NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

Exercise 3

Group 27:

Eirik Prestegårdshus
Tor Braastad Gylder

April 27, 2015

Abstract

Today, the microcontrolles has become so small and energy sensitive that energy-efficient programming has become a huge factor. The market for energy-efficient components and devices are increasing every day, not only for longer lifespan, but to gain the maximum out of everything so that we can spare the nature for its resources. The assignment is focused on the creation of a kernel device driver, and using it to interact with a simple game. The group need to use their knowledge and experience about energy-efficient coding. This assignment goes a step deeper than the earlier assignments, by creating a whole game from the upper level down to the hardware. While programming it would be a good idea to keep it as simple as possible with small updates on the screen at a time. This task also gives an understanding to the difference between dynamic and static power consumption, as well as how to cut down the dynamic power consumption. The static power consumption was on average 7.29mA, while the average dynamic power consumption was on average 15.9mA. Even though the spikes was greater than 40mA, the use of a asynchronous signal helped to shorten handling time of interrupts. This report provides a solution to the third lab exercise in TDT4258 Energy Efficient Computer Systems, in which we were supposed to make a computer game.

1 Introduction

The assignment given were to make a simple game with a kernel module and user-space interaction using an EFM32GG microcontroller. This report will explain in detail how the microcontroller were utilized, some energy measurements, how the game works and how it was built. A condition given by the exercise was to utilize Linux distribution and build the code to work with the predefined Linux environment. The game is similar to *Snake*, a very common game as it was included in almost every old Nokia phone.

2 Theory

In order to make the game run it'll need the following; Kernel space, User space, interrupts and how to interact between all the modules. This chapter will give some background information about this.

2.1 Building & compiling packages

Using µClinux as an operating system includes the kernel based of Linux kernel 2.0 as well as many user applications, libraries and tool chains. It's intended for microcontrollers that doesn't have Memory Management Units. The build system, *ptxdist*, is used to compile and convert the packages into binary files before the files are flashed to the development board. More documentation on the functions in *ptxdist* see the compendium in table 5.1. In this assignment a useful skeleton code were provided, since this assignment focuses on the kernel build and interaction with it.

2.2 User space vs kernel space

The Linux environment has two different modes; kernel and user. The user mode restricts the user direct access to hardware. Most programs runs in user space, the reason for this is that user space is pretty safe. Crashes and errors won't leak to memory not allocated as it does not have access to every part of the machine.

The kernel space has access to basically everything. In theory and practice, a program in kernel space is actually able to completely wreck the entire memory. As doing wrong operations in kernel space is very dangerous, a program here should be as simple as possible and proceed with caution.

2.3 Kernel space

The framebuffer works like a graphic card, it gives the user-space access to write to the screen on the development board. The driver on the other hand connects the gamepad (GPIO) with input and output information and gives access to interrupts. Both of these modules are character devices, what defines a char device is that they have I/O capability. The kernel module or the main driver will setup the machines most important functions. This module will also interacts with the connected computers components like CPU. It also gives access to everything in the machines hardware and works like a link between the user-space and the hardware. There are some rules the kernel module has to follow:

- The kernel must have a strictly set of functions so that the kernel knows exactly how the module should be used.
- The kernel module can only call functions which are defined in the kernel itself.
- Kernel modules needs to be programmed with parallelism in mind, because it needs to function correctly even if different processes are trying to use it at the same time.
- They are event based. That means that they can't have loops running eternity, this would hang the module. Instead they have functions that can be called by other programs when they need access to it.

2.4 User space

The game is made in user space, this is to prevent it from having access to every part of the operating system. The game is very similar to Snake, found in most old Nokia phones. The game consists of a snake, and the person controls it. The goal is to eat apples generated on the screen. Every time it eats an apple, the length of the snake will increase. A longer snake will prove a higher difficulty as it has to dodge it's own body. Every time you catch an apple, there will be a new one generated somewhere on the screen. It should also be able to raise or lower the speed of the snake.

2.5 Interacting between the modules

The user space program has to access the driver running in kernel space. Interacting with the kernel space is usually done by calling a few predefined function names all stored in a *file_operations* struct for every driver. The struct contains links to each function to be called when a call is made from user space. E.g write function will copy a buffer from user space and do some action with that.

