DEVELOPMENT OF A MOTOR SPEED CONTROL SYSTEM USING MATLAB AND SIMULINK, IMPLEMENTED WITH A DIGITAL SIGNAL PROCESSOR

by

ANDREW KLEE
B.S. University of Central Florida, 2003

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2005

ABSTRACT

This thesis describes an improved methodology for embedded software development. MATLAB and Simulink allow engineers to simplify algorithm development and avoid duplication of effort in deploying these algorithms to the end hardware. Special new hardware targeting capabilities of MATLAB and Simulink are described in detail.

A motor control system design served to demonstrate the efficacy of this new method. Initial data was collected to help model the motor in Simulink. This allowed for the design of the open and closed loop control systems. The designed system was very effective, with good response and no steady state error.

The entire design process and deployment to a digital signal processor took significantly less time and effort than other typical methods. The results of the control system design as well as the details of these development improvements are described.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

DSP	Digital Signal Processor
RTDX	Real Time Data Exchange
PWM	Pulse Width Modulator
QEP	Quadrature Encoder Pulse
GUIDE	Graphical User Interface Development Environment

CHAPTER ONE: INTRODUCTION

Background

In 2003, a senior design team at UCF started work on an autonomous underwater vehicle (AUV). Upon completion, the vehicle was to be entered in Autonomous Unmanned Vehicle Systems International's (AUVSI) 7th Annual Underwater Vehicle Competition. I came on board as a consultant after the senior design students had been working on the project for a few weeks. At that point, they had already decided on many of the main aspects of the design. These included the structural design, hardware choices, and software development methodology. After the competition was finished, the team had learned from its mistakes and realized what worked well.

Objective

For the competition, the vehicle had to satisfy certain weight, size, and safety requirements. It had to fit inside a six-foot long, by three-foot wide, by three-foot high box and weigh less than 140 lbs. The lighter the vehicle was, the more points it garnered. It needed to be designed to carry out several tasks present in the obstacle course. For the actual competition run, the vehicle was lowered into the water, where a diver positioned the vehicle at the starting point and disengaged the emergency stop. The vehicle had to be completely autonomous, with no communication with the shore, either wired or wireless. Once started, the vehicle would dive a few feet then move forward and pass through a floating PVC gate.

The vehicle should then attempt to find the light source, which is blinking at a particular frequency, and home in on it. Directly in front of the light source was located a target array, which consisted of five black rectangular bins surrounded by white border. When the vehicle was sufficiently close to the light source, and therefore over the target array, the vision system

attempted to discern the highest target bin, and commanded the vehicle to drop two markers in that bin. Once this was completed, the vehicle would scan around for the pinger, which was in the recovery zone. The vehicle would move towards the recovery zone and surface within the buoys.

Structure

The structure of the AUV was a radical departure from the typical layout of existing vehicles. One team member had noticed that underwater vehicles tend to have to deal with unwanted torque from each propeller, which degrades efficiency and can add errors to the control system. He had the novel idea to try and harness that torque and use it for navigation, effectively turning a problem into an advantage.

The vehicle needed to travel forward, dive and ascend, as well as rotate left and right. To move in the lateral direction, the vehicle has two horizontally oriented propellers inline. They're couterpitched, so rotating in opposite directions gives the vehicle a positive force in the lateral direction, but the torque from each propeller effectively cancels the torque from the other.

There are two propellers in the vertical direction, which intuitively cause the vehicle to move up and down. When these two propellers rotate at equal speeds, but opposite directions, the vehicle moves either up or down, and the torque cancellation keeps the vehicle from spinning. However, if both propellers were to spin the same direction at the same speed, the net force in the vertical direction is zero, yet the torque in that direction is considerable, enough to make the vehicle spin. A simple contol system can be set up to vary the speed and directions of each vertically oriented propeller so as to maintain or achieve a particular depth and heading.

Software

The team needed to choose the application software with which they would develop the control systems, artificial intelligence, and sensor interface systems. There are several solutions, each offering different levels of abstraction, performance, and ease of programming.

The solution the team came up with was to use MATLAB and Simulink from the Mathworks, Inc. This software has several distinct advantages. MATLAB allows one to use a high-level language which is similar to C++, known as m-code. MATLAB has hundreds of built-in functions which can be used to accomplish almost any task imaginable. These fields include math, statistics, video and image acquisition and processing, RF design, signal processing, simulation, and many more.

Simulink is a platform that has most of the same functionality of MATLAB, but allows engineers to design systems graphically, with a block diagram interface. Simulink has standard blocksets that allow the user to implement common tasks, such as I/O, summers, signal routing, scopes, etc. Additional blocksets can be purchased that add extra functionality to Simulink.

Stateflow is a component of Simulink which is useful for specifying the behavior of systems. Typically in the past, engineers would first specify the general behavior of the system on paper, in a block diagram (or some sort of pseudocode). They would then translate that diagram into operational code. Stateflow takes those two steps and makes them one. It allows the engineer to drag blocks into the model that represent states, transition events, and control logic, all of which can execute on an embedded target.

Real-Time Workshop is an important component of Simulink that allows users to develop Simulink models on their work computer and implement them on other targets. These targets can include other PCs, embedded computers, microcontrollers, DSPs, and even FPGAs. Some of

the supported embedded targets include Texas Instruments' C6000 and C2000 DSPs, Infineon's C166, and Motorolla's MPC555 and HC12.

Hardware

Once MATLAB was chosen as the software development scheme, hardware had to be chosen that could actually implement code generated from within MATLAB. The team chose the Prometheus from Diamond Systems, a PC/104 embedded computer. It has analog and digital I/O, four serial ports, typical PC connections such as keyboard, mouse, and VGA, and several expansion headers. The key reason for its selection was a dedicated blockset in Simulink for the Prometheus and its support for xPC Target. The specialized blockset greatly simplified design of the main system Simulink model. Rather than being forced to write code to interface with the Prometheus' peripherals, one could simply drag in the requisite blocks.

xPC Target provides a host-target prototyping environment that enables users to connect their Simulink and Stateflow models to physical systems and execute them in real time on PC-compatible hardware. xPC Target enables you to add I/O interface blocks to your models, automatically generate code with Real-Time Workshop and Stateflow Coder, and download the code to a second PC running the xPC Target real-time kernel. With a host PC connected to the Prometheus vi xPC Target, the team now had a system for developing system models, porting them to the Prometheus, and monitoring their real-time behavior.

The team soon established that the Prometheus was adequate for sensor interfacing, motor control, and state machine execution, but didn't have the performance or resources to execute the vision system. The team decided to add another computer that would be solely responsible for vision processing. The Mocha 5043 is a regular PC, only with a very small form factor. In addition to implementing vision processing, it only made sense to make the Mocha the

host in the xPC Target host/target interface as well. The Mocha was used for developing Simulink models and initiating code downloads to the Prometheus. The same link for downloading could also be used for duplex communication between the two computers.

Issues

In the end, the Prometheus ran the xPC Target kernel and the Mocha ran Windows XP©, with a full instance of MATLAB running at all times. Therefore, both machines needed to boot up before they could operate. If one crashed, it often affected the other. When problems occurred, it was often necessary to reset both machines, which could take up to two minutes to complete.

Heat was a huge issue with both computers. They needed to be sealed in water-tight boxes, which completely cut off all air circulation. The solution was to house each in a heavy aluminum box. A custom heatsink was made for the Mocha to thermally couple it to the box, and the surrounding water. This idea worked well enough when the vehicle was submerged in the water. However, in the air, both computers would quickly reach dangerous temperatures (60°C). When working in the lab, the team was forced to partially submerge the computers in bins of water. At the competition, when the computer boxes were attached, the team could only work on the vehicle out of water for a few minutes, before either quitting or dunking it back in the pool. This heat issue was a huge inconvenience, and even worse, an ever-present danger to the computer systems.

Performance was a key issue. The Prometheus handled its tasks adequately. However, the throughput of the vision system was only about two frames per second. This was surprising, considering the Mocha had a Pentium 4 CPU running at 2.4 GHz. Evidently, the overhead of

Windows XP, MATLAB and Java, and the webcam's acquisition software were adversely affecting its performance.

New Design

Development with MATLAB and Simulink were probably the most successful features of the previous design, providing the team with flexibility, rapid prototyping ability, and a method for easily sharing designs among different members of the group. The new design will definitely be centered on this software.

However, the limitations in the hardware have forced the team to look into other processors for the new vehicle. They have to be easily programmable with MATLAB and Simulink, possess very high performance, and require much less power than the previous computers. The new design will be based on Texas Instruments' F2812 and DM642 DSPs. The F2812 will replace the Prometheus and the DM642 will replace the Mocha. Both DSPs have much higher performance, use significantly less power, require no operating system, and are very compatible with MATLAB and Simulink. These hardware and software issues were sort of an inspiration for the topic of this thesis.

Hence, the problems faced by the first design gave me an opportunity to develop a new motor controller design as a topic for my thesis. In chapter 2, this thesis discusses the various aspects of Mathlab, Simulink and the Texas Instruments C2000 DSP system used as the hardware platform for the motor controller. Chapter 3 discusses the control system design and its parameter selection. Chapter 4 gives the measured result of the control system performance comparing theoretical predictions to actual measured responses. Finally, Chapter 5 summarizes the results of this thesis and discusses some future work

CHAPTER TWO: TOOLS USED

It is a very common challenge for engineers to develop and test software to carry out various tasks on embedded hardware. This can often be a very tedious process. Engineers usually plan out the algorithm graphically or with pseudocode. They must then translate that into a high level language, which involves writing many lines of code.

Though this has been the common methodology for many years, there is an emerging technique that involves developing algorithms with graphical tools. This improves development time, decreases programming complexity, and facilitates sharing of the design with coworkers. A leading software package that fits this mold is MATLAB, from The Math Works[1]. It has many impressive qualities that make it suitable for simulation, data manipulation, and end hardware targeting.

To demonstrate the efficiency and performance of MATLAB, it was used to develop a motor speed control system. Both open-loop and closed-loop systems were designed and simulated with MATLAB. They were then implemented using a TMS320F2812 digital signal processor (DSP) from Texas Instruments[2]. MATLAB has special tools included that allow it to compile code for this DSP, program it, and transfer data to and from it in real-time.

MATLAB

MATLAB encompasses numerous facets of engineering. There are built in functions dealing with video and image processing, filter design, communications, control system design, just to name a few. Rather than having to be an expert in every field, the engineer can focus on solving the problem at hand. All that is required is a basic knowledge of the MATLAB development environment, the help system, and Matlab's programming language. It's a simple matter to locate and integrate the appropriate functions to solve the current problem. MATLAB has very impressive data display capabilities, as can be seen in Figure 1. There are 2-D and 3-D plots for which the user can manipulate many parameters.

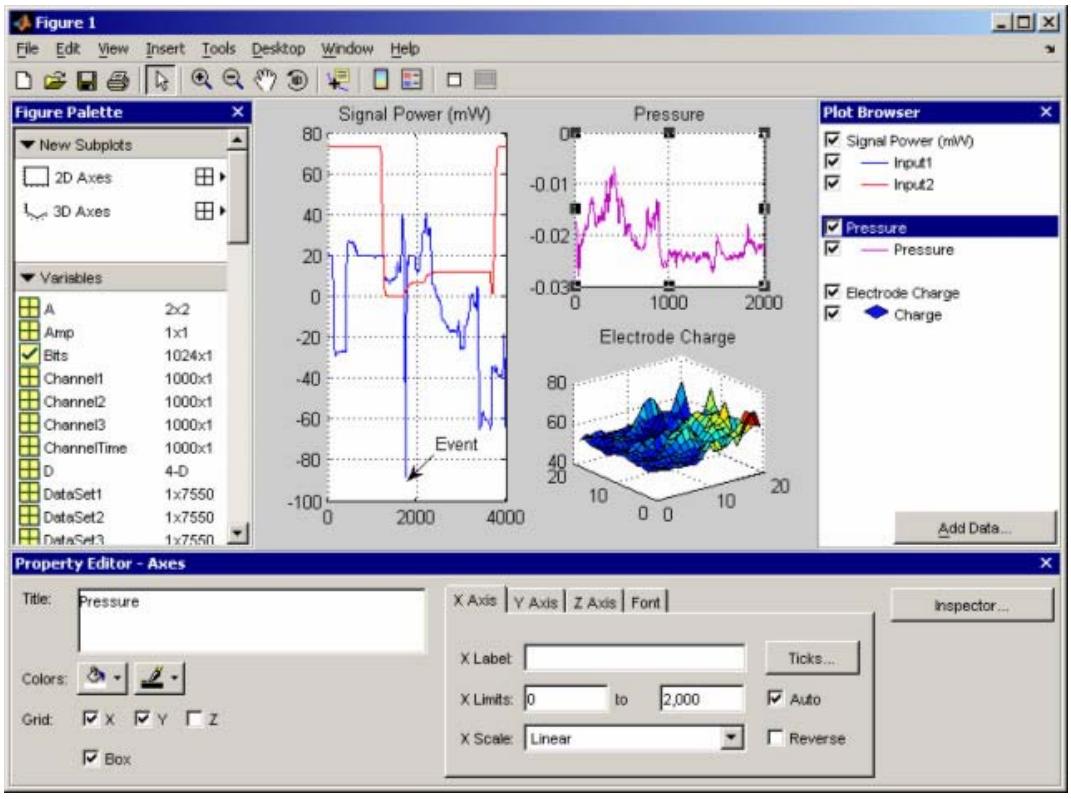

Figure 1: An Example MATLAB Plot

Simulink

Simulink is an extension of MATLAB that enables users to design and simulate systems by creating block diagrams. Systems can be modeled as continuous or discrete, or even a hybrid

of the two. Engineers choose from a large list of blocks, and simply drag them into a model. Blocks can include signal generators, scopes, function blocks, and even user defined blocks. The mouse can be used to draw connecting lines between blocks to assemble the system. A typical Simulink model is shown in Figure 2. There are blocks for math, signal routing, displays, etc.

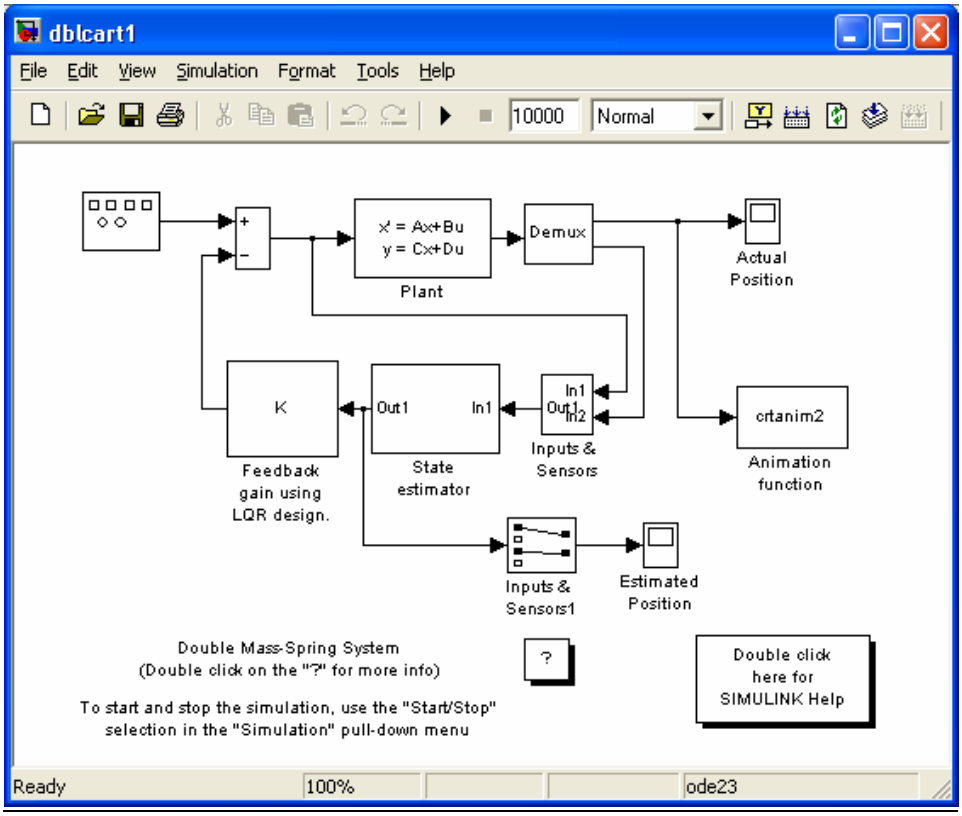

Figure 2: An Example Simulink Model

GUIDE

GUIDE stands for graphical user interface development environment. It provides a set of tools for creating graphical user interfaces (GUIs). An example of GUI development with GUIDE is shown below in Figure 3. GUIs are useful for letting users input data graphically and view results graphically, all in the same figure window.

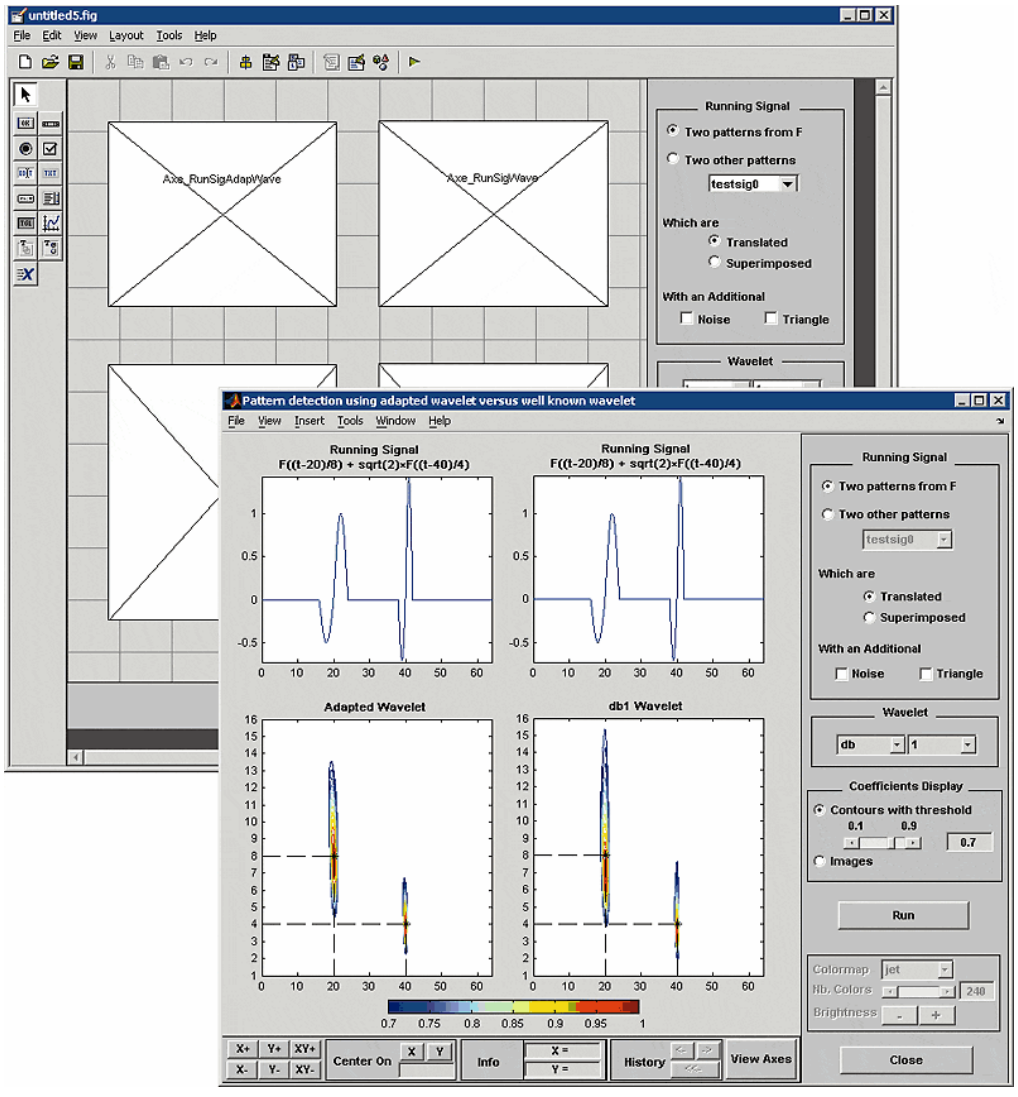

Figure 3: MATLAB's GUIDE Utility

Embedded Target For TI C2000 DSP

The Embedded Target for TI C2000 DSP integrates MATLAB and Simulink with Texas Instruments C2000 DSP processors. It lets engineers perform automatic code generation, prototyping, and embedded system deployment on TI C2000 processors. There are custom blocks that can be added to Simulink models to target the C2000 DSP's peripherals. Figure 4 shows some examples of these blocks. When these blocks are added to a model, the necessary code will be generated by MATLAB.

