2021/22

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Laboratório de Computadores

Grupo 7 – Turma 13

Chess Game Coded in C

Licenciatura de Engenharia Informática e Computação

Marco André Rocha	up202004891@up.pt
Ricardo André de Matos	up202007962@up.pt
Pedro Gomes	up20200xxxx@up.pt
Guilherme Freire	up20200xxxx@up.pt

Índice

1. - User instructions

This project consists of an **animated chess game** fully coded in C, using, for that, the hardware interface of the PC peripherals taught during our LC classes (embedded SW).

The game logic follows the core chess mechanics, however, **custom game rules were introduced** (e.g. instead of checkmate, to win the game, the enemy king must be taken).

The piece's movements are like the inspirational game but we added **custom animated sprite pieces** to bring this millennial old game to life. Furthermore, when taking an enemy piece, an explosion animation takes place.

The game can be played in **two different modes** that can be chosen in the start menu: *multiplayer* and *online*. In the first one, two players can play using a single machine, taking turns. In the online mode, thanks to the **serial port**, two Minix images can run the game simultaneously.

When connected via a serial port, players can also **send messages** to each other by just typing with the keyboard and sending them by pressing the space key.

Each player starts with 5 minutes on their respective clocks. If the time runs out, the other player wins. To count the time, the **RTC was used** but we'll go into further detail later.

The **piece's movement** can be done using a **mouse click & drag** into the new position or a mouse click on the desired piece and a second click onto its new position. The movement only takes place if it corresponds to one of the previously computed valid moves for that piece (shown with a blue circle).

Menus navigation can be achieved by using the **mouse** or the **keyboard** (*arrows* and *return)*. Mouse **animations while hovering** the buttons or selecting with the keyboard were also implemented.

To run the project just use the "*make*" command followed by "*lcom_run proj*".

Next, some images representative of the project's main features will be presented.

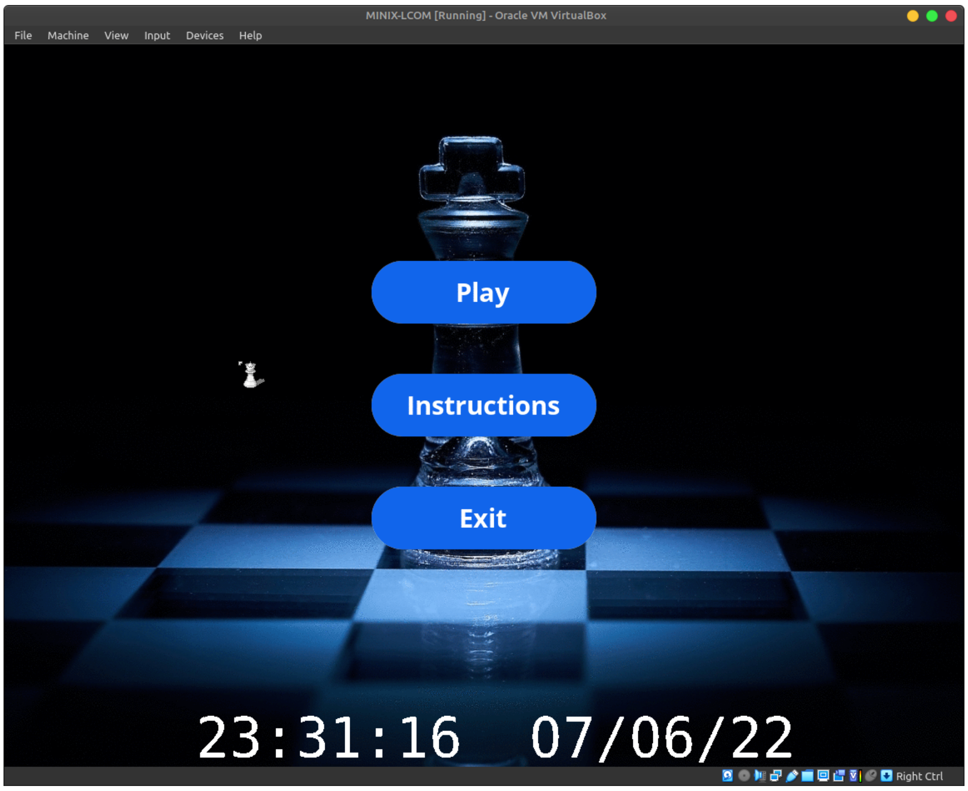
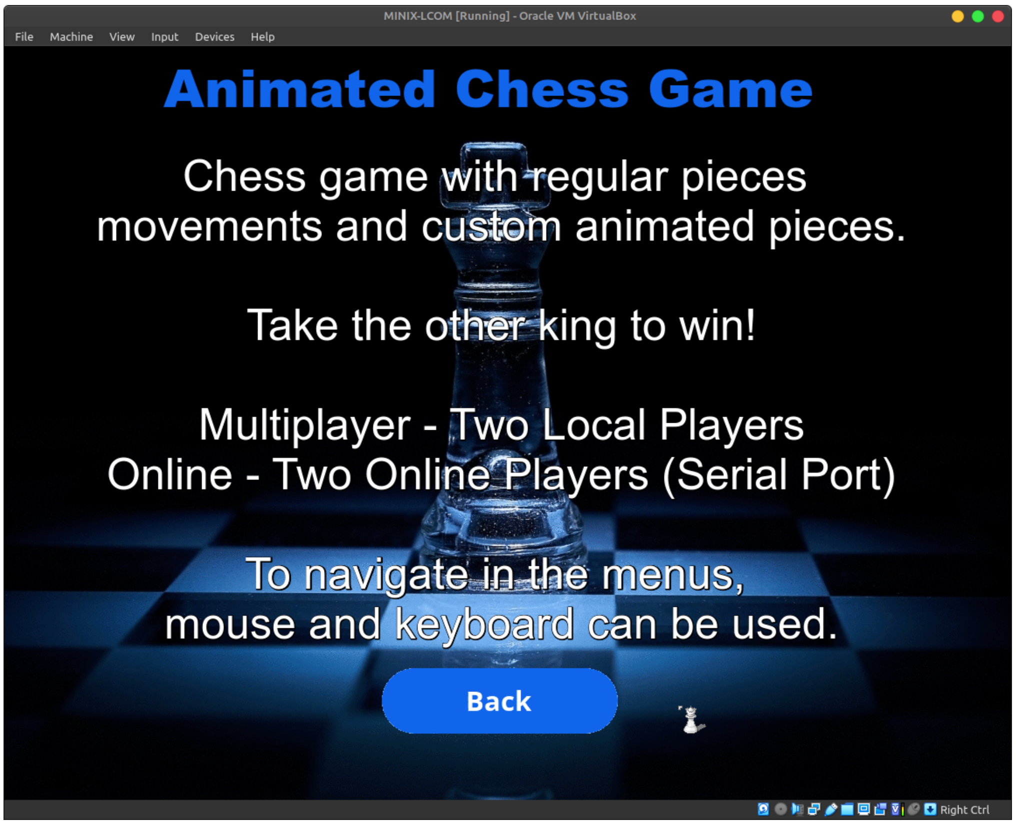

MINIX-LCOM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Play
Instructions
Exit
23:31:16 07/06/22
Right Ctrl

Image 1. Main Menu with RTC



MINIX-LCOM [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Animated Chess Game

Chess game with regular pieces movements and custom animated pieces.

Take the other king to win!

Multiplayer - Two Local Players
Online - Two Online Players (Serial Port)

To navigate in the menus, mouse and keyboard can be used.