2.5.1 Asynchronous notification

There are two common ways to implement this in Linux, either *Signal(0)* with the use of *fa_sync* in the *file_operations* struct or the recently popular option, using a sigaction object. This allows user space use signals in a similar fashion to the driver using interrupts. Using signals opposed to polling the driver for a read every time is a great improvement efficient wise as there's less work to be done.

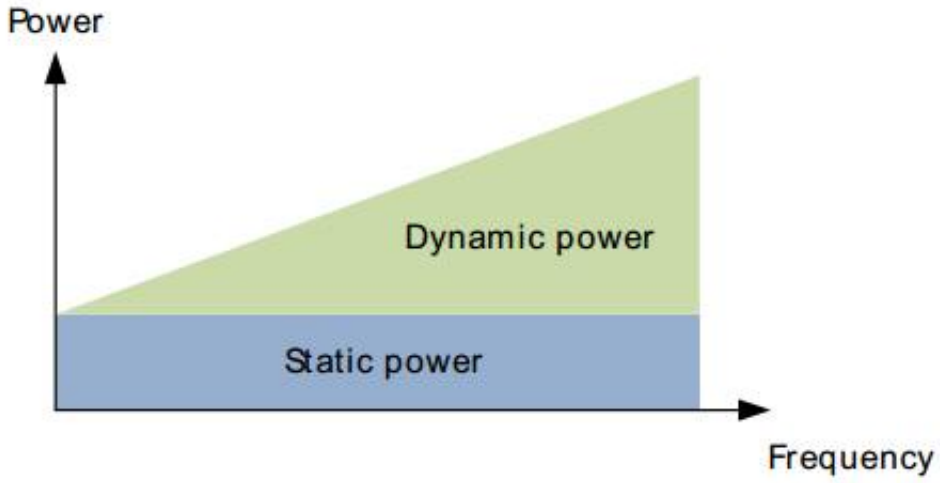

Figure 2.1: Power consumption as a function of frequency[1]

2.6 Energy Efficiency

When talking about reducing the energy-consumption for a microcontroller we talk about lowering the static and dynamic power consumption. Static power consumption it the generated power when there are no circuit activity, it's independent of the clock and constant. While on the other hand the dynamic power consumption is pendent on the clock frequency, by rising when the frequency increases and lowering when it decreasing. In the figure 2.1 above one can see that the static power consumption is constant, while the dynamic power consumption rises with the frequency.

3 Methodology

3.1 Driver

The first thing the driver needs to setup is *alloc_chrdev_region()*. This function will find out how many major numbers your device will use and allocate them. It will then allocate the I/O memory with *check_mem_region()* and *request_mem_region()*. When allocating, the kernel must memory map the information before it gains access to it, this is done with the *ioremap_nocache()*. In this assignment the use of nocache is a better implementation since all the I/O memory are already visible through nocacheable addresses. Beware that you need to freed these regions when they are no longer needed. This is usually done during the exiting of the kernel. Since the I/O memory has been mapped the kernel can setup the GPIO and how to interrupt. By creating a struct with *cdev_init()* and *cdev_add()* the kernel module can communicate with other modules. Now the other modules gains access to the kernels functions which are defined in this struct. A device class is created so that the other modules can find the kernel.

3.2 Game

The game utilizes the following device drivers to use to make it functioning; framebuffer, random and gamepad. The game should have the following specifications:

- Be able to pause
- Be able to reset the game
- Steer in the direction of the last button press
- Generate a new random apple when last one eaten
- Increase length of the snake when apple eaten
- Run at the same speed regardless of the snakes length
- Stop the game upon collision with itself
- Be able to go through the wall and appear on the other side

3.2.1 The Screen

The framebuffer is the device driver that handle the screen. When writing to bigger parts of the screen the framebuffer will quickly turn into a bottleneck for your program. This makes it very important to only update the parts of the screen that you actually have changed, rather than the entire screen. Every loop of the program there are only two things that has to be rendered, the snakes head and either erase the tail or generate a new apple.

Rather than writing values of the pixels to the device driver, the game utilizes a very common way of handling framebuffer, memory mapping. The game memory maps the array used by the framebuffer, this way it's easily accessed and can be written to directly.

A sensible way of doing it is to part the screen into a grid, with each square being of 10 times 10 pixels. This way the game only has to write to a maximum of 10*10*2 pixels every time, rather than every pixel. Only exception being when you need to reset, the screen will roll back, generate a new initial snake and update the entire screen.

