TYPE RACER

LCOM Project

Authored by:

Diana Freitas
up201806230

André Gomes
up201806224

January 6
2019/2020

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Table of Contents

Table of Figures

Goals

Specific course goals

All the goals of the course were met, namely:

- Use of the hardware interface of the most common PC peripherals;
    - Timer, Mouse, Keyboard, Video Card and Real Time Clock
- Development of low level and embedded software
    - Use of low level functions and registry access
- Programming, in a structured way, in C language
    - Use of one main interrupt while loop, data structures, documentation
- Use of various software development tools
    - Oracle VM VirtualBox
    - Minix 3
    - CLANG compiler and assembler
    - SVN: Version Control System
    - Redmine: Web application for project management
    - Doxygen
    - Visual Studio Code
    - GIMP

According to our Project Specification we also met the goals we intended for ourselves, implementing the devices and functionalities we set out to implement.

User instructions

Starting Screen

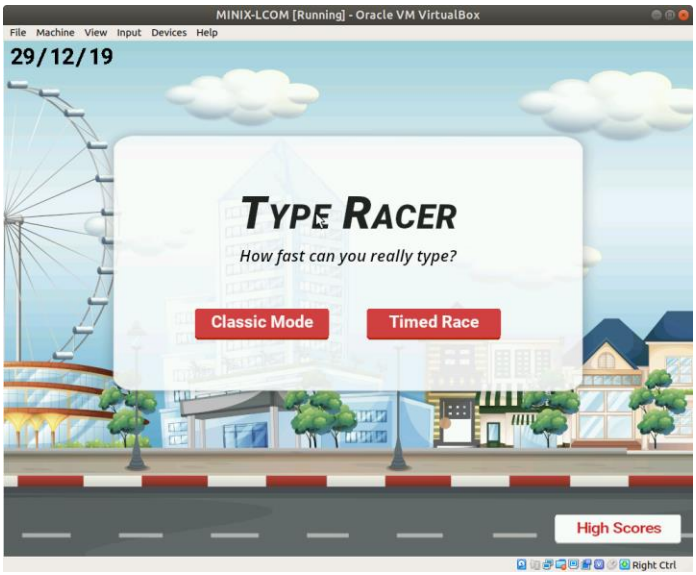

Figure 1 - Main Menu

At the start of the program, the user is presented with a main menu screen with the title of the game, the current date, a sub-sentence and 3 buttons equivalent to 3 options: “Classic Mode”, “Timed Race” and “High Scores”. It is possible to exit this main menu, thus exiting the game, by pressing *Esc*. at this point the user can choose one of the available game modes or the “High Scores” option.

Countdown

After pressing one of the game mode buttons, a countdown will be presented to indicate the start of the game and to allow the user to get ready.

Classic Mode

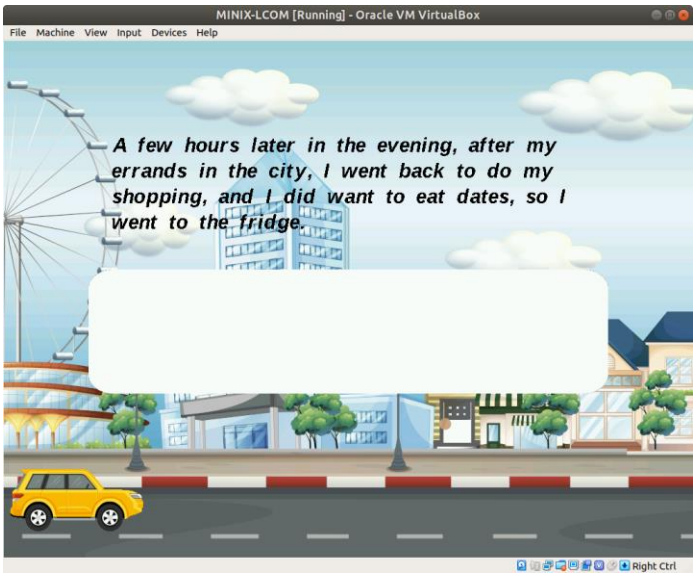

Figure 2 - Classic Mode Screen

In Classic Mode a random sentence is shown on the screen for the user to write, a blank space where the written sentence will appear and a yellow car that moves accordingly to the progress made will also show. At this point the user can press *Esc* to exit the current game mode and return to the main menu or start playing. To do this, the user needs to type the sentence he sees on the screen as fast as he can while trying not to commit spelling mistakes. The progress will be shown accordingly.

Correct State

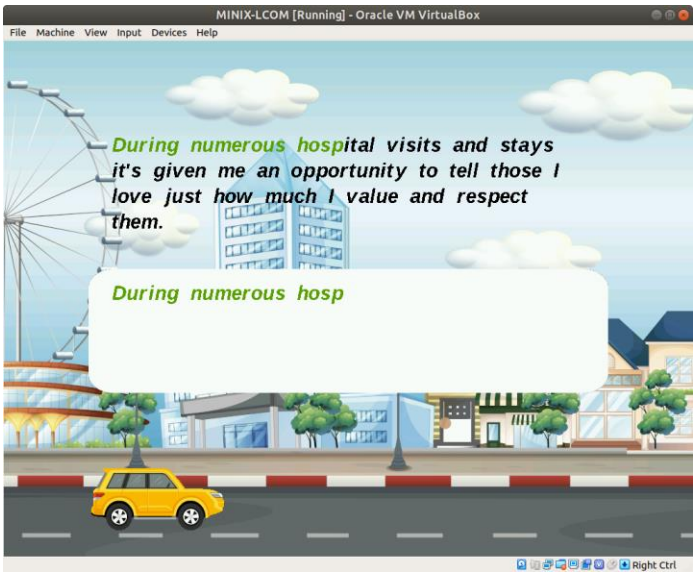

Figure 3 - Normal Game State

If the input is correct (the same as in the sentence) it will be written in the white box, the corresponding letter will turn green and the car will advance to the right.

