Miniproject

First of all: The miniproject differs from the normal exercises in some major ways. The previous exercises are made to teach you new skills, but for this project you will instead demonstrate what you have learned. Your project will be graded (as opposed to just a complete/incomplete score), and it is up to you to find and document the solution you see as the best one. The student assistants will only offer limited guidance, and will for the most part only help you with technical support.

1. Project overview

The file `miniproject-files.zip` on Blackboard contains the files necessary for this miniproject. You will also need the toolchain from the NGW100 exercise.

The executable `miniproject-server` is a simulation of a physical system, which you must implement a controller for. This controller should be a program that runs on the NGW100. The server also sends other signals to the NGW100, and these should be responded to as soon as reasonable, without sacrificing controller performance. Finally, the server logs its interactions with your program, and saves the result in text files, that can be converted to plots using the `plot4` and `plot5` scripts.

1.1. The dynamic system

The system running on the server has the following transfer function:

$$h(s) = \frac{y(s)}{u(s)} = \frac{0.1s + 1}{0.000005s^3 + 0.006s^2 + s}$$

And is discretized with zero-order hold and a sample time of 0.001 seconds, giving the C implementation

```
y = 2.8851*y1 - 2.772*y2 + 0.8869*y3 + 0.00009613*u1 - 0.000003642*u2 - 0.0000923*u3;
```

Your controller should stabilize this unstable system, and make it able to follow a reference. The system also has a noise component on the input, which means that you must correct for a steady state error.

A PI controller with Kp=10 and Ki=800 is a good place to start, but you are free to use whatever kind of controller or parameters you want.

1.2. Communication interface

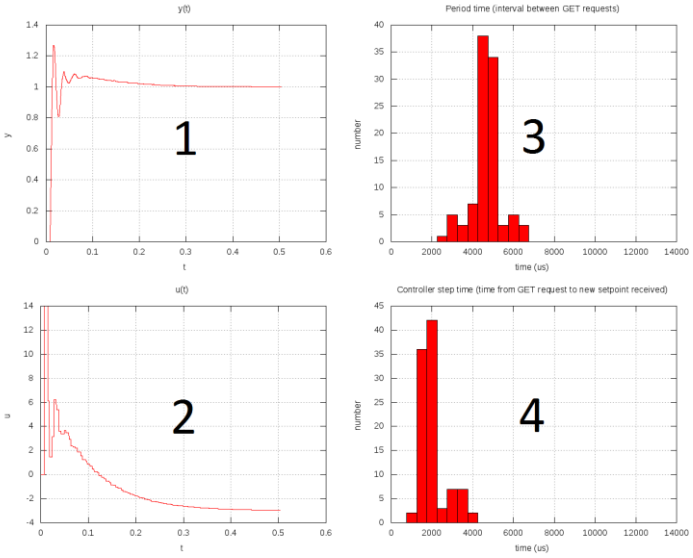
Communication between your client running on the NGW100 and the server running on the lab PC is done over the network, using UDP on port 9999. The contents of these packets are strings containing commands or values, and are outlined in the table below. The files `udp.c` and `udp.h` contain functions to create a connection, and send and receive UDP packets.

Command	String	Direction	Comments
START	"START"	To server	Starts the simulation
GET	"GET"	To server	Requests a `y` value from the server
GET_ACK	"GET_ACK:123.456"	From server	Returns the requested `y` value
SET	"SET:123.456"	To server	Sets the `u` value for the simulation
STOP	"STOP"	To server	Stops the simulation

1.3. Plotting

When the server simulation is started, it creates and appends to two `.dat` files containing the process simulation `y` and `u` values, as well as the timing information about the GET and SET commands. Run the following command to create a PNG image congaing graphs of the simulation results:

```
gnuplot plot4
```



1: The output from the simulation of the dynamic system (`y` value). It should ideally approach the set point quickly with minimal oscillations.
2: The input to the dynamic system (`u` values)
3: A histogram showing the distribution in timing between two successive GET requests. If your controller is running with a fixed period, these timings should be as close as possible to the timing you have specified in your code.
4: A histogram of the controller delay, that is the time between your client requesting a `y` value and setting a new `u` value. These timings can tell you something about the combined execution time and network delay time for your client, which should be lower than a fixed controller period.

