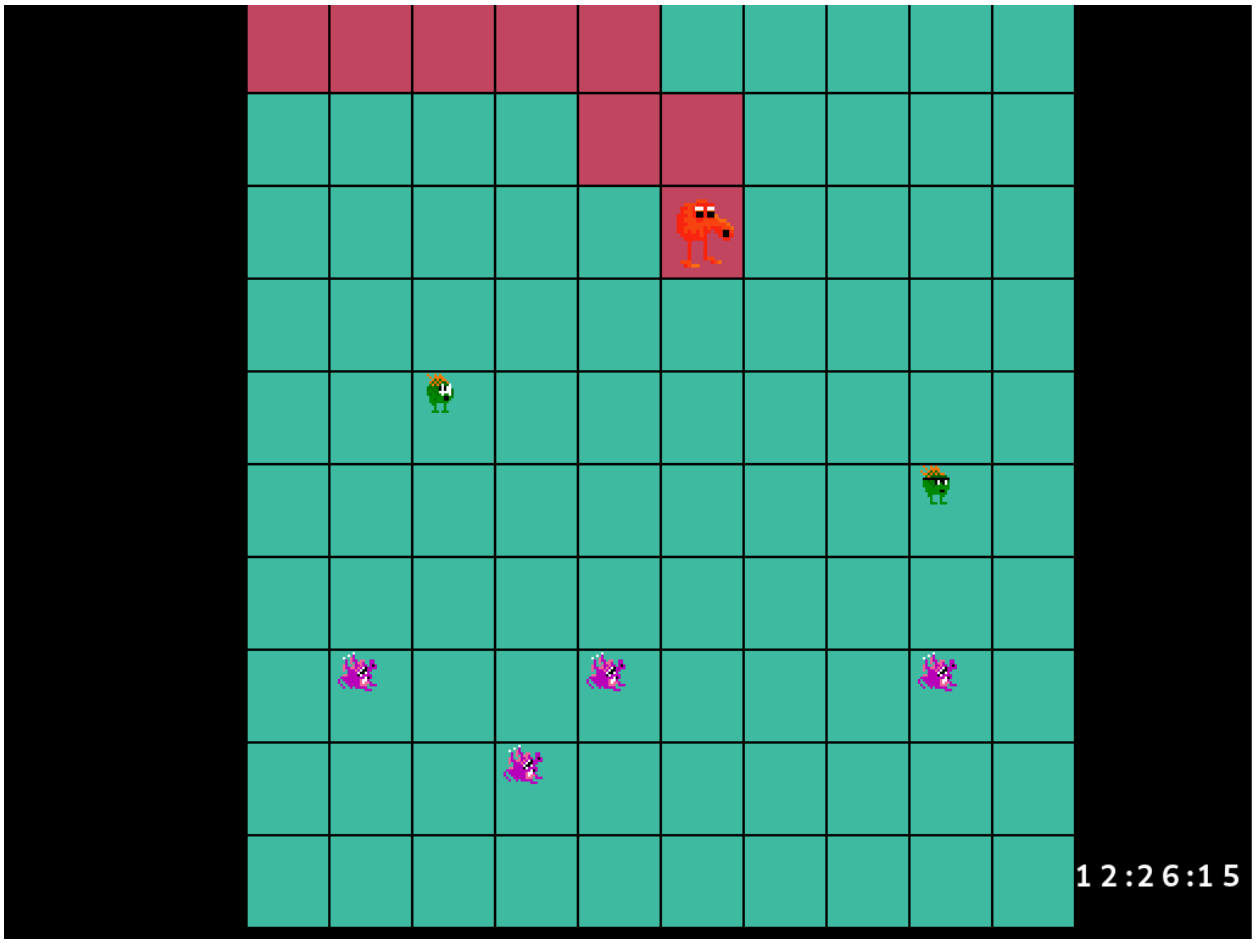
Q*bert 2D Edition

A 2D implementation of the popular arcade Q*bert

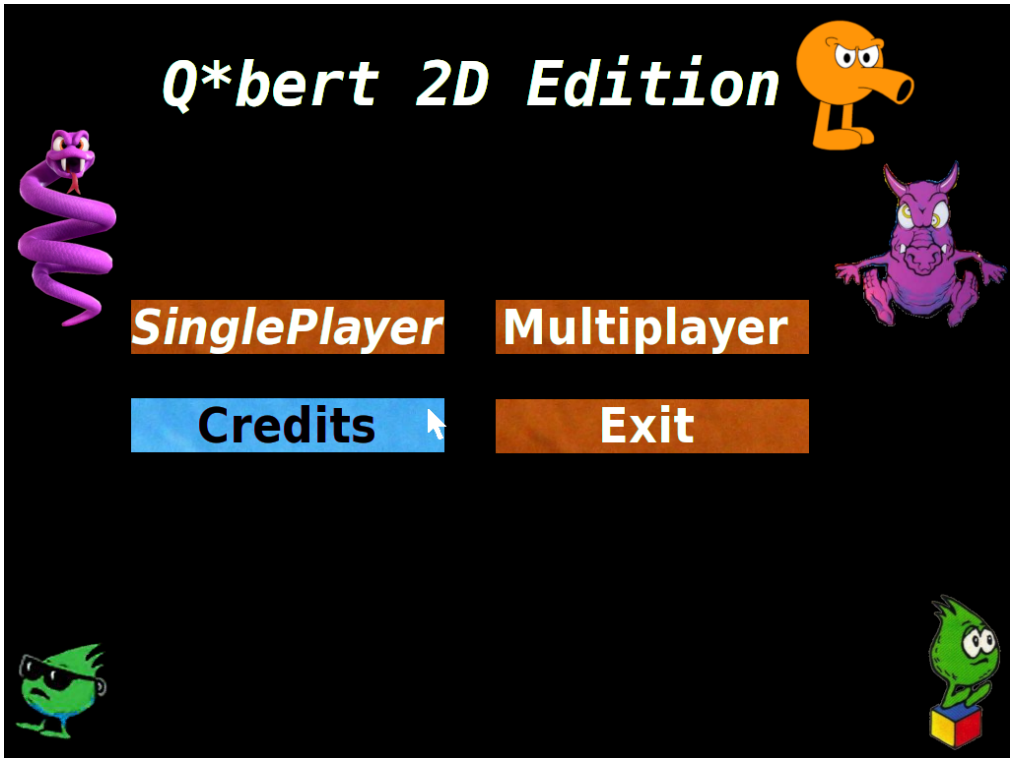


Faculty of Engineering of the University of Porto - LCOM, MIEIC 2nd Year T4G11
André Filipe Magalhães Rocha - up201706462@fe.up.pt
Joana Catarina Teixeira Ferreira - up201705722@fe.up.pt