Back

Right Ctrl

Image 2. Instructions Menu

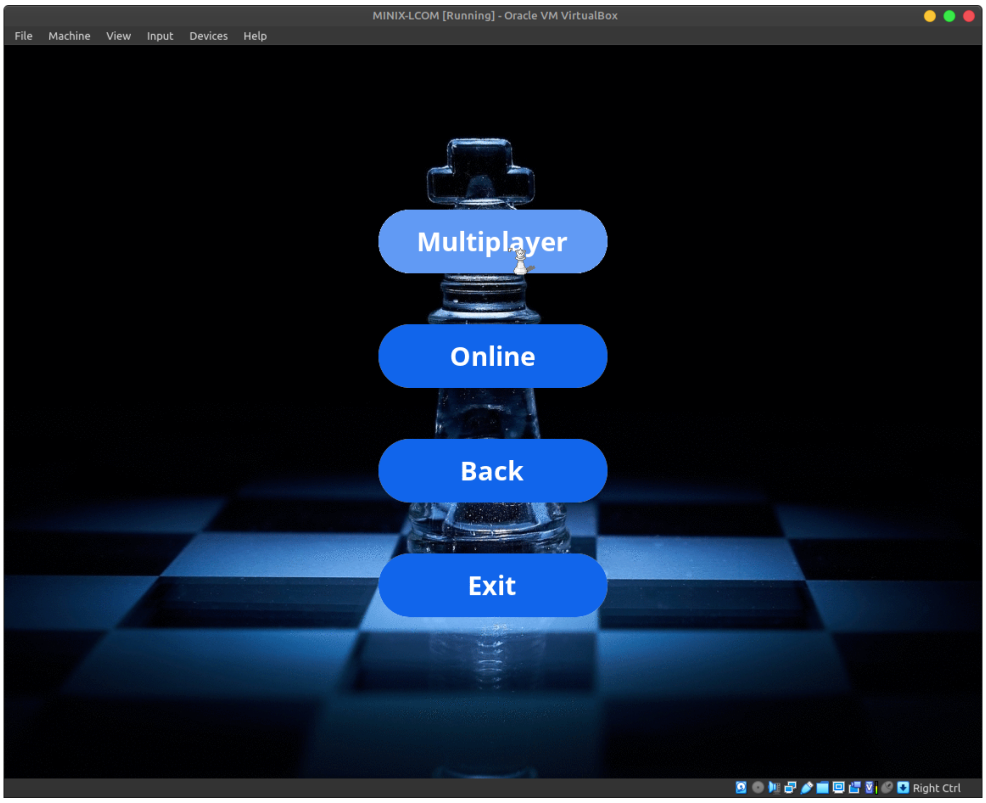

MINIX-LCOM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Multiplayer
Online
Back
Exit
Right Ctrl

Image 3. Game Menu and button hovering

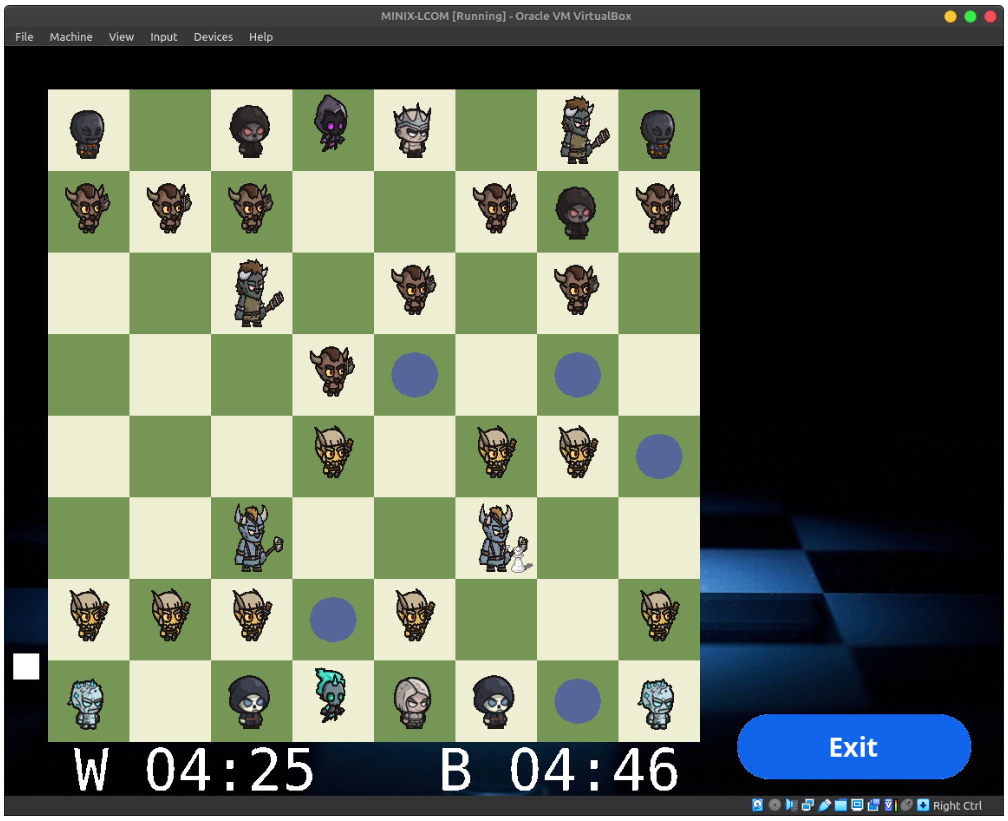

MINIX-LCOM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
W 04:25
B 04:46
Exit
Right Ctrl

Image 4. Gameplay and valid moves

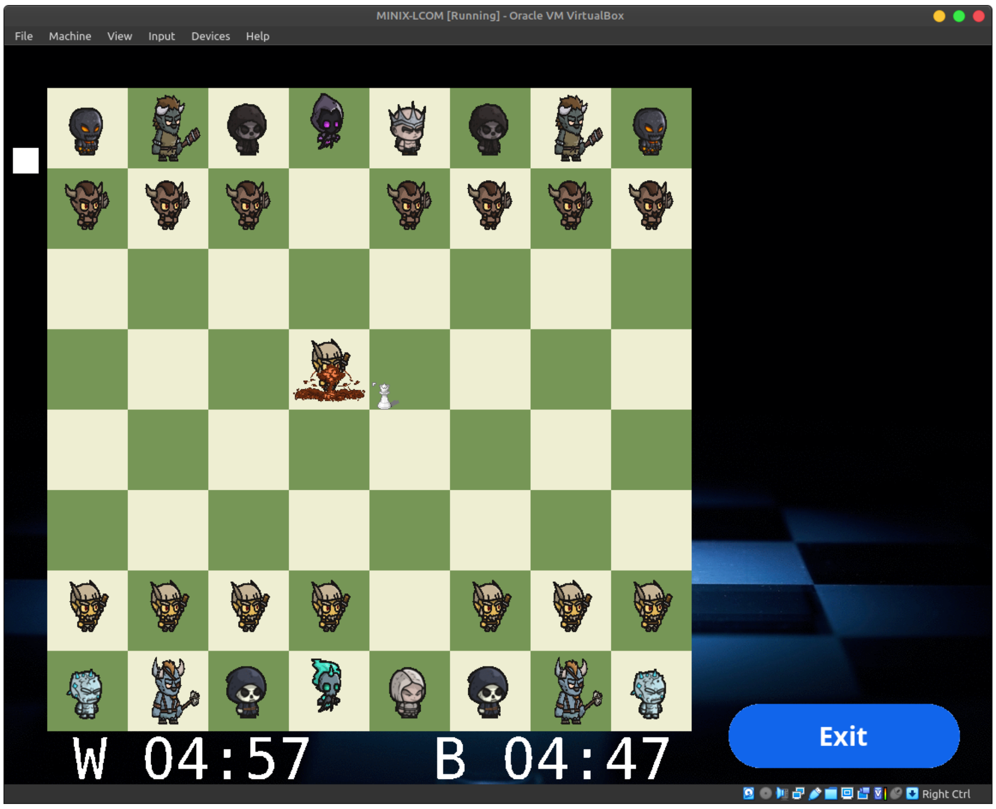

MINIX-LCOM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Exit
W 04:57
B 04:47
Right Ctrl

