LCOM Project - Group T15G5

Group Members

- Rafael Cunha (up202208957@fe.up.pt)
- Vasco Melo (up202207564@fe.up.pt)
- Gabriel Carvalho (up202208939@fe.up.pt)

User Instructions

Running the Game

To run the game, in MINIX, navigate to the project's 'src' folder and enter the following commands:

1st command - make clean

2nd command - make

3rd command - lcom_run proj.

Game Menu

Upon launching the game, you will be presented with the following Menu:

From here, you can choose to exit or play in single or multiplayer mode using arrow keys/enter or your Mouse.

Game Modes

- **Singleplayer Mode**: The game starts immediately upon selection.
- **Multiplayer Mode**: A request will be sent to another Minix user through the Serial Port. This request can be accepted or declined by the recipient. Keep in mind that you need to configure another Virtual Machine in order for the multiplayer mode to work.

Point of view of the player who invited someone:

Point of view of the invited player:

Gameplay

The game is a Portuguese "Sueca" card game. Players can use the arrow keys and enter to choose their card, or grab and drop it using the mouse. The game follows the traditional "Sueca" rules.

Upon starting a singleplayer match, you will be presented with the following Interface:

After sliding to choose an amount, you will be asked to choose a card from the top or bottom of the deck. This card's Suit will determine the Trump Suit for the round;

Then, the game itself will start:

Example of a Multiplayer match, side by side:

Players can exit to the main Menu by clicking ESC and choose to exit the game in the ESC Menu. All of these actions can be made both through the use of Keyboard and Mouse:

Upon stopping the game, you will have access to the elapsed time you've been playing for:

Game Rules

Players and Setup

The game is designed for 4 players, divided into 2 teams of 2. The game uses a standard 40-card deck (8s, 9s, and 10s are removed from a regular 52-card deck).

Card Ranking and Values

- **Ranks**: (from highest to lowest) Ace, 7, King, Jack, Queen, 6, 5, 4, 3, 2.
- **Values**: Ace: 11 points, 7: 10 points, King: 4 points, Jack: 3 points, Queen: 2 points, Other cards: 0 points

Gameplay

- **Dealing**: Each player is dealt 10 cards.
- **Trump Suit**: The card that the player got in the beggining of the game determines the Trump Suit.
- **Tricks**: Players play one card each in clockwise order. The player who wins the trick leads the next one. Players must follow the suit of the leading card if possible. If not, they can play any card. The highest card of the leading suit wins unless a trump card is played, in which case the highest trump card wins.

Scoring

During the game, the score is calculated and displayed in the top-left corner, being updated after each trick ends. The game itself is endless, which means that there will always be a new round, until the player(s) decide that they do not wish to play anymore.

Project Status

Used Devices Table

Device	What for	Interrupts
Timer	Refreshing the frame buffer and applying State Machine Changes	Y
Keyboard	Getting keyboard inputs from users	Y
Mouse	Getting mouse mouvement and gestures from users	Y
Video Card	Displaying different menus and screens through Sprites	N
RTC	Keeping track of total game time	N
Serial Port	Implementing multiplayer mode between two Minix users	Y

Timer

The timer is used in the main game loop (loop_interrupcoes()) to set the game's frame rate, synchronize game events, update the game state, and unsubscribe from the timer interrupt at the end of the game. It plays a crucial role in managing game states and events. Here's a brief summary of each function:

Functions

- process_timer(): Handles game timer events based on the current game state, including main menu timer, game timer, multiplayer game timers, and mouse events.
- process_main_menu_timer(): Manages main menu timer events and updates game states accordingly.
- process_game_timer(): Handles game timer events and updates game states for single-player mode.
- process_round(): Manages the processing of a game round, including determining the winner, resetting selected cards, and updating game states.

- process_display(): Handles the display of the game based on the current round, including card selection and removal.
- process_game_timer_P1(): Manages game timer events and updates game states for Player 1 in multiplayer mode.
- process_game_timer_P2(): Handles game timer events and updates game states for Player 2 in multiplayer mode.
- process_round_MUL(): Manages the processing of a round in multiplayer mode, including determining the winner and resetting game states.
- process_display_P1(): Handles the display processing for Player 1 in multiplayer mode, including card selection and removal.
- process_display_P2(): Manages the display processing for Player 2 in multiplayer mode, including card selection and removal.

