NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

TDT4258 Exercise 3

Group 10:

Anders Sildnes
Rasmus Stene
Ole Sørlie

November 21, 2016

1 Overview

This report outlines our solution for how to implement a small video-game in an energy-efficient fashion using the EFM32 Giant Gecko ARM Cortex-M3 micro-controller. Our code is written in C and interfaces with standard Linux system libraries and the Embedded Application Binary Interface (EABI) of the EFM32. For user interaction we have connected a DK3750 development board which has eight programmable buttons that can be held down and released. The DK3750 interfaces with our video-game as a Linux device driver. We also customize the Linux kernel to function better with our game and micro-controller.

The next sections describe how we have written our code and what optimizations we have applied to reduce overall power usage. In the end, in Chapter 2, we show power usage for our application.

1.1 StepMania: the Game

In StepMania, stationary arrows are placed on the top or the bottom of the screen. Then, arrows appear from the opposite side, sliding across the screen. Whenever the moving arrows align with the stationary arrows, players need to hit either left/up/-down/right respectively to what arrow that is shown. An example is shown in Figure 1.1. Players are given points whenever they hit a button and a sliding arrow is aligned with a stationary arrow. The closer they are to a perfect alignment, the more points they get. If the distance between an arrow and its stationary is too big, the players' score is reduced.

Additional rules in our implementation of the game is that an arrow may only affect your score once. In addition, arrows that go off the screen without the player hitting any button will automatically deduct health from the player. We did not implement other features such as needing to hit multiple buttons at once, or holding down buttons for a longer period of time. Furthermore, points will only be given to a player if an arrow is within a certain proximity of the stationary arrows. Hitting the "SW8" button ends the game. In order to re-start, users must re-launch the program. Game difficulty is decided before game starts, and refers to how likely it is that arrows come up on the screen.

1.1.1 Implementation

There are several global variables and one array of struct "Marker" used to keep track of state inside the game. The struct marker is for each arrow that moves across the screen, and its code is shown in Listing 1.1.

Figure 1.1: Screenshot of a typical StepMania in-game scene.

The game runs on two asynchronous event handlers, one that is triggered by a timer event and another that is triggered for every button push. We have implemented this using Linux signals[1] [3]. The reason we have a timer event is to control the speed of the game, that is, the interval between each movement of the markers. We chose to use a timer interrupt rather than sleeping as this is thought to be more energy efficient than polling on a sleep event.

Our program (and kernel) is not multi-threaded, which means that our code is subject to race conditions between events. To minimize the chance of an error happening, we only use methods marked "async-signal-safe" by the POSIX.1-2004 standard[3], and make sure that the handling methods are short. In essence, the responsibility of our signal handler is merely to update a global variable that notifies another branch of code that a change has been made. Optimally we would block other signals after initiating a handler, but the overhead of doing this discouraged us from doing it.

```
1    unsigned int yPos;
2    uint16_t movementSpeed;
3    // Whether or not player has been given points for this marker
4    char isHitByPlayerOne;
5    char isNotActive; // Whether or not marker is on screen
```

Listing 1.1: Marker struct.

The basic logic behind the game implementation is shown in Figure 1.2.

[1] capturing SIGALRM and SIGIO, asynchronous events for timer expiration and file input/output

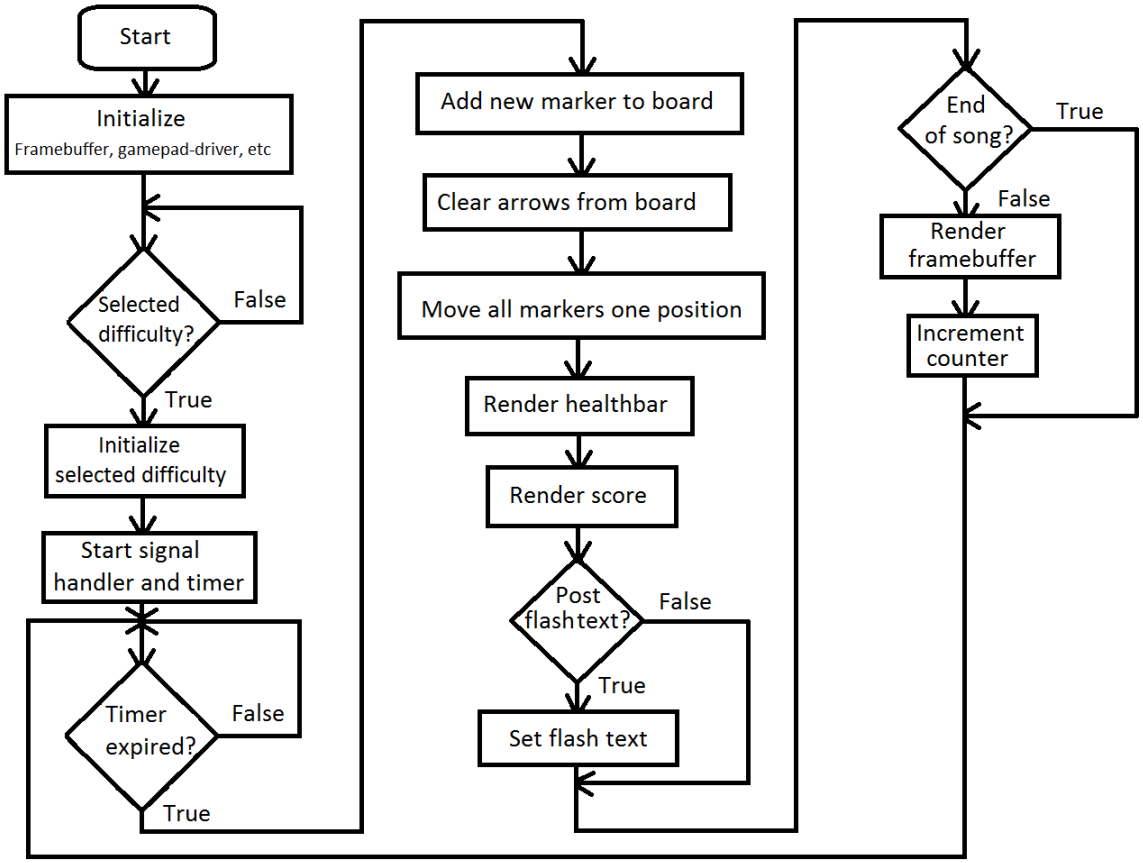

Figure 1.2: Simple flowchart of the game functions.

