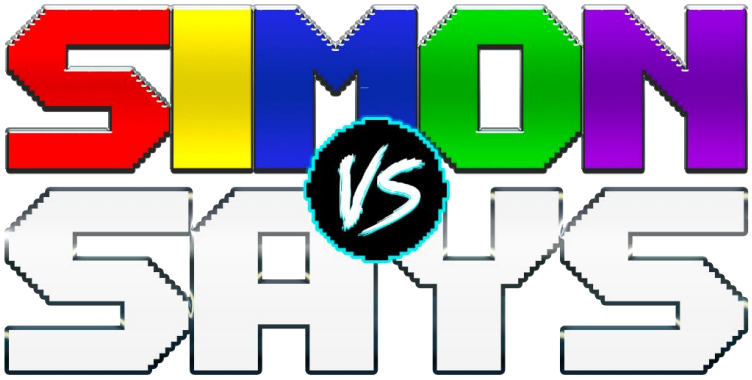
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

COMPUTER LABS



A MULTIPLAYER TAKE ON A SINGLE PLAYER CLASSIC

AUTHORED BY:
HENRIQUE SOUSA
MATEUS SILVA
MELISSA SILVA

CONTENTS

TABLE OF FIGURES

USER INSTRUCTIONS

BOOTING UP THE GAME

To reach the game's menu, all one needs to do is *download* the game's files from the project's *Git* repository. Afterwards, we advise the user to unzip the files into their *shared* folder, wherever they have their *Minix* virtual machine set up in their computer.

There are no special measures needed to be taken in order for the game to run properly — this if the players wish to share a PC/VM —, just boot up your virtual machine with *Minix*, use command `cd` to reach the game's directory, execute `make && depend` and you should be able to reach the game's main menu with the `lcom_run proj` command.

If the players wish to play on separate PCs/VMs, they must do the steps required to setup their virtual machines and connect them before booting up the game.

THE GAME'S MAIN MENU

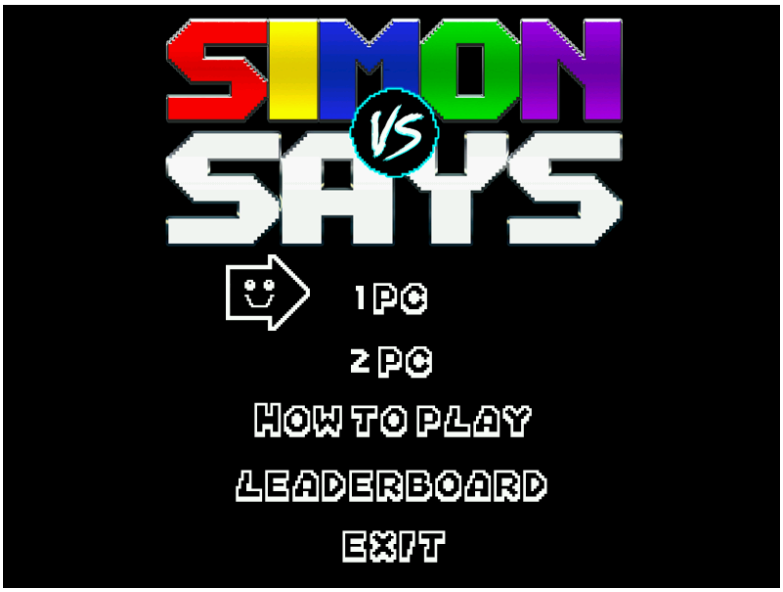


Figure 1 – Screenshot of the Game's *Main Menu*.

The main menu features the game's logo, all the different options the user can access and the *Exit* option, which is our own way to let the user exit the game – as an alternative to this, one can use the *Esc* key to close the game in either screen, working as a *Ctrl + C*[1] of sorts.

[1] This combination of keys is usually used to interrupt a program's execution in many IDEs.

To choose between the available options, the user can use both the *Up* and *Down* arrow keys in their keyboard and the *W* and *S* keys. To confirm a selected option, highlighted with a very happy arrow, the user should press the *Enter* key.

As a simple yet interesting feature, the option selection loops around on itself, meaning that every time we reach the *Exit* option in the menu, pressing the *S* or *Down* arrow keys will select the *1 PC* option. Similarly, doing *Up* or *W* from the *1 PC* option will select the *Exit* option.

HOW TO PLAY

Figure 2 - *Screenshot* of the Game's *How to Play* screen.

The game's *How to Play* screen is pretty straightforward, as it simply lets the player know how to play *Simon Says VS.* and offers a strategy we ourselves deem as a good and captivating one.

Simon Says VS. works as a multiplayer game where the two players must share a computer and mouse to try and prove just how well they can use their memory and how fast they can use it. In case the reader is unfamiliar, the original game simply gives the player a color sequence (usually through lighting up a set of buttons in a toy – see figure 3) that they must repeat by inputting it back.

In our version of the game, the player must memorize their adversary's inputted sequence and repeat it in their turn, which, if done right, grants them the chance to add 1 new element to the sequence. This new and longer sequence must be repeated by the other player, and so on, so forth until one player messes up.

The sequence consists of colors: red, yellow, blue, green, purple and white. Each and every one of these can be repeated as many times as a player wants, as there

is no restriction on which color to input in a player's turn, this, granted, only after they repeat the last accepted sequence. So, it is possible to input red even if all the elements inputted until then are also red.

To add to the fun, we created a special button, the *VS. Button*, which corresponds to the white color mentioned above. By default, the game will recognize this button's input as the color white. However, this changes in turns identified by a randomly generated number, which activates the *VS. Feature*.

If a player inputs the *VS. Button* as the new element to be added to the sequence in one of these lucky turns, they will have five full seconds to add to the sequence as much as they like and with no restrictions – by which we mean a player can simply input the same color over and over if they're feeling merciful towards their adversary.

And yes, after this short interval, the other player will have to repeat not only the sequence prior to the *VS.* press but also whatever was added after it to be able to "survive" his turn. So, as one can tell, being able to use the *VS. Feature* is a sure-fire way to give a big headache to any adversary with a big chance of forcing them to make a mistake and win.

The game gets a whole lot more interesting, however, if this doesn't work as a finishing move; best of luck to any player who find themselves in that situation.

1 PC

This option should only be chosen if the two players wish to share a PC/VM.

Selecting *1 PC* takes the player to the game's *Number of Rounds* screen, where the 2 players should agree on how many rounds they wish to play. We'd like to draw attention to the fact that the number of rounds can differ from the winner's score, this because of a special feature within the game.