3.2.2 The Snake

For storing the snake we actually had to try a couple of different solutions before reaching a final solution. The first idea was to make a dynamic snake that used built in functions in Linux, *malloc()* and *realloc()*. After the snake became a certain length, the *realloc()* did not function properly and fail to allocate more memory to the pointer made by *malloc()*. This sort of makes sense as the memory will be reallocated quite often, and always with only one additional position.

Another solution was to keep everything in a huge fixed-size array, and have an integer to determine how long the snake was. The idea was to keep the snakes head in position zero at all times, and the body following. For this to happen, every time it moves it has to swap every element backwards until it reaches the end, and then fill in the head. When the snake became long the speed of the game began to decrease, this was due to all of the operations performed to shift the snake by one.

The best and final solution turned out to be a huge fixed-size array. Shifting the entire snake all the time like in the previous try was not very clever. What we did was to reverse the order, and let the snake traverse through the array having two indicators tracking the snakes tail and its head. When it reaches the end, it simply continues at the start; overwriting the previous which are no longer needed. The game now has a much more stable speed, and does not have a bottleneck pulling the speed down.

This solution also improves the energy efficiency by a great deal since the microcontroller is able to sleep, in contradiction to working it's hardest doing matrix operations.

3.2.3 The Apple

So how does it generate a new apple? It has to seem random, and it has to be placed elsewhere from the snake. It's the possibility to write a pseudo random function which places an apple at a valid location. Instead of making it, the game uses a built in device driver *urandom*. The apple generator will simply brute force random positions, until it

finds a valid one. This will luckily not prove a problem since most likely the snake won't cover huge amounts of the screen.

3.2.4 User Interaction

The user should have the option to pause, resume, reset, increase and decrease difficulty and change direction of the snake. A value of which button is pressed has to be sent from the device driver for the gamepad and update values in the game. Rather than having the driver to ask for the last button pressed every time it moves the snake, its favourable if variables in the game will be changed asynchronous. Having a asynchronous solution also enables the ability to easily pause and resume, as well as reset the game. More information on how the asynchronous notification happen can be found in the next section, Signal Handling

3.3 Signal Handling

The gamepad is the only physical component which can give inputs to the board during the runtime of the program. After initializing, the driver waits for a button-press on the gamepad. To achieve a fully asynchronous solution built up by interrupts the program need a way to update the direction without the need to read the driver only when there's a change. *Signal()* is a way to pass signals from kernel space to user space. Whenever a signal is raised, the user space program interrupts its current task and performs a task specified by signal. One way to do it is by sending a signal to the game, and then the game will respond by reading the driver. Another way, is to use a special functionality of *sigaction()*; in the code named *info*. Setting the flag $info.si\_code = SI\_QUEUE$ allows user space programs read a 32 bit integer put in *info.si_int* by the driver. The game could then simply read from the signal itself, rather than reading the driver.

In order to send the signal queue to the right place, the driver needs to know were to send it. The user space program is identified by a process ID. The user-space can write these to the driver so the driver can figure out what process to notify.

4 Results

The functionality of the game feels responsive and handles quick and on point. It's no more slowed down by operations when the snake becomes long. When displaying head of the snake it also felt natural that each link of the snake should be separated, this was achieved by writing to 8 or 9 of 10 pixels, leaving the outer square black.

4.0.1 Energy consumption

As shown in the figure 4.1 below the static energy consumption lies around 7.2 mA, this is decent when the game is running on the board. When an interrupt is raised the power usage increases a lot. To lower the energy consumption during this interrupt, the interrupt has to take less time. This can be done in many different ways, but with a asynchronous signal sending before the interrupt is being read by the game, the lifespan of the interrupt decreases. As figure 4.1 describes on the side, the average power usage during an interrupt is 15.9 mA instead of 40 >mA. The energy consumption will rise as the speed of the snake increases. This is because time between each spike will be less, and there's more operations per time.

The energy consumption depends also on the implementation of the game. Earlier in this report we mentioned dynamic allocation of the snake. As the snake grew, the energy consumption grew as well. By memory mapping the screen and update it only with the used areas, simplify the code also decreased the power consumption to great extent. In short, the less operations it has to do, the better.

One thing that also draws power is the operating system, it requires power in order to provide functionality for the programs running. If compared to the previous exercise the power usage is multiplied by tenfolds. The main focus was to reduce the dynamic power consumption, Linux will passively drain somewhat static energy, usually it's the processor that eats up all the power. Easing the work for the processor it the main concern when utilizing an operating system.