Keyboard

The keyboard is primarily used for navigation (application control) and selection within the game's menus and during gameplay. The main function, loop_interrupcoes(), handles keyboard interrupts by subscribing to them and calling keyboard_ih and `process_teclado functions.`

Functions

- process_teclado(): Processes keyboard inputs based on the current game state, including main menu, connection, invitation, and gameplay.
- process_main_menu_teclado(): Manages keyboard inputs in the main menu for navigation and actions on button presses.
- process_game_teclado(): Handles keyboard inputs during the game for menu navigation, actions, and card selection in single and multiplayer modes.
- process_select_card_P1(): Manages card selection for Player 1 using arrow keys and the ENTER key.
- `process_select_card_P2()` : Manages card selection for Player 2 using arrow keys and the ENTER key.
- process_connection_teclado(): Processes keyboard inputs during the connection phase for navigation and actions.
- process_invitation_teclado(): Handles keyboard inputs during the invitation phase for navigation and actions.
- process_prepare_round_teclado(): Handles keyboard inputs during the round preparation phase.

Mouse

Handles interactions via mouse movements and clicks. The program supports mouse-driven navigation buttons, considering cursor position, supports drag-and-drop functionality for cards and also sliding functionality. All of this is done through the concept of position and collisions.

Functions

- process_rato(): Processes mouse inputs based on the current game state and mouse coordinates.
- process_rato_grab_drop(): Processes card grab and drop actions, checking for collisions and game state.
- process_rato_P1(): Processes mouse inputs for Player 1, based on the game state and mouse coordinates.
- process_rato_P2(): Processes mouse inputs for Player 2, based on the game state and mouse coordinates.
- process_rato_connection(): Processes mouse inputs during the connection phase, handling cancel button clicks.
- process_rato_invitation(): Processes mouse inputs during the invitation phase, handling accept or decline button clicks.
- process_rato_get_trump(): Processes mouse inputs in a specific game context related to "getting trump".

Video Card

Draws pixels on the screen, accommodating both XPM sprites. We simulate the use of fonts through these XPM sprites. The game's resolution is 800x600 pixels and it operates in the 0x115 video mode, which supports 24-bit (8:8:8) direct color. The implementation of double buffering was done via copy. Moving objects were dealt with through collision detection.

Functions

- `modo_grafico()` : Sets Minix in graphic mode.
- `const_buffer_frame()` : Creates physical and virtual buffers for the frame in the specified mode.
- `desenhar_pixel()` : Draws a pixel at the specified coordinates with the specified color.
- `printar_form_xpm()` : Prints a form from an XPM at the specified coordinates.
- `printar_form_xpm_sprite()` : Prints a form from an XPM sprite at the specified coordinates.
- `printar_form_xpm_sprite_rotated()` : Prints a rotated form from an XPM sprite at the specified coordinates.

- `molduraCarta()` : Draws a card frame at the specified coordinates with the specified color.
- `criarBufferSetDesenho()` : Creates a drawing set buffer.
- `copiarbufferSetDesenhoToBufferGrafico()` : Copies the drawing set buffer to the graphic buffer.
- `limparBufferGrafico()` : Clears the graphic buffer.

VBE

In the reg86_t struct, through the Minix sys_int86 function, we used the following parameters:

reg86.intno = 0x10 reg86.ah = 0x4F reg86.al = 0x02 reg86.bx = submodo | (1 << 14)

Real-Time Clock (RTC)

Used to read date/time information. It's main purpose is to make it possible to display the total game time elapsed at the end of the round in the game.

Structures

r tc_struct

This structure holds the RTC time data. It includes:

- uint8_t seconds: The seconds part of the time.
- uint8_t minutes: The minutes part of the time.
- uint8_t hours: The hours part of the time.

Functions

- `rtc_read_set()` : Reads and sets the RTC time.
- `rtc_read_output()` : Reads the output of the RTC based on the sent command.
- `rtc_get_time()` : Gets the current time from the RTC.

