Toronto Metropolitan University
Department of Electrical, Computer & Biomedical Engineering
ELE709 — Real-Time Computer Control Systems

W2025 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 extend the control program developed in Part A for implementation and testing on the actual motor hardware.

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

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

Note: Programs developed for this part of the project can be compiled and run on *any* `Linux` workstations within the ECBE Department's network.

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 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 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.

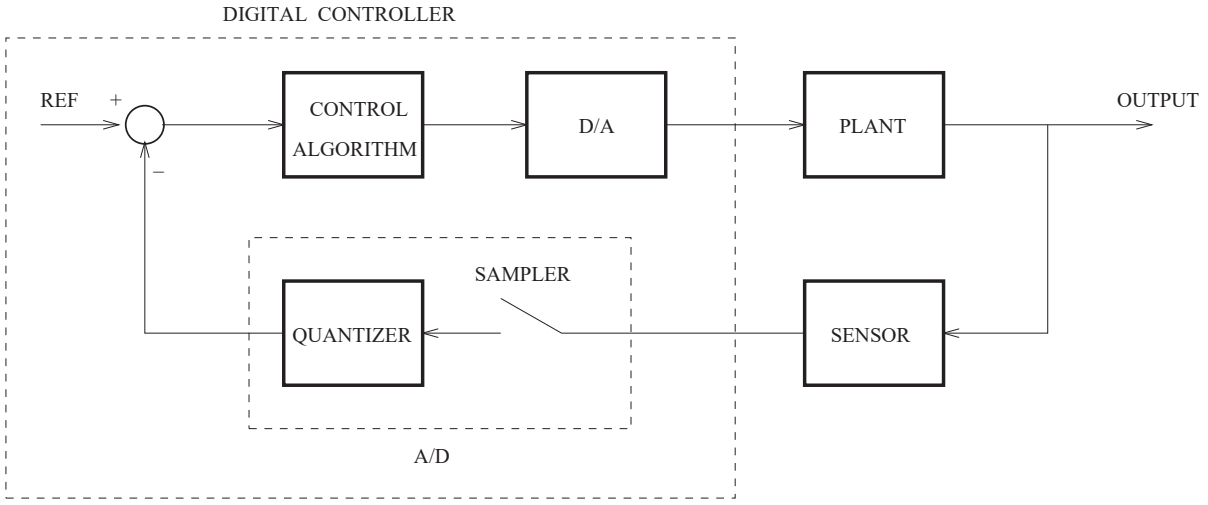

Figure 1: A Digital Control System

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 the DLaB 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 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 A.

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.

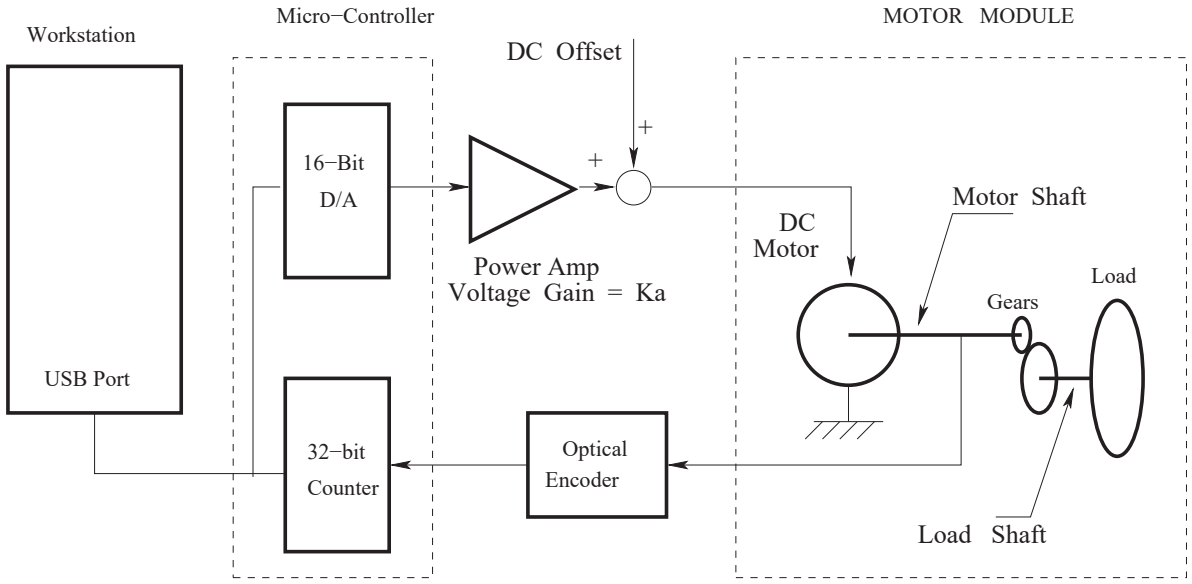

Figure 2: Block Diagram of Laboratory Module

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

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 Figure 9. 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);
}
```

After the proportional controller program is developed, follow the steps below to obtain the necessary support files for the project:

1. Login to the ECBE Department's network.
2. Enter the following command to create a folder for the project:

