NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4258 - Low-level Programming

Assignment 3

Group Number: 14
Board Number: EFM17/3

Group Members:
Stian Strømholm
Håkon Kindem
Stian Hubred
Torbjørn Bratvold

November, 2020

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

This report contains the description of a set of C programs and linux kernel drivers which implement the classic snake game on a EFM32GG board. The snake game can be controlled by pressing buttons on a gamepad, and the game itself is drawn on a display on the controller board.

The snake program consists of a set of functions organized in several c code files. A kernel driver exposes gamepad buttons to userspace programs through use of interrupts and signals. The snake game consists of four C-code files, one handling incoming signals from the gamepad kernel driver, a framebuffer handler drawing the game onto the controller deisplay, a file containing the snake game and related structs, and a file running the game. A picture showing the game in action is included in Figure 1.

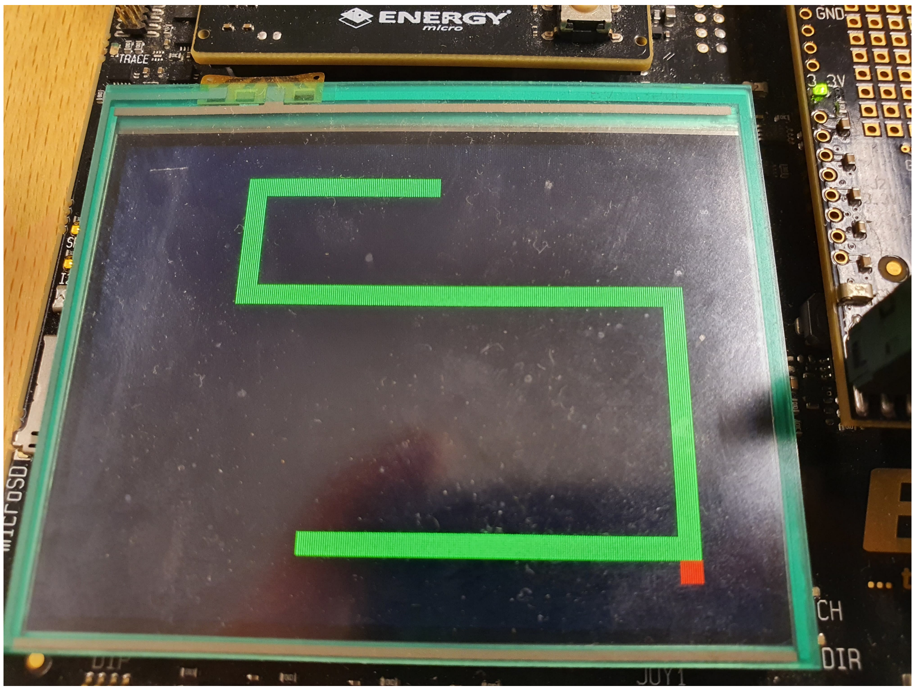

Figure 1: The snake game being played on a EFM32GG board.

2 Detailed tasks

The following sections cover interesting parts of the code package. The gamepad driver and the snake game is the main focus of this section.

2.1 Gamepad kernel driver

The gamepad kernel driver handles gamepad input to the micro-controller. This is done by configuring the buttons for input, setting up hardware interrupts on button presses, reading the button status when an interrupt is triggered, and asynchronously notifying the user-space application once the button status has been read. The user-space application can then read the button status from the driver.

The driver handles initialization and de-initialization through one init and one exit function. The driver does include error handling for initialization, making sure the driver is correctly set up before handling any interrupts.

