U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Legend of LCOM

## Concept clone of The Legend of Zelda

**Masters in Informatics and Computing Engineering**

Computer Laboratory

Class 4 – Group 8

2nd of January, 2017


Francisco Teixeira Lopes ei11056@fe.up.pt
Rui Miguel Oliveira up201000619@fe.up.pt

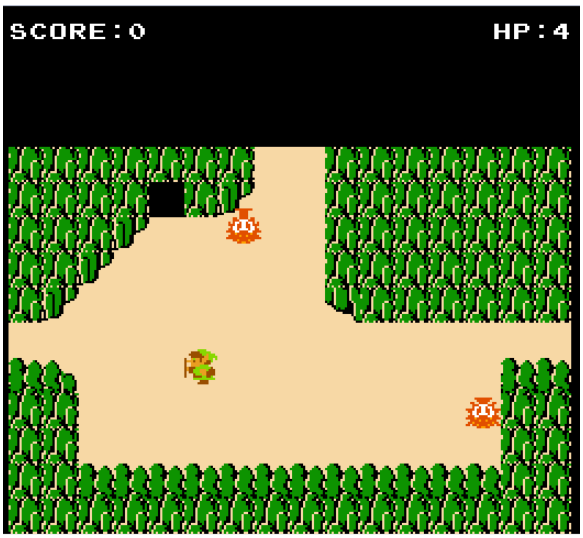
# Index

# User Instructions

## Player 1

Player 1 version of the game can be ran by typing “service run `pwd`/LoLCOM –args “player1””, this starts the program on the following menu:


The user can exit the program by pressing ESC, alternatively the user can select one of the two options on the menu, they’re related to the kind of graphical feature the game will use, double buffering or page flipping. After selecting one option the following screen shows up:




Now the user is in the game proper, WASD moves the character, as does moving the mouse, J attacks as does the Left Mouse Button. The scroll wheel and 4th and 5th buttons can be used if they exist on the user’s mouse. ESC quits back to the menu. You can not attack after getting hit.

The user can exit through any of the 3 exits shown above, there are 4 maps total, so one on each direction from this map. Enemies spawn each 5s through RTC alarms, killing enemies awards score and getting hit lowers HP. When HP hits 0 the game is over and the following screen shows up:


Pressing and releasing ESC or ENTER goes back to the main menu. That's it for the Player 1 part of the program.

## Player 2

Player 2 version of the game can be ran by typing "service run `pwd`/LoLCOM –args "player2"", the following screen greets the player, the player then hits 1, 2, 3 or 4 to spawn the corresponding enemy on Player 1's game, for a maximum of 3 enemies at the same time. A cooldown counter makes Player 2 have to think about what monster to choose, the powerful Red Lynel (number 4) can make a dent on the other player but comes at the cost of 9 seconds of inactivity. ESC exits back to MINIX. That's it for the Player 2 part of the program.


## Speaker1bit

For playing the speaker in square wave mode the user can issue the following command "service run `pwd`/LoLCOM –args "speaker1bit ZeldaOverworld.csv"", the 2nd argument is the music file to be loaded, one is provided with the game, that being the one in the example given.

## SpeakerPWM

For playing digital 6-bit music with the speaker the user can issue "service run `pwd`/LoLCOM –args "speakerPWM 19244 ZeldaOverworld.pcm"", the 2nd argument is the sample rate, 19244Hz is the recommended one (gives the most sample values, see Implementation Details), the 3rd argument is the music file, an unsigned 8-bit raw mono PCM file which gets converted

to 6-bit during loading. One can use ffmpeg to do quick conversions to this 8-bit format. ZeldaOverworld.pcm is provided with the program.

Reminder: Both speaker1bit and speakerPWM don’t work on VBox, they’re included in the program for demonstration purposes as only running MINIX natively can allow these functions to work.

# Project Status

## Device table

| Device | Usage | Interrupts |
|---|---|---|
| Timer | Updates display and game logic at a constant framerate | Yes |
| Graphics Card | Image/Sprite/Font drawing | N/A |
| Keyboard | Player interaction | Yes |
| Mouse | Player movement | Yes |
| RTC | Monster spawning with alarms | Yes |
| Serial Port | Send scancodes that spawn monsters | Yes (Receive) / No (Transmit) |
| PC Speaker | Playing square waves or digital music with PWM | Yes |

## Timer

Used to update the screen and game logic at a consistent pace. Both lolcom_player1() and lolcom_player2() use the timer to call a game loop, logic_gameloop() is called upon a timer interrupt while playing as Player 1. Besides these functions, the timer is essential to PC Speaker operation, which will be detailed in its own section.

