NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 3

Group 5:

Antoni Climent Muñoz
Marius C. K. Gulbrandsen
Andrea Mazzoli

November 28, 2019

Contents

1 Introduction

1.1 Description

In this project, the goal was to make a retro 70's-style game. A Linux kernel module was made, in order to access specific hardware and used this in the user space to program the game. The module accesses the GPIO pins, which are the buttons on the gamepad. There was a discussion between choosing to implement Tetris, Pong or Snake, and the choice landed on Snake. Therefore, the Linux kernel module has been used to control the direction of the snake. How a snake game looks like is shown in figure 1.1.

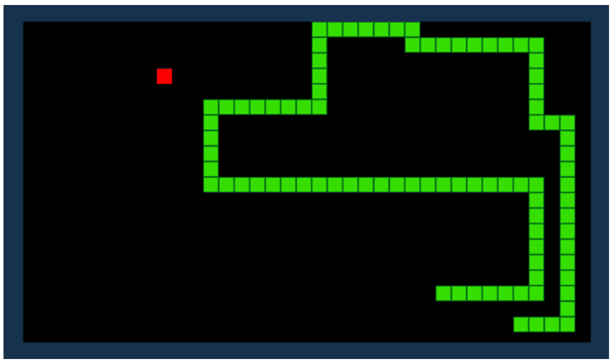

Figure 1.1: Snake game.

1.2 Game Specification

The game was first planned out in a specification that describes the functionality and limits of the game.

- The Snake is controlled with four buttons for *left, right, up, down.*
- The Snake cannot go through walls. If it hits a wall, it's game over.
- Eating yourself is game over.
- Eating one piece of food grows the snake at the tail.
- The food is randomly placed on the screen and never on the snake.

1.3 Extra Functionality

There is a start up screen with the SNAKE text to welcome the player. Additionally, there is a frame to define the snake-playing area. Planned extra functionality was:

- Play-button for starting the game.
- Pause the screen.
- Display Game Over.
- Record and display the highscore.

2 Solution

2.1 uClinux

In the first part of the exercise a Linux operating system (uClinux) had to be installed on the board. The main purpose of the system is that it is built through a build system called ptxdist. This will compile the kernel and the drivers so that everything could be flashed to the development board.

The aim of this part was also to familiarize with this kind of system and in particular with the Linux drivers. To do that the driver fb0 was used. This allow to print on the screen of the board, so this will be indispensable to do in the last part of the exercise when all the component will be together to be able to run the game on the board.

In Linux everything is represented as a file, so the fb0 driver was opened as a file and accessed in this way. To have a good implementation of the game will be use the function mmap() that allows to map the driver in an array directly in the memory. Thanks to that will be possible write pixels directly in an array that will represent the screen.

2.2 Kernel Module Implementation

This part explains how the gamepad-driver is realized. This has to be implemented as a kernel module. Its purpose is to read the input from the gamepad that is connected to the board. In this case, we have not used the LEDs on the gamepad, so the important part of the driver is the reading of the buttons. For doing so, it was set up memory requests. This is to book the hardware space that is needed for the reading of the buttons.

Then is is allocated the character device structure and got the number that the Linux system provided to the device (this number can change along different executions). After that it is activated the files functions (open, release, read and write), allocated, initialized (defining the file function that will be used) and passed the cdev structure to the Linux kernel. The last step was to make this driver visible in user space, in order to access it from the game. This was achieved using the functions class_create and device_create.

2.2.1 GPIO Setup

The setup of the GPIO pins was done in previous exercises, so the process was repeated in the kernel module setup. In that case, only the setup on the input of the gamepad was done, as for this exercise the LEDs were not needed and to not interfere with the OS, we did not use them.

2.2.2 Interrupts

Once the base solution was ended, it was tried to implement interrupts. All the steps needed are supposedly implemented in the code, but the interrupts did not work. To implement the interrupts the kernel was set up to send a signal to the user space when a GPIO interrupt happens. Upon getting this signal, a signal-handler in the user space would be invoked to handle the button presses.