Index

1. User’s Instructions

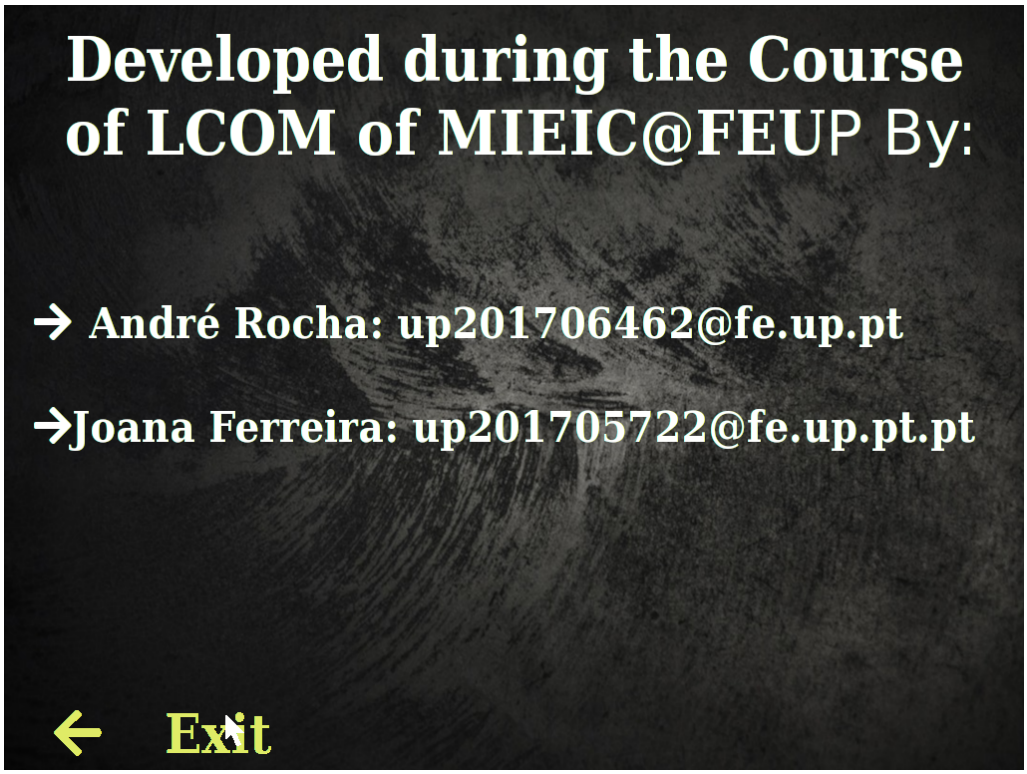
1.1 Initial Screen



As the program is initially booted, it is displayed the menu presented in the upper image with all the available options from the start. With the movement of the mouse and the use of the left button of it, it is possible to choose one of the of the following options presented:

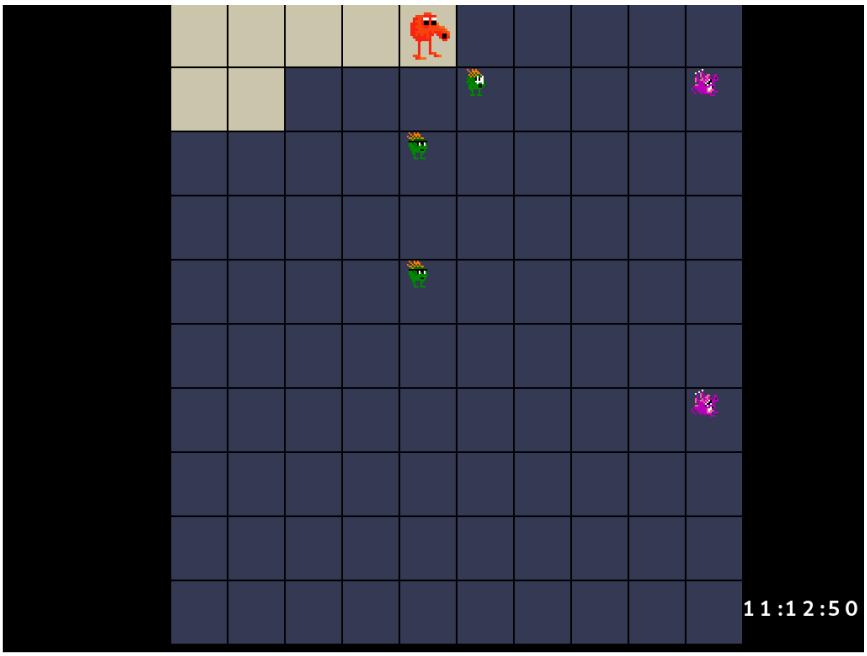
- SinglePlayer
- Multiplayer
- Credits
- Exit

1.2 Credits Menu



If the “Credits” option is pressed, this menu is shown to the user, where there is a list of the contributors to this project, as well as the due credits to the course which led us to the development of this program. The Player can then press the mouse over the “Exit” button to go back to the Initial Screen