Typing Error State

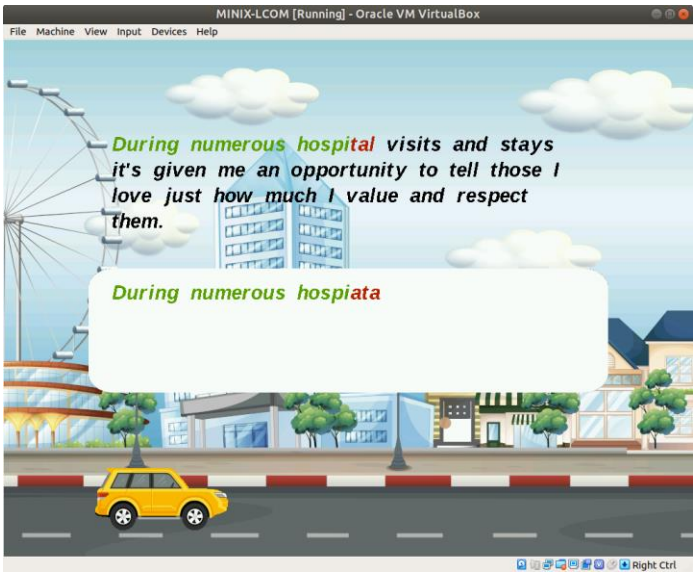

Figure 4 - Typing Error Game State

If the input is incorrect, the letters on the white box will become red and the corresponding letters on the sentence will also turn red. To return to the normal state the user needs to press *backspace* until all letters on the white box become green.

Error Message State

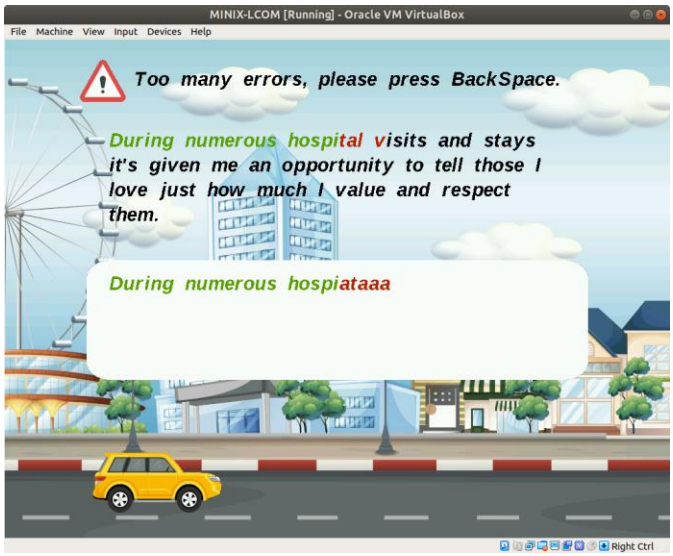

Figure 5 - Error Message Game State

When the game detects 5 errors it will enter an *Error Message State* in which a prompt will appear to warn the user to press *Backspace*, if the user presses any other key (i.e. a letter), the game will not accept it and won't display it on the screen. Once the *Backspace* key is pressed once, the game returns to *Typing Error State*.

Results

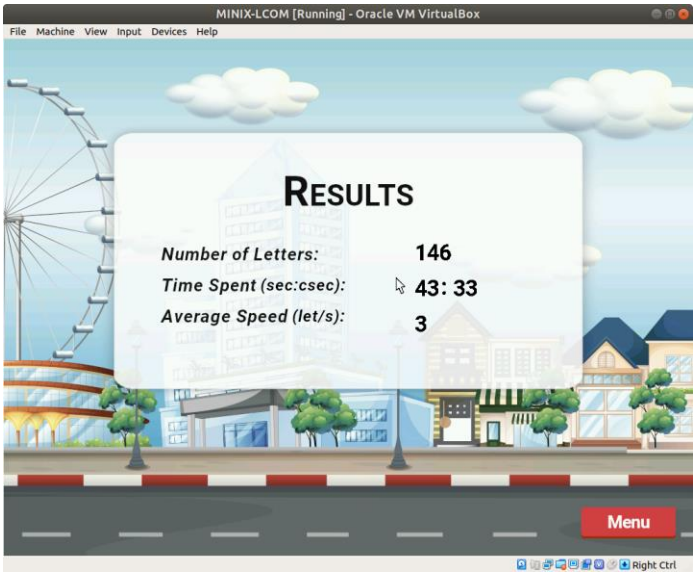

Figure 6 - Results Screen

When the user ends writing the sentence, the Results screen shows the number of letters typed, the time it took and the average speed in letters per second. The user can exit this screen by pressing the *Menu* button or pressing the *Esc* key.

Save Score

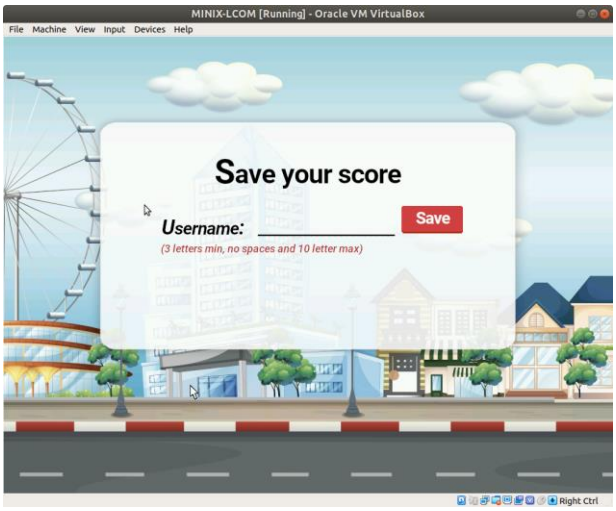

Figure 7 - Save Score Screen

After exiting the Results screen, the user should input a username to save his score. The result can be viewed along with the date and hour when it was achieved and with its correspondent username at the High Scores, but only if it reached the top 5.

