NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 3

Group 7:

Vegard Seim Karstang
Johannes Andersen
Daniel Fedai Larsen

November 18, 2016

1 Overview

In this exercise we were tasked with two things: writing a device driver for Linux and implementing a simple game utilizing this driver. The driver provides a stable and restricted way of communicating with the hardware. The requirement for our driver was for it to allow utilization of the buttons on the game-pad that is connected to the micro-controller. For our game, we decided to create "Pong". The frame-buffer on the micro-controller is used for updating the display, and simple game mechanics were implemented; i.e. collision and player and ball movement.

1.1 Solution

1.1.1 Driver

The driver, located in *driver_gamepad.c*, has two primary functions: *__init* and *__exit*. The init function, as the name implies, initializes the driver so that it is ready for use, while the exit function cleans up when the driver is closed.

We start by requesting a region of memory. This memory region corresponds to the GPIO memory locations we are interested in. We then map this to virtual memory. Then we set up interrupts for both **even** and **odd** and write the required values to the correct memory locations as we did in exercise 2. We finish by creating the char device so that our main program is able to access the driver's functions.

In addition two other methods are implemented: *signal_game* allows us to send signals to our game from the driver (this is called when an interrupt is detected) and *gamepad_read* allows us to actually read what values are stored in *GPIO_PC_DIN* (what buttons are pressed).

1.1.2 Game

The game is located in *game.c*. The game does not have direct access to the hardware, everything goes through the driver. We start by opening the driver from */dev/tdt4258* where it is created as a char device. We continue by setting up signals and opening the frame buffer from */dev/fb0*. We finish by mapping the frame buffers memory and if this succeeds, we go ahead and initialize our game. We have a game loop (*while(1)*) which constantly performs the following actions in sequence:

1. Clear the three objects in our scene
2. Update the players and ball positions
3. Draw the players and the ball

4. Write to the frame buffer (update the display)

5. Sleep for 20ms (to reduce energy consumption)

It is worth noting that we initially cleared and redrew the entire scene, meaning the entire frame buffer was redrawn every frame. This resulted in higher power consumption and a lower frame time. We later improved on this so that just the areas around where an object was in the previous frame is cleared and the new area in which the object resides is drawn. This is further discussed in 2.1 and 2.2.

The *driver_signal_handler* function is called whenever the game receives a signal from the driver that a button has been pressed. The game then calls *read* which is a function implemented by the driver that returns the result of reading specific memory locations (the *GPIO* pins).

There are quite a few helper functions in *game.c* that help us implement our game in a good way; these are described in the comments in *game.c*.

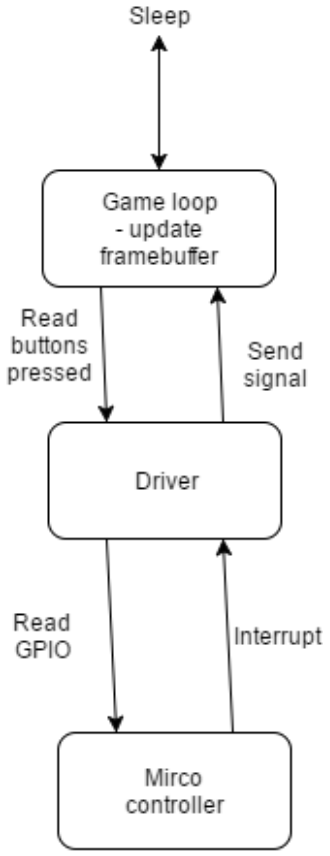

Figure 1.1: State machine

2 Energy Measurements

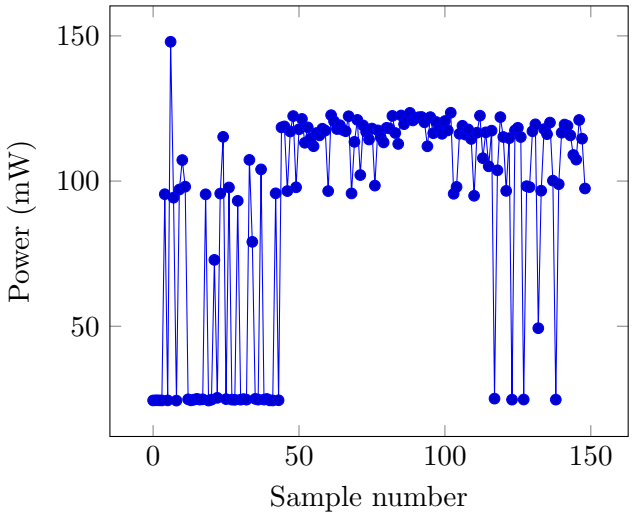

Figure 2.1: Power consumption plot generated from eAProfiler data: original solution

2.1 Original solution

In the original solution we updated the entire frame buffer for each frame. This means that each frame, the entire screen would be re-drawn (even if it still was just black). In many games this would be a requirement, but in a simple game like Pong, where in our case there are only three moving objects, this is not necessary, and leads to higher power consumption and lower frame time due to the micro controller having to "do more" each frame. The frame time was not measured, but we can see the power consumption results in Figure 2.1.

2.2 Improved solution

In our improved solution we altered how the frame buffer is updated. Instead of over-writing the entire buffer, we only over-write the sections that contained the data for the game pieces in the previous frame (clearing the data) and write to the sections that will contain data for the game pieces in the current frame being processed. This resulted in much better performance, about a halving in power consumption (from an

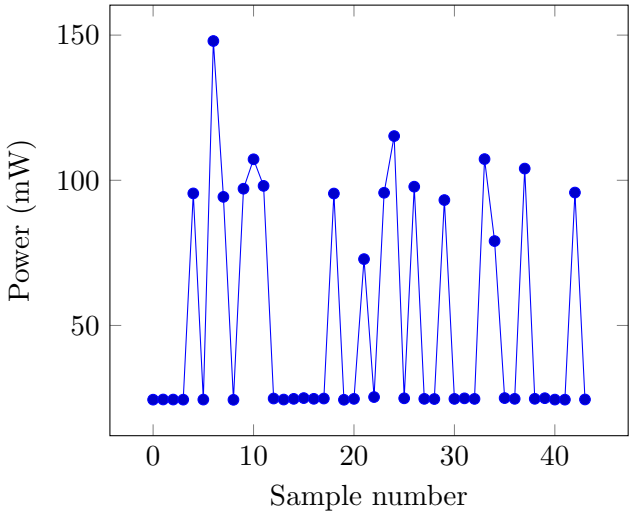

Figure 2.2: Power consumption plot generated from eAProfiler data: improved solution

average of 92mW to 53mW) and a big decrease in frame time. The results of the energy measurements can be seen in Figure 2.2. We clearly see that the micro controller spends a lot less time at approximately 100 mW.

One last thing to note is that both solutions implemented sleep (*usleep(20000)*) in order to decrease the power consumption. This is why we see the constant varying in the measurements.