Ryerson University
Department of Electrical, Computer, and Biomedical Engineering
ELE709 — Real-Time Computer Control Systems

W2021 Project — Digital PID Controller

1 Objectives

The objective of this project is to design and implement a digital PID controllers (basic and anti-windup) to control the position of a DC motor using concurrent programming with Pthreads.

The objective of Part A is to design the PID controller using the Ultimate Sensitivity Method, and to develop and test the control program using simulation functions.

The objective of Part B of the project is to further extend the functionality of the control program.

2 Part A: PID Controller Design, Program Development and Simulation

In a digital control system, the control variables are computed using a digital computer. In computing the control variables, the process variables which are normally continuous-time signals must be converted into numbers. After the control variables are computed, they must then be converted into continuous-time signals before applying to the system which is being controlled.

The process of converting a continuous-time signal into a sequence of numbers is known as *sampling*, and the process of converting a sequence of numbers into a continuous-time signal is known as *data reconstruction*. In a digital control system (see Figure 1), sampling and data reconstruction are performed using A/D and

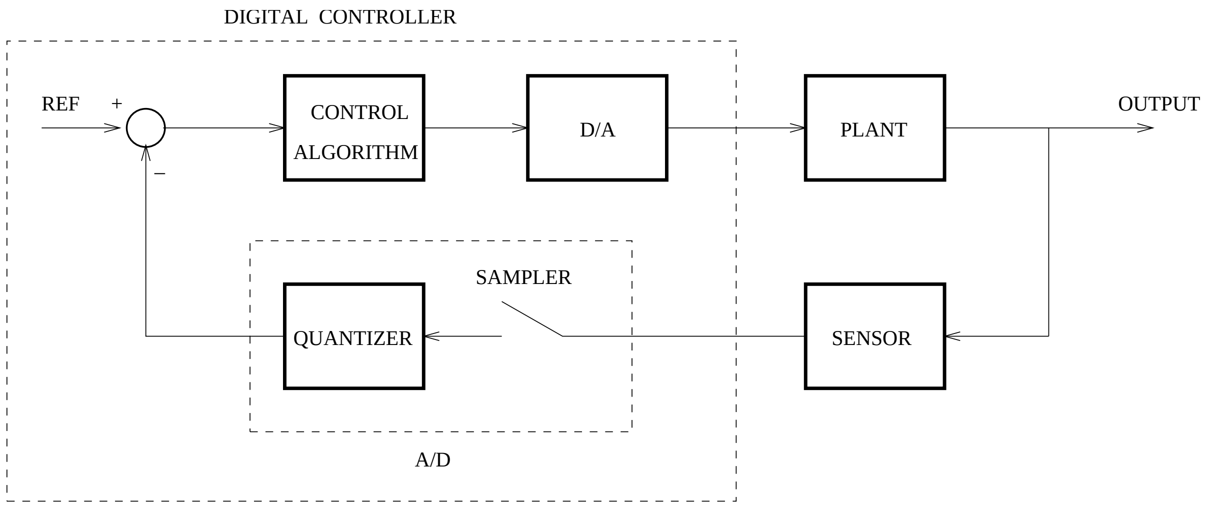


Figure 1: A Digital Control System

D/A converters respectively. An A/D converter not only samples the input, it also quantized the input signals according to the word size of the converter. For example, a 16-bit A/D converter will divide the range of the input signal into 2^{16} equal levels.

A block diagram of the various components of the laboratory module is shown in Figure 2. In the