1.3 SinglePlayer Menu



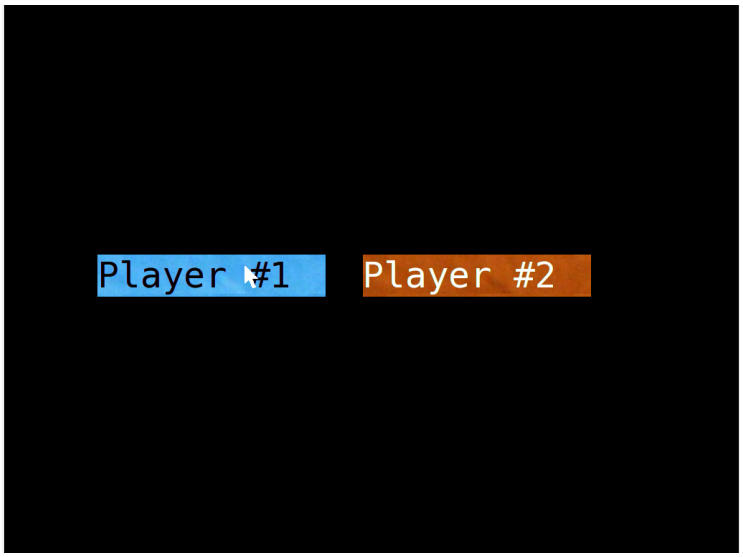
If the “SinglePlayer” option is pressed this menu is presented, where the main character (the Q*bert) and its enemies are presented. It is possible to control the Q*bert using the keyboard (**W** to go up, **S** to go down, **D** to go right and **A** to go left). The enemies are launched into the board in a **pseudorandom** way, in order to maintain some challenge in the game itself. Four types are launched into the board, all of them having different patterns of movement. The easiest enemies are the ones with vertical and horizontal paths, which are extremely predictable themselves. The other two have some randomness added to them, in which they always move diagonally, however, their vertical or horizontal part of the movement can possibly change, in order to catch the player off guard.

The game itself is **endless.** When the board is all colored, a new colored is then selected, in order to restart the job of repainting all the squares once again. The only way to lose is to suffer a collision with an enemy. If a collision happens, the ‘Game Over’ menu is presented.

1.4 ‘Game Over’ Menu

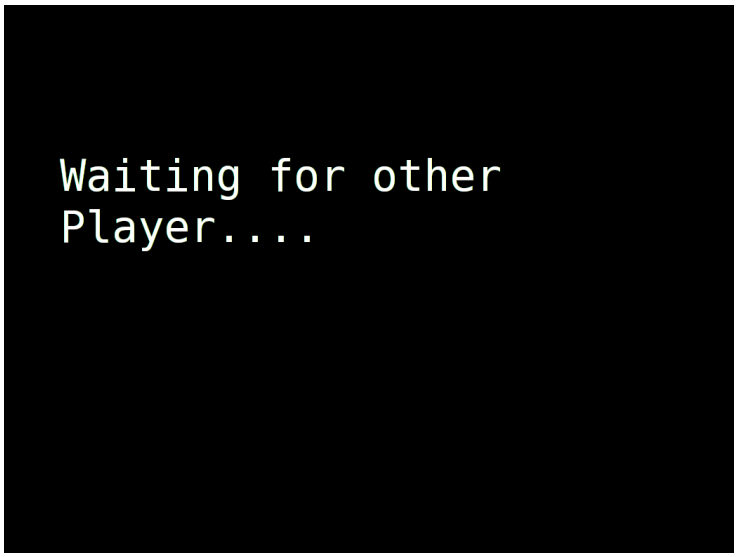
As explained above, when a collision with an enemy occurs, the player loses the game. This is then shown with this menu, the game over menu, that makes you aware that you have lost the current attempt to play the game. The press of any keyboard key sends you back to the main menu.

1.5 MultiPlayer Menu



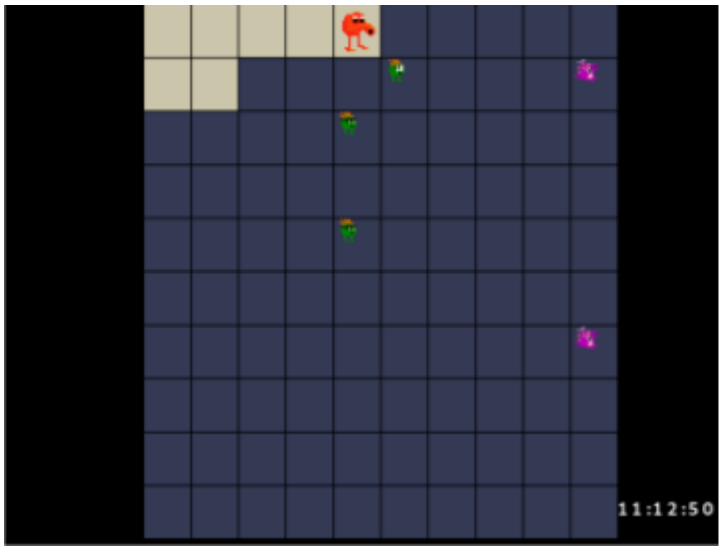
If the “Multiplayer” option is pressed, and the other computer is not connected, this menu is shown to the user in order for him to choose his status as Player 1 or Player 2 during the playthrough of the game. Otherwise, if the computer has already chosen his status, the game will start with the actual game with the opposing status.