Timed Race

The Timed Race mode follows some of the formatting for the Classic Mode: there is a countdown and the same correct/error states. However, in this mode there is also a 1 minute timer counting upwards. The user must write as many words as he can in that time, during which multiple random sentences are generated.

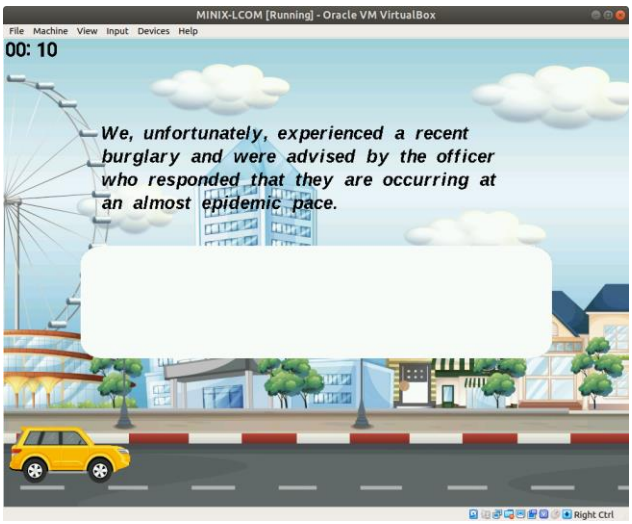

Figure 8 - Timed Race Game Mode

At the end of the 1 minute mark the Results screen shows up. The user is presented with the amount of characters he was able to correctly type in that time and his average speed.

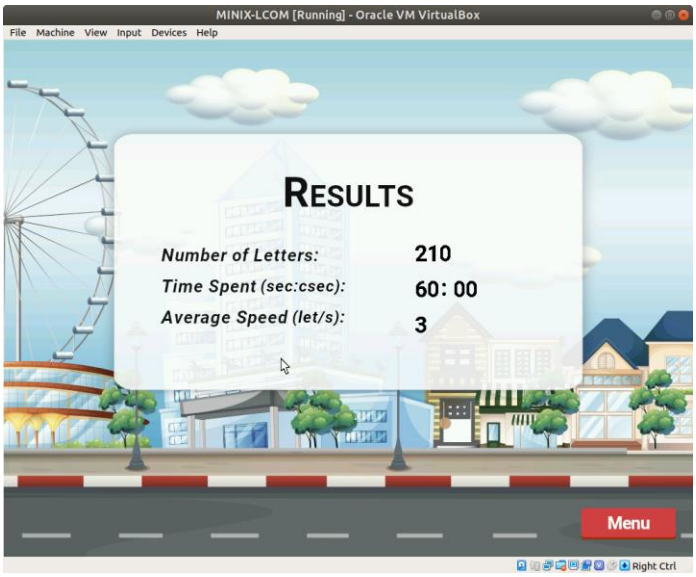

Figure 9 - Results Screen

High Scores

Figure 10 - High Scores Screen

It is possible to see the saved high scores (including the scores saved from previous sessions on a text file) by pressing the corresponding option on the main menu. The user can exit this screen by pressing the button on the bottom right corner or by pressing the *Esc* key.

Project Status

Functionalities implemented:
Timer, Keyboard, Mouse, Video Card, Real Time Clock.
Didn't implement UART.

Device	What for	Interrupts
Timer	Update Game Status / Refresh Screen (Draw frames)	Yes
Keyboard	Main part of game – type sentences and usernames	Yes
Mouse	Button selection – to go to a game mode or to High Scores from the main menu and to go back to the menu or save a score	Yes
Video Card	Display on screen the state of the game	Yes
RTC	Save date and hour of a high score after a game is played and display current date/time on the screen	No

Timer

Functionality

We used mostly the TIMER_COUNTER variable aided by the timer interrupt handler (void timer_int_handler()) to control the frame rate. Whenever there is a timer interrupt, depending on the current state, the timer is used differently to show the correspondent frame. The timer is used for the initial countdown and, also, to keep track of the time the user took in "Classic Mode" and to assure that in "Timed Race" the user as 60 seconds available. Most importantly, in all the game modes in which the user can use the mouse, the timer is used to refresh the screen at 60 frames per second – periodically. However, in both of the game mode, the timer is used to update the video RAM only if a valid key is pressed – i.e. if a key that is not used in any sentence is pressed, the input array is not changed and, therefore, there is no need to update what is shown on the screen.
The function called on each timer interrupt depends on the current game state mode, but all of them draw the frame on the screen.

Code

The game is based on time but mostly revolts around the usage of the TIMER_COUNTER variable throughout the program. With this, the directly timer related functions used were: *timer_subscribe_int(), timer_unsubscribe_int()* and *timer_int_handler().*

Keyboard

Functionality

In the main menu/results screen/high scores screen the keyboard expects the *Esc* make code to exit. In “Classic Mode” and “Timed Race” the keyboard is also used to exit the current mode by pressing *Esc* and to process the game state in *classic_mode()* or *timed_mode()*, using other functions such as *checkScanCode()* and *validateLetter().*

Code

The *kbc_ih()* is used on every keyboard interrupt to process the scan code (uint8_t SCANCODE[2]) of the pressed key. If the program is in one of the 2 game modes, the functions *kbc_handle(), checkScanCode()* and *validateLetter()* will use the input and process it to refresh the game state. Also, in Save Score, the *playerName (uint8_t scancode)* function is called to process the keys pressed by the user when typing its username.