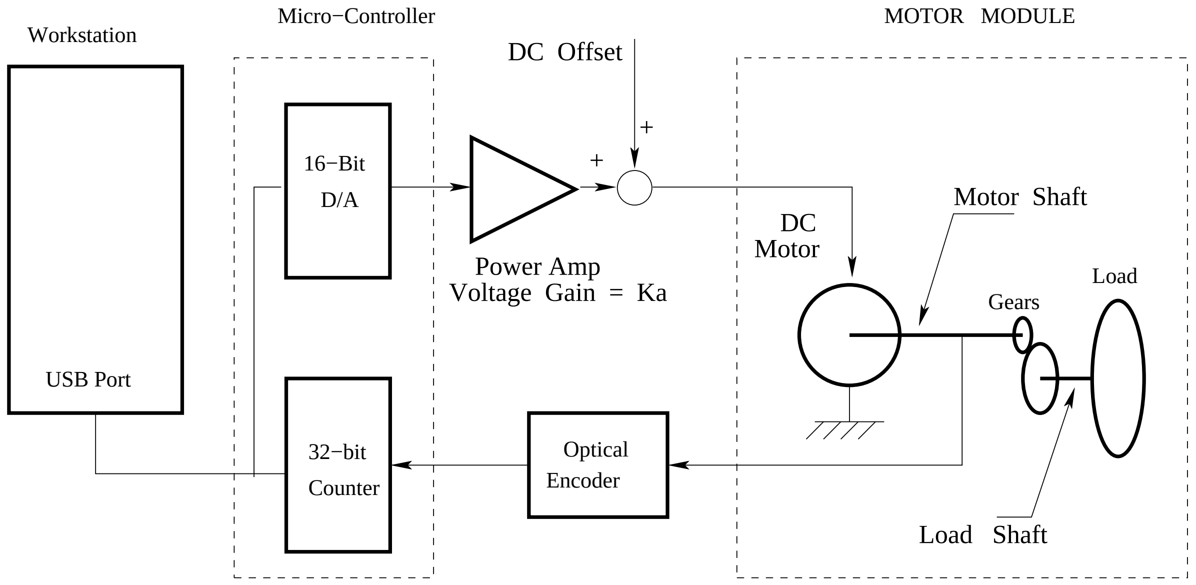


Figure 2: Block Diagram of Laboratory Module

laboratory module, sampling and reconstruction of continuous-time signals are performed using hardware available on the micro-controller board. In particular, data reconstruction is performed using a 16-bit D/A converter. Even though A/D converters are also available on the micro-controller board, they are not used because the angular position of the motor's load shaft can be obtained with an optical encoder with digital outputs. Specifically, the micro-controller board has hardware support for counting the number of pulses produced by the optical encoder. The angular position of the load shaft is then determined from this accumulated counter value. In this case, the sampling frequency determines how frequent the counter for the number of pulses is updated. There is also no need for a quantizer here because the value of the counter is an integer.

2.1 DLaB Library Functions

A library of C functions, DLaB, has been developed to simplify the simulation and implementation of digital controllers in the laboratory. The entire DLaB library contains several hundreds lines of C code. However, from a user's point of view, only several functions in the library are relevant.

The DLaB library functions can be used in either *Simulation* or *Hardware* mode. Furthermore, the syntax for using these functions in the simulation and hardware modes are *identical*. Hence, after the controller program has been developed and debugged in the simulation mode, it can be downloaded and run directly on the micro-controller in the laboratory for testing on the actual motor hardware without any further code modification.

A brief description of the relevant DLaB functions are given below. The exact syntax for calling these functions can be found in Appendix C.

1. `Initialize()`
   This function is used to set up the sampling frequency to be used by the controller. In the *Simulation* mode, it is also used to determine the appropriate mathematical model for simulating the behavior of the motor module. This function must be called before any of the remaining DLaB library functions can be used.

2. `Terminate()`
   This function shuts down communication between the workstation and the micro-controller. This function must be called when DLaB functions are no longer required.

3. `ReadEncoder()`
   This function returns the accumulated counter value of the number of pulses produced by the optical encoder.

4. `EtoR()`
   This function converts the counter value returned by the function `ReadEncoder()` into the angular position of the motor (in radian).

5. `DtoA()`
   This function is used to send a digital code to the D/A converter for conversion to an equivalent analog value.

6. `VtoD()`
   This function is used to convert a control voltage into the corresponding equivalent digital code for the D/A converter.

7. `plot()`
   This function is used to plot the result on the screen, or to save a hard copy of the graph in Postscript.

8. `Square()`
   This function is used to create a square wave with variable duty cycle.