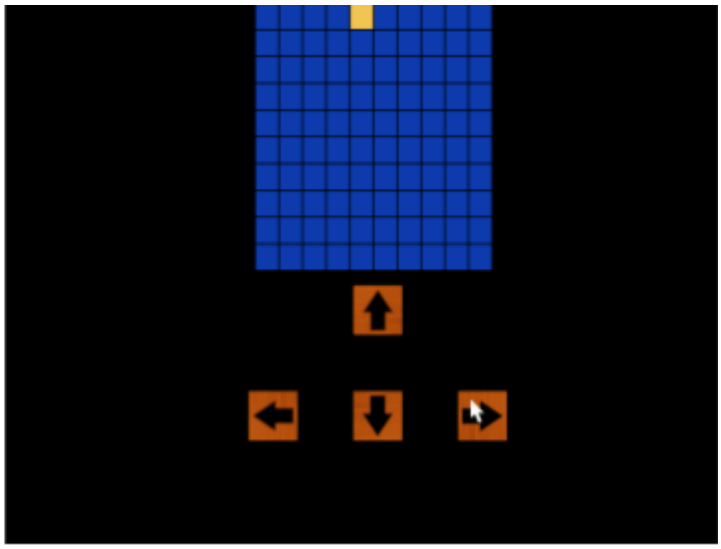
1.6 The Waiting Multiplayer Menu



When one of the following buttons is then pressed, a waiting menu is provided as the user shall wait for the other player to send a request through the serial port in order to start the game.

1.6 Multiplayer Game Menu





During the Multiplayer Game, Player 1 is able to see game exactly as it was when playing the SinglePlayer version of the game. Indeed for Player 1 visually and logically **there are no changes**; however, the launching of new enemies into the board is **now controlled by Player 2** instead of being pseudorandom like before. Player 2 can use the mouse to select which board position should the next enemy be in the next launch, which will be presented visually by a board in the Player 2's screen as well. When Player 1 loses the game, they're both sent back to the main menu.

2. Project Status

The following table will show which devices were implemented during the development of Q*bert 2D Edition. The interrupt handling cycle is presented in the module **interrupt_handling.c.**

Device	What for	Interrupts
Video Card	**Displaying menus' information and the game itself**	**N/A**
Keyboard	**Moving the Q*bert (game character) through the board**	**Yes**
Mouse	**Choosing Options in the menus** **Controlling the position where the new enemy will appear from as Player 2**	**Yes**
Timer	**Controlling the frame rate**	**Yes**
RTC	**Controlling the appearance of further enemies on the board** **Displaying current time**	**Yes, Periodic Interrupts**
Serial Port	**Communication between the two computers**	**Yes, with FIFOS**

2.1 Video Card

The Video Card is used in the **mode 0x144**, with the resolution 1024x768, using color mode (8:)8:8:8, (32 bits per pixel, with a total of 16.777.216 colors available).
It is also used **Double Buffering** during the whole program in order to make it visually more fluid for the user, using a support buffer to where all screen modifications are first written to.
There are also several **animations** associated with each enemy and the Q*bert itself. All animations have a period of 15 ticks, related to the movement of each character through the screen.
There are also several **collisions detection** mechanisms during the whole program, starting with the interaction with the mouse position and the buttons (possibly leading to different menus

as well as giving a hovering animation to the button itself), and most importantly between the Q*bert and the several enemies on the map, trying to “catch” him.
Even though some of the writing is coming purely from images, a font is also used for the display of time during the game itself (having been imported all the numbers as well as the ‘:’ as a separator.)
This implementation is spread through various files, being the most important ones video_graphics.c, menu.c and xpm.c.

2.2 Keyboard

The Keyboard is used for:

- Q*bert (the player) movement (using ‘W’ to go upwards, ‘A’ to go leftwards, ‘D’ to go rightwards and ‘S’ to go downwards).

The main implementation of this module is in keyboard.c

2.3 Mouse

The Mouse is used for:

- The navigation through the game menus, through the collision of the mouse with the buttons themselves
- The interaction (as Player 2) with Player 1, through the selection where the next enemy will appear.

The main implementation of this module is in ps2Mouse.c and mouse.c

2.4 Timer

Interruptions from the timer are used for:

- Updating the game frame rate
- Animations
- The detection of collisions between enemies and the Q*bert, during their movements
- Updating the game state, according to what has happened in the other devices.

The main implementation of this is under the module timer.c

2.5 RTC

The RTC is used for:

- The display of the current time during the playing of the game. For such, update interrupts are used in order to correctly get the current time.
- The launch of the next new enemy on the map, using periodic interrupts, with a frequency of 500ms.

The main implementation is under the module rtc.c

2.6 Serial Port

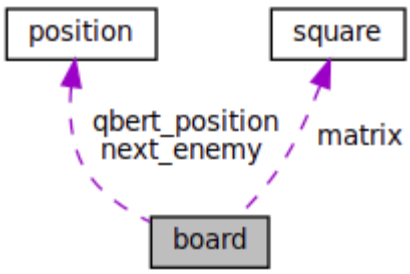
For the MultiPlayer game mode, the Serial Port is used in order to allow the communication between both players. For this program, the serial port is used with **interrupts and FIFOs** as well.
By selecting the MultiPlayer option on the main menu a **“ping”** is sent to the other computer in order to check if he is all ready to play. If he is, an **ACK** (acknowledgment) is sent back, with the information embedded as to which player he has already selected, thus starting the game for both players.
During the actual game, Player 2 sends information as to where will the next enemy appear (by scrolling through the map using the arrows), receiving an **ACK** if the information is properly received.
The main implementation of this protocol is done the serial_port.c module.

3. Code organization/structure

Modules

board.c

Overall Weight: 18%
Each members contribution: - André: 0% - Joana: 100%
Description: This module was created to manage both the multiplayer and singleplayer boards and consequently the movements of all the characters in them. Besides having functions to move the characters, it also has functions to check if the movements are OK or if there are collisions or out of the boundaries movements.