```
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. Move/upload the proportional controller program into the "ele709project" folder.

Task 2.1 *Proportional Control*
Obtain the project support files as described above. Compile the Proportional Controller program (say, `pc.c`*) 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). For example,*

```
cd ele709project
./simcc pc.c
./pc
```

Note that the development and testing of the controller program in Simulation Mode can be carried out on any `Linux` *workstation within the ECBE Department's network.*

Task 2.2 *PID Controller Design using the Ultimate Sensitivity Method*
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*.*

Basic PID Control

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

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

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*
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 and show it to the TA. 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.

Task 2.4 *Anti-Windup PID Controller*
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 and show it to the TA.*

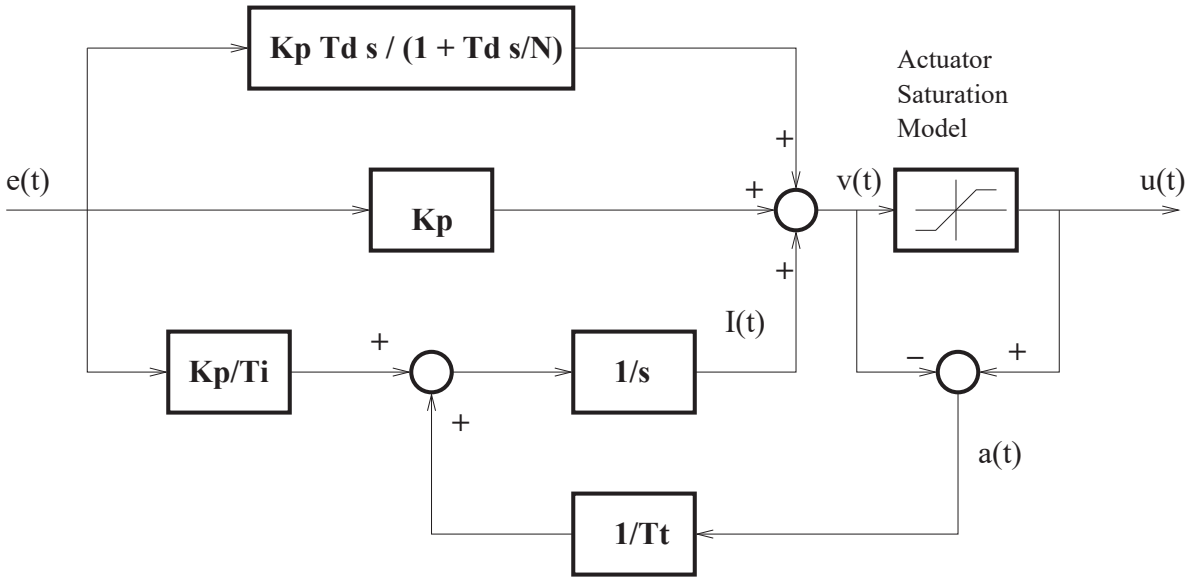

Figure 3: Anti-Windup PID control scheme

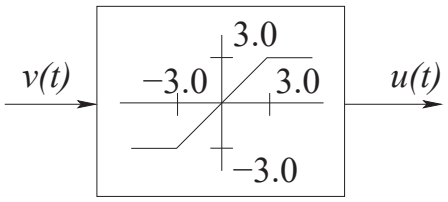

Figure 4: Actuator saturation model for the motor module

3 Part B: Testing on the Motor Hardware

Note: Programs for this part of the project should *only* be compiled using `hwcc` and run on the micro-controller of the motor module in the Control Systems Laboratory (ENG413).

The objective of this part of the project is to test the PID controllers (Basic and Anti-windup) developed in Part A on the micro-controller board to control the actual hardware of the motor module. The simulation program developed in Part A can be used, *without any changes*, on the micro-controller.

