NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 3

Group 4:

Sigve S. Nordgaard
Thomas Aven
Lasse Eggen

November 20, 2017

1 Overview

In solving this exercise we have written a device driver for the gamepad, which purpose is to communicate buttons pressed to the already existing framebuffer driver. In addition, we have written a wrapper around the provided framebuffer driver for the board's LCD screen, as well as an attempt at tackling sound with a DMA. To put this to the test, we have written a clone of the original Tetris for the Game Boy.

1.1 Project structure, Makefile, code style

Our project is divided into three parts: driver-gamepad-1.0, driver-sound-1.0, game-1.0. All code that resides in user space is written in game-1.0, while everyting kernel related resides in the drivers. We have organized the code in the most conventional way, by having all {c,h} files at the root of the source code tree:

driver-gamepad-1.0

- driver-gamepad.c: We decided to keep the driver as one monolithic file. The driver takes care of initializing a character device and passing it to the kernel, having been allocated the correct sections of IO-memory by the kernel.

driver-sound-1.0

- driver-sound.c: Again just one monolithic file. The driver takes care of all of DMA, PRS, Timer3 and DAC0, which should have been refactored out into their own drivers had we had the time. The driver takes care of initializing the DMA to give new samples to the DAC at the accordance of the PRS-system within the board.

game-1.0

- framebuffer.{c,h}: Wraps /dev/fb0 with mmap to provide a simple interface to blit regions of the LCD screen.
- game.c: Seeds our RNG and registers externals like the screen, gamepad and SIGIO. Ticks tetris in its main loop, but sleeps almost all of the time.
- gamepad.{c,h}: Initializes the gamepad through /dev/gamepad, and then takes a function pointer (in our case to our Tetris handler) that should handle logic around the state of the gamepad at async interrupt time.

- signal.{c,h}: Registers SIGIO to an async action it is given from a void pointer.
- tetris.{c,h}: The majority of our functionality. Contains functionality to blit the screen in tetris-specific ways, do tetromino manipulation, and board manipulation in general.
- util.{c,h}: Contains a function to convert RGB888 to RGB565, a function to convert a number to a string of decimals, and our own wrapper around nanosleep to be able to sleep for the full duration of our call, even if we are interrupted asynchronously by a signal.

For our code style we have once again used the Linux Kernel coding style throughout all of our project (by the use of pretty). For a full set of rules, refer to [2]. Outside this, our only exception is again the comment style. We also have a strict no-warning policy; any warning by the compiler should be taken seriously into consideration and not suppressed. In this exercise we also included header guards (in the form of #pragma once). The only way our Makefiles have been extended, are by adding phonies for prettify, and adding to the OBJS rule of the game, to link in our other modules.

1.2 Gamepad Driver

A device driver written as a kernel module can expose itself as a regular file to the rest of the OS, typically in the `/dev/` directory. We have written a character driver which is suitable for a simple device like the gamepad. The driver is built around the skeleton code provided for the exercise and is located in the driver-gamepad-1.0 which contains the driver code in driver-gamepad.c and the provided Makefile. We noticed too late, however, that the platform would be more suited for a platform driver, as it was hidden away in the appendix, and thus decided to spend more time on the sound driver than converting what we already had. The only addition to the Makefile is a make pretty target with an including .indent.pro file.

1.2.1 Basic Setup

Starting off we created our own device-specific structure, *gp_dev*, to mimic scull. In this structure lies the device number, using the dev_t type, split in a major and a minor part. It is possible to define whether to use *alloc_chrdev* or *register_chrdev* by specifying a major number explicitly, but that is a waste of memory here (and we don't need to use MKDEV MAJOR or MINOR as we are not concerned with loading any other modules at all). We therefore dynamically allocate character device structures as recommended in LDD by calling the function *alloc_chrdev_region*. [1]

Next we register the file functions that make it possible for the user program to handle the driver as a file. The gamepad-driver implements only the necessary device methods and initializes the methods by using the file operations structure created as *gp_fops*. This structure points to the functions *gamepad_read*, *gamepad_open*, *gamepad_release* and *gamepad_fasync*. While gamepad_open is a dummy function, since no special operations are needed for opening the gamepad, the functionality of the other two functions are described in the following sections.

Lastly in the basic initialization process we have to allocate and initialize the *cdev* structure. This is the kernels internal structure that represents character devices. We allocate and register this structure by calling the *cdev_init* function with a pointer to the file operations structure, and at the same time passing a cdev pointer to our gp_dev struct.