Graphics Card

Functionality

To refresh the screen, we used double buffering (*double_buffering()*) – an important topic discussed in detail in the respective section. The screen Also, we implemented a moving object, the moving car during the game screens. There was no need for collision detection. We used 0x117 VBE mode with a resolution of 1024x768 and 64K colors (5:6:5) - Direct Color Mode with 2^{16} colors. For each state we implemented a different draw function (*drawMainMenu(), classicModeDraw(), timedModeDraw(), resultScreen(), highScoreScreen(), save_score_screen()*). The main background, the traffic lights and the warning sign were obtained from *FreePik, FlatIcon* and *Vector Stock* (1,2,3). The letters and numbers use mainly *Roboto* font. All the images needed to be edited to change the proportions, sizes and to be converted to a new format

(.xpm) using GIMP. These images can be found on the folder (proj/sprites) and subsequently subfolders.

(1) https://br.freepik.com/vetores-gratis/cena-da-cidade-com-a-roda-gigante-e-edificios_5348314.htm#page=2&query=Street&position=36
(2) https://www.flaticon.com/free-icon/warning_752755?term=warning%20sign&page=1&position=5
(3) https://www.vectorstock.com/royalty-free-vector/set-of-traffic-lights-icon-red-green-and-orange-vector-22987753

Code

We created some functions in the module that are reutilized often: *draw_xpm_image(), draw_sentence(), draw_input_sentence(), draw_letter(), get_double_buffer(), double_buffering()*, draw_pixel(), draw_name() and other functions used to draw the frames for each game mode (i.e. drawMainMenu()).

Mouse

Functionality

The mouse is used as a cursor in the main menu, results screen, save score screen and high scores screen. When the first frame of a mode/game state is drawn, the mouse cursor is shown at the center of the screen. We then detect its displacement on every interrupt and increment the initial position and the state of the left mouse button to keep track of the mouse's movement and position on the screen and, also, to detect if the user is trying to press a button.

Code

The *mouse_ih()* is used on every mouse interrupt and, if the game is in one of the states mentioned above, the *mouse_handler()* function is called to assure the cursor is being correctly drawn and to refresh its current position. The mouse position is drawn on menu drawing functions, such as *drawMainMenu(),* but only after all the other elements are drawn. In these functions, if statements are used to check if the cursor is on top of a button with the left mouse button pressed.

RTC

Functionality

The RTC is implemented to display the current date on the Main Menu screen and to describe the date and time when a new high score was beaten.

Code

Because there was no lab assignment to the RTC we implemented the functions needed for our program. *rtc_read()* to read the data and address registers and *rtc_read_date()* that is used at the start of the program to display the current date and to save the date and time to add a new high score. As well as some "supporting" functions such as *bcd()* and *bcd_to_binary()* which allow us to check if the date and time information is stored in BCD and, if it is, to convert it to binary. Further details about the implementation of the RTC will be explained below in the respective section.

Code organization/Structure

These are some of the most important modules that were necessary to run the program. The device modules (except for the RTC and Video Card) are not included below because the functions used were developed in the labs.

Game.c / game.h

Undoubtedly the most important module in our project. Here we have declared a lot of data structures (fig. 10 - 13), with descriptions as doxygen format comments, and the starting function of our program as well - *gameStart()*, in which the space needed for the game variables and data structures is allocated and where some of those variables are initialized. Only one "while loop" was used in this program – as explained further below; located in this module in the function *gameStateHandler()*. In this module there are also the functions that load the xpm's in the beginning of the program and free the memory at the end, the functions to fill the screen, to calculate the game state and to validate inputs. It has an approximate 65% weight.

Highscores.c / highscores.h

In this module we only have 4 functions: *get_highscores()* to retrieve the saved high scores from a .txt file and load them into memory; *new_highscore()* to check if the speed at the end of a round is enough to be a high score; *set_highscores()* to write the current high scores back to the .txt file and *add_new_highscore()* to insert the new high score and relocate the old ones. This corresponds to a weight of 5%.

Rtc.c / rtc.h / rtc_macros.h

In the rtc_macros file we created the necessary macros to implement the RTC as well as a data structure to save the data we read from the device (fig. 14). On this module we also defined the functions needed to interact with the device. To be able to read the current date we need functions to aid *rtc_read_date()*, those are *isUpdating()*, *rtc_read()*, *bcd()* and *bcd_to_binary()*. Used in some places and easy to implement, corresponds to a 5% weight.

Sentences.c / sentences.h

In the header file for this module we implemented more data structures to save information about each letter and number, as well as to save the sentences used in the game (fig. 15). Arrays were created to store all the letters, numbers and the error message as well as a two dimensional array to store the sentences. There are also the respective functions to load this data into memory : *loadLetters(), loadsentences(), allocateTwoDimenArrayOnHeapUsingMalloc()* and to free the memory allocated at the end of the program: destroyTwoDimenArrayOnHeapUsingFree(…) and destroySentencesAndLetters().

Vc.c / vc.h

This module is included because we not only used some of the functions needed for the lab assignment, such as : *vbe_get_info_mode()*, *graphics_mode()* and *draw_xpm_image()*; but also added a lot specific to our program: *draw_sentence(), draw_letter(), draw_name()*. As well as the functions needed to execute double

buffering : *get_double_buffer(), double_buffering()* and *free_double_buffer().* It contributes with a weight of 15%.

Xpm.h

In this file we have the #include of all the xpm files used, 95 in total.