Recommended Work Flow

It is important that you test and debug all the control programs developed for Part A under the Simulation Mode ahead of the laboratory session. Once you are in the laboratory, transfer the `ele709project` folder to the micro-controller as follows:

1. Go to a workstation with an attached motor module.
2. Turn on the power of the motor module. The 2 LED lights on the front panel will turn on and the motor position pointer will begin to spin rapidly.
3. Wait for the LED light near the bottom right corner to turn off. The motor position pointer should also stop spinning, or it may rotate at a much slower rate. This should take no longer than a couple

of minutes.

4. The micro-controller is now up and running.
5. In order to access the micro-controller, login to the workstation using your EE network account first.
6. Move the mouse pointer to the menu bar at top of the Desktop and click on "`Places`" and then "`Home Folder`". A new window (hereafter, referred to as the "`Network Home Window`") would pop up displaying all your files and folder in your home folder on the EE network drive.
7. Go to the "`Network Home Window`" and locate the "`ele709project`" folder within your "`Home Folder`" (but do *not* open the folder).
8. Go back to the menu at the top of the Desktop and click on "`Places`" and then "`Connect to Server`". A window similar to the one in Figure 5 would appear. Go to this window and do the following:

Figure 5: Connect to Server Window

user name : user
passwd : letusein

(a) Select "`SSH`" for "`Service type`".
(b) Enter "`usbhost`" for "`Server`".
(c) Enter "`/home/user`" for "`Folder`".
(d) Enter "`user`" for "`User Name`"
(e) This step is optional: Check the "`Add bookmark`" box and enter "`ENG413-Servo`" for "`Bookmark name`".
(f) Click on "`Connect`".
(g) A new window will appear and you will be prompted for the password. Enter the login password provided by the laboratory instructor. If this is the first time you connect to the `usbhost` server, then a message similar to the one shown in Figure 6 may appear. Simply click on "`Log In Anyway`" and continue. A new window showing the home folder of the micro-controller, `/home/user`, would then appear (see Figure 7). Hereafter, this window will be referred to as the "`Micro-Controller Home Window`". Any new files created on the micro-controller in `/home/user` will automatically appear in this window. Keep this window open through out the entire laboratory session.

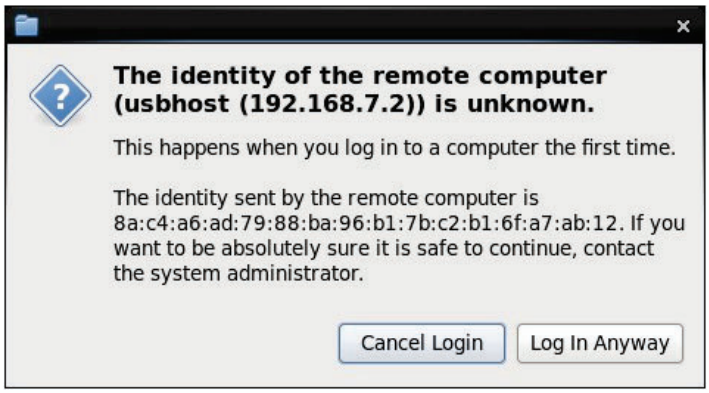

Figure 6: Confirmation Message

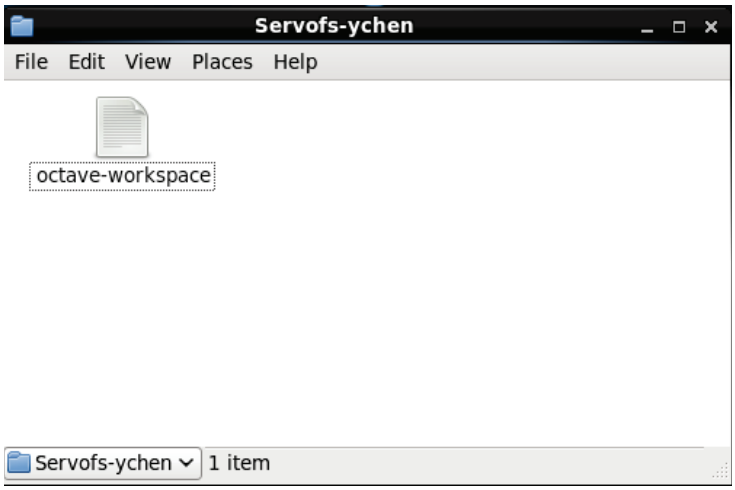

Figure 7: The Micro-Controller Home Window.