After handling other setup that needs to be done, described below, we add the cdev structure to the kernel with *cdev_add*, the function that tells the kernel about the cdev structure. Reqeuired arguments is DEV_NR_COUNT, set to 1 and passed as the count argument - representing the number of device numbers that should be associated with the device, and the device number created in alloc_chrdev_region. This call is done after setting up I/O and allocating memory since the call adds the device to the kernel and should not be done before the device is ready.

To make the driver appear as a file in the /dev directory we at last create a class by using the *device_create* function, given a pointer to the owner in our fops structure, and use this class to create the device itself using *device_create*.

1.2.2 I/O Memory Allocation and Mapping

I/O memory regions must be allocated prior to use. This is done in the driver using the *request_mem_region* function. The memory regions we need to allocate are the regions used for general purpose I/O, GPIO_PC, needed to access the buttons, and the interrupt request registers GPIO_IRQ. The length of the region we allocate corresponds to the number of 4 byte registers we need access to. This is respectively 9 and 8 for the GPIO_PC and GPIO_IRQ registers.

Subsequently we have to ensure that the allocated I/O memory is accessible to the kernel. This is done by setting up a mapping with the *ioremap* function, designed to assign virtual addresses to I/O memory regions. Since we are working on hardware we specifically used the *ioremap_nocache* function to ensure uncached access to memory in the mapped region. We store the address given from ioremap in our gp_dev struct so that we can write to these memory regions using the *ioread* and *iowrite* functions recommended by LDD to use when writing and reading to and from I/O memory. [1] These functions are used in the subsequent sections to set up the buttons and enable the interrupts on the EFM32 board as we have done in the two previous exercises by setting drive strength, input mode and internal pull-up for the buttons, and setting the EXTIPSELL and EXTIFALL flags for the interrupts

1.2.3 Interrupt handling

Now we are finally ready to add the device to the kernel as described in the last portion of the basic setup section. Following this is the last step of the __init function: setting up the interrupt handler. This is done using the *request_irq* function to request that the handler function, *gpio_irq_handler*, be called whenever the kernel receives a given interrupt. We need to allocate one interrupt line for the even and odd GPIO interrupts respectively. Finally the initiation is completed by activating the interrupts, writing $0xFF$ to the GPIO_IRQ_IEN register.

Interrupts are implemented in the driver with an asynchronous queue, *fasync_struct*. When the FASYNC flag of the file is set, the function that .fasync in the file operations structure points to will be called. This is the *gamepad_fasync* method which in turn calls *fasync_helper*, the method that adds the device to the list of interested processes. Then, when the interrupt handler is called, the *kill_fasync* function is used to signal interested processes by sending a SIGIO signal to the POLL_IN band. On closing the device driver fasync_helper is used again to remove the file from the list of active asynchronous readers. This is done when calling *gamepad_release*.

The interrupt handler also checks the Interrupt Flag, GPIO_IRQ_IF, state and writes this to the Interrupt Flag Clear register, GPIO_IRQ_IFC, to clear the read interrupt. A final peculiarity of the interrupt request handler is that we want to ignore the first interrupt being sent. This is because the EFM32GG seems to always send an extra interrupt when interrupts are set up. We ignore this as pushing it to the asynchronous queue will cause us to be one message behind at all times.

1.2.4 Reading

.read in our file operations structure points to *gamepad_read*, which handles reading and is the method we use to access the gamepad from our game. It uses the I/O memory allocated by request_mem_region to read from the GPIO_PC_DIN register. Because active high is easier to work with, we then invert the bits read before using *copy_to_user* to copy the button state from the kernel space to the user space. If someone tries to read more than the one byte we read from the data input register, we return a 0 - marking end-of-file.

1.2.5 Exiting and error handling

Exiting and error handling is handled as suggested in LDD. [1] In the __init function, if at any time an allocation or kernel command fails, every change made on the memory or kernel up at that point will be reverted by using goto-statements to fall through all required cleaning up. This includes everything discussed in the previous sections: allocating chrdev and memory regions for the general purpose I/O and the interrupt request handler, failing to add the chrdev to the kernel, and failing to create and add the driver class. All these operations are also called when exiting the the module, calling our __exit function.

2 Framebuffer

2.1 Setup

The wrapper written for the framebuffer driver that is already provided by the system uses mmap to create a consistent and easy interface to write to the screen. *mmap_fb*, *unmap_fb*, *setup_screen* and *teardown_screen* all take care of setup to talk to the driver in the intended manner. *paint_screen*, *update_screen*, *paint_region* and *update_region* take care of changing the underlying array structure and actually blitting the given regions given, by interfacing with the driver with custom ioctl functionality.