2.1.1 Interrupt handling

Listing 1 shows the implemented interrupt handler function GPIO_IRQHandler. This function reads button status, clears the interrupt flag and sends a signal to the user-space application.

```
struct cdev cdev;
dev_t devno;
struct class *cl;
struct resource *gpio;
struct fasync_struct *async_queue;
uint8_t button_status;

struct file_operations gamepad_fops = {
        .owner = THIS_MODULE,
        .read = gamepad_read,
        .write = gamepad_write,
        .open = gamepad_open,
        .release = gamepad_release,
        .fasync = gamepad_fasync,
};

/*
 *
 * INTERRUPT HANDLER
 *
 */

irqreturn_t GPIO_IRQHandler(int irq, void *dev_id, struct pt_regs *regs)
{
        printk("driver: interrupt received\n");

        // Write the pressed buttons to the button status variable
        button_status = ioread32(GPIO_IF);

        // Clear interrupt flags
        iowrite32(0xff, GPIO_IFC);

        // Notify user application(s) through async signal
        if (async_queue) {
                kill_fasync(&async_queue, SIGIO, POLL_IN);
        }

        return IRQ_HANDLED;
}
```

Listing 1: Kernel driver code.

2.1.2 Init and exit functions

The init function in Listing 2, gamepad_init(void) handles initialization of the driver in a "safe" manner, making sure the driver is not available for use until the initialization has successfully completed. The function will abort and produce error status messages if any steps fail to complete. The init function allocates memory space, registers and sets up the hardware device, the gamepad, and requests hardware GPIO interrupts.

The exit function, gamepad_cleanup(void) is important for correct kernel operations. This function ensures de-allocation of memory, removal of the registered device and release of reserved interrupt lines. The kernel will lockup over time without these actions as the driver will gradually take up resources without releasing them, effectively making the resources unavailable to the kernel after exiting the driver.

```
/*
 * template_init - function to insert this module into kernel space
 *
 * This is the first of two exported functions to handle inserting this
 * code into a running kernel
 *
 * Returns 0 if successfull, otherwise -1
 */

static int __init gamepad_init(void)
{
        // Allocate major and minor numbers
        int err = alloc_chrdev_region(&devno, 0, 1, DRIVER_NAME);

        if (err < 0) {
                printk("driver: can't get major number.\n");
                return err;
        }

        // Register device
        cdev_init(&cdev, &gamepad_fops);
        cdev.owner = THIS_MODULE;
        err = cdev_add(&cdev, devno, 1);

        if (err < 0) {
                printk("driver: error %d adding gamepad.\n", err);
                return err;
        }

        // Make device visible
        cl = class_create(THIS_MODULE, DRIVER_NAME);
        device_create(cl, NULL, devno, NULL, DRIVER_NAME);

        // Allocate the GPIO part of memory
        gpio = request_mem_region(GPIO_PA_BASE, GPIO_MEM_SIZE, DRIVER_NAME);

        if (gpio != NULL) {
                printk("driver: GPIO memory allocated to gamepad driver.\n");
        }

        // Set up the buttons, like in previous exercises
        iowrite32(0x33333333, GPIO_PC_MODEL);
        iowrite32(0xff, GPIO_PC_DOUT);

        // Request interrupt lines for even and odd gpio interrupt
        request_irq(IRQ_NUM_GPIO_EVEN, (irq_handler_t) GPIO_IRQHandler, 0,
            ↪ DRIVER_NAME, &cdev);
        request_irq(IRQ_NUM_GPIO_ODD, (irq_handler_t) GPIO_IRQHandler, 0,
            ↪ DRIVER_NAME, &cdev);

        // Enable GPIO interrupt generation, like in previous exercises
        iowrite32(0x22222222, GPIO_EXTIPSELL);
        iowrite32(0xff, GPIO_EXTIFALL);
        iowrite32(0xff, GPIO_IEN);

        // Initialize button_status
        button_status = 0x00;

        printk("driver: Hello World, here is your gamepad speaking\n");
        return 0;
}

/*
 * template_cleanup - function to cleanup this module from kernel space
 *
 * This is the second of two exported functions to handle cleanup this
 * code from a running kernel
 */

static void __exit gamepad_cleanup(void)
{
        // Undo allocations from init, reverse order
        // Release interrupt lines
        free_irq(IRQ_NUM_GPIO_EVEN, &cdev);
        free_irq(IRQ_NUM_GPIO_ODD, &cdev);

        // Release GPIO memory region
        release_mem_region(GPIO_PA_BASE, GPIO_MEM_SIZE);

        // Delete cdev
        cdev_del(&cdev);

        // Unregister major and minor numbers
        unregister_chrdev_region(devno, 1);

        printk("driver: Short life for a small module...\n");
}
```

Listing 2: Kernel driver code.