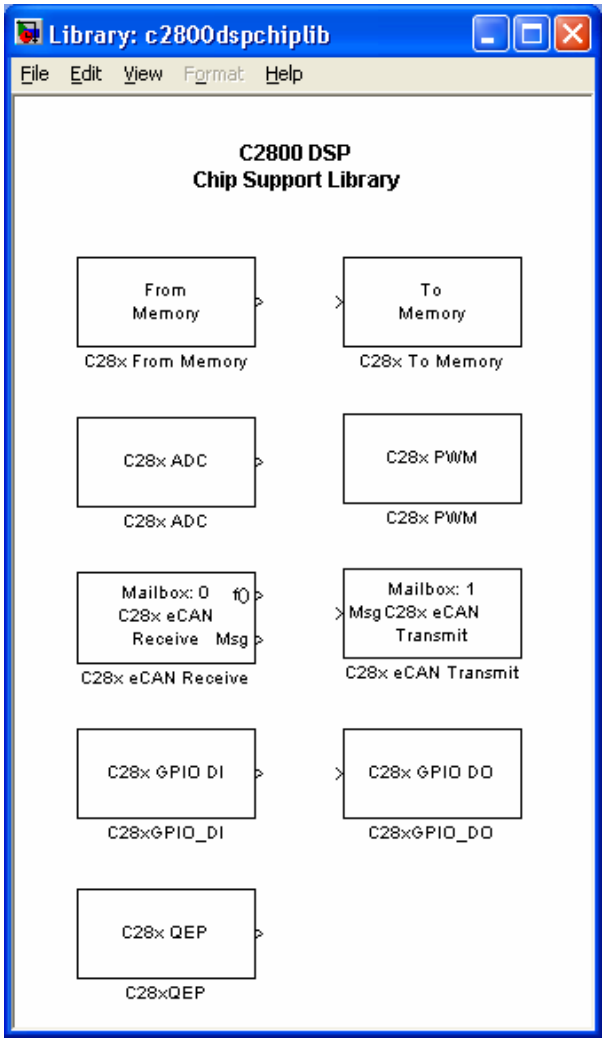

Figure 4: Simulink Blocks for TI C2000 DSP

TI TMS320F2812 DSP

The F2812 DSP is an excellent choice for this experiment for several reasons. The board used in this experiment is a development kit from Spectrum Digital called the F2812 eZdsp board, shown in Figure 5[3]. It's relatively inexpensive, at around $300. The F2812 has several useful peripherals including six pulse width modulators (PWM), sixteen channels of twelve-bit analog-to-digital converters, and two quadrature encoder input modules (QEP). Most importantly, the eZdsp kit is compatible with MATLAB and Simulink.

Figure 5: Spectrum Digital F2812 eZdsp Development Kit

The PWM outputs enable the DSP to efficiently drive the motor. By varying the duty cycle of the PWM signal, a varying amount of power is delivered to the motor. This is more effective than continuously changing the voltage. The quadrature encoder input module allows the DSP to accurately read the speed of the motor.

Experimental Setup

The motor used in the experiment is a Pittman 8222D116 motor, shown in Figure 6. It's rated for operation up to 24 volts. It has a built-in encoder that outputs 1000 pulses per revolution. At this resolution, the DSP's quadrature encoder inputs are capable of reading speeds of up to 5000 rpm.

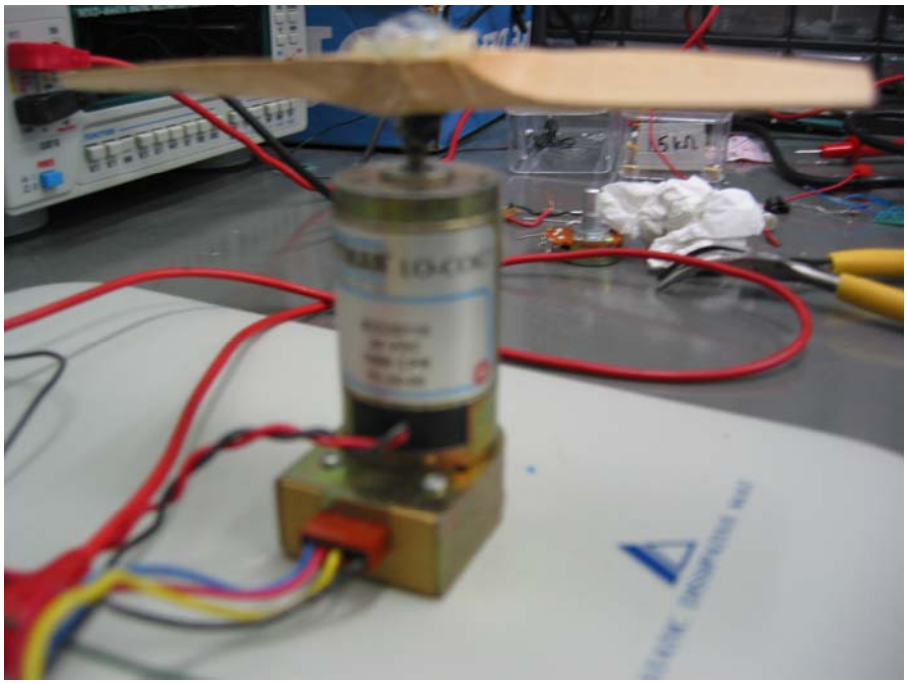

Figure 6: Pittman 8222D116 Motor with Built-in Encoder

The DSP cannot directly drive the motor. Its PWM outputs can source about 4 mA of current. The motor consumes about 100 mA when it's supplied with 12 V. To provide this current, a motor driver integrated circuit is used. The L298 from ST Microelectronics is capable of powering two motors with up to 2 Amps each. Only one channel is necessary for this experiment and the motor only spins in one direction.

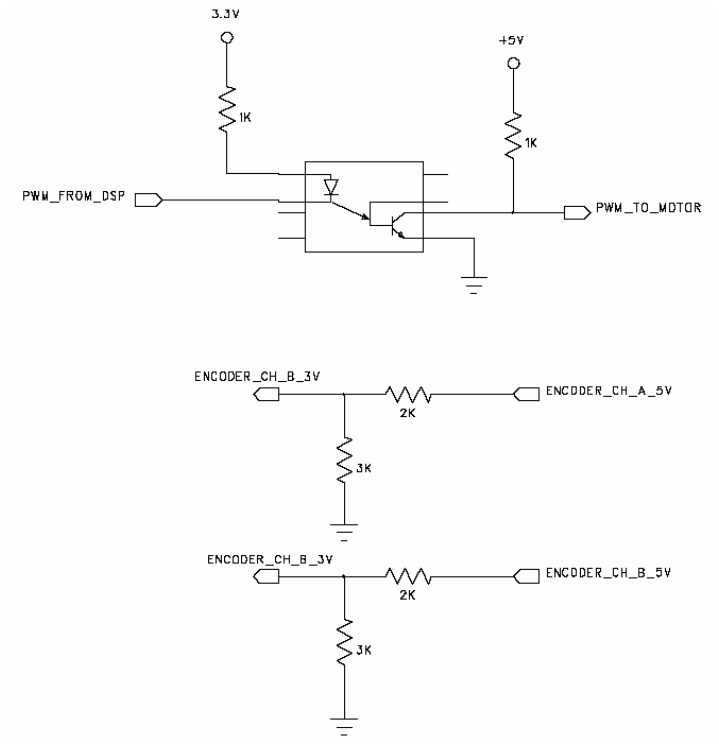

Figure 7: Additional Circuitry

Some signal conditioning is required to properly interface the DSP board to the motor and encoder. A small circuit board was assembled to address these issues. The encoder's outputs are 5V signals. The DSP's input pins are not tolerant of 5V levels. In fact, the entire chip can possibly be damaged by overvoltage signals. To reduce the levels to 3V, a simple voltage divider is used for both channels of encoder output data. Though the motor driver chip provides some electrical isolation between the motor and DSP, an alternate solution with more protection was adopted. The PWM output on the DSP was connected to the input side of an optocoupler. The output of the optocoupler is connected to the control input of the motor driver chip. The signal is converted from electrical to light to electrical energy. The motor and motor driver chips are powered from a different power supply than the DSP. Now, even if the motor is

drawing high levels of current, the DSP is unaffected. This circuitry is shown in Figure 7. The block diagram for the entire system is shown in Figure 8. Figure 9 shows the F2812 eZdsp board, motor driver board, interface board, and motor.

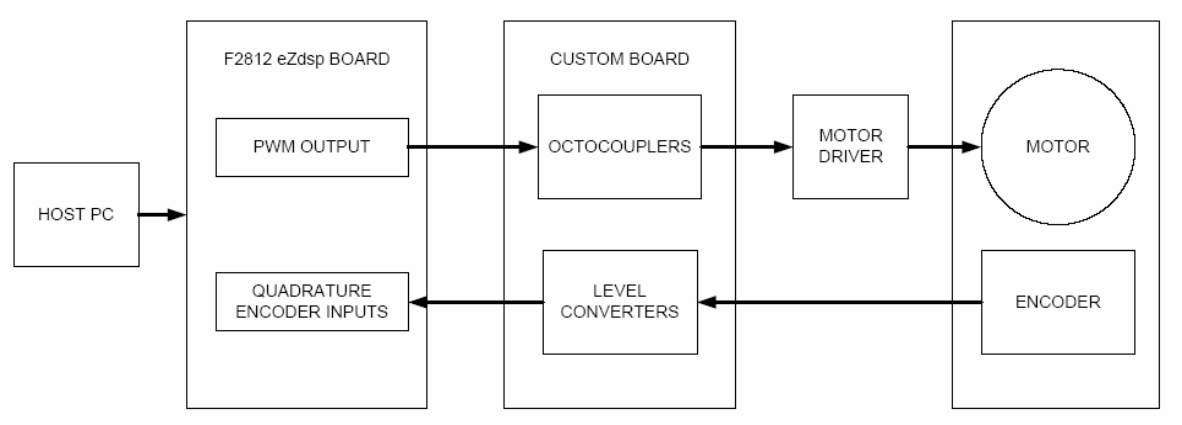

Figure 8: System Block Diagram

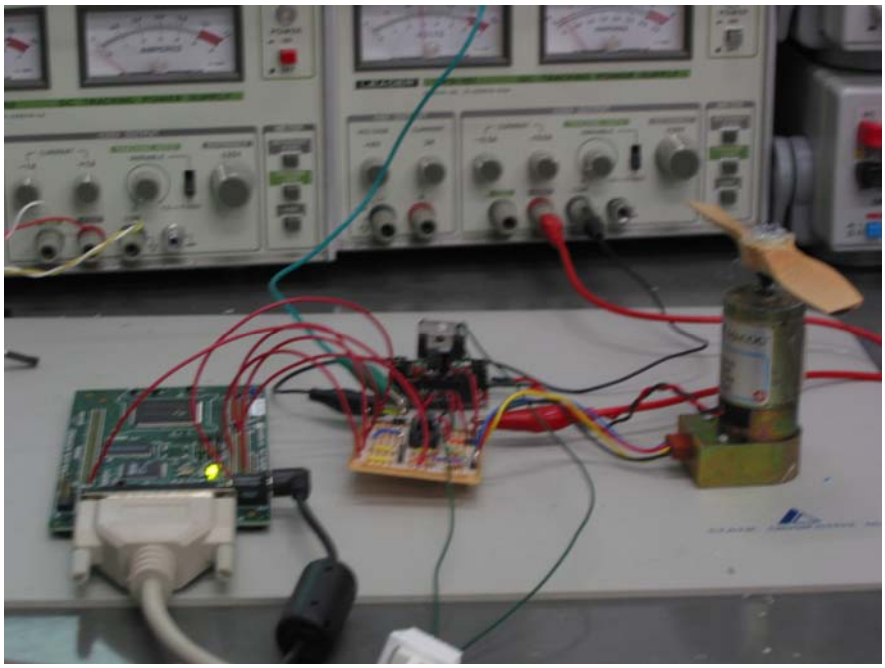

Figure 9: Experimental Setup

Development Process

Development with Simulink for the F2812 consists of a few steps. The user first creates a Simulink model on the host computer, using special blocks from the Embedded Target for C2000 Blockset to target the DSP's peripherals.. Pressing CTRL-B causes MATLAB to build that model into C code that is executable in the DSP. MATLAB causes TI's Code Composer Studio IDE to open, compile the active project, and load it in the DSP for execution. The computer is connected to the F2812 eZdsp board via a parallel cable. This connection allows for programming of the DSP as well as real-time communication with the DSP via RTDX. This stands for Real Time Data Exchange, and is TI's technology for debugging their DSPs without halting their execution.

The model can be configured to run indefinitely or a set amount of time. The models in these experiments run for four seconds each. They determine the power delivered to the motor and interpret the quadrature pulse train from the encoder to determine the motor's speed. The models also contain the blocks that carry out the control system relevant to that particular test. Host side m-files give the user the ability to reload the DSP, start its execution, retrieve data readings after each trial is finished, and halt execution of the DSP. These commented m-files are listed in Appendix A.

CHAPTER THREE: MODELING AND CONTROL

Motor Dynamics

An armature controlled DC motor with a load inertia mounted on its shaft is shown in Figure 10. The inputs are the armature voltage $e_0(t)$ and the load torque $T_L(t)$. The outputs are the motor torque $T(t)$ and angular speed of the motor $\omega(t)$. Dependent variables (in addition to the outputs) are the armature current $i(t)$ and back emf of the motor $v_b(t)$. R and L are the electrical resistance and inductance of the armature circuit while B and J are the viscous damping coefficient and load inertia. K_b and K_T are the back emf and torque constants of the motor.

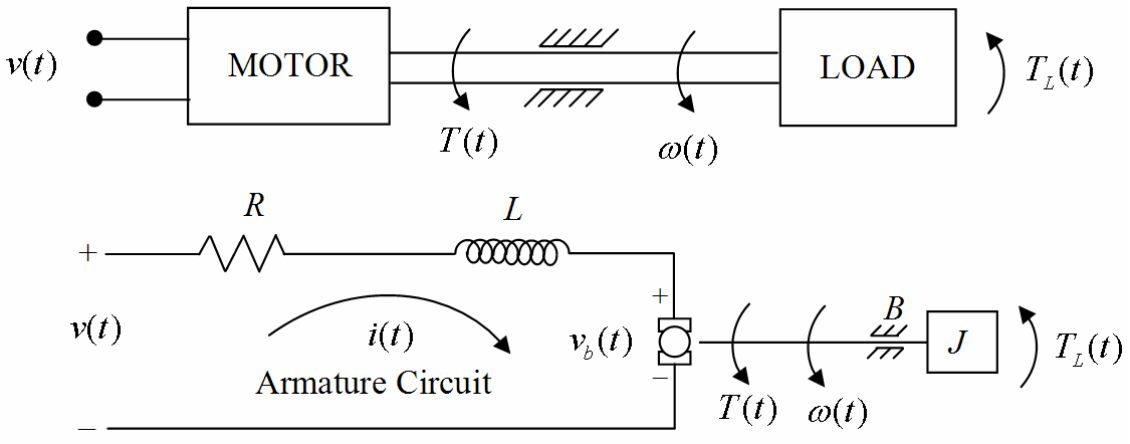

Figure 10 Armature Controlled DC Motor and Load

The following equations govern the dynamics of this electromechanical system:[4]

$$v(t) = Ri(t) + L\frac{d}{dt}i(t) + v_b(t), \tag{3.1}$$

$$v_b(t) = K_b\omega(t), \tag{3.2}$$

$$T(t) = K_T i(t), \tag{3.3}$$

and $J\frac{d}{dt}\omega(t)+B\omega(t)=T(t)+T_L(t)$. (3.4)

Eq (3.1) results from applying Kirchoff's law for voltage drops around the armature circuit. Eq (3.4) is based on Newton's law for rotational systems. Eqs (3.2) and (3.3) couple the electrical and mechanical operation of the motor.

Laplace transforming Eqs (3.1) - (3.4) provides the basis for constructing the block diagram shown in Figure 11.

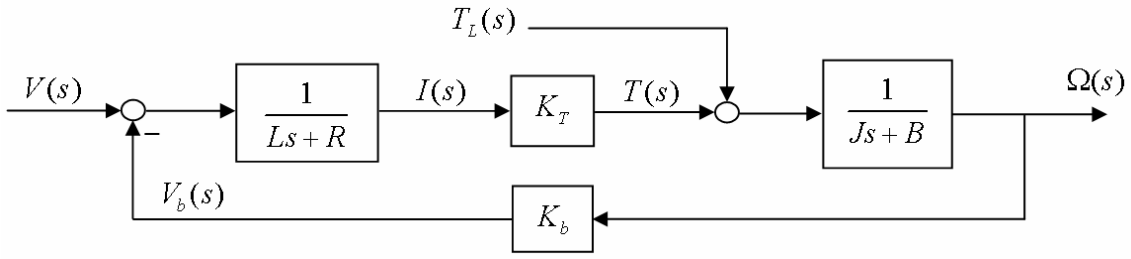

Figure 11: Block Diagram of Armature Controlled DC Motor

$V(s)=L\{v(t)\}$ and $T_L(s)=L\{T_L(t)\}$ are Laplace transforms of the inputs, $I(s)=L\{\boldsymbol{i}(t)\}$, $T(s)=L\{T(t)\}$, $V_b(s)=L\{v_b(t)\}$ and $\Omega(s)=L\{\omega(t)\}$ are Laplace transforms of the dependent variables.

The transfer functions $G_\Omega(s)$, $G_L(s)$ are obtained by block diagram algebra or Mason's Gain formula (Dorf). The results are

$$G_\Omega(s)=\left.\frac{\Omega(s)}{V(s)}\right|_{T_L(s)=0} = \frac{\left(\frac{1}{Ls+R}\right)K_T\left(\frac{1}{Js+B}\right)}{1+K_b\left(\frac{1}{Ls+R}\right)K_T\left(\frac{1}{Js+B}\right)}, \quad (3.5)$$

$$= \frac{K_T}{(Ls+R)(Js+B)+K_bK_T}, \quad (3.6)$$

$$G_L(s) = \left.\frac{\Omega(s)}{T_L(s)}\right|_{E_0(s)=0} = \frac{\left(\frac{1}{Js+B}\right)}{1+K_b\left(\frac{1}{Ls+R}\right)K_T\left(\frac{1}{Js+B}\right)}\,, \tag{3.7}$$

and

$$= \frac{Ls+R}{(Ls+R)(Js+B)+K_bK_T}. \tag{3.8}$$

By superposition, a property of linear systems, the motor response to combined changes in armature voltage and load torque is

$$\Omega(s) = G_\Omega(s)V(s) + G_L(s)T_L(s)\,. \tag{3.9}$$

From Eq (3.9), the block diagram of the motor is as shown in Figure 12.

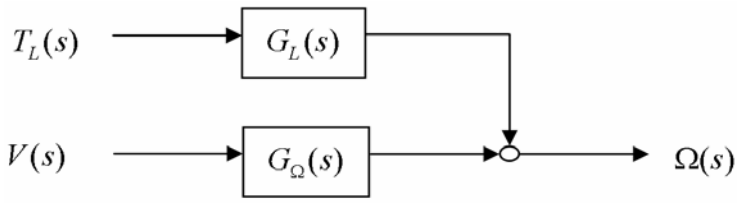

Figure 12: Block Diagram of Motor

The motor is modeled as a second order system with characteristic polynomial

$$\Delta(s) = (Ls+R)(Js+B) + K_bK_T\,. \tag{3.10}$$

Dividing $G_\Omega(s)$ in Eq (3.6) by *JL* and equating the result to the standard form of a second order system transfer function,

$$G_\Omega(s) = \frac{\frac{K_T}{JL}}{\left(s+\frac{R}{L}\right)\left(s+\frac{B}{J}\right)+\frac{K_bK_T}{JL}} = \frac{K\omega_n^2}{s^2+2\zeta\omega_n s+\omega_n^2}\,. \tag{3.11}$$

Solving for the steady-state gain (from voltage to angular speed) K_m, the natural frequency ω_n and the damping ratio ζ in terms of the motor parameters results in

$$K_m = \frac{K_T}{BR + K_b K_T}, \tag{3.12}$$

$$\omega_n = \left(\frac{BR + K_b K_T}{JL} \right)^{1/2}, \tag{3.13}$$

$$\text{and } \zeta = \frac{(BL + JR)}{2\left[JL\left(BR + K_b K_T \right) \right]^{1/2}}. \tag{3.14}$$

The characteristic roots [poles of $G_\Omega(s)$] are obtained by solving $\Delta(s) = 0$.

$$s_1, s_2 = -\zeta \omega_n \pm \sqrt{\zeta^2 - 1}\, \omega_n . \tag{3.15}$$

The transfer function $G_\Omega(s)$ in Eq (3.11) is expressible in terms of the characteristic roots and motor time constants by

$$G_\Omega(s) = \frac{K\omega_n^2}{(s - s_1)(s - s_2)} = \frac{K\omega_n^2 \tau_1 \tau_2}{(\tau_1 s + 1)(\tau_2 s + 1)}, \tag{3.16}$$

where $\tau_1 = -\frac{1}{s_1}, \quad \tau_2 = -\frac{1}{s_2}$. Substituting Eqs (3.13) and (3.14) into Eq (3.15) yields

$$s_1, s_2 = -\frac{1}{2JL}\left[(BL + JR) \pm \left\{ (BL + JR)^2 - 4JL(BR + K_b K_T) \right\}^{1/2} \right]. \tag{3.17}$$

The motor time constants are therefore

$$\tau_1, \tau_2 = \frac{2JL}{(BL + JR) \pm \left\{ (BL + JR)^2 - 4JL(BR + K_b K_T) \right\}^{1/2}}. \tag{3.18}$$

The armature inductance of a DC motor is almost negligible. Ignoring terms involving L in the denominator of Eq (3.18) gives an approximate expression for τ_1:

$$\tau_1 \approx \left[\frac{2JL}{(BL + JR) + \left\{ (BL + JR)^2 - 4JL(BR + K_b K_T) \right\}^{1/2}} \right]_{L \approx 0} = \frac{L}{R}. \tag{3.19}$$

The second time constant is simplified as follows.

$$\tau_2 \approx \lim_{L \to 0} \left[\frac{2JL}{(BL+JR) - \left\{ (BL+JR)^2 - 4JL(BR+K_b K_T) \right\}^{1/2}} \right]. \tag{3.20}$$

The limit as $L \to 0$ in Eq (3.20) is indeterminate. By L'Hospital's Rule,

$$\tau_2 \approx \lim_{L \to 0} \left[\frac{\frac{d}{dL}(2JL)}{\frac{d}{dL}\left[(BL+JR) - \left\{ (BL+JR)^2 - 4JL(BR+K_b K_T) \right\}^{1/2} \right]} \right], \tag{3.21}$$

$$\approx \lim_{L \to 0} \left[\frac{2J}{B - \frac{1}{2}\left[(BL+JR)^2 - 4JL(BR+K_b K_T \right]^{-1/2} \left[2(BL+JR)B - 4J(BR+K_b K_T) \right]} \right], \tag{3.22}$$

which reduces to $$\tau_2 \approx \frac{JR}{BR+K_b K_T}. \tag{3.23}$$

Eqs (3.19) and (3.23) are expressions for the electrical and mechanical time constants respectively of the motor and generally differ by several orders of magnitude. For example, the time constants of a typical Maxon F series DC motor [5] are $\tau_e = 55\mu$sec and $\tau_m = 52$ msec.