2.2 Tetris

Tetris makes use of the wrapper for the framebuffer by again creating its own abstractions on top of the already provided functionality. Specifically, Tetris's basic blitting blocks are divided into areas of 10x10 pixels, and glyph blocks (glyphs here being a letter or digit of a character string to blit) are divided into basic blocks of 5x5 pixels. Tetris also makes a distinction between just painting to the underlying data structure, and actually updating the screen. This ensures that Tetris updates regions of the screen that are as small as possible at a time, as to make the game feel more responsive than if we were to always update the entire screen.

3 Sound driver

3.1 Introduction

To create a sound driver, we early decided that we wanted to try to make the DMA to work for us. However, to make sure we were on the right track, we decided we would first try to synthesize sound with the real time timer ourselves.

This did not work out however, as we figured out after trying to tune a few kernel configurations to our advantage. We quickly realized that there was no way that the Linux hardware timer implementation would do the required job with the initial settings of not having enabled high resolution timers, as well as having a tick rate as low as a 100Hz (as can be seen in /proc/timer_list, where the resolution of all the timers are set to 0.01s, and the event_handler is set to to tick_handle_periodic). We tried a tickless solution with NO_HZ, which hung the CPU, and a higher tick rate, which made the CPU sluggish. We therefore decided to go straight to DMA implementation without trying any further. We did, however, notice that PREEMPT and PREEMPT_RCU is enabled, but did not try without these newer kernel features that may have caused problems.

We failed to implement a sound driver with the DMA, but we have a notion that we are very close after working on it more than the rest of the exercise all together. The PRS-system works very well, but we receive an error on the AHB bus when the DMA tries to read its descriptors. We have tried a lot to make it work, and have come to the conclusion that the control block passed to the DMA with DMA_CTRLBASE points to memory allocated by the CPU that might not equal the address on the bus line. We have tried the infamous virt_to_phys to make it translate, but both the DMA handle and the CPU handle returned from an allocation in a dma_pool both return the same 512-byte aligned address. The allocated memory is part of the (S)DRAM given to the kernel in System Type in kernelconfig with base address 0x16000000.

We have not cleaned up the code in the sound driver into smaller, coherent blocks, as we were trying to get it to work until we had no more time, but it should be clear what it is doing by following the flow of execution anyway. We have also not bothered doing the deallocatoin of resources, as we never made the driver work quite right.

3.2 Implementation

The driver for the DMA, and sound in general, is implemented according to [3]. The driver first initalizes a character device the same way as the gamepad driver (it was intended to work in such a way that user space programs could write a wanted frequency to its file descriptor to change the sound). The driver also requests memory regions for upper and lower DMA memory regions, the PRS, Timer3 and DAC0.

When the driver has requested and received its memory, it sets up the peripherals for the DMA to feed DAC0. This is done by first enabling the PRS-system by writing SOURCESEL to 0b011111 for Timer3, and SIGSEL for overflow to PR_CH0_CTRL. We then initialize Timer3 with a period of 0x13D, to trigger the DMA 44100 times every second. The DAC is then initialized normally, but sets the control registers of its channels to do conversions on PRS triggers, and not data writes, so that it receives data from the DMA whenever it is ready.

We then allocate memory for the DMA control block and data with a call to *dma_pool_create* and then *dma_pool_alloc* with 512-byte alignment so that the lower eight bits can all be used according the the way intended by the DMA to select channel and then src, dst and control of this given channel. The control block is then populated with pointers to the memory we want to write to the DAC, as well as the correct control register settings. For us, the data are two single pointers to two single bytes, one representing the minimum of the amplitude in the primary channel, and the other representing the max amplitude in the alternate channel. This means that we with ping pong mode and looping can set the length of a single DMA cycle such that it contains the amount of transfers of a half period of our required frequency, which we are using a square wave to create. When user space then signals a change in frequency, we can change the amount of transfers in a single cycle to change the frequency accordingly.

It seems things are set up correctly when we read all our setup registers one by one through the driver, but the AHB-bus is still throwing ERRORC in DMA_IF and interrupting after we enable channel 0, which means that something went wrong on the bus. We were unable to figure out why, but have made sure the control block registers of channel 0 points to the correct memory regions for DMA transfer.