2.1.3 Kernel driver access

The kernel can access the driver in various ways. Opening and writing is simply handled by writing this event to console. The driver does not have any functionality for these actions. Release of the driver is handled by the gamepad_release function which de-registers the driver for asynchronous signals. gamepad_read enables the ability to read the status of the buttons and sending this to the user application. This gives access to the core driver functionality. gamepad_fasync registers and de-registers the gamepad driver for asynchronous signals.

```
/*
 * file op functions
 */

static int gamepad_open(struct inode *inode, struct file *filp)
{
        // We don't do anything here
        printk("driver: gamepad driver opened.\n");
        return 0;
}

static int gamepad_release(struct inode *inode, struct file *filp)
{
        // remove from async queue (signals)
        gamepad_fasync(-1, filp, 0);

        printk("driver: gamepad driver released.\n");
        return 0;
}

// Sends an 8 bit number showing the status of each of the 8 buttons: 1 = pressed
static ssize_t gamepad_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
        // We send the status of the buttons to the user application
        copy_to_user(buf, &button_status, 1);

        printk("driver: gamepad driver read.\n");
        return 1;
}

static ssize_t gamepad_write(struct file *filp, const char __user *buf, size_t count, loff_t *offp)
{
        // The driver has no write functionality

        printk("driver: gamepad driver written to.\n");
        return 0;
}

static int gamepad_fasync(int fd, struct file *filp, int mode)
{
        // Register user application for async signals

        printk("driver: fasync invoked.\n");
        return fasync_helper(fd, filp, mode, &async_queue);
}
```

Listing 3: Functions handling operations done with the driver.

2.2 Reading buttons in user-space

The read-buttons module utilizes the gamepad driver to read the button status from the user-space application. It consists of two functions: setup_driver and signal_handler, and the code is shown in Listing 4.

The setup_driver function first opens the char driver file, exiting if the operation is unsuccesful, and then registers the user-space application for asynchronous notification from the gamepad driver through signals. In the driver, this invokes the gamepad_fasync function, as described in the previous subsection, and the gamepad driver will from that point on signal the user-space application when a button has been pressed. The function signal_handler is registered as the handler for such signals. When invoked, the signal_handler function will first obtain the button status by reading the gamepad drver, then translate it into a character signifying which button was pressed, and finally invoke the on_button_pressed function in the snake module with the character as an argument. The on_button_pressed function is discussed in section 2.4.

```

static FILE *driver_file;
```

```
void setup_driver()
{
    // Open driver

    driver_file = fopen("/dev/gamepad_driver", "rb");

    if ( driver_file == NULL ) {
        printf("game: error opening gamepad-driver.\n");
        exit(1);
    }
    printf("game: opened gamepad-driver.\n");

    // Reqeust Async Notification (signal)

    signal(SIGIO, &signal_handler);
    fcntl(fileno(driver_file), F_SETOWN, getpid());
    int oflags = fcntl(fileno(driver_file), F_GETFL);
    fcntl(fileno(driver_file), F_SETFL, oflags | FASYNC);

        printf("game: driver setup completed.\n");
}

void signal_handler(int sig)
{
        printf("game: signal detected.\n");

    if (sig == SIGIO) {
        uint8_t button_status = (uint8_t) getc(driver_file);
        char button = (char) 0;

        if (button_status & 0x01) {
            printf("game: button L pressed.\n");
            button = 'L';
        }

        else if (button_status & 0x02) {
            printf("game: button U pressed.\n");
            button = 'U';
        }

        else if (button_status & 0x04) {
            printf("game: button R pressed.\n");
            button = 'R';
        }

        else if (button_status & 0x08) {
            printf("game: button D pressed.\n");
            button = 'D';
        }

        else if (button_status & 0x20) {
            printf("game: button u pressed\n");
            button = 'u';
        }

        else if (button_status & 0x80) {
            printf("game: button d pressed.\n");
            button = 'd';
        }

        if (button != (char) 0) {
            on_button_pressed(button);
        }
    }
}
```

Listing 4: read-buttons source code.