1.2 Gamepad Driver

The gamepad needs a device driver in order to communicate with the Linux kernel. This section will go over, step by step, how the driver was implemented and what choices were made during the implementation. There was used an old standard of implementation for the driver, so newer device drivers will be structured slightly differently.

1.2.1 What Should the Driver Do?

In Stepmania, the gamepad will be used for the arrows mentioned in the previous section, and for navigating in the game menu. So the driver should just make it possible to read the state of the gamepad-buttons. The user application should preferably not have to continuously poll the driver, so the driver should also implement interrupt-handling.

1.2.2 Driver Setup

Some setup-details must be covered before the driver can communicate with the user program. The type of driver used in this assignment is a character device driver, which is suitable for most simple hardware devices. Below is a listing with the definitions used throughout the code together with the global structures. The structures are also added with the code where they are used. See Listing 1.2.

```
1   // definitions
2   #define DRIVER_NAME "driver-gamepad"
3   #define IRQ_NUMBER_EVEN 17
4   #define IRQ_NUMBER_ODD 18
5   #define DATA_LENGTH 0x04
6
7   //structs and statics
8   struct cdev my_cdev = {
9   .owner = THIS_MODULE
10  };
11  struct class *cl;
12  struct fasync_struct* async_queue;
13  struct resource *gpioPc_memRegion;
14  static dev_t my_dev_t = 0;
```

Listing 1.2: Definitions and structures

The first step is to make the driver as a kernel module. Linux runs in two different modes: user mode and kernel mode. User programs run in user mode and has a limited set of permissions. The driver needs to run in kernel mode in order to have full access to everything, and in order to do this easily the driver will be implemented as a kernel module. The advantages of using a kernel module is that it can be loaded to the kernel dynamically, so that the kernel doesn't need to be recompiled every time the driver is changed. A kernel module needs to follow a set of limitations: 1. It must contain an init-function and an exit-function, and be told the names of these. 2. It can only use functions defined in the Linux Kernel. 3. It must be able to handle parallelism if multiple processes tries to use it. 4. Can't have loops running to eternity as this will cause the kernel to hang.

A "Hello World"-template with all the necessary code for a minimal kernel module was provided and we decided to start out with this. The template is shown in Listing 1.3

```
1   static int __init template_init(void) {
2   printk("Hello World, here is your module speaking");
3   }
4   static void __exit template_cleanup(void) {
5   printk("Short life for a small module..")
6   }
7   module_init(template_init);
8   module_exit(template_cleanup);
9   MODULE_DESCRIPTION("Device driver for gamepad.");
10  MODULE_LICENSE("GPL");
```

Listing 1.3: Kernel template

The second step of setup is asking for access to I/O hardware registers and memory map it. Asking for access is done to ensure that there are no other drivers accessing the same hardware. The EFM32GG has no virtual memory, but it is standard Linux practice to map it anyway[5]. See Listing 1.4

```
1     struct resource *gpioPc_memRegion;
2   void *gpioPc_base;
3
4   if ((gpioPc_memRegion = request_mem_region(GPIO_PC_BASE, GPIO_PC_LENGTH,
      DRIVER_NAME)) == NULL) {
5         printk("GPIO_PC_BASE allocation error");
6         return -1;
7     }
8     gpioPc_base = ioremap_nocache(gpioPc_memRegion->start, GPIO_PC_LENGTH);
```

Listing 1.4: Allocating and memory mapping I/O registers

Thirdly, the registers in the controller was set to enable the GPIO pins for input, enable internal pull-up and enable interrupts. More details on the use of interrupts are explained later. See Listing 1.5

```
1     // Setting up GPIO pins
2     iowrite32(0x33333333, gpioPc_base + MODEL_OFFSET);  // set pins C0-7 as
      input
3     iowrite32(0xff, gpioPc_base + DOUT_OFFSET);  // enabling internal pullup
4
5     // Enable interrupts in the controller
6     iowrite32(0x22222222, gpioPc_base + EXTIPSELL_OFFSET);
7     iowrite32(0xff,       gpioPc_base + EXTIFALL_OFFSET);
8     iowrite32(0x00ff,     gpioPc_base + IEN_OFFSET);
9     iowrite32(0xff,       gpioPc_base + IFC_OFFSET);
```

Listing 1.5: Setting register-values

Before the user program is given access to device, a device number must be allocated making the device identifiable. This can be done statically by hard-coding the device numbers, or dynamically, letting the kernel allocate a number for you. Here the dynamic solution is used. The device number consists of two numbers: a major and a minor number. The major number identifies what driver is being used, while the minor number identifies the device. See Listing 1.6

```
1     static dev_t my_dev_t = 0;
2
3     // Dynamically allocating the device numbers
4     result = alloc_chrdev_region(&my_dev_t, 0, 1, "devno");
5     if (result < 0) {
6         printk(KERN_ERR "chrdev alloc error");
7         unregister_chrdev_region(my_dev_t, 1);
8         return -1;
```

```
    }
```

Listing 1.6: Allocating device numbers

In order to give the user program access to the driver, some structures and functions must be initialized in the kernel. There are four functions necessary to operate the driver as a file. The write-function is not used in this driver as writing to the gamepad is not necessary (or possible). How the functions were implemented will be covered later in this section. See Listing 1.7

```
    static int my_open(struct inode *inode, struct file *filp) {
    return 0;
}
    static int my_release(struct inode *inode, struct file *filp) {
    return 0;
}
    static ssize_t my_read(struct file *filp, char __user *buff, size_t count
    , loff_t *offp) {
    return 1;
};
    static ssize_t my_write(struct file *filp, const char __user *buff,
    size_t count, loff_t *offp){
    return 1;
};
```

Listing 1.7: Functions for file-operations

After creating the functions they must be registered such that the kernel knows how they should be used. That is done in the file-operations structure in the following listing. In addition, a cdev-structure is created pointing to the file-operations structure. The cdev-structure is the internal kernel structure for representing character devices. The cdev-structure is then passed to the kernel, giving access to the file-operations. The fasync-operation is used later during interrupt-handling. See Listing 1.8

```
    struct cdev my_cdev = {
        .owner = THIS_MODULE
    };

    static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .read = my_read,
    .write = my_write,
    .open = my_open,
    .release = my_release,
    .fasync = my_fasync
    };

    cdev_init(&my_cdev, &my_fops);
    result = cdev_add(&my_cdev, my_dev_t, 1);
    if (result < 0) {
        printk(KERN_ERR "add cdev error");
        return -1;
    }
```