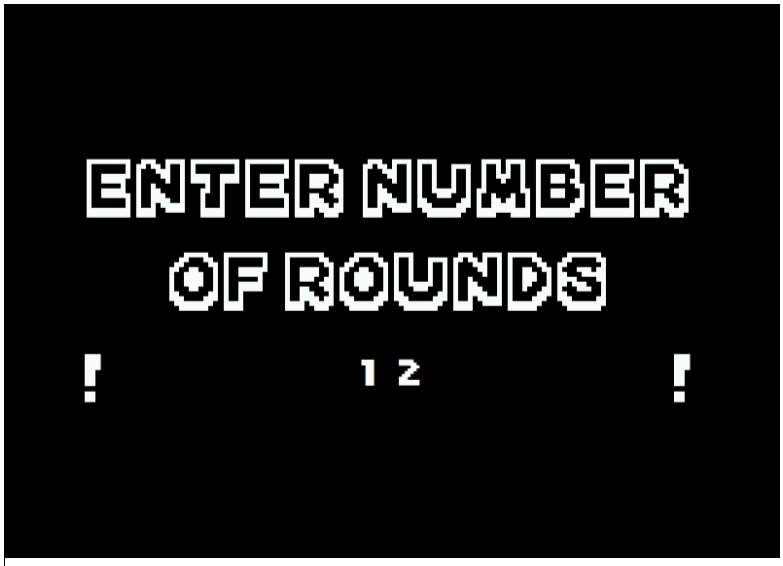


Figure 3 - *Screenshot* of the Game's *Enter Number of Rounds* screen

Input on this screen's text box must be numerical – to help the players, we made it so no letter or symbol keys are accepted in this screen, even if pressed. To confirm the input, one must use the *Enter* key.

After this, the game shows the players its *Default Board*.

Figure 4 – *Screenshot* of the Game's Default Board (*Play* Screen).

The game's board features six buttons, all pressable with the mouse. To keep things intuitive, pressing a button means pressing the mouse's left button with the cursor — also shown in the figure above — hovering over it.

We call this screen the *Default Board* since no buttons are lit up – pressing a button will make it light up. This means the five buttons around the central *VS. Button* will turn into a lighter shade of their default color*;* the *VS. Button*'s lit up version shows a red contour in its text.

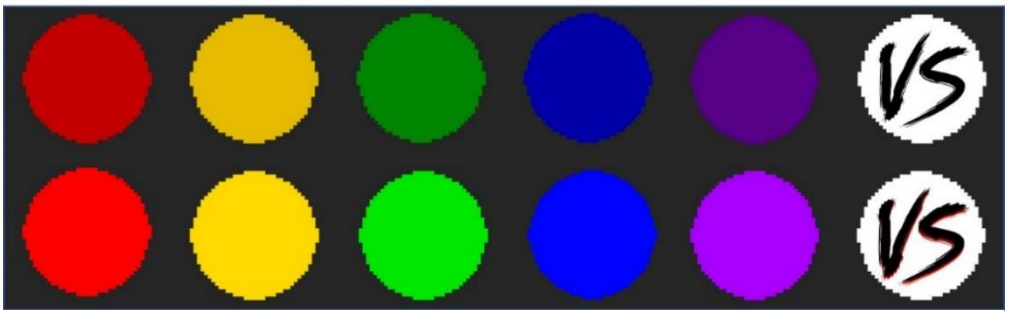


Figure 5 – All the different button variations from the game.

In the upper left corner of the screen, a player's number (either 1 or 2 – who gets to be each of these is up to the players themselves) is shown to tell the player's whose turn is it. Just below this is the game's current score – to show the players what their score will be in case they win.

In the screen's other upper corner, there's a timer – a player has 3 seconds to press each sequence button. However, there isn't any time restrictions for a player to choose the color they wish to add to the sequence.

Whenever a match ends, the game will automatically determine whether there's a winner or a tie. This results in the players being greeted with a *End Screen*, which will be one of the following:

Figure 6 - All the Game's *End* screens – player 1, player 2 and tie, respectively.

If the game reaches one of the first two screens, after the game registers an *Enter* key press, it checks if the final score is higher than the lowest score in the game's

Leaderboard. If so, it takes the player to the *High Score Screen* of the game (see *Leaderboard* section). If not, it'll simply go to the game's *Leaderboard*.

When a game ends with a tie, it'll also simply go to the game's *Leaderboard*.

2 PC

This option should only be selected if each of the two players are in a different PC/VM.

Selecting it takes the player to the game's *Host/Connect* screen, where the two players should select who will host the game (and, therefore, be Player 1) and who will connect to the host (and be Player 2).

The player who selected *Host* is taken to the game's *Number of Rounds* scene, while the other immediately goes to the game's default board. After the host inputs the number of rounds they wish to play, they are also taken to the game's default board, thus allowing the start of the game.

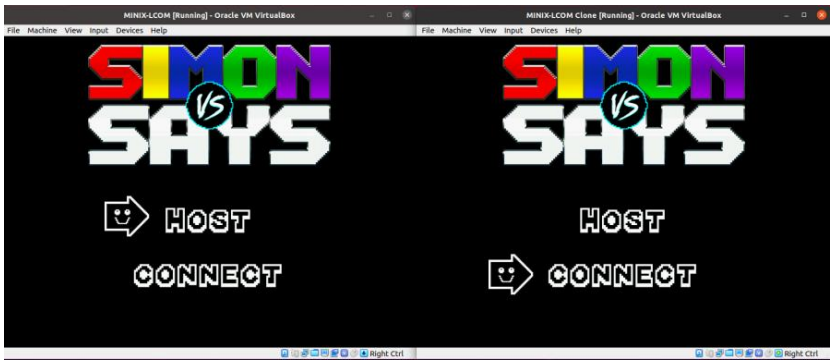


Figure 7 – The Game's *Host/Connect* screen, in two different VMs.

THE LEADERBOARD

Figure 8– *Screenshot* of the Game's *Leaderboard* screen.

The game's *Leaderboard* is quite simple: it features five contemplated winners, identified by name. Along with that is their obtained score, which is the size of the color sequence memorized – for example, *Daisy* was able to memorize a sequence of 202 elements! What a brain! –, and the day when this High Score was obtained.

Though it is only reachable if a player is able to climb into the *Leaderboard*, we'd thought it'd be best to also mention the game's High Score screen, which appears whenever a score of a finished game – which doesn't count ties – deserves to be put in the game's *Leaderboard*.