Serial Port

Features Used

Interrupts

The UART employs interrupts to signal when data is available for reading or writing. `Functions such as serial_port_subscribe_interrupt() and` serial_port_unsubscribe_interrupt() manage subscription and unsubscription from these interrupts.

FIFOs

The FIFO structure, represented by fila_sp, includes an array of integers (valores), maximum capacity (capacidade), start index (inicio), end index (fim), and current size (tamanho).

- criar_fila_sp(): Creates a new queue, initializing the fila_sp struct and allocating memory for `valores`.
- `inserir_fila_sp(status, valor)`: Inserts a value into the queue if data is available, using `colocar_fila_sp()` if possible.
- `colocar_fila_sp(valor)`: Inserts a value into the queue at fim, increments fim and `tamanho`.
- `limpar_fila_sp()`: Clears the queue by resetting `inicio`,`fim` , and `tamanho`.
- `pop_fila_sp()`: Removes a value from the queue at `inicio`, increments `inicio`, and decrements `tamanho`, then returns the removed value.

Data Exchanged and Exchange Frequency

Functions enviar_byte() and receber_byte() handle the transmission and reception of single-byte data via UART. The exchange frequency depends on the application's requirements.

Communication Parameters

`The serial_process(), serial_process_main_menu(), serial_process_P1(), and` serial_process_P2() functions manage serial inputs during different game states. They execute actions accordingly to the game's current state. The frequency of data exchange in these functions is determined by the game state and associated actions.

Code Structure

Modules

Timer

Same implementation as the one done in lab2.

Percentage weight of this module in our project: 10%

Mouse

The mouse works mostly like it does in lab4. We synchronize bytes received from the mouse (mouse_sync_bytes), convert byte arrays into a packet structure (mouse_bytes_to_packet), and write commands directly to the mouse (write_mouse).

Percentage weight of this module in our project: 7%

Video Card

Mostly the same implementation as in lab5, but includes some extra functions and adjustments for specific needs in the game.

Percentage weight of this module in our project: 15%

RTC

In our RTC module, the rtc_read_set function reads the RTC mode and time, interacting with the RTC's registers. The rtc_get_time function reads the current time, checking the RTC's status. If the RTC is in BCD mode, time units are converted to binary. The rtc_time structure and mode variable store the current time and RTC mode, respectively.

Percentage weight of this module in our project: 1%

UART

Manages the serial port. It uses the Line Status Register (UART_LSR) to read the port status and the Interrupt Enable Register (UART_IER) to control interrupts and manage a queue. Data transmission is handled through the COM1 port.

Percentage weight of this module in our project: 4%

interrupt

Subscribes and unsubscribes to interrupts for every module in the project that uses interrupts. That is, Timer, Keyboard, Mouse and UART.

Percentage weight of this module in our project: 1%

handler

Handles every interrupt that the "Interrupts" module receives

Percentage weight of this module in our project: 1%

aux_dispositivos

This module includes auxiliar functions for every module above.

Percentage weight of this module in our project: 4%

aux_func

Handles tasks like converting RTC time and scores to integer arrays, converting numbers to sprites, managing and using a random number generator, and calculating time spent between two RTC timestamps.

Percentage weight of this module in our project: 1%

collision

Defines the structure and functions for handling collision detection. It includes functions for allocating and freeing collision arrays, checking if a point is inside a collision area, and specific functions for checking collisions with various buttons and other UI elements.

Percentage weight of this module in our project: 5.5%

sprite

Manages the lifecycle of sprites in the game. It handles the creation of sprites from XPM files, their storage in memory, and their removal when no longer needed.

Percentage weight of this module in our project: 13%

composer

Through the help of the sprite module, this module is responsible for correctly showing the entirety of the User Interface in the screen. It also defines most of the state machines needed for the game to work properly.

Percentage weight of this module in our project: 18%

rules

Defines the data structures for Cards and Players and declares functions for managing the game rules and actions.