Listing 1.8: File-operation and cdev structures

The final step of the initialization is to create the virtual driver-file and placing it in the /dev/ folder in the root filesystem. By opening this file the user program can read from the gamepad through the driver using normal I/O operations. See Listing 1.9

```
    struct class *cl;

    // Creating the dev-file
    cl = class_create(THIS_MODULE, DRIVER_NAME);
    device_create(cl, NULL, my_dev_t, NULL, DRIVER_NAME);
```

Listing 1.9: Creating class and dev-file

In the cleanup-function all that has been set up has to be torn down. See Listing 1.10

```
static void __exit template_cleanup(void)
{
  // Releasing memory region and unmapping memory map
  release_mem_region(GPIO_PC_BASE, GPIO_PC_LENGTH);
  iounmap(gpioPc_base);

  // Deleting character device
    cdev_del(&my_cdev);

    // Removing device number and class
    device_destroy(cl, my_dev_t);
    class_destroy(cl);
    unregister_chrdev_region(my_dev_t, 1);
}
```

Listing 1.10: Cleanup-function

1.2.3 Functions for File Operations

The functions mentioned in the previous subsection needs to be defined so that they will work for the gamepad. The open function is called when the user program tries to open the device driver through the kernel. As most necessary operations are done in the init-function, the open-function can be left empty, but considering that there are a limited amount of interrupt-request (IRQ) channels and that interrupts are used only when the driver is open, the IRQ-channels are allocated here.[5] More detail on interrupt-handling are covered later. It was implemented as shown in Listing 1.11

```
static int my_open(struct inode *inode, struct file *filp) {
    // Initializing interrupt requests
    if (request_irq(IRQ_NUMBER_EVEN, (irq_handler_t) interruptHandler, 0,
    DRIVER_NAME, &my_cdev) != 0) {
        printk("Interrupt initialization error, EVEN");
    }
    if (request_irq(IRQ_NUMBER_ODD, (irq_handler_t) interruptHandler, 0,
    DRIVER_NAME, &my_cdev) != 0) {
```

```
7           printk("Interrupt initialization error, ODD");
8       }
9
10      return 0;
11  }
```

Listing 1.11: Open-function

The release function does the exact opposite of the open function, much like the cleanup function compared to the init function. This can also be left empty as all the necessary cleanup can be done in the cleanup function, but as the open function is not empty, the release function must deallocate the IRQ-channels. See Listing 1.12

```
1  static int my_release(struct inode *inode, struct file *filp) {
2
3      // Freeing IRQ
4      free_irq(IRQ_NUMBER_EVEN, &my_cdev);
5      free_irq(IRQ_NUMBER_ODD, &my_cdev);
6
7      return 0;
8  }
```

Listing 1.12: Release-function

The read function is the one implementing the core functionality of the driver, by actually reading the GPIO-pin register and making it available to the user program. See Listing 1.13

```
1  static int my_release(struct inode *inode, struct file *filp) static ssize_t
       my_read(struct file *filp, char __user *buff, size_t count, loff_t *offp)
        {
2      // Casts
3      uint32_t gpioPc_value = 0;
4      void *gpioPc_base;
5
6      // Create virtual memory
7      gpioPc_base = ioremap_nocache(gpioPc_memRegion->start, GPIO_PC_LENGTH);
8
9    // Read data from pin-register
10     uint32_t data = ioread32(gpioPc_base + DIN_OFFSET);
11     copy_to_user(buff, &data, DATA_LENGTH);
12     printk("Done reading!");
13     return 1;
14 };
```

Listing 1.13: Read-function

1.2.4 Interrupts

Full interrupt-handling is used in the device driver. This means that when a button is pushed the driver generates a signal and then sends it to the user program. The user program then fetches the current value of the GPIO-pins from the driver. This lets the

processor focus on running the game without having to continuously poll the driver. The drawback is that it is a bit harder to implement. First of all, the driver and the controller must be initialized for interrupt handling. The controller is initialized by setting the registers as shown in Listing 1.5. The driver is initialized by calling the functions shown in Listing 1.11. These functions register the given interrupt-handler function, shown in Listing 1.14, and assign it to the IRQ-channels given. Here the IRQ-channel numbers are 17 for the even GPIO-pins and 18 for the odd GPIO-pins.

```
static irq_handler_t interruptHandler(int irq, void *dev_id, struct pt_regs *
    regs) {
    uint32_t gpioPc_value = 0;
    void *gpioPc_base;

    // Create virtual memory
    gpioPc_base = ioremap_nocache(gpioPc_memRegion->start, GPIO_PC_LENGTH);

  // read interrupt flag
    gpioPc_value = ioread32(gpioPc_base + IF_OFFSET);

  // Clear interrupt pending bit
    iowrite32(gpioPc_value, gpioPc_base + IFC_OFFSET);

  // send a signal to the user program
    if (async_queue) {
        kill_fasync(&async_queue, SIGIO, POLL_IN);
    }

  // Release virtual memory
    iounmap(gpioPc_base);

    return (irq_handler_t) IRQ_HANDLED;
}
```

Listing 1.14: Interrupt-handler function

When the driver is ready for interrupt-handling it needs to implement asynchronous signaling. To get this to work the user program needs some code saying that it should read from the driver when it receives a signal, but here the focus will be on the driver-side. Firstly, the driver needs a structure to keep track of the asynchronous readers connected to it, and a fasync-helper function is defined for adding and removing files to this structure[5]. See Listing 1.15. The fasync-helper function is added to the file-operations structure shown in Listing 1.8.

```
struct fasync_struct* async_queue;

static int my_fasync(int fd, struct file* filp, int mode) {
    return fasync_helper(fd, filp, mode, &async_queue);
}
```

Listing 1.15: Fasync-struct and -function

When this structure is in place, all that is left is sending a signal to the program waiting in the asynq_queue when a button is pressed. This is done by the kill_fasync function shown in Listing 1.14.

1.3 Graphics

This section is about how the graphics were handled on the system which includes how the framebuffer was used, how the bitmap file type was implemented, and how the game graphics were rendered. Important code snippets pertaining to each section is shown and explained in each section.