Image 5. Explosion Animation

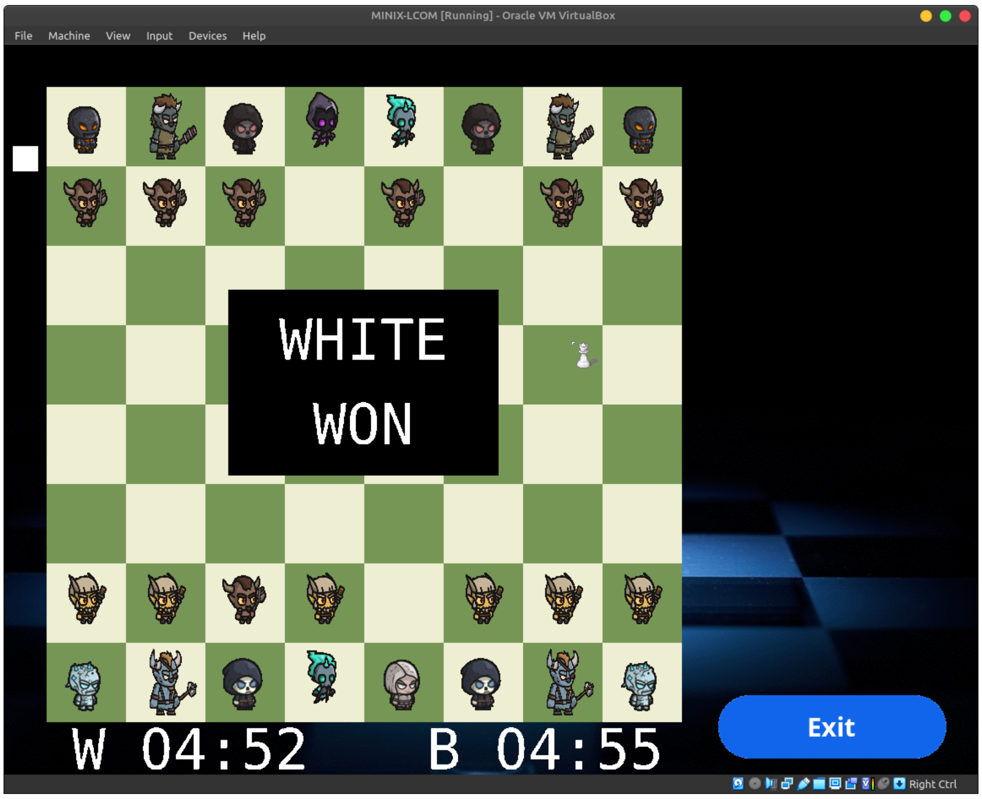

MINIX-LCOM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
WHITE WON
Exit
W 04:52
B 04:55
Right Ctrl

Image 6. Winning Message

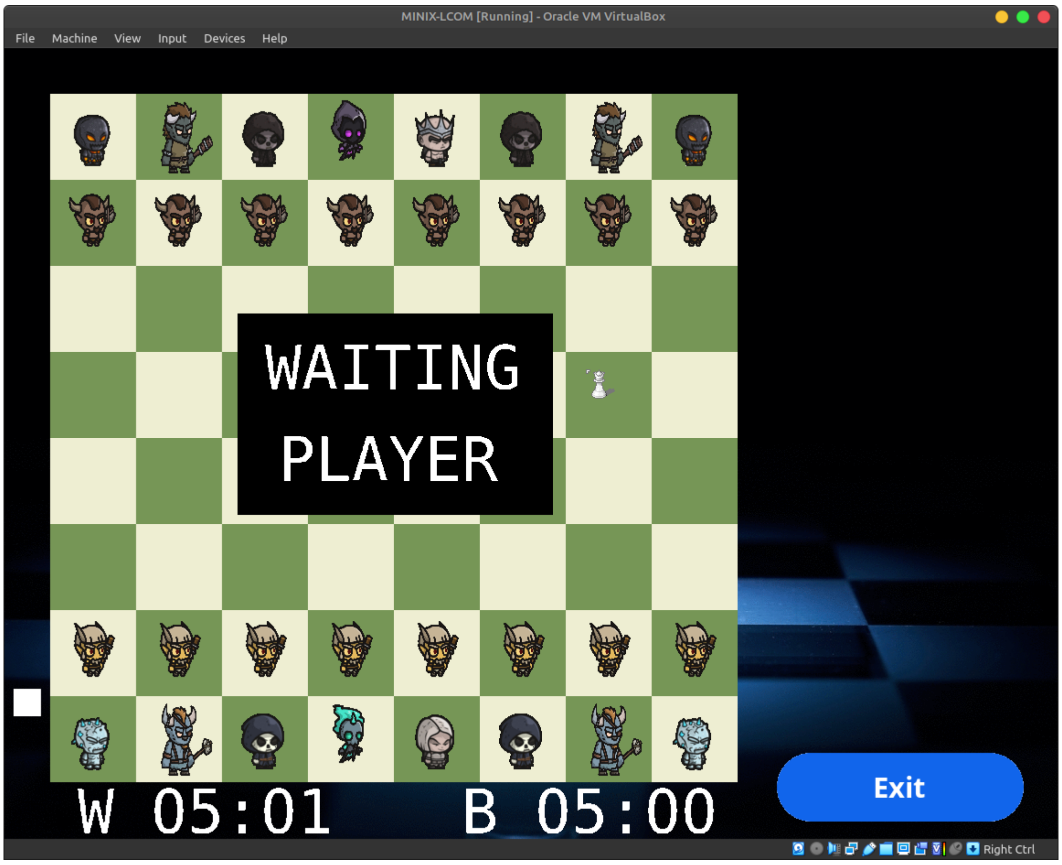

MINIX-LCOM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
WAITING PLAYER
W 05:01
B 05:00
Exit
Right Ctrl