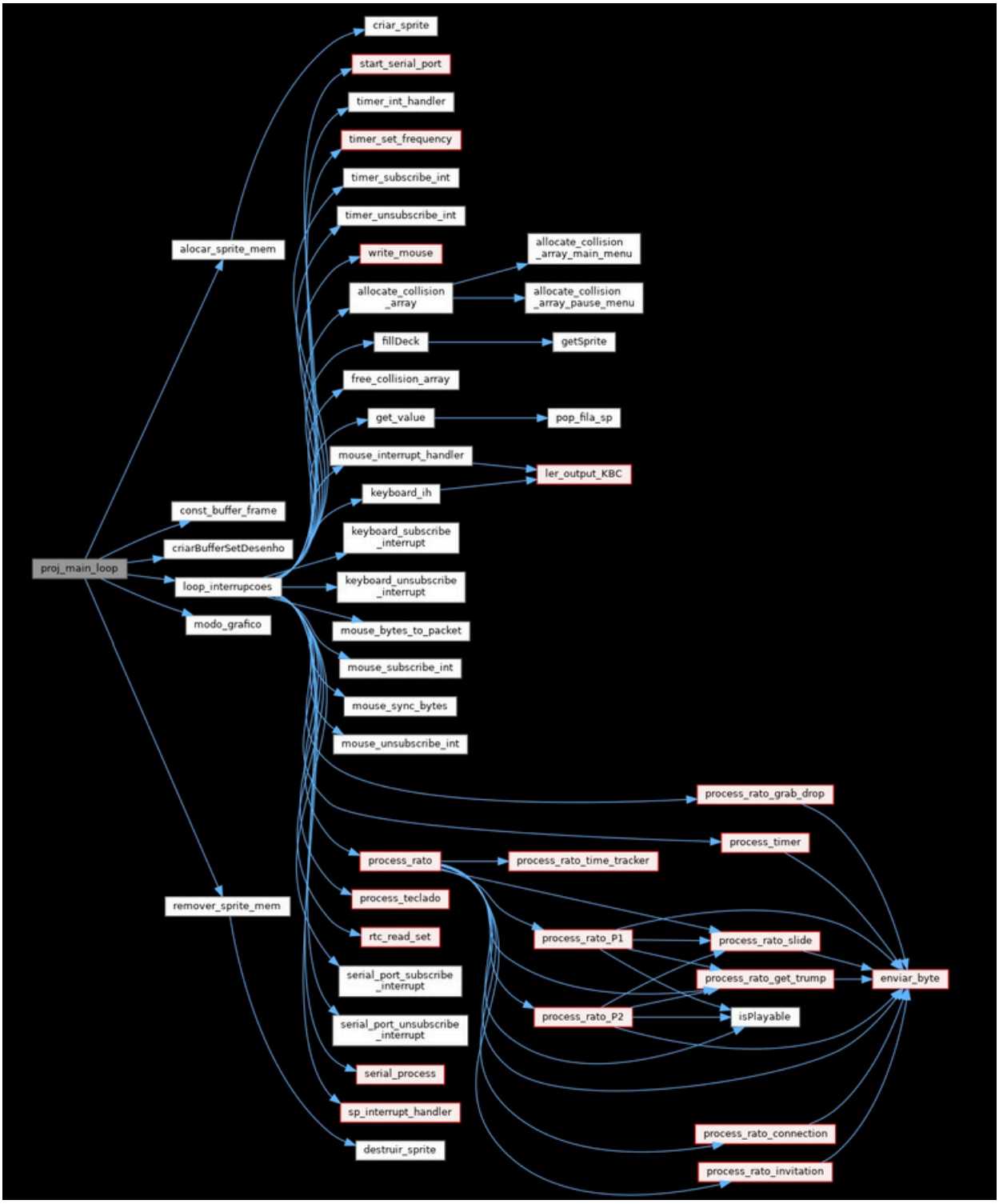
Percentage weight of this module in our project: 12.5%

game

Manages the main game loop and hardware interrupts from the timer, keyboard, mouse, and serial port. It initializes, subscribes, processes, and unsubscribes from these interrupts, handling errors along the way, all of this is done through the direct or undirect help of every other module in the project.

Percentage weight of this module in our project: 7%

Function Call Graph



Implementation Details

State Machines

State machines are responsible for the application flow, including the display of menus, the movement of different objects, etc.