4.1 Problems & Debugging

As mentioned earlier there was a problem when reallocating a dynamic memory numerous times. The solution to this was to use a huge fixed-size array which most likely is big enough that the snake won't eat up it's own tail. The chosen size for the array is 400, this might seem odd as the snake can be up to 24*32=768 elements long. When that length was used it created yet another problem. It seemed like it used some memory that were used elsewhere and the values could suddenly change. We did not have the time to experiment with this a lot and chose 400 as a reasonable size, as our users would most

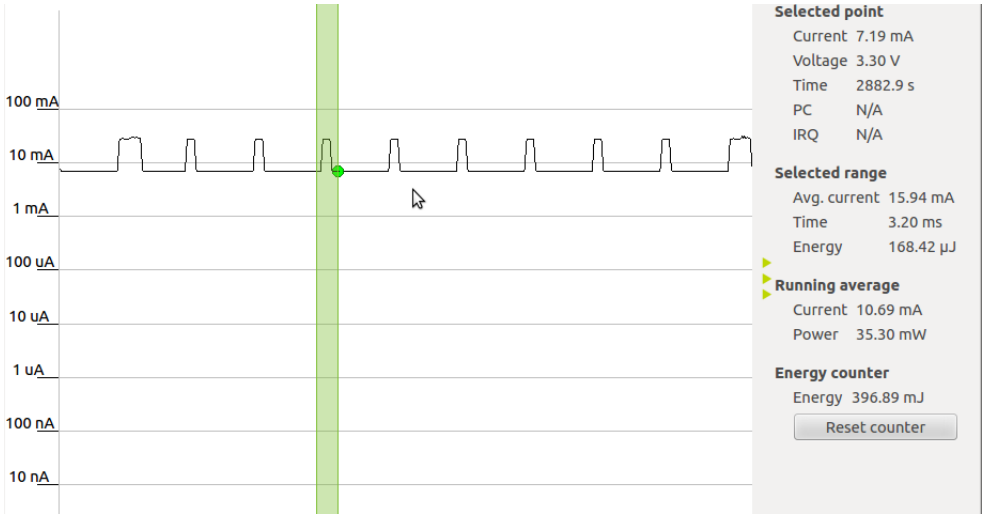

Figure 4.1:
eAProfiler graph of the energy consumption of the game with signal interrupting

likely never catch more than 360 apples. In case that happen, the user should really consider upping the speed. In our testing, an array of size 400 have worked flawlessly. In some few cases the flashing of the board didn't flash everything. The problem lay in a root module in the given skeleton code. One way to overcome this problem was by; resting the board, import a new .zip file of the given support files and export it.

4.2 Further Improvements not yet Implemented

There are a few additional commands that can easily be added to the *sig_handler* that can improve the functionality. A case were you press reset and pause at the same time could exit the game safely deallocating every memory and closing open device drivers, and the proceed to turn the microcontroller off.

One could also add a functionality to make it easier to take sharp turns at high speed. In case it's two buttons pressed at the time it will find the valid next direction based on the current direction, and then it'll handle as it was only the one valid button pressed. Having them both pressed for a long time will cause the snake to go diagonally, in a "zig-zag" pattern of course.

There also might be additional power-saving options within Linux, but we did not have the time to explore this opportunity further. It could have been a timer counting down, and being reset upon button-press. If the timer runs out, the screen can be turned of to save power.

5 Conclusion

The game is simple, yet very fun and functioning properly. We feel we reached the specifications of the game and with the clever use of signals, and the framebuffer. The signals provide the functionality to update variables in the game asynchronously in the background without the actual game asking for a update. Only updating the parts of the framebuffer it actually uses saves the processor a lot of effort, and contributes to a much more energy efficient implementation. Clever use of internal variables and easing the processing work does also prove a great difference, as there are hundreds of different ways to implement the same functionality. This was easily spotted when the game speed actually slowed down because the processor could not keep up with all the matrix operations.

Linux provides a nice environment to use, with a couple of useful already implemented functions and drivers and a proper framework with user and kernel space. However, that comes at a cost. The cost is higher energy consumption in order to run the operating system. That put aside, the main concern was to reduce the dynamic power usage, primarily used by the processor running.

Bibliography

[1] Silicon Labs. Energy optimization application note, 2013.