MATLAB Based DC Motor Control Lab

Benjamin J. Engle
Advisor: Dr. John Watkins

Wichita State University
College of Engineering
December 4, 2007

Table of Contents

1.0 Introduction

Wichita State University currently offers a variety of control courses. These range from introductory courses to those covering advanced topics such as optimal and robust control. A course in digital control is now being offered and has received great student interest. In a recent semester the 50 available seats were reserved on the first day of registration. The number of available seats was increased to 80, but these were quickly filled and many students still had to be turned away. This digital control class covers the underlying theory of digital control and the design of various classical controllers, such as PD, PID, PI, and lead compensators. However, there is currently not a lab component available to complement this course.

Many students struggle to understand the abstract concepts used in control. The understanding of concepts such as frequency response, poles and zeros, stability, and controller implementation can be greatly enhanced by a lab component [2]. Schools that have implemented a control laboratory have reported high student ratings and good industry feedback [2] [3]. In addition, a lab is of tremendous value when teaching topics such as modeling, implementation, and operation that are needed to properly prepare a student to work in industry [1].

In recent years this lack of a control lab has been addressed in various ways at many universities. For example, the University of Illinois addressed the issue with a 2000 square foot interdepartmental control lab with a full time lab manager [2]. The University of North Florida addressed the issue by creating an electrical engineering specific control lab with 13 lab stations [3]. Work has also been completed at the University of Minnesota to develop take home lab kits in order to further reduce the cost and space required for the lab [4].

Wichita State University currently owns a number of Quanser Engineering Trainers (QET). Among other components, this trainer contains a DC motor with an optical angle encoder, a programmable microcontroller with a serial port interface, and the necessary circuitry to power and interconnect these main components. A review of the QET is available in [8] and further discussion of the hardware is presented in Section 2 of this report. Custom code can be created for the installed microcontroller that can interface with the hardware as well as communicate with a PC over the serial port.

The goal of this project was to write C code for the microcontroller installed in the QET and to interface with it over a serial port on a standard PC using a graphical user interface in MATLAB. The purpose of this is to provide a low cost solution that will allow a student to implement and test a digital controller design. In this manner the student can obtain hands-on experience with implementing a digital controller without burden of writing all of the microcontroller and MATLAB code from scratch.

2.0 Quanser Engineering Trainer Hardware

The Quanser engineering trainer (QET) DC motor control trainer (DCMCT) is a unit produced by Quanser Consulting of Markham, Ontario, Canada. A picture of the QET DCMCT is shown in Figure 1. The Wichita State University units come with an installed QIC (Quanser Integrated Controller) board that contains a PIC 16F877 microcontroller. This unit also comes equipped with a high quality Maxon DC motor, a linear power amplifier that allows the QIC to apply a control voltage to the motor, a 1024 line optical encoder to measure the motor angle, a series resistor to allow the QIC to measure the current being used by the motor, and the hardware necessary to allow the microcontroller to interface with a PC through the provided serial port. Figure 2 shows a block diagram of the QET as it is used in this project for motor angular position control.

The trainer supports other capabilities that are not being used in this project. These include the ability to interface with Quanser HIL (hardware in the loop) boards through the provided analog ports, the availability of a potentiometer to read the angle of the motor, and the availability of a breadboard to prototype custom circuits.

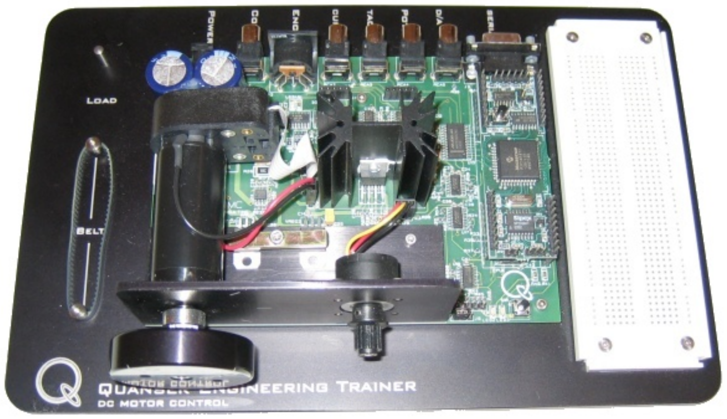


Figure 1 Quanser Engineering DC Motor Control Trainer

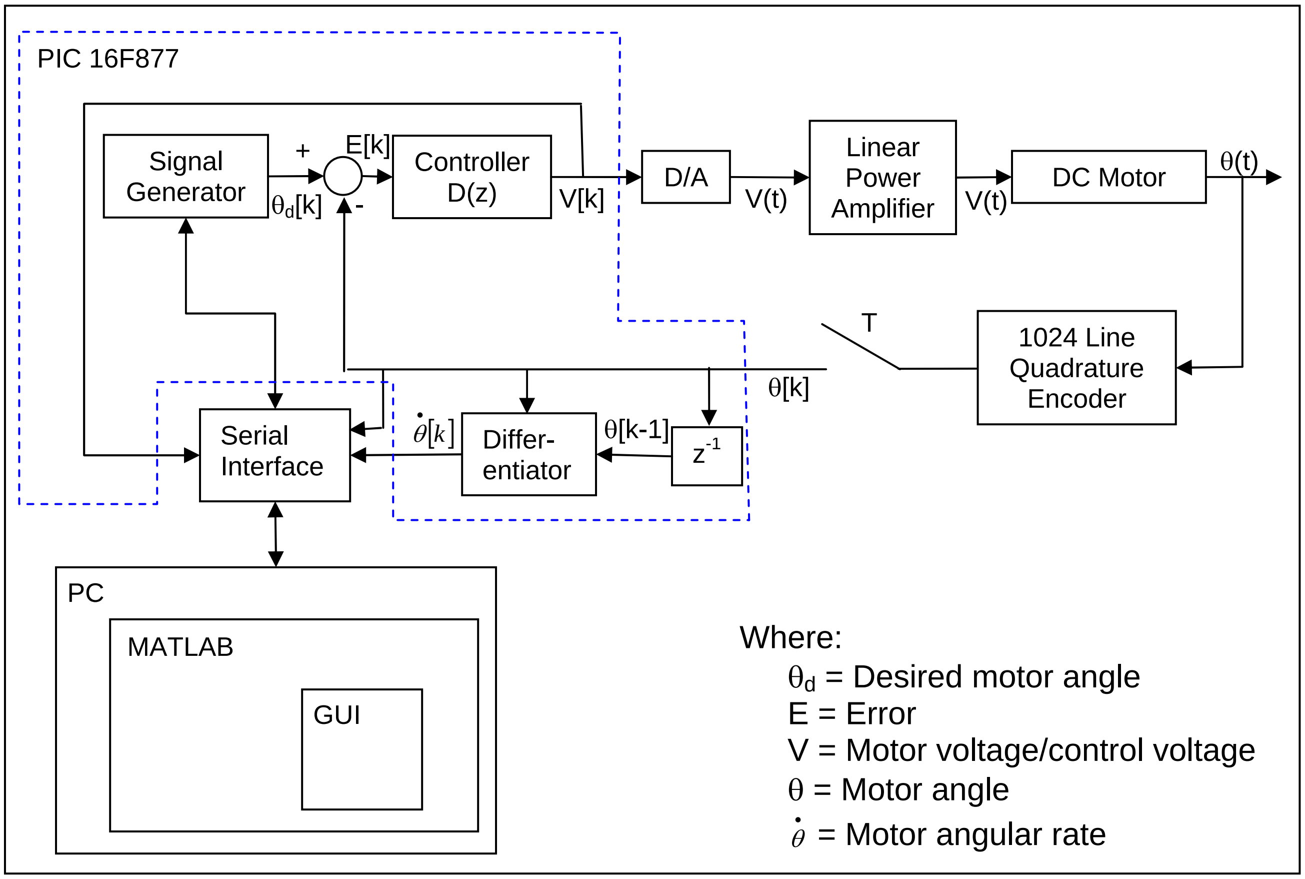


Figure 2 System Block Diagram for Angular Position Control

2.1 PIC 16F877 Microcontroller

The Microchip PIC 16F877 microcontroller installed on the QIC is a high performance RISC (reduced instruction set computer) CPU (central processing unit). The microcontroller supports clock speeds up to 20 MHz, hardware interrupts, and USART (Universal Synchronous/Asynchronous Receiver Transmitter) for serial port communication [11]. Additionally, as it is installed on the QIC it supports low power programming through the available serial port.

In this project the maximum clock speed of 20 MHz was used to provide minimum calculation time and maximum control frequency. For this project the 16F877's 16 bit timer/interrupt is used to run a digital controller at the desired clock frequency. Information is passed to and from the 16F877 via the serial port at a baud rate of 115,200 bits/second.

The code written for the 16F877 was written using the PCW PIC C compiler produced by Custom Computer Services, Inc. (CCS). Using C code provides a higher level software interface to the hardware than could be accomplished using assembly language. Also, because the microcontroller code written for the trainer by Quanser is in C, this allows some of their code to be reused.

2.2 Position Measurement

The angular position of the motor is measured by a 1024 line quadrature angular encoder with a resolution of approximately 0.0879°. The encoder's measurement is made available to the 16F877 as a 24 bit integer value. Note that the encoder is initialized to zero at the start of the code so that the angle measured is the difference from the initial motor angle. Due to the resolution of the encoder and the use of the 24 bit counter, the range of values that can be read by the encoder is approximately $\pm 4096*2\pi$.

2.3 Current Measurement

The current being drawn by the motor is read using a series resistor and is passed to the microcontroller as an analog voltage. This voltage is read using the 16F877's 10 bit analog to digital converter and is used to calculate the motor current. The resolution of the current measurement is approximately 0.5 mA ±10%.

3.0 Microcontroller C Code

C programming code was written specifically for this application. A listing of this code is available in Appendix A and the flowcharts for this code are available in Appendix C. This code implements all of the control logic as well as the necessary routines to communicate with the MATLAB code described in Section 4.0.

There are many functions in the C code, most of which should not need to be modified. These functions provide capabilities whose purpose is unlikely to change, such as serial communication, control timing, reference signal generation, and hardware interaction. This allows the student to concentrate on the programming of their control design without the overhead of learning hardware implementation specific code. The only function that must be changed is the control function that is described in Section 3.4.

3.1 Signal Types

Various signal types are supported by the code. These signal types are described in detail along with figures in the help file [12]. The available signal types are:

- Impulse
- Step
- Ramp
- Parabolic
- Triangle Wave
- Saw Tooth Wave
- Square Wave
- Sine Wave
- Cosine Wave
- Stair Step

3.2 Angular Rate Calculation

The angular rate is determined by finding the first derivative of the position measured by the angular encoder. Two different methods were implemented in the C code for this. The first uses a simple finite difference calculation as shown in (3.1).

$$\dot{\theta} = \frac{\theta_{Now} - \theta_{last}}{T} \tag{3.1}$$

The second method is the same as that used by Quanser and is derived by taking the bilinear transform of the continuous high pass filter shown in (3.2).

$$\dot{\theta} = \left(\theta_{Now} - \theta_{last}\right) \cdot \frac{s}{s/300 + 1} \tag{3.2}$$

Both methods are implemented in the code and are made available by the GUI. Either method may be used by the student. Based upon work with both methods there is little difference in either the noise or results of the two methods.

3.3 Sample Rate Specification

The top of the C code contains many constants and variable declarations that should not be modified by the student. However, there is one that must be modified by the student. This is the constant named “fs” that defines the sampling frequency, in Hz. For example, to define a sampling rate of 150 Hz the fs definition should be modified as follows:

```
#define fs 150
```

Due to hardware limitations and the 20 MHz clock rate of the 16F877, the lowest sample rate that can be used is 10 Hz. In addition, due to the computation time required to complete a control step, the maximum sample rate is approximately 150 Hz. For simple signal types, such as a step signal, higher sample rates may be achievable. If sample rates higher than 150 Hz are desired then an oscilloscope would be necessary to verify that the desired sample rate can be achieved for a particular signal type and controller.

