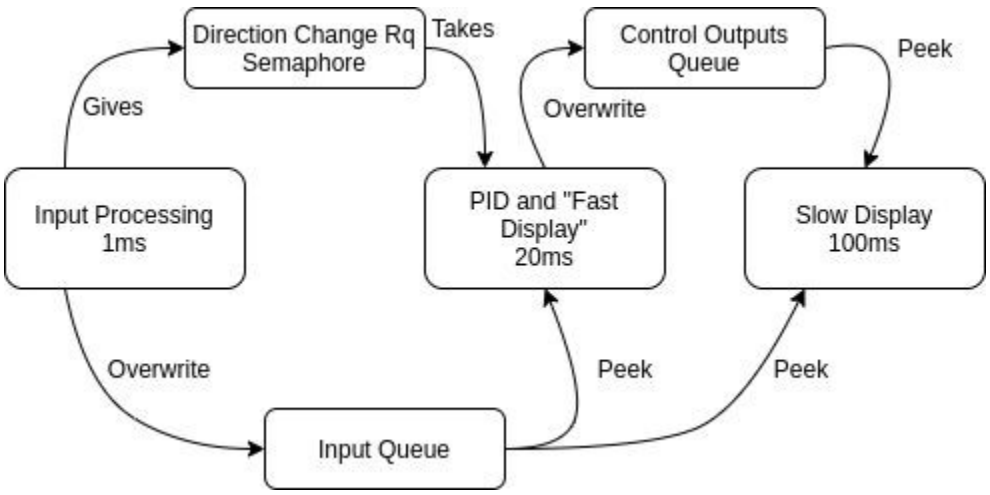
RTOS Architecture

Task Model

The tasking model was largely interval based, with a pattern whereby tasks, after completing their work, were delayed until the next 1, 20, or 100ms interval. The rationale for this was that with the current rotary encoder hardware and driver, the rotary encoder needs to be polled very frequently as it cannot generate interrupts and does not catch ticks at all in hardware. In order to get a good response from the rotary encoder and thus to change the motor speed, it needed to run very quickly (1ms interval). This problem could have also been solved more elegantly by using better hardware and driver IP in place of the mediocre one provided by Digilent.

A 20ms task was created for the PID controller because of the desire to have a fixed time interval between consecutive steps of the controller. The contributions from the PID controller's integral and derivative term rely on a known time step, and instead of determining the step whenever the task is called asynchronously from a semaphore or other synchronizing element, it was decided just to have it run at a medium priority with a known interval.
A third task was created at a 100ms interval for updating the PmodOLEDrgb. The PmodOLEDrgb can take a significant time to complete it's write, so it was decided to make this a somewhat infrequent task so that it did not jeopardize the update of the rotary encoder (thus motor speed setpoint) or the time step assumptions of the PID controller.

Effectively the input task ran at 1ms intervals and provided data via a single element queue to the 20ms and 100ms tasks. In turn the 20ms task took information provided by the 1ms task (rpm setpoint, rpm measured, signal to kill the motor) and performed the PID step, setting the motor duty cycle and then outputting data to a different single element queue. Additionally the 20ms task set the "low latency" displays such as the 7segment corresponding to measured and set rotation speed as well as the "gains in use" LEDs. The 100ms task ran the least frequently and at the lowest priority. It required peeking at both queues in order to set the display for the current value of the gains (provided by 20ms task) and the measured speed (provided by 1ms task).



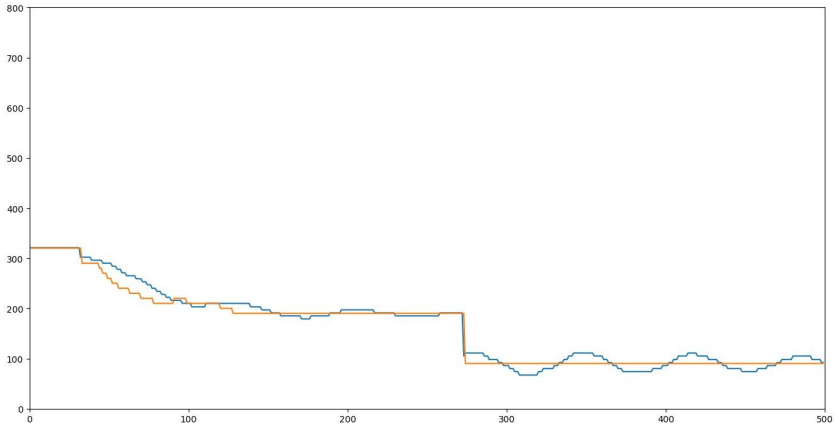
Having a 1ms task meant that the tick rate needed to be significantly increased from the 100 ticks per second default in the Xilinx SDK. With a 10,000 ticks per second rate, every 10th interval is devoted to the input task, while the remaining ticks are devoted to the 20ms, and 100ms task.

Data Exchange Concepts

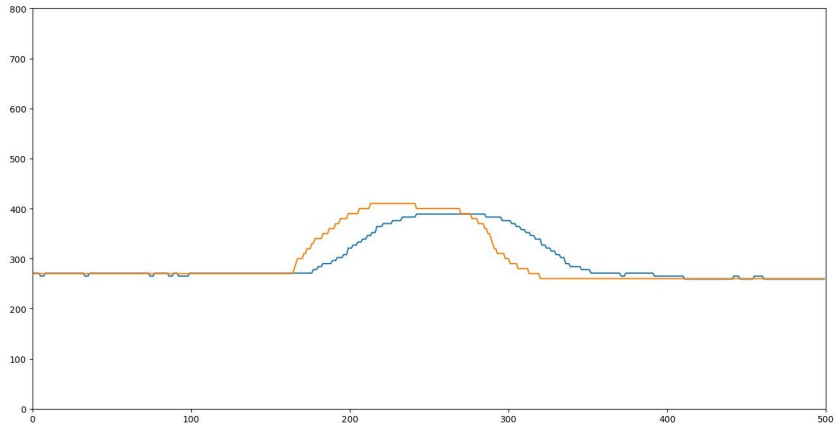
Data is exchanged throughout the application using several concepts including:

- Single element queues
    - Used to provide data from higher frequency tasks to lower frequency tasks. Effective for situations where only the last known state matters.
    - The 1ms tasks provides the measured speed, request to kill the motor, and the set speed (determined from the rotary encoder). Only 1 out of 20 updates will be read by the 20ms task, but this PID control task doesn't need to know what the setpoint was 5ms ago, it only cares about the current value.
    - Tasks putting elements in queue always overrides the single element
    - Tasks received from the queue always peek the element (such that multiple tasks can use the last known state whenever they run)
- Motor Directional Change Request Semaphore
    - Given by the 1ms input handling to indicate that a request has come in from the rotary encoder to change the motor direction
    - Taken (non-blocking attempt every 20ms) by the PID controller to ignore the controller temporarily in order to set the speed to zero, delay task, change direction, then begin controlling again.
- Global boolean flags
    - Used by individual tasks to indicate to the WDT handler that they are being called at least somewhat frequently and have not had an exception.
    - Flags are set false by the WDT after verification and are expected to be set again by the time the WDT interrupt occurs again.

Appendix 1 - PID Response Images



Controller response shows oscillation at speeds below 100, probably due to inertial issues and very low duty cycles



Controller response lagging aggressive setpoint change by approximately 300ms

Typical step up shows slight overshoot in favor of reaching the setpoint quickly. Approximately 250ms to reach the setpoint for a 70rpm step