Major functions:

logic_gameloop(). – updates display and logic for Player 1

Minor functions:

lolcom_player1(), lolcom_player2() – driver_receive loops

logic_serial_tick(), logic_display_serial() – updates display and logic for Player 2

## Graphics Card

Used to draw pixels on screen, these can take the form of sprites, fonts, plain images, etc. The video mode used was 0x112, a 24-bit direct color mode with a resolution of 640x480 which was configured to run with a linear framebuffer.

Double buffering is implemented as well as page flipping with VBE Function 07h, Player 1 can choose between these two buffering methods on the main menu.

Simple animations are implemented with the usage of structs that maintain the status of the current animation for a given object. The actual animation is implemented with either entirely different frames or with sprite sheets.

Collisions between entities is done with a simplified AABB approach (Axis-aligned bounding box) and collision with the environment is done through checking 8 hotspot pixels around the entities with the collision data of the map.

Major functions:

Vg_init(), vg_init_values() – uses VBE Function 01h to initialize video mode
Vg_pageflip() – uses VBE Function 07h to implement page flipping

Minor functions:

Video_gr.c – the remaining functions draw the different kinds of structs that make up the game world part of the program

## Keyboard

The keyboard is used for player input, be it selecting options on a menu or actually playing the game. See User Instructions section for key bindings.

Major functions:

logic_kbd_input() – handles all keyboard inputs for the game, using state machines

Minor functions:

keyboard.c – remaining functions, mostly similar to lab3

## Mouse

The mouse can be used by Player 1 to move around, the left button is used to attack. Of particular interest is the usage of a 4th packet which contains information about the scroll wheel and the 4th and 5th button if they exist, specifics about this will be on the Implementation Details section.

Major functions:

logic_mouse_input() – used for Player 1 interaction
mouse_magic_sequence() – used for activating the 4th packet

Minor functions:

Mouse.c – remaining functions, mostly similar to lab4

## Real Time Clock

Used for spawning monsters for Player 1 to fight. The monsters are spawned by an alarm interrupt from the RTC which is reset each time it attempts to spawn a monster. The monsters spawned from the RTC are limited to 3.

Major functions:

Logic_rtc_handler() – reads stat register C, tries to spawn monster and then resets alarm
rtc_setalarm_s() – sets alarm x amount of seconds after the current time, uses a mixture of RTC functionality so it works at any given time (23:59:55 for example)

Minor functions:

RTC.c – remaining functions, an implementation of lab6

## Serial Port

Transmitter side: a separate driver_receive cycle waits on inputs from Player 2 and sends them to Player 1 to spawn monsters, only valid inputs are sent. Monsters limited to 3 which

totals 6 with the RTC. Transmission parameters are 8N1 9600 bitrate with no interrupts turned on and no FIFOs.

Receiver side: main game loop gets interrupts from the UART and spawns monsters according to the info received, up to a limit of 3. Receive parameters are 8N1 9600 bitrate with data received interrupts turned on and no FIFOs.

Major functions:

logic_serial_handler() – Receiver side handling of data received

uart_send() – checks LSR of the UART and sends byte if buffer is empty

uart_receive() – reads IIR and checks LSR for errors, if no errors returns data read from RBR

uart_version() – not used in the project but interesting nonetheless, determines UART version

Minor functions:

UART.c – remaining functions, an implementation of lab7

## PC Speaker

This device works only on native MINIX, any virtualization won't cut it (at least as of January 2017). There are two different implementations of the speaker in the project, firstly, the expected beeping mode of the speaker (square wave generation), secondly, and more interestingly, 6-bit digital playback using Pulse-Width Modulation techniques. These will be analyzed further in the Implementation Details section. In the project, the speaker is used to play accompanying music, this can be done on a separate computer, from our own tests, running the game and the speaker at the same time is not viable on the hardware that would be able to run MINIX, at least not without excellent optimization.

Major functions:

speaker_PWM – plays unsigned 8-bit raw mono PCM (converted by ffmpeg for example), scales it to 6-bit values since the PC Speaker isn't capable of more, details on Implementation Details section

speaker_square – plays single notes using Timer 2 square wave generation, reads .csv file with the notes and duration to play a song

Minor functions:

Speaker.c – remaining functions, mostly rehashed from lab 2

# Code Structure

## Modules

### Keyboard

Mostly similar to lab3 version, can read scancodes from the KBC, can write to KBC, can write commands to KBC and has utility functions to clear the output buffer so MINIX doesn't hang.