Image 7. Awaiting Connection Message

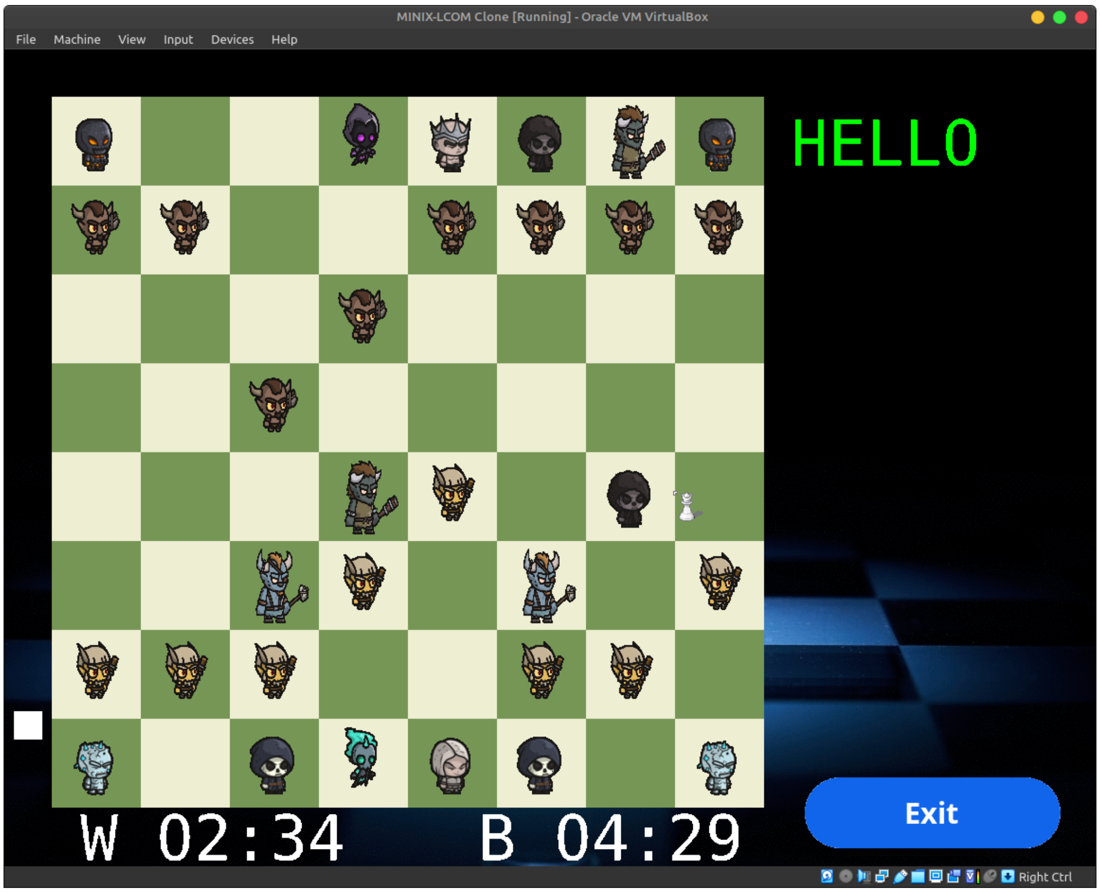

MINIX-LCOM Clone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
HELLO
W 02:34
B 04:29
Exit
Right Ctrl

Image 8. Online Chat

2. - Project status

The following table indicates the implementation state of all the devices used:

Device	What for	Interrupt / Polling
Timer	Controlling frame rate Animation tempo Check state & display	**Interrupt**
KBD	Menu / Game Select Opt. Chat online End Game	**Interrupt**
Mouse	Menu / Game Select Opt. Grab piece and move Pick piece and move	**Interrupt**
Video Card	Menus Game Display w/Animations and fonts	N / A
RTC	Menu Time Chess Clocks	**Interrupt**
Serial Port	Online Gaming Chat	**Interrupt**, receive data, transmitter empty and line status

Table 1. Devices Implementation

2.1 - Timer

Timer interrupts are used for

- Define a Frame rate (60hz).
- Update Animation. Some animations have different speeds of change from the others. We use the timer for that too.
- Check the state of the game to know which state to display.

Check /proj/src/drivers/timer/timer.c for the driver's code. Check **handle_ih()** in /proj/src/game/int_handlers/independent/independent_ih.c to see the independent interrupt handling.

Furthermore, check **EVENTS handle_timer_evt(EVENTS event)**, in /proj/src/game/int_handlers/dependent/dependent_ih.c**,** and the calling functions to see how we update the animation, and how we know which state we should display.

2.2 - Keyboard

Keyboard interrupts are used for

- Selecting menu and Game Options corresponding to the return codes that we see in 4.2. To select an option you can press the **Up** or **Down** arrows until your option is with the hoovering effect and then press **“Enter”.** This effectively changes the game state.
- Sending messages to other users, the message must contain letters from **‘A’ to ‘Z’**, to send the message using **“Space”**
- **Force the game to stop** any time by pressing **“ESC”** on the keyboard

Check /proj/src/drivers/keyboard for the driver’s code, the **KBC** can be found in /proj/src/drivers/kbc. The independent interrupt handling is the same as the above topic.

The event handler is on /proj/src/game/int_handlers/dependent/dependent_ih.c**,** in a function called **EVENTS handle_kbd_evt(EVENTS event)**.

To convert scancodes to ascii we use this table.

2.3 - Mouse

Mouse interrupts are used for

- Selecting **Menu and Game Options**, using mouse movement and pressing the left mouse button, changing the game state.
- Control the mouse image position that appears on the screen, moving the mouse around.
- Pick or grab a piece and select or leave the piece in a new position, **making a move**. More information on this on section 4.2.2.

The Driver is in /proj/src/drivers/mouse**,** independent interrupt handling is the same as the keyboard (more on this in section 4.8.1).

Function **EVENTS handle_mouse_evt(EVENTS event)** is responsible for the dependent IHs. The changing of states on the mouse, presented in that function, will be discussed in section 4.2.2**.**

2.4 - Video Card

We used mode ***0x14C***, with an ***1152x864*** resolution, and ***direct color*** mode with ***32 bits per pixel.*** We used Graphics in

- Drawing the background, menu options, and other **static animations**.
- Drawing **animated sprites** and **fonts**.
- Drawing **shapes** on the screen, especially rectangles/squares

More on the **Video Card** on /proj/src/drivers/graphics**.** Animations are defined in the /proj/src/game/views/animation module, important functions are **int draw_animSprite(...)**, **int(vg_draw_rectangle)(...)** , and **void draw_text(...)** , which you can find on /proj/src/game/views/font/font.c**.**

To render the animations we use a **double buffer**. Check the **flush_screen** function in the graphics driver module.

2.5 - RTC

The use of the RTC is linked to the **periodic interrupts** (once per second).
We use RTC interrupts for:
- Used for the chess game clocks as it will later be detailed (4.3).
- Displaying data and time on the screen

More on the **RTC** on /proj/src/drivers/rtc. The drawing of the clock is done in function **void draw_clock()**, in /proj/src/game/views/views.c , the player clock logic is done in the function **void draw_game_clock()** in the same file.

2.6 - Serial Port

The serial port is used for **duplex communication** by enabling the receiving and transmission of data interrupts (Check function ***set_ier*** on the independent module). It is configured for a Baud rate of 115200 b/s. We did not use a queue to send the bytes to the UART but rather another approach that will be detailed in section 4.4.

The interrupts are used for:

- Signal that data is available. Reading the data and decoding the message through the use of a protocol (4.5).
- Transmitter is empty. Data can be sent.

You can check the function ***handle_ser_evt*** presented in the /int_handlers/dependent module, where data is received and the flag (4.4) is set. The data transmission is presented sparsely in the code.

Moving a piece is done in the ***grab_state*** and in the ***pick_state*** of the mouse_st module. In the ***handle_kbd_evt*** you can check how we send a message and in the ***handle_mouse_evt*** you can check how we send the communication status. The UART driver is in the /proj/src/drivers/serial_port.

3. Code organization/structure

Our code is divided into 2 main modules: *game* and *devices*. The first module holds all game logic while the second one contains all the device drivers' code (basically the code developed in the labs with minor adaptations for the most part).

Also worth mentioning is a third folder where we store all game's XPM: *assets*.

In the **"*drivers*" folder** we have the following sub-modules for each implemented driver:

Device	Developed by	Relative Project Weight
Timer	*	2%
KBD	*	3%
KBC	*	6%
Mouse	*	4%
Graphics	*	8%
RTC	Marco (100%)	5%
Serial Port	Ricardo (100%)	8,5%

Table 2. Devices Weight in the project

***Important Note:**

We settled to use the solutions from one of the members and evidently compare the solutions of the remaining members to check if they were right.

As these labs were part of the continuous work on the curricular unit we don't take them into consideration when grading the individual project participation.

