Ryerson University
Department of Electrical & Computer Engineering
ELE709 — Real-Time Computer Control Systems

W2017 Project — Real-Time Digital PID Controller

1 Objectives

The objective of this project is to design and implement a real-time 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 real-time implementation on a real-time Linux operating system.

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 (50%)

Important: Programs developed for this part of the project can be compiled using `simcc` and run on *any* workstations within the EE 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 both hardware and software on the DSP board. In particular, data reconstruction is performed using a 16-bit D/A converter available on the DSP board. Even though a 16-bit A/D converter is also available, it is not used because it is possible to achieve the same objective (i.e. determining

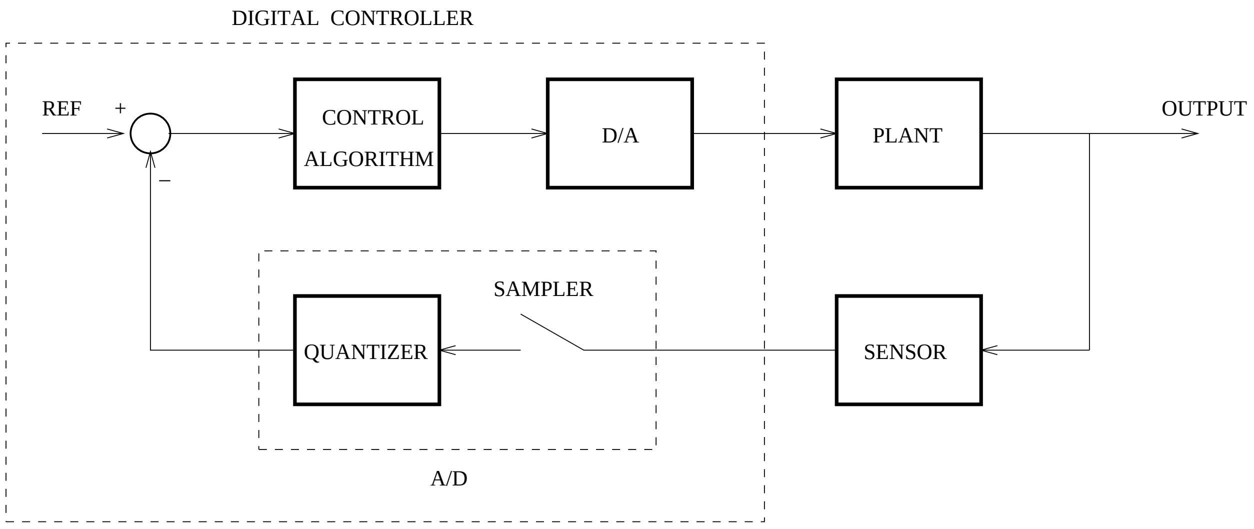

Figure 1: A Digital Control System

the angular position of the motor's load shaft) with an optical encoder with digital outputs. Specifically, a simple program has been written for the DSP board to accumulate the number of pulses produced by the optical encoder. The angular position of the load shaft is then determined from this accumulated value. In this case, the sampling frequency determines how frequent the counter for the number of pulses is updated. There is no need for a quantizer here because the value of the counter is an integer. More information on the D/A converter and optical encoder can be found in Appendix A.

2.1 DLaB Library Functions

A library of C functions, DLaB, has been developed to simplify the implementation of digital controllers on the workstations in the laboratory. The entire DLaB library contains several hundreds lines of C and assembly language 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. The syntax for using the DLaB functions in the simulation and hardware modes are *identical*. A flag is passed to the `Initialize()` function to indicate which mode is to be used.

A brief description of the relevant DLaB functions are given below. Details on these functions can be found in Appendix B.

1. `Initialize()`
   This function is used to set up the mode of operation (simulation or hardware) of the DLaB library. In the *Hardware* mode, it is used to setup communication between the workstation and DSP board, and the sampling frequency `Fs`. In the *Simulation* mode, it is 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.

