SCHOOL OF ELECTRICAL ENGINEERING AND TELECOMMUNICATIONS
ELEC4633: Real Time Engineering
Laboratory – Real-Time Motor Control and Monitoring

Aim

The task is to design and implement software for real-time motor control and monitoring.

At the completion of this exercise, you should be able to:

- Write a device driver for real-time data acquisition and motor actuation.
- Implement an appropriate data storage mechanism.
- Interact with the real-time device driver through the use of Linux programs and appropriate inter-process communication methods such as message passing and shared memory in RTAI.
- Understand and implement a *server-client* application for data interaction with the real-time motor device driver.
- Implement simple feedback control of the motor position, while supplying an appropriate setpoint via a Linux program.

Specification

Two Linux programs (clients) will be responsible for "communicating" with a real-time device driver through a third Linux task (server). The first Linux program will request and display motor position data periodically from the server. The second will provide setpoint data to the server to be used in simple proportional feedback control of the motor position. The real-time device driver will exist as one or more real-time tasks, responsible for sampling motor position periodically at approximately 10Hz, and also providing a calculated control actuation at the same rate. Communication between the real-time task(s) and the Linux server will be through a shared memory.

LABORATORY CHECKPOINTS

Each checkpoint may attract a different mark. There are six (5) checkpoints in this lab exercise.

1 Introduction

This laboratory exercise will be carried out in several stages, gradually building up to a complete real-time monitoring and control application for the motor. The stages need to be implemented in order.

All .c codes needed to support the lab exercise are provided in the Lab 2 file and can be downloaded from Moodle.

Stage 0 - Overall Design Specification

A software process model, or *Q-Model* is useful for high level design. It allows you to depict your software system simply as a set of software processes as well as a set of messages that link these software processes. The links in the Q-Model do not need to specify how the messages are passed, just the content of the messages.

Exercise

In this preliminary stage, it is important to consider the complete system, and develop a plan for the software. The remaining stages are then used to gradually implement this plan. In this stage, you will draft a Q-Model for the real-time monitoring and control system, as well as specify the methods of message passing between each of the processes.

From the program specification above and from Figure 1 below, let us reiterate the responsibilities of the software:

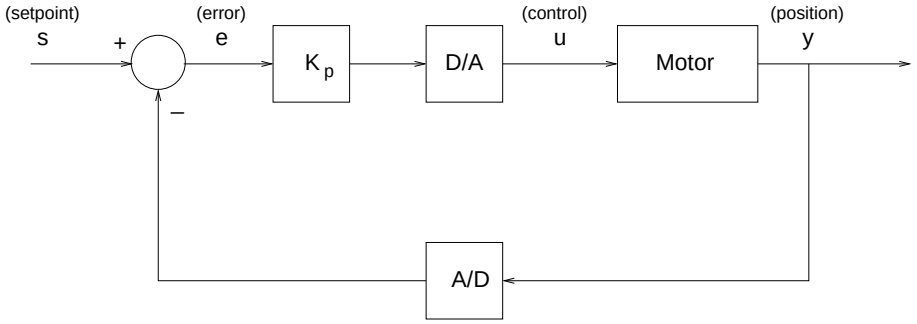

Figure 1: Proportional Motor Position Control

1. A real-time device driver is needed for data acquisition and control. This will mean that the program needs to sample the motor position periodically, calculate a control signal, and provide the actuation (send the control signal out to the motor). The sampled data will initially occur as digital data via the A/D converter. Storage of the data needs to occur in an appropriate buffer, preferably circular. All of these responsibilities can be carried out using one or more real-time tasks.
2. A Linux program will be needed to display data as it is generated. This program is known as a *client*.
3. A second client is needed to provide setpoint information to the real-time task calculating the control.
4. Both clients do not communicate directly with the real-time device driver, but through a Linux *server* program. Each client will communicate with the server program through the message passing facilities in Linux. The server is responsible for communicating directly with the real-time task(s) via the use of shared memory.

Remember, that this is only a high level design exercise, and you are not expected to provide more details of the (lower level) design. This will come in the following stages.

Draw your Q-Model, also indicating the methods of data communication.

Checkpoint #1 (1 mark) Marks will be given for correct Q-model

Stage 1 - Real-Time Data Acquisition and the Comedi Driver

In this stage, you are required to construct the real-time device driver responsible for data acquisition.

Exercise

1. The data being sampled is position data from the motor. The position will be sampled via the National Instruments data acquisition card (DAQ) inside the computer. This card has 12-bit A/D and D/A converters. The data acquisition card will be driven by the *Comedi* data acquisition driver project. The Comedi driver includes many functions to drive the data acquisition card, however for the purpose of this lab exercise, you will only really be interested in a few:

Function	Description
comedi_open	To open the device (DAQ) for use (use in init_module in real time).
comedi_data_read	To read analogue data from a A/D converter on the DAQ.
comedi_data_write	To write data to a D/A converter on the DAQ.

2. A sample program *ComediDriver.c* is provided in Moodle. In the sample program, construct a real-time periodic task for data sampling. Select an appropriate sampling rate for the motor. As a suggestion, approximately 10Hz would be sufficient. Think about what the data should look like. Keep in mind that the A/D converter is 12-bits.

Stage 2 - Data Conditioning and Storage

This stage builds on stage 1, by processing the sampled data and storing it for display by a Linux program.