Functions call graph

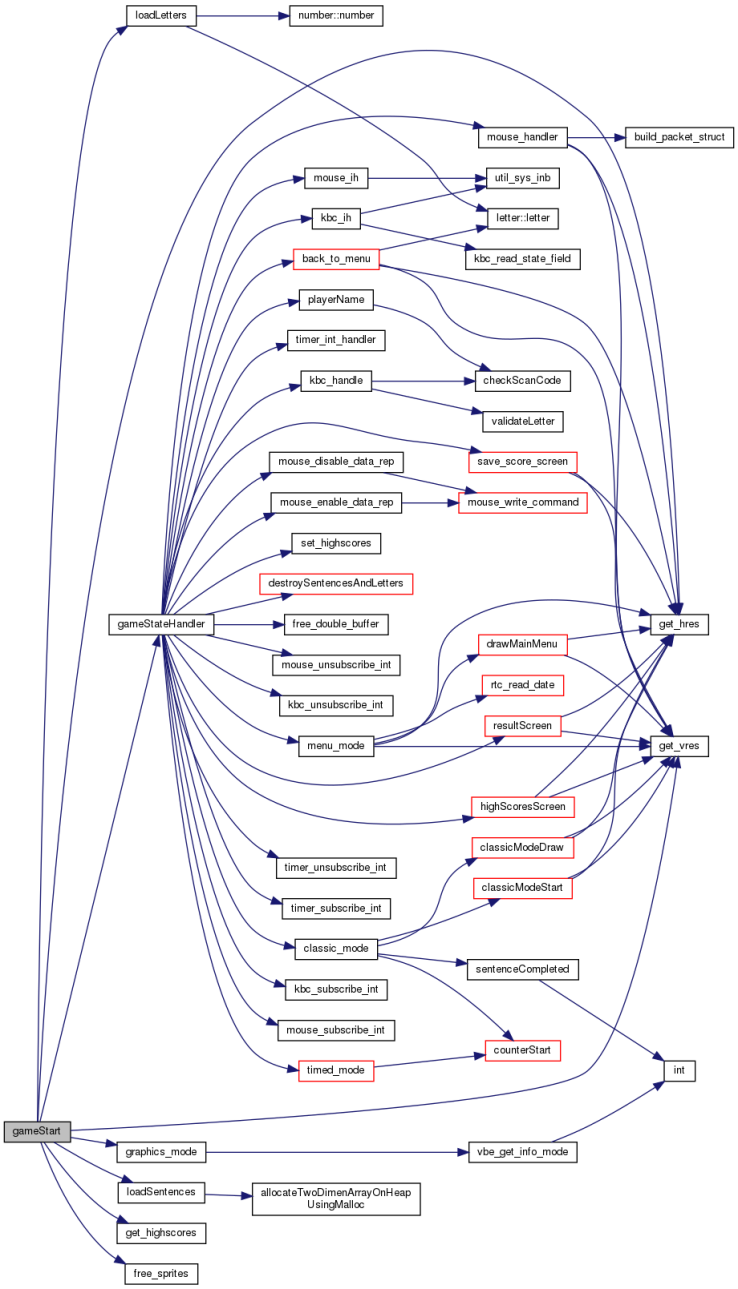

Figure 11 - Staring Function Call Graph

Implementation details

The implementation of most of the project's features required knowledges acquired in Theoretical Lessons and specially in Labs but also some other details that we had to explore and learn throughout the project.

Interruptions loop, event driven code and state machine

One of the main aspects we want to highlight is the use of only one cycle (while loop) to deal with the interruptions from all the peripherals used – in gameStateHandler(). This required a lot of code structuring and restructuring so that this loop was as organized and efficient as possible.
To structure the project with only one interrupt loop we used switch cases for each of the conditions that dealt with interrupts from a specific peripheral.
In these switch cases we use a case for each of the modes of the Game (mode enumeration), particularly the ones that have a "screen" associated with them. This strategy allowed us to deal with the interruptions we needed in each of the modes and specially to associate the transition between modes with the actions of the user – event driven code. For example, when the "Menu" button is pressed in High Scores screen, the variable that identifies the mode of the game is changed from "HIGHSCORES" to "MENU" so that in the next iteration of the loop, depending on which driver generated the interruption and on the actual state of the game, the game responds in the expected way.
To sum up, we consider the strategy used in this function and in its loop to be an effective method for developing robust event driven code, working such like a state machine and updating the game state with the actions of the user– button clicks, keys pressed, mouse movements.

Timer & Graphics Mode

Regarding the use of the Timer, we felt that the knowledge required had been previously deepened in all the labs, as most of them implied the use of this peripheral at least in one of its functions (i.e. read scancodes/mouse packets for a time period).

Therefore, as in labs, we used the timer's interrupt handler to increment a global counter to keep track of the time elapsed during the game. This counter is set to zero when a game begins. It is also used for the countdown and to calculate the speed of the user's typing. Taking all this into account, the use of the timer was only a novelty in terms of combining it with the Graphics Mode to control the frame rate and to implement some small functionalities such as "printing" the time elapsed on the screen (to show the speed and the time elapsed in "Timed Race").
First, generating the frames on each interrupt of the timer was easier to implement by using the switch case referred above, that calls a draw function according to the game mode. Furthermore, to show the moving car for example, another counter was necessary to increment the horizontal position of the car (moveCar) accordingly to the number of letters of the sentence and to draw it in the correct position on each timer interrupt.
Moreover, the solution found to print the time was to use an implementation of itoa(), a function that converts the time elapsed in an array of chars. As this function was not available as a C Library function we used a solution found in "Techie Delight" that, converting the time into an array of chars, allows us to use the xpm's of all the numbers and to compare them with the elements of an array of number structs - which associates an image of a number with its respective char.

Double Buffering

To refresh the screen and prevent an unexpected and average behavior while rendering the frames and showing the images on the screen, that could be produced if the bytes in the frame buffer were being changed while they were being scanned for display on the screen, we used Double Buffering. This technique implies using two buffers - a back buffer where the frames are first rendered and the usual frame buffer (*video_mem),* to which, once the frame has been completely rendered on the back buffer, the frame is copied – with *memcpy*. Then, the frame buffer is the one displayed on the screen.

Memory Allocation and Deallocation

Due to some initial "low memory" warnings we were led to believe that we would benefit from deeper knowledge about memory allocation.

