SCHOOL OF ELECTRICAL ENGINEERING AND TELECOMMUNICATIONS
ELEC4633: Real Time Engineering
Laboratory Module 2 – 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.
- Deal with numerical issues such as data scaling.
- 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, RT-FIFO's 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 either RT-FIFO's or shared memory.

LABORATORY CHECKPOINTS

Please make yourself aware of the checkpoints throughout this lab exercise. You will need to demonstrate your work to one of the demonstrators in order to get marked off for each checkpoint. Each checkpoint may attract a different mark. There are six (6) 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.

Any additional code needed to support the lab exercise will be provided in the Lab 2 archive file lab2.tar.gz. This can be downloaded from Moodle, and "unpacked" using the command:

```
tar -xvzf lab2.tar.gz
```

2 Method

It is expected that this exercise will take **2+** laboratory periods to complete if you are sufficiently prepared. To complete the lab in this time, it may be also necessary to undergo some coding outside the scheduled lab times.

It is mandatory that when developing programs, you take an incremental approach, whereby you only add small sections of code at a time before testing.

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:

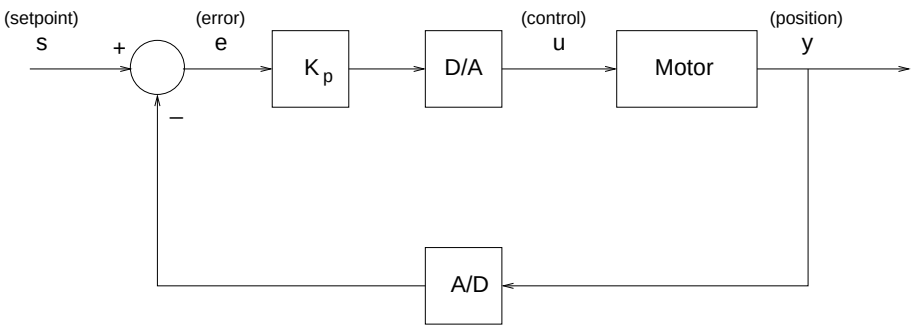

Figure 1: Proportional Motor Position Control

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. It will therefore need to be scaled appropriately to suitable units. 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 eith RT-FIFO's or 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.

Show the demonstrator your Q-Model, also indicating the methods of data communication, so that it can be checked before proceeding.

Checkpoint #1 (1 marks): Obtain a signature from your demonstrator.

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. Example code showing how to use this driver is contained within the lab 2 archive. 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.

   Have a look at the example code and see if you can get a feel for what each of the functions do. The code is a simple user space program, but the same function calls can be made within real time code. For more information about this driver, consult the demonstrators.
2. 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.
3. Test to see if it is functioning correctly by displaying the data inside the real-time thread. Think about what the data should look like. Keep in mind that the A/D converter is 12-bits, and observe what the motor is really doing so you can predict what the data will look like.
4. For working at home, or without the use of the data acquisition card, you code can simply replace the use of the driver functions with simple statements to simulate the sampling of data.

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. Scale the data, so that it is in "sensible" engineering units. These engineering units may the actual voltage range of A/D converter, or it may be a bipolar range, such as $[-5, +5]$, or $[-1, +1]$. Test the scaling function works by comparing the scaled data and unscaled data.
2. 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.
3. 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.

Checkpoint #2 (2 marks): Obtain a signature from your demonstrator.

Stage 3 - Data Communication and Display

Ultimately, the real-time device driver needs to communicate with Linux programs. Specifically, it will be sharing data with the Linux "server" program. In this stage, you are required to construct the Linux server program, although it will not be acting as a server just yet. For this stage, the Linux program is simply going to display the data sampled by the real-time task.

Exercise

1. Construct a Linux program (one with a `main()`) whose job it is to display the sampled data. 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.
2. To communicate the data, either RT-FIFO's or 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.

Checkpoint #3 (2 marks): Obtain a signature from your demonstrator.

Stage 4 - Simple Position Feedback Control

In this stage, we are going to "close the loop", by using the position data for simple *proportional* control. Modifications in this stage will mainly occur in the real-time task(s). 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 in code.
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. Scaling will have to be implemented again here as the control signal will be in "engineering" units, and the D/A converter expects a digital (12-bit) number. Be careful that signals don't "saturate", that is, go outside the required range (you cannot provide 120% actuation).
3. The setpoint can be "hard-coded" into the real-time code if required. Keep in mind that the setpoint range will necessarily be the same as the range for the sampled data specified in Stage 2.
4. Confirm that the controller is doing what it should do.... ensuring the measured position data tracks the setpoint signal "reasonably well". Experiment with the controller by changing the setpoint and gain to different values.

Checkpoint #4 (2 marks): Obtain a signature from your demonstrator.

Stage 5 - 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 5, 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. The Linux program constructed in stage 3 can be used as the server with some modification. It already has access to the real-time sampled data throught the shared memory buffer.

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 will need to check the *man* pages for more complete descriptions of these functions. Also, the lab 2 archive contains a simple example of two programs which communicate via these message passing functions. These are contained within the files msgsend.c and msgreceive.c.

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. Similar to stage 3, construct a Linux client program which will display sampled data periodically. Once again, rather than displaying data one at a time, display data in bulk at a slower rate is more appropriate.
2. Modify the Linux program from stage 3 to act as the server program. It will still have access to the shared memory containing the sampled data, but will now be responsible for providing the data (in chunks) to the display client.
3. You will need to think about how this is going to 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().

Checkpoint #5 (2 marks): Obtain a signature from your demonstrator.

Stage 6 - Client/Server Design for Setpoint Specification

The final stage, extends the work done in stage 5, 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 5, 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. Remember that the value has to be in the correct range specified in stage 2. 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. As mentioned earlier, the server can perform different jobs depending on what type of message is sent. That is, you can have a "display" type message, and a "setpoint" type message.
3. You will need to modify the real-time thread code so that the setpoint is no longer hard-coded as before.
4. Test the completed application, ensuring the real-time controller still responds as expected, with the user supplied setpoint data.

Checkpoint #6 (2 marks): Obtain a signature from your demonstrator.

3 Conclusion

At the successful conclusion of this lab exercise, you should be familiar with:

1. Using real-time tasks for the purpose of real-time data acquisition.
2. The type (digital/analog) and range of signals around a feedback control loop, as well as the associated issues such as data scaling, conditioning, and data storage.

3. How to write/use a simple real-time device driver for data acquisition and control.
4. The concept of a *server-client* program pair, and how to construct simple applications using servers and clients.
5. How to use Linux message passing methods for simple inter-process communication.
6. The use of simple proportional (P) control for controlling motor position.