9. You can now transfer the `ele709project` folder in the ECBE network drive to the micro-controller by simply dragging it from your "`Network Home Window`" and dropping it onto the "`Micro-Controller Home Window`".

   DO NOT CLOSE the "`Network Home Window`" and "`Micro-Controller Home Window`" before the end of the laboratory session.

10. Files from the micro-controller can be transferred back to the ECBE network home folder in a similar manner as in the previous step, but in the opposite direction.

Once the controller programs have been successfully transferred to the micro-controller, they can then be tested on the motor hardware as follows:

1. Login to the micro-controller:

   (a) Move the mouse pointer to the menu bar at top of the Desktop and click on "`Applications`" followed by "`ENG413 Controls`" and then "`Connect to Servo`". See Figure 8.

   (b) A new terminal window (hereafter referred to as the "`Microcontroller Terminal Window`") will pop up.

(c) You are now logged onto the micro-controller.

2. Go to the "`Microcontroller Terminal Window`" and enter the following `Linux` commands:

```
cd ele709project
ls -l
```

You should see the files transferred earlier from your ECBE network account listed.

3. To test your control program, say `pid.c`, it must first be re-compiled using the command:

```
./hwcc pid.c       //(or if this doesn't work, use any other hwcc compiler files in the folder)
```

This will produce an executable file: `pid`, which can then be run by entering:

```
./pid
```

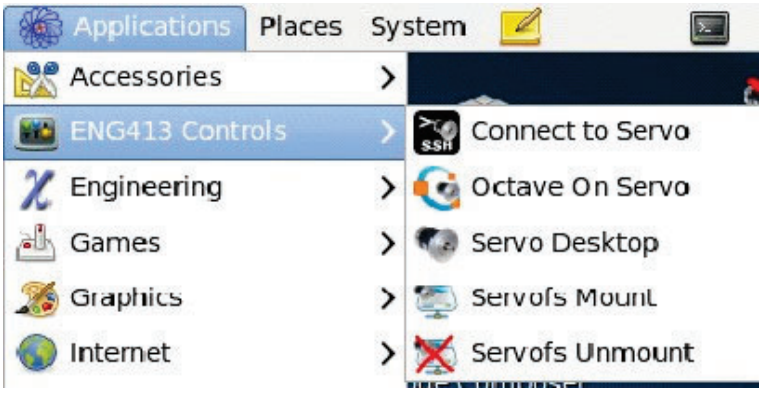

Figure 8: ENG413 Controls Applications Menu

Task 3.1 *Controller Testing on Motor Hardware*
Use a sampling frequency of $F_s = 500$ *Hz, run the antiwindup PID controller for 10 seconds with a square wave reference input of magnitude* 120° $(2\pi/3$ *rad), frequency* 0.5 *Hz and 50% duty cycle. Observe and save a hard copy the response of the system. Show your result to the TA.*

Task 3.2 *Effect of Loading*
Open up another terminal window to the micro-controller (as per earlier instructions) and enter the commands:

```
cd ele709project
./hwcc newload.c
./newload
```

While the `newload` *program is running, return to the first terminal window opened earlier and run the control program again (with the same controller parameters, sampling frequency and square wave input). Plot and save a hard copy of the response of the system. Show your result to the TA. Is the response the same as before? Explain any discrepancy. Terminate the* `newload` *program by switching back to the second terminal window and press "Ctrl-C" (i.e. press the C key while holding the Ctrl key).*

IMPORTANT: Transfer all the files that you want to save from the micro-controller back to your EE network home folder (by dragging and dropping) before turning off the motor module. All the files on the micro-controller will be lost as soon as the power is turned off.

4 Part C: 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.

Extend the control program (in Simulation Mode only) 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.

5 Deadlines and what to submit

The project is scheduled for a 4-week period. However, the various parts of the project must be demonstrated and results submitted according to the following.

Week	Demonstration	Submission
9	Task 2.1 (10%)	No submission
10	Task 2.2 Task 2.3 (20%)	Answer Questions 1, 2, and 3 of `Project_Qs.pdf` and code files (15%)
11	Task 2.4 Task 3.1 (20%)	Answer Questions 4 and 5(a) of `Project_Qs.pdf` and code files (15%)
12	Task 3.2 (10%)	Answer Questions 5(b) and 6 of `Project_Qs.pdf` and code files (10%) + Part C: Further Extension (Bonus 10%)

Note: Submissions, including your code files, are to be submitted to D2L. Include the names and student numbers of all group members as comments at the top of the code files.
A reduction of 20% per week will be applied to late demonstration and/or submission.