Every member's solution to the labs is in a branch with the corresponding member's name. Thus, to verify engagement with the curricular unit those branches can be checked.

The above modules are important for the **connection of the kernel with the peripherals**. The **KBC** is also a very important module as we will discuss in 4.8.1.

In the following subsections, we'll describe in greater detail the "*game*" directory.

3.1 - *Game Directory*

3.1.1 - *game.c*

- **Description:** This module sets up the views (menu + game), the board, and subscribes to the devices interrupts, handling events produced.
- **Relative Weight:** 2.5%
- **Developed by:** Ricardo (100%)

3.1.2 - *int_handlers/dependent/dependent_ih.c*

- **Description:** Module responsible for handling application dependent events.
- **Relative Weight:** 10%
- **Developed by:** Ricardo (70%) + Marco (30%)

3.1.3 - *int_handlers/independent/independent_ih.c*

- **Description:** Module responsible for handling application-independent interrupts. Calling the appropriate interrupt handlers.
- **Relative Weight:** 7%
- **Developed by:** Ricardo (80%) + Marco (20%)

3.1.4 - *protocol/communication_protocol.c*

- **Description:** This module defines our communication protocol for sending/receiving data via the serial port.
- **Relative Weight:** 6%
- **Developed by:** Ricardo (100%)

3.1.5 - *state_machine/menu_st.c*

- **Description:** This module implements the menu state machine and respective transitions.
- **Relative Weight:** 4%
- **Developed by:** Marco (70%) + Ricardo (30%)

3.1.6 - *state_machine/mouse_st.c*

- **Description:** This module implements the mouse state machine and respective transitions.
- **Relative Weight:** 2%
- **Developed by:** Ricardo (100%)

3.1.7 - *objects/pieces.c*

- **Description:** This module is the ”main brain” of the game. It is responsible for setting up the board with the pieces and computing the possible piece movements and executing those moves.
- **Relative Weight:** 8%
- **Developed by:** Marco (70%) + Ricardo (20%) + Pedro (10%)

3.1.8 - *views/views.c*

- **Description:** This module is responsible for every visual output of the program. That includes drawing the background, menus, mouse, game, pieces, and some game messages.
- **Relative Weight:** 10%
- **Developed by:** Marco (60%) + Ricardo (30%) + Pedro (8%) + Guilherme (2%)

3.1.9 - *views/animation/animation.c*

- **Description:** This module turns single XPM files into animated sprites with ease and efficiency.
- **Relative Weight:** 8%
- **Developed by:** Marco (100%)

3.1.10 - *views/font/font..c*

- **Description:** This module allows for writing ASCII text to the screen by mapping C strings (char*) into font XPMs that are then drawn to the screen.
- **Relative Weight:** 4%
- **Developed by:** Marco (100%)

3.1.11 - *views/sprite/sprite..c*

- **Description:** Module works as a Sprite Class with constructor and get/set methods for ease of creation and use of sprites throughout the code.
- **Relative Weight:** 2%
- **Developed by:** Ricardo (70%) + Marco (30%)

Note:
Also worth mentioning:

- Documentation was made by Marco, Pedro, and Ricardo
- XPMs were made/converted by Marco and Guilherme

3.2 - Main Functions Description

The **proj_main_loop** is responsible for setting up the video card and calling the **game_loop.** The **game_loop,** as the name suggests, is the loop of our program. It is responsible for allocating the sprites, subscribing the interrupts, and calling, in the loop, both the **independent interrupt handler** and **the dependent interrupt handler**. At the end of the loop, it is also responsible for unsubscribing the IHs and freeing the sprites allocation.

The **handle_ihs** function is responsible for handling the interrupts, making the connection between the devices and our program. This function will return an event so that our program knows which interrupt occurred.

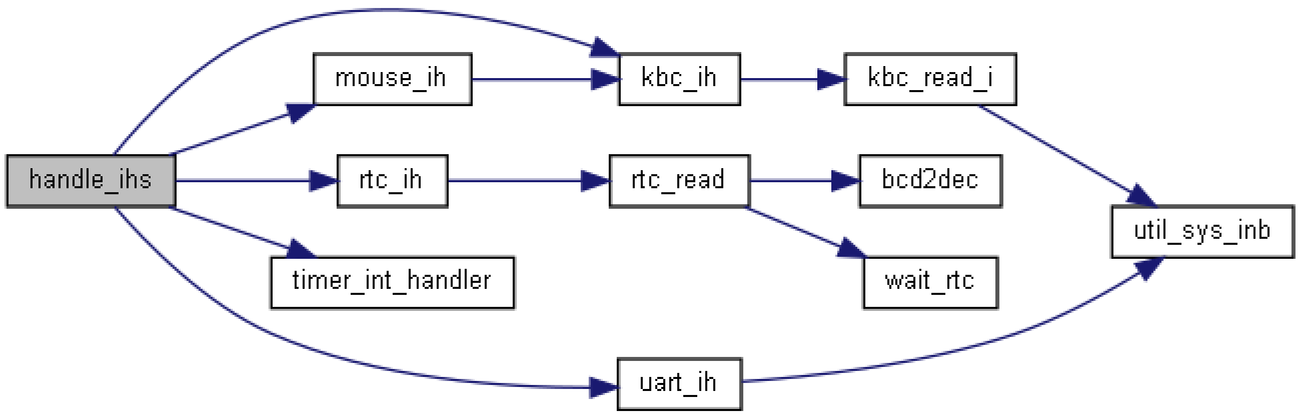
Then, the **handle_evt,** after receiving an event, will handle the events that occurred taking the necessary actions and implications of that events. This separation allows us to reuse the **handle_ihs** and make them independent of the program that uses them.

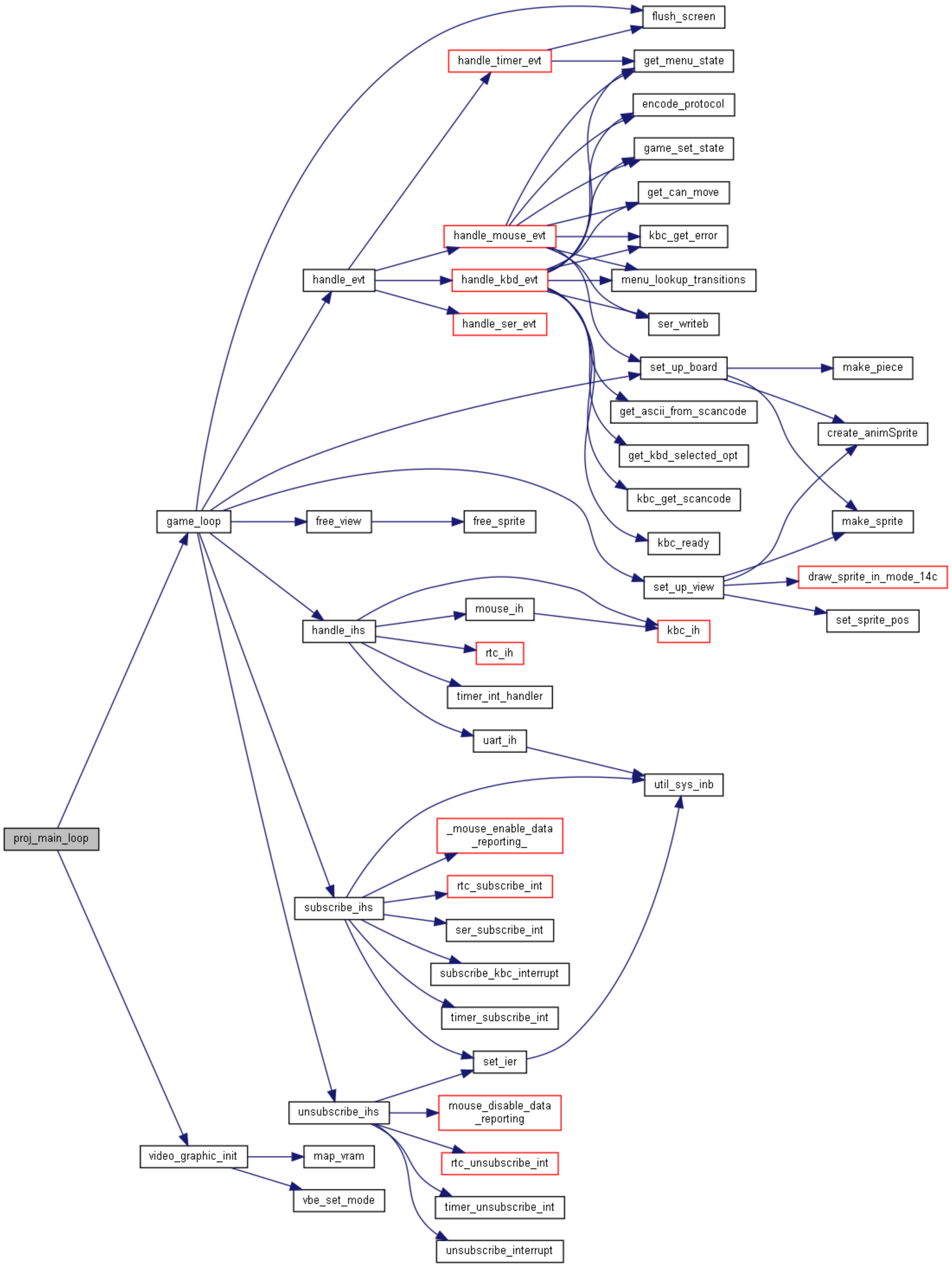
The **driver_recieve** is obviously presented in the **handle_ihs** function.