Developed by: already developed

### Mouse

Mostly similar to lab4 version. A notable difference is the usage of a 4th packet, this functionality is activated by the function mouse_magic_sequence() and allows for the usage of the scroll wheel and up to two extra buttons.

Developed by: Francisco Lopes (4th packet)

### RTC

Implements partial functionality of lab6, allows reading of any CMOS register, writing to any CMOS register, displays date in human readable format, displays RTC configuration (the 4 status registers), can reset the RTC to the configuration MINIX expects, can set the date (doesn't change weekday or century byte), can get the date and fill a struct with the information and can set an alarm a certain amount of seconds after the current date.

Structs:

> Rtc_date_t – stores a date read from the RTC, can be used in rtc_set_date() and is used by rtc_set_alarm_s

Developed by: Francisco Lopes

### Speaker

Implements the PC Speaker, can be used to play in square wave mode or PWM mode, in square wave mode Timer 0 sets the frequency of Timer 2 after the duration of the previous note has ended, this information is read from a .csv file (ZeldaOverworld.csv in resources). PWM mode uses Timer 0 to call Timer 2 (in mode 0) at a high frequency, this makes it so the PC Speaker only gets applied voltage for less time than it needs to reach full extension, allowing it to reach positions that weren't available, this makes it follow a soundwave pattern that resembles the waveform view from a music player, thus allowing it to play digital music. This module is a bit messy, it took a lot of time to learn how to play digital music on the PC Speaker and the code quality suffered, hopefully the comments are enough to make some sense of it.

Developed by: Francisco Lopes (programming), Rui Oliveira (file adapted from piano sheet)

### Timer

Mostly similar to lab2 version.

Developed by: already developed

### VBE

Mostly similar to lab5 version

Developed by: already developed

### Video_gr

Similar to lab5 version but adapted to project structs. Draw functions specifically for structs containing game world info. Additionally VBE Function 07h is implemented here which allows for page flipping, a sort of hardware “double buffering”.

Developed by: Francisco Lopes (page flipping), Rui Oliveira (some drawing functions)

### UART

Implements partial functionality of lab7, can read UART registers, write to UART registers, reset COM1 or COM2 to the expected configuration by MINIX, can get communication parameters and store it on a struct made for this purpose (UART_config_t), can use UART_config_t to display communication parameters in human friendly way, can use UART_config_t to set the communication parameters of COM1 or COM2, can send bytes through COM1 (checks LSR for buffer empty) and can also receive bytes on COM1, checks the IIR and then the LSR and if no errors occurred returns the data read from the RBR.

Structs:

UART_config_t – stores communication parameters such as data bits, stop bits, parity, bitrate and interrupts activated

Developed by: Francisco Lopes

### LoLCOM

Main function of the game, can be called with command line parameters to choose what functionality it should start. Can run Player 1 mode which is the main game loop and the bulk of the program, it features a driver_receive cycle that dispatches interrupts to the Logic Module whose purpose is parsing them and updating the game accordingly. Can run Player 2 mode in which the game runs on a small keyboard / timer loop where the player can hit certain keys and make monsters spawn in Player 1’s game by sending data through the serial port. Can run the PC Speaker in square wave mode or PWM mode which is only useful in native MINIX.

Developed by: Francisco Lopes, Rui Oliveira (initial draft)

### Logic

Handles everything to do with the game, interrupts, data structs, game state, etc. Keeps the game world in static structs only accessible by the Logic Module (encapsulation) and updates the status of the game world according to what events and interrupts happen. Has a handler for interrupts that calls appropriate functions to deal with the data received and, additionally, a state machine to keep track of Player 1's game, Player 2 doesn't have a state machine as it only has a single helpful state.

Structs (defined in LoLCOM.h):

Cooldown_t – struct used by entities so the game logic knows when to act upon the entities' states

Map_t – struct that holds information about a map, tileset used, array that represents a tile on the tileset and collision data about very tile on the current map

Font_t – struct that holds a font capable of writing to the screen, number array that allows dynamic numbers (up to 999)

Png_t – struct that holds information for writing a PNG image to screen

Entity_t – struct that contains entity data (player, monsters, sword), also has a current state member so that the game logic can recognize what's happening to each entity individually.