It is common to ignore the armature inductance entirely and treat the motor as a first order component. This assumption introduces negligible error since the frequency components of the inputs $e_0(t)$ and $T_L(t)$ are well below the break frequency associated with the electrical time constant, $\omega_b = 1/\tau_e$.

The transfer function $G_\Omega(s)$ becomes a first order lag,

$$G_\Omega(s) = \frac{\Omega(s)}{V(s)} = \frac{K_m}{\tau_m s + 1}, \tag{3.24}$$

where K_m is the steady-state motor gain (rpm/volt) in Eq (3.12) and τ_m is the motor time constant, same as the mechanical time constant in Eq (3.23). The same result follows directly from Eq (3.6) with $L = 0$.

The actual motor is driven by a pulse width modulated (PWM) signal with period *T* and duty cycle $f = P/T$ like the one shown in Figure 13.

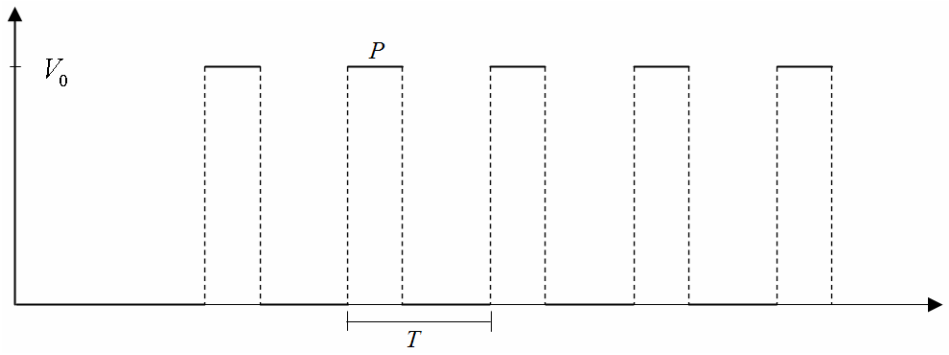

Figure 13: Pulse Width Modulated (PWM) Input to Motor

For a variable duty cycle $f(t)$, the effective voltage applied to the motor is

$$v(t) = f(t)V_0 = \frac{P(t)}{T}V_0 . \tag{3.25}$$

Replacing $V(s)$ by $V_o F(s)$ in Eq (3.24) gives

$$G_\Omega(s) = \frac{\Omega(s)}{V_0 F(s)}, \qquad F(s) = L\left\{ f(t) \right\}. \tag{3.26}$$

which leads to a new motor transfer function

$$\hat{G}_\Omega(s) = \frac{\Omega(s)}{F(s)} = V_0 G_\Omega(s) = V_0 \frac{\Omega(s)}{V(s)} = V_0 \frac{K_m}{\tau_m s + 1} = \frac{\hat{K}_m}{\tau_m s + 1}, \tag{3.27}$$

where $\hat{K}_m = V_0 K_m$ is the motor gain in rpm per % duty cycle.

Determination of Motor Gain and Time Constant

The motor parameters were determined on the basis of its step response about a steady-state operating point, namely $\overline{f} = 50\%$, $V_0 = 12$ volts, $\overline{\omega} = 1615$ rpm. The period of the PWM signal was set at 853 μsec. A series of runs numbered #1 thru #10 were conducted where the duty cycle f was incremented from $\overline{f} = 50\%$ in steps of 10% up to 100% and decremented from $\overline{f} = 50\%$ in steps of 10% down to zero. The step responses were obtained using a sampling rate of 50 Hz ($T = 0.02$ sec). Several are shown in Figure 14.

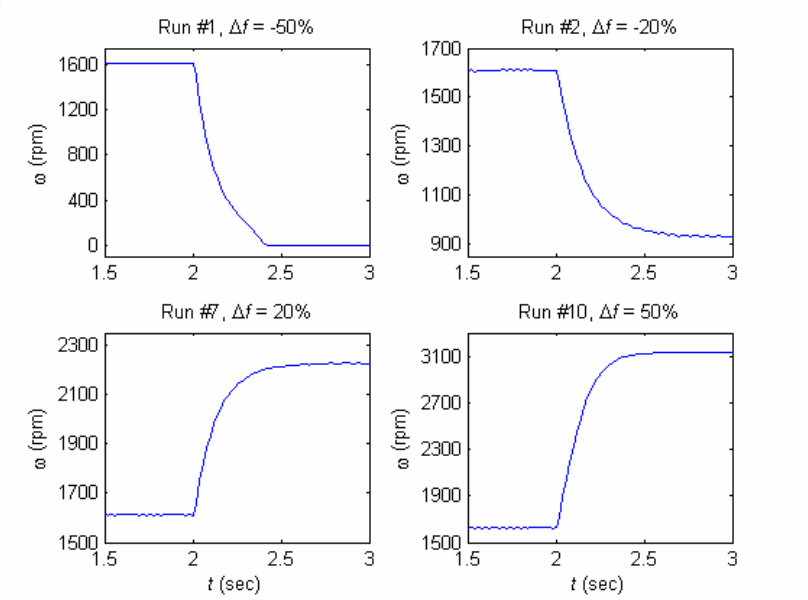

Figure 14: Step Response of Motor for Several Runs

For each step response, the time constant τ and steady-state motor gain $\hat{K}_m$ were determined as illustrated in Figure 14. The gain $\hat{K}_m$ was obtained as the difference between the maximum and minimum rpm values divided by the % change in duty cycle from the nominal value of 50%, i.e.

$$\hat{K}_m = \frac{\omega_{\max} - \omega_{\min}}{\Delta f} = \frac{(\bar{\omega} + \hat{K}_m \Delta f) - \bar{\omega}}{\Delta f} \quad \text{(rpm per \% change in duty cycle)}. \tag{3.28}$$

The motor time constant was estimated as the time required for the speed to increase from the initial steady-state value $\omega(t_0) = \bar{\omega}$ at time t_0 up to the value

$$\omega(t_0 + \tau) = \bar{\omega} + \hat{K}_m (1 - e^{-(t-t_0)/\tau}) \Delta f \Big|_{t=t_0+\tau}, \tag{3.29}$$

$$= \bar{\omega} + \hat{K}_m (1 - e^{-1}) \Delta f, \tag{3.30}$$

$$= \bar{\omega} + 0.6321 \Delta f. \tag{3.31}$$

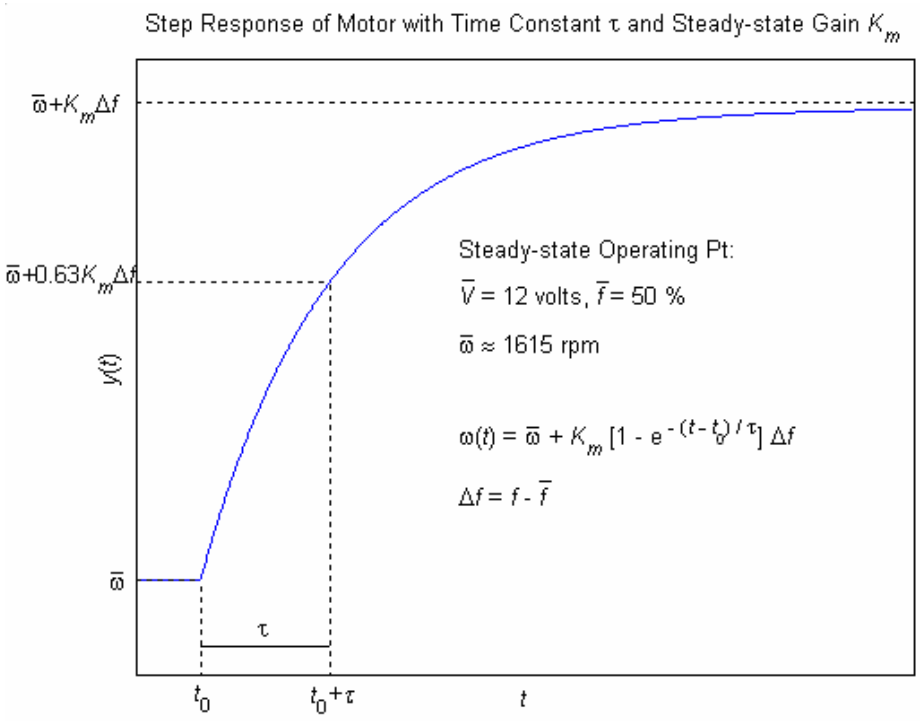

Figure 15: Step Response of a DC Motor

Table 1 shows the results of estimating the motor parameters from the 10 recorded step responses. Since the motor does not begin to turn until the duty cycle is somewhere between 10% and 20%, the averages in the last row exclude the results for Runs #1 and #2 where the total duty cycle f was 0% and 10%, respectively.

Table 1: Experimental Results for Motor Gain and Time Constant

Run	Δf (%)	f (%)	$\hat{K}_m$ (rpm per % change)	τ (sec)
1	–50	0	32.41	0.17
2	–40	10	40.44	0.25
3	–30	20	35.50	0.19
4	–20	30	34.39	0.19
5	–10	40	33.90	0.17
6	10	60	31.43	0.15
7	20	70	30.98	0.15
8	30	80	29.88	0.13
9	40	90	30.21	0.14
10	50	100	30.41	0.17
Ave	Runs 1-10		$(\hat{K}_m)_{ave} = 32.95$	$\tau_{ave} = 0.171$
Ave	Runs 3-10		$(\hat{K}_m)_{ave} = 32.08$	$\tau_{ave} = 0.161$

Open-loop Control of Motor

It's possible to control the motor speed open-loop as shown in Figure 16.

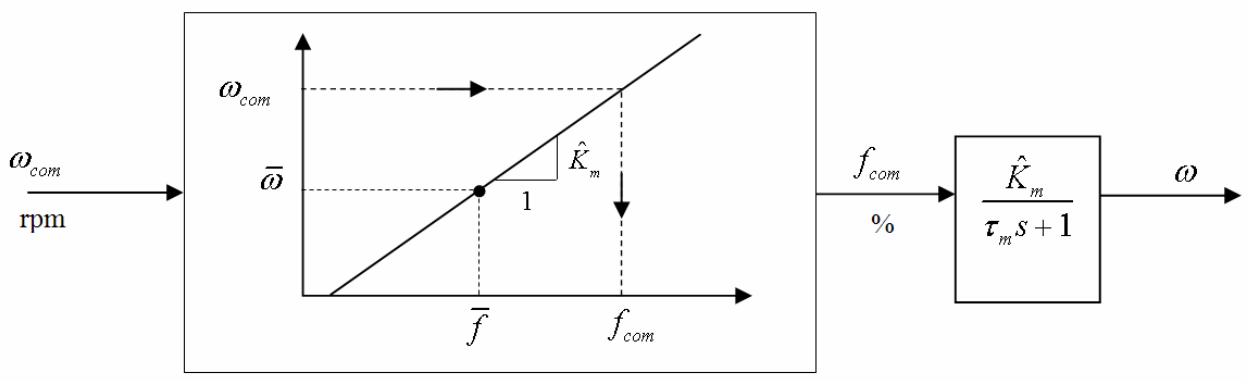

Figure 16: Open-loop Control of Motor Speed

The commanded speed ω_{com} is converted to a commanded duty cycle f_{com} using the operating characteristic of the motor. Open-loop control is satisfactory provided the operating characteristic is a good fit to the measured steady-state response of the motor. Equally important is the assumption that there are no external disturbances acting on the motor during open-loop control which were not present when the motor response data was obtained. The presence of external disturbances effectively changes the operating characteristic. A family of operating characteristics exist, one member for each set of disturbances and the open loop control would only work if the disturbances were measurable and the appropriate operating characteristic chosen to determine the required duty cycle for a commanded speed.

The operating characteristic of the motor is shown in Figure 17 along with the measured steady-state speeds from the 10 step responses.

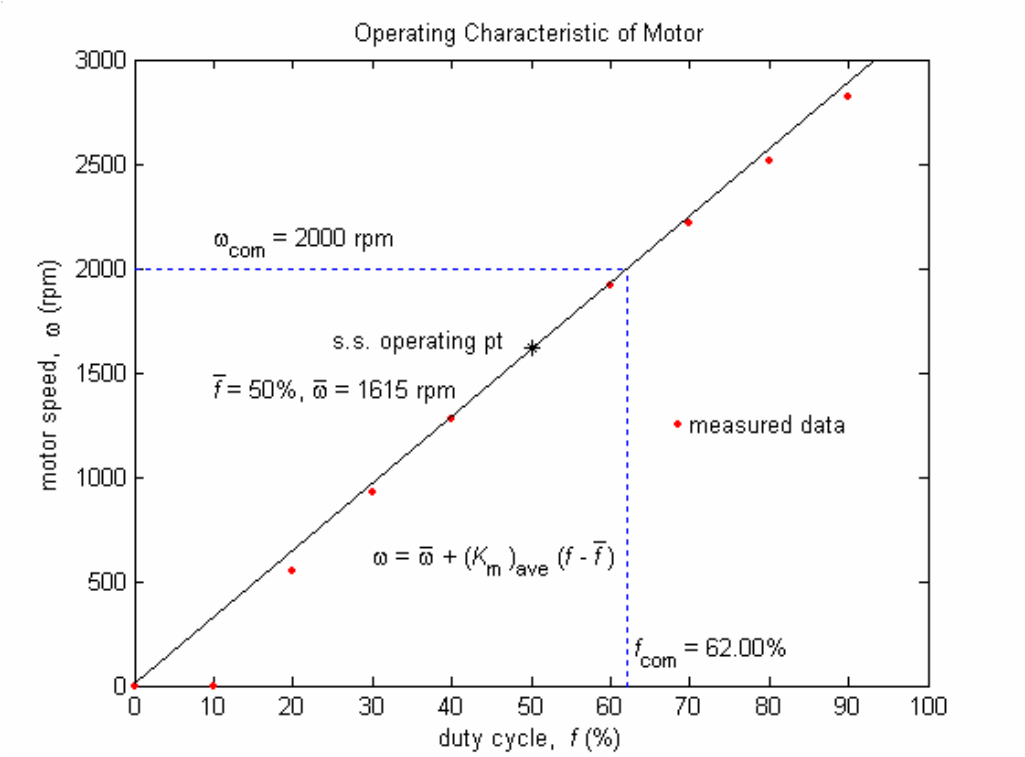

Figure 17: Operating Characteristics for Open-loop Control of Motor

With open-loop control and a commanded motor speed of 2000 rpm, the necessary duty cycle is computed from

$$f_{com} = \bar{f} + \frac{\omega_{com} - \bar{\omega}}{(\hat{K}_m)_{ave}} = 50 + \frac{2000 - 1615}{32.08} = 62.00\%. \qquad (3.32)$$

Closed-loop Control

A unity feedback closed-loop control system for controlling angular speed of a DC motor is shown in Figure 18. A sensor (tachometer, encoder, etc) is required in the feedback path to convert angular speed to a signal compatible with the type of controller. The sensor gain is combined with the controller transfer function in Figure 17 so that the command input may be shown in the same units as the control system output, i.e. rpm.

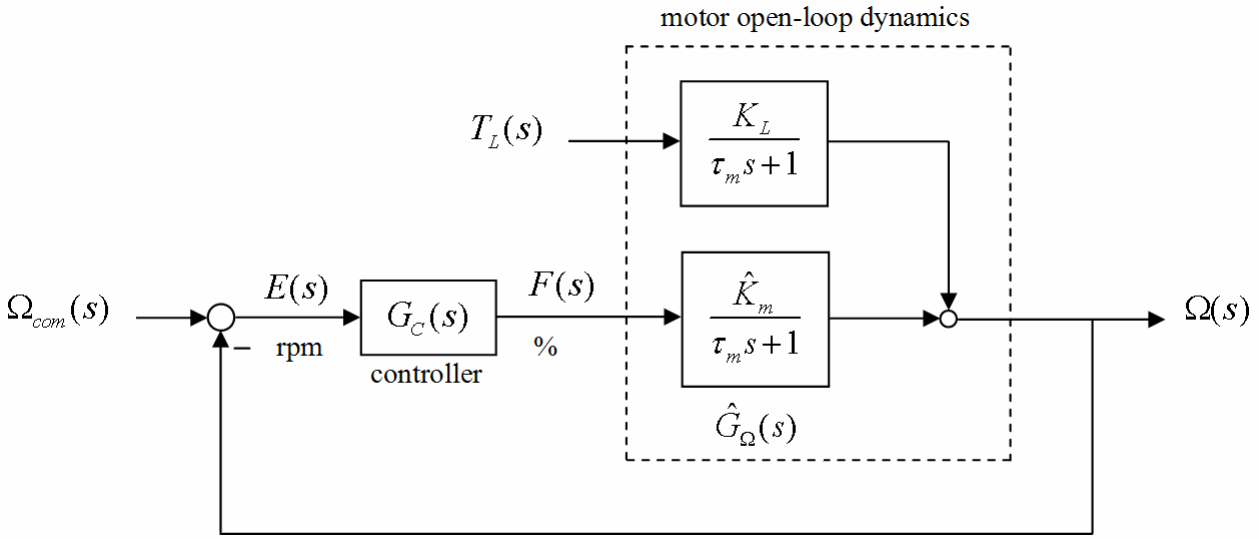

Figure 18: Closed-loop System for DC Motor Speed Control

The closed-loop control system transfer function from command input $\Omega_{com}(s)$ to angular speed $\Omega(s)$ is

$$T(s) = \frac{\Omega(s)}{\Omega_{com}(s)} = \frac{G_C(s)\hat{G}_\Omega(s)}{1+G_C(s)\hat{G}_\Omega(s)}, \tag{3.33}$$

$$= \frac{G_C(s)\dfrac{\hat{K}_m}{\tau_m s+1}}{1+G_C(s)\dfrac{\hat{K}_m}{\tau_m s+1}} = \frac{\hat{K}_m G_C(s)}{\tau_m s+1+\hat{K}_m G_C(s)}. \tag{3.34}$$

Transient response performance of the control system involves speed of response (rise time, settling time), overshoot, pole placement (dominant time constants), etc. Root locus is the classical approach to design of the controller [6]. Frequency response performance of the control system involve stability margins, bandwidth, cutoff frequencies, etc. Bode diagrams are used to synthesize the controller in this case.

Both methods are applicable in the design of digital control systems where an embedded microprocessor serves as the digital controller. Interfaces are present to receive and transmit information from the continuous system, i.e. the DC motor. Its common however to design an analog controller to satisfy the transient or frequency response requirements of the control system and then convert the result to a digital controller using an approximation.

A conventional three term P-I-D controller will be used to control the motor speed. The continuous form is described by

$$G_C(s) = \frac{\Delta F(s)}{E(s)} = K_P + \frac{K_I}{s} + K_D s . \tag{3.35}$$

The integral component makes the closed-loop system Type 1 which assures zero offset (steady-state error) for a step change in commanded motor speed. The derivative component is available if needed to improve stability by adding damping to the system.

Substituting Eq (3.35) into (3.34) gives

$$T(s) = \frac{\Omega(s)}{\Omega_{com}(s)} = \frac{\hat{K}_m\left(K_P + \frac{K_I}{s} + K_D s\right)}{\tau_m s + 1 + \hat{K}_m\left(K_P + \frac{K_I}{s} + K_D s\right)}, \tag{3.36}$$

$$= \frac{\hat{K}_m(K_D s^2 + K_P s + K_I)}{(\hat{K}_m K_D + \tau_m)s^2 + (\hat{K}_m K_P + 1)s + \hat{K}_m K_I} . \tag{3.37}$$

The initial design starts with a P-I controller $(K_D = 0)$. The closed-loop transfer function reduces to

$$T(s) = \frac{\Omega(s)}{\Omega_{com}(s)} = \frac{\hat{K}_m(K_P s + K_I)}{\tau_m s^2 + (\hat{K}_m K_P + 1)s + \hat{K}_m K_I}, \tag{3.38}$$

and the characteristic polynomial is

$$\Delta(s) = \tau_m s^2 + (\hat{K}_m K_P + 1)s + \hat{K}_m K_I \,. \tag{3.39}$$

A root-locus plot consists of the roots of $\Delta(s)$ as the control parameter K_C varies from zero to ∞. For a negative feedback control system with open-loop gain $K_P G(s)H(s)$, the characteristic equation is

$$\Delta(s) = 1 + K_P G(s)H(s) = 0 \,. \tag{3.40}$$

Rearranging terms in Eq (3.39) and setting it equal to zero,

$$\Delta(s) = s(\tau_m s + 1) + \hat{K}_m K_I + K_P \hat{K}_m s = 0 \,. \tag{3.41}$$

Dividing Eq (3.41) by $s(\tau_m s + 1) + \hat{K}_m K_I$ gives

$$1 + K_P \left[\frac{\hat{K}_m s}{s(\tau_m s + 1) + \hat{K}_m K_I} \right] = 0 \tag{3.42}$$

Comparing Eqs (3.40) and (3.42), the equivalent open-loop function for plotting a root-locus is

$$G(s)H(s) = \left[\frac{\hat{K}_m s}{s(\tau_m s + 1) + \hat{K}_m K_I} \right]. \tag{3.43}$$

Using the average DC motor parameter values $(\hat{K}_m)_{ave} = 32.08 \ \dfrac{\text{rpm}}{\text{\% change in duty cycle}}$, $(\tau_m)_{ave} = 0.161 \text{ sec}$ and choosing the integral constant $K_I = 1\,\%$, produces the root-locus plot shown in Figure 19.