2.2 Implementation of the Control Program

The project can be done entirely on the same VM used for Lab 1 to 5. It will be assumed that this is the case for the rest of this document. Students who wish to work on the project through remote connection to the EE network should consult Appendix A for additional instructions.

One way to implement the motor control program is to use a process with 2 Pthreads: `main()` and `Control()`. The `main()` thread is used to provide the user's interface, to set up and terminate communication with the micro-controller, and to create the `Control()` thread. The `Control()` thread is used solely to implement the digital control algorithm (which, in this case, is the PID algorithm).

An overview of the relationship between these two Pthreads, and their relationship to the hardware of the control system is given in Appendix B. As shown in Figure 2, the motor position is obtained through an optical encoder which produces a series of pulses as the motor's shaft rotates. A 32-bit counter is used in the micro-controller to keep track of the number of pulses. Through the `Initialize()` DLaB function, a timer is set up to produce a software interrupt every $T_s = 1/F_s$ seconds to trigger an interrupt service routine (ISR).

This ISR retrieves the counter value for the optical encoder and then post the semaphore "`data_avail`" to indicate that an optical encoder reading is available. Once the semaphore "`data_avail`" is acquired by the `Control()` thread, the counter value for the optical encoder can be obtained by calling the `ReadEncoder()` function. The counter value is then converted using the `EtoR()` function to obtain the motor's position in radians. Once the motor's position is determined, a control algorithm (such as Proportional or PID) can then be used to calculate the control value for correcting any tracking error. Finally, the control value is converted into a digital code for the DtoA converter using the `VtoD()` function, and sent using the `DtoA()` function to the D/A converter on the micro-controller board.

Proportional Control

As the first step of this project, a simple controller – the Proportional Controller, is implemented digitally. The control program developed is then extended to implement the PID controller.

Recall that the transfer function of a continuous-time proportional controller is

$$G_c(s) \;=\; \frac{U(s)}{E(s)} \;=\; K_p,$$

where $U(s) = \mathcal{L}\{u(t)\}$ is the controller output, $E(s) = \mathcal{L}\{e(t)\}$ is the tracking error, and K_p is the constant controller gain, Hence, the difference equation for the digital controller emulating the continuous-time proportional controller $G_c(s)$ is simply

$$u_k \;=\; K_p e_k,$$

where $u_k = u(kT)$ and $e_k = e(kT)$ (T is the sampling period).

Assume that the following menu selections are required:

r:	Run the control algorithm
p:	Change value of K_p
f:	Change value of sample frequency, F_s
t:	Change value of total run time, T_f
u:	Change the type of inputs (Step or Square)
	For Step input, prompt for the magnitude of the step
	For Square input, prompt for the magnitude, frequency and duty cycle
g:	Plot motor position on screen
h:	Save a hard copy of the plot in Postscript
q:	exit

A possible partial implementation of the `main()` thread is given in the following C/pseudo code:

```
#include "dlab_def.h"
...
pthread_t Control;
sem_t data_avail;      // Do not change the name of this semaphore
// Declare global variables (common data), for example:
#define MAXS 10000     // Maximum no of samples
                       // Increase value of MAXS for more samples
float theta[MAXS];     // Array for storing motor position
float ref[MAXS];       // Array for storing reference input
...
Kp = 1.0;         // Initialize Kp to 1.
```

```
run_time = 10.0;  // Set the initial run time to 10 seconds.
Fs = 200.0;       // Set the initial sampling frequency to 200 Hz.
Set motor_number. // Check your motor module for motor_number.
Initialize ref[k] // Initialize the reference input vector ref[k].
...
while (1) {
    Print out selection menu.
    Prompt user for selection.
    switch (selection) {
         case 'r':
             ...
             sem_init(&data_avail, 0, 0);
             Initialize(Fs, motor_number);
             pthread_create(&Control, ...);
             pthread_join(Control, ...);
             Terminate();
             sem_destroy(&data_avail);
             break;
         case 'u':
             prompt user for type of input: Step or Square
             if type == step {
                prompt for magnitude of the step
                set up the step reference input {ref[k]}
             }
             if type == square {
                prompt for magnitude, frequency and duty cycle
                set up {ref[k]} using DLaB function Square()
             }
             break
         case 'p':
             Prompt user for new value of Kp;
             break;
               .
               .
         case 'h':
             Save the plot results in Postscript;
             break;
         case 'q':
             We are done!
             exit(0);
         default:
             Invalid selection, print out error message;
             break;
    }
}
```