Developed by: Francisco Lopes (70%, game logic and data structs), Rui Oliveira (30%, game content production and loading)

## Relative weight of modules

| Module | Relative Weight |
|---|---|
| Keyboard | 6% |
| Mouse | 6% |
| RTC | 4% |
| Speaker | 10% |
| Timer | 5% |
| VBE | 2% |
| Video_gr | 5% |
| UART | 4% |
| LoLCOM | 8% |
| Logic | 50% |

## Permitted code

### Stbi_image.h

Library made by Sean Barret (game developer, API developer) to load every type of image commonly used. Allows to define usage of only one type of image, in the project only PNG loading is used. No changes were made to the file.

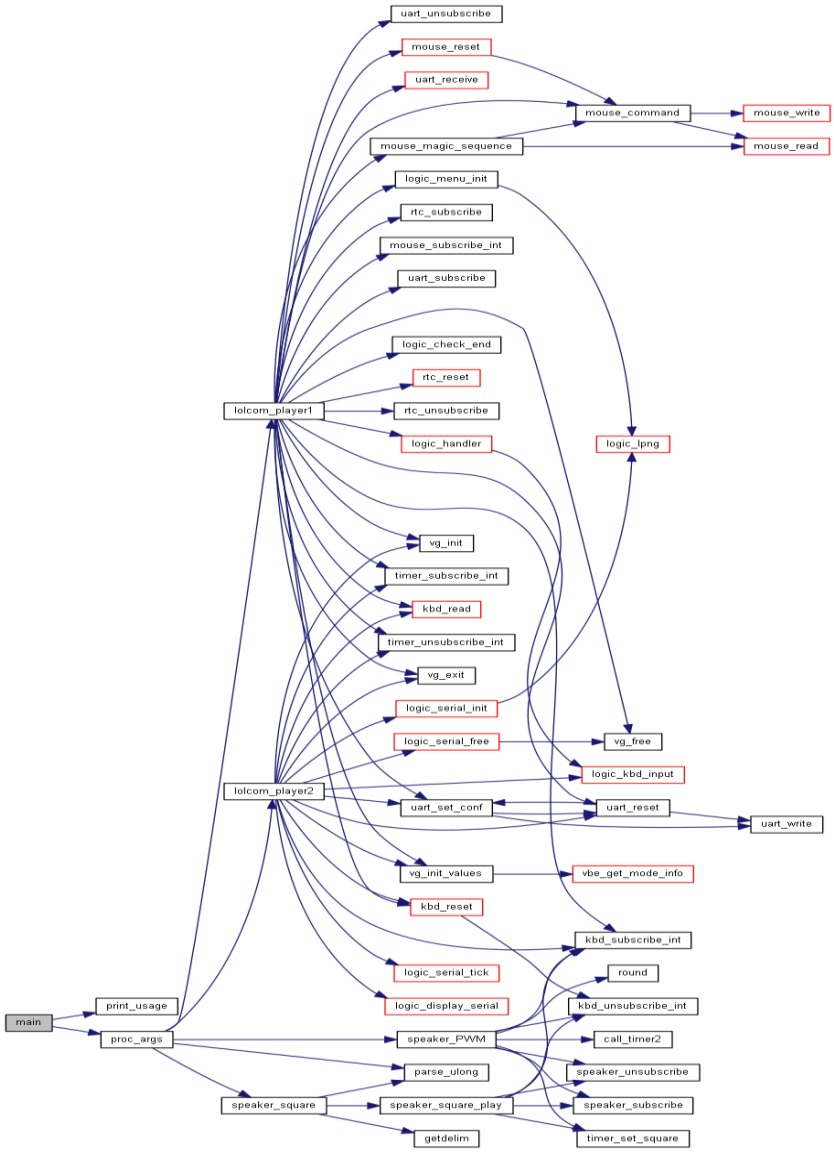
Source: https://github.com/nothings/stb/blob/master/stb_image.h

### Helper

Module that contains helper functions, getdelim() was imported from a source file found on GitHub because MINIX doesn't have getdelim(). Getdelim() reads a file stream up to the specified delimiter. Changes were made so that getdelim() assigned a C string without break lines, carriage returns and the delimiter character, similar to the same function in C++.