1.3.1 Framebuffer Driver

The framebuffer driver is responsible for displaying text and graphics on the LCD screen of the development kit. The framebuffer is a portion of RAM which contains a table of values where each element contains a value which corresponds to the color of a single pixel. For instance, if the first element of the framebuffer is 0xFFE0, and the bit-depth is 16-bits, this corresponds to a RGB565 color value which would paint the first pixel of the LCD yellow. In the same way, 0xFFFF would be a white pixel and 0x0000 would be a black pixel.

Several global structs are defined in order to retrieve information about the screen which are then used in other functions in the program, as is shown in Listing 1.16. The first and foremost is the mmapFrameBuffer pointer, which will later point to the array stored in memory which is mapped to the framebuffer driver. The second is `fb_fd`, which is used to open the framebuffer as a "file". The variable finfo and vinfo is the fixed and variable information about the screen used. These contain information about the screen size, bits per pixel, and so forth. The last is the rect variable, which is used to update parts of the framebuffer at a time.

```
1    uint8_t *mmapFrameBuffer;
2    static int fb_fd = -1;
3    struct fb_fix_screeninfo finfo; //Contains information about the screen
4    struct fb_var_screeninfo vinfo;
5    struct fb_copyarea rect;  //Contains the part of the screen to update
```

Listing 1.16: Some of the global variables.

The framebuffer is initialized by calling the initFrameBuffer function. The first thing the function does is to disable the terminal cursor function in Linux, which would otherwise have been a small blinking black box somewhere on the screen. This is done by running a special echo argument as shown in Listing 1.17.

```
1    //Disable cursor in LCD screen
2  system("echo -e -n '\033[?25l' > /dev/tty0");
```

Listing 1.17: Disabling the cursor.

The next thing the function does is to open the framebuffer device as a file, and then using the ioctl[2] function to read the fixed screen info and variable screen info from the

[2] InputOutputCoNTrol

device, as shown in Listing 1.18.

```
    //Open the framebuffer "file" in read/write mode
  fb_fd = open("/dev/fb0",O_RDWR);

  //Retrieve the fixed screen information
  temp = ioctl(fb_fd, FBIOGET_FSCREENINFO, &finfo);

  //Retrieve the variable screen information
  temp = ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);
```

Listing 1.18: Opening and reading the framebuffer device.

The final step of the function is to map the framebuffer to an array in memory. This makes it much simpler to write to the screen since you deal with a simple array instead of using the lseek and write functions for finding and writing to each pixel. As shown in Listing 1.19, the memory size of the screen is read from the fixed screen info, and then used when mapping the driver.

```
    FrameBufferSize = finfo.smem_len;

    //mmap framebuffer
    mmapFrameBuffer = mmap(NULL, FrameBufferSize, PROT_READ | PROT_WRITE,
          MAP_SHARED, fb_fd, (off_t) 0);
```

Listing 1.19: Using mmap on the framebuffer.

1.3.2 Bitmap File Support

It was decided early on that in order to make the relatively complex graphics required to implement a game like Stepmania, we would need a way to create images beforehand and then display them on the screen. One of the easiest image types to implement is the bitmap image file. This is because a bitmap file is almost just a 2D array containing pixel color information and can be read and used directly, instead of image types like JPEG which needs to be decompressed before it can be used. The downside is that bitmap images take up more space, which is rather limited on the microcontroller. The bitmap image files are included in the build by adding them to the game.make file in /rules.

The bitmap file consists of two headers and a pixel map. The headers contain information about the bitmap file itself as well as information about the pixel map. The structure of the headers and pixel map is shown in Figure 1.3 - 1.5

In order to use bitmaps with the system, several functions were created. The first is the LoadBMPToMemory function, which open the bitmap file, reads information from the header files, and then creates a pointer which points to the first element of the pixel map. The first part of the function simply initializes the pointer used to open the bitmap and the pointer array in which the pixel map will be stored, as well as open the

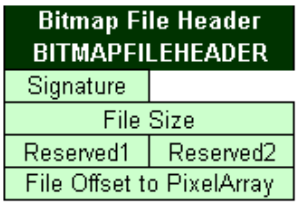

Figure 1.3: The file header of a bitmap file.

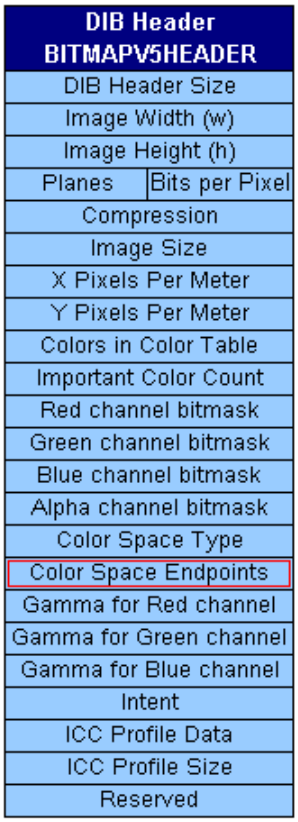

Figure 1.4: The info header of a bitmap file.

bitmap file with the fopen function, as shown in Listing 1.20.

```
1     FILE *FilePtr;
2   unsigned char *bitmapImage;
3
4   //Open the BMP file
5   FilePtr = fopen(FileName,"rb");
```

Listing 1.20: Initializers in the bitmap memory function.

Image Data PixelArray [x,y]					
Pixel[0,h-1]	Pixel[1,h-1]	Pixel[2,h-1]	...	Pixel[w-1,h-1]	Padding
Pixel[0,h-2]	Pixel[1,h-2]	Pixel[2,h-2]	...	Pixel[w-1,h-2]	Padding
			⋮		
Pixel[0,9]	Pixel[1,9]	Pixel[2,9]	...	Pixel[w-1,9]	Padding
Pixel[0,8]	Pixel[1,8]	Pixel[2,8]	...	Pixel[w-1,8]	Padding
Pixel[0,7]	Pixel[1,7]	Pixel[2,7]	...	Pixel[w-1,7]	Padding
Pixel[0,6]	Pixel[1,6]	Pixel[2,6]	...	Pixel[w-1,6]	Padding
Pixel[0,5]	Pixel[1,5]	Pixel[2,5]	...	Pixel[w-1,5]	Padding
Pixel[0,4]	Pixel[1,4]	Pixel[2,4]	...	Pixel[w-1,4]	Padding
Pixel[0,3]	Pixel[1,3]	Pixel[2,3]	...	Pixel[w-1,3]	Padding
Pixel[0,2]	Pixel[1,2]	Pixel[2,2]	...	Pixel[w-1,2]	Padding
Pixel[0,1]	Pixel[1,1]	Pixel[2,1]	...	Pixel[w-1,1]	Padding
Pixel[0,0]	Pixel[1,0]	Pixel[2,0]	...	Pixel[w-1,0]	Padding

Figure 1.5: The pixel map of a bitmap file.