3.4 Student Modified Control Function

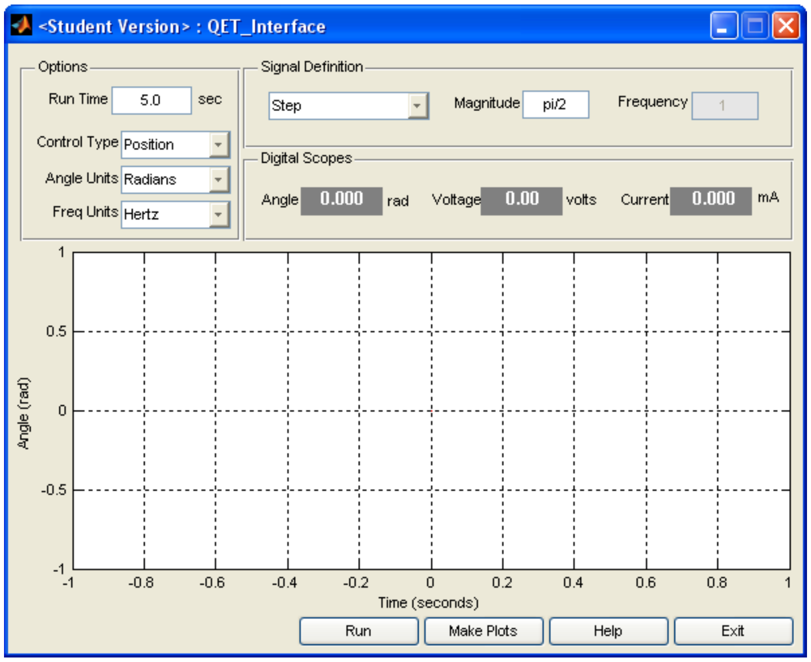
The only function that must be updated by the student to implement their desired controller is the function named “RunController”. This function will be called at the specified sample rate and allows the implementation of digital controllers. Much of the code in this function does things that will nearly always be required and thus probably won't need to be changed. This is the code that does things such as generate the new reference signal, reads the angle from the encoder, reads the motor current from the analog to digital converter, and calculates the angular rate.

However, there are areas of this function that will need to be modified to implement whatever controller the user desires. The first area is labeled “controller parameters” and stores any constant parameters that are needed for the controller. These are typically things such as controller gains, poles, and zeros.

The second area of the function that will need to be modified is labeled “controller logic” and is where the student implements their control logic. These modifications will depend upon the specific controller being implemented. Examples of what may need to be done in this area are available in Section 6.0.

4.0 MATLAB Graphical User Interface

A graphical user interface was written to simplify the interface with the QET. A picture of the GUI is shown in Figure 3. The GUI allows the user to specify the desired signal type and run options for the QET. The user can also create a standard set of plots from the GUI. The GUI contains an input area for options, an input area for the signal definition, an area for the digital scopes, a plot area where data are plotted while the QET is being run, and four buttons that operate the form.


<Student Version> : QET_Interface

Options
Run Time 5.0 sec
Control Type Position
Angle Units Radians
Freq Units Hertz

Signal Definition
Step
Magnitude pi/2
Frequency 1

Digital Scopes
Angle 0.000 rad
Voltage 0.00 volts
Current 0.000 mA

Figure 3 MATLAB QET Graphical User Interface

A help file has been written that details the operation of the graphical user interface. The following sections give a basic overview of the operation of the GUI. However, additional information and details can be found in the help file.

4.1 Graphical User Interface Options

The options area allows the user to specify various options for the input and outputs of the GUI. These options are as follows.

- Run Time - This is the total time, in seconds, that the QET should run.
- Control Type - This tells the GUI what type of control the user has implemented on the QET. This allows the GUI to place the reference signal on the proper plot when the "Make Plots" button is pressed and to show the correct data in the plot area while the QET is running.
- Angle units - This allows the user to specify the angle units to be displayed on the scopes and in the output plots. Also, if the user has selected the

rate or position control type then this also inputs the units for the input magnitude.

- Frequency units – Some signal types require a frequency input (such as a sine wave). For these signal types this option tells the GUI whether the input frequency is in Hz or rad/s.

4.2 Signal Definition

The signal definition input area allows the user to specify characteristics of the desired signal. The inputs are as follows.

- Signal Type - This allows the user to specify the desired signal type. The available signal types are listed in Section 3.1.
- Magnitude - This allows the user to specify the signal magnitude or rate, as applicable. The units of this input (degrees or radians) for position or rate control are specified in the options frame. Further information about what this value represents for the various signal types is available in the help file.
- Frequency - This allows the user to specify the signal frequency (if applicable). If this isn't applicable to the selected signal type then this input will be disabled. The units of this input are specified in the options frame.

4.3 Digital Scopes and Plot Area

The digital scopes frame contains the current values of angle, voltage, and current while the QET is running. The plot area displays the current reference signal. In addition, the actual results from the QET will be displayed on this plot. The data that is displayed on this plot is determined by the selection made for the control type in the options input area.

4.4 Operation Buttons

There are four buttons at the bottom of the GUI. The first button is labeled “Run” and runs the QET using the current input options and signal definition. The second button is labeled “Make Plots” and will create plots of control voltage, motor current, motor angular rate, and motor angle versus time for the last completed run. The third button is labeled “Help” and will display the help file when pressed. The fourth button is labeled “Exit” and will close the GUI.

4.5 Step Response Calculations

When the user runs a step signal then the percent overshoot, settling time, and steady state error will be calculated and displayed in the plot area. Also, if the user clicks on the “Make Plots” button then the step response characteristics will also be displayed on the resulting plots.

4.6 Output MATLAB Workspace Variables

When the user runs the QET through the GUI various output variables are placed in the base MATLAB workspace. These variables contain the input characteristics and output data from the QET. This allows the student to do custom post processing on the control results. Further details of these variables are available in the help file.

5.0 Help File

An HTML help file has been generated to assist in using the use of the QET and MATLAB interface [12]. A picture of this help file is available in Figure 4. This help file contains an overview of the QET hardware, information on running the QET from MATLAB, an overview of the MATLAB GUI, and some examples on controller implementation. This help file is available to the user by pressing the help button on the MATLAB GUI.

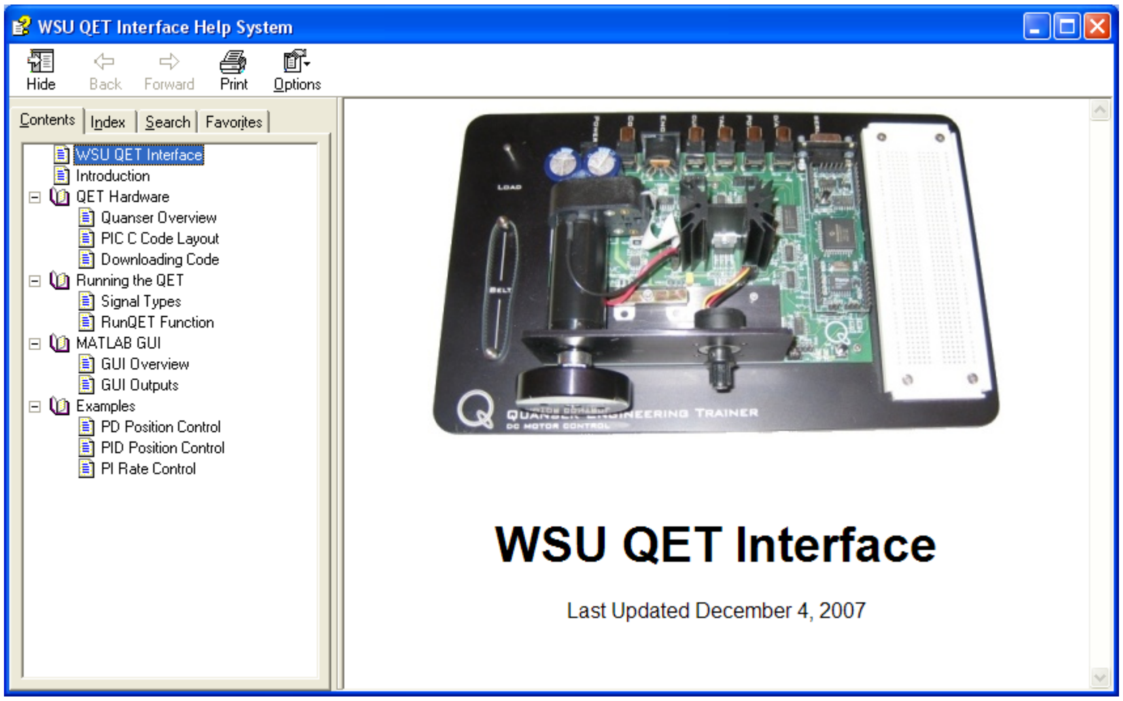


Figure 4 Help File

6.0 Examples

This section contains examples of both a proportional integral-derivative (PID) angular position controller and a proportional integral (PI) angular rate controller. Only the implementation of these controllers is discussed as the design of these controllers is outside the scope of this project.

When implementing a controller into the C code the user must first set the sample rate at the top of the code to the desired value as shown in section 3.3. Next the user must modify the function named “RunController” to implement their control logic. Most of the code in this function should remain unmodified as it completes calculations that will always be necessary. However, other parts of the code must be changed to implement the controller and it is these changes that are discussed in the following examples.

The variables that will always be used in the “RunController” function are listed below. The code that calculates these variables should not be modified.

- AngleLast – The angle of the motor, in radians, at the previous time step
- Angle – The angle of the motor, in radians, read from the angular encoder at the current time step
- RefSigLast – This is the reference signal at the previous time step. The meaning of the reference signal will depend on the control type that is being completed. For example, if the implemented controller is intended to control angular position then this reference signal would be the desired angular position.
- RefSig – The reference signal at the current time step
- Current – The current through the motor, in milliamps, at the current time step
- AngleFD – The first derivative of the motor angle, in radians/second, at the current time step using (3.1)
- AngleFDFilt - The first derivative of the motor angle, in radians/second, at the current time step using the bilinear transform of the continuous time filter shown in (3.2)
- Volt – The motor voltage, in volts, that should be applied to the motor at the current time step

6.1 PID Position Controller

The motor position may be controlled using a proportional integral-derivative controller. This control should greatly reduce or eliminate any steady state error that may be present using a gain or proportional derivative controller. The downside to this method is that adding an integrator will degrade the transient performance of the system.

Below is the transfer function of a proposed proportional integral-derivative position controller that was designed for the QET.

$$G_c(z) = \frac{V(z)}{E(z)} = \frac{\mathrm{K}\,(\mathrm{z} - \mathrm{a1})(\mathrm{z} - \mathrm{a2})}{\mathrm{z}\,(\mathrm{z} - 1)} \qquad \textbf{(5.1)}$$

where

V(z) = Motor voltage
E(z) = Angle Error = Desired Angle - Actual Angle
K = 6.6502
a1 = 0.8751
and a2 = 0.9190

This transfer function can be implemented using the following difference equation:

$$\mathrm{V}[\mathrm{k}] = \mathrm{V}[\mathrm{k}-1] + \mathrm{K} \cdot \{\mathrm{E}[\mathrm{k}] - (\mathrm{a1} + \mathrm{a2}) \cdot \mathrm{E}[\mathrm{k}-1] + \mathrm{a1} \cdot \mathrm{a2} \cdot \mathrm{E}[\mathrm{k}-2]\} \qquad \textbf{(5.2)}$$

The MATLAB simulation of this controller predicts a 20% overshoot and a settling time of 0.64 seconds as shown in the following figure.