4 Tetris

4.1 Introduction

Tetris is an arcade game where shapes made by tiles falls down to the ground. When a shape reaches a solid ground, either the ground or another previously stopped shape, it stops and freezes in place. Shapes are spawned at the top of the game area, and falls down at a speed of 1 line per second. When a full width of the board is filled in one single row, the row is cleared and the score is incremented. The board is modeled like a bucket and the shapes can only move inside it and not interfere with other shapes.

This chapter only briefly touches on the functionality of the Tetris implementation, as implementing a slightly complex game is not part of the exercise. We have therefore also preceded each function in our Tetris with a brief description of its functionality, should the reader be interested. We have also described a few intricacies with the implementations at the end of the chapter, that is relevant to the exercise moreso than the game itself.

To play the game, execute *modprobe driver-gamepad && game.*

Input actions movement

Each shape has 5 actions: step {left, right, down}, rotate and fall through.

Restrictions

- The shapes cannot move further left, right or down than the boardś edges.
- A shape cannot move through other shapes.

4.1.1 Scoring points: trickle down

When a row is filled, all rows above the complete row are shifted one row down. The top row is cleared.

4.2 Implementation

The implementation has inputs SW{1, 2, 3, 4, 5, 6, 8} from the gamepad and the output is the screen.

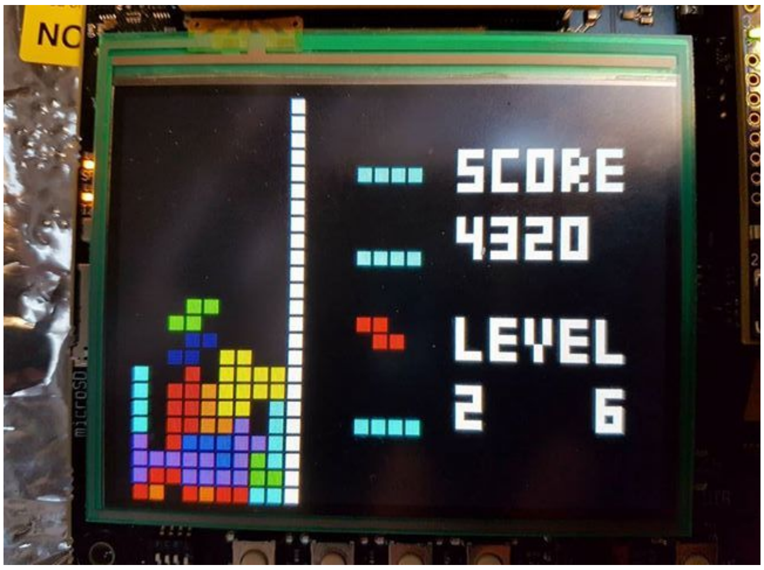

Figure 4.1: Tetris in play

4.2.1 Button mapping

Actions

- SW1: Step left
- SW2: Rotate
- SW3: Step right
- SW4: Fall through
- SW8: Step down

Game

- SW5: Exit game
- SW6: Restart game

4.2.2 Data structures

Shapes are represented by 4x4 matrices and stored in a 3D array. Each shape has 16 tiles which can be solid, and any shape with height and width from 0 to 4 can be represented. Solid tiles are represented by 1 and non-solid are 0.

A classic square shape can be represented by 2x2 tiles solid.

```
[[0, 1, 1, 0],
 [0, 1, 1, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]]
```

The board is represented by a 24x10 matrix. When shapes are frozen, they are added to the board and an update of the framebuffer is triggered. Rotation is as simple as rotating a matrix. Shapes travel down the non-solid tiles of the board. Once the bounding box of one of the solid tiles of a moving shape touches a frozen tile head to head, moving shape on top, the shape will freeze if it's not moved for a few ticks.

Text is represented the same way. Glyphs are represented by solid and non-solid tiles in an NxM matrix are.

4.2.3 Drawing

Painting tiles is the way we represent everything on the screen. Only small, required regions of the framebuffer are updated when a tetromino moves, and the screen is redrawn on specific actions such as when a tetromino is transferred to the board. The model is built on representing tiles with a border of 1 pixel. This loosely translates the screen into an inflated screen: one model (tile) pixel is translated into 10 screen pixels.

Projection

A projection of where the tetromino will land will displayed for the player. This was implemented to help weaker players, like us, enjoy the game.

4.2.4 Incoming tetromino queue

The next four tetrominoes that are inbound are shown to the direct right of the game area. This is implemented by carefully allocating (and freeing) dynamic memory and passing it to sys/queue.h macros that implement a singly linked list. The next shape that you will receive can therefore be seen at all times.