The next thing the function does is to read the header information from the bitmap file, which tells us how big the pixel map is. A malloc call is then made which allocates enough memory so that we can store the entire pixel map in a char pointer. The code for this is shown in Listing 1.21.

```
1     //Read the BMP file header
2     fread(bitmapFileHeader, sizeof(BITMAPFILEHEADER),1,FilePtr);
3
4   //read the BMP info header
5   fread(bitmapInfoHeader, sizeof(BITMAPINFOHEADER),1,FilePtr);
6
7   bitmapImage = (unsigned char*)malloc(bitmapInfoHeader->biSizeImage);
```

Listing 1.21: Opening and memory allocation for the bitmap file.

Finally, a fseek is used, which places the FilePtr pointer at the beginning of the pixel map. Then a fread is used, which reads and stores the pixel map and makes the pointer bitmapImage point to it, as is shown in Listing 1.22. Finally the FilePrt is closed and the function returns the bitmapImage.

```
1     fseek(FilePtr, bitmapFileHeader->bfOffBits, SEEK_SET);
2
3   //Load the BMP image data
4   fread(bitmapImage,bitmapInfoHeader->biSizeImage,1,FilePtr);
5
6   fclose(FilePtr);
```

Listing 1.22: Reading and storing the pixel map.

We now have the pixel map from the bitmap image file loaded into the memory of the device. The next task is to put the content of the pixel map into the framebuffer. This is done using the LoadBMPToFrameBuffer function, which is partially shown in Listing 1.23. Several arguments can be passed to the function such as rotation (90,180,270 degrees), type and if the image should be loaded at a certain offset in the framebuffer array. If the type is set to 1, then the function ignores pixels which are either 0x0000 or 0xFFFF, which corresponds to completely white or completely black. This makes it possible to render the bitmap files almost as png files, because the white background in each image isn't rendered. This means that nothing you want rendered on the screen can be completely white when setting the type to 1, but must be slightly grayer. If the type is anything but 1, the entire image including white and black is loaded normally. There is also another function called LoadPartialBMPToFrameBuffer, which is similar to the LoadBMPToFrameBuffer function, but differs in that it can be used to only load small parts of the pixel map instead of the whole image. The Pixelcolor function simply shifts the value read from the array in order to retrieve the RGB565 formatted pixel.

```
1  int *LoadBMPToFrameBuffer(unsigned char *bitmapData, int rotation, int type,
       int xoffset, int yoffset)
2  {
3      padding = (BMPlength*6)%4;
4
5    for(x = 0; x < BMPheight; x++) {
6      for(y = 0; y < BMPlength; y++) {
7        long location = (y+xoffset) * (vinfo.bits_per_pixel/8) +
8                        (x+yoffset) * finfo.line_length;
9        int position;
10
11       switch(rotation) {
12           //90 degrees
13         case 1: position = BMPlength*y*2+x*2+y*padding;
14         break;
15         //0 degrees
16         case 2: position = (BMPheight-x)*(BMPlength)*2+y*2-x*padding;
17         break;
18         //180 degrees
19         case 3: position = BMPlength*x*2+y*2+x*padding;
20         break;
21         //270 degrees
22         case 4: position = (BMPheight-y)*(BMPlength)*2+x*2-y*padding;
23         break;
24         //0 degrees
25         default: position = (BMPheight-x)*(BMPlength)*2+y*2-x*padding;
26         break;
27       }
28
29       if(type == 1) {
30         if((PixelColor(bitmapData[0+position],bitmapData[1+position]) < 0
     xFEFE) && (PixelColor(bitmapData[0+position],bitmapData[1+position]) > 0
     x0000)) {
```

```
31            *((uint32_t*)(mmapFrameBuffer + location)) =  PixelColor(bitmapData
    [0+position],bitmapData[1+position]);
32          }
33        }
34        else {
35          *((uint32_t*)(mmapFrameBuffer + location)) =  PixelColor(bitmapData
    [0+position],bitmapData[1+position]);
36        }
37      }
38    }
39
40    return 0;
41  }
```

Listing 1.23: Parts of the LoadBMPToFrameBuffer function.

After the whole or parts of the pixel map is loaded to the framebuffer, it has to be updated. This is done with the UpdateFrameBuffer function, which is shown in Listing 1.24. The function is quite simple. It defines a rectangle, and then updates that rectangle in the framebuffer using the ioctl function.

```
1       int yr,xr;
2       int xupdate = 2;
3       int yupdate = 240;
4       rect.width = xupdate;
5       rect.height = yupdate;
6       for(yr = 0 ; yr < (vinfo.yres/yupdate) ; yr++) {
7         rect.dy = (yr*yupdate);
8         for(xr = 0; xr < (vinfo.xres/xupdate) ; xr ++) {
9           rect.dx = xr*xupdate;
10          ioctl(fb_fd, 0x4680, &rect );
11
12        }
13      }
```

Listing 1.24: Updating the framebuffer .

There is also a function similar to the UpdateFrameBuffer function called UpdateFrameBufferPosition, which only updates a small part of the framebuffer depending on the arguments passed to it. By utilizing the functions described above, bitmap images can be loaded, manipulated and positioned on the LCD screen. Because the images are relatively large, it is important that strict memory control is implemented, and the memory is freed using the free function and the pointer nulled after each image has been loaded to the framebuffer. This is to prevent the device from running out of RAM. The next section describes how the game graphics itself was implemented.

1.3.3 Game Graphics

Now that bitmap images can be used, making good looking graphics becomes much easier. Several bitmap images are used in the project, using a total storage space of

around 600-700kB. One of the most important bitmaps are the GameOverlay.bmp, which is shown in Figure 1.6. That bitmap serves as the basis for the rest of the game graphics, which are simply placed on top of the GameOverlay image.