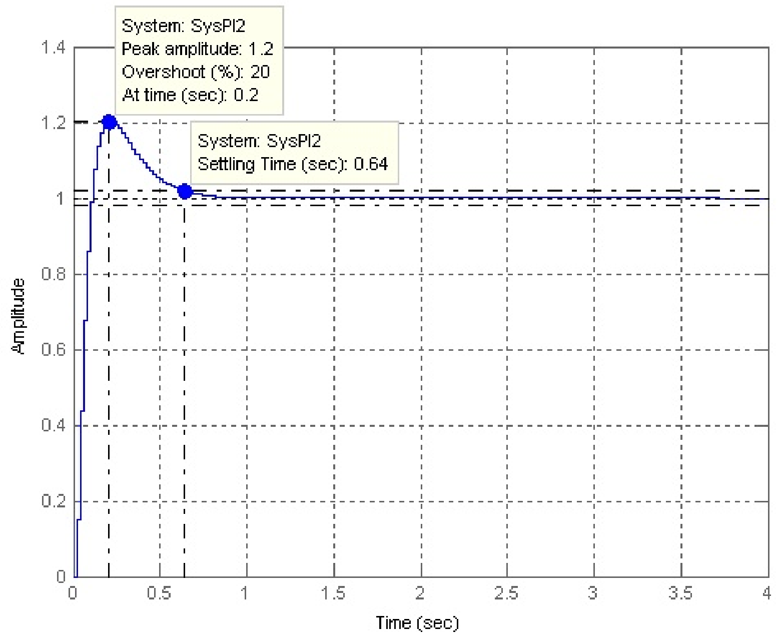


Figure 5 Predicted PID Position Control Example Results

To implement this controller on the QET the user should define the controller gain and zero locations at the top of the “RunController” function in the "controller parameters" section. The following code shows an example of this.

```
//PID controller parameters
const float K = 6.6502;
const float a1 = 0.8751;
const float a2 = 0.9190;
```

The code that calculates the reference signal, motor angle, and motor angular rate is already completed for the user as discussed in the introduction to this section. Thus, once the controller parameters are defined the next step is to write the code to calculate the error at the current time step and save the necessary number of error measurements from previous time steps. From (5.2) it can be seen that for this example the errors from the current time step and previous two time steps are needed. The following code snippet is an example of what may be used to store the errors.

```
Errors[2] = Errors[1];          //Error two time steps ago
Errors[1] = Errors[0];          //Error at last time step
Errors[0] = RefSig - Angle;  //Error at current time step
```

For a controller with an integral term the user must also save the last control voltage. The following code snippet is an example of storing the last control voltage.

```
VoltLast = Volt;
```

For the previous two code snippets to be valid the user must define both the three element array “Errors” and the variable “VoltLast” at the top of the function. The following is an example of the necessary variable definitions.

```
static float Errors[3];
float VoltLast;
```

Once the controller parameters are defined the user must add the PID control logic to the "controller logic" section of the “RunController” function. The following code shows the implementation of difference equation (5.2).

```
Volt = VoltLast + K*(Errors[0] - (a1+a2)*Errors[1] + a1*a2*Errors[2]);
```

Once these code changes have been implemented the code can be compiled and loaded onto the 16F877 microcontroller as discussed in the help file. The GUI can then be used to quickly obtain the step response of the closed loop system. The results obtained should be similar to those shown in Figure 6. These results show that the overshoot is approximately the 20% that was predicted. Also, as expected, there is no steady state error. However, the settling time is more than the 0.64 seconds that was anticipated, likely due to unmodeled friction in the system.

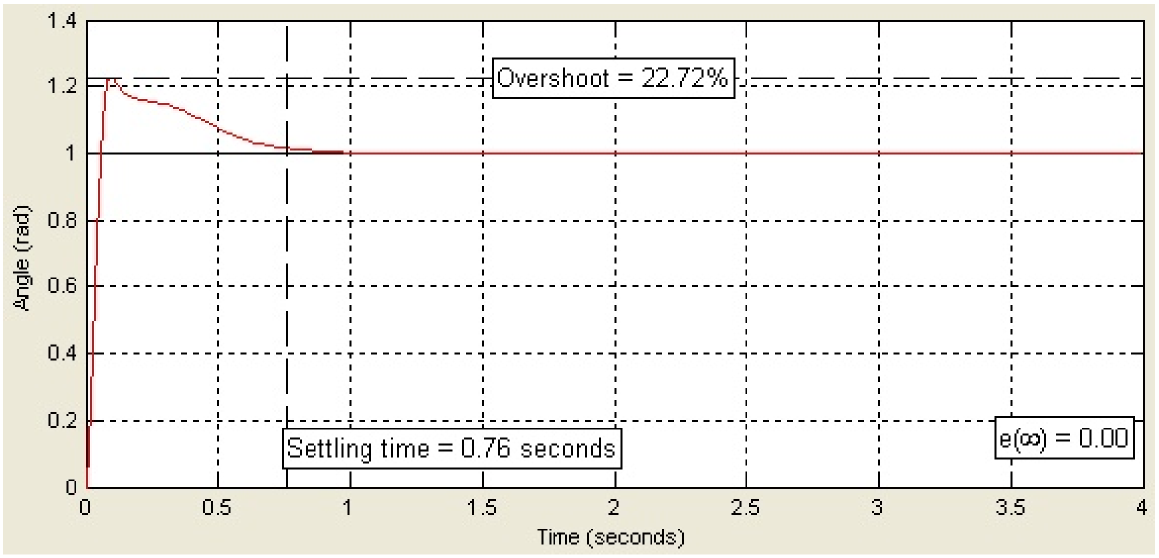


Figure 6 PID Example Control Results

6.2 PI Angular Rate Controller

The motor angular rate may be controlled using a proportional integral controller. Below is the transfer function of a proposed proportional integral rate controller that was designed for the QET.

$$G_c(z) = \frac{V(z)}{E(z)} = \frac{\mathrm{K}\,(\mathrm{z}-\mathrm{a})}{(\mathrm{z}-1)} \qquad \textbf{(5.3)}$$

where

V(z) = Motor voltage
E(z) = Angle Error = Desired Angle - Actual Angle
K = 0.038955
and a = 0.7778

This transfer function can be implemented using the following difference equation:

$$\mathrm{V}[\mathrm{k}] = \mathrm{V}[\mathrm{k}-1] + \mathrm{K}\cdot\{\mathrm{E}[\mathrm{k}] - \mathrm{a}\cdot\mathrm{E}[\mathrm{k}-1]\} \qquad \textbf{(5.4)}$$

The MATLAB simulation of this controller predicts a 5% overshoot and a settling time of 0.74 seconds as shown in the following figure.

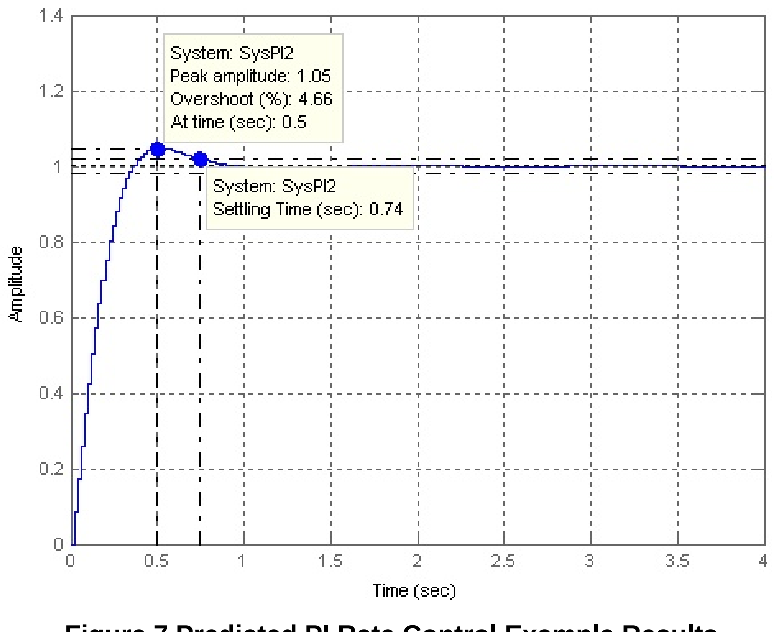


Figure 7 Predicted PI Rate Control Example Results

To implement this controller on the QET the user should define the controller gain and zero locations at the top of the "RunController" function in the "controller parameters" section. The following code shows an example of this.

```
//PI rate controller parameters
const float K = 0.038955;
const float a = 0.7778;
```

The code that calculates the reference signal, motor angle, and motor angular rate is already completed as discussed in the introduction to this section. Thus, once the controller parameters are defined the next step is to write the code to calculate the error at the current time step and save the necessary number of error measurements from previous time steps. From (5.4) it can be seen that only the error from the current and previous time steps are needed. The following code snippet is an example of what may be used to store the errors.

```
Errors[1] = Errors[0];              //Error at last time step
Errors[0] = RefSig - AngleFD;  //Error at current time step
```

For the previous code snippets to be valid the user must define the two element array “Errors” at the top of the function. The following is an example of the necessary variable definition.

```
static float Errors[2];
```

Once the controller parameters are defined the user must add the PI control logic to the "controller logic" section of the “RunController” function. The following code shows the implementation of difference equation (5.4).

```
Volt = VoltLast + K*(Errors[0] - a*Errors[1])
```

Once these code changes have been implemented the code can be compiled and loaded onto the 16F877 microcontroller as discussed in the help file. The GUI can then be used to quickly obtain the step response of the closed loop system. The results obtained should be similar to those shown in Figure 8. This figure shows that the overshoot is less than the 5% that was predicted. Also, the settling time is less than the 0.74 seconds that was anticipated. These better than expected results are likely due to unmodeled friction in the system. Note that this step response was run with a magnitude of 16π radians/second (8 revolutions per second) to reduce the impact of the unmodeled friction on the system.

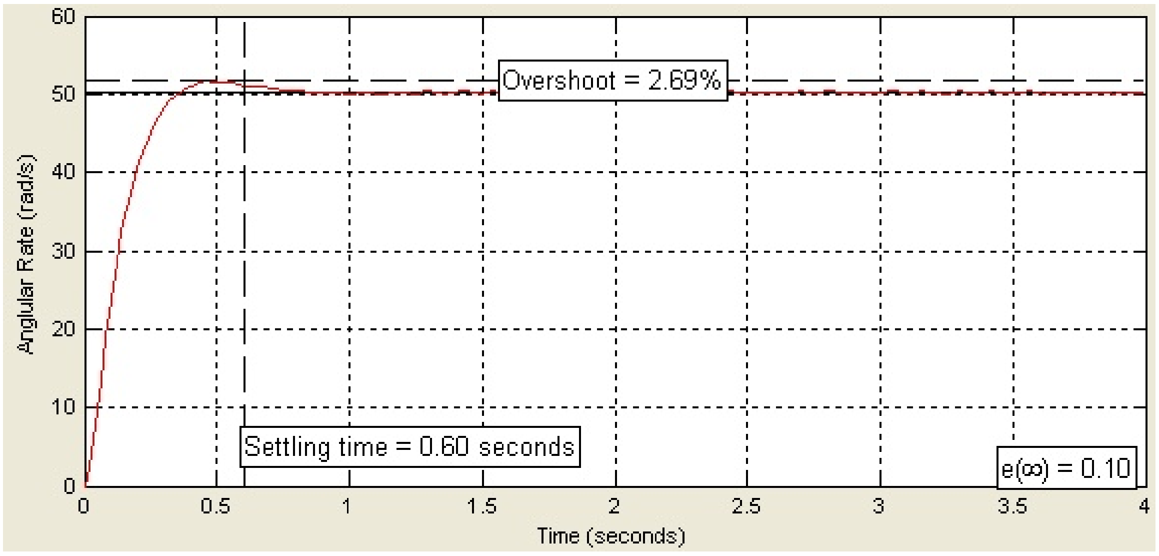


Figure 8 PID Example Control Results

7.0 Conclusion

Wichita State University currently offers various control courses, including a digital control course. However, there is currently not an available lab component for this digital control course. The goal of this project was to create a system that would allow a student to quickly implement and test a controller design. Toward this goal custom C code was written for the PIC microcontroller installed on the Quanser Engineering Trainers currently owned by Wichita State University. The necessary code modifications to implement a user designed controller were also discussed. A MATLAB graphical user interface was created that allows simple interaction with the microcontroller for the purposes of testing the controller performance. An HTML help file was also generated to assist in the use of the C code and MATLAB GUI. Finally, examples of the implementation of PID angular position and PI angular rate controllers were discussed.