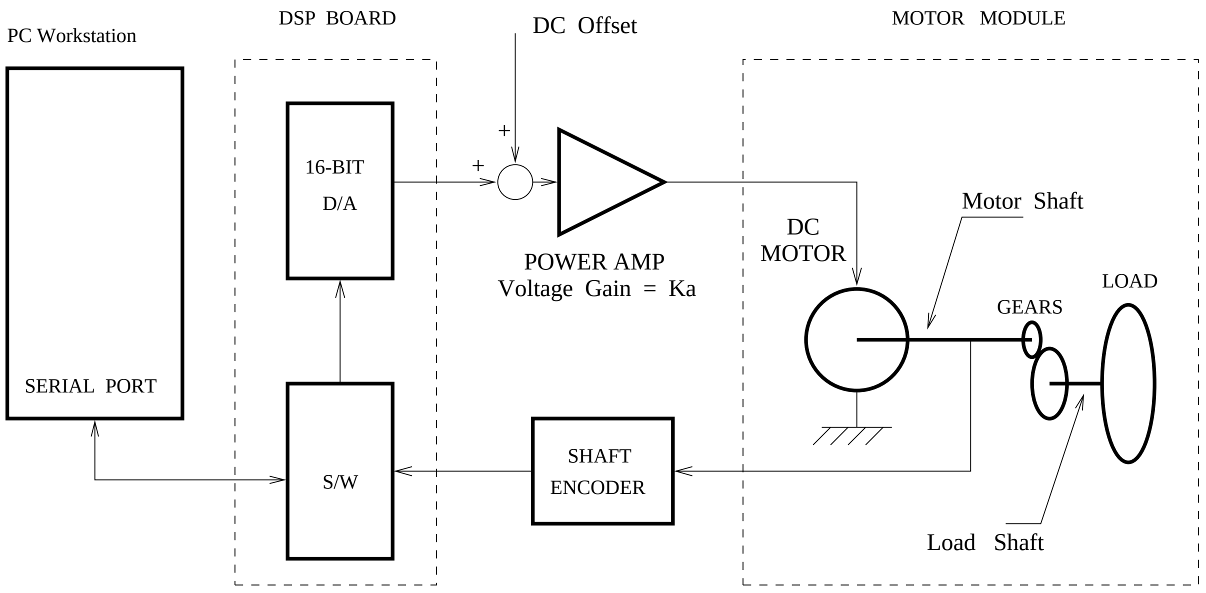

Figure 2: Block Diagram of Laboratory Module

2. `Terminate()`

   This function resets the hardware of the DSP board and shuts down communication between the workstation and DSP board. This function must be called when DLaB functions are no longer required.

3. `ReadEncoder()`

   This function returns the accumulated value of the number of pulses produced by the optical encoder.

4. `EtoR()`

   This function converts the 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 the graph of the result 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 hardware, 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 4. As explained in Appendix A, the motor position is obtained through an optical encoder which produces a pulse for every full revolution of the motor's shaft. A 16-bit counter is used in the DSP board to keep track of the number of pulses. Through the `Initialize()` DLaB function, an interrupt is produced every $T_s = 1/F_s$ seconds and an interrupt service routine (ISR) is called. This ISR is a function that has already been written to run in the kernel of the RTOS. All it does is to receive 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 DSP 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_freq`
t:	Change value of `run_time`
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 results on screen
h:	Save the Plot results in Postscript
q:	exit

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

```
#include "dlab.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 5000      // 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 = 3.0;   // Set the initial run time to 3 seconds.
Fs = 200.0;       // Set the initial sampling frequency to 200 Hz.
Set motor_number. // See lab_layout.pdf for your 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(DLAB_SIMULATE, 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);
}
```

Task 2.1 *Develop the Proportional Controller program using the DLaB Simulation Functions. Download and install the script file* `simcc` *available in the project directory. Compile the Proportional Controller program with* `simcc` *and run the program to obtain the closed-loop step response to a step input of* $55°$*. Save the graph of the response for your report. Note that the development and testing of the controller program in Simulation mode can be carried out on any workstations within the EE Department's network.*

Task 2.2 *Next, a PID controller is designed using the Ultimate Sensitvity Method. Run the Proportion Controller program again to determine the utlimate gain and period of oscillation, then 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 20 seconds to ensure that the step response exhibits sustained oscillations.*

Basic PID Control

The Proportional Controller 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 *Develop 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, obtain the (untuned) closed-loop response to a step input of 55° and save the graph for your report.*

Anti-Windup PID Control Scheme

An anti-windup PID control scheme for avoiding the problem of integrator saturation is shown in Figure 3.