Source: https://gist.github.com/ingramj/1105106

## Function Call Graph



# Implementation Details

## RTC

The Real Time Clock was fairly simple to implement. However, during research on the RTC information about a Century Byte was found, supposedly at address 0x32, this was indeed the case in the computers that were tested. Given that, the code reads address 0x32 and checks if it makes sense by verifying if it equals 0x19, 0x20 or 0x21, indicating 20th, 21st and 22nd century respectively.

## UART

The serial port is a lot harder to get right than every other module, as such, and given the scope of the game, a fairly simple version of it was implemented, it transmits when needed and receives by interrupts. While not something that was implemented, it seems the modem lines of the serial cable (RTS, CTS) can be used for hardware flow control through software, there were indeed resources about this specific method but due to time constraints the group wasn't able to pursue it further.

## Page Flipping

Page flipping is briefly mentioned in lab5, trying to use it involved reading the VESA 2.0 specification and quite a bit of experimentation. Function 07h is responsible for setting the start of visible VRAM, as such it can be used to create two or more logical pages that can be swapped (flipped) fast to accomplish hardware "double buffering". Using the option for setting the display start during vertical retrace provided no benefits so it was removed, possibly it only works with CRT monitors.

## 4th Mouse Packet

Reading through OSdev.org pages about the PS/2 mouse unearthed information about a possible 4th packet that mouse devices had at the time, a "magic sequence" of commands had to be issued and after it the id of the mouse should be checked as it would've changed to 3 if it were compatible with the 4th packet. This would've activated the scroll wheel, but there was also the possibility of a 4th and 5th button on the mouse, a second "magic sequence" of commands could be issued and the id checked again for a change to 4, this would mean that the 4th and 5th buttons are now active. As a proof of concept the 4th packet is activated during Player 1's game and the scroll wheel can be used for walking up/down and the 4th and 5th buttons for walking left/right. This method wasn't tested on hardware but it seems that VBox might force all mouse devices to report as being 4th packet compatible, an old PS/2 mouse with only two buttons was tested and it still activated the 4th packet. This could be due to VBox interfering or to the PS/2 > USB adapter, which was active instead of passive.

Source: http://wiki.osdev.org/Mouse_Input

## PC Speaker / MINIX Installing

While using the PC Speaker in square wave mode is trivial, doing so in virtualized MINIX seems to be impossible, as such using the PC Speaker in square wave mode wasn't the challenge itself. Instead, installing MINIX on a computer was the challenge, having that, square wave generation was trivial. The Speaker module uses a text file that contains notes and duration of notes to play music, this way, any music can be programmed for the PC Speaker as long as it contains only one note at a time.

About installing MINIX:

For lack of time to get the proper photos, this will be a textual description.

- Get a computer that can run MINIX, the website has official specs, mainly it should at best be a Pentium 4 without Hyperthreading. It is imperative that the computer has an IDE HDD, be it 2.5" or 3.5".
- Get an IDE -> USB adapter, this will enable us to access the disk without needing old operating systems getting in the way.
- After connecting the IDE HDD to a modern PC running Windows / Linux / Mac, one has to create raw disk access with VBox, this means VBox actually uses the physical disk as storage space for the virtual machine.
- Creating raw disk access can be done by following the instructions here (http://www.serverwatch.com/server-tutorials/using-a-physical-hard-drive-with-a-virtualbox-vm.html) .
- Now just add the new disk to the Virtual Machine, preferably as IDE Second Master
- Inside the VBox Minix use program "part" to check that the disk is indeed there, it should be c0d2.
- Run "dd if=/dev/c0d0 of=/dev/c0d2" this will create a perfect copy of the virtual disk. This step takes a long time but if it fails immediately wiping the partition table of the IDE HDD seems to help. At least on Windows it caused conflicts when being recognized by Windows at the same time.
- Now that the copy is done, mount the partitions of the IDE HDD with the same structure as the virtual disk.
- "mount /dev/c0d2s0p0 /mnt", "mount /dev/c0d2s0p1 /mnt/home" and "mount /dev/c0d2s0p2 /mnt/usr
- Check with "df" that the partitions match in size, "part" here isn't helpful because it will detect that the partition table of the IDE HDD is wrong (unless it happens to be a 2GB one), do not attempt to "fix" the partition table.
- Now it should be plug and play on any old computer, this was tested with a Pentium 4 1.4GHz from around 2000, a Pentium 4 2.8GHz from 2004 and a Toshiba Tecra 9000 which has a Pentium 3 ~900MHz. Two disks were tested, a 2.5" IDE one from the Toshiba and a Maxtor 40GB 3.5" IDE.
- One thing to note, MINIX will try to boot AMD Lance but since it's no longer in VBox there is no such network card, Ctrl + C or Ctrl + D cancels its attempt to connect and skips to the login screen. After that "netconf" can be ran to set network card to null, or even better, to a valid network card, which would allow actual SSH from Eclipse from another computer (this was tested between Francisco Lopes' current desktop and the Pentium 4 2.8Ghz desktop, it works flawlessly and allowed for easy tinkering with the PC Speaker)