8.0 Future Work

Although much has been accomplished towards the development of a control lab there is additional work that may prove helpful. The first would be to develop a lab manual. This manual should have topics in it similar to those covered by Quanser in their lab manual, such as system modeling and controller design.

Also, in this project only simple controllers were implemented. These were PD position, PID position, and PI rate controllers. In the future it would be interesting to develop and implement more advanced controllers such as two degree of freedom and robust controllers. Even a neural net controller may be implemented if it is simple enough to fit in the remaining ROM space on the PIC.

Finally, serial port communication errors are only minimally handled in the current PIC C code. Additional handshaking between MATLAB and the PIC may be helpful in ensuring more reliable operation. Due to the slow speed of MATLAB relative to the serial communication it may prove useful to move the serial communication itself out of MATLAB and into specifically written compiled code. Microsoft’s .NET framework provides a class for operating the serial port that may prove useful for this enhancement.

9.0 References

[1] Åström, Karl Johan. “Challenges in Control Education.” Lund University, Sweden, 2006.

[2] Alleyne, Andrew G., Block, Daniel J., Meyn, Sean P., Perkins, William R., and Spong, Mark W. “An Interdisciplinary, Interdepartmental Control Systems Laboratory.” IEEE Control Systems Magazine, v25, n1, February 2005.

[3] Choi, Chiu H. “Undergraduate Controls Laboratory Experience.” Proceedings of the 2004 American Society for Engineering Education Annual Conference and Exposition, Jun 20-23 2004.

[4] Durfeee, William, Li, Perry, and Waletzko, David. “Take-Home Lab Kits for System Dynamics and Controls Courses.” Proceedings of the 2004 American Control Conference, June 30-July 2, 2004.

[5] Dixon, Warren E., Dawson, Darren M., Costic, B. T., Queiroz, Marcio S. “A MATLAB-Based Control Systems Laboratory Experience for Undergraduate Students: Toward Standardization and Shared Resources.” IEEE Transactions on Education, v 45, n 3, Aug. 2002.

[6] Åström, Karl J., and Apkarian, Jacob. “A Laptop Servo for Control Education.” IEEE Control Systems Magazine, v24, n5, October 2004.

[7] “Products in Control: Engineering Trainer”, IEEE Control Systems Magazine, v24, n2, April 2004.

[8] Berstein, Dennis S. “The Quanser DC Motor Control Trainer.” IEEE Control Systems Magazine, v25, n3, June 2005.

[9] “User Guide: Quanser Engineering Trainer DC Motor Control”, Quanser Consulting Inc.

[10] “PICmicro™ Mid-Range MCU Family Reference Manual.” Microchip Technology Inc., December 1997.

[11] “PIC16F87X Data Sheet.” Microchip Technology Inc., 2001.

[12] Engle, Benjamin. “WSU QET Interface Help System.” November 2007.

Appendix A: PIC Microcontroller C Code

Below is a listing of the microcontroller routines developed for this project. The routines in the header files q_config.h, encoder.h, daout.h, and init_qmt.h were provided with the QET by Quanser and only slightly modified. As this code is shown it is setup for the PID position control example of Section 6.2.

Main Routines

```
#include "16f877.h"
#include "q_config.h"
#include "encoder.h"
#include "daout.h"
#include "init_qmt.h"

#include <math.h>
#include <string.h>
#include <stdlib.h>

#ZERO_RAM

#define CLK 20000000    //Clock frequency

#use delay(clock = CLK)                                      //Set clock frequency
#use rs232(baud=115200, bits=8, parity=N, xmit=PIN_C6, rcv=PIN_C7, errors)    //Setup
serial port

//User defined constants
#define  fs 50            //sampling frequency in Hz (10 Hz is the slowest supported
frequency)
#define  ts (1/(float)fs)  //sampling period in second

//Signal types
#define SIGIMP 1  //Impulse signal
#define SIGSTP 2  //Step signal
#define SIGRMP 3  //Ramp signal
#define SIGPAR 4  //Parabolic signal
#define SIGTRI 5  //Triangle wave signal
#define SIGSAW 6  //Sawtooth wave signal
#define SIGSQA 7  //Square wave signal
#define SIGSIN 8  //Sine wave signal
#define SIGCOS 9  //Cosine wave signal
#define SIGSTR 10 //Stair wave signal

//Signal information
int8  SigType;    //Current signal type (see above definitions)
float SigMagRate; //Signal magnitude or rate, depending on signal type
float SigFreq;    //Signal frequency (hz), if applicable to current signal type

//Global variables
short int DoControl;       //Flag as to whether to do control or not
short int Running;         //Flag as to whether we are running or not

long EndCnt;               //Count to end the program when reached

long TimeCnt;              //Last control time index
float RefSig;              //Current reference signal (desired angle or angular velocity)
float RefSigLast;          //Last  reference signal
float Angle;               //Last read angle in radians
float Volt;                //Last set motor voltage output
float Current;             //Last read motor current in milliamps
float AngleFD;             //Last calculated motor angle first derivative (rad/s) using
the direct method
float AngleFDFilt;         //Last calculated motor angle first derivative (rad/s) from
the filtering method

long SquareCount;          //Next count to flip square wave on
```

```
short int TriSgn;          //Current sign of triangle wave (0 means negative and 1 means
positive)

long preset_counter;       //Value to set counter to at each interrupt

#define BUFSIZ 5           //Recieving buffer size (DON'T CHANGE!)
char RecDatBuf[BUFSIZ];    //Recieving data buffer
long  iRecData;            //Next available location in data recieve buffer
short int RecDatAvail;     //Flag as to whether data has been recieved
int1 RecErr;               //Flag as to wheter a recieve error has occured

int1 FirstTime;

//Setup angular rate filter parameters
float Omega = 300.;
float ca;
float cb;

//Function prototypes
float CalcRefSignal(long TimeCnt);
float GetTri();
float GetSquare();

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
//The following routine will be called at the sample rate to generate the new
//reference signal as well as complete the actual control and write the output
void RunController()
{

   float AngleLast;
   float VoltLast;
   float AngleFDFiltLast;
   static float Errors[3];

   //-----------------------------------------------------------------------
   //Start of controller parameters
   //-----------------------------------------------------------------------

   //PID controller parameters
   const float k = 6.650160854277;
   const float a1 = 0.87508921708793;
   const float a2 = 0.91901606680500;

   //-----------------------------------------------------------------------
   //End of controller parameters
   //-----------------------------------------------------------------------

   DoControl = 0;

   //Turn on LED
   output_high(LED3);

   //Increment output count (as long as this isn't the first time)
   if (!FirstTime)
      TimeCnt++;

   //Read the encoder
   AngleLast = Angle;
   Angle = read_encoder(ENCODER0, X_AXIS);

   //Save current reference signal
   RefSigLast = RefSig;

   //Get new desired angle
   RefSig = CalcRefSignal(TimeCnt);

   //Read the current
   set_adc_channel(2);
   delay_us(20);
   Current = ((float)Read_ADC()-512.0)*0.00543*1000; //Read ADC and convert to
milliampere
```

```
   //Calculate first derivative of angle
   if (!FirstTime)
      {
      AngleFD = (Angle-AngleLast)/ts;

      AngleFDFiltLast = AngleFDFilt;
      AngleFDFilt = ca*(Angle-AngleLast) - cb*AngleFDFiltLast;
      }

   //---------------------------------------------------------------------------
   //Start of controller logic
   //---------------------------------------------------------------------------

   //Save current voltage
   VoltLast = Volt;

   //Set all errors to zero the first sample period
   if (FirstTime) memset(Errors, 0, sizeof(Errors));

   //Save errors for position control
   Errors[2] = Errors[1];        //Error two time steps ago
   Errors[1] = Errors[0];        //Error at last time step
   Errors[0] = RefSig - Angle;   //Error at current time step

   //Controller logic
   Volt = VoltLast + k*(Errors[0] - (a1+a2)*Errors[1] + a1*a2*Errors[2]);   //PID

   //---------------------------------------------------------------------------
   //End of controller logic
   //---------------------------------------------------------------------------

   //Ouptut controller voltage
   daout(Volt);

   //Turn LED off
   output_low(LED3);

}

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
float CalcRefSignal(long TimeCnt)
   {

   float NewSig, t;
   float TrigX;

   t = TimeCnt*ts;

   //Figure out the multiplier to the trig signals
   if (SigType == SIGSIN || SigType == SIGCOS)
      {
      TrigX = t*SigFreq;
      TrigX = TrigX - floor(TrigX);
      }

   switch (SigType)
      {

      //Impulse signal
      case SIGIMP:
         if (TimeCnt == 0)
            NewSig = (SigMagRate/ts);
         else
            NewSig = 0;

         break;

      //Step signal
      case SIGSTP:
         NewSig = SigMagRate;
         break;

      //Ramp signal
      case SIGRMP:
         NewSig = SigMagRate*t;
         break;

      //Parabolic signal
      case SIGPAR:
         NewSig = SigMagRate*t*t;
         break;

      //Triangle wave signal
      case SIGTRI:
         NewSig = GetTri();
         break;

      //Sawtooth wave signal
      case SIGSAW:

         NewSig = 2*SigMagRate*((t*SigFreq) - floor((t*SigFreq)+0.5));
         break;

      //Square wave signal
      case SIGSQA:
         NewSig = GetSquare();
         break;

      //Sine wave signal
      case SIGSIN:
         NewSig = SigMagRate*sin(TrigX*2*PI);
         break;

      //Cosine wave signal
      case SIGCOS:
        NewSig = SigMagRate*cos(TrigX*2*PI);
         break;

      //Stair wave signal
      case SIGSTR:
         NewSig = SigMagRate*(floor((t*SigFreq)+0.0000001)+1);

      }

      return NewSig;

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
float GetTri()
   {

   float NewSig, Rate;

   if (TimeCnt == 0)
      NewSig = 0;
   else
      {
      Rate = 4*SigMagRate*SigFreq;

      if (!TriSgn)
         Rate = -Rate;

      NewSig = RefSigLast + Rate*ts;

      if (NewSig > SigMagRate)
         {
         NewSig = 2*SigMagRate - NewSig;
         TriSgn = 0;
         }
      else if (NewSig < -SigMagRate)
```

```
         {
         NewSig = -2*SigMagRate - NewSig;
         TriSgn = 1;
         }
      }
   return NewSig;
   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
float GetSquare()
   {
   float NewSig;

   if (TimeCnt == 0)
      {
      SquareCount = fs/(2*SigFreq);
      NewSig = SigMagRate;
      }
   else if (TimeCnt == SquareCount)
      {
      SquareCount = SquareCount + fs/(2*SigFreq);
      NewSig = -RefSigLast;
      }
   else
      NewSig = RefSigLast;

   return NewSig;
   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
void FloatToIEE754(float *Val)
   {

   int1 Tmp;
   int8 *Ptr;

   Ptr = Val;

   Tmp = shift_left(&Ptr[1], 1, 0);
   Tmp = shift_right(&Ptr[0], 1, Tmp);
   shift_right(&Ptr[1], 1, Tmp);

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
void IEE754ToFloat(float *Val)
   {

   int1 Tmp;
   char *Ptr;

   Ptr = Val;

   Tmp = shift_left(&Ptr[1], 1, 0);
   Tmp = shift_left(&Ptr[0], 1, Tmp);
   shift_right(&Ptr[1], 1, Tmp);

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
#SEPARATE void OutputFloat(float ValIn)
   {

   char *c;
   float Val;

   Val = ValIn;

   //Save pointer to float so we can easily access the bytes individually
   c = &Val;
   //Convert number to IEEE 754
   FloatToIEE754(&Val);

   //Send data (order of bytes reversed since PC is expecting littleEndian)
   putc(c[3]);
   putc(c[2]);
   putc(c[1]);
   putc(c[0]);

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
#SEPARATE void OutputLong(long Val)
   {

   char *c;

   //Save pointer to float so we can easily access the bytes individually
   c = &Val;

   //Send data
   putc(c[0]);
   putc(c[1]);

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
#SEPARATE long GetLong()
   {

   char *c;
   long NewInt;

   c = &NewInt;

   c[0] = RecDatBuf[1];
   c[1] = RecDatBuf[2];

   return NewInt;

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
#SEPARATE float GetFloat()
   {

   char *c;
   float NewFloat;

   c = &NewFloat;

   c[0] = RecDatBuf[4];
   c[1] = RecDatBuf[3];
   c[2] = RecDatBuf[2];
   c[3] = RecDatBuf[1];

   IEE754ToFloat(&NewFloat);

   return NewFloat;

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
void SetSampleFrequency(float freq) {
   setup_timer_1(T1_INTERNAL | T1_DIV_BY_8);

   //The counter is incremented every 4 clocks.
   //When the counter resets (reaches 2^16 = 65536) it generates an interupt
   //This interupt calls the RunController function (above)
   //Setting the following value to the timer will cause it to interupt at the
   //desired frequency.
   preset_counter = 65536 - floor(((CLK/8)/4) / freq  );
   set_timer1(preset_counter);
}

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
void main()
   {

   //Local variables
   long LastOutputCnt;       //Index of the last dataset output
   char Cmd;
   int IntVal;

   //Initialize
   init_QMT();

   //Calculate angular rate filter parameters
   ca = 2*omega/(2+omega*ts);
   cb = (-2+omega*ts)/(2+omega*ts);

   //Choose current ADC channel
   set_adc_channel(2);

   init_encoder(ENCODER0);       //Initialize the encoder

   //Initialize outputs
   daout(0);                     //Send 0 voltage to DA output
   output_low(LED3);             //Set LED3 to low
   output_high(LED2);            //Set LED2 to high
   output_low(PIN_B4);           //Enable external D/A

   //Initialize variables
   DoControl = 0;
   RefSig = 0;
   RefSigLast = 0;
   Volt = 0;
   Current = 0;
   AngleFD = 0;
   AngleFDFilt = 0;

   SquareCount = 0;
   TriSgn = 1;

   iRecData = 0;
   RecDatAvail = 0;
   RecErr = 0;

   EndCnt = 0;
   TimeCnt = 0;
   LastOutputCnt = 0;

   Running = 0;
   FirstTime = 1;

   enable_interrupts(INT_RDA);   //Turn on recieve data available interrupt
   enable_interrupts(GLOBAL);    //Enable interrupts globally

   while(1)
      {

      if (Running == 1)
         {

         //Check if we should do control
         if (DoControl == 1)
            RunController();

         //Check if we need to ouput data
         if ((LastOutputCnt != TimeCnt) || FirstTime)
            {
            putc('d');
            OutputLong(TimeCnt);
            OutputFloat(RefSig);
            OutputFloat(Angle);
            OutputFloat(Volt);
            OutputFloat(Current);
            OutputFloat(AngleFD);
            OutputFloat(AngleFDFilt);

            LastOutputCnt = TimeCnt;
            FirstTime = 0;
            }

         //Check for end of time
         if (TimeCnt >= EndCnt)
            break;

         }

      //Check for data in read buffer
      if (RecDatAvail == 1)
         {

         RecDatAvail = 0;

         Cmd = RecDatBuf[0];

         if (RecErr)
            {
            //Ignore bad data
            RecErr = 0;
            }
         else
            {

            //Run the desired command
            if (Cmd == 'S')
               {
               //This is the new signal type
               SigType = GetLong();
               }
            else if (Cmd == 'M')
               {
               //This the new signal magnitude or rate
               SigMagRate = GetFloat();
               }
            else if (Cmd == 'F')
               {
               //This is the new signal frequency
               SigFreq = GetFloat();
              }
            else if (Cmd == 'E')
               {
               //This is the count at which to end
               EndCnt = GetLong();
               }
            else if (Cmd == 'G')
               {
               //This is the flag to GO!
               output_low(LED2);                         //Set LED2 to low
               Running = 1;

               SetSampleFrequency(fs);                   //Set sample frequency
               enable_interrupts(INT_TIMER1);            //Turn on timer1 interrupt
               enable_interrupts(GLOBAL);                //Enable interrupts globally
               }
             else if (Cmd == 'f')
               {
               //Sampling frequency request
               putc('f');
               OutputFloat(ts);
               }
```

```
           }
         }
      }

      //Send flag to tell the computer to stop reading data
      putc('X');

      //Stop the interrupts
      disable_interrupts(GLOBAL);                //Disable interrupts globally
      disable_interrupts(INT_TIMER1);            //Turn off timer1 interrupt
      disable_interrupts(INT_RDA);               //Turn off receive data available
interrupt

      //Stop motor by sending 0 voltage to DA output
      daout(0);

      //Wait 1 second for motor to stop spinning
      delay_ms(1000);

      reset_cpu();

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
//Use interrupt to recieve serial data when it is available
#INT_RDA
void ReadSerialData()
   {

   if ((RS232_ERRORS & 0x7) != 0)
      {
      //An error ocurred
      RecErr = 1;
      output_high(LED2);
      output_high(LED3);
     }
   else
      {
      //Save current transmission
      RecDatBuf[iRecData] = getc();
      iRecData++;

      //Check for full buffer
      if (iRecData >= 5)
         {
         RecDatAvail = 1;
         iRecData = 0;
         }
      }

   }

//#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
//Signal the compiler that the next function is the timer1 interrupt
#INT_TIMER1
void SetControlFlag()
   {

   //Reset the counter to the correct count
   //This is necessary to achieve the desired sample rate
   set_timer1(preset_counter);

   DoControl = 1;

   }
```