6 References

1. "Advanced Linux Programming," M. Mitchell et al., New Riders Publishing, 2001.
2. "ELE709 - W2019 Project," Y.C. Chen, Ryerson University, 2019.
3. "Digital Control Engineering: Analysis and Design," M. Sami Fadali and A. Visioli, Academic Press, 2009.

Appendix

A 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/proje** 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 by `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 on any of the workstations 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= ±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);
...
```

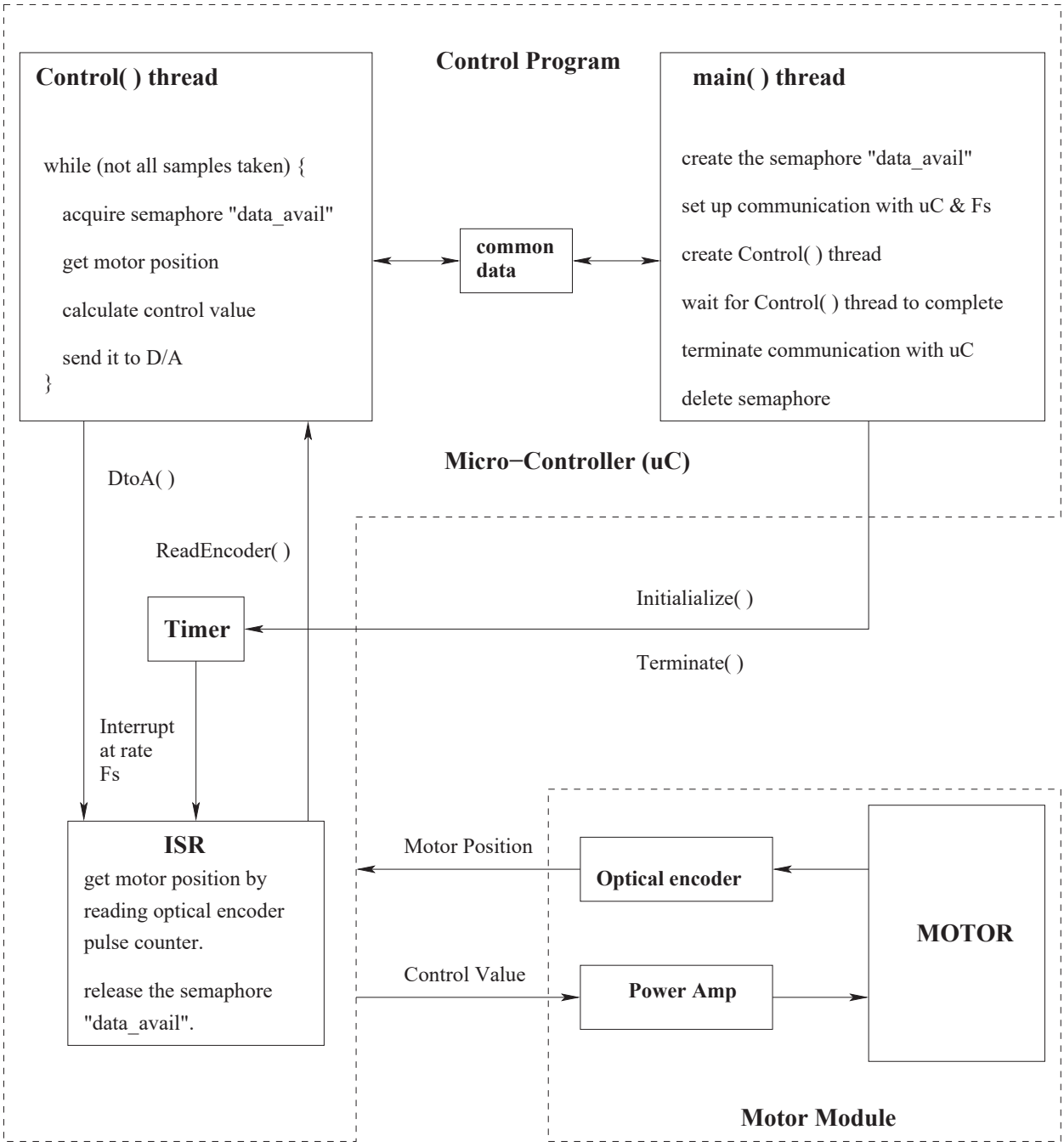

Figure 9: Overview of the Control System