Figure 9 – *Screenshot* of the Game's *New Hiscore* screen.

This screen's only feature is allowing the winner to input the way they wish to be remembered in the game's *Leaderboard* – as long as they keep it under 10 characters.

PROJECT STATUS

As requested, this section in the project's report will be used to show the reader the final status of the project's implementation, providing information on what devices we used, what these were used for and how we used them[2].

The following table concentrates all the above-mentioned information:

DEVICE	USAGE	INTERRUPTIONS
Timer	Controls the time interval given to a player to press each button.	Yes
Keyboard	User Input (winner name, number of rounds, menu option selection…)	Yes
Video Card	All of the game's graphics and screens.	No
Mouse	Button presses (when playing)	Yes
Real Time Clock	Getting dates to store in the game's *Leaderboard* and getting out of *End* screens with alarms.	Yes
Serial Port	Connecting PCs during *2 PC* game.	Yes.

Table 1 – *I/O* Devices Table**.**

TIMER

FUNCTIONALITY

The *i8254 Timer* is tied to the game's framerate, keeping it at a minimum of 60 *fps*[3] throughout its runtime. It is also used to distinguish two types of input given by the mouse — these being a simple left button press and a left button hold.

As expressed above, this device also keeps track of the three second interval during each a player must press one button.

IMPLEMENTATION

The implementation of the device works with a counter, which is reduced every time a 1/60th of a second passes – this results in a screen refresh, which keeps the game in its minimum framerate. This counter is reset with 3 seconds (or 180 ticks) once a button in the sequence is correctly pressed.

KEYBOARD

FUNCTIONALITY

As stated in the table, the keyboard is used to obtain direct input from the winner's chosen name in the game's *New High Score* screen, as well as a number in the *Number of Rounds* screen. Along with this, the *Up* and *Down* and *W* and *S* keys

[2] This pertains to the technique used in the device's implementation: for example, polling or interruptions.
[3] *Frames per second*.

are used in the games main menu to select one of its options, which can be confirmed with an *Enter* press.

Besides that, in the *How to Play* and *Leaderboard* screens, pressing the *Backspace* key makes the game return to its *Main Menu*.

IMPLEMENTATION

Most of the functions made for this device were repurposed from the subject's *Lab 3*. The main addition is the function `keyboard_get_scancode`, which was used, as the name suggests, to get the *scancode* (both *make* and *break* codes) of each key pressed and released on the keyboard.

This function was used to obtain the desired keys' *scancodes*, to create two *enum* types to help with the device's implementation: one for all the keyboard's keys *breakcodes* and the other for any relevant special keys' *makecodes* (for example, *Up* and *Down* arrows, *Enter*, *Backspace* and *Esc).* This function solved a problem we had while implementing the keyboard, as it let us get codes for the special keys, which we couldn't get otherwise – this excludes the *Esc* key, as it was provided by the teacher during *Lab 3* classes.

Another important addition was the `assemble_key_press`, as it detects all key presses and returns an adequate instance of the `key_press` *struct,* which includes the *char* code of the pressed key and specifies whether it is a *make* or *break* code.

VIDEO CARD

FUNCTIONALITY

We consider this device the most important one of the game, as it's responsible for drawing the entire game console, which is, obviously, needed for the player to be able to interact with it. All the game's graphics were converted to the *XPM* format from a *PNG* origin – we used to do this rather tedious part of the project.

Figure 10 – *GIMP's* application icon.

The mode we picked for the game is the ***0x114 VBE Mode***, which encompassed a ***800x600 resolution*** with ***16 bits per pixel*** and ***direct color mode*** **(5_6_5)**.

IMPLEMENTATION

To initialize the graphic mode, we adapted *Lab 5* functions `get_mode_info` and `set_vbe_mode`. When implementing this device, we found an issue we could not solve with the first mentioned function, which is our version of `vbe_get_mode_info`, a function provided by teachers in *Lab 5*.

We believe this problem was related to the usage of function `lm_alloc`, as the *Computer Labs: Lab 5 – VBE Function 0x01: Return VBE Mode Information* slide presentation states it might cause an issue since it often *fails to allocate the desired memory*. Upon trying the provided possible solutions, we unfortunately found that they weren't permanent and stopped working further into the project's development.

Our solution involves initializing the game with the mentioned `vbe_get_mode_info` when we first boot *Minix* and then switching back to our function.

As we experienced movement lag in the mouse cursor movement, we decided to implement ***double buffering***. We went about it by copying the screen's graphics to a secondary buffer we called `buffer`, while we named the first and main one `video_mem`.

We have two different *draw* functions: to *draw* backgrounds, we call `draw_xpm`, which simply uses `memcpy` to copy the entire image array to the buffer in one fell swoop and we also have `draw_xpm_x_y`, which is used to display numbers and letters, which have 2 different sizes (either 42x44 or 19x28) and are copied along with the mouse cursor once again by using double buffering.

For the *High Score* and *Enter Number of Rounds* screen, which require user input, we designed various functions that, together, form text boxes to display the user's current input as it changes.

MOUSE

FUNCTIONALITY

The mouse is only used in the context of a match, letting the players selects buttons in their turns to try and repeat the last accepted sequence. Thus, we use the mouse movement and its left button.

IMPLEMENTATION

Most of the functions you'll find in our *mouse* module were repurposed from *Lab 4* with very few to no adjustments, as they worked as is. However, the mouse module isn't the only set of files where we deal with mouse input.

In our *game* module, we made `game_update_status_mouse`, which tracks the cursor's position and left button presses while also dealing with overflow and keeping the mouse inside bounds, so it isn't possible to move it outside the game's screen/display. If a left button press is detected, it checks if the cursor is hovering over one of the board's buttons through other functions in the module.

REAL TIME CLOCK

FUNCTIONALITY

This device is used to get the date at which a new High Score is obtained. This date is then stored in the game's *Leaderboard* data structure and displayed on screen.

It is also used to set up an alarm of 5 seconds to change from the *End* screens of the game to the *Leaderboard* or *High Score* screens — the latter depending on whether a new high score was reached.

IMPLEMENTATION

The device works, for example with function `read_date`, that reads the current date in its entirety and transfers this information to a *struct* which represents a date. Another function, `bcd_to_int`, converts the format in which the RTC stores a date to the usual integer format – we consider it a helper function.

There's also `set_timed_alarm`, an essential function for the setup the alarms mentioned above, called after `enable_alarm_int`, which enables alarm interruptions and `disable_updates`, that disables updates. Finally, there's `disable_periodic_int`, which disables periodic interruptions.