16F877.h Routines

```
//////// Standard Header file for the PIC16F877 device ////////////////
#device PIC16F877 *=16
#nolist
//////// Program memory: 8192x14  Data RAM: 367  Stack: 8
//////// I/O: 33   Analog Pins: 8
//////// Data EEPROM: 256
//////// C Scratch area: 77   ID Location: 2000
//////// Fuses: LP,XT,HS,RC,NOWDT,WDT,NOPUT,PUT,PROTECT,PROTECT_5%
//////// Fuses: PROTECT_50%,NOPROTECT,NOBROWNOUT,BROWNOUT,LVP,NOLVP,CPD
//////// Fuses: NOCPD,WRT,NOWRT,DEBUG,NODEBUG
////////
////////////////////////////////////////////////////////////////// I/O
// Discrete I/O Functions: SET_TRIS_x(), OUTPUT_x(), INPUT_x(),
//                         PORT_B_PULLUPS(), INPUT(),
//                         OUTPUT_LOW(), OUTPUT_HIGH(),
//                         OUTPUT_FLOAT(), OUTPUT_BIT()
// Constants used to identify pins in the above are:

#define PIN_A0  40
#define PIN_A1  41
#define PIN_A2  42
#define PIN_A3  43
#define PIN_A4  44
#define PIN_A5  45

#define PIN_B0  48
#define PIN_B1  49
#define PIN_B2  50
#define PIN_B3  51
#define PIN_B4  52
#define PIN_B5  53
#define PIN_B6  54
#define PIN_B7  55

#define PIN_C0  56
#define PIN_C1  57
#define PIN_C2  58
#define PIN_C3  59
#define PIN_C4  60
#define PIN_C5  61
#define PIN_C6  62
#define PIN_C7  63

#define PIN_D0  64
#define PIN_D1  65
#define PIN_D2  66
#define PIN_D3  67
#define PIN_D4  68
#define PIN_D5  69
#define PIN_D6  70
#define PIN_D7  71

#define PIN_E0  72
#define PIN_E1  73
#define PIN_E2  74

////////////////////////////////////////////////////////////////// Useful defines
#define FALSE 0
#define TRUE 1

#define BYTE int
#define BOOLEAN short int

#define getc getch
#define fgetc getch
#define getchar getch
#define putc putchar
#define fputc putchar
```

```
#define fgets gets
#define fputs puts

////////////////////////////////////////////////////////////////// Control
// Control Functions:  RESET_CPU(), SLEEP(), RESTART_CAUSE()
// Constants returned from RESTART_CAUSE() are:
#define WDT_FROM_SLEEP  3
#define WDT_TIMEOUT     11
#define MCLR_FROM_SLEEP 19
#define MCLR_FROM_RUN   27
#define NORMAL_POWER_UP 24
#define BROWNOUT_RESTART 26


////////////////////////////////////////////////////////////////// Timer 0
// Timer 0 (AKA RTCC)Functions: SETUP_COUNTERS() or SETUP_TIMER_0(),
//                              SET_TIMER0() or SET_RTCC(),
//                              GET_TIMER0() or GET_RTCC()
// Constants used for SETUP_TIMER_0() are:
#define RTCC_INTERNAL   0
#define RTCC_EXT_L_TO_H 32
#define RTCC_EXT_H_TO_L 48

#define RTCC_DIV_1      8
#define RTCC_DIV_2      0
#define RTCC_DIV_4      1
#define RTCC_DIV_8      2
#define RTCC_DIV_16     3
#define RTCC_DIV_32     4
#define RTCC_DIV_64     5
#define RTCC_DIV_128    6
#define RTCC_DIV_256    7


#define RTCC_8_BIT      0

// Constants used for SETUP_COUNTERS() are the above
// constants for the 1st param and the following for
// the 2nd param:

////////////////////////////////////////////////////////////////// WDT
// Watch Dog Timer Functions: SETUP_WDT() or SETUP_COUNTERS() (see above)
//                            RESTART_WDT()
//
#define WDT_18MS        8
#define WDT_36MS        9
#define WDT_72MS       10
#define WDT_144MS      11
#define WDT_288MS      12
#define WDT_576MS      13
#define WDT_1152MS     14
#define WDT_2304MS     15

////////////////////////////////////////////////////////////////// Timer 1
// Timer 1 Functions: SETUP_TIMER_1, GET_TIMER1, SET_TIMER1
// Constants used for SETUP_TIMER_1() are:
//      (or (via |) together constants from each group)
#define T1_DISABLED         0
#define T1_INTERNAL         0x85
#define T1_EXTERNAL         0x87
#define T1_EXTERNAL_SYNC    0x83

#define T1_CLK_OUT          8

#define T1_DIV_BY_1         0
#define T1_DIV_BY_2         0x10
#define T1_DIV_BY_4         0x20
#define T1_DIV_BY_8         0x30

////////////////////////////////////////////////////////////////// Timer 2
// Timer 2 Functions: SETUP_TIMER_2, GET_TIMER2, SET_TIMER2
// Constants used for SETUP_TIMER_2() are:
#define T2_DISABLED         0
#define T2_DIV_BY_1         4
#define T2_DIV_BY_4         5
#define T2_DIV_BY_16        6

////////////////////////////////////////////////////////////////// CCP
// CCP Functions: SETUP_CCPx, SET_PWMx_DUTY
// CCP Variables: CCP_x, CCP_x_LOW, CCP_x_HIGH
// Constants used for SETUP_CCPx() are:
#define CCP_OFF                         0
#define CCP_CAPTURE_FE                  4
#define CCP_CAPTURE_RE                  5
#define CCP_CAPTURE_DIV_4               6
#define CCP_CAPTURE_DIV_16              7
#define CCP_COMPARE_SET_ON_MATCH        8
#define CCP_COMPARE_CLR_ON_MATCH        9
#define CCP_COMPARE_INT                 0xA
#define CCP_COMPARE_RESET_TIMER         0xB
#define CCP_PWM                         0xC
#define CCP_PWM_PLUS_1                  0x1c
#define CCP_PWM_PLUS_2                  0x2c
#define CCP_PWM_PLUS_3                  0x3c
long CCP_1;
#byte   CCP_1    =                      0x15
#byte   CCP_1_LOW=                      0x15
#byte   CCP_1_HIGH=                     0x16
long CCP_2;
#byte   CCP_2    =                      0x1B
#byte   CCP_2_LOW=                      0x1B
#byte   CCP_2_HIGH=                     0x1C
////////////////////////////////////////////////////////////////// PSP
// PSP Functions: SETUP_PSP, PSP_INPUT_FULL(), PSP_OUTPUT_FULL(),
//                PSP_OVERFLOW(), INPUT_D(), OUTPUT_D()
// PSP Variables: PSP_DATA
// Constants used in SETUP_PSP() are:
#define PSP_ENABLED                     0x10
#define PSP_DISABLED                    0

#byte   PSP_DATA=                       8

////////////////////////////////////////////////////////////////// SPI
// SPI Functions: SETUP_SPI, SPI_WRITE, SPI_READ, SPI_DATA_IN
// Constants used in SETUP_SSP() are:
#define SPI_MASTER       0x20
#define SPI_SLAVE        0x24
#define SPI_L_TO_H       0
#define SPI_H_TO_L       0x10
#define SPI_CLK_DIV_4    0
#define SPI_CLK_DIV_16   1
#define SPI_CLK_DIV_64   2
#define SPI_CLK_T2       3
#define SPI_SS_DISABLED  1

#define SPI_SAMPLE_AT_END 0x8000
#define SPI_XMIT_L_TO_H  0x4000

////////////////////////////////////////////////////////////////// UART
// Constants used in setup_uart() are:
// FALSE - Turn UART off
// TRUE  - Turn UART on
#define UART_ADDRESS           2
#define UART_DATA              4


////////////////////////////////////////////////////////////////// ADC
// ADC Functions: SETUP_ADC(), SETUP_ADC_PORTS() (aka SETUP_PORT_A),
//                SET_ADC_CHANNEL(), READ_ADC()
// Constants used for SETUP_ADC() are:
#define ADC_OFF                0          // ADC Off
#define ADC_CLOCK_DIV_2    0x100
```

```
#define ADC_CLOCK_DIV_8     0x40
#define ADC_CLOCK_DIV_32    0x80
#define ADC_CLOCK_INTERNAL  0xc0          // Internal 2-6us


// Constants used in SETUP_ADC_PORTS() are:
#define NO_ANALOGS                          7    // None
#define ALL_ANALOG                          0    // A0 A1 A2 A3 A5 E0 E1 E2
#define AN0_AN1_AN2_AN4_AN5_AN6_AN7_VSS_VREF 1    // A0 A1 A2 A5 E0 E1 E2 VRefh=A3
#define AN0_AN1_AN2_AN3_AN4                 2    // A0 A1 A2 A3 A5
#define AN0_AN1_AN2_AN4_VSS_VREF            3    // A0 A1 A2 A5 VRefh=A3
#define AN0_AN1_AN3                         4    // A0 A1 A3
#define AN0_AN1_VSS_VREF                    5    // A0 A1 VRefh=A3
#define AN0_AN1_AN4_AN5_AN6_AN7_VREF_VREF 0x08    // A0 A1 A5 E0 E1 E2 VRefh=A3 VRefl=A2
#define AN0_AN1_AN2_AN3_AN4_AN5           0x09    // A0 A1 A2 A3 A5 E0
#define AN0_AN1_AN2_AN4_AN5_VSS_VREF      0x0A    // A0 A1 A2 A5 E0 VRefh=A3
#define AN0_AN1_AN4_AN5_VREF_VREF         0x0B    // A0 A1 A5 E0 VRefh=A3 VRefl=A2
#define AN0_AN1_AN4_VREF_VREF             0x0C    // A0 A1 A5 VRefh=A3 VRefl=A2
#define AN0_AN1_VREF_VREF                 0x0D    // A0 A1 VRefh=A3 VRefl=A2
#define AN0                               0x0E    // A0
#define AN0_VREF_VREF                     0x0F    // A0 VRefh=A3 VRefl=A2
#define ANALOG_RA3_REF         0x1         //!old only provided for compatibility
#define A_ANALOG               0x2         //!old only provided for compatibility
#define A_ANALOG_RA3_REF       0x3         //!old only provided for compatibility
#define RA0_RA1_RA3_ANALOG     0x4         //!old only provided for compatibility
#define RA0_RA1_ANALOG_RA3_REF 0x5         //!old only provided for compatibility
#define ANALOG_RA3_RA2_REF              0x8   //!old only provided for compatibility
#define ANALOG_NOT_RE1_RE2              0x9   //!old only provided for compatibility
#define ANALOG_NOT_RE1_RE2_REF_RA3      0xA   //!old only provided for compatibility
#define ANALOG_NOT_RE1_RE2_REF_RA3_RA2  0xB   //!old only provided for compatibility
#define A_ANALOG_RA3_RA2_REF            0xC   //!old only provided for compatibility
#define RA0_RA1_ANALOG_RA3_RA2_REF      0xD   //!old only provided for compatibility
#define RA0_ANALOG                      0xE   //!old only provided for compatibility
#define RA0_ANALOG_RA3_RA2_REF          0xF   //!old only provided for compatibility


// Constants used in READ_ADC() are:
#define ADC_START_AND_READ     7   // This is the default if nothing is specified
#define ADC_START_ONLY         1
#define ADC_READ_ONLY          6



//////////////////////////////////////////////////////////////////// INT
// Interrupt Functions: ENABLE_INTERRUPTS(), DISABLE_INTERRUPTS(),
//                      EXT_INT_EDGE()
//
// Constants used in EXT_INT_EDGE() are:
#define L_TO_H              0x40
#define H_TO_L                 0
// Constants used in ENABLE/DISABLE_INTERRUPTS() are:
#define GLOBAL                    0x0BC0
#define INT_RTCC                  0x0B20
#define INT_RB                    0x0B08
#define INT_EXT                   0x0B10
#define INT_AD                    0x8C40
#define INT_TBE                   0x8C10
#define INT_RDA                   0x8C20
#define INT_TIMER1                0x8C01
#define INT_TIMER2                0x8C02
#define INT_CCP1                  0x8C04
#define INT_CCP2                  0x8D01
#define INT_SSP                   0x8C08
#define INT_PSP                   0x8C80
#define INT_BUSCOL                0x8D08
#define INT_EEPROM                0x8D10
#define INT_TIMER0                0x0B20

#list
```