The composer module defines several enums for state machines that manage the different aspects of the game:

`state_machine_game` : Transitions between main menu, game, end, and multiplayer modes.

state_main_menu: Interactions with start, multiplayer, and exit buttons.

state_game: Game initialization, start, end of round, score calculation, and round preparation.

state_menu_pause: Interactions with resume, exit, and restart buttons during pause.

game_round: States for different players and waiting periods in a game round.

grab_state: The process of grabbing in the game.

state_multiplayer: Determines who the different players in a multiplayer game are

state_sp: Represents the waiting state for a Serial Port Interrupt from another player before a multiplayer game starts.

state_connection: Represents the different states of the multiplayer connection menu.

state_invited: Represents the different states of the multiplayer invitation menu.

state_select_card: States of the top or bottom card selection menu.

state_split: States of the deck splitting menu.

state_time_tracker: States of a time tracker.

state_prepare: States of game preparation, including shuffling, trump selection, and sliding.

Multiplayer Details (UART)

Due to the amount of complexity that synchronizing two games simultaneously implies, we needed to minimize the amount of information transmitted between the Virtual Machines to reduce delays and memory losses during this process. Consequently, the game creates two individual instances, with communication limited to identifying the players (player 1 and player 3) and their respective card choices. Additionally, if a player leaves or resets their game, the same action is mirrored for the other player to maintain game synchrony.

Problem: How will both Virtual Machines maintain the same cards for the two players, given the random shuffling of the decks?
Possible Solution: When a player shuffles a deck, send the card information, one by one, to the other player.

Issue with Possible Solution: This could lead gto asynchrony between the two Virtual Machines and memory loss, as it would require transmitting a large amount of card data and information.

Our Solution: We created a "seeded" deck shuffler that works through pseudo-random numbers. If the seed is the same between two players, it will generate the exact same shuffle for their respective decks. Thus, Player 1's Virtual Machine generates a random seed and sends it to Player 2's Virtual Machine, ensuring identical deck shuffling.

Elapsed Playing Time (RTC)

For both single and multiplayer modes, we use the Real-Time Clock to store the exact time when players start and stop their match. When they leave the game, we calculate the duration of their playtime by subtracting the start time from the end time. This allows us to display the total amount of time (in minutes) that they have played.

Layering

We make use of sprites to display everything in our game. For button, slider, grab, etc., responsiveness, it was necessary to separate the different elements into distinct sprites to show the appropriate sprite based on the user's actions. Therefore, creating Layers in the frame buffer was crucial for displaying the intended content on the screen.

How we did it:

We interpreted each screen as a set of elements in a frame. In our program, each Timer interrupt generates a completely new frame from scratch, deleting the previous frame. The process works as follows:

1. **Background Creation:** The program first creates the background.
2. **Element Addition:** Then, it adds elements that are not part of the background.
3. **Mouse Overlay:** Finally, the mouse sprite is appended over all other elements.

Only after the entire frame is built is it sent to the frame buffer to be shown on the screen. All of this is done within Timer Interrupts.

Synchronous Keyboard and Mouse functionality

Since the game is based on options that include choosing Menus, Cards, etc., we made it possible for the user to choose between using the Keyboard or Mouse at all times. Switching from Mouse to Keyboard or vice versa could confuse the user if the options previously chosen with the first device did not reset after switching devices. To address this, we ensured that when the user changes devices, the options from the previous device reset, making the game UI more user-friendly.

How we did it:

We created a Mouse-On and a Keyboard-On system. If the user was previously using the Keyboard and switches to the Mouse, the system turns off the Keyboard and only shows and activates the Mouse options. The same happens in reverse: switching from Mouse to Keyboard will deactivate the Mouse options and only show and activate the Keyboard options.

Conclusions

Problems

Minix Time Miscalculations:

Due to the fact that the Minix operating system itself miscalculates elapsed time, our RTC functionality for calculating the time at the end of the game may not return the correct amount of time. However, this is a Minix issue, and there is nothing we could do to resolve it.

Like to haves

End of Round Menu:

We designed and almost implemented an end-of-round menu that would display the winning team color, as well as the scores after each round, and ask the user if they wanted to play again or exit the game. However, due to lack of time, we did not complete its implementation, so we decided to leave it behind.

Lessons Learned

This project taught us extensive lessons on developing software from scratch, particularly in Low-Level language contexts. We also gained hands-on experience in creating a fully functional game with hardware functionality implemented from the ground up.