4.2.5 Scoring and levels

Scoring is based on how many lines is cleared. This means that clearing just one line gets you a lot less points than scoring a tetris, which is to clear four lines at once.

The scoring system is a copy of the retro system for Game Boy. There are a total of 9 levels, a new one is reached every time 10 lines is cleared. The time between ticking the player's tetromino is set to be a full second minus 100ms for every level. This means that when a player has reached level 9, there is only 100ms between the ticks to add difficulty to the game. The two numbers under the LEVEL label are the current level to the left, and the remaining lines to clear before reaching the next level.

4.2.6 Intricacies

For our implementation of Tetris, we met some challenges that were especially tricky.

The first being that nanosleep would be woken up by an asynchronous signal from the gamepad, and not keep sleeping. To solve this, we created our own wrappers, __nanosleep, __mssleep and __sleep, that sleep for nanoseconds, milliseconds and seconds respectively. This means that if you call __sleep(1), the process will go back to sleep after a signal handling, and sleep further until it has slept the full second.

We also had problems with the asynchronous signal from the gamepad happening at the exact same time as we were already executing code to tick our game, which would not turn out completely deterministic. To solve the problem, we wrapped the part of our main loop that has to happen atomically in a mutex lock. This means that if we are signalled during this execution, the signal handler will see that the thread of execution is locked, and set a deferred flag, as well as its current state. When the atomic block then finishes the execution, it calls the handler again with the deferred flag set, so that the signal handler can still do its work, but in a deferred manner. Had the application been multithreaded, this would not have worked, but since we only have one flow of execution, simple flags will do without using more complicated POSIX constructs like sigset

The combination of these made Tetris feel very responsive, with no hiccups. The only problem we have had other than this, is that we at two occasions would receive a kernel panic after playing for about five minutes, with an instruction pointer that pointed nowhere near our code. The problems disappeared if we flashed all images to the board again. We have come to the conclusion that we somewhere in our code have an overflow that writes a piece of memory we should not. If we are unlucky, our game might be loaded in such a way that writing this overflown memory actually triggers a panic - however, most of the time it is mapped such that writing there is seemingly harmless, and we notice nothing. We have not spent time tracking this down, as it has only happened twice out of hundreds of playthroughs (and we have let the game idle by itself for hours without restarting with no problems).

4.3 Finite State Machine of Tetris game implementation

Some transitions with actions "always" are coupled together. The TICK transition happens every {0.1, ..., 1.0} seconds, depending on which level the current game has. Nanosleep transitions to the Sleep state. To transition from Sleep, either the timer has to Top out or a SIGIO event has to occur. After a SIGIO and a Draw Region has returned to the Game Loop it Nanosleep transitions to Sleep for remaining msecs.