q_config.h Routines

```
////////////////////////////////////////////////////////////////////////////////////////
////////////////
//
//  q_config.h
//
//  This file does the general set up for the QIC Processor Core. This file should be the
included
//  on the first line of any program written by the user.
//  The user may add addtional line to this file as necessary, but they should note that
the CCS
//  compiler loads lines in order. The compiler essentially "inlines" the files in the
main source
//
//  This file is desiged to be compiled without change using the CCS compiler, version
3.051
//  The development environment used was the Microchip MPLAB IDE, with the CCS compiler
linked
//  as the C compiler.
//
//  Author: A.D.
//  Copyright 2002, Quanser Consulting Inc.
//
////////////////////////////////////////////////////////////////////////////////////////
////////////////


#fuses HS,NOWDT,NOPROTECT,PUT,NOLVP                     // high speed oscillator option
HS
                                                        // watchdog timer is DISABLED
                                                        // the code protection bits are
DISABLED

#DEVICE ADC=10                                          // set the ADC to 10 bit mode
#ZERO_RAM                                               // initialize all RAM locations
to zero by default

#define PUSHBUTTON PIN_B0  //define PUSHBUTTON as PIN_B0
#define LED3 PIN_C0         //define LED as PIN_C0
#define LED2 PIN_C1        //define LED as PIN_C1

// The following lines set up the port address so that we can use direct reads and
// writes to memory locations to read and write ports directly.
// this is the most efficient way of reading / writing 8 bits ( a byte ) of data
// to / from a port.

#byte PORTA = 5
#byte PORTB = 6
#byte PORTC = 7
#byte PORTD = 8
#byte PORTE = 9

// Specify the ports to use the fast_io option. care must be taken when using this option
as
// the compiler does not add any extra code to set up the port. The bits in a port
// must be set up ahead of time to be either inputs or outputs.

#use fast_io(a)
//#use fast_io(b)
#use fast_io(c)
#use fast_io(d)
#use fast_io(e)
```