In the project we used many xpm's for the letters, sentences and numbers. This required us to use one and two dimensional arrays of structs for the alphabets and sentences. To do so we used functions that use *malloc* to allocate space for the number of structs we need to store in the array. For the user's input and for the sentence generated randomly on the screen in the game modes, we need to allocate and deallocate memory on each game because the sentences have a different number of letters and so, the input buffer must be empty when the game starts. We also developed our skills using pointers because we needed to pass the arrays as an argument to functions without losing its value between function calls.
At the end of the game loop, all the allocated space for the sentences is freed.

DrawSentence()/DrawInput()

During Labs the only type of content we needed to present on graphics mode was xpm images. However, in our project we deal with images of letters and numbers that need to be positioned according to each other's width so that a sentence or word is formed without breaking words in half, for example. For that we developed functions that, by measuring the width of the letters xpm's between two consecutive spaces, allow any sentence to be correctly printed onto the screen without formatting issues.
Furthermore, this functions call *draw_letter(…)* passing the color as a function parameter. In its turn, the draw_letter function checks if the original color of the xpm's pixels corresponds to the one passed as parameter. If the two colors are different, then it draws the pixel with the color passed as parameter instead of the one in the xpm, otherwise the original color is used. This function was developed to avoid using letters xpms of 3 different colors and to take advantage of the code to solve the problem.

Keyboard

Taking into account that the keyboard is, along with the graphics, the peripheral that takes a bigger role in the game modes, we feel some of the functions developed to process the scan codes and to register the state of the keyboard - type of key pressed; deserve a more detailed description. We developed three functions regarding this matter:

- *void kbc_handle(letter ** sentence, letter ** input, unsigned int * index)*

This function calls the checkScanCode function and, based on its return value, calls the validateLetter function. It decides if the screen should be refreshed – the key is valid; or if the key is invalid and therefore should be ignored. Also, if the sentence is complete it sets the Boolean variable accordingly.

- *keyPressed checkScanCode(uint8_t * scancode)*

The first function called to process the scan code is used to check if the scan code is accepted or if it refers to a key that is not used in the sentences. First, this function checks if the scan code has two bytes or one. As we only dealt with letters and numbers, which are associated with one byte scan codes, if the scan code has one byte the function checks if it is a make code or break code. If it is a make code then it checks if it refers to the Shift or to the Caps Lock key, setting a boolean variable accordingly to identify uppercase or lower case letter, or if it is a dot, an esc key or finally a letter or backspace. The function returns one of the values of the enum keypressed which identifies the type of key pressed or an error state – when too much error occur.

- *inputValidation validateLetter(letter * let, uint8_t * scancode, letter ** input, unsigned index)*

The next function is only called if the previous function returns VALID to check if the letter typed corresponds to the one in the sentence. Again, we use an enum input validation for the return value, which identifies if the letter is correct, wrong or if it is a backspace. This function updates the error variable and builds the input array based on the index of the letter the user should be writing and on the number of errors.

Rtc

Finally, we decided to use the RTC, which was only covered in Theoretical Lessons.
For that we developed the functions needed to use this peripheral by polling.
First, taking into account that reading or writing a register of the RTC always requires writing the address of the register to the Address Register(0x70) and reading or writing one byte from or to the Data Register (0x71), the function *rtc_read(…)* was developed to make this two actions easy to repeat without having to always write the *sys_inb(…)* and *sys_outb(..)* calls.
The realTime struct was also developed to store the information of the date read from the registers in rtc_read_date() function, in which an auxiliar function (*isUpdating()*) is

used to avoid getting data from the registers while they are still updating its content – by checking the 7th bit of Register A which is set to one if an update is in progress. Then, by using another auxiliary function, *bcd()*, it checks if the data is in binary coded decimal by reading Register's B 2nd bit which is set to zero if it is in BCD. Finally, the value is converted to binary if it was in BCD and stored in the correspondent realTime struct, returned to be used in the game.

Highscores

Lastly, to fully accomplish the goals we initially established for our project, we devoted some time to understanding how to handle text files in C.
In order to save the highest scores throughout game sessions, some functions were developed to read and write text files.
To read the file we used C library functions such as *fscanf(…)* to read numeric values from the file, particularly the date, time and writing speed and *fgets(…)* to store the usernames in arrays of chars.
To write the file we had to use another function from the C library, *fprintf(…)* with the correct format specifiers to print the desired variables.
We placed the file inside the source folder and used the full path, from the "home" directory, to read and write the file. However, to make sure the program doesn't behave unexpectedly, if the file is not in the right directory then the *get_highscores (Game * game)* function is exited and prints a message to inform that the file was not found ("printf("Highscore file not found\n")") without crashing the program. Instead, the only difference compared with the behavior expected when the file is found is that the array of high score structs, that would be naturally filled with the highest scores from previous game sessions, will be empty at the start of the game.
During the game, the scores are stored in the array, which is filled and reordered if new high scores are accomplished.
At the end of the game, if the path exists, the new file is generated if it didn't exist before or updated if it already existed. Furthermore, if the path is incorrect, the high scores are not stored but the program doesn't behave abnormally.

Conclusions

LCOM is a rewarding, but laborious subject. As it is right now, with slides that do not provide the student with enough informative material and knowledges, it is made for the students to teach themselves by reading the html pages, by searching the internet and by reading and learning with what other students did in previous year - studying their work, which is available on GitHub, to get an initial understandment of each lab assignment.

We think the subject can benefit from getting a more accessible and structured web page for the students to guide themselves, step by step, instead of skipping chapters in an initial reading to understand concepts used throughout the assignment.