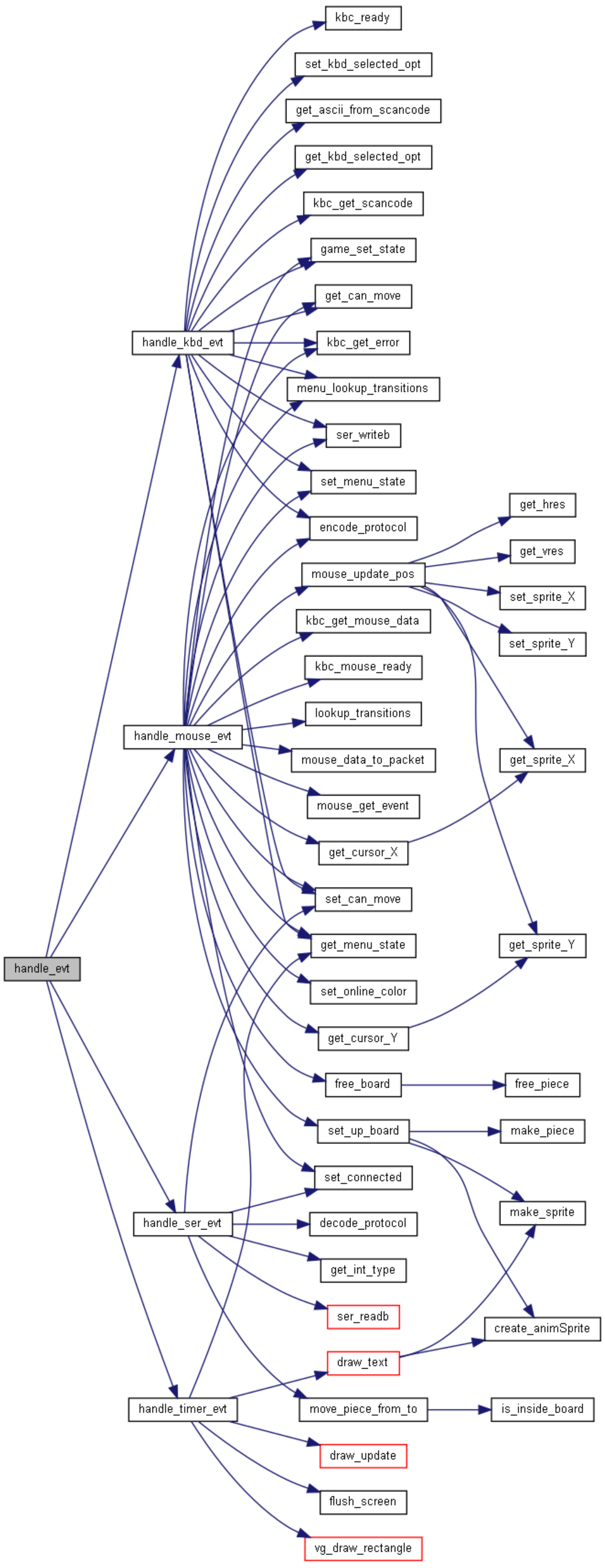
Interrupt handlers are in the **/proj/src/game/int_handlers** module. The **game_loop** is in the **/proj/src/game/game.c** file.

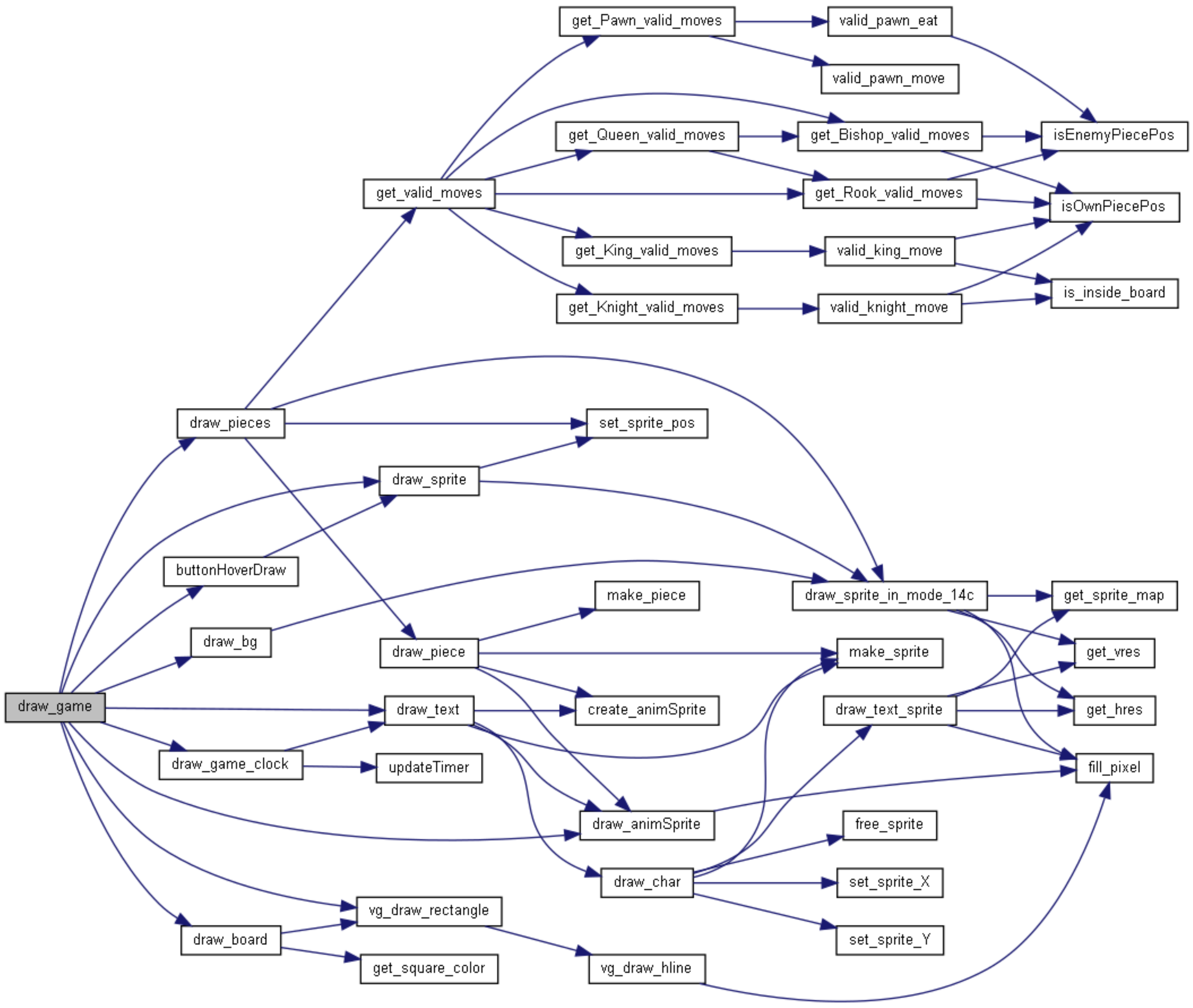
Another important function of the game is the **draw_game,** presented in the view.c file.