encoder.h Routines

```
#byte PORTD = 8

#define ALL_OUT         0x00
#define ALL_IN          0xFF

#define SLAVE0          0x00
#define SLAVE1          0x01

#define READ_ENC        PIN_A4
#define WRITE_ENC       PIN_B3

// macros to enable / disable the encoder
#define DISABLE_DEV     output_high(PIN_B1); output_high(PIN_B2);
#define ENABLE_ENC      output_high(PIN_B1); output_low(PIN_B2);
#define DISABLE_ENC     output_low(PIN_B1); output_low(PIN_B2);
#define ENABLE_DA       output_high(PIN_B2); output_low(PIN_B1);


#define ENC_AXIS        PIN_C5       // '0' = X axis ; '1' = Y axis
#define ENC_DC          PIN_C4       // '0' = DATA   ; '1' = COMMAND

#define DA_BYTE_A0            PIN_C4 // DA Nibble select A0
#define DA_BYTE_A1            PIN_C5 // DA Nibble select A1

#define ENCODER0        0x00
#define ENCODER1        0x01
#define X_AXIS          0x00
#define Y_AXIS          0x04
#define ENC_DATA                 0x00        // data register of ENCODER CHANNEL
#define ENC_CONTROL           0x08           // control register ENCODER CHANNEL
#define CLOCK_DATA            0      // FCK frequency divider
#define CLOCK_SETUP           0X98   // transfer PR0 to PSC (x and y)
#define INPUT_SETUP           0XC1   // enable inputs A and B (x and y)
#define QUAD_X1               0XA8   // quadrature multiplier to 1 (x and y)
#define QUAD_X2               0XB0   // quadrature multiplier to 2 (x and y)
#define QUAD_X4               0XB8   // quadrature multiplier to 4 (x and y)
#define BP_RESET              0X01   // reset byte pointer
#define BP_RESETB             0X81   // reset byte pointer (x and y)
#define CNTR_RESET            0X02   // reset counter
#define CNTR_RESETB           0X82          // reset counter (x and y)
#define TRSFRPR_CNTR          0X08   // transfer preset register to counter
#define TRSFRCNTR_OL          0X90   // transfer CNTR to OL (x and y)
#define EFLAG_RESET           0X86   // reset E bit of flag register
#define ENABLE_INDEX    0xE7         // RCNTR pin is indexed
#define DISABLE_INDEX   0xE6         // RCNTR pin is not indexed

// global variables
float calx1, calx1_mid, calx1_high, offset_x1;

// data is returned in these variables.
// see the example file encoder.c on how to reconstruct them
// to form an angular measurement

float calculate_angle();
void calibration_parameters();

static int encoder_data[3];


/**
 * The inp(address) function reads from the proper encoder register based on
 * what address value is passed in. The function uses a bit mask to determine
 * which register is being read and sets up the addressing correctly to return
 * the proper data.
 * Valid values for address should be X_AXIS and Y_AXIS, ored with the
 * the encoder data register address (ENC_DATA) as defined in the #define
 * statements of the header.
 * Returns: the value read from the encoder
 **/
int inp(int address){
  int data;

             // PORT D is the bus connected to the encoder
  set_tris_D(ALL_IN);                             // set PORT D to INPUT
  output_low(READ_ENC);                           // set the READ (/RD) line low
  ENABLE_ENC;                                     // set the ENABLE (/CS) line low
                                                  // the chip is now ready to read
  output_bit(ENC_DC, bit_test(address, 3));       // set the D/C bit to read DATA or
                                                  // control based on the address
passed in
  output_bit(ENC_AXIS, bit_test(address, 2));     // set the appropriate axis bit (X or
Y)
                                                  // based on the address passed in
  delay_cycles(1);                                // wait for the data to be ready,
same as NOP
  data = PORTD;                                   // read in the data from PORT D
  output_high(READ_ENC);                          // return the /RD line high
  set_tris_D(ALL_IN);                             // set PORT D to INPUT
  DISABLE_DEV;                                    // disable the encoder (/CS high)
  return(data);
}

/**
 * The outp(address, value) function sends a value out to the proper register of the
 * encoder IC. The function receives two parameters, the address of the register to
 * write to, and the value to be sent to that register.
 * The function uses a bit mask to determine the proper register to write to and
 * to set the appropriate address bits.
 * Note that when writing to the preset registers of the encoder IC, 3 consecutive
 * writes are needed to fill the internal register.
 **/
void outp(int address, int value){
  set_tris_D(ALL_OUT);                            // set PORT D to output
  output_low(WRITE_ENC);                          // set the /WR line low
  ENABLE_ENC;                                     // enable the encoder IC
  output_bit(ENC_DC, bit_test(address, 3));       // set the D/C bit appropriately
based on the
                                                  // the address argument
  output_bit(ENC_AXIS, bit_test(address, 2));     // set the axis bit appropriately
based on
                                                  // the address argument
  delay_cycles(1);                                // wait for the operation to complete
(NOP)
  PORTD = value;                                  // set PORT D to the data value
passed in
  output_high(WRITE_ENC);                         // return the /WR line high
  set_tris_D(ALL_IN);                             // set PORT D to output
  DISABLE_DEV;                                    // disable the encoder IC
}

/**
 * This function is designed to setup the encoder for the most common functions.
 * This function *must* be called prior to performing any reads of the encoder
 * to ensure that it is setup properly.
 * In most instances, this function can be left as is, but it can be modified
 * by the user to meet their needs.
 **/

void init_encoder(int address)
{

        /*initialize the ENCODER CHIP*/
   calibration_parameters();

//while(0) // from MQ3.DRV

        outp(address|ENC_CONTROL, EFLAG_RESET);      // reset E bit of flag register
        outp(address|ENC_CONTROL, BP_RESET);         // reset byte pointer (x and y)
```

```
       outp(address|ENC_DATA, CLOCK_DATA);          // FCK frequency divider
       outp(address|ENC_CONTROL, CLOCK_SETUP);      // transfer PR0 to PSC (x and y)
       outp(address|ENC_CONTROL, INPUT_SETUP);      // enable inputs A and B (x and y)
       outp(address|ENC_CONTROL, QUAD_X4);          // quadrature multiplier to 4 (x and
y)
       outp(address|ENC_CONTROL, CNTR_RESET);       // reset counter (x and y)

while(0)
{
  outp(address | ENC_CONTROL, EFLAG_RESET);         // reset the error bit in the FLAGs
register
  outp(address | ENC_CONTROL, BP_RESETB);           // reset the byte pointer
  outp(address | ENC_DATA| X_AXIS, CLOCK_DATA);     // setup the clock for the X axis
  outp(address | ENC_DATA| Y_AXIS, CLOCK_DATA);     // setup the clock for the Y axis
  outp(address | ENC_CONTROL, BP_RESETB);           // reset the byte pointer
  outp(address | ENC_CONTROL, CLOCK_SETUP);         // transfer PR0 to PSC (x and y)
  outp(address | ENC_CONTROL, QUAD_X4);             // read the encoder in quadrature (x4
mode)
  outp(address | ENC_CONTROL, INPUT_SETUP);         // enable inputs A and B (X and Y)
  outp(address | ENC_CONTROL, CNTR_RESETB);         // reset the counters for X and Y
axis

  // added by A.D. to disable index pulse
  outp(address | ENC_CONTROL, 0xE0);

}


}

float preset_encoder(int address, int axis, int low, int mid, int high)
{

       outp(address|ENC_CONTROL|axis, BP_RESETB);   // reset byte pointer
       outp(address|ENC_DATA|axis,low );
       outp(address|ENC_DATA|axis,mid );
       outp(address|ENC_DATA|axis,high);            // output preset value
   outp(address | ENC_CONTROL |axis , TRSFRPR_CNTR);// transfer the counters to the ouput
latches

}

/**
 * This function read_encoder(axis) is used to read the encoder and return the value
 * of the registers stored in the array encoder_data. The user will then need to
 * reconstruct the data to form a 24 bit word. encoder_data[0] is the lowest byte, and
 * encoder_data[2] is the highest byte. To
 *
 */
float read_encoder(int address, int axis){
  outp(address | ENC_CONTROL, BP_RESETB);           // reset the byte pointers so that we
read the

// registers in the proper order
  outp(address | ENC_CONTROL | axis , TRSFRCNTR_OL);// transfer the counters to the ouput
latches
  encoder_data[0] = inp(ENC_DATA | axis);           // read the first byte, store it.
  encoder_data[1] = inp(ENC_DATA | axis);           // read the second byte, store it.
  encoder_data[2] = inp(ENC_DATA | axis);           // read the third byte, store it.

  return calculate_angle();

  // the data is stored in the variables encoder_data[0-2] as
  // individual bytes to make up a 24 bit word
}

float calculate_angle()
{

   float high_m, angle_r;
```

```
    //Mask off the sign bit to do angular calculation
    high_m = encoder_data[2]&0x7f;
    angle_r = calx1 * encoder_data[0]+ calx1_mid * encoder_data[1] + calx1_high * high_m;

    // compute the offset if the angle was negative
    if((encoder_data[2]&0x80) == 0x80)
    {
          angle_r = angle_r - offset_x1;
    }

    return angle_r;
}

void calibration_parameters()
{

 // calculate the calibration parameters - radians
 calx1 = 0.00153398078789 ; //2pi/4096.;            //conversion from encoder counts to
degrees

 calx1_mid = 0.39269908169872; //calx1*256;          //multiplication factor for the
middle byte (in degrees)

 calx1_high = 1.005309649148734e+002; //calx1_mid*256;      // multiplication factor for
the upper byte (in degrees)

   offset_x1 =   1.286796350910379e+004;// calx1_high*128.;        // factor to account
for negative values

}
```

daout.h Routines

```
float voltageToIntegerValue(float);

//Digital to Analog output
void daout(float voltage)
{
    int low_byte,high_nibble;
    long value;
    value = voltageToIntegerValue(voltage);

    low_byte = value&0xff;
    set_tris_D(ALL_OUT);                     // set PORT D to output
    ENABLE_DA;
    output_bit(DA_BYTE_A0, 0 );              // Bring A0 low
    output_bit(DA_BYTE_A1, 0 );              // Bring A1 low
                                      // enable the D/A Chip
    PORTD = low_byte;                 // set PORT D to the value of the low 8 bits
    output_low(WRITE_ENC);            // set the /WR line low
    delay_cycles(1);
    output_high(WRITE_ENC);           // return the /WR line high


    //ENABLE_DA;
    high_nibble = value>>8;
    high_nibble = high_nibble&0x0f;

    PORTD = high_nibble;                     // set PORT D to the data value of the high
4 bits
    output_bit(DA_BYTE_A0, 1 );              // Bring A0 high
    output_bit(DA_BYTE_A1, 1 );              // Bring A1 high
    output_low(WRITE_ENC);            // set the /WR line low

    //delay_cycles(1);        // wait for the operation to complete (NOP)
    output_high(WRITE_ENC);   // return the /WR line high
    set_tris_D(ALL_IN);               // set PORT D to input - must do otherwise it loads
the system
    DISABLE_DEV;
}


float voltageToIntegerValue(float voltage)
    {
    long integer_da=0;

    voltage = voltage*0.33;

    if (voltage >= 4.99)
        voltage = 4.99;
    if (voltage <= -4.99)
        voltage = -4.99;

   integer_da = (2047 + (long)((voltage / 5.0) * (float)2047));    //Convert voltage to
equiv integer value


    return integer_da;        //Return integer value

    }
```

init_qmt.h Routines

```
void init_QMT()
{
   set_tris_a(0b1101111);            // RA0-3 analog inputs, RA4 output, RA5 input
   set_tris_b(0b11000001);           // B all out except for B0 - connected to switch
   port_b_pullups(TRUE);
   set_tris_c(0b10001100);           /* PIN RC7 is serial RX from HOST */
   set_tris_d(ALL_IN);               /* PORTD is all digital OUTPUTS */
   set_tris_e(ALL_IN);               /* PORTE is all digital OUTPUTS */

   output_high(PIN_B3);              // set write and read high
   output_high(PIN_A4);

   output_high(PIN_C4);              // set data registers selects high
   output_high(PIN_C5);

   output_high(PIN_B2);              // chip selects to high to select inexistent device
   output_high(PIN_B1);

   output_low(PIN_C0);               // turn off both LED's
   output_low(PIN_C1);

   output_high(PIN_B4);              //Turn off the extrenal D/A
   daout(2047);                      // put out zero volts

   output_low(PIN_B6);               //Stream indicator

   setup_adc_ports(ANALOG_RA3_REF); /* RE2 - RE0 are DIGITAL I/O and RA3 is REF*/
    setup_adc(ADC_CLOCK_INTERNAL);
}
```