Installation instructions

Instead of executing “make” in the proj folder it is necessary to execute it in proj/src. Before that, if for any reason the path for the highscores.txt file is not the correct one, it may be changed in highcores.c -> get_highscores -> line 3 -> fp = fopen("/home/lcom/labs/proj/src/highscores.txt", "r");

Figures

```
/**@brief Struct that stores the name of a player*/
typedef struct name{
  char * nameAr;                /**< @brief the name of the player in an array of char*/
  unsigned int nameSize;        /**<@brief the number of letter of the name*/
  letter * nameXpm;             /**< @brief the name of the player in an array of xpms*/
} name;

/**@brief Struct that stores the information about a highscore*/
typedef struct highScore{
  int speed;            /**< @brief number of letter per second*/
  realTime date;        /**< @brief the date of the highscore*/
  name name;          /**< @brief the name of the player*/
} highScore ;

/**@brief Mouse left button state and position*/
typedef struct mouse{
  int mouseX;           /**< @brief horizontal position of the cursor on the screen*/
  int mouseY;           /**< @brief vertical position of the cursor on the screen*/
  bool lbPressed;       /**< @brief true if left button is pressed and false otherwise*/
}mousePos;

/**@brief Game mode identifier*/
typedef enum mode{
  MENU,                 /**< @brief Menu Screen*/
  CLASSIC,              /**< @brief Classic Game Mode*/
  TIMED,                /**< @brief Timed Game Mode*/
  RESULTS,              /**< @brief Results Screen*/
  HIGHSCORES,           /**< @brief High scores screen*/
  SAVE_SCORE,            /**< @brief Save score screen - ask for the name of the player*/
  EXIT                  /**< @brief Exit the game*/
} mode;
```

Figure 12 - Main Data Structures 1

```
/**@brief Indicates the type of key that was pressed*/
typedef enum keyPressed{
  DOT,                  /**< @brief Final dot was pressed*/
  CAPS,                 /**< @brief Caps or Shift was pressed or release*/
  NOT_VALID,            /**< @brief A key that is not a letter nor a recognized symbol was pressed*/
  VALID,                /**< @brief A letter was pressed - makecode; or BackSpace*/
  ERRORSTATE,           /**< @brief Indicate too much errors were made*/
  ESC                   /**< @brief Esc key was pressed */
} keyPressed;

/**@brief The state of the game*/
typedef struct gameState{
  mode mode;         /**< @brief The current mode of the game*/
  bool start;        /**< @brief determines if it is the first frame to draw*/
  bool drawGame;     /**< @brief Indicates if a new fram should be drawn or not*/
} gameState;

/**@brief Indicates the result of a game*/
typedef struct result{
  int timeSpentSec;           /**< @brief Seconds spent writing the last sentence*/
  int timeSpentCentSec;       /**< @brief Centiseconds spente writing the last sentence*/
  int speed;                  /**< @brief Average number of letters written by second*/
} result;
```

Figure 8 - Main Data Structures 2

```
/**@brief Mouse event enumeration for the actual state*/
typedef struct Game{

  uint8_t * road_sprite;                        /**< @brief Game Scenery Sprite*/
  uint8_t * car_sprite;                         /**< @brief Car Sprite*/
  uint8_t * field_sprite;                       /**< @brief Filed to write Sprite*/
  uint8_t * error_sprite;                       /**< @brief Error Sign Sprite*/
  uint8_t * menu_sprite;                        /**< @brief Menu Sprite*/
  uint8_t * button1_sprite;                     /**< @brief Classic Mode Unpressed Button Sprite*/
  uint8_t * button2_sprite;                     /**< @brief Timed Mode Unpressed Button Sprite*/
  uint8_t * button1Pressed_sprite;              /**< @brief Classic Mode Pressed Button Sprite*/
  uint8_t * button2Pressed_sprite;              /**< @brief Timed Mode Pressed Button Sprite*/
  uint8_t * mouse_sprite;                       /**< @brief Mouse Cursor Sprite*/
  uint8_t * results_sprite;                     /**< @brief Results Image Sprite*/
  uint8_t * menu_button_sprite;                 /**< @brief Back to Menu Unpressed Button Sprite*/
  uint8_t * menu_button_pressed_sprite;         /**< @brief Back to Menu Pressed Button Sprite*/
  uint8_t * redlight_sprite;                    /**< @brief Redlight Sprite*/
  uint8_t * yellowlight_sprite;                 /**< @brief Yellowlight Sprite*/
  uint8_t * greenlight_sprite;                  /**< @brief Greenlight Sprite*/
  uint8_t * highscores_sprite;                  /**< @brief High Scores Image Sprite*/
  uint8_t * highscores_button_pressed_sprite;   /**< @brief High Scores Pressed Button Sprite*/
  uint8_t * highscores_button_unpressed_sprite; /**< @brief Save Scores Unpressed Button Sprite*/
  uint8_t * saveScores_sprite;                  /**< @brief Save Scores Image Sprite*/
  uint8_t * saveScore_button_sprite;            /**< @brief Save Scores Pressed Button Sprite*/
  uint8_t * saveScore_button_unpressed_sprite;  /**< @brief Save Scores Unpressed Button Sprite*/

  xpm_image_t carImg;                           /**< @brief Game Scenery xpm*/
  xpm_image_t roadImg;                          /**< @brief Car Sprite xpm*/
  xpm_image_t fieldImg;                         /**< @brief Filed to write xpm*/
  xpm_image_t errorImg;                         /**< @brief Error Sign xpm*/
  xpm_image_t menuImg;                          /**< @brief Menu xpm*/
  xpm_image_t button1Img;                       /**< @brief Classic Mode Unpressed Button xpm*/
  xpm_image_t button2Img;                       /**< @brief Timed Mode Unpressed Button xpm*/
  xpm_image_t button1PressedImg;                /**< @brief Classic Mode Pressed Button xpm*/
  xpm_image_t button2PressedImg;                /**< @brief Timed Mode Pressed Button xpm*/
  xpm_image_t mouseImg;                         /**< @brief Mouse Cursor xpm*/
```