For this, it is used a struct ‘board’ (in the picture), that is defined in ‘game.h’. There are also functions to draw both the boards in the buffer.

```
typedef struct square
{
    bool color; /**< True if the square was painted by the Qbert */
    character_name name; /**< The character that is in the square */
} square;
```

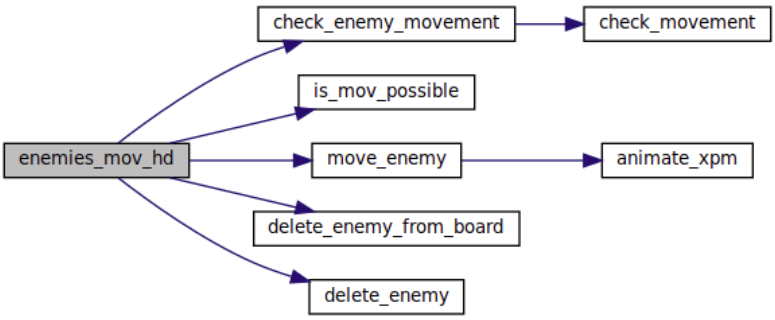
By using bitwise operations, the function to generate random colors guarantees that the board is always initialized with two random colors that always contrast between them.

clock.c

Overall Weight: 2%
Each members contribution: André: 100% - Joana: 0%
Description: This module was created to print the current time in the buffer, using the RTC.

controlSprite.c

Overall Weight: 7%
Each members contribution: André: 30% - Joana: 70%
Description: This module was created to handle the movement of the enemies in each frame. Therefore, it has an array with all the enemies that are currently in the game and functions to change this array, by adding or deleting enemies.
It is also responsible for animating and moving the characters (*enemies_mov_hd*), whilst checking if there are collisions (calling functions from the *board.c*) and either deleting the enemies or signaling Game Over.