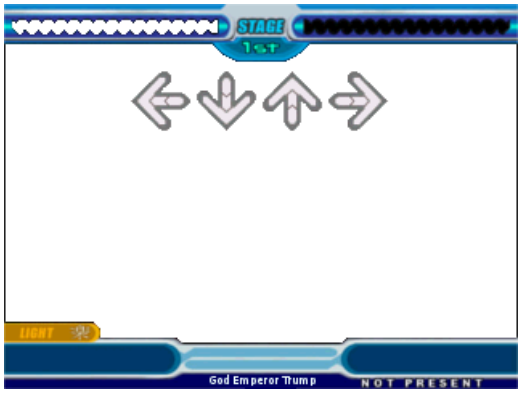

Figure 1.6: The game overlay for Stepmania.

In the early version of the game, a background image was rendered first, and afterwards the GameOverlay.bmp was rendered on top of the background. This is why most of the GameOverlay file is white, because you can make the program ignore the white parts of the image, letting the background show through. The original background image is shown in Figure 1.7. By having a seperate background image, we could easily change the background image, even while playing the game, without having to change the GameOverlay.

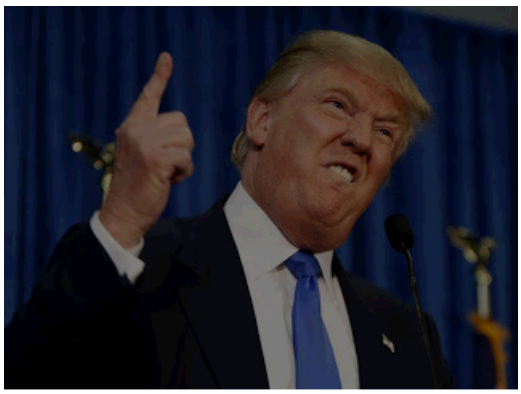

Figure 1.7: A random image serving as a background for the game.[6]

The downside of having a separate background and game overlay image is that both take the same amount of space on the disk, which is around 150kB. Therefore, instead of using over 300kB of our relatively limited memory, it was decided to combine both the background image and overlay image into a single image, as shown in Figure 1.8.

This takes away the ability to swap backgrounds, but saves around 150kB of space.

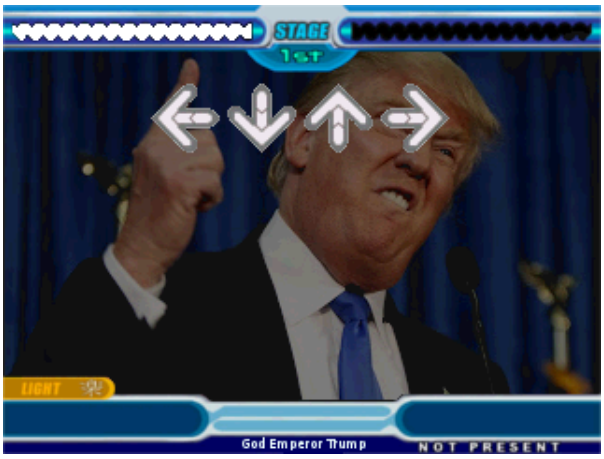

Figure 1.8: The game overlay rendered on top of the background image.

In order to make the game consume as little processing power as possible while playing the game, and in order to make the game more efficient, only parts of the screen is continuously updated while the game is playing. At the beginning of the game, the GameOverlay image is loaded, and the whole framebuffer is updated. Afterwards, only selected regions are updated using specific functions, as shown in Figure 1.9.

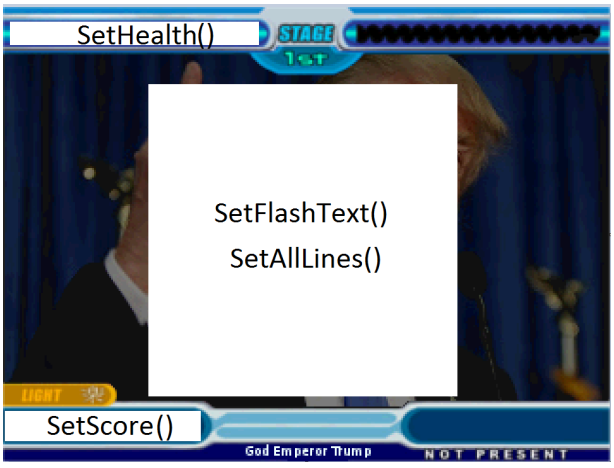

Figure 1.9: The parts of the screen updated while playing.

The renderHealth function, partly shown in Listing 1.25, is a relatively simple function. It takes a number between 0-129 as an argument, where 129 if full health and 0 is no health. If the health is over 90, it renders the health bar as green. If it is below 90 it

is rendered as yellow, and below 40 it is rendered as red. The bar it self is rendered using the DrawPrimitive function, which simply draws a solid colored block of a certain size at a chosen location on the screen. The part of the health bar not rendered green, yellow or red is rendered black. Part of the GameOverlay is then rendered on top of the block, which gives the look as shown in Figure 1.10.

```
1       int color = GREEN;
2
3       if(health<90) {color = YELLOW;}
4       if(health<40) {color = RED;}
5
6       DrawPrimitive(color, health, 10, 4, 11);
7       DrawPrimitive(0x0000, 129-health, 10, health+4, 11);
8
9       bitmapdata = LoadBMPToMemory(IMAGE_FOLDER "GameOverlay.bmp");
10
11      LoadPartialBMPToFrameBuffer(bitmapdata,0, 3, 9, 131, 16);
12
13      UpdateFrameBufferPosition(3, 9, 131, 14);
```

Listing 1.25: Parts of the renderHealth function.

The UpdateFrameBufferPosition and LoadPartialBMPToFrameBuffer functions ensures that only the small region used by the health bar is affected when the renderHealth function is called. The numbers passed as arguments to those functions might seem arbitrary, but they ensure that the right part of the screen is rendered. As mentioned above, this ensures that as little processing power as possible is used to perform this action.

Figure 1.10: The health bar rendered in game.

Similarly to the renderHealth function, the renderScore function only updates a small part of the screen. The renderScore function picks one of the 10 images shown in Figure 1.11, which each represents a number between 0-9. The image is then rendered to the correct position on the screen.

Figure 1.11: The numbers used to print the score.