Figure 14- Main Data Structures 3 (Game)

```
  xpm_image_t mouseImg;                          /**< @brief Mouse Cursor xpm*/
  xpm_image_t resultsImg;                        /**< @brief Results Image xpm*/
  xpm_image_t menu_buttonImg;                    /**< @brief Back to Menu Unpressed Button xpm*/
  xpm_image_t menu_button_pressedImg;            /**< @brief Back to Menu Pressed Button xpm*/
  xpm_image_t redlightImg;                       /**< @brief Redlight xpm*/
  xpm_image_t yellowlightImg;                    /**< @brief Yellowlight xpm*/
  xpm_image_t greenlightImg;                     /**< @brief Greenlight xpm*/
  xpm_image_t highscoresImg;                     /**< @brief High Scores Image xpm*/
  xpm_image_t highscores_button_pressedImg;      /**< @brief High Scores Pressed Button xpm*/
  xpm_image_t highscores_button_unpressedImg;    /**< @brief Save Scores Unpressed Button xpm*/
  xpm_image_t saveScoresImg;                     /**< @brief Save Scores Image xpm*/
  xpm_image_t saveScore_buttonImg;               /**< @brief Save Scores Pressed Button xpm*/
  xpm_image_t saveScore_button_unpressedImg;     /**< @brief Save Scores Unpressed Button xpm*/

  gameState state;            /**< @brief Keep track of the state of the game*/

  unsigned int moveCar;       /**< @brief Position where the car is going to be drawn*/
  float oneMove;              /**< @brief Number of pixels the car moves when the input is correct*/

  unsigned int nLet;              /**< @brief total number of letter written in a game mode*/
  unsigned int numberOfLetters;   /**< @brief number of letters of a sentence in classic mode*/
  keyPressed keyState;            /**< @brief Indicates the type of key that was pressed*/
  bool completed;                 /**< @brief Indicates that the sentence is complete*/
  bool errorState;                /**< @brief Signs a error state - maximum number of errors*/
  uint8_t errors;                 /**< @brief Number of errors*/
  bool capital;                   /**< @brief True if Caps or Shift is active*/
  int CapsMakeCount;              /**< @brief Auxiliar variable to check Caps key state*/

  realTime * date;                /**<@brief The real date to show on the main menu*/

  unsigned int NUMBEROFSCORES;    /**< @brief The number of scores registed (5 max)*/
  highScore* highscores;          /**< @brief Game high scores (5 max)*/
  name * name;                    /**< @brief Name of the player*/

  mousePos mc;                    /**< @brief Mouse left button and position information*/

  result result;                  /**< @brief Result of the last played game*/

} Game;
```

Figure 15 - Main Data Structures 4 (Game)

```c
/** @defgroup rtc_macros rtc_macros
 * @{
 *
 * Constants for programming the RTC
 */

typedef struct realTime{

  uint32_t minutes;
  uint32_t hour;
  uint32_t day;
  uint32_t month;
  uint32_t year;

} realTime;

#define RTC_IRQ         8 /**< @brief RTC IRQ line */

/* Registers/Ports */
#define RTC_ADDR_REG    0x70  /**<loaded with the address of the RTC register to be accessed*/
#define RTC_DATA_REG    0x71  /**<used to transfer the data to/from the RTC's register accessed*/

#define RTC_SEC         0x00       /**<  @brief Seconds*/
#define RTC_SEC_ALARM   0x01      /**<  @brief Seconds Alarm*/
#define RTC_MIN         0x02       /**<  @brief Minutes*/
#define RTC_MIN_ALARM   0x03       /**<  @brief Minutes Alarm*/
#define RTC_HOUR        0x04      /**<  @brief Hours*/
#define RTC_HOUR_ALARM  0x05       /**<  @brief Hours Alarm*/
#define RTC_WEEK_DAY    0x06       /**<  @brief Week Day*/
#define RTC_DAY         0x07      /**<  @brief Month Day*/
#define RTC_MONTH       0x08       /**<  @brief Month*/
#define RTC_YEAR        0x09       /**<  @brief Year*/
```

Figure 9 - RTC Data Structure and some macros

```c
/**@brief Information about a letter - scancode, char, lower or upper case, sprite and xpm image*/
typedef struct letter{
    char letter;        /**< @brief The char associated with the letter*/
    bool capital;       /**< @brief True if the letter is uppercase and false if it is lowercase*/
    uint8_t breakCode;  /**< @brief The keyboard break code associated with the letter*/
    uint8_t makeCode;   /**< @brief The keyboard make code associated with the letter*/
    xpm_image_t img;    /**< @brief The xpm image of the letter*/
    uint8_t * sprite;   /**< @brief Letter's sprite*/
}letter;

/**@brief Information about a number - char, sprite and xpm image*/
typedef struct number{
    char number;           /**< @brief The char associated with the number*/
    xpm_image_t img;       /**< @brief The xpm image of the number*/
    uint8_t * sprite;      /**< @brief Number's sprite*/
}number;

#define nLetters  59       /**< @brief The number of letters and symbols in the game's alphabet*/
#define nSentences  30     /**< @brief The number of sentences that may be generated in the game*/
#define nChars  163        /**< @brief The max number of chars of a game's sentence*/

letter * abc;              /**<@brief The game's alphabet */
// 0-25 -> a,b,c,d,..,z
// 26-51 -> A,B,C,D,..,Z
// 52, 53, 54, 55, 56  ->  . , : ; '

number *numbers;           /**<@brief The game's numeric alphabet */

/**<@brief An array of char sentences that may be generated when playing the game */
static char sentencesChar[nSentences][nChars]={
    {"You control the content, so if you misspell a name or get a number wrong, it is your own
    fault."},
    {"The European Union wants a complete ban on drift net fishing until the end of the next year,
    otherwise it will take some strict measures."},
    {"According to a press release issued by district officials, the student drove a vehicle to
    school belonging to a parent and the weapon was in that vehicle."},
    {"Additionally, I doubt many people run for office simply because they need a job and love
    politics."},
```

Figure 10 - Sentences module data structures