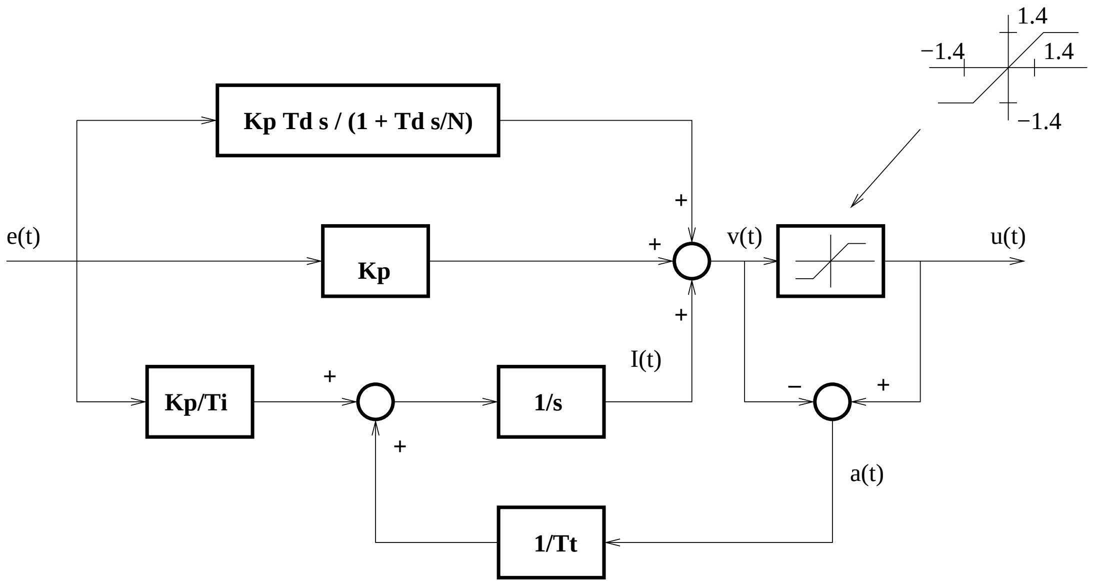

Figure 3: Anti-Windup PID control scheme

Task 2.4 *Write a C function* `satblk()` *to implement the saturation block in Figure 3 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 parameters as the basic PID controller and a value of $T_t = 0.01$, obtain the (untuned) closed-loop response to a step input of 55° and save the graph for your report.*

3 Part B: Real-Time Implementation (10%)

Important: Programs developed for this part of the project can *only* be compiled using `hwcc` and run on workstations in the Control Systems Laboratory (ENG413).

The objectives of this part of the project is to modify the PID controllers (Basic and Anti-windup) developed in Part A to run in real time under the RTOS RTAI (Real-Time Applications Interface) to control the actual hardware of the motor module.

The Real-Time Application Interface (RTAI) is a hard real-time extension of the Linux kernel. It provides the functionality of an industrial-grade real-time operating system that can be seamlessly accessible from the Linux programming environment. Furthermore, RTAI also provides a module called LX/RT that allows users to develop and run RT applications in the user space (as opposed to the kernel space). This is an important advantage, especially during the development stage, because even if the RT application being developed crashed in the user space it will not bring down the entire system.

The DLaB simulation functions were used in Part A of the project to develop and debug control programs for the digital implementations of the PID controllers. These simulation programs can easily be modified using RTAI function calls and the DLaB hardware functions to control the actual motor hardware. The modifications consist of the following steps:

1. Change the parameters `DLAB_SIMULATE` to `DLAB_HARDWARE` in the `Initialize()` function call.
2. Replace the soft real-time POSIX semaphore and its associated function calls with a hard real-time RTAI FIFO semaphore and associated function calls to provide synchronization between the ISR and the control thread.
3. Insert the required RTAI function calls to convert the control thread into a real-time task, and the required Linux function calls to lock the control thread in real memory to prevent it from being paged out to virtual memory on disk. Next, insert the required RTAI function to start the RT timer at the beginning of the control program, and to stop it at the end.

Details of the required modifications are presented next.

3.1 Modifications for Real-Time Implementation

Step 1

In order to access the actual motor hardware, the `DLAB_SIMULATE` argument to the `Initialize()` function must first be changed to `DLAB_HARDWARE`. The other arguments to the `Initialize()` should remain unchanged.