2.3 Framebuffer driver

Listing 5 contains the code necessary for using the framebuffer driver together with the game to produce graphics on the EFM32's screen. The function initialize_screen() initializes the framebuffer and maps it to an array in the memory using mmap(). Each element in this array corresponds to one pixel on the screen.

The function clear_screen() will delete the mapping to the memory, while the update_screen() function will update the screen. Blackout_screen() is used to reset the screen to the default black color before drawing the objects.

The two graphic aspects of the game, namely the apples and the snake itself, can be broken down to squares. The draw_square() function takes in the x and y coordinates of the upper left corner of the square as well as the color of the square. The size of the square is 10x10 pixels. This draw function is utilized in the snake program to draw squares in the correct positions.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdint.h>
#include <linux/fb.h>
#include <sys/ioctl.h>

#include "framebuffer.h"

int fbfd;
int pos_x;
int pos_y;
int i;
int j;
int screensize;
uint16_t* screen;
struct fb_copyarea rect;
static uint16_t colors[] = {BLACK, GREEN, RED, DARK_GREEN};


int initialize_screen()
{
        printf("initialize_screen\n");

        fbfd = open("dev/fb0", O_RDWR);
        if(fbfd == -1)
        {
                printf("open_error\n");
                exit(1);
        }
        printf("opened_file");


        rect.dx = 0;
        rect.dy = 0;
        rect.width = 320;
        rect.height = 240;

        printf("setting_screen_settings\n");

        screensize = rect.width*rect.height*2;
        printf("Calculating_screensize\n");

        screen =(uint16_t*) mmap(0, screensize, PROT_READ | PROT_WRITE,
            ↪ MAP_SHARED, fbfd, 0);
        printf("Mapped_screen_to_memory\n");
        if(screen == MAP_FAILED)
        {
        printf("mapping_screen_error\n");
        close(fbfd);
        exit(1);
        }
        printf("Finished_initialize\n");

        return 1;
}

void clear_screen()
{
        munmap(screen, screensize);
        close(fbfd);
}

void blackout_screen()
{
    int i;
    for(i = 0; i < screensize; i++) {
        screen[i] = colors[0];
    }
}

void update_screen()
{
        ioctl(fbfd, 0x4680, &rect);
}

void draw_square(int pos_x, int pos_y, int color_val)
{
        for(i=0; i<BLOCK_SIZE;i++)
        {
                for(j=0; j<BLOCK_SIZE; j++)
                {
                        screen[pos_x+j+(i+pos_y)*rect.width] = colors[color_val];
                }
        }
```

```
}
```

Listing 5: Framebuffer source code.

2.4 Snake program

For our game we decided to recreate the classic snake game. The game follows the rules that the snake grows one block for each apple it eats. The game ends if the snake tries to eat itself or goes out of bounds. The snake itself is represented as the struct we can see in Listing 6. The board is represented as a one dimensional array, but for easier interaction between the board and the snake, Cartesian coordinates are used for the main game logic and then converted to one dimensional array positions.

The board_size variables are used for checking that the snake is inside of the board and for converting between 1D and 2D coordinates.The _pos variables are used for updating and keeping track of the movement and position of the head of the snake. The tail_indx is used to keep track of the length of the snake body. Each block in the snake body is represented by each number in the body array. The corresponding number represent the body part position in a 1D board array. The array positions with no corresponding snake part are initialised to -1. The alive variable is 1 if the snake is alive and 0 otherwise, the variable being 0 means game over. apple_pos is the position of the apple in the 1D board array. The direction char is the current moving direction of the snake represented by L (left), R (right), U (up), D (down). next_direction is used to store the player inputted direction of the snake. And will update the direction char each time the snake moves.