SERIAL PORT

FUNCTIONALITY

To execute the connection between two machines, we use the serial port with interruptions. Both machines can receive and send data, depending on whose turn it is.

Everything that happens in one machine happens at the same time in the other; both machines are taken to the *High Score* screen so the winning player gets recorded in both machines.

IMPLEMENTATION

As mentioned, the device works with interruptions from both machines, managed by `uart_handler`.

Other important functions are `check_thr_empty`, which checks whether the Transmitter Holding Register is empty, this to send information between machines and `serial_write_byte`, which, in turn, writes the information we wish to transmit.

CODE ORGANIZATION/STRUCTURE

In this section, we'll go a little more into the technical aspects of the project's development. Along with this, we hope to provide information on how work was distributed between each member of our group.

To preface it, we'd like to ask the reader to consider that a member's participation in a module must be considered as the maximum – this means that if *X* and *Y* are the sole participants in a module, then 50% of it was done by each one.

GAME

FILES: `game.h, game.c`
WEIGHT IN PROJECT: 9%
IMPLEMENTED BY: Henrique, Mateus, Melissa

This is quite possibly the most important module in the whole project; thus, it has one of the highest weights we attributed. This file features the entire game logic, along with the most important functions — for example, the function we mentioned in Section 2.

Given this module's magnitude, all members of the group worked in it. Without a doubt, the most important functions are `game_update_status_timer` and `game_update_status_mouse`, as they manage a major part of the game's logic while keeping it running and going forward.

Regarding data structures, we have the *enum* `button_click_t`, which encompasses all buttons of the game; *enum* `Play_Screens`, with all of the game's boards; *enum* `Winners`, which includes all of the game's *End* screens and, finally, the most important structure of the game, *struct* `mouse_state`, which we have mentioned in Section 2.

All of these data structures and functions are documented in Doxygen.

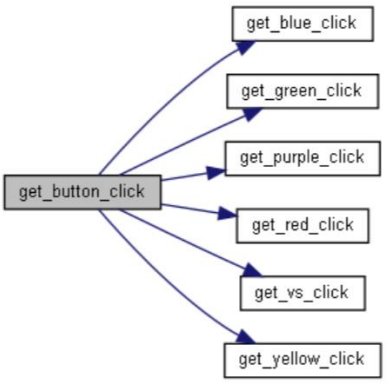


Figure 11 – Function Call graph for `get_button_click`, used to distinguish which buttons are pressed.

GRAPHS

FILES: `graphs.h`
WEIGHT IN PROJECT: 2%
IMPLEMENTED BY: Mateus

This module comprises the game's graphics, thus including all the graphs' XPM images used in the game. However, it was a little challenging to figure out how to do it, as we'd never seen examples on how to fulfill such task. The file was coded solely by Mateus, yet in making the game's original PNG graphics and its conversion, he had Melissa's help.

HIGH SCORE

FILES: `High Score.h, High Score.c`
WEIGHT IN PROJECT: 5%
IMPLEMENTED BY: Mateus, Melissa

High Score, *Game* and *Leaderboard* behave like a team, in our code. As we mentioned, there's a possibility that the winner might have a bigger score than the lowest score in the leaderboard. If this happens, *Game* will call *High Score* through jumping states on a state machines – which we'll discuss in Section 4.

High Score mainly used features from other modules, such as *VC* and *Keyboard*. However, it features a link with the *Leaderboard* and *Real Time Clock* modules through function `getDate`. One of the most challenging things about this module was trying to figure out how to properly get the winner's input and change it as needed — this because we weren't used to work with C's strings, which we found notably hard to change. To get past this, we used `strncat` and `memmove` from the *`string.h`* library.

The module mostly uses extern data structures. A very important one would be `new_hiscore`, a `text_box` *struct* used, as its name says, as a text box for the winner's chosen name input. It's created with the help of the *Video Card* module.

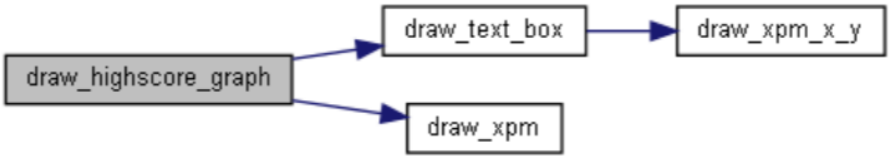


Figure 12 – Function Call graph for `draw_highscore_graph`, which displays the *High Score* screen.

HOW TO PLAY

FILES: `how_to_play.h, how_to_play.c`
WEIGHT IN PROJECT: 4%
IMPLEMENTED BY: Melissa

We considered this module rather important as it presents the game's instructions to the player in a straight forward way, which is needed for players hoping to get to know the game's rules before experiencing for the first time.

It has two functions, `draw_how_to_play_graphics`, which connects it to the *Video Card* module and `how_to_play_update_status`, that uses the *Keyboard* module's functions.

Figure 13 – Function Call graph for `draw_how_to_play_graphics`, which displays the *How to Play* screen.

I8042

FILES: `i8042.h`
WEIGHT IN PROJECT: 2%
IMPLEMENTED BY: Henrique, Mateus

This module was simply repurposed from *Lab 2*, with some adaptations to help ourselves in its usage. It features all the *macro* constants needed to program the *Timer* module.

I8254

FILES: `i8254.h`
WEIGHT IN PROJECT: 2%
IMPLEMENTED BY: Henrique, Mateus

This module was simply repurposed from *Labs 3* and *4*, with some adaptations to help ourselves in its usage. It features all the *macro* constants needed to program the *Keyboard* module.

KEYBOARD

FILES: `keyboard.h, keyboard.c`
WEIGHT IN PROJECT: 4%
IMPLEMENTED BY: Henrique, Mateus, Melissa

This module is of utmost importance in our game, as it deals with user input — except for the mouse's left button presses — but also helps with alerting when it's time to change screens via specific key presses, as we mentioned before.

Most of the functions from this module were imported from *Lab 3*, excluding, however, `keyboard_get_scancode` and `assemble_key_press`. There are three highlights in this module's data structures: *enum* break_codes, that contains the *break* codes from all *keypresses*; *enum* `special_keys`, which keeps the *char* codes for some special keys and is used in various modules to detect when it's time for a screen change and, finally, *struct* `key_press`, which stores the data from a keypress and is also used in various modules.