Appendix B: MATLAB QET Interface Code

The following is a listing of the MATLAB code that uses the serial port to interface with the QET. This function is called by the GUI to complete the user specified control task.

```
function [Data, DataCount, SamplePeriod, Error] = ...
    RunQET(SigType, SigMagRate, SigFreq, RunTime, hScopeCallback, handles)

%RunQET - Runs a case on the Quanser engineering trainer and returns the
%results
%
%   [Data, DataCount, SamplePeriod, Error] = RunQET(SigType, SigMagRate,
%   SigFreq, RunTime, hScopeCallback, handles) runs the specified case on
%   the Quanser Engineering Trainer and returns a matrix of results along
%   with the number of time samples, the sample period, and an error flag.
%
%   The input SigType specifies the signal type to run.  This can be one of
%   the following values:
%     1 - Impulse signal
%     2 - Step signal
%     3 - Ramp signal
%     4 - Parabolic signal
%     5 - Triangle wave signal
%     6 - Sawtooth wave signal
%     7 - Square wave signal
%     8 - Sine wave signal
%     9 - Cosine wave signal
%     10 - Stair step signal
%
%   The input SigMagRate defines the signal magnitude or rate depending on
%   the signal type.  For a ramp or parabolic signal it specifies a rate.
%   For all the other signal types it specifies a magnuitude.  For more
%   information please see the associated help file.
%
%   The input SigFreq specifies the signal frequency in Hz, if applicable
%   to the speicifed signal type.  If the frequency is not applicable to
%   the specified signal type (such as for a step signal) then this should
%   be set to 0.
%
%   The input RunTime specifies the total run time in seconds.
%
%   The input hScopeCallback specifies the handle to the scope callback
%   procedure.  If the callback isn't desired then this value should be set
%   to zero.
%
%   The input handles is the handles array that will be passed to the scope
%   callback function.  If the scope callback value is set to zero then
%   this should also be set to zero.
%
%   The output Data contains all of the returned data.  This array will
%   have DataCount rows in it and will contain seven columns.  The columns
%   are:
%       1 - The data index for this row returned by the QET
%       2 - The reference signal
%       3 - The actual angle of the motor
%       4 - The voltage being applied to the motor
%       5 - The current being drawn by the motor
%       6 - The first derivative of the motor angle using the simple method
%       7 - The first derivative of the motor angle using the filtering method
%
%   The output SamplePeriod is the sample period, in seconds, being used by the QET
%
%   The output Error is a boolean flag as to whether an error occured.
%
%   For additional information please see the help file.

    Data = [];
    DataCount = 0;
    SamplePeriod = 0;
    Error = true;
    if nargout ~= 4
        msgbox('Invalid number of return arguments for RunQET!')
        return
    end

    if nargin < 4
        msgbox('Invalid number of arguments for RunQET!')
        return
    end

    %Set current elapsed time
    TimeNow = 0;
    DataCount = 0;

    %Close all serial port objects
    CloseAllSerialPorts

    %Create serial port object
    S = serial(ComPortName, 'BaudRate', 115200, 'Parity', 'none', 'DataBits', 8,
'StopBits', 1, 'InputBufferSize', 50000, 'Timeout', 1);

    %Open serial port
    fopen(S);

    %Get sampling frequency
    pause(0.1);

    if ~WriteToQET(S, 'f', 'uchar') fclose(S); return; end;
    if ~WriteToQET(S, 0, 'float32') fclose(S); return; end;

    PauseCount = 0;
    PauseStep = 0.1;
    while S.BytesAvailable < 5
      pause(PauseStep);
      PauseCount = PauseCount + 1;

      if PauseCount > 2/PauseStep
          S
          msgbox('The QET is not responding.  Please verify it is reset and try again')
          fclose(S);
          return
      end

    end

    Cmd = fread(S, 1, 'uchar');

    if Cmd ~= 'f'
        disp('Unexcected return value for frequency');
        return
    end

    SamplePeriod = fread(S, 1, 'float32');

    %Determine number of signals to read
    EndCnt = ceil(RunTime/SamplePeriod-0.0001);

    if (EndCnt > 2^16)
        msgbox('Too long of a runtime was entered for the sample rate.  Please choose a
shorter run time or lower the sample rate.  Reset the QET and try again.')
        return
    end

    %Set the end of time count on QET
    if ~WriteToQET(S, 'E', 'uchar') fclose(S); return; end;
    if ~WriteToQET(S, EndCnt, 'int16') fclose(S); return; end;
    if ~WriteToQET(S, 0, 'int16') fclose(S); return; end;

    %Initially size data array
    Data = zeros(EndCnt,7);
    %Set reference signal type and data
    if ~WriteToQET(S, 'S', 'uchar') fclose(S); return; end;
    if ~WriteToQET(S, SigType, 'int16') fclose(S); return; end;
    if ~WriteToQET(S, 0, 'int16') fclose(S); return; end;

    if ~WriteToQET(S, 'M', 'uchar') fclose(S); return; end;
    if ~WriteToQET(S, SigMagRate, 'float32') fclose(S); return; end;

    if ~WriteToQET(S, 'F', 'uchar') fclose(S); return; end;
    if ~WriteToQET(S, SigFreq, 'float32') fclose(S); return; end;

    %Send start flag
    if ~WriteToQET(S, 'G', 'uchar') fclose(S); return; end;
    if ~WriteToQET(S, 0, 'float32') fclose(S); return; end;

    %Read data
    Cmd = 0;
    BytesReq = 0;

    RunLoop = true;
    while RunLoop

        %disp(S.BytesAvailable);
        if Cmd == 0

            %Read command character if it is available
            if S.BytesAvailable >= 1

                Cmd = fread(S, 1, 'uchar');

                %Determine number of bytes required to complete this command
                if Cmd == 'X'
                    %This is the stop flag
                    Cmd = 0;
                    BytesReq = 0;
                    RunLoop = false;
                    break;
                elseif Cmd == 'd'

                    %This is the data flag (1 long integer and 4 floating point numbers)
                    BytesReq = 2 + 4*4;

                    DataCount = DataCount + 1;

                end

            end

        end

        %Check for command data available
        if Cmd ~= 0 && BytesReq ~= 0

            if S.BytesAvailable >= BytesReq

                if Cmd == 'd'

                    %Read data
                    QETDataIndex = fread(S, 1, 'uint16');
                    RefSig = fread(S, 1, 'float32');
                    ActAng = fread(S, 1, 'float32');
                    Volt = fread(S, 1, 'float32');
                    Current = fread(S, 1, 'float32');
                    AngFD = fread(S, 1, 'float32');
                    AngFDFilt = fread(S, 1, 'float32');

                    %Save data
                    Data(DataCount, 1) = QETDataIndex;
                    Data(DataCount, 2) = RefSig;
                    Data(DataCount, 3) = ActAng;
                    Data(DataCount, 4) = Volt;
                    Data(DataCount, 5) = Current;
                    Data(DataCount, 6) = AngFD;
                    Data(DataCount, 7) = AngFDFilt;

                    if nargin == 6 && isa(hScopeCallback, 'function_handle')
                        hScopeCallback(QETDataIndex*SamplePeriod, RefSig, ActAng, Volt,
Current, AngFD, handles);
                    end

                end

                Cmd = 0;

            end

        end

    end

    %Close serial port and clean up
    fclose(S);
    delete(S);
    clear S;

    Error = false;

function CloseAllSerialPorts

    Ports = instrfind({'Port'}, {ComPortName});

    for i = size(Ports,1):-1:1
        fclose(Ports(i));
        delete(Ports(i));
    end

function [PortName] = ComPortName()

    if evalin('base', 'exist(''QETPortName'')')
        PortName = evalin('base', 'QETPortName');
    else
        PortName = 'COM1';
    end

function [Success] = WriteToQET(S, Val, Format)

    %Since MATLAB sporadically times out on this statement we'll try it up to 5 times
    %KNOWN BUG: see http://www.mathworks.com/support/bugreports/details.html?rp=250986
    Success = false;
    for i=1:5

        Errored = false;
        try
            fwrite(S, Val, Format);
        catch
            Errored = true;
        end

        if ~Errored
            Success = true;
            break;
        elseif i == 5
            disp('Unable to write to QET device.  Please reset it and try again.');
            return;
        end

    end
```

Appendix C: PIC C Code Flow Charts

Flowchart Discussion

Flowcharts of the pertinent PIC C code are available in Figure 9 through Figure 10. Figure 9 contains a flowchart of the main function that is run whenever the PIC is started (power is applied or the reset button is pressed). Figure 10 contains a flowchart of the function that is run whenever the “Run Control” process block is encountered in the main function.

While the main function is executing there are two other functions that may be run when specific events occur. These events are the timer interrupt and the serial RDA (received data available) interrupt. The timer interrupt occurs whenever the amount of time specified by the sample rate (variable name “fs”) has passed since the last timer interrupt (or since the time when the interrupt was enabled). The serial RDA interrupt occurs whenever a single character (1 byte) has been received over the serial port. When these events occur the execution of the main function is stopped (interrupted) and the appropriate function is executed. A flowchart of the function that is executed when the timer interrupt occurs is shown in Figure 11. A flowchart of the function that is executed when the serial RDA interrupt occurs is shown in Figure 12.

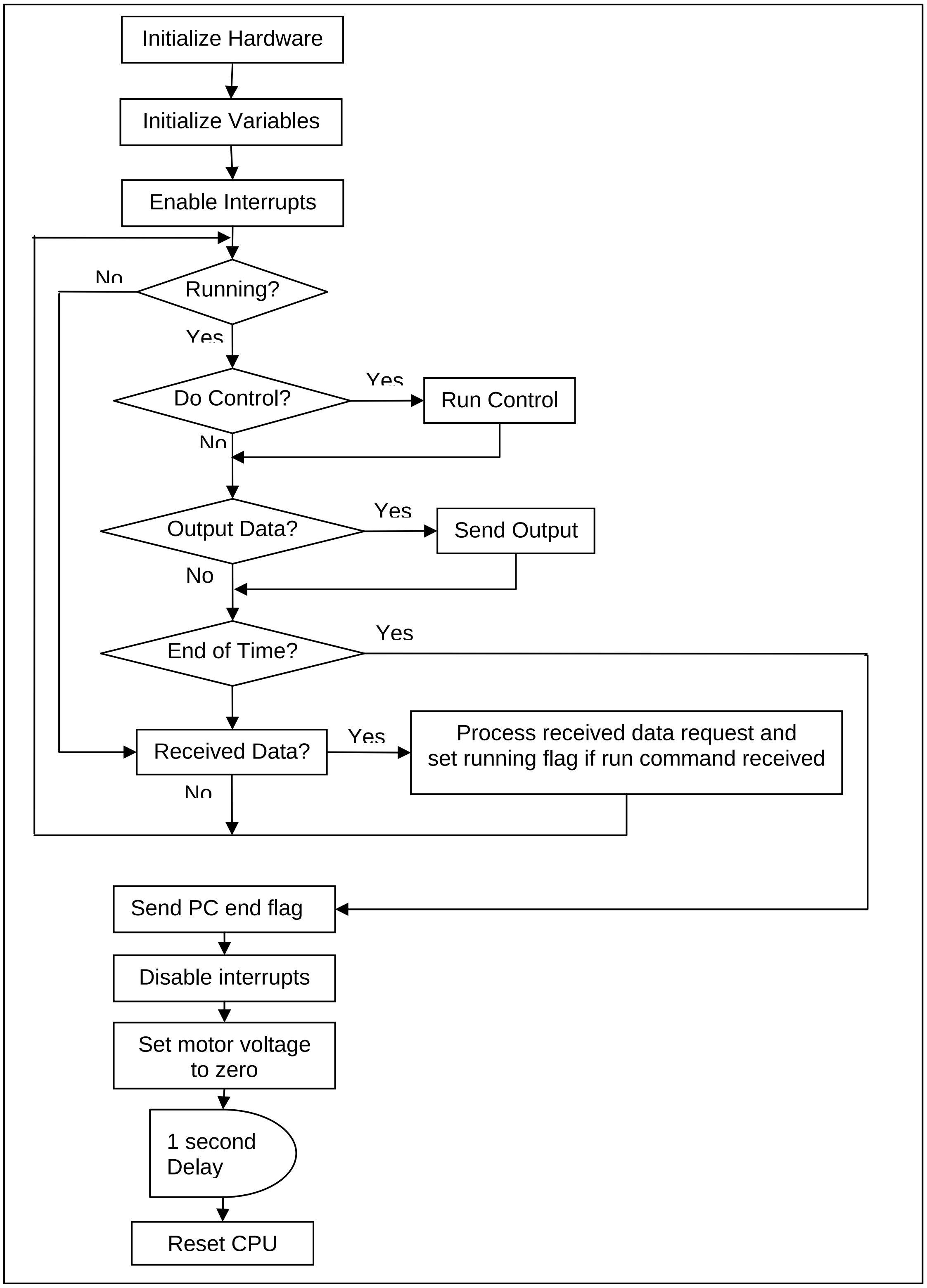


Figure 9 Main Function Flowchart

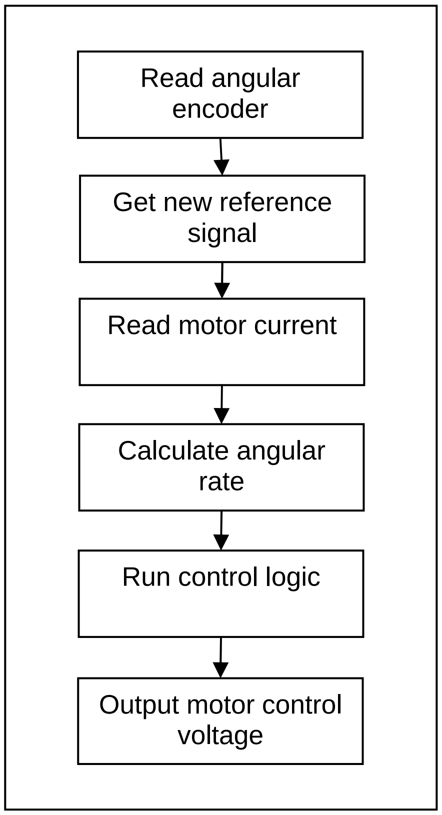


Figure 10 Run Control Function

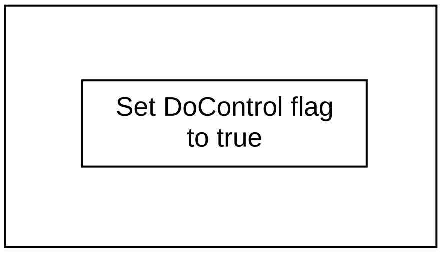


Figure 11 Timer Interrupt Function Flowchart

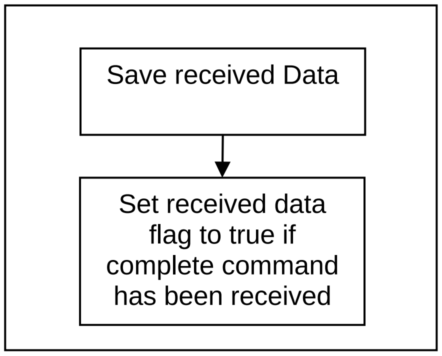

Save received Data
Set received data flag to true if complete command has been received

Figure 12 Serial Received Data Available Interrupt Function Flowchart