3.2 - Functions Call Graphs




proj_main_loop
game_loop
video_graphic_init
map_vram
vbe_set_mode
handle_evt
handle_timer_evt
handle_mouse_evt
handle_kbd_evt
handle_ser_evt
flush_screen
get_menu_state
encode_protocol
game_set_state
get_can_move
kbc_get_error
menu_lookup_transitions
ser_writeb
set_up_board
make_piece
create_animSprite
get_ascii_from_scancode
get_kbd_selected_opt
kbc_get_scancode
kbc_ready
make_sprite
free_view
free_sprite
set_up_view
draw_sprite_in_mode_14c
set_sprite_pos
handle_ihs
mouse_ih
kbc_ih
rtc_ih
timer_int_handler
uart_ih
util_sys_inb
subscribe_ihs
_mouse_enable_data_reporting_
rtc_subscribe_int
ser_subscribe_int
subscribe_kbc_interrupt
timer_subscribe_int
set_ier
unsubscribe_ihs
mouse_disable_data_reporting
rtc_unsubscribe_int
timer_unsubscribe_int
unsubscribe_interrupt


kbc_ready
set_kbd_selected_opt
get_ascii_from_scancode
get_kbd_selected_opt
kbc_get_scancode
game_set_state
get_can_move
handle_kbd_evt
kbc_get_error
menu_lookup_transitions
ser_writeb
set_menu_state
get_hres
encode_protocol
get_vres
mouse_update_pos
set_sprite_X
kbc_get_mouse_data
set_sprite_Y
kbc_mouse_ready
lookup_transitions
handle_mouse_evt
mouse_data_to_packet
get_sprite_X
mouse_get_event
get_cursor_X
set_can_move
handle_evt
get_menu_state
get_sprite_Y
set_online_color
get_cursor_Y
free_board
free_piece
set_up_board
make_piece
set_connected
make_sprite
handle_ser_evt
decode_protocol
get_int_type
ser_readb
create_animSprite
draw_text
handle_timer_evt
move_piece_from_to
is_inside_board
draw_update
flush_screen
vg_draw_rectangle


get_Pawn_valid_moves
valid_pawn_eat
valid_pawn_move
get_Queen_valid_moves
get_Bishop_valid_moves
isEnemyPiecePos
get_valid_moves
get_Rook_valid_moves
isOwnPiecePos
get_King_valid_moves
valid_king_move
get_Knight_valid_moves
valid_knight_move
is_inside_board
draw_pieces
set_sprite_pos
draw_sprite
buttonHoverDraw
draw_sprite_in_mode_14c
get_sprite_map
make_piece
draw_bg
draw_piece
make_sprite
get_vres
draw_game
draw_text
create_animSprite
draw_text_sprite
get_hres
draw_game_clock
updateTimer
fill_pixel
draw_animSprite
free_sprite
draw_char
set_sprite_X
vg_draw_rectangle
draw_board
set_sprite_Y
get_square_color
vg_draw_hline

4. - Implementation Details

4.1 - Efficient Sprite Animation

Animations can be very demanding in terms of memory because each frame of the image must be loaded and quickly drawn in a sequence. We found a simple yet powerful approach that, to our empirical experience, guarantees better memory efficiency without sacrificing performance.

Instead of storing each XPM frame as an individual variable and then loading them all up into memory and making an array of XPMs and cycling through them to create an animation, we opted to have a single XPM (and thus a single variable) to represent an entire animation like so:

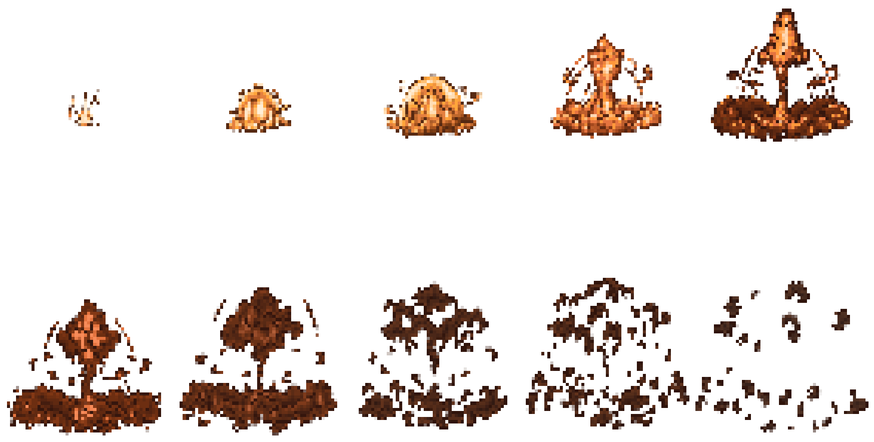


Image 9. Explosion Animation XPM

When we want to create the animation, we just have to move the image map pointer to the correct part of the XPM to draw the first frame, then move the pointer and draw the second frame, and so on.

The functions were developed to ensure compatibility with any animation, leaving room for expansion. Therefore, adding new animations becomes effortless and seamless with only two function calls needed (*create* and *draw*).

Check the implementation at game/views/animation/animation.c.