2. Part one

Task A

Create a program that runs on the NGW100 that controls the simulation. The program should run for 2 seconds before ending the simulation. For the first second, the set point should be 1, then the set point should be set to 0. The response should show that the `y` value is stable after about 0.3 seconds (reasonable amounts of overshoot and a decreasing amount of steady state error are acceptable – use your judgement).

Almost everything you have used when programming for Linux before can be used on the NGW100 platform, including `pthreads`. However, `clock_nanosleep` is not implemented in the version of the C library that comes with Buildroot. An implementation of this (and some of the other `timespec` functions we have used previously) is found in `time.h` in miniproject.zip

If you are using a PID controller, the period is important for the control of the simulation. A period that is too long may struggle with controlling the system, and result in oscillation. A period that is too short means the execution time of your code and the network delays may be larger than the period you have set, resulting in incorrect calculations if you assumed that the period was constant. This will be harder to see on the simulation output plot, but you can see it clearly on the top right plot displaying the server's perspective of your controller period.

Example code for a PID controller is given in the miniproject.zip file from Blackboard.

Remember to send the START and STOP signals to the server.

3. Part two

In this part we will introduce a second part of the `miniproject-server` program. In addition to simulating a dynamic system, it also sends out signals that must be responded to, much like the response time test from the BRTT.

3.1. Communication interface

These signals are also sent as UDP packets, as described below

Command	String	Direction	Comments
SIGNAL	"SIGNAL"	From server	This packet should be responded to with a SIGNAL_ACK
SIGNAL_ACK	"SIGNAL_ACK"	To server	The response to a SIGNAL

The server will start sending signals when the simulation is started, and will stop when the simulation is stopped.

3.2. Plotting

When running the signal test, an additional file with results is created. To plot the content of this file together with the ones from the last part, use

```
gnuplot plot5
```

The fifth plot in the bottom right shows the distribution of the signal response times (time between SIGNAL sent and SIGNAL_ACK received).

If the gnuplot command gives errors, it might be that the `signal_time.dat` file is empty. This happens when the server does not get any responses to the signals.

Task B:

Extend your program from part one, such that it also answers the signals. In the first part, you could have implemented everything as a single thread, but now we have two tasks whose only interaction is sending and receiving messages on the network. You should use (at least) two different threads, one for the dynamic system controller, and one for responding to signals. You should probably receive the network messages in a thread that is separate from these two. How you choose to communicate internally between these threads is up to you.

If you are writing a kernel module (or you have some other very good reason to not use threads), you can disregard the requirement to use multiple threads.

Submission:

Your submission should include all your code, and any makefile or build script you used. You do not have to include any executables (because they just take space). You only

need to include the code as it looked after part two. Make sure your code is readable - code quality *does* matter this time.

In a separate file, you should briefly document why you have taken the engineering choices you have, including:

- How many threads you had, and why
- How they interact, and why that approach was chosen
- How you chose the period (and other parameters) for your controller
  Or what kind of controller you used, if it isn't periodic
- Analysis of the plots, including
  - The quality of the timing of both the period and response time of the controller
  - The quality of the response times for the signal task
  - How introducing the signal task affected the timings of the controller

If you are wondering if something else should be documented, ask the student assistants. You do not need to write an "essay" or "report", the format is up to you.

You should also include the at least one figure from both plot4 and plot5, and you can include more if you find that it is necessary for explaining something.

Use standard file formats for the text and images you include (txt, pdf, doc, png - and so on). Compressing all your files to a single zip/tar before uploading is also appreciated.