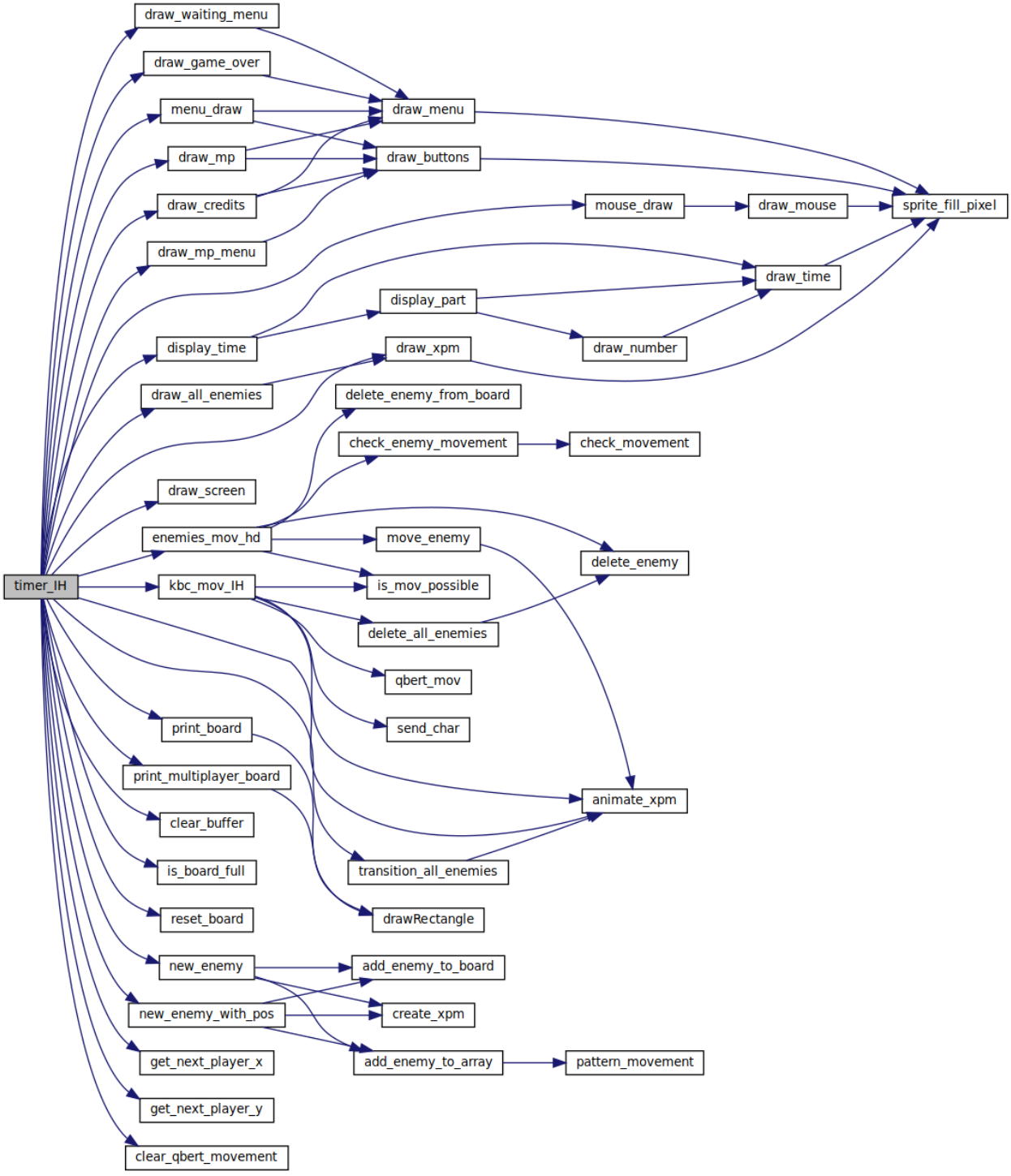


event_handlers.c

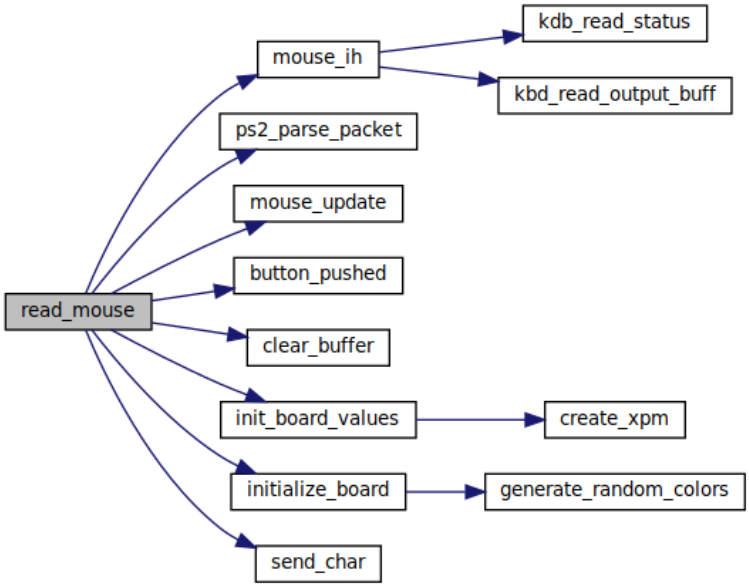
Overall Weight: 24%
Each members contribution: André: 60% - Joana: 40%
Description: This module was created to handle events from each peripheral device.

- **Timer**

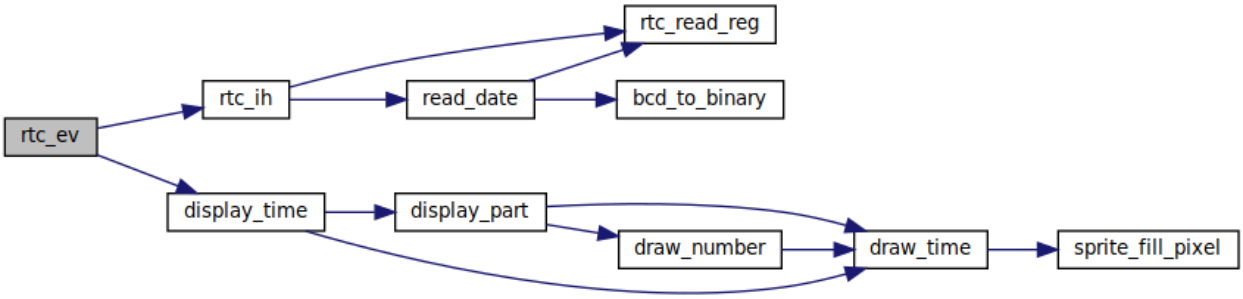
The *timer_IH* sets the frame rate. For each interrupt, it runs the game logic, prints the result of it and changes the state in the state machine that the game represents (check *4. Implementation Details - State Machine*).



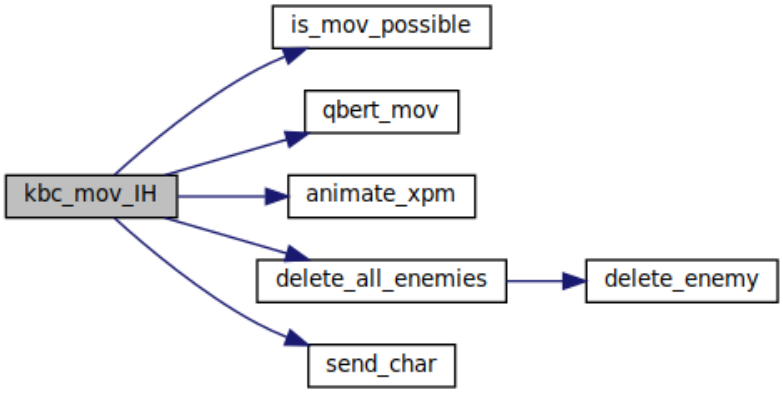
- **Mouse**
  The *read_mouse* function implements a state-driven methodology where according to the game state at the moment it will act upon it differently. Firstly the information is read and parsed (through the *mouse_ih* and *ps2_parse_packet*), then being acting according to it. It is firstly visually updated, by summing the offset in x and y coordinates, with which we can take conclusions about its position (done on *mouse_update*).
  All other functions are related to what will happen if any of the present buttons on the current screen are pressed.



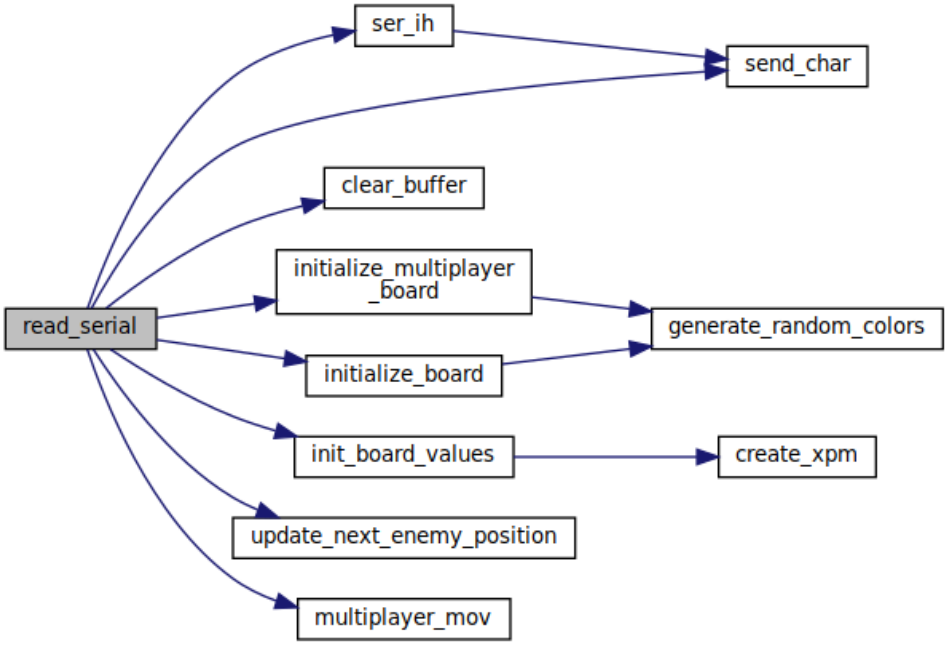
- **RTC**
  The *rtc_ev* function is responsible for determining when the enemies should move.