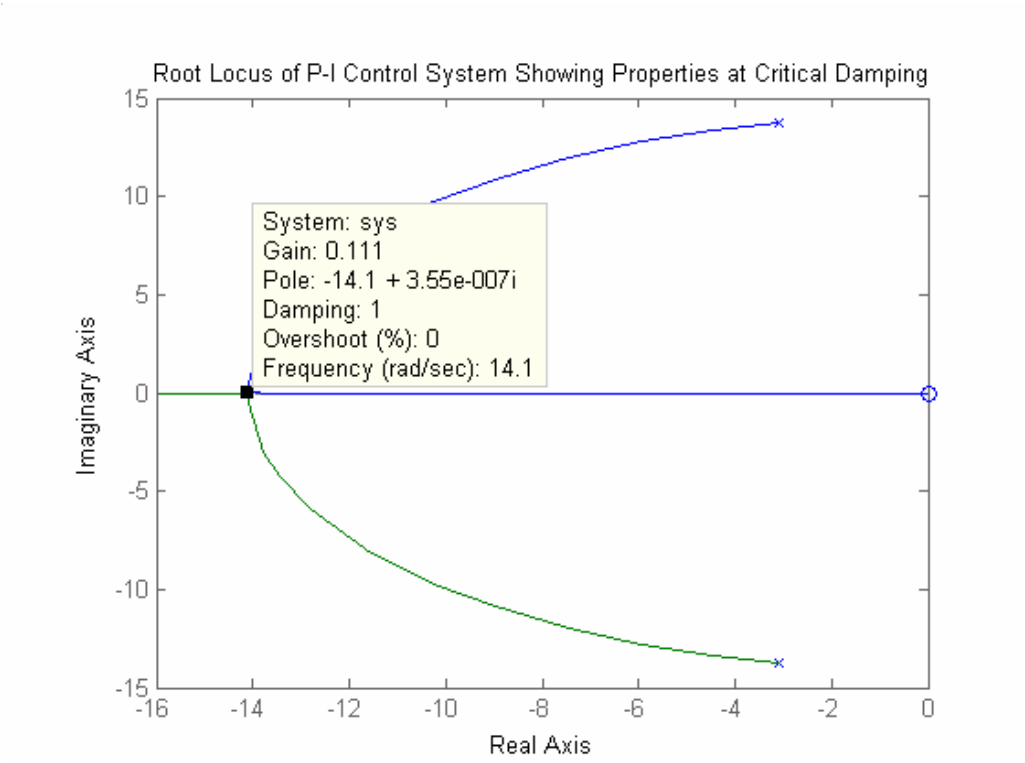

Figure 19: Root-locus for P-I $(K_I = 1)$ Control System Showing Critical Damping (ζ = 1)

Dividing the characteristic polynomial in Eq (3.39) by τ_m and equating the result to the standard form of a second order system characteristic polynomial gives

$$s^2 + \frac{(\hat{K}_m K_P + 1)}{\tau_m} s + \frac{\hat{K}_m K_I}{\tau_m} = s^2 + 2\zeta\omega_n s + \omega_n^2 \,. \tag{3.44}$$

Solving for the closed-loop system natural frequency and damping ratio yields

$$\Rightarrow \qquad \omega_n = \left(\frac{\hat{K}_m K_I}{\tau_m} \right)^{1/2}, \quad \zeta = \frac{\hat{K}_m K_P + 1}{2(\hat{K}_m K_I \tau_m)^{1/2}} \,. \tag{3.45}$$

The closed-loop system responds faster when the characteristic roots are complex $(0 < \zeta < 1)$ making the system underdamped. The system is critically damped $(\zeta = 1)$ with a

double pole on the real axis when the roots of Eq (3.44) are real and equal. To find the controller gain at critical damping, K_{crit} :

$$\zeta = \frac{\hat{K}_m K_{crit} + 1}{2(\hat{K}_m K_I \tau_m)^{1/2}} = 1 \quad \Rightarrow \quad K_{crit} = \frac{2(\hat{K}_m K_I \tau_m)^{1/2} - 1}{\hat{K}_m}. \tag{3.46}$$

Substituting the values $\hat{K}_m = (\hat{K}_m)_{ave} = 32.08$ rpm per % duty cycle, $\tau_m = (\tau_m)_{ave} = 0.161$ sec and $K_I = 1$ % in Eq (3.46) results in $K_{crit} = 0.1106$. This is in agreement with the Gain value of 0.111 shown in Figure 19. The open loop system corresponds to $K_P = 0$ where the two loci branches begin. Hence, the closed-loop system is underdamped provided $0 < K_P < 0.1106$.

A common design value for the damping ratio of a second order control system is $\zeta_{des} = 0.707$. Figure 20 shows the required controller gain to achieve an underdamped system with $\zeta = 0.707$ is $K_P = 0.0691$. This is easily checked by solving for ζ in Eq (3.45) with $K_P = 0.0691$. Hence, this gives

$$\zeta = \frac{\hat{K}_m K_P + 1}{2(\hat{K}_m K_I \tau_m)^{1/2}} = \frac{32.08(0.0691) + 1}{2[32.08(1)0.161]^{1/2}} = 0.707. \tag{3.47}$$

The loci branches in Figures 19 and 20 where the roots are complex are arcs of a circle with radius equal to the natural frequency ω_n. Hence, the natural frequency of the underdamped control system is independent of the damping ratio chosen for the design value. It is obtained from

$$\omega_n = \left(\frac{\hat{K}_m K_I}{\tau_m} \right)^{1/2} = \left[\frac{32.08(1)}{0.161} \right]^{1/2} = 14.1057 \text{ rad/sec} \tag{3.48}$$

and is in agreement with the value shown in Figures 17 and 18.

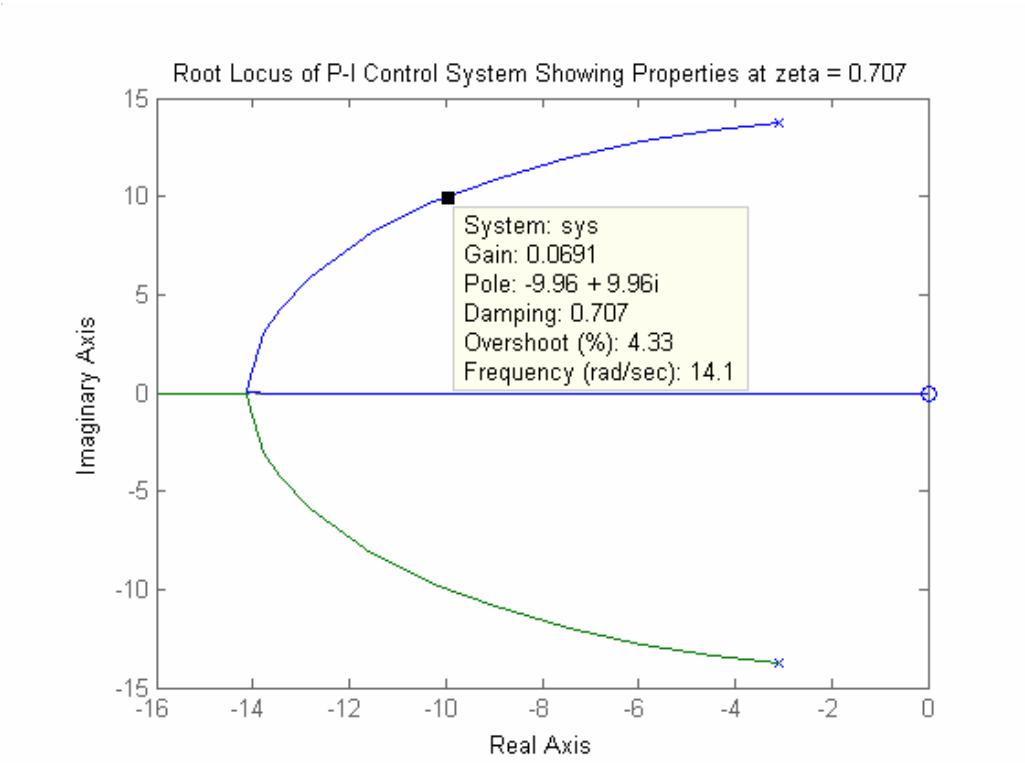

Figure 20: Root-locus Showing Point at Design Conditions (ζ = 0.707)

The exponential damping term in the transient response of the control system is $e^{-\zeta_{des}\omega_n t}$ and therefore the time constant of the exponential envelope is $1/\zeta_{des}\omega_n = 0.1003$ sec which is less than the average time constant $(\tau_m)_{ave} = 0.161$ sec of the open-loop motor. This implies the control system responds faster than the motor operating open-loop without feedback control. The control system response exhibits little overshoot (4.33%) and most importantly has zero offset (steady-state error) in response to a step change in command speed.

The predicted step responses of the open-loop and closed-loop control systems to a commanded input of 2000 rpm are shown in Figure 21. The transient response times agree with the predicted values $5(\tau_m)_{ave} = 0.8062$ sec and $5 \times 1/\zeta_{des}\omega_n = 0.5014$ sec.

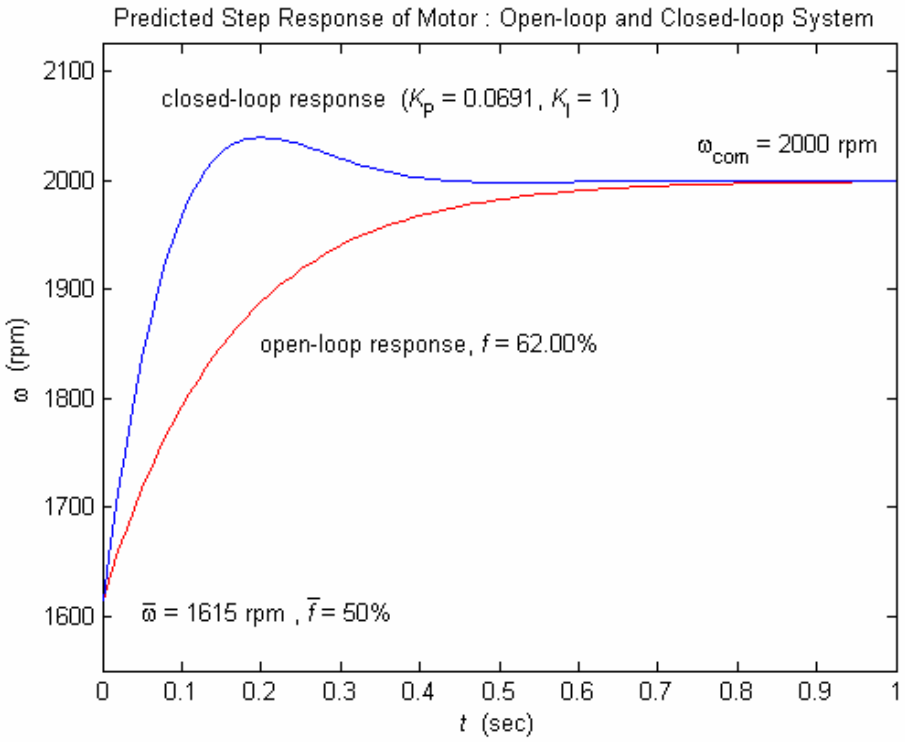

Figure 21: Step Response of Open-loop Motor and Closed-loop System

Converting from Continuous to Discrete Controller

Once an acceptable continuous controller transfer function is known, a discrete approximation for use with a digital control system running at $1/T$ Hz is determined using the bilinear transform. [Kuo] The discrete approximation to a continuous P-I-D controller is

$$D(z) = G_C(s)\Big|_{s \leftarrow \frac{2}{T}\left(\frac{z-1}{z+1}\right)}, \tag{3.49}$$

$$\text{and} \qquad = K_P + \frac{K_I}{s} + K_D s\Big|_{s \leftarrow \frac{2}{T}\left(\frac{z-1}{z+1}\right)}. \tag{3.50}$$

The resulting digital compensator is

$$D(z) = \frac{\left(K_P + 0.5K_I T + \frac{2K_D}{T}\right)z^2 + \left(K_I T - \frac{4K_D}{T}\right)z - K_P + 0.5K_I T + \frac{2K_D}{T}}{z^2 - 1}. \tag{3.51}$$

A block diagram of the digital control system is shown in Figure 22. The continuous motor speed $\omega(t)$ is sampled at times $0, T, 2T, 3T, ..., kT$ and converted to a discrete value $\omega(kT)$ or $\omega(k)$ for short.

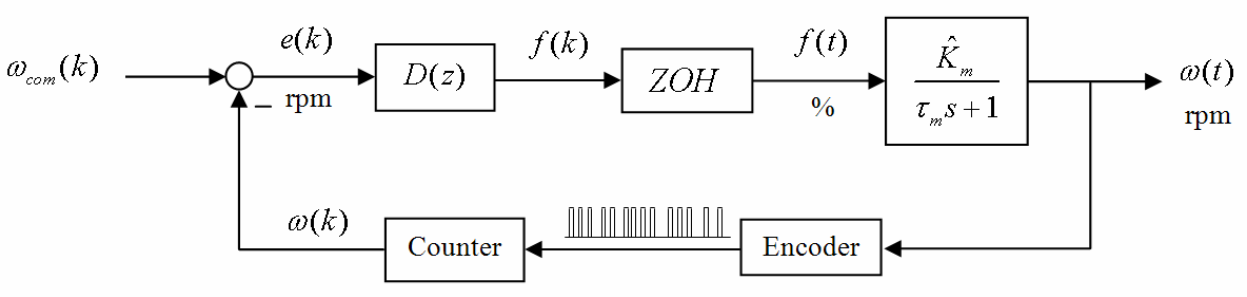

Figure 22: Block Diagram of Motor Speed Digital Control System

The duty cycle $f(k)$ is computed at the same times in the digital controller based on a difference equation relating $f(k)$ and $e(k)$. The difference equation is obtained from Eq (3.51) as follows.

$$D(z) = \frac{F(z)}{E(z)} = \frac{b_0 z^2 + b_1 z + b_2}{z^2 - 1}, \tag{3.52}$$

where $$b_0 = K_P + 0.5K_I T + \frac{2K_D}{T}, \tag{3.53}$$

$$b_1 = K_I T - \frac{4K_D}{T}, \tag{3.54}$$

and $$b_2 = -K_P + 0.5K_I T + \frac{2K_D}{T}. \tag{3.55}$$

Multiplying the numerator and denominator of Eq (3.52) by z^{-2} and then cross-multiplying terms results in Eqs (3.56) and (3.57) below gives:

$$\frac{F(z)}{E(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - z^{-2}}, \tag{3.56}$$

$$(1-z^{-2})F(z)=(b_0+b_1z^{-1}+b_2z^{-2})E(z). \tag{3.57}$$

Taking the inverse z-transforms of both sides and solving for $f(k)$ results in the control algorithm difference equation for a P-I-D controller,

$$f(k)=f(k-2)+b_0e(k)+b_1e(k-1)+b_2e(k-2),\quad k=0,1,2,3,... \tag{3.58}$$

The discrete values $f(k),\ k=0,1,2,....$ are converted to the piecewise continuous signal $f(t)=f(k),\ kT\le t<(k+1)T$ by the *ZOH* bock in Figure 22.

The motor is assumed to be spinning at constant speed before the occurrence of a step change in commanded speed. The initial conditions $e(-1), e(-2)$ and $f(-2)$ must be specified each time a step change occurs. Figure 23 illustrates the case where the initial steady-state is $(f=\bar{f}, \omega=\bar{\omega})$ when the commanded speed changes from $\bar{\omega}$ to ω_1 at $k=0$. The previous errors $e(-1)$ and $e(-2)$ are both zero and $f(-2)=\bar{f}$.

After the transient response dies out, a new steady-state $(f=f_1, \omega=\omega_1)$ is reached. Sometime later, the command speed changes to ω_2. The time of the new commanded speed can be treated as $k=0$. The commanded speed and the actual speed are both equal to ω_1 at $k=-1,-2$ making $e(-1)=e(-2)=0$. The duty cycle initial condition is $f(-2)=f_1$.

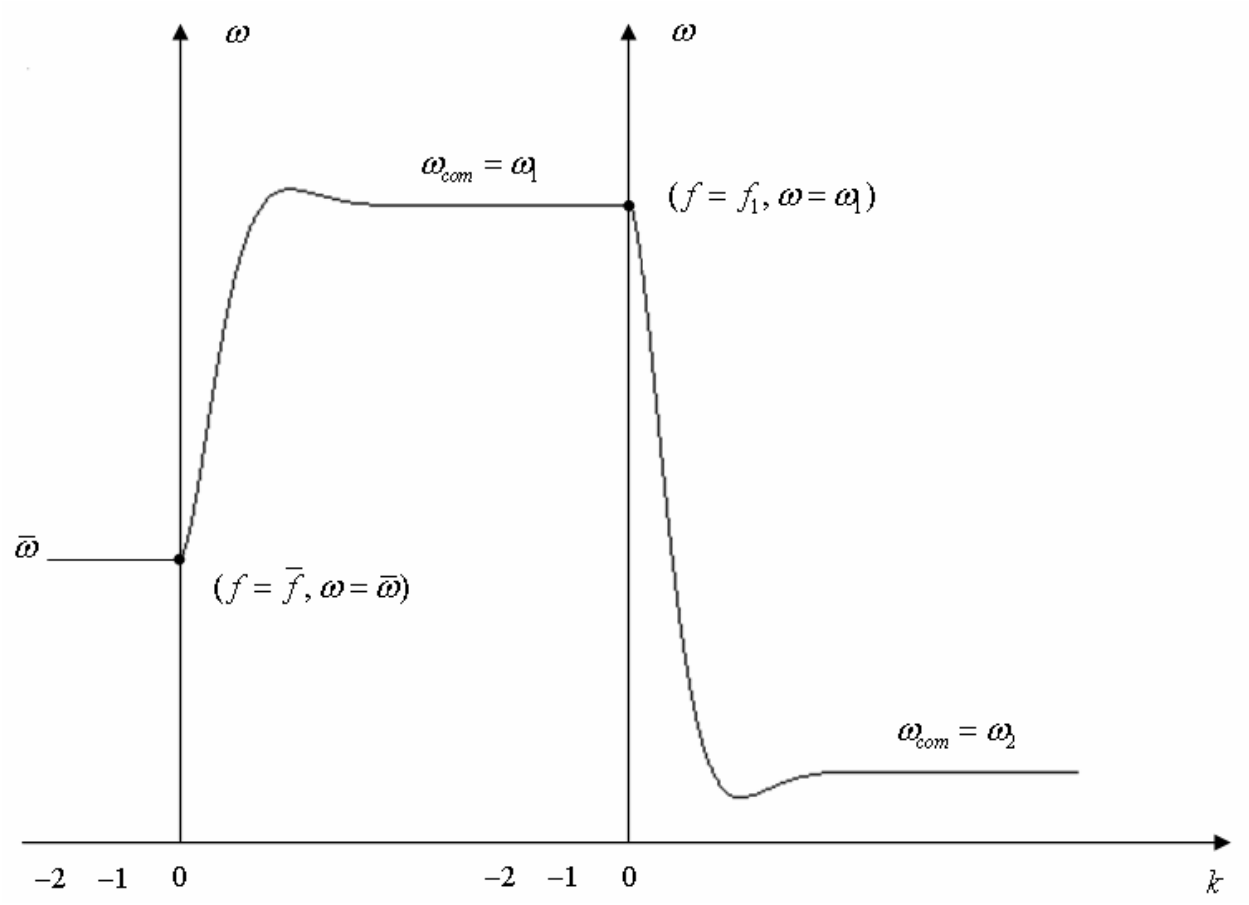

Figure 23: Closed-loop Response to Consecutive Changes in Command Speed

In the next chapter, a number of experiments are performed to model the motor and design a control system for it.

CHAPTER FOUR: FINDINGS

Initial Testing

Ten separate step tests were performed on the motor. Each trial lasted four seconds: two seconds of 50% duty cycle and two seconds of another fixed duty cycle. The Simulink model used to carry out this test is shown in Figure 24. The block that looks like a circuit board is used to configure the build options for the model. The switch in the model passes one of the two inputs on to the PWM block. When the step attached to its control line changes from zero to one at two seconds, the switch toggles from 50% input to the other input. During the complete four second trial, encoder readings are sampled every 20 ms and stored in an RTDX buffer. After all the data is collected, it can be retrieved by the host computer.

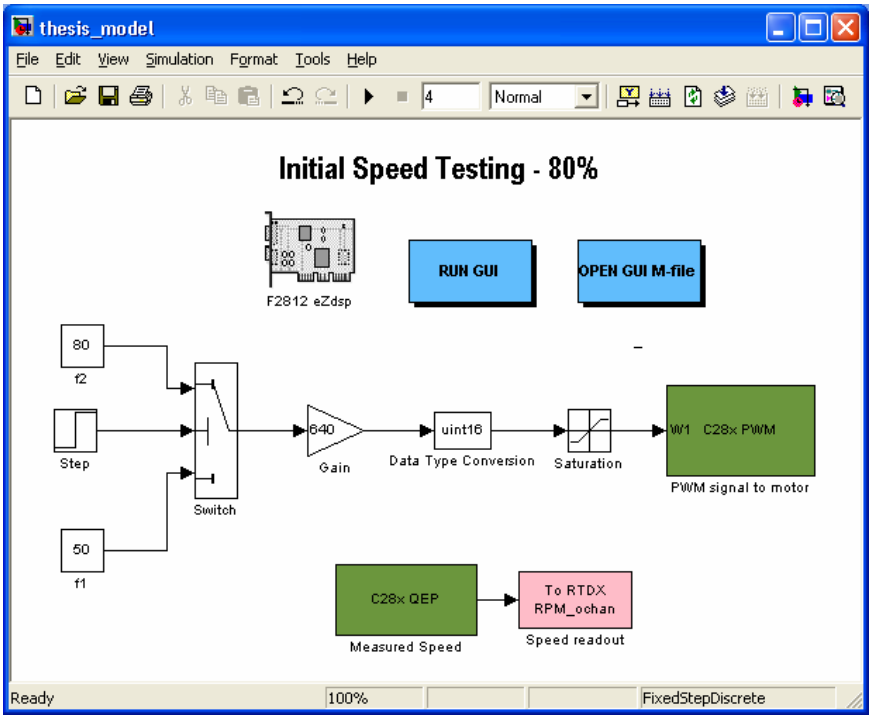

Figure 24: Step Test Simulink Model

Below are the results of the 10 trials in Figures 25 through 34. Before each trial was run, the DSP was halted and the motor was at full speed. Therefore, for the first few milliseconds of each graph, the speed ramps down to roughly 1615 RPM. At two seconds, the commanded duty cycle changes to a new value, which can range from 0% to 100%. The measured speed takes some time to approach the new value. Theoretically, it should take the motor the same amount of time to reach any new commanded speed.