A partial C/pseudo code implementation of the `Control()` thread is given next.

```
void *Control(void *arg)
{
   ...
   k = 0;
   no_of_samples = (int)(run_time*Fs);
   while (k < no_of_samples) {
      sem_wait(&data_avail);
      motor_position = EtoR(ReadEncoder());
      calculate tracking error: ek = ref[k] - motor_position;
      calculate control value: uk = Kp*ek;
      DtoA(VtoD(uk));
      theta[k] = motor_position;
      k++;
   }
   pthread_exit(NULL);
}
```

Motor Number

The DLaB `Initialize()` function requires a motor number (`motor_number`) as one of its arguments. Usually, this number can be found on the physical motor module in the Control Systems Laboratory. As access to the Laboratory is not possible, use the formula below to obtain your assigned motor number:

$$\texttt{motor\_number} = d_9 + 1,$$

where d_9 is the last digit of your student number.

Task 2.1 *Proportional Control (15%)*
Using the partial C/pseudo code above as a guide, develop a Proportional Controller program (say, `pc.c`*) enter it with a text editor. Compile it with* `simcc` *and run the program with $F_s = 200$ Hz to obtain the closed-loop step response to a step input of $50°$ ($5\pi/18$ rad). Start up the VM and enter the following commands:*

```
cd
cd ele709/project
vi pc.c (replace "vi" with your favorite editor)
simcc pc.c
./pc
```

Save a hard copy of the step response for submission.

Task 2.2 *PID Controller Design using the Ultimate Sensitivity Method (10%)*
Run the Proportion Controller program again (with the same F_s and step input) to determine the ultimate gain K_u and period of oscillation P_u, then use them to calculate the PID controller parameters K_p, T_i and T_d. Note that, in order to obtain an accurate value of the ultimate gain, the run time for the controller should be set to at least 30 seconds to confirm that the step response does exhibit sustained oscillations. Once the value of the ultimate gain K_u is determined, re-run the control program for a shorter duration to determine the value of the period of oscillation P_u. Save a hard copy of the step response showing the period of oscillation for submission.

Basic PID Control

The Proportional Controller program developed in the previous section is now extended to implement the basic PID controller:

$$G_c(s) = K_p \left(1 + \frac{1}{T_i s} + \frac{T_d s}{1 + T_d s/N}\right). \tag{1}$$

In order to implement this controller its difference equation must first be obtained. For this project, the difference equation should be developed using the *forward rectangular rule* to perform numerical integration, and the *backward difference rule* to perform numerical differentiation. The PID control program should initialize, in the `main()` thread, N to 20, K_p, T_i and T_d to values determined in Task 2.2 earlier. It should also allow for the following *additional* menu selections:

i:	Change value of T_i
d:	Change value of T_d
n:	Change value of N

Task 2.3 *Basic PID Controller (30%)*
Derive the difference equation for implementing the basic PID controller in Eq. (1). Next, develop the basic PID control program as described earlier. Using the values of K_p, T_i and T_d obtained from the Ultimate Sensitivity method and a sampling frequency of $F_s = 200$ Hz, obtain the (untuned) closed-loop response to a square wave input of magnitude $\pm 50°$, frequency of 0.5 Hz and duty cycle 50%. Run the controller for a total run time of $T_f = 10$ secs. Save a hard copy of the response for submission. Is the performance of the Basic PID Controller satisfactory? If not, explain why.

Anti-Windup PID Control Scheme

An anti-windup PID control scheme to overcome the problem of integrator saturation is shown in Figure 3 with the Actuator Saturation Model shown in Figure 4.