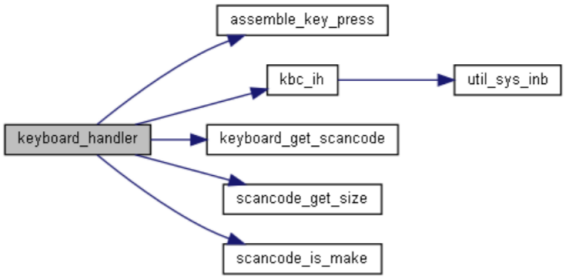


Figure 14 – Function Call graph for `keyboard_handler`, which deals with *Keyboard* interruptions, for example.

HOST

FILES: `host.h, host.c`
WEIGHT IN PROJECT: 3%
IMPLEMENTED BY: Henrique, Mateus, Melisa

This module is extremely similar to *Menu*, except it lets the users choose if they want to be the host or connect to a game. It also features two functions tied to another state machine.

It includes the *enum Host_Selection*, which contains each screen to be printed based on the host's arrow position.

LEADERBOARD

FILES: `leaderboard.h, leaderboard.c`
WEIGHT IN PROJECT: 9%
IMPLEMENTED BY: Mateus, Melissa

Leaderboard is a very special module, and it was surprisingly challenging to develop, as well. Besides interacting with the *Video Card*, *Keyboard* and *High Score* modules, it is the only module in the whole project that deals with a *.txt* file: *leaderboard.txt* — because of this, it's also the first module to be called upon by *Proj* — and also the only module whose screen can be reached through multiple ways.

saveData and *retrieveData* proved very difficult to implement, as *Minix* didn't support the functions one would hope to use to deal with file reading and writing. To replace those, we soon found *fscanf* and *fprintf*, as well as *memcpy* which was incredibly useful in moving information between data structures, which soothed many headaches.

It also features one of the most important data structures of the game, namely *struct* cont_winner, which is used to save a *Leaderboard*-contemplated High Score.

Figure 15– Function Call graph for `retrieveData`, which only calls `loadDummySt`, a function that merely sets the parameters of a *dummy* `cont_winner` struct.

MENU

FILES: `menu.h, menu.c`
WEIGHT IN PROJECT: 3%
IMPLEMENTED BY: Henrique, Mateus, Melissa

We consider this module essential to the game since it lets the users reach many of the other game screens. It features two functions deeply tied to a state machine, which we'll discuss later in this report. It's connected to the *Video Card* and *Keyboard*, and we'd like to mention how proud we are of the `menu_update_status` function, for a multitude of reasons, one being the fact we were able to loop the selection around its extremes with both the `Up`, `Down`, `W` and `S` arrow keys.

It includes the *enum* `Menu_Selection`, which contains each screen to be printed based on the Menu's arrow position.

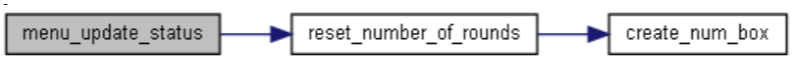


Figure 16 – Function Call graph for `menu_update_status`, our favorite function from this module.

MOUSE

FILES: `mouse.h, mouse.c`
WEIGHT IN PROJECT: 3%
IMPLEMENTED BY: Henrique, Mateus

The entirety of this module was repurposed from work done for the course's *Lab 4* practical work. However, it holds its weight in this project, as it's the only way the users can interact with the game's board.

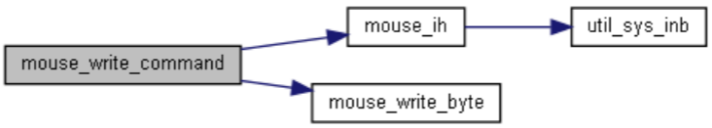


Figure 17 – Function Call graph for `mouse_write_command`.

MULTIPLAYER GAME

FILES: `mp_game.h, mp_game.c`
WEIGHT IN PROJECT: 10%
IMPLEMENTED BY: Henrique, Mateus, Melissa

This module is, to put it simply, a modified version of the *Game* module, which adapts the game logic to make it fit the usage of two machines, as opposed of just one.

Like *Game* and all the other modules involving the serial port, every member of the group worked in it, given its difficulty. Also, and in similarity with *Game*, `mp_game_update_status_mouse` is very important, as is `mp_game_update_status_uart`, which is responsible for management of the serial port and communication between machines.

While the *Timer* plays an important part in dealing with time management in *Game*, this implementation of the game leans more heavily on the *Real Time Clock*, giving it most of the roles fulfilled by *Timer* in *Game*.

Regarding its data structures, they're the exact same as *Game*'s. Thus, we'd like to redirect your attention to functions we created, namely `mp_remote_click_button`, which handles the buttons clicked by the connected player; `host_mp_game`, that initiates the game for the host, and `connect_mp_game`, which connects the second player to the game created by the host.

Figure 18 – Function Call graph for `mp_game_update_status_rtc`, another important function.

NUMBERS AND LETTERS

FILES: `num_and_letters.h, num_and_letters.c`
WEIGHT IN PROJECT: 3%
IMPLEMENTED BY: Henrique, Mateus, Melissa

Numbers and Letters is necessary to display the player's input and to display the information present in the *Leaderboard*. To be honest, the main struggle here wasn't really the coding but the making of the graphs, as we needed to remake them several times to ensure the best conditions for the players — namely, the maximum space possible for their input.

In this module, we essentially load both sets of numbers and letters to `xpm_image_t` arrays: `letters[28]`, which stores letters, *char* space and a *char* called `empty`, which we used to help us display the removal of input through the

Backspace key; `numbers[10]`, to store numbers of the same size; `lilletters[27]`, which stores letters and a / bar for the dates in the *Leaderboard* and, finally, `lilnumbers[10]`, which stores numbers with the same size as the last mention array: 19x28.

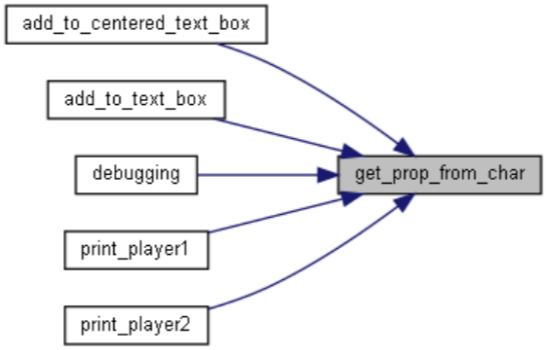


Figure 19 – Function Caller graph for `get_prop_from_char`.