Exercise

1. The sampled data exists in digital form, generated from a 12-bit A/D converter. Implement a storage buffer for the sampled data. In thinking about what size the buffer should be, a reasonable expectation is to store enough for approximately 10 seconds worth of data.
2. The buffer should be implemented circularly,... that is, a *circular buffer*. A circular buffer is one where the *last* element meets up with the *first*, such that there is really no end. So when you get to the end of the buffer, storage must continue from the start again, thus writing over older data.
3. Enter appropriate code into the provided sample program *ComediDriver.c*.

Checkpoint #2 (2 marks): Marks will be given for correct code inserted into the provided sample program *ComediDriver.c*.

Stage 3 - Simple Position Feedback Control

In this stage, we are going to "close the loop", by using the position data for simple *proportional* control. You are to implement the proportional control algorithm:

$$u = K_p e = K_p(s - y)$$

where s is the setpoint or desired position, y is the actual sampled position, e is the position error, K_p is known as the proportional gain (a constant), and u is the acutation or control signal to be applied to the motor input. The higher the gain, the faster the response of the motor to setpoint changes, and the closer the error is driven to zero (although there are trade-offs).

Exercise

1. Implement the proportional control algorithm shown above. Note that the motor has to reach the setpoint from its initial position through the shortest path as illustrated in Figure 2. Enter appropriate code into the provided sample program *ComediDriver.c*.
2. Once the controller is coded, provide the actuation to the motor via the use of the write functions in the *Comedi* driver. That is, the control signal u has to be "sent" to the motor input via the D/A converter. Enter appropriate code into the provided sample program *ComediDriver.c*.

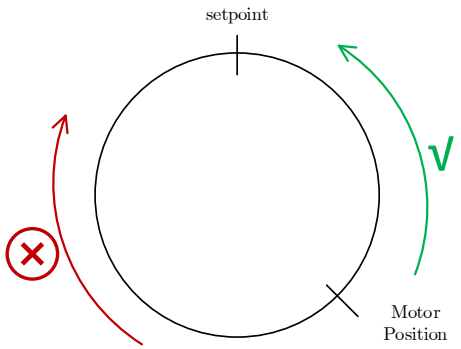

Figure 2: Motor Position Control

Checkpoint #3 (2 marks): Marks will be given for correct code inserted into the provided sample program *ComediDriver.c*.

Stage 4 - Client-Server Design for Displaying Data

You are to construct a $client-server$ pair of processes in this stage. The server program *serves* requests made by the client program. For stage 4, the client will be a Linux program which requires sampled data from the server, and then displays it. The server program will be responsible for providing the sampled data only. Rather than displaying one data every sampling period, have the program display *chunks* of data periodically, for example 10 data every second, or 50 data every 5 seconds, etc.

To communicate the data, shared memory will have to be set up in both the real-time task and Linux program. To keep count of the number of valid data on the buffer, you may need to implement a data counter of some sort. Depending on your implementation, you may need to keep track of the position of the newest data, as well as the position of the oldest data. When using shared memory, you will have to share the whole buffer, as well as the data counter and/or newest and oldest indices.

Communication between the client and server programs is not to be achieved via shared memory however. Data will be passed from the server to client via Linux *message passing*.

Message passing in Linux can be achieved by sending messages to, and receiving messages from, message queues. The following functions are used for this purpose:

msgget() To get a message queue identifier.

msgsnd() To send a message to a specified queue.

msgrcv() To receive a message from a specified queue.

You can check the *man* pages for more complete descriptions of these functions.

Importantly, when using a message queue, complex data structures can be sent/received, and different message "types" can be specified. That is, different actions can be taken by the receiving task, depending on what message type is specified.

Exercise

1. Construct a Linux client program which will display sampled data periodically. Rather than displaying data one at a time, display data in bulk at a slower rate is more appropriate.
2. Now create the server program. The server is responsible for providing the data (in chunks) to the display client.
3. You will need to think about how this can be achieved, as there is more than one way to implement the data transfer. For example, the server program could send the sampled data periodically regardless of whether the display client has requested it. Alternatively, the server could "sit" idle waiting for a request from the display client to send back available data, and then reply with the data "attached". This second approach involves more process synchronisation.
4. When referring to synchronisation, you will need to think about whether the client and server program will "block" on calls to msgsnd() and msgrcv(). That is, blocking when the message queue is full when using msgsnd(), or when the message queue is empty when using msgrcv().
5. Enter appropriate code into the provided sample code *Client.c*.
6. Enter appropriate code into the provided sample code *Server.c*.

Checkpoint #4 (3 marks): Marks will be given for correct code inserted into the provided sample programs *Client.c* and *Server.c*.

Stage 5 - Client/Server Design for Setpoint Specification

The final stage, extends the work done in stage 4, where an additional Linux client program is created to provide the setpoint value. The setpoint value can be entered in by the user. Once again, the "setpoint" client will send the setpoint data to the same server program from stage 4, which in turn shares the setpoint with the real-time thread code via shared memory.

Exercise

1. Create a Linux client program which prompts the user to enter a setpoint value in. Once a value is entered, the program can then send the setpoint to the server program via a message.
2. Modify the server program again, to accommodate an additional client sending it messages.
3. Enter appropriate code into the provided sample code *Client.c*.
4. Enter appropriate code into the provided sample code *Server.c*.

Checkpoint #5 (3 marks): Marks will be given for correct code inserted into the provided sample programs *Client.c* and *Server.c*.