Step 2 (Soft Real-Time Modifications)

As explained in the Project Instructions for Part A, the position of the motor's shaft is obtained by using an optical encoder which produces pulses to increment or decrement a 16-bit counter on the DSP board. Once a new value is received, a hardware interrupt is generated by the DSP board. This, in turn, causes an interrupt service routine (ISR) to be executed in the *kernel* of the RTAI RTOS that runs the PC Workstation. Once the ISR obtained the motor position, it posts a semaphore to signal the control thread (which is blocked while waiting on this semaphore) that new data is available. As soon as the ISR posted the semaphore, the control thread is unblocked so that the control value can be calculated and sent to the D/A converter. This is what will happen if the actual motor hardware is used.

For the control simulation program in Part A of the project, all of the above operations are *simulated*. In particular, there is no need for an ISR as no hardware interrupts are actually produced by the DLaB simulation functions. Hence, it suffices to use a POSIX semaphore and the associated POSIX function calls to synchronize the control thread and the "simulated ISR".

The synchronization method between the `Control` thread and the "simulated ISR" via a POSIX semaphore must now be replaced by its hard real-time counterpart; and so must the associated POSIX function calls. Since the ISR runs in the kernel space and the control thread runs in the user space, a special synchronization method must be used. This can be done by using an RTAI kernel object called *RTAI FIFO* (similar to a message queue) as shown below:

(a) Replace the declaration of the soft real-time semaphore

```
sem_t data_avail;
```

by

```
int data_avail;
```

The integer variable `data_avail` is used to store the file descriptor of the RT FIFO that will be used by the control thread and the ISR for synchronization.

(b) Replace the statement

```
sem_wait(&data_avail);
```

in the control thread function by

```
rtf_sem_wait(data_avail);
```

(c) Replace the statement

```
sem_init(&data_avail, 0, 0);
```

in the `main()` thread by the following statements:

```
            if ( (data_avail = open("/dev/rtf0", O_WRONLY)) < 0 ) {
                printf("Error: cannot open /dev/rtf0.\n");
                exit(1);
            }

            if ( rtf_sem_init(data_avail, 0) != 0) {
                printf("Error: cannot init RT FIFO semaphore.\n");
            }
```

and the statement

```
            sem_destroy(&data_avail);
```

by the following statements:

```
            rtf_sem_destroy(data_avail);
            close(data_avail);
```

Task 3.1 *Make the modifications described in Steps 1 and 2 to allow the control program to run in soft real-time and to access the actual hardware of the motor module. Download and install the script file* `hwcc` *from the project directory and compile the modified program with it. Turn on the power of the motor module, make sure the "DSP" switch is in the "upward" position and press the "Reset" button on the motor module. This should be done* **everytime** *before running any control program in the Hardware mode. Run the modified control program for 15 seconds with a square wave reference input of magnitude* $100°$*, frequency* 0.5 *Hz and 50% duty cycle. Observe and save the response of the system. Next, download the file* `newload.c` *from the project directory and compile it with* `hwcc`*. Open two terminal windows, say A and B. Re-run the control program again (for 15 seconds, with the same square wave reference input) in terminal window A. As soon as the control program starts the motor calibration stage, switch to terminal window B and run* `newload` *from it.* **(Caution: Do not move the computer mouse away from terminal window B.)** *Wait until the control program finishes, i.e. until the control program prompts for user's input again. Now, terminate the* `newload` *program by entering Ctrl-C in terminal window B (you may need to do it several times). Then, switch back to terminal window A. Plot and save the response of the system. Is the response the same as before? Explain any discrepancy.*

Step 3 (Hard Real-Time Modifications)

This step is used to convert the control thread from soft real-time into RTAI hard real-time. This is done by adding the following program code at the beginning of the thread function for the control thread (comments are included for explanations):

```
// Declare an RTAI RT task
unsigned long control_task_name;
RT_TASK *control_task;

// Allow non-root user to use POSIX soft real-time process management
//  and memory lock functions.
rt_allow_nonroot_hrt();

// Register this thread with RTAI
control_task_name = nam2num("CONTROL");
control_task = rt_task_init(control_task_name, 0, 0, 0);

// Make this a hard real-time task
rt_make_hard_real_time();

// Prevent paging by locking the task in memory
mlockall(MCL_CURRENT | MCL_FUTURE);
```