```
struct Snake
{
    uint8_t board_size_x;
    uint8_t board_size_y;
    int8_t x_pos;
    int8_t y_pos;
    uint16_t tail_indx;
    uint32_t body[SNAKE_MAX_LENGTH];
    bool alive;
    bool game_running;
    uint32_t apple_pos;
    char direction;
    char next_direction;
};
```

Listing 6: Snake struct

Listing 7 shows the handling of button presses. Two buttons have been assigned to start and reset of the game board. Four other buttons handle four directions. only two directions will work at any given time, namely the ones normal to the direction of travel on the board.

```


static struct Snake snake;

void on_button_pressed(char button) {
    switch (button)
    {
        case 'u':
            if (snake.game_running) {
                stop_game();
            }
            else if (snake.alive) {
                start_game();
            }
            break;

        case 'd':
            snake_reset();
            break;

        case 'L':
            if (snake.alive && snake.direction != 'R') {
                snake.next_direction = 'L';
            }
            break;

        case 'U':
            if (snake.alive && snake.direction != 'D') {
                snake.next_direction = 'U';
            }
            break;

        case 'R':
            if (snake.alive && snake.direction != 'L') {
                snake.next_direction = 'R';
            }
            break;

        case 'D':
            if (snake.alive && snake.direction != 'U') {
                snake.next_direction = 'D';
            }
            break;

        default:
            break;
    }
}
```

Listing 7: Snake button press handler

Listing 8 Shows the case handling for snake death conditions. The snake dies whenever it leaves the screen dimensions or whenever it collides with itself.

```
void snake_is_dead() {
    if (snake.x_pos == snake.board_size_x || snake.y_pos == snake.board_size_y){
        snake.alive = false;
        return;
    }
    if (snake.x_pos < 0 || snake.y_pos < 0){
        snake.alive = false;
        return;
    }

    int i;
    for (i = 1; i < SNAKE_MAX_LENGTH; i++) {
        if (snake.body[0] == snake.body[i]) {
            snake.alive = false;
            return;
        }
    }

}
```

Listing 8: Snake death condition handler

Listing 9 shows how the updating of head position and body position is handled in the update_snake_head_position(). Input from the button press handler in Listing 7 is used to determine and update head movement direction. update_snake_body_pos() iterates through a array containing body element positions and moves them one position up. head_pos_to_array_pos() simply adds the hew head pos to index 0 of the body position array.

```


void update_snake_head_position() {
    switch (snake.direction)
    {
    case 'R':
        snake.x_pos++;
        break;

    case 'L':
        snake.x_pos--;
        break;

    case 'U':
        snake.y_pos--;
        break;

    case 'D':
        snake.y_pos++;
    default:
        break;
    }
}

void update_snake_body_pos() {
    int i;
    for (i = snake.tail_indx; i > 0; i--) {
        if (snake.body[i] != -1) {
            snake.body[i] = snake.body[i - 1];
        }
    }
    return;
}

void head_pos_to_array_pos() {
    int array_pos = snake.x_pos + snake.y_pos*snake.board_size_x;
    snake.body[0] = array_pos;
    return;
}
```

Listing 9: Snake head and body position handlers

Listing 10 shows functions implementing logic for apple spawning on the game screen and handling of an apple being eaten whenever the snake hard hits an apple. the spawn_apple function spawns an apple at a random position on the display and verifies that this position does not overlap with the snake body, moving the apple if this is the case. The eat function simply increments the snake tail array position whenever the head position is equal to the apple position. The function ends by spawning another apple using the spawn_apple() function.

```
void spawn_apple() {
    time_t t;
    short apple_pos = −1;
    /* Intializes random number generator */
    srand((unsigned) time(&t));


    int board_size = snake.board_size_x*snake.board_size_y;
    while(1){
        apple_pos = rand() % board_size;
        int i;
        for (i = 0; i <= snake.tail_indx; i++) {
            if (snake.body[i] == apple_pos) {
                apple_pos = −1;
                break;
            }
        }
        if (apple_pos != −1) {
            snake.apple_pos = apple_pos;
            return;
        }
    }

}


void eat(uint32_t last_tail_pos) {
    if (snake.body[0] == snake.apple_pos) {
        if (snake.tail_indx < SNAKE_MAX_LENGTH − 1) {
            snake.tail_indx++;
            snake.body[snake.tail_indx] = last_tail_pos;
        }
        spawn_apple();
    }
}
```

Listing 10: Apple spawn and apple eating handlers

Listing 11 shows the move_snake function which updates relevant position arrays before checking whether the apple is eaten using functions described in earlier sections.