- **Keyboard**
  The *kbc_mov_IH* function implements a state-driven methodology with two modes: Game and Game Over.
  In the first one, it is responsible for moving the Qbert on the board, depending on the user input. In the last one, it is responsible for leaving the Game Over screen and returning to the Main Menu, if the user presses any key.



- **Serial port**
  The *read_serial* has the function to act upon the information being received from the serial port. Thus, all functions presented are only called through interrupts of the type “Received Data Ready”. It is important to state the importance of the *update_next_enemy_position* as well as *multiplayer_mov* which are the ones who will handle the change in information as Player 1 when receiving information from Player 2.



interrupt_handling.c

Overall Weight: 3%
Each members contribution: André: 100% - Joana: 0%
Description: This module was created to handle interrupts, therefore it is responsible for subscribing and unsubscribing them and for calling the specific handlers for each peripheral device (defined in *event_handlers.c*).

keyboard.c

Overall Weight: 4%
Each members contribution: André: 50% - Joana: 50%
Description: This module was implemented in Lab3 and is used in this project to handle keyboard interrupts.
The function *is_scancode_make* is used in the Game Over screen to avoid leaving the screen too fast by accident.

menu.c

Overall Weight: 2%
Each members contribution: André: 100% - Joana: 0%
Description: This module was created to draw all the menus, ‘Game Over’ and ‘Credits’ screens.

proj.c

Overall Weight: 1%
Each members contribution: André: 50% - Joana: 50%
Description: This module is the first to be called and, thus calls for the initialization of the video mode and starts the game.

ps2Mouse.c

Overall Weight: 4%
Each members contribution: André: 50% - Joana: 50%
Description: This module was implemented in Lab4 and is used in this project to handle mouse interrupts.

rtc.c

Overall Weight: 3%
Each members contribution: André: 100% - Joana: 0%
Description: This module handles the RTC interrupts as well as reading and interpreting all of the information coming from their registers.

serial_port.c

Overall Weight: 4%
Each members contribution: André: 100% - Joana: 0%
Description: This module handles the Serial Port interrupts as well as handling and interpreting their outcome. It is also used to send communication between devices, using the FIFO's.

timer.c

Overall Weight: 4%
Each members contribution: André: 50% - Joana: 50%
Description: This module was implemented in Lab2 and is used in this project to handle Timer interrupts.

vbe.c

Overall Weight: 1%
Each members contribution: André: 100% - Joana: 0%
Description: This module was developed in Lab5 and is used in this project to check the video mode.

video_graphics.c

Overall Weight: 4%
Each members contribution: André: 50% - Joana: 50%
Description: This module was developed in Lab5, with slight modifications to accommodate the use of different sprite structures with more information per pixel, and is used in this project to handle the buffer and video graphics.
It is also specifically used to draw the squares of the board as well as all of the sprites used during the whole game.

video_settings.c

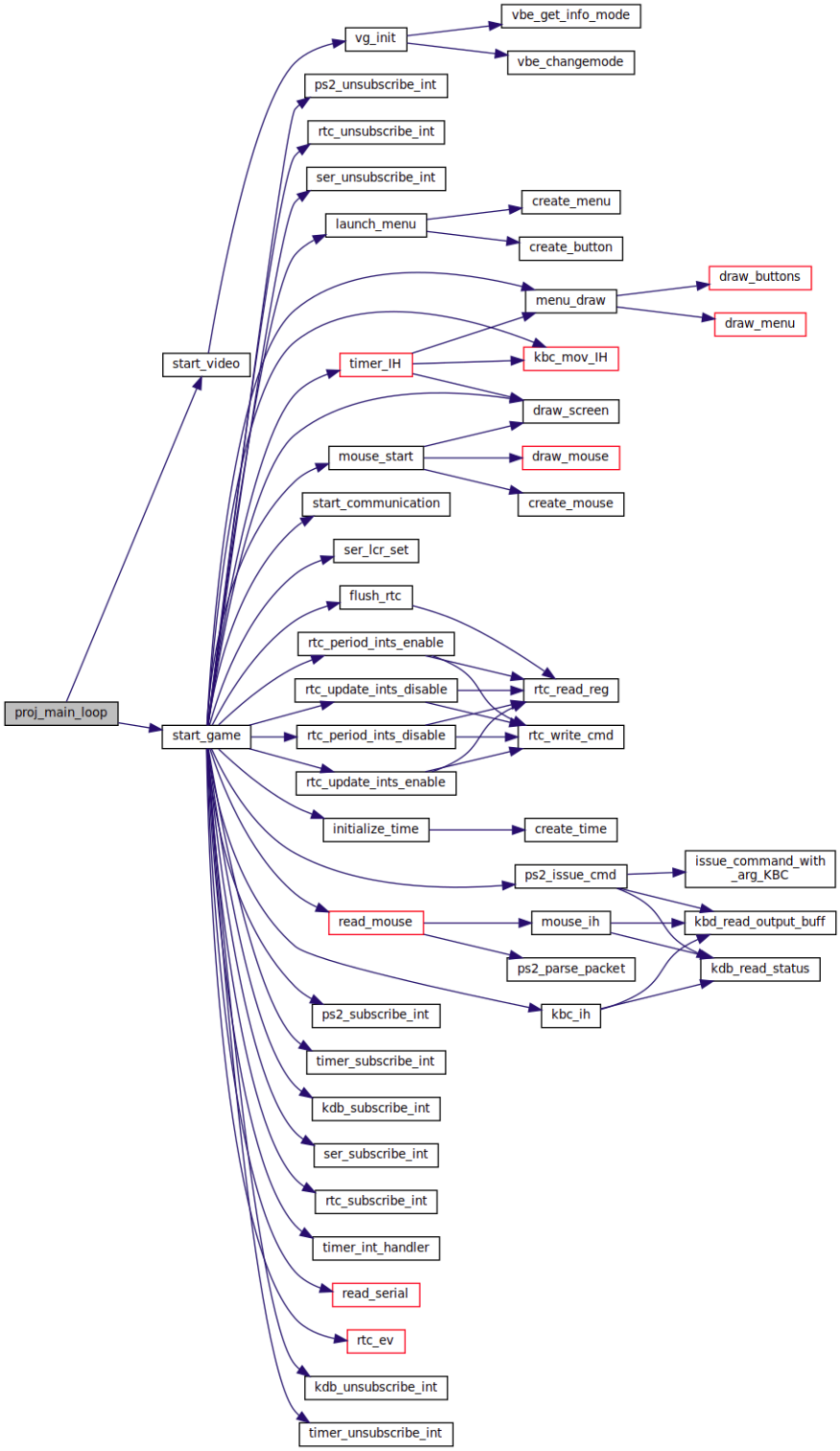
Overall Weight: 1%
Each members contribution: André: 100% - Joana: 0%
Description: This module was created to initialize the video settings needed for this process.