NUMBER OF ROUNDS

FILES: `number_of_rounds.h, number_of_rounds.c`
WEIGHT IN PROJECT: 3%
IMPLEMENTED BY: Henrique, Melissa

Number of Rounds preludes the actual game, since when players select *Play* in the game's main menu, they're taken to this module's screen. It's similar yet simpler than *High Score*, as we only handle input of integer digits instead of *C* strings.

It is connected to *Video Card*, through use of a `text_box` *struct* named `num_rounds` and to *Keyboard*, through the input management and the game itself. As it was designed to get the number of rounds the players would like to have in a match, this value is stored in an `unsigned int` variable, `num_rounds_value`, later used in the *Game* module.

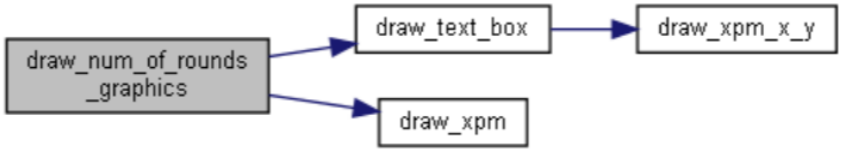


Figure 20 – Function Call graph for `draw_num_of_rounds_graphics`.

REAL TIME CLOCK

FILES: `rtc.h, rtc.c`
WEIGHT IN PROJECT: 5%
IMPLEMENTED BY: Henrique

The *Real Time Clock* or *RTC* is one of the extra devices introduced by the teacher after the end of our *Lab* classes. At first, it was only used to get dates to improve our Leaderboard, which was rather lacking in information with just a winner's name and their score.
However, the real difficulty started when we decided to set up alarms as a way to exit the *End* screens, which we'd have to admit caused a lot of problems due to

lack of information. Then, its implementation wasn't as challenging as we initially thought.

We have two data structures in this module: an array of `int` variables, `monts_no_days[12]`, which contains, as the name suggests, the number of days in each month and the *struct* `date`, created to store a date.

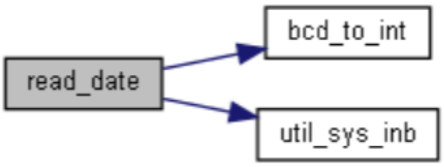


Figure 21 – Function Call graph for `read_date`.

SCREEN

FILES: `screen.h, screen.c`
WEIGHT IN PROJECT: 3%
IMPLEMENTED BY: Henrique, Melissa

This module's code is very similar in difficulty to *Graphs*, and it is also connected to it. The real difficulty was figuring out how to load all the images included in the module graphs to a data structure.

Screen has a very important data structure, which is externally accessed by a lot of other modules: `Screen`. It keeps all the information in regard to all of the game's screens, via a `xpm_image_t` array that stores every single game screen and an *enum* `Game_Modes`, which contains the various modes of the game based on what screen should appear.

Figure 22 – Function Caller graph for `loadscreens`.

SIMON

FILES: `simon.h, simon.c`
WEIGHT IN PROJECT: 5%
IMPLEMENTED BY: Henrique, Mateus, Melissa

Simon is responsible for the project's workflow, featuring our only driver_receive cycle and handling of all the interruptions from the various devices used. We also initialize the game's graphic mode and switch back to text mode here.

It possesses, quite possibly, the most important function in the whole game: `event_handler`, that deals with a state machine and all the calls to the *Mouse*, *Keyboard* and *Timer* status functions. It's also here that we call a very important function to the process of double buffering: `copy_to_video`.

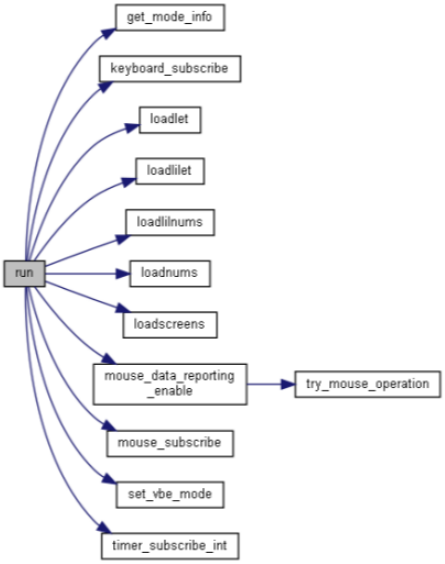


Figure 23 – Function Call graph for `run`, that does a lot of the heavy lifting mentioned.

TIMER

FILES: `timer.h, timer.c`
WEIGHT IN PROJECT: 3%
IMPLEMENTED BY: Henrique, Mateus

Just like the *i8042* module, *Timer* was mostly repurposed from *Lab 2*, with any changes needed to fit into the project's design.

UART

FILES: `uart.h, uart.c`
WEIGHT IN PROJECT: 9%
IMPLEMENTED BY: Henrique, Mateus, Melissa

The *UART* or *Universal Asynchronous Receiver/Transmitter* device controls the serial port, constituting the second extra device introduced by the teacher after the end of our *Lab* classes.

It was as challenging to implement as we first imagined, especially due to the lack of possibility of debugging. This module is essentially a helper module to *Multiplayer Game,* where its functions are used to make the game playable using two different machines.

The most important functions are `check_thr_empty` and `serial_write_byte.`

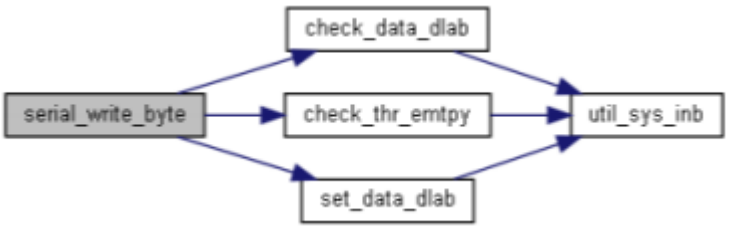


Figure 24 – Function Call graph for `serial_write_byte.`

UTILS

FILES: `utils.h, utils.c`
WEIGHT IN PROJECT: 2%
IMPLEMENTED BY: Henrique, Mateus

Just like *Timer*, it was mostly repurposed from the work done for *Lab 2*, with any changes needed done.

VIDEO CARD

FILES: `vc.h, vc.c`
WEIGHT IN PROJECT: 9%
IMPLEMENTED BY: Henrique, Mateus, Melissa

Video Card is one of the most complex modules in our game, losing first place to *Game*. We started by importing most of the code from *Lab 5*; however, due to many issues that came up, we redeveloped most functions, this to be able to implement double buffering.