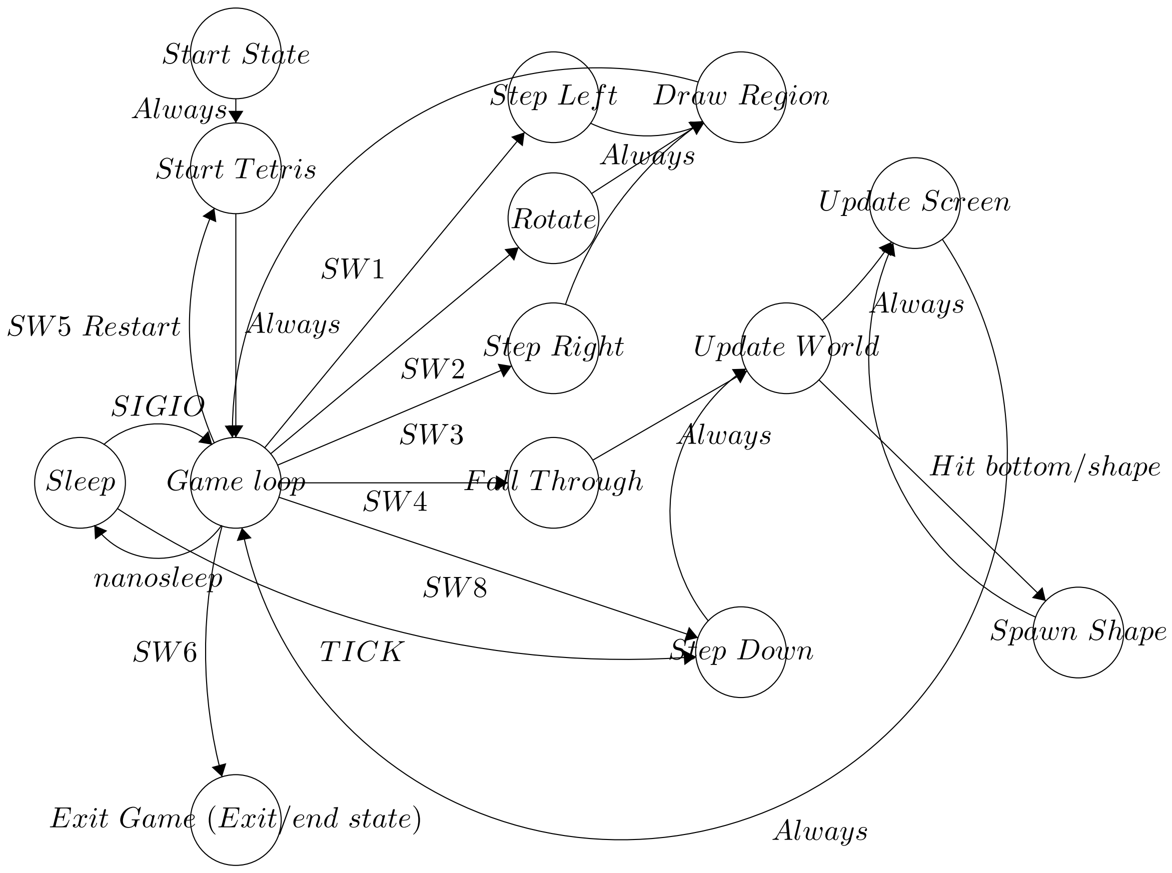

Figure 4.2: State machine for Tetris game.

5 Energy Measurements

For energy measurements, we did not achieve anything significant, even though we tried several power saving techniques:

- Configure kernel config to use opportunistic sleep, run-time PM core functionality, CPU idle PM support, and most of all using tickless idle on the CPU. While we did get the Cortex-M3 to run with event_handler: tick_nohz_handler, it did not have any immediate effect on power consumption, which stayed at around a steady 7.5mA (as seen in the figure below). We have left out our extra kernel configuration of the code delivery, as we didn't think it was necessary to deliver the small changes.
- We sleep at any time the CPU is doing nothing. This is achieved by calling our own custom sleep functions that go back to sleep after handling signals to sleep as much as possible.
- We update as little as possible of the screen as needed when the game state is updated. This includes only drawing the tiles of the board that are actually occupied, only drawing the actual tetromino, and rarely touching the static parts like SCORE and LEVEL.

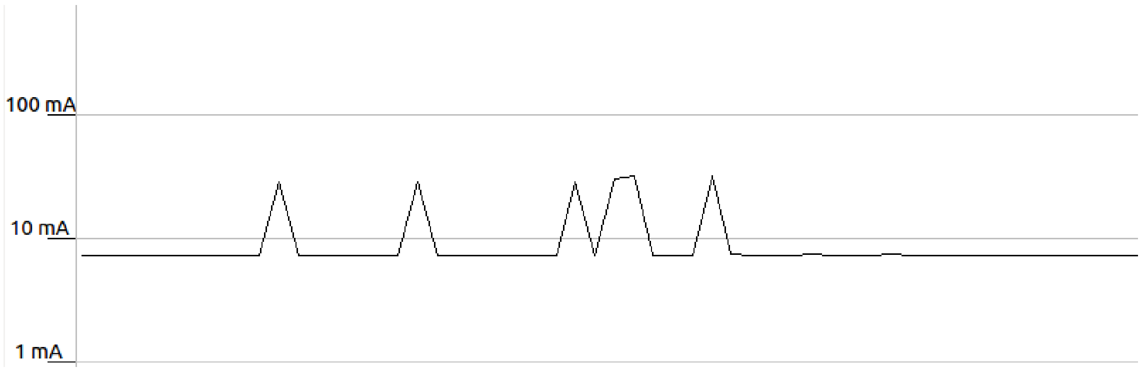

Figure 5.1: Power measurement

Bibliography

[1] Kroah-Hartman Corbet Rubini. *Linux Device Drivers, Third Edition*. O'Reilly, 2005.

[2] https://www.kernel.org/doc/html/v4.10/process/coding-style.html. *Linux kernel coding style*.

[3] Silicon Labs. *EFM32GG Reference Manual*.