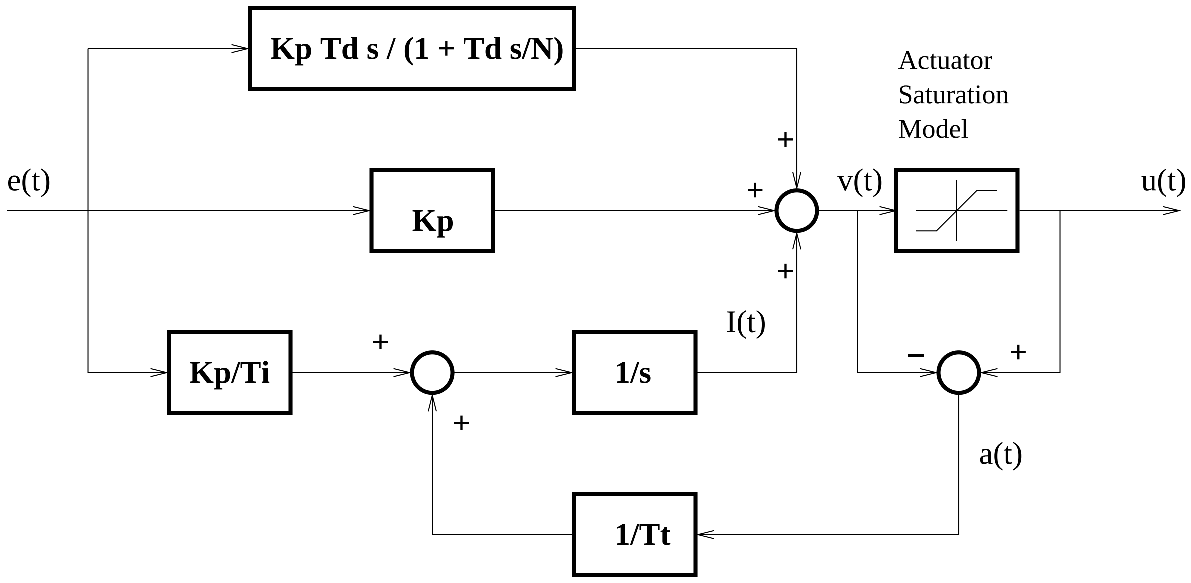


Figure 3: Anti-Windup PID control scheme

Task 2.4 *Anti-Windup PID Controller (25%)*
Write a C function `satblk()` *to implement the Actuation Saturation Model in Figure 4 as* `u = satblk(v)`*. Next, modify the difference equation for the basic PID controller in Eq. (1) to implement the anti-windup PID controller in Figure 3. Using the same controller parameters,* F_s *and square wave input as in Task 2.3 and a value of* $T_t = 0.01$*, obtain the (untuned) closed-loop response of the system. You should see a significant improvement in the response compare to that of the Basic PID Controller if the anti-windup PID controller works correctly. Save a hard copy of the response for submission.*

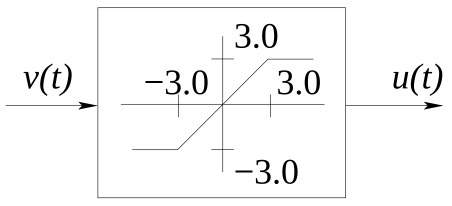


Figure 4: Actuator saturation model for the motor module

3 Part B: Further Extension

The current design of the control program only allows the controller to run for a fixed period of time, as specified by the value of `run_time`. As a result, the run-time of the controller is fixed. Also, the controller parameters cannot be changed throughout the entire run time of the controller.