At the end of the control thread function, just before calling `pthread_exit()`, add the following code:

```
// reset to soft real-time
rt_make_soft_real_time();

// un-register the control thread from RTAI
rt_task_delete(control_task);

// re-enable paging
munlockall();
```

Now, start the RT timer by adding the following program code in the `main()` thread, just before the creation of the `control()` thread:

```
int period_in_ticks;

period_in_ticks = (int)( ((double) 1.0e9)/((double) Fs));
start_rt_timer(period_in_ticks);
```

In the above, `Fs` is the value of the sampling frequency used by the digital control algorithm. Next, add the statement "`stop_rt_timer();`" to stop the RT timer after the control thread is done (i.e. after the "`close(data_avail);`" statement in Step 2).

Task 3.2 *Repeat Task 3.1 described in Step 2, but with the hard real-time modifications included in the control program. Compare the response obtained with and without running the* `newload` *program. Are there any discrepancies? Explain.*

4 Part C: Further Extension (20%)

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 Report (20%)

A formal report is not required, but you are required to submit an informal report containing at least the following:

- Derivation of the difference equations for implementing the Basic PID and Anti-windup PID algorithms.
- The responses (simulation, soft and hard real-time) of the motor module obtained using the Basic PID and Anti-windup PID algorithms as described in Parts A and B earlier. Include the values of the controller parameters and any explanations **on the graphs of the step responses**.
- Commented program listing for the *control thread function* used in the real-time Anti-windup PID control program. Only the program listing for the control thread function is required.

- Commented program listing for the extended control program as described in the "Further Extension" section. Program listing for the entire extended control program is required. Include any comments, explanations or diagrams which may help with the understanding of the program.

The report will be graded not only on its technical merit, but also on the communication and organizational skills of the author(s) as exhibited through the report.

6 Deadlines

The project is scheduled for a 3-week period. However, the various parts of the project must be demonstrated and the associated C code submitted according to the following deadlines:

Proportional Controller in Simulation mode: Tasks 2.1 and 2.2	First week
Basic and Anti-windup PID in Simulation mode: Tasks 2.3 and 2.4	Second week
Anti-windup PID in Hardware mode: Tasks 3.1 and 3.2	Third week
Further Extension in Simulation mode	Third week
Report	TBA

The program code (the hard RT and extended control programs only) should be zipped and renamed using your lastname and email to `ychen@ee.ryerson.ca`. **A reduction of 20% per week will be applied to late demonstration and submission.**

7 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.
3. Detailed documentation for RTAI functions are available through the RTAI web page: `http://www.rtai.org/documentation/magma/html/api`

Appendix

A The Laboratory Module

This appendix contains important information on the various components of the laboratory module. Not all of these information are required for the project, but they are included for the sake of completeness.
The laboratory module consists of:

1. An armature controlled DC gear motor,
2. An optical encoder for reading the position of the motor shaft,
3. A Motorola DSP56002EVM digital signal processing board with 16-bit D/A and A/D converters, and
4. A power amplifier.

A block diagram of the laboratory module has been shown in Figure 2. The components of the laboratory module are described in the following sections.

The DC Motor

The motor used in the laboratory is an armature controlled DC gear motor. The physical parameters for the motor are given in Table 1. These parameters are required when determining a model

Table 1: Physical Parameters for Motor

Armature resistance, R_a (Ω)	3
Armature inductance, L_a (H)	0.15×10^{-3}
Back emf constant, K_b (N m/A/rad)	6.09×10^{-3}
Motor torque constant, K (N m s/rad^2)	6.09×10^{-3}
Motor moment of inertia, J_m (N m s^2/rad^2)	0.25×10^{-6}
Motor viscous friction coefficient, B_m	1.23×10^{-7}
Gear ratio (motor shaft/load shaft)	1/17.2
Load moment of inertia, J_l (N m s^2/rad^2)	2×10^{-5}
Load viscous friction coefficient, B_l (N m s/rad^2)	negligible
Power Amp gain, K_a	2.25 or 1

for the motor.