2.3 Display

To control the display, a "display-driver" was implemented. This is of course not a real driver, but display-functions for drawing on the screen. As we are implementing snake, the most important thing would be to draw squares and rectangles. For doing this a **display_draw_rect()** function was implemented. To keep everything efficient a **display_refresh()** was implemented as well. The purpose of this function is to only refresh the part of the screen that we wrote to. The drawing function is shown below.

```c
void display_draw_rect(uint16_t x, uint16_t y, uint16_t w, uint16_t h,
    uint16_t color)
{
  // Variable calculations
  uint16_t start_pos     = SCREEN_WIDTH * y + x;
  uint16_t width_offset  = start_pos    + w;
  //uint16_t heigth_offset = start_pos    + SCREEN_WIDTH * h;  // Not used
  //uint16_t end_pos       = width_offset + SCREEN_WIDTH * h;  // Not used

  // Draw square
  for (int i = 0; i < h; i++) {

    uint16_t row_offset = i * SCREEN_WIDTH;
    for (int j = start_pos; j < width_offset; j++)
      map[j + row_offset] = color;
  }

  // Refresh the new part of the screen
    display_refresh(x, y, w, h);
}
```

Function for drawing a rectangle.

2.4 Game Coding

This game consists of a moving snake (controlled with the gamepad buttons) in order to eat the food that appears on the screen, making the snake longer. To do so, the driver was started by opening access to the driver work. Once did it a set of functions that were in charge of drawing the screen efficiently were provided, and then the logic implementation of the game was started. The main loop can be seen in the state machine figure bellow, but lets get a little bit more into detail.

In this game the aim is to move the Snake (controlled with the gamepad buttons) in order to eat the food that appears on the screen, making the snake longer. To do so, the driver created was started by opening access to the driver work. Once did it a set of functions that were in charge of drawing the screen efficiently were realized, and then the logic implementation of the game was started. The main loop in figure 2.1 is represented will be explained in details below.

First of all, all the following parameters are initialized: the first piece of food position, the length of the snake and the position of it. Then comes the main loop, that consists on a switch case that makes the parameter passed to the snake_update function change depending on the button pressed, in order to change the behaviour of the next update. At the end of this function is always displayed the changes on the screen (only the part of the screen that is being actually changed). Evaluating the update function it is possible explain the several steps. First of all it checks if it is in a game over state. This is important to check if the snake is trying to eat its tail or if it has hit the wall, because in both the cases the game is over. If so, the game will return into the initial state. Then, depending on the button pressed (or not pressed) it will calculate the new head direction and update the position of each block forming the snake. If the head is touching food, then it is added a new block in the tail in order to make it bigger. The ending game loop let us the following flow of states:

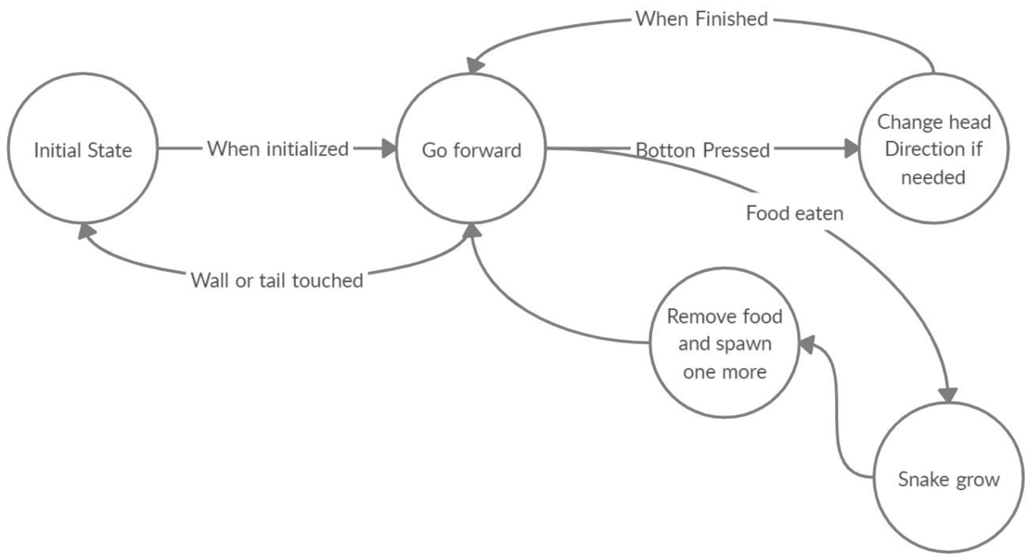

Figure 2.1: State Machine of the game loop

3 Results

3.1 Final Game

The start up screen for the game can be seen in figure 3.1 and the finished game is shown in figure 3.2.

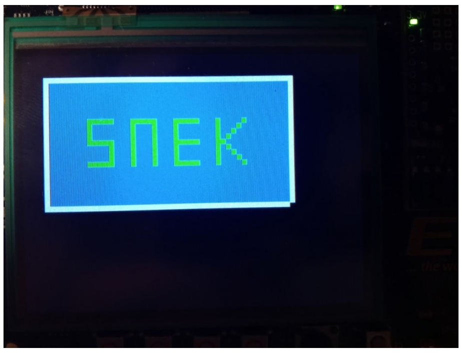

Figure 3.1: Start up screen.

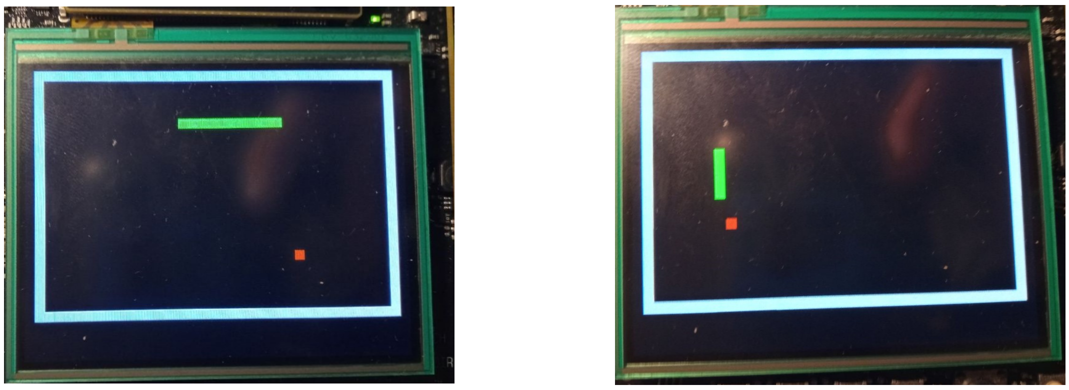

Figure 3.2: Finished snake game.

4 Conclusion

In this exercise, there was not implemented any energy saving in the same way as in exercise 1 and 2. As Linux is used, the OS is in charge of energy usage, and it is not easy to control, as many other things are being executed. Even so, to make the game as energy efficient as possible, the game sleeps for as long as it can before updating the screen and going back to sleep. The display only updates the new parts of the screen to make it more efficient.

The interrupts were going to be implemented and the code was very close to working, but the time to finish it was sadly not there. In the view of energy efficiency in our snake-design, it could be argued that interrupts would be even less efficient, as it would have to wake up at every button press to run the signal handler. In our current design the CPU sleeps for the entire time, and only upon waking up does it read the GPIO input and update the screen. This has another drawback of course, and that is the input-lag of the buttons, as we can only read the button in the "awake" state.

Playing the game, it runs as expected and the only noticeable issue is the input lag due to lack of interrupts, which makes the game slightly harder to play, but still lots of fun!