Task 3.1 *Extension (20%)*
Extend the control program to include the following features:

1. *The controller runs continuously, unless the user sends a request to stop it.*
2. *The controller parameters (K_p, T_i, T_d, N) and the sampling frequency (F_s) can be changed interactively while the controller is running. The new parameters should take effect starting from the next sample.*
3. *The last 3 seconds of the system output is always stored so that it can be plotted if requested.*

4 Deadlines and What to Submit

The project is scheduled for a 3-week period from Week 9 to Week 11. The following table shows the deadlines for submitting the various parts of the project:

Week	Tasks	Submission
09	Proportional Controller in Simulation mode: Tasks 2.1 and 2.2	none
10	Basic and Anti-windup PID in Simulation mode: Tasks 2.3 and 2.4	Answer sheet 1, code and graphs for Week 9
11	Further Extension in Simulation mode: Task 3.1	Answer sheet 2, code and graphs for Week 10
12		Code for extension

A report is not required. Archive your submission and email to your TA as follows:

In Week 10:
```
tar cvf lastname-project-wk9.tar task2.1.c graph-task2.1.pdf graph-task2.2.pdf proj_ans1.pdf
```

In Week 11:
```
tar cvf lastname-project-wk10.tar task2.3.c task2.4.c graph-task2.3.pdf graph-task2.4.pdf
                     proj_ans2.pdf
```

In Week 12:
```
tar cvf lastname-project-wk11.tar task3.1.c
```

5 Reference

1. “Advanced Linux Programming,” M. Mitchell *et al.*, New Riders Publishing, 2001.

2. “Digital Control Engineering: Analysis and Design,” M. Sami Fadali and A. Visioli, Academic Press, 2009.

Appendix

A Project through Remote Connection to EE Network

Students who wish to work on the project through remote connection to the EE network (instead of on the VM) can follow the steps below:

1. Follow the instructions on D2L to install an Xserver (VcXserv for Windows 10 or XQuartz for MacOS) on your computer and use it to connect to one of the lab computers. There is no need to install an Xserver for native Linux users, just use "ssh -Y" to connect to the EE network and then to the lab computers.

2. Once connected to a lab computer, enter the following command to create a folder for the project:

   ```
   cd
   mkdir ele709project
   ```

3. Enter the following commands to copy the required support files for the project from the network drive:

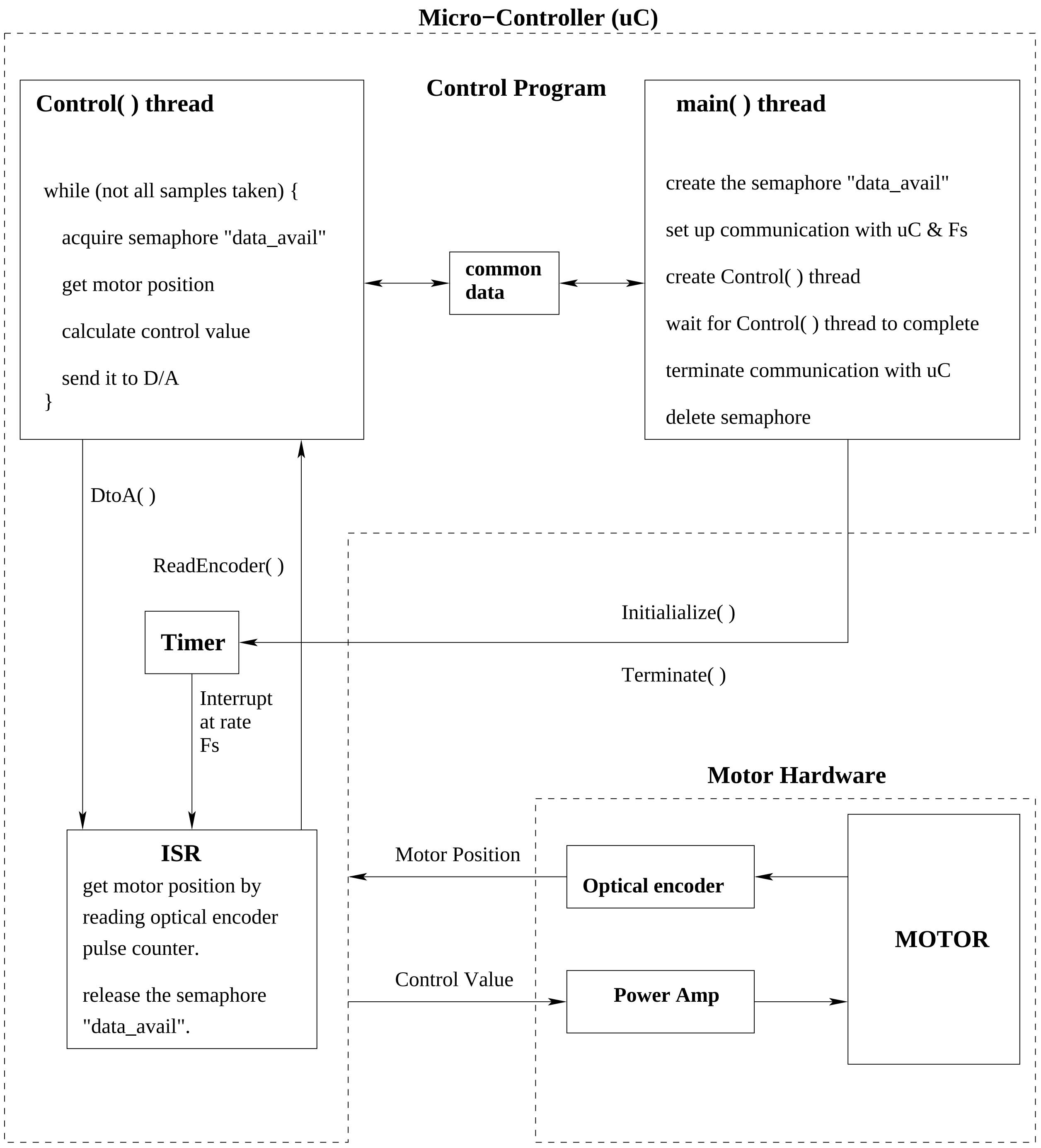
   ```
   cd ele709project
   cp /home/courses/ele709/project/*  .
   ```