The Optical Encoder

The angular position of the motor is measured by an HP HEDS5540 incremental optical encoder. It translates rotary motion of the motor shaft into three TTL compatible digital outputs, Channels A, B and I. The output from Channel I is simply an index pulse which is produced once for each full rotation of the shaft. This output is used by the DLaB `Initialize()` routine to calibrate a zero reference position of the shaft. The position of the shaft and its direction of movement are obtained by decoding the outputs from Channels A and B. Decoding of these outputs is performed by a program running on the DSP board, and the position of the shaft can be obtained by calling the DLaB routine `ReadEncoder()`.

The DLaB routine `ReadEncoder()` returns an integer which represents the accumulated value of a counter. The value of this counter is a 16-bit signed number with the following meaning:

Value of counter	Meaning
+1024	Motor shaft has made 1 full clockwise rotation from the zero reference position.
0	Motor shaft is at the zero reference position.
−1024	Motor shaft has made 1 full counter clockwise rotation from the zero reference position.

The Digital Signal Processing Board

The main functions of the DSP board are to perform sampling of the motor position and to convert the output of the digital controller into an analog signal. As was mentioned earlier, sampling of the motor position is performed using an optical encoder and a software program running on the DSP board.

Digital to analog conversion is performed by a 16-bit D/A converter available on the DSP board which has been programmed to convert a 16-bit 2's-complement number linearly into an analog voltage between ±1.4 volts. Thus, a value of +32767 will be converted to 1.4 volts, and a value of −32768 will be converted to −1.4 volts. Values between −32768 and +32767 are obtained using linear interpolation.

The Power Amplifier

A power amplifier is required because the output current of the D/A converter is insufficient to drive the motor directly. In addition to increasing the output current, the output voltage of the D/A is also amplified by a factor of K_a. The value of K_a is either 1 or 2.25, depending on the module you are using. Your laboratory instructor will provide you with the value of K_a for your module in the class.

Appendix

B 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 Pthread and RTAI functions can be found through the course Web Page.

The `dlab.h` Header Files

Synopsis

```
#include "dlab.h"
```

Description

The header file `dlab.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.h` header file can be downloaded through the course Web Page, and should be placed in the working directory for the project.

The `Initialize()` Function

Synopsis

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

Description

The `Initialize()` function is used to select the mode of operation for the DLaB library functions. To use the DLaB library functions in the *Simulation* mode, set the value of `op_mode` to `DLAB_SIMULATE`. To use the DLaB library functions in the *Hardware* mode, set the value of `op_mode` to `DLAB_HARDWARE`. `DLAB_SIMULATE` and `DLAB_HARDWARE` are predefined constants in the `dlab.h` header file and their values should never be modified.

When working in the hardware mode, the `Initialize()` function sets up communication between the PC Workstation and the DSP board. It also sets the DSP board up to produce a hardware interrupt every `1/Fs` seconds.

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 obtained by consulting the file `lab_layout.pdf` in the project directory.

The `Initialize()` must be called before any of the remaining DLaB functions can be used. A return value of 0 by `Initialize()` indicates the call is successful.

Example

The following program segment shows how `Initialize()` is used to set up DLaB functions to work in simulation mode:

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

Once the simulation program has been debugged, it can be tested on the actual motor hardware by simply replacing `DLAB_SIMULATE` with `DLAB_HARDWARE` in the `Initialize()` function call.

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

Synopsis

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

Description

The `ReadEncoder()` function returns the 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.h"
...
float motor_position;
...
motor_position = EtoR(ReadEncoder());  // Determine the motor position
                                       //  in radians.
...
```

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

Synopsis

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

Description

The `DtoA()` function sends the digital value `digital_code` through the serial port of the PC Workstation to the D/A converter on the DSP board. The output of the D/A converter is an analog voltage between ±1.4 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.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.h"
void Terminate();
```

Description

The `Terminate()` function resets the DSP board and shuts down communication between the PC Workstation and DSP board. 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.h"
...
Terminate();
...
```

The `plot()` Function

Synopsis

```
#include "dlab.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 the graphs in Postscript for printing. 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.