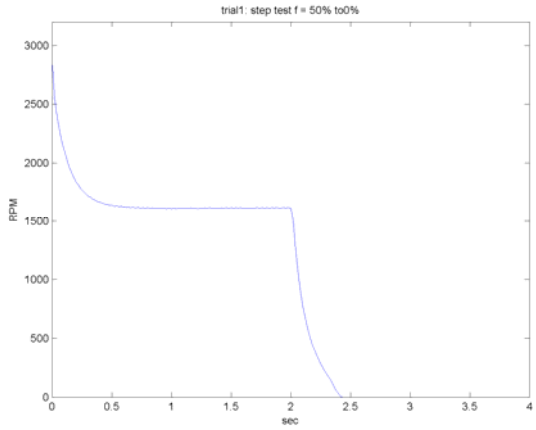

Figure 25: Step Test Trial 1, f = 0%

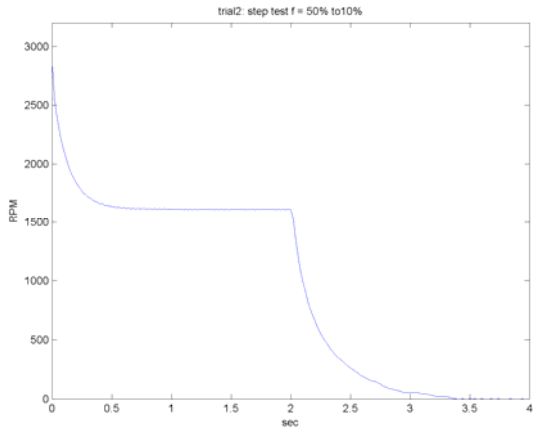

Figure 26: Step Test Trial 2, f = 10%

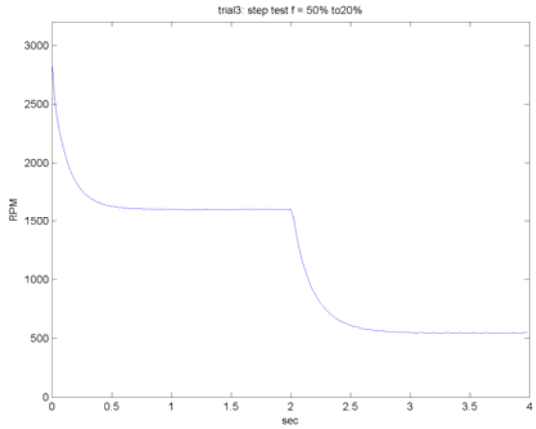

Figure 27: Step Test Trial 3, f = 20%

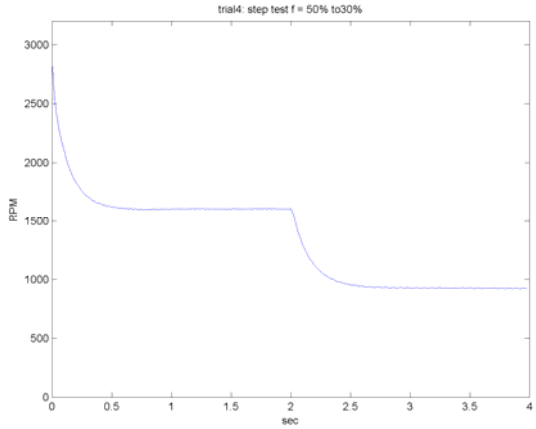

Figure 28: Step Test Trial 4, f = 30%

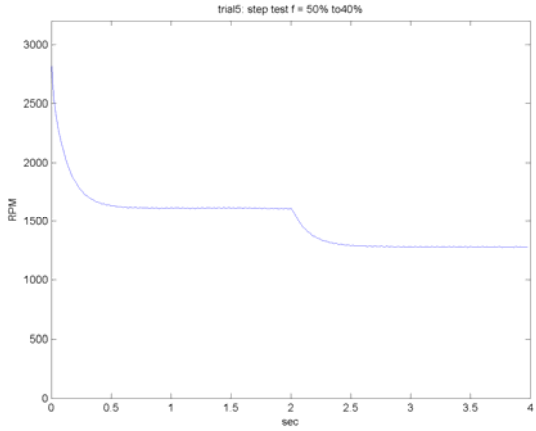

Figure 29: Step Test Trial 5, f = 40%

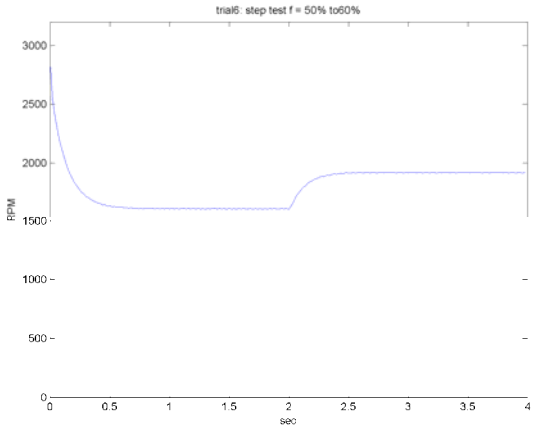

Figure 30: Step Test Trial 6, f = 60%

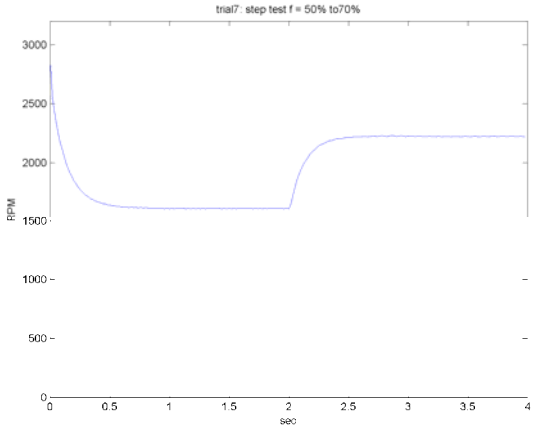

Figure 31: Step Test Trial 7, f = 70%

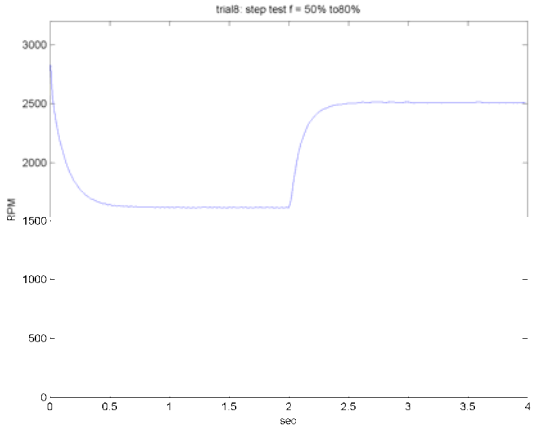

Figure 32: Step Test Trial 8, f = 80%

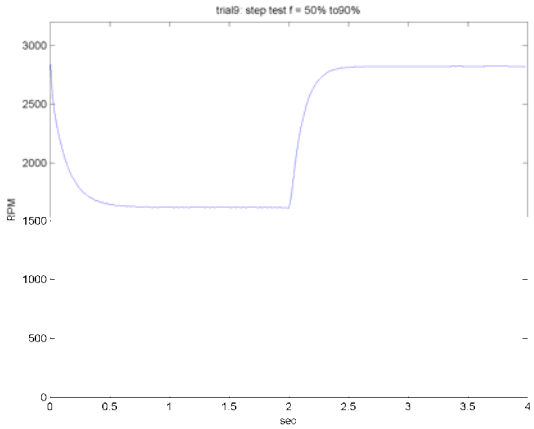

Figure 33: Step Test Trial 9, f = 90%

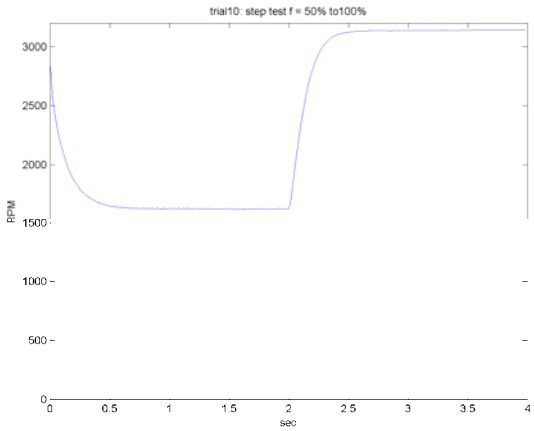

Figure 34: Step Test Trial 10, f = 100%

In each test, regardless of the change in commanded duty cycle, the motor required roughly 800 ms to transition from 1615 RPM to the new speed. The average was taken from all the trials. This value is very close to the expected value, i.e. five times the motor's time constant (805 ms).

Open Loop

The open-loop testing was carried out in two trials. Each lasted four seconds. Both consisted of a commanded speed of 1615 RPM for the first two seconds. Trial 1 stepped up the commanded speed to 2300 RPM. Trial 2 stepped down the commanded speed to 1000 RPM.

Figure 35 shows the Simulink model used to carry out these tests. This model is similar to the one for initial testing, but has added the open-loop gain and is referenced to a PWM duty cycle of 50%.

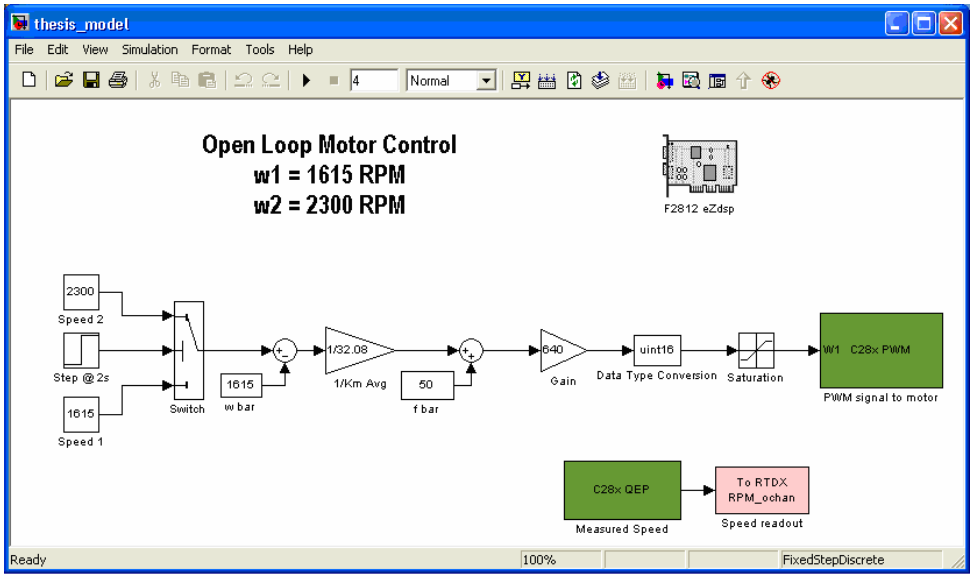

Figure 35: Open-loop Simulink Model

The step response for trial 1 is shown in Figure 36. The step response for trial 2 is shown in Figure 37. The theoretical open-loop responses for each trial are plotted as well as the measured results. In both cases, they match very closely. However, there is a small error, since the open-loop expression doesn't exactly represent the motor's response. In addition, for the open-loop response to be successful, the load on the motor cannot change.

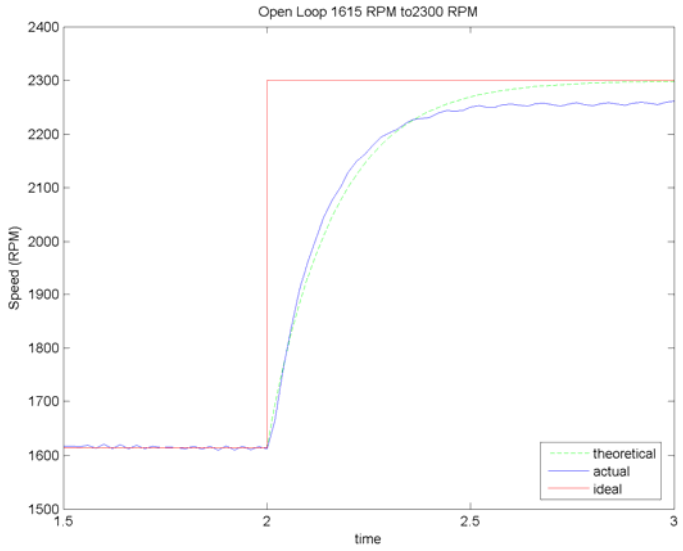

Figure 36: Open-loop Response ω = 1615 rpm to ω = 2300 rpm (close-up)

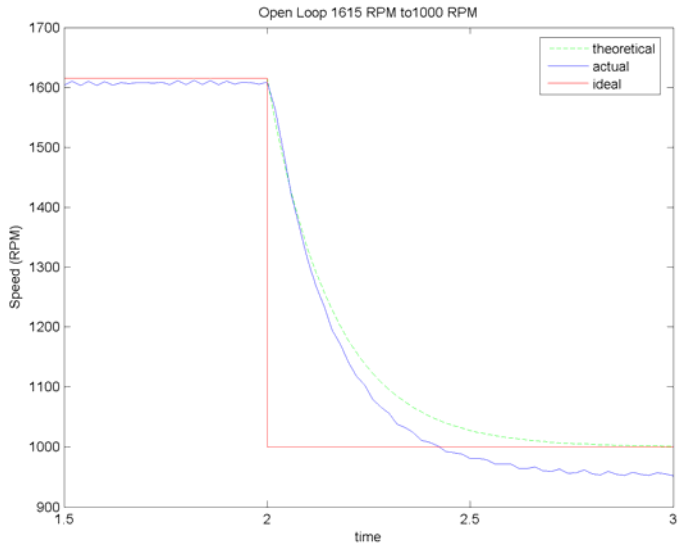

Figure 37: Open-loop Response ω = 1615 rpm to ω = 1000 rpm (close-up)

Closed Loop

The simulated closed loop response was modeled via two Simulink models, an analog version and a digital version. The analog control system is simulated in Figure 38. Notice the continuous PID Controller block. This system will have the best possible response.

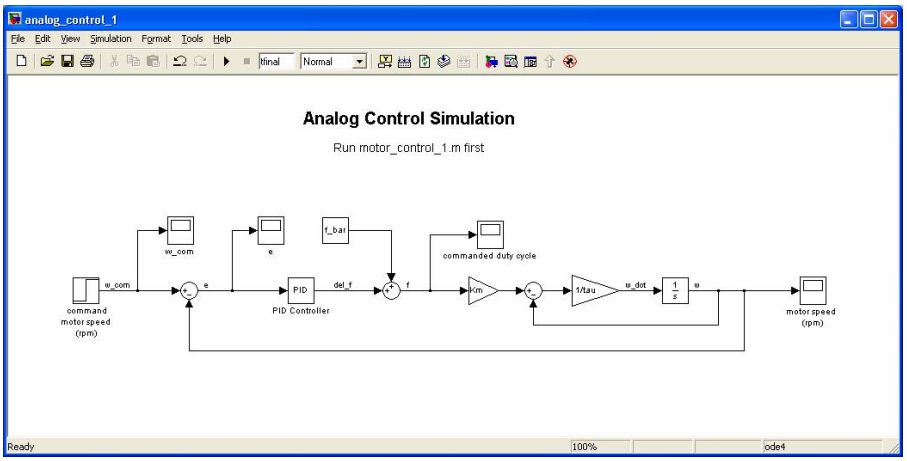

Figure 38: Analog Control Simulation

The digital control system is modeled in Figure 39. The analog PID block has been replaced by a discrete transfer function block. The response of this system will not be as fast as the analog, since it can only update at discrete intervals.

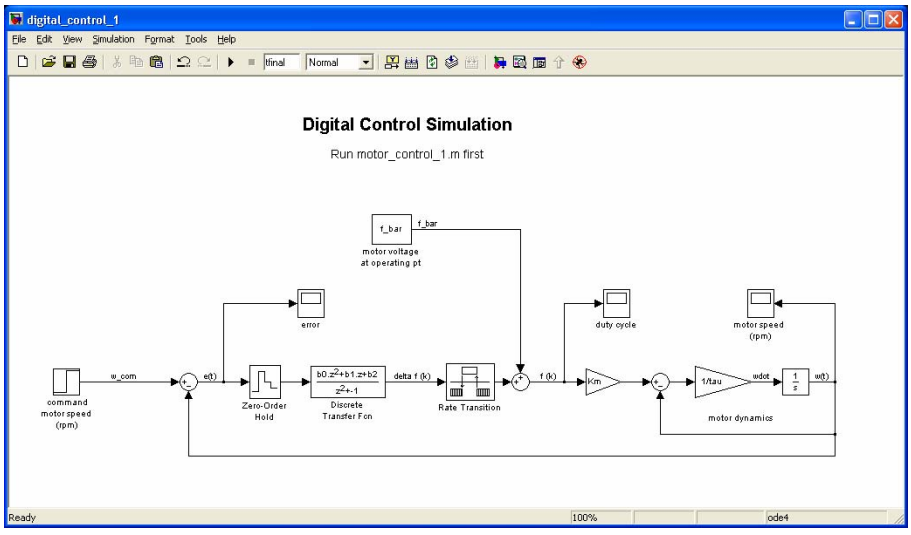

Figure 39: Digital Control Simulation

The closed-loop control system was carried out via the Simulink model shown below in Figure 40. Six trials were carried out. In each, the proportional gain Kp was held constant at 0.02 while the integral gain Ki was incremented by 0.2, ranging from 0 to 1.

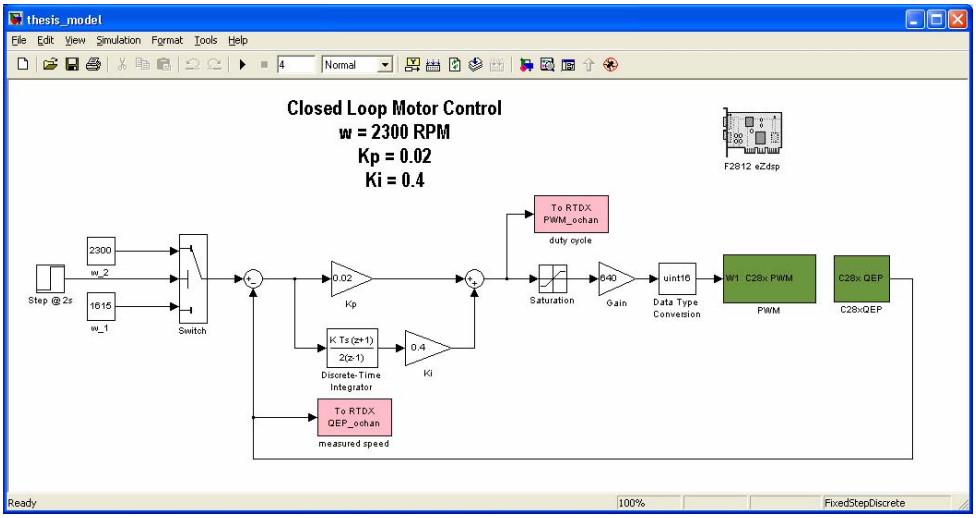

Figure 40: Closed-loop Simulink Model

Figures 41 through 46 show the complete step responses for each trial. The first trial shows that with no integral gain, the motor does not approach the commanded speed. In the remaining trials, the motor successfully approaches the commanded speed. With little integral gain, such as in Figure 42, the system is overdamped. With higher integral gain, the system becomes more underdamped, as can be seen in Figure 46. This provides quicker rise times, at the expense of more ringing, when compared to Figure 42.