Parts of the code for the renderScore function is shown in Listing 1.26. It takes a number as an argument, and renders that number on the screen. The function getDigit simply returns the digit in a specific location in a number. For instance, digit 3 in 24601 would return 6. Since each bitmap image which represents a number is stored

as 1.bmp, 2.bmp, etc, it is easy to simply manipulate a file string which points to the correct image. The numbers passed in the LoadBMPToFrameBuffer and UpdateFrameBufferPosition again refers to specific position on the display.

```
1  int number;
2  unsigned char *bitmapdata;
3  char filestring[35] = IMAGE_FOLDER "1.bmp";
4
5  for(number = 0; number < 8; number++) {
6    filestring[29] = (char)('0'+getDigit(score,number));
7    bitmapdata = loadBMPToMemory(filestring);
8
9    loadBMPToFrameBuffer(bitmapdata, 0, 1, 5+(8-
     number)*10-number*2, 215);
10
11   free(bitmapdata);
12   bitmapdata = NULL;
13 }
```

Listing 1.26: Parts of the renderScore function.

The SetFlashText function takes a number between 0 and 3 as an argument and prints one of the texts shown in Figure 1.12. Each of these are presented depending on when the player pressed the keypad in relation to when the arrow was over that specific key. For instance, if the player presses the gamepad key when the arrow is covering 70% of the grey arrows shown in GameOverlay.bmp, then the player might be presented with the text "Great". If the arrow is covering 20%, the player might be presented with "Boo", and so forth.

Figure 1.12: The different flash texts used in the game.

Parts of the SetFlashText function is shown in Listing 1.27. It is a very simple func-

tion that opens the correct bitmap file corresponding to the text the game wants to display. It then updates the framebuffer with the image. Again, the arguments passed to the LoadBMPToFrameBuffer function ensures that the text is placed almost in the center of the screen and slightly shifted upwards. Here, information about the size of each image is read from the header files of each bitmap.

```
    switch(type) {

        case MISS: bitmapdata = loadBMPToMemory(IMAGE_FOLDER "Miss.bmp");
        break;

        case BOO: bitmapdata = loadBMPToMemory(IMAGE_FOLDER "Boo.bmp");
        break;

        case GREAT: bitmapdata = loadBMPToMemory(IMAGE_FOLDER "Great.bmp");
        break;

        case PERFECT: bitmapdata = loadBMPToMemory(IMAGE_FOLDER "Perfect.bmp"
    );
        break;

        default: bitmapdata = loadBMPToMemory(IMAGE_FOLDER "Miss.bmp");
        break;
    }
    if(!bitmapdata)
    {
        fprintf(stderr, "text\n");
        clearMemory();
        exit(1);
    }
    loadBMPToFrameBuffer(bitmapdata,  0, 1, (vinfo.xres)/2 - (
    bitmapInfoHeader->biWidth)/2 , (vinfo.yres)/2-35);
```

Listing 1.27: Parts of the SetFlashText function.

The final, and perhaps most important function, is the moveMarker function. This function is responsible for rendering the moving arrows on the screen. There are four different colored arrows in the game, as shown in Figure 1.13. However, only the red arrow is currently used in the game. Its code is shown in Listing 1.28.

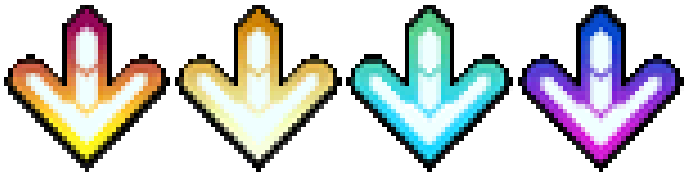

Figure 1.13: The different arrows used in the game.

```
bitmapData = LoadBMPToMemory(IMAGE_FOLDER "GameOverlay.bmp");
LoadPartialBMPToFrameBuffer(bitmapData,
                            0, 1, 79, 35, (37+4)*4, 175);

  // ...

// for each marker
{
  if (markerColumns[i].isNotActive)
              continue;

      markerColumns[i].yPos -= markerColumns[i].movementSpeed;

      yPos = markerColumns[i].yPos;
      if (yPos >= ARROW_CEILING && yPos <= ARROW_STARTING_YPOS)
      {
          loadBMPToFrameBuffer(bitmapData, rotation, 1, COLUMN_POSITION +
    (ARROW_PADDING * c), markerColumns[i].yPos);
      }
      else // perform game-logic if out-of-screen
          // ...
}
```

Listing 1.28: Parts of the moveMarker function.

1.4 Coding Convention

Our code is linted using GNU Indent with a four space indent and Linux coding style (applied using `make pretty`). For the game, we separate header and source files in folders "include" and "src" respectively. Documentation is put in the source files, not the headers. The driver is relatively small so we keep it to a single source file.

1.5 Energy-saving Techniques

1.5.1 Code Optimizations

Wherever we have methods that acts as a subroutine, we have tried to inline the code to avoid overhead in method invocation. In our last report (exercise 2) we showed that this can reduce overall power consumption. Aggressive inlining may also be counter-productive, so we have made sure to carefully select our methods. Furthermore, we used one-dimensional rather than two-dimensional arrays. Other than that, we rely on the compiler (arm-cortexm3-uclinuxeabi-gcc version 4.7.2) to statically analyze and optimize our code, using the -O2 flag.

1.5.2 Compiling Kernel

Our code runs inside a BusyBox[1] environment, with Linux kernel version 3.12 and the uClinux operating system[4]. We use utilities from the BusyBox environment to load our kernel modules and to start/end the game. We have applied customizations to the default configurations of the kernel and BusyBox. This results in reduced Linux image sizes from 2.1 MB to 0.9 MB.

We have also enabled tickless idle[7] for our kernel. This enables the CPU, when idle, to stop polling on timer clocks, and rather wait for the next scheduled event. This can save power for applications where there is more frequent idling, few interrupts and timers. The impact is discussed in Chapter 2.

1.5.3 Reducing CPU Load

Several techniques were used to reduce the overall CPU load, and by extension lower the energy consumption somewhat. One of the methods were to only render the parts of the screen where changes were occurring. This makes the game faster, and the CPU therefore works less in a given period of time.

2 Energy Measurements

Unconfigured BusyBox/OS idle	around 10 mA
BusyBox/OS idle	around 10 mA
BusyBox/OS idle, driver loaded	around 10 mA
Game loaded, waiting for song selection	10 mA
Game running, full difficulty	33 mA

Table 2.1: Power usage

As can be seen in Table 2.1, enabling our custom kernel does not improve energy efficiency by a lot, even in idle. Regarding tickless idling, this indicates that even though we use a limited set of programs for our development-board, there are still to many interrupts and timer events for our system to enter longer idle states. This could be because of the relatively slow clock frequency of our CPU, not being able to process events fast enough for a "race-to-sleep". This is similar to the results in [2].