xpm.c

Overall Weight: 15%
Each members contribution: André: 90% - Joana: 10%
Description: This module has a series of functions to parse the information coming from an image of the standard XPM as well as handling the writing of them and their animations. To process the XPMs we used *read_xpm*, prepared by Professor Pedro Silva.
This mode also has functions to create each character, using the struct *character_t* below (defined in *game.h*), and to update each speed in order to move and animate them, by selecting a different XPM.
Furthermore, all of the sprites used for the game, from the menus, to the buttons and the mouse, are also animated and drawn in this module.

```
typedef struct{
  xpm_image_t img; /**< The image of the character */
  int x, y; /**< The x and y position of the character in the buffer */
  int x_board, y_board; /**< The x and y position of the character in the board*/
  int xspeed, yspeed; /**< The x and y speed of the character*/
  int type; /**< The type of the character*/
  xpm_image_t transition_img; /**< The animated image of the character */
  bool transitioning; /**< True if the character is currently in the middle of an
animation*/
}character_t;
```

Function call graph

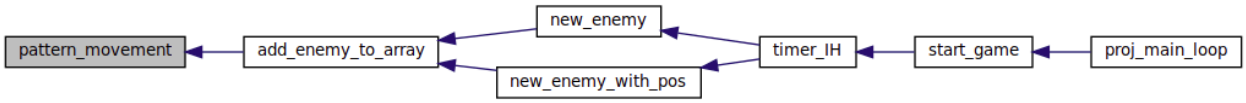


This graph's length is limited.

4. Implementation details

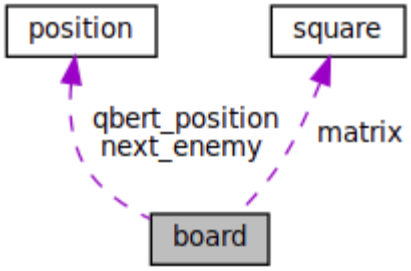
State Machine

The game implementation is a State Machine, as depending on the state (Singleplayer or Multiplayer and Player 1, Multiplayer and Player 2, Main Menu and Game Over), the interruptions produce different results.
For instance, as seen below, when there is an interrupt from the RTC, the game creates a new enemy, however, if it is in Singleplayer mode, *new_enemy* is called with a random position. In contrast, in multiplayer mode, the handler calls *new_enemy_with_pos* with the initial position previously selected by Player 2.



Object-Oriented Programming

To implement this game we chose an Object-Oriented approach. For this, we used ‘structs’, ‘typedef’ and some OOP concepts.
One example of this is the struct ‘board’ which depends on the structs ‘square’ and ‘position’ (below). We also build a class for the characters.



Contrasting colors

Once we wanted to guarantee the initial two colors of the board would contrast (be very different - almost opposite) but at the same time were random, we had to find a way to generate a contrasting color from a randomly selected one. After several tries, we ended up using bitwise operations to do it: XOR with 0xffffffff.

UART Implementation

During the development of the serial port, it was needed to learn a series of concepts related to the development of a communication protocol that led to our method to communicate between two computers. Since the serial port itself has an enormous amount of errors associated with it, it was necessary to build a solid protocol which could be application independent if the need came.
Inspired in the logic of webhooks, our communication protocol is able to act upon “listening” to a new event, given by a interrupt of the UART. This way, our state machine would receive outside information which would change the course of its state as a whole.
In order to provide stability to the visual part, for every information sent from Player 2 to Player 1, an **ACK** (acknowledgment) is sent in order to actually change any of the given visual information currently provided in the board of Player 2.

5. Conclusions

To conclude, we do consider this an important course, as well as acknowledge the value of the information it does indeed transmit to its students. Besides the important knowledge of how a computer works, it does transmit valuable knowledge on low-level programming, much than any course until this date.

However, it is still needed to point out its lack of initial preparation, since most of the concepts are shown for the first time, and there is a big gap towards anything we had encountered before. Moreover, the lack of information on online platforms, due to the use of a specific less known OS, makes it so that it is considerably hard to start working on the first lab. It is, therefore, necessary to give credits to the appointed monitors of each class, whose experience with the course makes it one of the most valuable assets at the beginning.

Moreover, it is also important to point out the amount of work this course accounts for, which is definitely not proportional to the appointed number of credits. It has significantly more work than any other 6 credits course, and even more than AEDA, whose credits are 7.5.

6. Appendix - Installation Instructions

In order to execute our project, firstly it is needed to give the necessary permissions to the executable. For that from the **proj** directory, it is needed to run the following command: “lcom_conf add conf/proj”.
Afterwards, it is necessary to enter the **src** directory with the command “cd src”, in order to finally run the “make” command in order to compile the project itself.