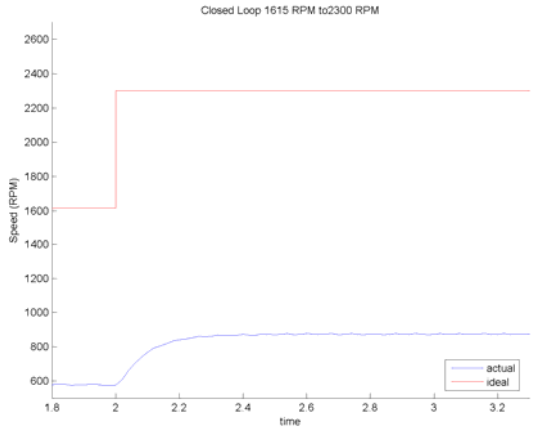

Figure 41: Closed-loop Response Kp = 0.02 Ki = 0

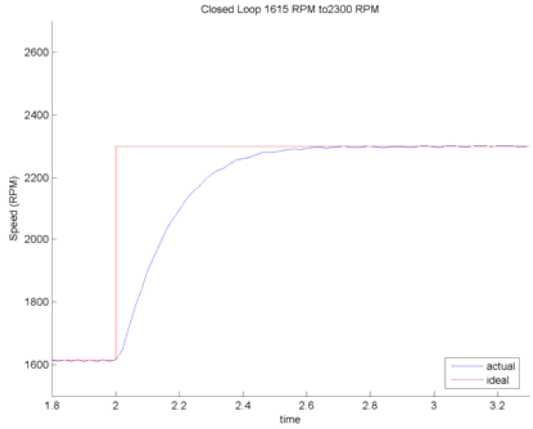

Figure 42: Closed-loop Response Kp = 0.02 Ki = 0.2

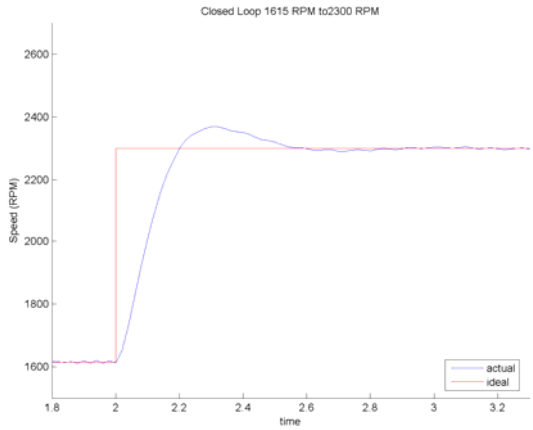

Figure 43: Closed-loop Response Kp = 0.02 Ki = 0.4

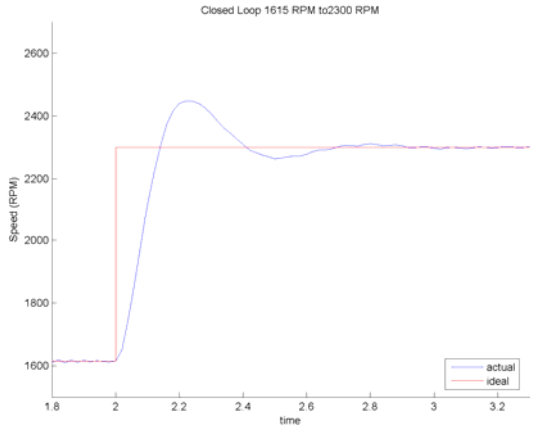

Figure 44: Closed-loop Response Kp = 0.02 Ki = 0.6

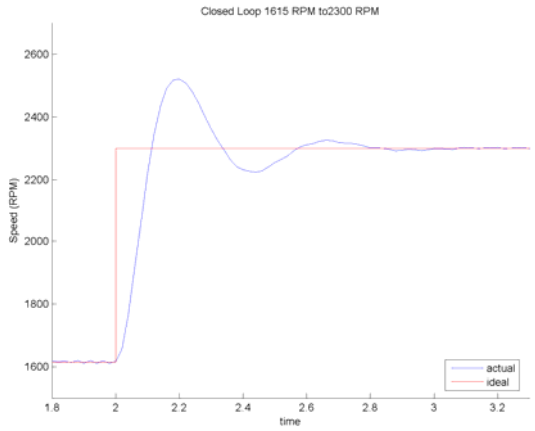

Figure 45: Closed-loop Response Kp = 0.02 Ki = 0.8

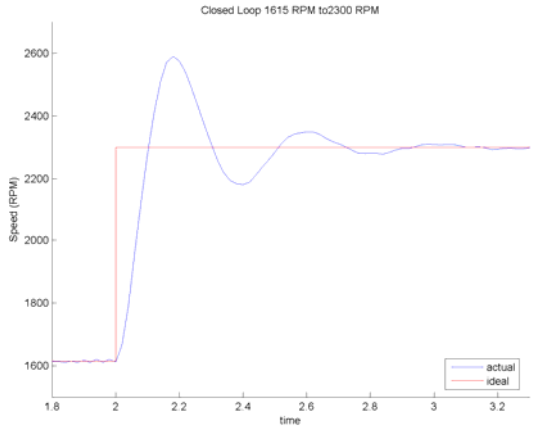

Figure 46: Closed-loop Response Kp = 0.02 Ki = 1

In Chapter 3, the analog control system was designed to achieve a damping ratio of 0.707. The controller parameters were $K_P = 0.0691$ and $K_I = 1$. This system was tested two consecutive times, to investigate the accuracy and repeatability of the measurement system. The results are shown in Figure 47. Notice there is no ringing, very little overshoot, and a relatively quick response time. Furthermore, the measured responses are nearly identical.

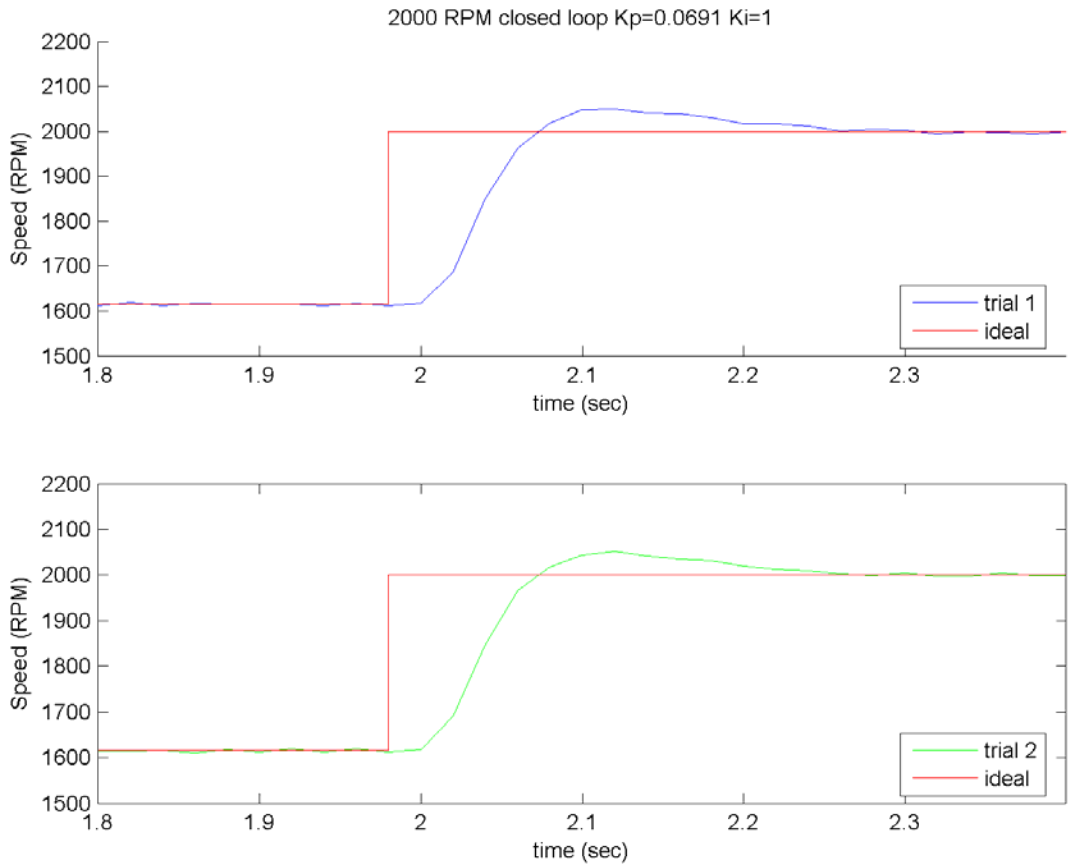

Figure 47: Closed-loop Kp = 0.0691 Ki = 1

CHAPTER FIVE: CONCLUSION

This experiment was meant to verify that using MATLAB and Simulink to develop an embedded control system offers many advantages over the typical method of hand translation from pseudocode to source code. There is an inherent advantage in using Simulink to model the control system. It saves time and effort, allowing the engineer to design the system in a straightforward manner, rather than wasting time writing source code from scratch. Only recently has Simulink had the capability to directly target hardware.

To demonstrate all of these techniques, a motor control sytem was designed using Simulink and the F2812 DSP. The F2812 had the appropriate peripherals to properly interface with the motor and encoder. In addition, the Embedded Target for TI C2000 DSPs blockset in Simulink allows for simple software interfacing to these peripherals. It was now possible to create Simulink models for motor testing, open-loop system design, as well as closed-loop system design without writing any lines of code.

It was soon determined that some code was necessary for acquiring data from the DSP. RTDX can send data from the DSP to the host computer while the DSP is running. A few m-files are used to operate the DSP, load different project files, configure RTDX channels, and manage the exchange of information between DSP and host PC. Originally, the intention was to have the DSP send speed data in real time. However, RTDX read and write commands don't take exactly the same amount of time to execute for each function call. This made reading and storing data erratic, as well as unnecessarily slow. Instead, the data recovery scheme was changed so that the DSP reported its speed readings only after the experiment trial had

completed. This way, each speed reading was exactly 20 ms apart. This allowed for the highest precision, as well as a standard sample period for speed readings from every trial.

Using this method of recording speed worked very well. Accurate readings of the response of every system were available. When the sample rate of the system was increased to 10 ms, there were serious problems. When sampling from the quadrature encoder module, the system was unable to accurately read speed measurements. An incredible amount of noise was now present in the plot of speed data. The fastest sampling period that would still work was determined to be 20 ms. In future experiments, the reason for this limit could be determined. Perhaps, at 1000 pulses per revolution, the encoder had too much resolution for the DSP to accurately track at the speeds the motor was turning. Perhaps the DSP itself was not running at its maximum system clock frequency.

Some more work could be done to analyze the differences between the continuous and discrete closed-loop systems. It's important to model the motor very accurately. Experimentation could let the user determine how fast the sampling frequency of the discrete system must be to approach the performance of the continuous system.

MATLAB's graphical user interface capabilities were very useful as well. There is a separate GUI for each stage of development. For initial testing, a GUI allowed the user to command any PWM duty cycle from 0 to 100 by grabbing and moving a simple slider. A real-time scrolling plot of the measured speed was available in the same GUI. Other GUIs were created for the open and closed-loop systems. They enabled the user to command a particular speed in RPM. Real-time plots showed measured speed of the motor as well as PWM duty cycles commanded by the control system, all in real-time. All of these GUIs were very useful when used in conjunction with the regular trials.

Lots of future work can be done to exploit the advantages of MATLAB and Simulink and their hardware targeting capabilities. The F2812 has other peripherals that would make it well-suited to other embedded applications. These include on board analog to digital converters, a CAN module, digital inputs and outputs, and on-chip memory. Simulink also has an Embedded Target for TI C6000 DSPs, which allows it to target TI's high performance C6000 family of DSPs. These are well suited for advanced high-level algorithm development, especially multimedia. The DM642, for example, has built in video and audio ports. Simulink gives the user the power to target these high-performance DSPs with a minimal amount of effort, allowing for more rapid prototyping and more efficient use of development time.

APPENDIX A: SOFTWARE LISTING

record_step_tests.m

This function loads the compiled executable in the DSP, configures RTDX, and runs a four-second trial. After the trial is finished, the host computer reads the collected data from the DSP, plots it, and saves it.

```
function speed = record_step_tests()
% Execute step test, then retrieve speed data from DSP

clc
clear
close all
format compact

PWM = 80;

cc = prepare_target_and_load();
configure(cc.rtdx,1024,1);  % configure RTDX, 1 buffer of 1024 bytes
open(cc.rtdx, 'RPM_ochan', 'r');
enable(cc.rtdx, 'RPM_ochan');
enable(cc.rtdx);

numMsgs = cc.rtdx.msgcount('RPM_ochan');
cc.rtdx.flush('RPM_ochan', numMsgs);

run(cc)
clc
pause(5)          % wait for DSP to finish

t = 0:0.02:3.98;  % t = 0 to 3.98 sec in 20 ms increments
z = 1;
while cc.rtdx.msgcount('RPM_ochan')>1
    speed(z) = readmsg(cc.rtdx,'RPM_ochan','double');
    z = z + 1;
end
cc.halt

figure; plot(t,speed); title(strcat(num2str(PWM),' % duty cycle'))
xlabel('sec'); ylabel('RPM')
XMIN = 0;
XMAX = 4;
YMIN = 0;
YMAX = 3500;
axis([XMIN XMAX YMIN YMAX])
print('-dpng', '-r200', strcat(num2str(PWM), '_%_duty_cycle'))
```

plot_steps.m

This function loads a MAT file with the collected step data. It plots all the step tests and saves them.

```
% plot_steps.m
% load all 10 step test data sets, plot, and save picture

close all
clc

load step_tests

XMIN = 0;
XMAX = 4;
YMIN = 0;
YMAX = 3200;

for i = [1:10]
    if i<=5
        PWM = (i-1)*10;
    else
        PWM = i*10;
    end
    figure
    plot(t,speed(i,:))
    title(strcat('trial',num2str(i),': step test f = 50% to',num2str(PWM),
'%'))
    xlabel('sec'); ylabel('RPM')
    axis([XMIN XMAX YMIN YMAX])
    if i==10
        filename = strcat('step_test',num2str(i));
    else
        filename = strcat('step_test0',num2str(i));
    end
    print('-dpng', '-r200', filename)
end
```

record_open_loop.m

This function loads the compiled executable in the DSP, configures RTDX, and runs a four-second trial. After the trial is finished, the host computer reads the collected data from the DSP, plots it, and saves it. The theoretical open-loop response is calculated and compared to the measured response.

```
function speed = record_open_loop()
% Execute open-loop test, retrieve data, plot, and save

clc
clear
close all
format compact

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Program DSP and execute test
cc = prepare_target_and_load ();
configure(cc.rtdx,1024,1);  % configure RTDX, 1 buffer of 1024 bytes
open(cc.rtdx, 'RPM_ochan', 'r');
enable(cc.rtdx, 'RPM_ochan');
enable(cc.rtdx);

run(cc)
clc
pause(5)            % wait for test to finish

t = 0:0.02:3.98;    % t = 0 to 3.98 sec in 20 ms increments
z = 1;
while cc.rtdx.msgcount('RPM_ochan') > 1
    % collect speed readings for entire run
    speed(z) = readmsg(cc.rtdx,'RPM_ochan','double');
    z = z + 1;
end
cc.halt

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate open-loop response

f_bar = 50;
w_bar = 1615;
w_2 = 2300;
tau = 0.161;
ave_Km = 32.08;
del_f = (w_2 - w_bar)/ave_Km;
f_2 = f_bar + del_f;

for i=1:200
    if t(i)<2
        wi(i) = w_bar;
    else
```

```
        wi(i) = w_bar + ave_Km * del_f * (1 - exp(-(t(i)-2)/tau));
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot results

plot(t,wi,'g--')                        % plot theoretical open-loop response
hold on

plot(t,speed,'b')                       % plot measured speed

plot([t(1) t(101)],[w_bar w_bar],'r')   % plot ideal step response
plot([t(101) t(101)],[w_bar w_2],'r')
plot([t(101) t(200)],[w_2 w_2],'r')

title(strcat('Open Loop 1615 RPM to ', num2str(w_2), ' RPM'))
xlabel('time'); ylabel('Speed (RPM)')
legend('theoretical','actual','ideal')
XMIN = 0;
XMAX = 4;
YMIN = 0;
YMAX = 3500;
axis([XMIN XMAX YMIN YMAX])

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Save results

filename = strcat(num2str(w_2), '_RPM_open_loop');
print('-dpng', '-r200', filename)
save(filename,'t','wi','speed')
```

replot_open_loop.m

This m-file lets the user plot collected data from the open-loop tests without having to rerun the trials. It loads a MAT file with the open-loop results, then plots and saves them.

```
% replot_open_loop.m
% Loads data from mat file and plots it.  This is especially useful for
% changing the axes of a plot without having to rerun the test on the motor

clc
clear
close all
format compact

w_bar = 1615;
w_2 = 2300;

filename = strcat(num2str(w_2), '_RPM_open_loop');
load(filename)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot results

plot(t,wi,'g--')                        % plot theoretical open-loop response
hold on

plot(t,speed,'b')                       % plot measured speed

plot([t(1) t(101)],[w_bar w_bar],'r')   % plot ideal step response
plot([t(101) t(101)],[w_bar w_2],'r')
plot([t(101) t(200)],[w_2 w_2],'r')

title(strcat('Open Loop 1615 RPM to ', num2str(w_2), ' RPM'))
xlabel('time'); ylabel('Speed (RPM)')
legend('theoretical','actual','ideal','Location','SouthEast')
XMIN = 1.5;
XMAX = 3;
YMIN = 1500;
YMAX = 2400;
axis([XMIN XMAX YMIN YMAX])

filename = strcat(num2str(w_2), '_RPM_open_loop_zoomed');
print('-dpng', '-r200', filename)
```

record_closed_loop.m

This function loads the compiled executable in the DSP, configures RTDX, and runs a four-second trial. After the trial is finished, the host computer reads the collected data from the DSP, plots it, and saves it. The theoretical closed-loop response is calculated and compared to the measured response.

```
function speed = record_closed_loop()
% Execute closed-loop test, retrieve data, plot, and save

clc
clear
close all
format compact

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Program DSP and execute test
cc = prepare_target_and_load ();
configure(cc.rtdx,1024,2);  % configure RTDX, 2 buffers of 1024 bytes each
open(cc.rtdx, 'QEP_ochan', 'r');
open(cc.rtdx, 'PWM_ochan', 'r');
enable(cc.rtdx, 'QEP_ochan');
enable(cc.rtdx, 'PWM_ochan');
enable(cc.rtdx);

run(cc)
clc
pause(5)            % wait for test to finish

t = 0:0.02:3.98;    % t = 0 to 3.98 sec in 20 ms increments
z = 1;
while cc.rtdx.msgcount('QEP_ochan') > 1
    % collect speed readings for entire run
    speed(z) = readmsg(cc.rtdx,'QEP_ochan','double');
    z = z + 1;
end
z = 1;
while cc.rtdx.msgcount('PWM_ochan') > 1
    % collect control system's calculated PWM values for entire run
    duty_cycle(z) = readmsg(cc.rtdx,'PWM_ochan','double');
    z = z + 1;
end
cc.halt

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate closed-loop response
Kp = 0.02;
Ki = 0.4;

f_bar = 50;
w_bar = 1615;
```

```
w_2 = 2300;
tau = 0.161;
ave_Km = 30.37;
del_f = (w_2 - w_bar)/ave_Km;
f_2 = f_bar + del_f;

% ave motor gain (rad/sec per % change in duty cycle)
ave_Km_rad_per_sec = (ave_Km/60)*2*pi;
n=ave_Km_rad_per_sec*[Kp Ki];
d=[tau ave_Km_rad_per_sec*Kp+1 ave_Km_rad_per_sec*Ki];
sys=tf(n,d);
del_w_com = w_2 - w_bar;
T=0:0.02:1.98;
%  step response of closed loop system to unit step input
[y,tt] = step(sys, T);
del_w = del_w_com*y;
w = w_bar + del_w;
% step response of closed-loop system
wi = [w_bar*ones([1 100]) w'];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot results

% top plot %%%%%%%%%%%%%%%%%%%%%%%%%%%%
subplot(2,1,1)
% plot(t,wi,'g--')                  % plot theoretical closed-loop response
hold on

plot(t,speed,'b')                         % plot measured speed

plot([t(1) t(100)],[w_bar w_bar],'r')   % plot ideal step response
plot([t(100) t(100)],[w_bar w_2],'r')
plot([t(100) t(200)],[w_2 w_2],'r')

title(strcat(num2str(w_2), ' RPM closed loop Kp=',num2str(Kp),'
Ki=',num2str(Ki)))
xlabel('time'); ylabel('Speed (RPM)')
legend('actual','ideal','Location','SouthEast')
XMIN = 0;
XMAX = 4;
YMIN = 0;
YMAX = 3500;
axis([XMIN XMAX YMIN YMAX])

% bottom plot %%%%%%%%%%%%%%%%%%%%%%%%%%
subplot(2,1,2)
plot(t,duty_cycle)  % plot measured PWM commands from control system
hold on

plot([t(1) t(100)],[f_bar f_bar],'r')   % plot ideal commanded RPM
plot([t(100) t(100)],[f_bar f_2],'r')
plot([t(100) t(200)],[f_2 f_2],'r')
YMAX = 100;
axis([XMIN XMAX YMIN YMAX])
```

```
xlabel('time'); ylabel('PWM Duty Cycle (%)')


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Save results

filename = strcat(num2str(w_2), '_RPM_closed_loop_Kp-',num2str(Kp*1000),'_Ki-
',num2str(Ki*1000));
print('-dpng', '-r200', filename)
save(filename,'t','wi','speed')
```

replot_closed_loop.m

This m-file lets the user plot collected data from the closed-loop tests without having to rerun the trials. It loads a MAT file with the closed-loop results, then plots and saves them.

```
% replot_closed_loop.m
% Loads data from mat file and plots it.  This is especially useful for
% changing the axes of a plot without having to rerun the test on the motor

clc
clear
close all
format compact

w_bar = 1615;
w_2 = 2300;

Kp = 0.02;
Ki = 0.4;


filename = strcat(num2str(w_2), '_RPM_closed_loop_Kp-',num2str(Kp*1000),'_Ki-
',num2str(Ki*1000));
load(filename)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot results

% plot(t,wi,'g--')                 % plot theoretical closed-loop response
hold on

plot(t,speed,'b')                        % plot measured speed

plot([t(1) t(101)],[w_bar w_bar],'r')    % plot ideal step response
plot([t(101) t(101)],[w_bar w_2],'r')
plot([t(101) t(200)],[w_2 w_2],'r')

title(strcat('Closed Loop 1615 RPM to ', num2str(w_2), ' RPM'))
xlabel('time'); ylabel('Speed (RPM)')
legend('actual','ideal','Location','SouthEast')
XMIN = 1.8;
XMAX = 3.3;
YMIN = 1500;
YMAX = 2700;
axis([XMIN XMAX YMIN YMAX])

filename = strcat(num2str(w_2), '_RPM_closed_loop_Kp-',num2str(Kp*1000),'_Ki-
',num2str(Ki*1000),'zoomed');
print('-dpng', '-r200', filename)
```