Also, it can be observed that loading the driver does not affect power usage much, i.e. the asynchronous polling does indeed let the CPU do other activities.

As can be seen from Figure 2.1, the power consumption when the game was running were about 33mA. This was when using the unchanged kernel.

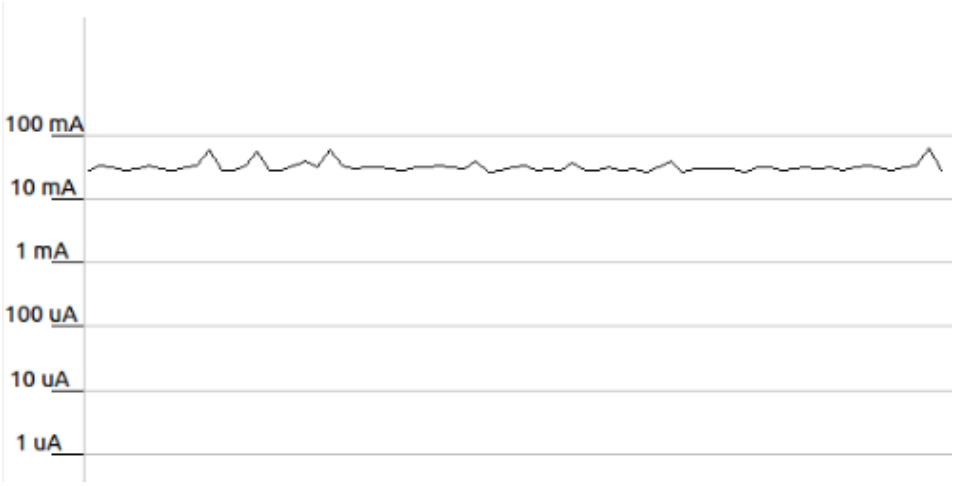

Figure 2.1: Power usage during game, using unconfigured kernel

Figure 2.2 shows the game running with a configured kernel. As can be seen, there

isn't that much difference between the configured and unconfigured kernel.

Figure 2.2: Power usage during game, from start

2.1 Discussion

The game we decided to implement does not allow for many opportunities for the CPU to sleep. When the game is playing, information is constantly presented on the screen and processing is being done. In order to save power when processing, we have limited the parts of the screen which is updated to small regions and made those regions as small as possible. As said before, this also makes the game faster because each frame rendered takes less time to render, which also reduces the overall power consumption.

It might have been possible to reduce the energy consumption somewhat by further tinkering with the kernel. However, even after removing several unused functions and enabling tickless-idle, the total energy consumption remained relatively unchanged. Also, some functions could have been optimized somewhat in order to make them faster and more efficient.

2.2 Conclusion

Our game works and is relatively bug-free, apart from some small rendering issues. While the energy consumption when the game was running were relatively high, it is to be expected when running a game which requires a lot of processing and graphical rendering.

Figure 2.3 shows the game running on the development board. Apart from some small bugs and some issues with rendering the graphics, the game worked as expected

and was fully playable.

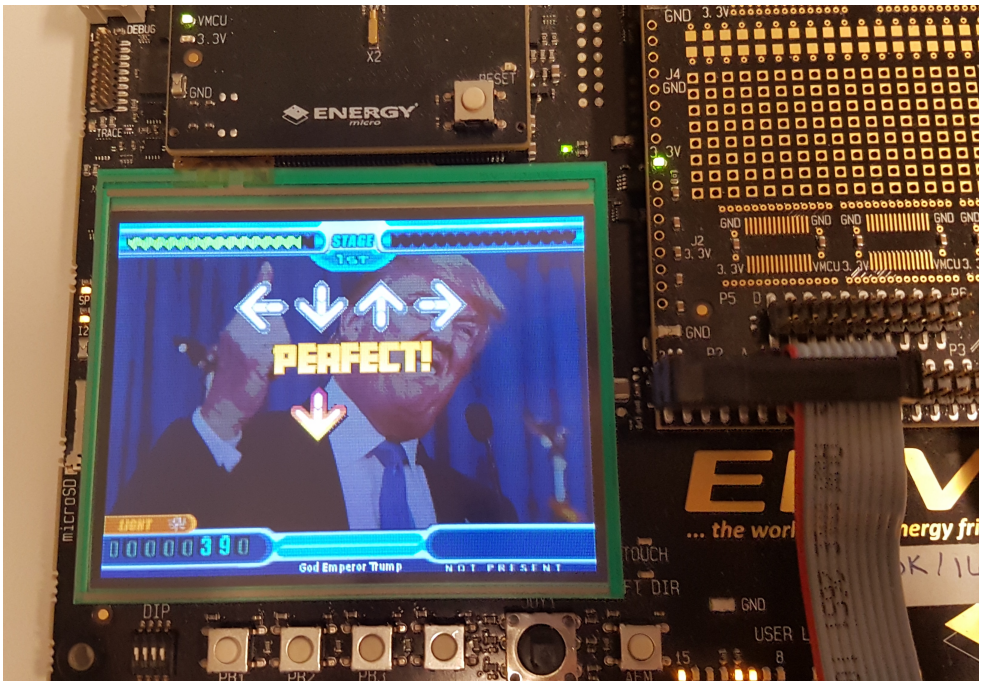

Figure 2.3: The game running on the dev board.

Bibliography

[1] Busybox. https://busybox.net/about.html. (Accessed on 11/18/2016).

[2] The impact of a tickless kernel - phoronix. http://www.phoronix.com/scan.php?page=article&item=651&num=1. (Accessed on 11/18/2016).

[3] signal(7) - linux manual page. http://man7.org/linux/man-pages/man7/signal.7.html. (Accessed on 11/18/2016).

[4] uclinux™ – embedded linux microcontroller project – home page. http://www.uclinux.org/. (Accessed on 11/18/2016).

[5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[6] Reuters/Dominick Reuter. Donald trump. http://www.businessinsider.com/nbc-donald-trump-2015-6?r=US&IR=T&IR=T, Jun 2015. (Accessed on 11/08/2016).

[7] Suresh Siddha, Venkatesh Pallipadi, and AVD Ven. Getting maximum mileage out of tickless. In *Proceedings of the Linux Symposium*, volume 2, pages 201–207. Citeseer, 2007.