Side Note: the bug where SSH closes when copying large files does not happen on an actual Ethernet cable SSH connection but it still happens from VBox to the native MINIX, fairly safe to assume VBox is the culprit.

## PC Speaker Pulse-Width Modulation

A bit of history. In the late 80s discrete sound cards were expensive commodities, as such there was demand for a way of playing digitized music just with the PC Speaker, which almost every computer had. RealSound was a technology invented just for that, it allows controlling the speaker's amplitude of displacement and thus allows 6-bit digitized PCM audio playback. This technology was first used in games on 1988 (World Class Leader Board and Echelon) but with the lowering of prices on discrete sound cards it eventually rendered RealSound obsolete. Another hallmark of PC Speaker PWM was in the DOS port of Pinball Fantasies in 1992 which used the technique to produce an engaging soundtrack, it was a video of this game playing through PC Speaker that alerted to the existence of this technique.

Source: https://en.wikipedia.org/wiki/RealSound

Source: https://en.wikipedia.org/wiki/Pinball_Fantasies

The PC Speaker normally works by generating a square wave with Timer 2, which extends the speaker cone and produces sound, in other words, the speaker only has a resting position and a fully extended position, 0 or 1, no voltage or full voltage. Speaker PWM is based on trying to control the extension of the speaker (the so called amplitude of displacement), with this as basis, the trick is to not let the speaker fully extend, a few websites had tutorial on this technique, the gist of it is using Timer 0 to control the sample rate, and for each Timer 0 interrupt, Timer 2 gets loaded with a new counter value which it will reduce and go high when it reaches 0 (Interrupt on Terminal Count, mode 0), this means the speaker will extend for the fraction of time between Timer 2 going high and Timer 0 resetting the count, thus allowing intermediate values of the extension. Knowing this it becomes a matter of converting music to unsigned 8-bit raw mono PCM and loading it into Timer 2. The catch is that the speaker is only capable of somewhat faithfully reproducing 6-bit audio (higher counts always make it fully extend), so there's a need for a scaling function that takes the 8-bit values and converts them to 6-bit values proportionately.

On the project, 19244Hz was used as sample rate, it allows for 1193181 / 19244Hz = 62 values on Timer 2, any more and Timer 0 will interrupt the count before it ends, this gives almost 6-bit audio, but going any lower would create an audible sine wave overlapping the digitized audio.

While the tutorials aren't hard to follow there's a quirk that makes it seem as if the technique doesn't work, apparently MINIX's Timer 0 handler interferes with the sound from the PC Speaker, this can be seen on a machine running MINIX by changing the frequency of Timer 0 and then eliciting a beep from MINIX, this can be done by pressing delete when the command line is empty. That realization was the crux of getting PWM to work on MINIX, it was already working but the noise overlap from MINIX's own handler interfering made it hard to realize. As such, the solution is subscribing with IRQ_EXCLUSIVE, this has considerable risk on getting MINIX to hang though, tickdelay stops working and keyboard input must be done with a replacement function or ignoring delays, it is also imperative to return control to MINIX or the system hangs and a restart becomes mandatory.

Sources used for programming PC Speaker PWM:

https://web.archive.org/web/20070227143517/http://fly.cc.fer.hr/GDM/articles/sndmus/speaker2.html

http://www.intel-assembler.it/portale/5/Programming-the-PC-Speaker/Programming-the-PC-Speaker-8253-8254.asp

## Conclusion

The group has differing opinions on the course so we found it better to submit our appreciation of it through the self-evaluation forms.

# Appendix

“make” should copy all the resources to the right folder and also copy the config file to /etc/system.conf.d

In case the project doesn’t run:

- Please make sure /conf/LoLCOM is copied to /etc/system.conf_d.
- Please also make sure the entire /resources folder is copied to /tmp at the root level.