   Don't forget to enter the "." at the end of the last command.

4. All the C code developed for the project should be stored in the "`ele709project`" folder created above and can be compiled and executed as follows:

   ```
   ./simcc project.c
   ./project
   ```

B Overview of the Control Program



C Descriptions of DLaB Functions

This appendix describes the syntax for calling the DLaB Library Functions. The DLaB Library Functions can be used either in the Simulation or Hardware mode. Programs should always be debugged and tested first in the Simulation Mode.

Documentations on POSIX functions can be found through the course Web Page.

The `dlab_def.h` Header Files

Synopsis

```
#include "dlab_def.h"
```

Description

The header file **`dlab_def.h`** includes the necessary system header files and provides correct prototyping for the DLaB Hardware Functions. It must be included in any control program that requires access to the motor hardware.

The **`dlab_def.h`** header file and other project support files should be copied from the folder

```
/home/courses/ele709/project
```

on the EE Network drive and placed in the same folder as the rest of your project files.

The `Initialize()` Function

Synopsis

```
#include "dlab_def.h"
int Initialize(float Fs, int motor_number);
```

Description

The `Initialize()` function is used to set the sampling frequency, `Fs`, to be used by the controller.

When working in the hardware mode, the `Initialize()` function sets up communication between the Workstation and the micro-controller.

When working in the simulation mode, the `Initialize()` function uses the value of `motor_number` to determine the appropriate mathematical model to be used for simulating the behavior of the motor module. The value of `motor_number` for your workstation can be found on the top cover of the motor module.

The `Initialize()` must be called before any of the remaining DLaB functions can be used. A positive return value from `Initialize()` indicates the call is successful.

Example

The following program segment shows how `Initialize()` is used to set up a sampling frequency of $F_s = 200$ Hz and to use the model for motor number 2 (in simulation mode).

```
#include "dlab_def.h"
...
float Fs;
int motor_number;
...
Fs = 200.0;         // sampling frequency = 200 Hz
motor_number = 2;  // use motor number 2
if (Initialize(Fs, motor_number) < 0) {
    printf("Error in Initialize()\n");
    exit(99);
}
```

The `ReadEncoder()` and `EtoR()` Functions

Synopsis

```
#include "dlab_def.h"
int ReadEncoder(void);
float EtoR(short int counter_value);
```

Description

The `ReadEncoder()` function returns the current value of the 16-bit counter used to accumulate the number of pulses produced by the optical encoder. It is used in conjunction with the `EtoR()` function to determine the motor's angular position in radians.