Input management was made smoother with functions like `create_writeable_text_box`, `add_to_text_box`, `draw_text_box`, `add_to_centered_text_box`, `erase_text_box`, `create_static_text_box` and `remove_from_centered_text_box`. All these functions work to create a *struct* text_box used in various modules — for example, *High Score* and *Leaderboard*.

We have several variable in this module, mainly thanks to `get_mode_info`, responsible for obtaining all the information needed from the graphic mode — to name a few examples: `bppixel`, an `unsigned int` variable to store the value of bits per pixel and `*video_mem`, a `static void` pointer used as the frame-buffer VM address.

IMPLEMENTATION DETAILS

GAME LOGIC

In the game, when a mouse package is received, it updates the state of a mouse global variable that contains the mouse's position and whether its left button was pressed. Upon a click, we check if the player has pressed one of the buttons through function `get_button_click`. If that's true, then the respective button will be lit up in the next frame, this through changing the global variable `ps` in the `light_button` function.

All the game logic regarding the memorization of the sequence and the players is done in `click_button`. While the previous sequence isn't completed, the player must click the buttons in the same order, having 3 seconds for each button press. In case he fails to do so, he will lose. However, if he successfully pulls this off, he can insert one more button into the sequence.

As explained previously, if he happens to press the white button in a special round (which is kept in check with function `is_VS_round`), then the player can input any number of buttons for the following 5 seconds. Afterwards, the players will change, and the logic simply repeats.

At the end of the game, while in the *End* screens, there's an alarm set up, that will change to the next screen automatically after 5 seconds have passed, or the player can press **Enter**.

STATE MACHINES

We have several state machines in different modules.

The first one we created was `Game_Modes`, from *Screen*. This *enum* contains the various modes of the game and is used to know which screen and module should be called next, proving itself to be incredibly important to control the game's flow. It exists as part of the struct Screen, also defined in the module mentioned and present in almost every module. The management of this state machine is present in function `event_handler`, from *Simon*.

Similarly, we have a state machine, once again as an enumeration type, called `Menu_Selection`, in *Menu*. It features the same function we just mentioned but, instead of all game screens, it simply handles the four menu screens, one for each position of the pointer arrow. Its handling is done through `draw_menu_graphics`, also from the *Menu* module.

In the *Host* module, we have an enumeration type, called *Host_Selection*, very similar to *Menu_Selection*. This, however, only handles the two *Host* screens: *Host* and *Connect*. Its handling is done through `draw_host_graphics`, also from the *Host* module.

The *Game* module features three different state machines: `Play_Screens`, containing all the screens that should be printed depending on the pressed button; `Winner`, which encompasses all the *End* screen of the game (one for a Player 1 win, one for a Player 2 win and a final one for a Tie) and, finally, `button_click_t` which checks what button from the board was pressed. The first two are very much like `Game_Modes` and `Menu_Selection`. The third one simply exists to help change `Play_Screens`' mode during the game.

`Winner` is handled in function `draw_winner_graphics`, while `Play_Screens` is mainly handled in `draw_buttons` and `button_click_t` is dealt with inside function `light_button`.

EVENT DRIVEN CODE

One aspect we wish to highlight again is the use of only one `driver_receive` cycle, done in function `run` from module *Simon*. We also managed to structure our code in a way where only one function manages interruptions from all devices and updates the game's status with the player's actions — mouse movement and clicks and the keys pressed — using the very originally named function `event_handler`.

Other than that, we used all the state machines mentioned above to better develop robust event driven code.

VIDEO CARD PROBLEMS

As stated before, we faced many issues with the *Video Card*.

We started off with functions developed in *Lab 5*, however we immediately started having lag problems when trying to display letters and numbers while the player was inputting them through keyboard: the corresponding graphics wouldn't stay static on screen, always flickering.

At the time, we used a rather archaic solution: only printing the background graph once instead of each time a keyboard press was detected, as it was the case until then. However, the issues reappeared when implementing the mouse, with apparently worse lag than before. We tried many approaches and ideas to no avail, until we realized we'd have to rework the *Video Card* implementation to add double buffering.

This fixed the letters and numbers issue in a more elegant way, but somehow made the mouse's problem quite peculiar: while on Henrique's two computers and FEUP's computers the lag had completely disappeared, Melissa's and Mateus' PCs still experienced the mouse lag.

We tried to single out the issue and go at it in full force, asking for assistance from our practical classes teacher, Sara Fernandes, and our theory classes teacher, Pedro Souto, as well. Though we considered the problem solved when Henrique found he didn't experience it, since not all members could use the mouse without lag, its testing was rather difficult.

DOUBLE BUFFERING

To implement double buffering, we chose to create two different functions and leave the decision of which one to call be affected by what we wished to display on screen. We have `draw_xpm`, which we used to strictly draw background graphics, since these are the only ones that cover the game's entire screen: it essentially copies all the desired images' bytes to our secondary buffer, `buffer`.

```
int(draw_xpm)(xpm_image_t screen)
{
  memcpy(buffer, screen.bytes, x_res * y_res * bytes_per_pixel);
  return 0;
}
```

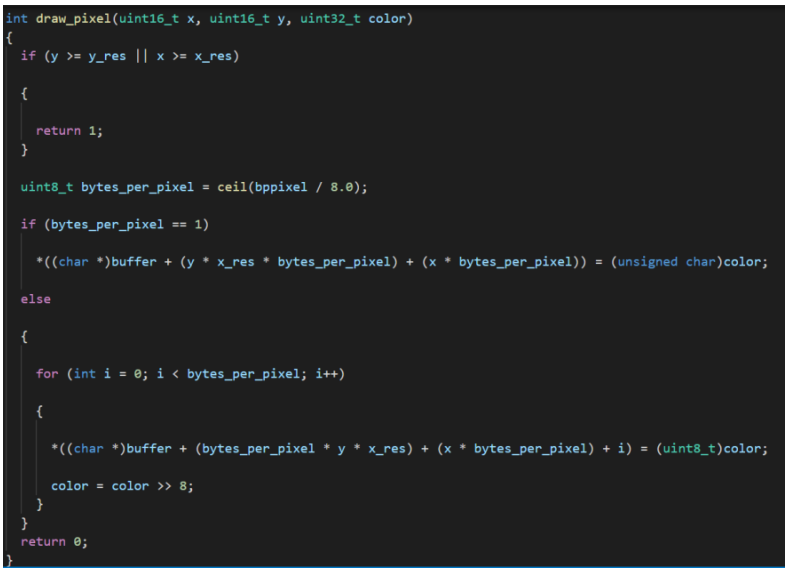
Figure 25 – *Screenshot* of `draw_xpm`'s implementation.