Example

The following program segment shows how this function is used:

```
...
#include "dlab.h"
...
float v1[100], v2[100];
float Fs;
int no_of_points;
...
no_of_points = 50;
// Plot the graph of v1 and v2 vs time on the screen
plot(v1, v2, Fs, no_of_points, SCREEN, "Graph Title", "x-axis", "y-axis");
...
// Save the graph of v1 and v2 vs time in Postscript
plot(v1, v2, Fs, no_of_points, PS, "Graph Title", "x-axis", "y-axis");
...
```

The `Square()` Function

Synopsis

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

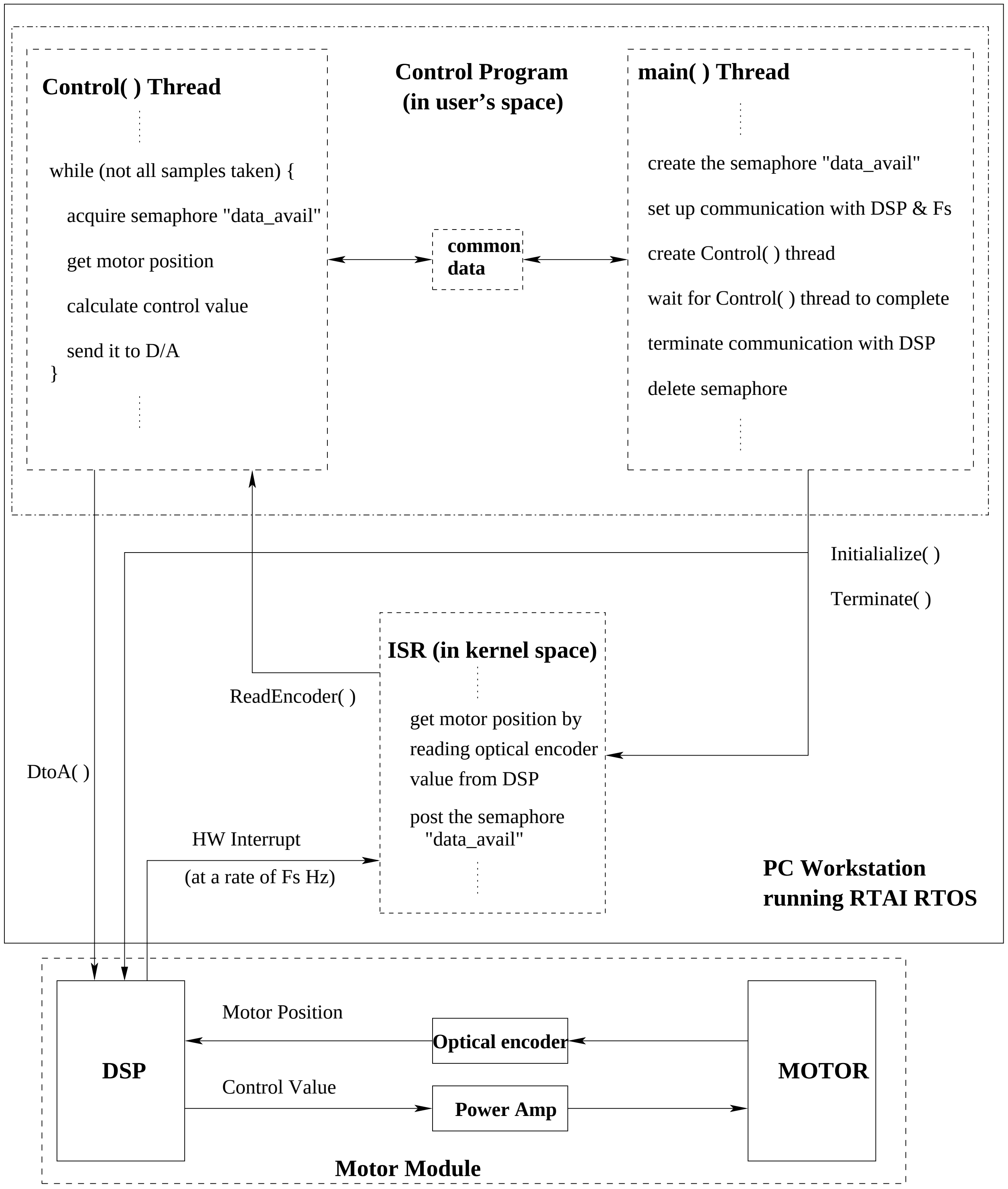

Figure 4: Overview of the Control System