Example

The following program segment shows how these two functions are used:

```
...
#include "dlab_def.h"
...
float motor_position;
...
motor_position = EtoR(ReadEncoder());  // Determine the motor position
                                       //  in radians.
...
```

The `DtoA()` and `VtoD()` Functions

Synopsis

```
#include "dlab_def.h"
int DtoA(short int digital_code);
short int VtoD(float control_voltage);
```

Description

The `DtoA()` function sends the digital value `digital_code` to the D/A converter on the micro-controller board. The output of the D/A converter is an analog voltage between ±3.0 volts, and is converted by treating `digital_code` to be a 16-bit 2's-complement number. The function `VtoD()` is used to convert the control voltage `control_voltage` into `digital_code`.

A value of 0 is returned by `DtoA()` to indicate the call is successful.

Example

The following program segment shows how these two functions are used:

```
...
#include "dlab_def.h"
...
float control_voltage;
...
control_voltage = ...        // The control_voltage is calculated
                             //   using a control algorithm (e.g. PID).
DtoA(VtoD(control_voltage)); // It is converted by VtoD and sent to the D/A.
...
```

The `Terminate()` Function

Synopsis

```
#include "dlab_def.h"
void Terminate();
```

Description

The `Terminate()` function resets the micro-controller and shuts down communication between the Workstation and micro-controller. This function must be called when access to the motor hardware is no longer required.

Example

The following program segment shows how this function is used:

```
...
#include "dlab_def.h"
...
Terminate();
...
```

The `plot()` Function

Synopsis

```
#include "dlab_def.h"
void plot(float *v1, float *v2, float Fs, int no_of_points,
          int term, char *title, char *xlabel, char *ylabel)
```

Description

This function is used to plot the graphs of the data contained in the arrays `v1` and `v2` against time. The sampling frequency used to obtain the data points is specified by `Fs`. The number of elements in the arrays, `no_of_points`, must be the same. The variable `term` is used to specify the output devices for the plot. Setting `term` to `SCREEN` will plot the graphs on the computer screen, and setting `term` to `PS` will save a hard copy of the graphs in Postscript format. The user will be prompted for a file name for saving the graphs if the `PS` option is chosen. The title of the graph can be specified using the character pointer `title`. The labels for the $x-$ and $y-$axis can be specified using the character pointers `xlabel` and `ylabel` respectively.

Note that the graph saved in Postscript format can be converted into PDF format as follows:

```
ps2pdf graph.ps graph.pdf
```

where `graph.ps` is the (input) Postscript file and `graph.pdf` is the (output) PDF file.

Example

The following program segment shows how this function is used:

```
...
#include "dlab_def.h"
...
float ref[100], theta[100];
float Fs;
int no_of_points;
...
no_of_points = 50;
// Plot the graph of ref and theta vs time on the screen
plot(ref, theta, Fs, no_of_points, SCREEN, "Graph Title", "x-axis", "y-axis");
...
// Save the graph of ref and theta vs time in Postscript
plot(ref, theta, Fs, no_of_points, PS, "Graph Title", "x-axis", "y-axis");
...
```

The `Square()` Function

Synopsis

```
#include "dlab_def.h"
void Square(float *y, int maxsamples, float Fs,
            float mag, float freq, float dc);
```

Description

This function is used to create a square wave `y` with magnitude ±`mag` (in radian) and frequency `freq` (in Hertz) with a duty cycle of `dc` (in percent). The sampling frequency used to obtain the data points is specified by `Fs`. The number of data points to be created is specified by `maxsamples`.

Example

The following program segment shows how this function can be used to create a square wave (with magnitude= $\pm 50°$, frequency= 0.5 Hz, and a duty cycle of 50%) and store it in the array `y`. The sampling frequency is 100 Hz and the number of data points to be produced is 200.

```
...
#include "dlab_def.h"
...
float y[200]
...
Square(y, 200, 100.0, 50.0*pi/180.0, 0.5, 50);
...
```