motor_model.m

This m-file is used to determine motor parameters. It also is used to design the desired closed-loop system.

```
% motor_model.m
% 1. plots generic first order motor step response
% 2. reads experimental motor data and calculates motor parameters
%    Km (rpm per % duty cycle) and tau (sec)
% 3. plots root-locus for P-I control
% 4. plots open and closed loop system step responses
% 5. plots operating characteristic for open-loop control

clc, clear all, close all
warning off

%%%%% 1. plot of generic motor step response
tau=1; K=5;
t0=0.5;
tfinal=6*tau;
ybar=1.5;

dt=tfinal/1000;
for i=1:1000;
    t(i)=(i-1)*dt;
    if t(i) < t0
        y(i)=ybar;
    else
        y(i)=ybar+K*(1-exp(-(t(i)-t0)/tau));
    end
end % for

plot(t,y)
hold on
axis([0 tfinal 0.5 ybar+K+0.5])
set(gca,'XTick',[ ],'YTick',[ ])
xlabel('\itt')
ylabel('\ity\rm(\itt\rm)')
title('Step Response of Motor with Time Constant \tau and Steady-state Gain
\itK_{\itm}')

tt=t0+tau;
yy=ybar+K*(1-exp(-1));
plot([0 tt],[yy yy],':k')
plot([tt tt],[0 yy],':k')
plot([0 t0],[ybar ybar],':k')
plot([t0 t0],[0 ybar],':k')
plot([0 tfinal],[ybar+K+0.05 ybar+K+0.05],':k')
text(0.45,0.25,'\itt_{\rm0}')
text(1.35,0.25,'\itt_{\rm0}+\tau')
text(-0.2,1.5,'\omega')
text(-0.19,1.70,'_')
```

```
text(-1.0,4.7,'\omega+0.63\itK_{\itm}\rm\Delta\itf')
text(-0.99,4.96,'_')
text(-0.8,6.5,'\omega+\itK_{\itm}\rm\Delta\itf')
text(-0.79,6.77,'_')
text(2.5,3,'\omega\rm(\itt\rm) = \omega \rm+ \itK_{\itm} \rm[1 - e^{ -
\rm(\itt - t_{\rm0}\rm) / \tau}\rm] \Delta\itf'      )
text(3.02,3.25,'_')
plot([t0 t0+tau],[0.8 0.8],'k')
text(0.95,1,'\tau')
text(2.5,2.4,'\Delta\itf \rm= \itf \rm- \itf')
text(3.12,2.67,'_')
text(2.5,5,'Steady-state Operating Pt:')
text(2.5,4.5,'\itV \rm= 12 volts, \itf \rm= 50 %')
text(2.52,4.8,'_')
text(3.56,4.79,'_')
text(2.5,4,'\omega \approx 1615 rpm')
text(2.515,4.22,'_')

%%%%% 2. read experimental data from motor step responses and determine
%%%%% motor parameters %%%%%
clear t
load step_tests_v6.mat
t=t(51:end);
speed=speed(:,51:end);
figure
subplot(2,2,1)
w1=speed(1,:);
plot(t,w1)
set(gca,'XTick',[1.5:0.5:3],'YTick',[0:400:1750])
axis([1.5 3 -100 1750])
ylabel('\omega \rm(rpm)')
title('Run #1, \Delta\itf \rm= -50%')
Km1=(min(w1)-max(w1))/(-50);

subplot(2,2,2)
w4=speed(4,:);
plot(t,w4)
set(gca,'XTick',[1.5:0.5:3],'YTick',[900:200:1700])
axis([1.5 3 850 1700])
ylabel('\omega \rm(rpm)')
title('Run #2, \Delta\itf \rm= -20%')
Km4=(min(w4)-max(w4))/(-20);

subplot(2,2,3)
w7=speed(7,:);
plot(t,w7)
set(gca,'XTick',[1.5:0.5:3],'YTick',[1500:200:2350])
axis([1.5 3 1500 2350])
xlabel('\itt \rm(sec)')
ylabel('\omega \rm(rpm)')
title('Run #7, \Delta\itf \rm= 20%')
Km7=(max(w7)-min(w7))/20;

subplot(2,2,4)
```

```
w10=speed(10,:);
plot(t,w10)
set(gca,'XTick',[1.5:0.5:3],'YTick',[1500:400:3500])
axis([1.5 3 1500 3300])
xlabel('\itt \rm(sec)')
ylabel('\omega \rm(rpm)')
title('Run #10, \Delta\itf \rm= 50%')
Km10=(max(w10)-min(w10))/50;

%% begin find Km
k=1;
for i=1:10
    if i <=5
      delf(i)=-50+(i-1)*10;
    else
      delf(i)=-50+i*10;
    end % if
    delw(i)=max(speed(i,50:end))-min(speed(i,50:end));
    Km_full(k)=abs(delw(i)/delf(i));
    k=k+1;
end % for
ave_Km_full=mean(Km_full); % ave motor gain for Runs 1-10 (rpm per % change
in duty cycle)
Km=Km_full([3:10]); % remove Km(1), Km(2) from Km_full
ave_Km=mean(Km);  % ave motor gain for Runs 3-10 (rpm per % change in duty
cycle)
% Km_full,ave_Km_full,Km,ave_Km

%% begin find tau
eps=0.05;
k=1;
for i=1:10
    w=speed(i,:);
    ratio=abs((w-w(1))/(w(end)-w(1)));
    R=find(abs(ratio-0.6321)<eps);  % find indices in w where ratio is
satisfied
    ti=t(R); % find times in t array where ratio is satisfied
    num=length(ti);
    sum=0;
    for j=1:num
        sum=sum+ti(j);
    end
    tau_full(k)=(sum/num)-t(50);
    k=k+1;
end
ave_tau_full=mean(tau_full); % ave motor time constant for Runs 1-10 (sec)
tau=tau_full([3:10]);  % remove tau(1), tau(2) from tau_full
ave_tau=mean(tau); % ave motor time constant for Runs 3-10 (sec)
% tau_full,ave_tau_full,tau,ave_tau


%%%%% 3. root-locus for P-I control %%%%%
ave_Km_rad_per_sec=(ave_Km/60)*2*pi; % ave motor gain (rad/sec per % change
in duty cycle)
```

```
KI=0.4%1; % integral costant (%)
n=[ave_Km_rad_per_sec 0];
d=[ave_tau 1 ave_Km_rad_per_sec*KI];
figure
rlocus(n,d)
Kcrit=(2*sqrt(ave_Km_rad_per_sec*KI*ave_tau)-1)/ave_Km_rad_per_sec;

zeta_des=0.707; % design damping ratio
KC_des=(2*sqrt(ave_Km_rad_per_sec*KI*ave_tau)*zeta_des-1)/ave_Km_rad_per_sec;
% controller gain for design conditions
zeta_des=0.5*(ave_Km_rad_per_sec*KC_des+1)/sqrt(ave_Km_rad_per_sec*KI*ave_tau
); % check on design damping ratio

wn=sqrt(ave_Km_rad_per_sec*KI/ave_tau); % natural frequency of underdamped
control system
tau_eff=1/(wn*zeta_des);

%%%%% 4.  open and closed loop system step responses %%%%%
figure

%% begin open-loop
w_com_bar=1615; % command speed at ss operating pt
w_bar=w_com_bar; % motor speed at ss operating pt (rpm)
w_com=2300; % command speed (rpm)
del_w_com=w_com-w_com_bar; % change in command speed (rpm)
w_com=w_com_bar+del_w_com; % total command speed (rpm)
T=0:0.01:6*tau_eff;
n_OL=ave_Km;
d_OL=[ave_tau 1];
sys_OL=tf(n_OL,d_OL);
[y_OL,tt]=step(sys_OL,T);
del_f=del_w_com/(ave_Km);
del_w_OL=del_f*y_OL;
w_OL=w_bar+del_w_OL;
plot(tt,w_OL,'r')
hold on
plot([0.131 0.46],[1990 1990],'k')

%% begin closed-loop
n_CL=ave_Km_rad_per_sec*[KC_des KI];
d_CL=[ave_tau ave_Km_rad_per_sec*KC_des+1 ave_Km_rad_per_sec*KI];
sys_CL=tf(n_CL,d_CL);
[y_CL,tt]=step(sys_CL,T); % unit step response of closed loop system (rpm)
del_w_CL=del_w_com*y_CL; % deviation in step response of closed loop system
(rpm)
w_CL=w_bar+del_w_CL; % total response of closed-loop system (rpm)
plot(tt,w_CL,'b')
axis([0 2 1500 2400])
plot([0.465 0.57],[2150 2150],'k')
xlabel('\itt  \rm(sec)')
title('Predicted Step Response of Motor: Open-loop and Closed-loop System ')
text(0.5,2000,'open-loop response, \itf \rm= 71.35%')
text(0.6,2150,'closed-loop response  (\itK_{\rmP} \rm= 0.0123, \itK_{\rmI}
\rm= 1)')
```

```
text(1.3,2250,'\omega_{com} \rm= 2300 rpm')
text(0.13,1600,'\omega \rm= 1615 rpm , \itf \rm= 50%')
text(0.137,1628,'_')
text(0.55, 1637,'_')
ylabel('\omega  \rm(rpm)')

%%%%% 5. plots operating characteristic for open-loop control %%%%%
f=[0:10:40 60:10:100];
w_meas=speed([1:10],end)';
figure
plot(f,w_meas,'.r','MarkerSize',12)
hold on
xlabel('duty cycle,  \itf \rm(%)')
ylabel('motor speed,  \omega (rpm)')
title('Operating Characteristic of Motor')
fi=linspace(0,100,2);
f_bar=50; % duty cycle at s.s. operating conditions (%)
wi=w_bar+ave_Km*(fi-f_bar);
plot(fi,wi,'k')
axis([0 100 0 3000])
plot([f_bar],[w_bar],'*k')
text(25,1650,'s.s. operating pt')
text(10,1425,'\itf \rm= 50%, \omega \rm= 1615 rpm')
text(10,1545,'_')
text(22.6,1520,'_')
w0=2300;
f0=f_bar+(w0-w_bar)/ave_Km;
plot([0 f0],[w0 w0],'b:')
plot([f0 f0],[0 w0],'b:')
text(30,600,'\omega \rm= \omega + (\itK_{\rmm} \rm)_{\rmave} \rm(\itf - \itf
\rm)')
text(36,725,'_')
text(58,750,'_')
text(10,2450,'\omega_{\rmcom} = 2300 rpm ')
text(72,150,'\itf_{\rmcom} \rm= 71.35%')
plot([7],[2000],'.r','MarkerSize',12)
text(8,2000,'measured data')
```

open_loop_GUI.m

This function provides the functionality of the open-loop GUI. When sliders in the GUI are updated, this function carries out RTDX writes to the DSP to update those values. Measured speed is displayed in a scrolling graph.

```
function varargout = open_loop_GUI(varargin)
% thesis_GUI M-file for thesis_GUI.fig
%  thesis_GUI, by itself, creates a new thesis_GUI or raises the existing
%  singleton*.
%
%  H = thesis_GUI returns the handle to a new thesis_GUI or the handle to
%  the existing singleton*.
%
%  thesis_GUI('CALLBACK',hObject,eventData,handles,...) calls the local
%  function named CALLBACK in thesis_GUI.M with the given input arguments.
%
%  thesis_GUI('Property','Value',..) creates a new thesis_GUI or raises the
%  existing singleton*.  Starting from the left, property value pairs are
%  applied to the GUI before thesis_GUI_OpeningFunction gets called.  An
%  unrecognized property name or invalid value makes property application
%  stop.  All inputs are passed to thesis_GUI_OpeningFcn via varargin.
%
%  *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%  instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES
% Copyright 2002-2003 The MathWorks, Inc.
% Edit the above text to modify the response to help thesis_GUI
% Last Modified by GUIDE v2.5 17-Feb-2005 17:48:33
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @thesis_GUI_OpeningFcn, ...
                   'gui_OutputFcn',  @thesis_GUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before thesis_GUI is made visible.
function thesis_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
```

```
handles.output = hObject;    % Choose default command line output
guidata(hObject, handles);  % Update handles structure

x_max = 100;
y_max = 3500;

set(handles.figure1,'visible','off')

% Set up axes for RPM plot ********************************
axes(handles.axes_RPM);
set(handles.axes_RPM,'drawmode','fast');
set(handles.axes_RPM,'xlim',[0 x_max]);
set(handles.axes_RPM,'ylim',[0 y_max]);
set(handles.axes_RPM,'xlimmode','manual');
set(handles.axes_RPM,'ylimmode','manual');
set(handles.axes_RPM,'zlimmode','manual');
set(handles.axes_RPM,'climmode','manual');
set(handles.axes_RPM,'alimmode','manual');
set(handles.axes_RPM,'layer','bottom');
set(handles.axes_RPM,'nextplot','add');
xlabel('Time');
ylabel('RPM');

% Plot some dummy data first to get axes handle
allNaN = NaN*ones(1,x_max);
plot(allNaN);
handles.h1 = line('parent',handles.axes_RPM);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Connect to Code Composer Studio(tm), and load out file into target
global cc;
cc = prepare_target_and_load();
configure(cc.rtdx,1024,1);  % configure RTDX, 1 buffer of 1024 bytes

open(cc.rtdx, 'PWM_ichan', 'w');
open(cc.rtdx, 'RPM_ochan', 'r');

enable(cc.rtdx, 'PWM_ichan');
enable(cc.rtdx, 'RPM_ochan');
enable(cc.rtdx);
cc.rtdx
run(cc)

writemsg(cc.rtdx,'PWM_ichan', 0);% initialize PWM to zero

pause_time = 0.1;

frameSize = 1;
xlimit = x_max;
NumOfFrames = xlimit/frameSize;
yLines = handles.h1;
set(handles.figure1,'visible','on');
r = cc.rtdx;
```

```
while(isenabled(cc.rtdx))
   set(yLines, 'ydata', allNaN,'xdata',[1:xlimit], 'Color', 'r');

   for k = 1:NumOfFrames

        % Don't plot if yLines is destroyed
        if ~ishandle(yLines)
            return;
        end
        yAll=get(yLines,'ydata');
        x=(k-1)*frameSize+1;
        y=(k-1)*frameSize+frameSize;

        % Read Encoder values from RTDX channel
        numMsgs = r.msgcount('RPM_ochan');
        if (numMsgs > 0),
            if (numMsgs > 1),
        % flush frames as necessary to maintain real-time display of taps
                r.flush('RPM_ochan',numMsgs-1);
            end
            yAll(x:y) = (readmsg(cc.rtdx,'RPM_ochan', 'double'));
        end

        set(yLines,'ydata',yAll,'xdata',[1:xlimit]);
        set(handles.value_RPM, 'string', num2str(ceil(yAll(x))));
        pause(pause_time);
    end
end

guidata(hObject, handles);  % Update handles structure

% end thesis_GUI_OpeningFcn()*******************************************


% --- Outputs from this function are returned to the command line.
function varargout = thesis_GUI_OutputFcn(hObject, eventdata, handles)
% Get default command line output from handles structure
varargout{1} = handles.output;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function slider_PWM_Callback(hObject, eventdata, handles)
PWM_commanded = get(hObject,'Value');
set(handles.value_PWM, 'string', num2str(ceil(PWM_commanded)));

global cc;
writemsg(cc.rtdx, 'PWM_ichan', double(PWM_commanded))


function slider_PWM_CreateFcn(hObject, eventdata, handles)
% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function HaltDemo_Callback(hObject, eventdata, handles)

global cc;
r = cc.rtdx;

% clean up RTDX
try
    r.disable('PWM_ichan');
    r.disable('RPM_ochan');
    r.disable;
catch
    % if channels are not open, nothing to close
end

cc.reset;

if ishandle(handles.figure1)
    close(handles.figure1)
end
```

closed_loop_GUI.m

This function provides the functionality of the closed-loop GUI. When sliders in the GUI are updated, this function carries out RTDX writes to the DSP to update those values. These sliders can control commanded speed, proportional gain, and integral gain. Measured speed and the system's commanded duty cycle are displayed in a scrolling graph.

```
function varargout = closed_loop_GUI(varargin)
% thesis_GUI M-file for thesis_GUI.fig
%  thesis_GUI, by itself, creates a new thesis_GUI or raises the existing
%  singleton*.
%
%  H = thesis_GUI returns the handle to a new thesis_GUI or the handle to
%  the existing singleton*.
%
%  thesis_GUI('CALLBACK',hObject,eventData,handles,...) calls the local
%  function named CALLBACK in thesis_GUI.M with the given input arguments.
%
%  thesis_GUI('Property','Value',..) creates a new thesis_GUI or raises the
%  existing singleton*.  Starting from the left, property value pairs are
%  applied to the GUI before thesis_GUI_OpeningFunction gets called.  An
%  unrecognized property name or invalid value makes property application
%  stop.  All inputs are passed to thesis_GUI_OpeningFcn via varargin.
%
%  *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%  instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES
% Copyright 2002-2003 The MathWorks, Inc.
% Edit the above text to modify the response to help thesis_GUI
% Last Modified by GUIDE v2.5 04-Mar-2005 11:42:43
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @thesis_GUI_OpeningFcn, ...
                   'gui_OutputFcn',  @thesis_GUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before thesis_GUI is made visible.
function thesis_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
```

```
handles.output = hObject;    % Choose default command line output
guidata(hObject, handles);  % Update handles structure

x_max = 100;
y_max = 3500;

% Set up axes for QEP plot ********************************
set(handles.figure1,'visible','off')
axes(handles.axes_RPM_measured);
set(handles.axes_RPM_measured,'drawmode','fast');
set(handles.axes_RPM_measured,'xlim',[0 x_max]);
set(handles.axes_RPM_measured,'ylim',[0 y_max]);
set(handles.axes_RPM_measured,'xlimmode','manual');
set(handles.axes_RPM_measured,'ylimmode','manual');
set(handles.axes_RPM_measured,'zlimmode','manual');
set(handles.axes_RPM_measured,'climmode','manual');
set(handles.axes_RPM_measured,'alimmode','manual');
set(handles.axes_RPM_measured,'layer','bottom');
set(handles.axes_RPM_measured,'nextplot','add');
xlabel('Time');
ylabel('RPM');

% Plot some dummy data first to get axes handle
allNaN = NaN*ones(1,x_max);
plot(allNaN);
handles.h1 = line('parent',handles.axes_RPM_measured);


% Set up axes for RPM plot ********************************

y_max = 100;

axes(handles.axes_duty_cycle);
set(handles.axes_duty_cycle,'drawmode','fast');
set(handles.axes_duty_cycle,'xlim',[0
x_max]);%set(handles.axes_duty_cycle_measured,'xlim',[0 400]);
set(handles.axes_duty_cycle,'ylim',[0 y_max]);
set(handles.axes_duty_cycle,'xlimmode','manual');
set(handles.axes_duty_cycle,'ylimmode','manual');
set(handles.axes_duty_cycle,'zlimmode','manual');
set(handles.axes_duty_cycle,'climmode','manual');
set(handles.axes_duty_cycle,'alimmode','manual');
set(handles.axes_duty_cycle,'layer','bottom');
set(handles.axes_duty_cycle,'nextplot','add');
xlabel('Time');
ylabel('%');

% Plot some dummy data first to get axes handle
allNaN = NaN*ones(1,x_max);
plot(allNaN);
handles.h2 = line('parent',handles.axes_duty_cycle);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Connect to Code Composer Studio(tm), and load out file into target
global cc;
cc = prepare_target_and_load();
configure(cc.rtdx,1024,3);  % configure RTDX, 3 buffers of 1024 bytes each

open(cc.rtdx, 'KP_ichan', 'w');
open(cc.rtdx, 'KI_ichan', 'w');
open(cc.rtdx, 'RPM_ichan', 'w');
open(cc.rtdx, 'QEP_ochan', 'r');
open(cc.rtdx, 'PWM_ochan', 'r');

enable(cc.rtdx, 'KP_ichan');
enable(cc.rtdx, 'KI_ichan');
enable(cc.rtdx, 'RPM_ichan');
enable(cc.rtdx, 'QEP_ochan');
enable(cc.rtdx, 'PWM_ochan');
enable(cc.rtdx);
cc.rtdx
run(cc)

writemsg(cc.rtdx,'KP_ichan', 0);    % initialize Kp to 0
writemsg(cc.rtdx,'KI_ichan', 0);    % initialize Ki to 0
writemsg(cc.rtdx,'RPM_ichan', 0);   % initialize RPM to 0


global pause_time
pause_time = 0.1;

frameSize = 1;
xlimit = x_max;
NumOfFrames = xlimit/frameSize;
yLines = handles.h1;
yLines_PWM = handles.h2;
set(handles.figure1,'visible','on');
r = cc.rtdx;

while(isenabled(cc.rtdx))
   set(yLines, 'ydata', allNaN,'xdata',[1:xlimit], 'Color', 'r');%, 'Marker',
'*');
   set(yLines_PWM, 'ydata', allNaN, 'xdata', [1:xlimit], 'Color', 'r');

   for k = 1:NumOfFrames

        % Don't plot if yLines is destroyed
        if ~ishandle(yLines)
            return;
        end
        yAll=get(yLines,'ydata');
        yAll_PWM = get(yLines_PWM, 'ydata');
        x=(k-1)*frameSize+1;
        y=(k-1)*frameSize+frameSize;
```

```
        % Read Encoder values from RTDX channel
        numMsgs = r.msgcount('QEP_ochan');
        if (numMsgs > 0),
            if (numMsgs > 1),
        % flush frames as necessary to maintain real-time display of taps
                r.flush('QEP_ochan',numMsgs-1);
                r.flush('PWM_ochan',numMsgs-1);
            end
            clc
            yAll(x:y) = (readmsg(cc.rtdx,'QEP_ochan', 'double'));
            yAll_PWM(x:y) = (readmsg(cc.rtdx,'PWM_ochan', 'double'));
        end

        set(yLines,'ydata',yAll,'xdata',[1:xlimit]);
        set(yLines_PWM, 'ydata', yAll_PWM, 'xdata', [1:xlimit]);
        set(handles.value_RPM_measured, 'string', num2str(ceil(yAll(x))));
        set(handles.value_duty_cycle, 'string', num2str(ceil(yAll_PWM(x))));
        pause(pause_time);
    end
end

guidata(hObject, handles);  % Update handles structure

% end thesis_GUI_OpeningFcn()*****************************************

% --- Outputs from this function are returned to the command line.
function varargout = thesis_GUI_OutputFcn(hObject, eventdata, handles)
% Get default command line output from handles structure
varargout{1} = handles.output;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function slider_RPM_commanded_Callback(hObject, eventdata, handles)
PWM_commanded = get(hObject,'Value');
set(handles.value_RPM_commanded, 'string', num2str(PWM_commanded));
global cc;
writemsg(cc.rtdx, 'RPM_ichan', double(PWM_commanded))%/100))

function slider_RPM_commanded_CreateFcn(hObject, eventdata, handles)
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function HaltDemo_Callback(hObject, eventdata, handles)

global cc;
r = cc.rtdx;

% clean up RTDX
try
    r.disable('KP_ichan');
    r.disable('KI_ichan');
```

```
    r.disable('RPM_ichan');
    r.disable('QEP_ochan');
    r.disable('PWM_ochan');
    r.disable;
catch
    % if channels are not open, nothing to close
end

cc.reset;

if ishandle(handles.figure1)
    close(handles.figure1)
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function slider_KP_Callback(hObject, eventdata, handles)
KP = get(hObject,'Value');
% set(handles.value_KP, 'string', num2str(ceil(PWM_period_commanded)));
set(handles.value_KP, 'string', num2str(KP));
global cc;
writemsg(cc.rtdx, 'KP_ichan', double(KP))

function slider_KP_CreateFcn(hObject, eventdata, handles)
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function slider_KI_Callback(hObject, eventdata, handles)
KI = get(hObject,'Value');
% set(handles.value_KI, 'string', num2str(ceil(PWM_period_commanded)));
set(handles.value_KI, 'string', num2str(KI));
global cc;
writemsg(cc.rtdx, 'KI_ichan', double(KI))

function slider_KI_CreateFcn(hObject, eventdata, handles)
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end
```