```
void move_snake() {
    snake.direction = snake.next_direction;
    uint32_t last_tail_pos = snake.body[snake.tail_indx];
    update_snake_body_pos();
    update_snake_head_position();
    head_pos_to_array_pos();
    snake_is_dead();
    eat(last_tail_pos);
}
```

Listing 11: Function handling snake movement updates.

The game graphics are drawn on the controller display by the draw_apple and draw_screen function in Listing 12.

```
void draw_apple()
{
        int x_pos = snake.apple_pos % snake.board_size_x * BLOCK_SIZE;
        int y_pos = snake.apple_pos / snake.board_size_x * BLOCK_SIZE;
        draw_square(x_pos, y_pos, 2);
}

void draw_screen(){
    blackout_screen();
    int j = 0;
    while (snake.body[j] != -1) {
        int x_pos = (snake.body[j]%snake.board_size_x) * BLOCK_SIZE;
        int y_pos = (snake.body[j]/snake.board_size_x) * BLOCK_SIZE;
        if (j == 0) {
            draw_square(x_pos,y_pos, 3);
        }
        else {
            draw_square(x_pos,y_pos, 1);
        }
        j++;
    }
    draw_apple();
    update_screen();
}
```

Listing 12: Functions using the framebuffer to draw game graphics on the micro-controller display.

The game needs functions for starting, stopping and resetting a game of snake. Listing 13 Implements these functions. Snake_reset sets appropriate values and draws the screen once before a game is started. run_game continuously updates the snake position and redraws the screen. this function runs until the snake dies or the game_running variable is set to false. stop_game and start_game simply sets a value controlling the while loop inside run_game.

```
void snake_reset()
{
    snake.board_size_x = BOARD_SIZE_X;
    snake.board_size_y = BOARD_SIZE_Y;
    snake.x_pos = 0;
    snake.y_pos = 0;
    snake.tail_indx = 0;
    snake.alive = true;
    snake.game_running = false;
    snake.apple_pos = 200;
    snake.direction = 'D';
    snake.next_direction = 'D';

    snake.body[0] = 0;
    int i;
    for (i = 1; i < SNAKE_MAX_LENGTH; i++) {
        snake.body[i] = -1;
    }

    draw_screen();
}
```

```
void run_game()
{
    if (!snake.game_running) {
        return;
    }

    while (snake.game_running && snake.alive) {
        move_snake();
        draw_screen();
        usleep(SNAKE_INTERVAL);
    }

    printf("snake: snake is dead.\n");
}

void stop_game()
{
    snake.game_running = false;
}

void start_game()
{
    snake.game_running = true;
}
```

Listing 13: Functions using the framebuffer to draw game graphics on the micro-controller display.

2.5 The main function

The main function is the starting point for the user-space program, and is shown in Listing 14. It will first initialize the previously discussed modules by invoking the functions initialize_screen, setup_driver and snake_reset. It then enters an infinte loop where it invokes first the pause function from the unistd library, and then the run_game function from the snake module. The pause function will stop execution of the program until a signal is received, and the run_game function runs the snake game, returning once the game is not running. In short, the function of the loop is to wait for a button press when the game is not running, instead of exiting the program.

```
int main(int argc, char *argv[])
{

        printf("Hello World, I'm game!\n");
        initialize_screen();
        setup_driver();
        snake_reset();

        while (true) {
                pause();
                run_game();
        }

        return 0;
}
```

Listing 14: The main function of the user-space program.

3 Conclusion

The kernel driver works as intended, receiving and handling button changes on the gamepad. The snake game is able to use signal input from the kernel driver. The game works as intended, implementing the classic snake game on the EFM32 controller.

Bibliography