Another function we possess is `draw_xpm_x_y`, which is used to draw both sets of numbers and letters, along with the mouse cursor.

```
int draw_xpm_x_y(xpm_image_t img, uint16_t x, uint16_t y)
{
  int bcount = 0;

  for (int i = y; i < img.height + y; i++)
  {
    for (int j = x; j < img.width + x; j++)
    {
      if (((uint16_t *)(img.bytes))[bcount] != xpm_transparency_color(img.type))
        draw_pixel(j, i, ((uint16_t *)(img.bytes))[bcount]);
      bcount++;
    }
  }

  return 0;
}
```

Figure 26 – *Screenshot* of `draw_xpm_x_y`'s implementation.

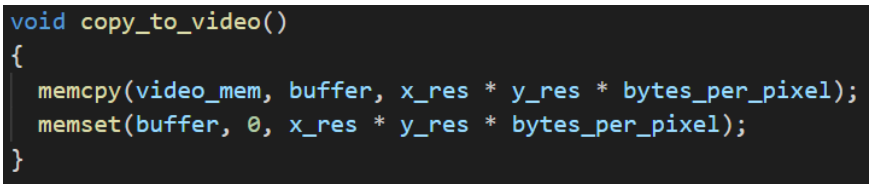
This function calls `draw_pixel`, which enacts our strategy to draw all our graphics: doing it pixel by pixel.



```
int draw_pixel(uint16_t x, uint16_t y, uint32_t color)
{
  if (y >= y_res || x >= x_res)

  {

    return 1;
  }

  uint8_t bytes_per_pixel = ceil(bppixel / 8.0);

  if (bytes_per_pixel == 1)

    *((char *)buffer + (y * x_res * bytes_per_pixel) + (x * bytes_per_pixel)) = (unsigned char)color;

  else

  {

    for (int i = 0; i < bytes_per_pixel; i++)

    {

      *((char *)buffer + (bytes_per_pixel * y * x_res) + (x * bytes_per_pixel) + i) = (uint8_t)color;

      color = color >> 8;
    }
  }
  return 0;
}
```

Figure 27 – *Screenshot* of `draw_pixel`'s implementation.

`draw_pixel` copies all pixels to our secondary buffer (`buffer`). After each game state change, `event_handler`, from *Simon*, calls `copy_to_video`.



```
void copy_to_video()
{
  memcpy(video_mem, buffer, x_res * y_res * bytes_per_pixel);
  memset(buffer, 0, x_res * y_res * bytes_per_pixel);
}
```

Figure 28 – *Screenshot* of `copy_to_video`'s implementation.

This last function copies the contents from the secondary buffer to the main one (`video_mem`), through use of `memcpy`. It then sets `buffer` to zero, in order to clean it and thus making it ready to get new content.

SERIAL PORT AND MODIFIED GAME LOGIC

To use the serial port for the implementation of multiplayer gameplay, we had to make a few changes in our game logic.

Though most of what's stated in the GAME LOGIC section still stands here, there are some changes we'd like to highlight: for example, in order to memorize the sequence each player does, besides using the function `click_button`, we also have `mp_remote_click_button`, which essentially does the same as `mp_click_button`, an adaptation of the function we first mentioned. The difference lies in what they manage: `mp_remote_click_button` only deals with the second player's sequence, while the other deals with the host player's sequence.

We also have a function, `mp_buttons_handler`, that manages the player's clicks. While in the 1 PC version of the game both players share the screen and thus play using the same machine, this isn't the case here. We ultimately had to create a function which would stop a player from inputting buttons whenever their turn wasn't up.

Regarding the serial port, we had to develop several functions whose job was to send information to the other machine, be it who was the winner of a match or a tie between the players; buttons clicked in a player's turn (to light them up in the other machine as well); a signal to tell a player it's their turn or that the current round is a *VS* round.

CONCLUSION

Through this project's development, we faced *a lot* of different issues which somehow kept coming back to *Minix*'s own inner workings. We managed to solve some of them, while others — like the mouse lag — only reached a sort of *partial* resolution. Dealing with *Minix* made developing and testing extremely difficult, making us lose time with what could be considered frivolous details. For example, we really don't like how we can't print and draw graphics at the same time.

Having the *Labs* to fall back on helped quite a bit in saving time, as many of our implemented devices had already been developed during their respective classes — with the sole exception of the *Video Card*, which took a distinguishable amount of reworking.

It was fun to develop this game, even if working on it sometimes induced long and extreme states of stress. Working together helped a lot with this, as there was always one of us to help the others get back up on their feet, all resulting in an extremely gratifying sensation when finishing our first ever game that makes use of various PC components instead of just the keyboard's input, as we did in *PROG* and *AEDA*.

We must express our gratitude and thanks to our practical teacher, Sara Fernandes, who always provided quick and efficient responses to every time we asked for assistance in development.

On another hand, we were told it'd be appreciated if we evaluated *Computer Labs* as a course in this section along with our final thoughts on the project.

In all three semesters we've had until now, *Computer Labs* (or *LCOM*, its Portuguese acronym) was, without any shadow of a doubt, the most difficult course we've ever had to deal with. It pains us to say this, but we also feel all the frustration and stress could've been avoided, as we can affirm that we now have a reasonable understanding of what we were supposed to learn — without it, we wouldn't have been able to do this project.

Besides *Minix* itself, the difficulty of the course lies on the lack of material provided in classes, specially given for the *Lab* activities, as we feel they leave a lot of details implicit that could save us hours upon hours of work that do not grant us a *Test Succeeded* message from the *LCF*. Or, when those details are included in the material provided, a lack of organization in its content keeps us from noticing them.

One suggestion we'd like to propose is having provided tests for the devices we don't cover in classes, namely the *RTC* and *Serial Port*, just like we have for the

rest of the devices. This would help check if our code was working correctly before adding them to a project's source code. This is true especially for the serial port, whose debugging was insanely difficult and made us spend a lot of hours we feel weren't, in fact, needed.

LCOM, however, is definitely an interesting subject and one where we first learned how to deal with several hardware devices through low-level programming — and it was always *very* rewarding to see our code was working.

APPENDIX

Before executing the game, please check if the path file for *leaderboard.txt* is the correct one for you PC. The path file can be changed in `leaderboard.c`, in lines 176 and 188.

Linux's path is usually *home/lcom/labs/proj/leaderboard.txt*.