motor_control.m

This m-file calls two simulink models to simulate the analog and digital closed-loop systems and compare them.

```
% m-file "motor_control.m"
% m-file that sets parameters for analog and digital control of DC motor
% uses discrete transfer D(z) for digital controller
% computes f(k) as f_bar + delta f(k)
% motor speed is initialized at w_bar and changed using Step block

clc
clear
close all
load 2300_RPM_closed_loop_Kp-20_Ki-1000.mat

%%%%% set motor command and motor parameters
w_bar = 1615; % motor speed at operating pt (rpm)
f_bar = 50; % duty cycle at s.s. operating pt
w_com = 2300; % command motor speed at operating pt (rpm)
w_com_step_time_anal = 2; % time of occurrence of change in command motor
speed (sec)
Km = 32.08; % motor DC gain (rpm/volt)
tau = 0.161; % motor time constant (sec)
KP = 0.0691; KI = 1; KD = 0; % PID parameters
tfinal = 4; % sim time (sec)
T=0.001; % integration step size for RK-4 integration of motor dynamics (sec)


%%%%% analog control %%%%%
sim('analog_control_1') % calls Simulink model 'analog_control_1'
t = w_analog(:,1); % sim time (sec)
w_comm = w_com_analog(:,2); % command motor speed (rpm)
f_anal = f_analog(:,2); % duty cycle output from controller (%)
w_anal = w_analog(:,2); % motor speed (rpm)

figure
subplot(2,1,1)
plot(t,w_comm,'r')
hold on
plot(t,w_anal,'b')
XMIN = 1.8;
XMAX = 3.2;
YMIN = 1550;
YMAX = 2900;
axis([XMIN XMAX YMIN YMAX])
ylabel('\omega\rm(\itt\rm)  (rpm)')
title('Simulated Analog Control System Motor Speed (KP = 0.0691, KI = 1)')

subplot (2,1,2)
plot(t,f_anal)
YMIN = 0; YMAX = 100;
```

```
axis([XMIN XMAX YMIN YMAX])
xlabel('\itt \rm(sec)')
ylabel('\itf\rm(\itt\rm)  \rm(%)')
title('Simulated Analog Control System Duty Cycle (KP = 0.0691, KI = 1)')


%%%%% digital control %%%%%
figure
Ts = 0.02; % sampling time for digital control (sec)
% D(z) is based on bilinear transform of G(s)=KC + KI/s +KDs
% D(z) = E0(z)/E(z) = [bo(z^2) + b1(z) + b2]/[(z^2)-1]
b0 = KP+0.5*KI*Ts+2*KD/Ts;  % digital controller coefficient
b1 = KI*Ts-4*KD/Ts; % digital controller coefficient
b2 = -KP+0.5*KI*Ts+2*KD/Ts; % digital controller coefficient
w_com_step_time_dig = w_com_step_time_anal-Ts; % time of occurrence of change
in command motor speed (sec)
sim('digital_control_1')
w_dig = w_digital(:,2); % motor speed
f_dig = f_digital(:,2); % duty cycle (%)

subplot(2,1,1)
plot(t,w_comm,'r')
hold on
plot(t,w_dig)
XMIN = 1.8;%1.9;
XMAX = 3.2;%2.5;
YMIN = 1550;
YMAX = 2900;    %2175
axis([XMIN XMAX YMIN YMAX])
ylabel('\omega\rm(\itt\rm)  (rpm)')
title('Simulated Digital Control System Motor Speed (KP = 0.0691, KI = 1)')

subplot (2,1,2)
t_i = t(1:20:end);
f_dig_i = f_dig(1:20:end);
plot(t,f_dig,'r')
YMIN = 0; YMAX = 100;
axis([XMIN XMAX YMIN YMAX])
xlabel('\itt \rm(sec)')
ylabel('\itf\rm(\itk\rm) (%)')
title('Simulated Digital Control System Duty Cycle (KP = 0.0691, KI = 1)')


%%%%% compare simulated and measured closed-loop responses %%%%%
% load closed_loop.mat % loads tt speed duty_cycle,  all are 1x200
figure

subplot(2,1,1)
plot(t,w_anal,'b') % plot simulated analog motor speed
hold on
plot(t,w_dig,'r') % plot simulated digital motor speed
plot(tt,speed,'k') % plot measured motor speed
YMIN = 1550; YMAX = 2900;
axis([XMIN XMAX YMIN YMAX])
```

```
title('Simulated Analog, Simulated Digital, and Measured Speed')
legend('simulated analog','simulated digital','measured',4)

subplot(2,1,2)
plot(t,f_anal,'b') % plot simulated analog duty cycle
hold on
plot(t,f_dig,'r') % plot simulated digital duty cycle

for i=102:length(tt)-1
    plot([tt(i-1) tt(i)],[duty_cycle(i) duty_cycle(i)],'k')
    plot([tt(i) tt(i)],[duty_cycle(i) duty_cycle(i+1)],'k')
end

YMIN = 45; YMAX = 100;
axis([XMIN XMAX YMIN YMAX])
title('Simulated Analog, Simulated Digital, and Measured Duty Cycle')
legend('simulated analog','simulated digital','measured',1)
```

APPENDIX B: GRAPHICAL USER INTERFACE

Following are some screenshots of graphical user interfaces that allow a user to adjust the control system's parameters and view results in real-time. The Simulink models used for the GUIs are almost identical to their fixed-value counterparts. Constant values are replaced with RTDX inputs. This allows controls in the GUI, such as sliders and text boxes, to transmit parameters to the DSP in real-time. Figure 48 shows the model used for the open-loop GUI.

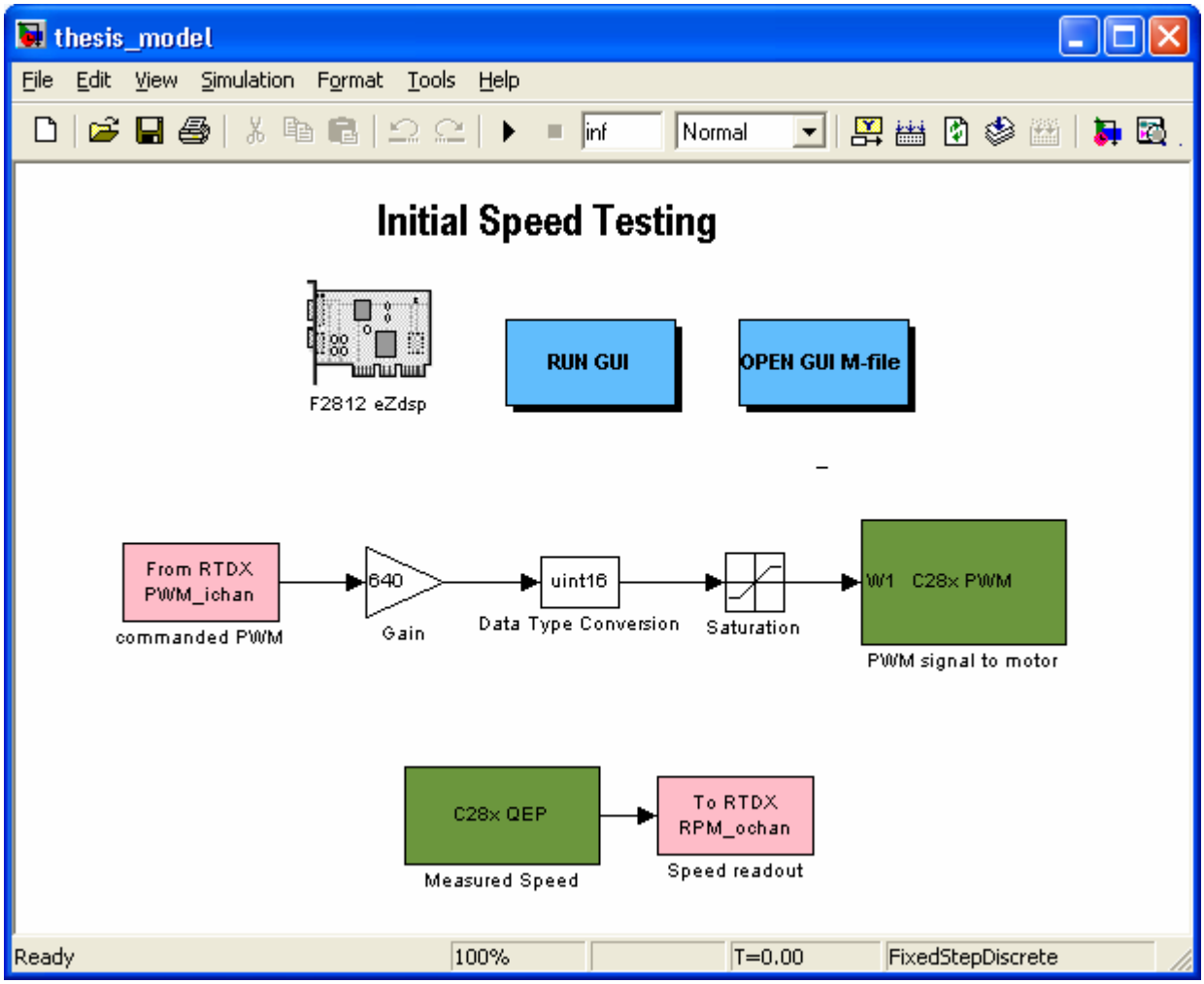

Figure 48: Initial Speed GUI Simulink Model

The GUI shown in Figure 49 allows the user to adjust the PWM duty cycle from 0 to 100%. A scrolling plot of speed readings is displayed as well.

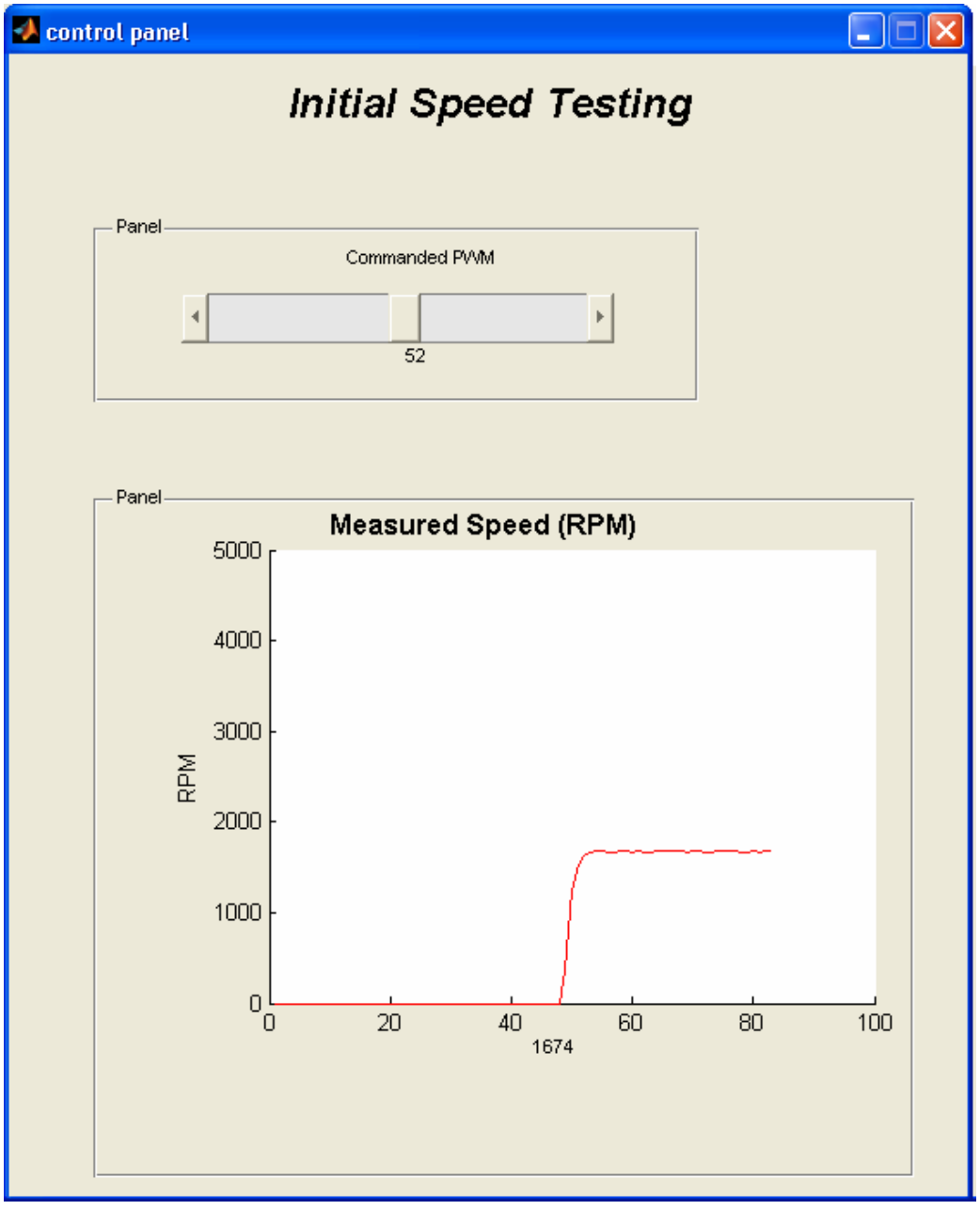

Figure 49: Initial Speed GUI

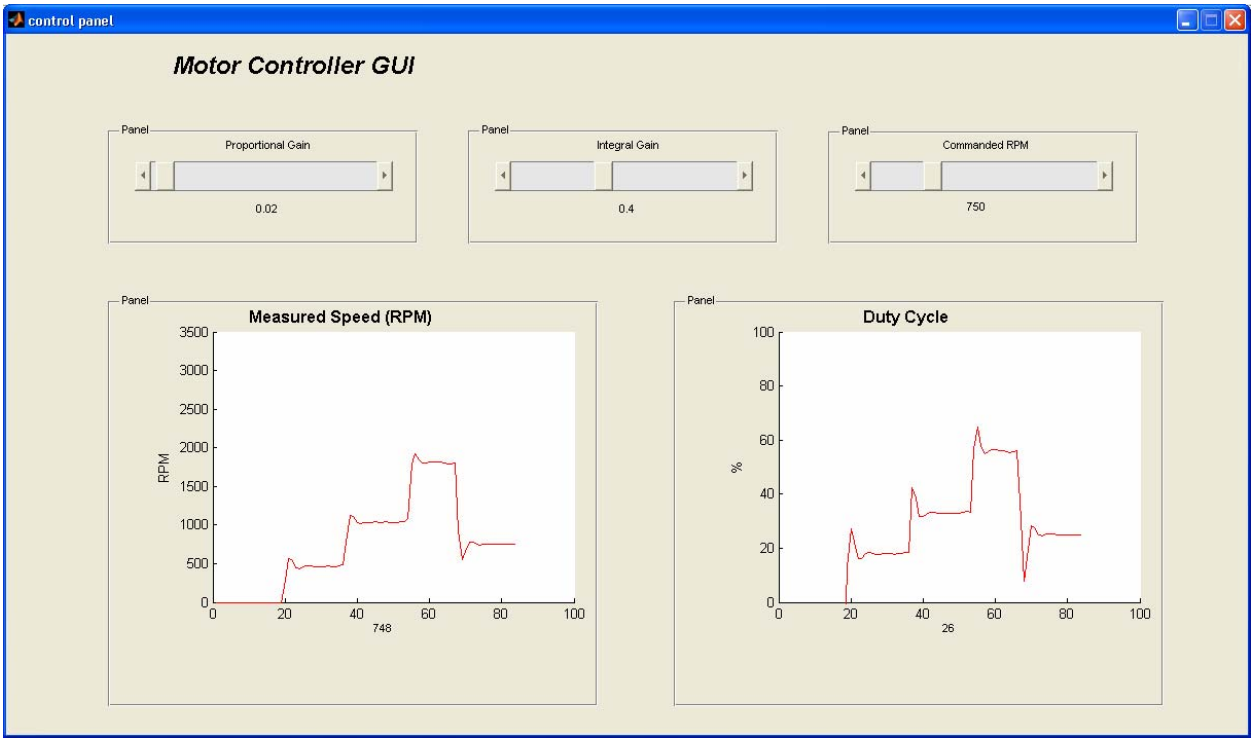

Figure 50: Closed-loop GUI

The closed-loop GUI in Figure 50 looks similar to the open-loop GUI, but has some subtle differences. Instead of commanding a particular PWM duty cycle, the user controls the commanded speed. The user can also adjust the proportional and integral gains. The plot on the left displays measured speed readings. The plot on the right indicates the PWM duty cycle that the control system is commanding.

In Figure 51, notice the values that were previously fixed are now connected to RTDX sources. This allows the user to modify the proportional and integral gains and instantly see the effects on the motor's response.

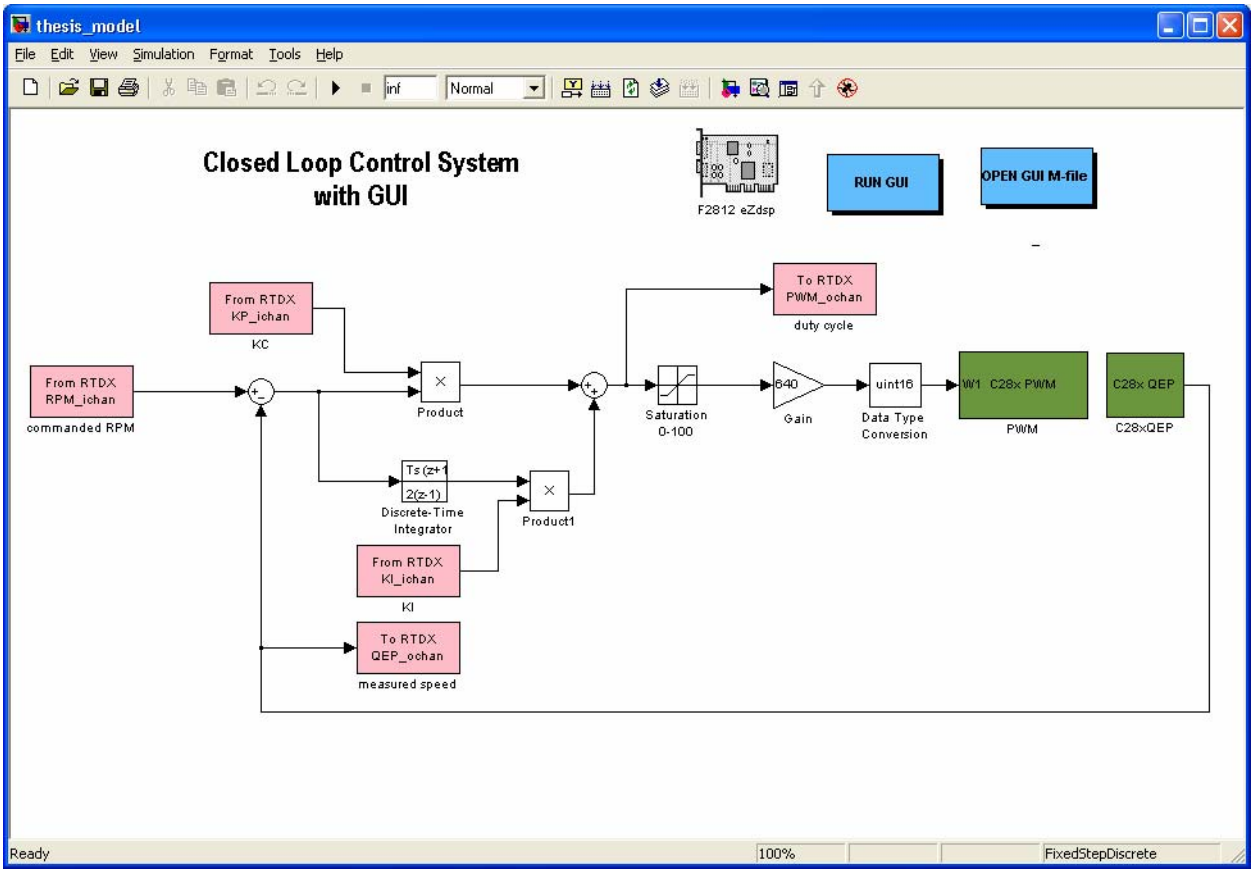

Figure 51: Closed-loop GUI Simulink Model

LIST OF REFERENCES

1. The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098
2. http://focus.ti.com/lit/ds/symlink/tms320f2812.pdf
3. http://www.spectrumdigital.com/cgi/catalog.cgi?show_product=761128
4. Modern Control Systems. Dorf, Richard C. and Bishop, Robert H. Addison Wesley Longman, Inc. 1998.
5. www.maxonmotorusa.com/products
6. Automatic Control Systems: Sixth Edition. Kuo, Benjamin C. Prentice Hall, Inc. 1991.
7. Mobile Robotic Car Design. Kachroo, Pushkin and Mellodge, Patricia. The McGraw-Hill Companies, Inc. 2005.
8. Building Robot Drive Trains. Clark, Dennis and Owings, Michael. The McGraw-Hill Companies, Inc. 2003.
9. http://www.mathworks.com/access/helpdesk/help/toolbox/tic2000/