4.2 - State Machines using virtual tables

4.2.1 - Game State Machine

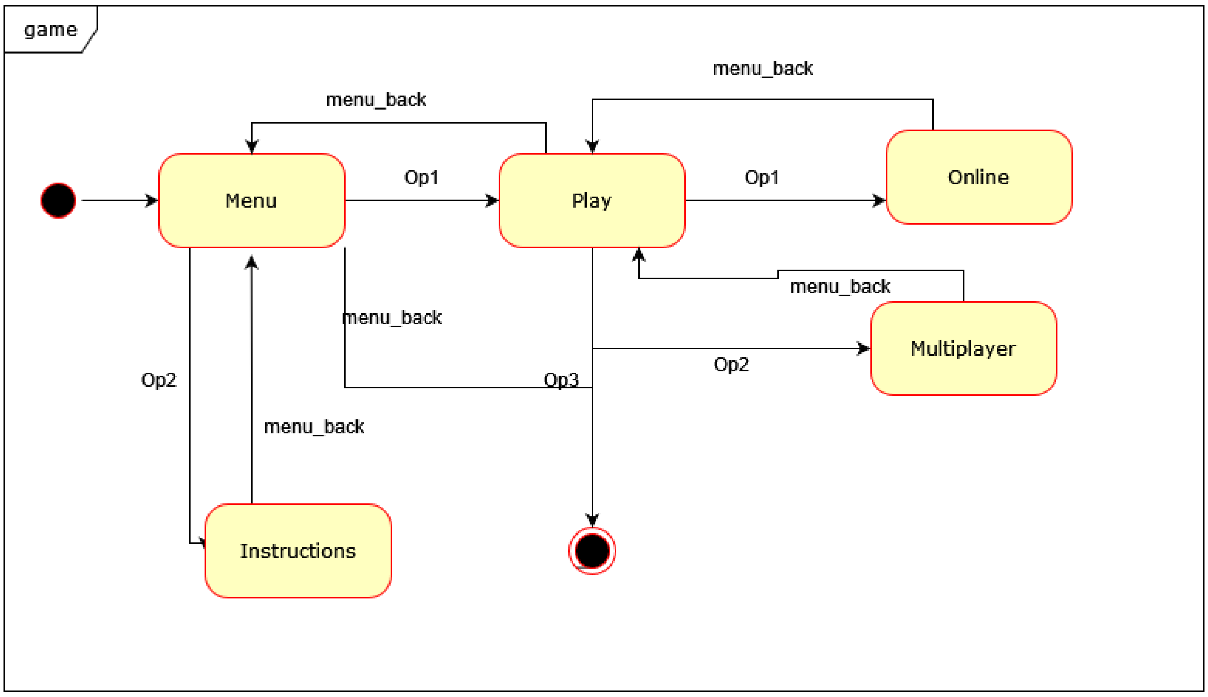


Image 10. Game State Machine

Events will trigger changes in the game state machine. Those changes can be a result of a **Mouse Event** or a **KeyBoard Event**. For example, pressing a button will trigger one event. Every state knows what **ret_code** to raise if a button was pressed, however, they don't know what state they should go to next.

That is defined in a **transition table** that will tell based on which state we are in and on the return code by that state, the state we will be doing next, including staying in the same state. To call the functions of the state we use arrays that **use pointers to functions** allowing us to know what state function we will call, this is what is called a **Virtual Table**.

You can check more implementation details on /proj/src/game/state_machine/menu_st.h**.**

The important functions to look at are **enum state_codes menu_lookup_transitions(...);** and the struct typedef struct **menu_transition**.

Finally, in the **menu_st.c** you can find the array with the function pointers
int (*menu_state[])(struct mouse_ev *event, int x, int y).

4.2.2 - Mouse State Machine

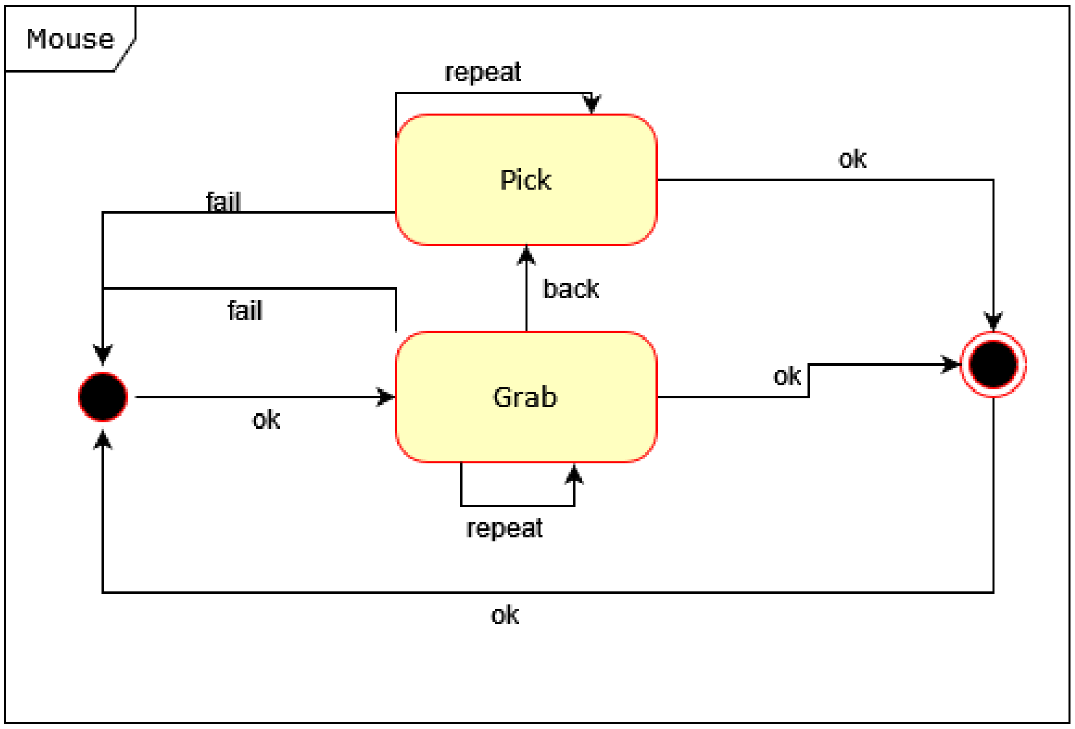


Image 11. Mouse State Machine

The Mouse state machine allows us to pick or grab a piece. When receiving a mouse event packet, a state will tell if we have grabbed a piece. If so, we have two options: either we want to continue to grab the piece or we have just selected them to make a two-pick move (picking a piece and the move place to move that piece).

This machine state has to inspect the state of the game as it needs to know if we are playing, online or multiplayer states. It also needs to call the game API to know if a move is correct or wrong and make it.

Additional implementation details can be checked in **/**proj/src/game/state_machine /mouse_st.h**.** Implementation is almost the same as the game state machine, with different return codes and states.

Both state machines were adapted from the following link: state machines tutorials.

4.2.3 - Online State Machine

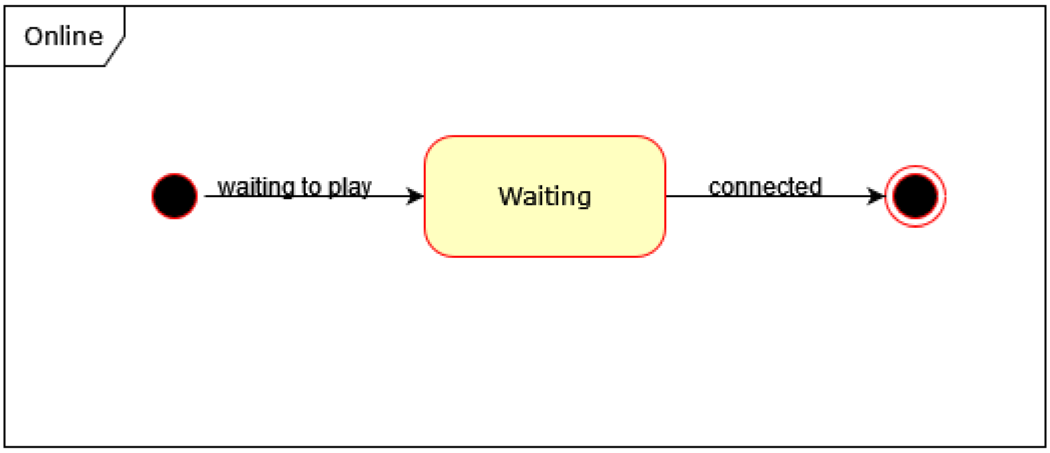


Image 12. Online State Machine

To know if a connection between two players can be established we develop yet another state machine. This one is simpler: a player will inform the other players that they want to play, then the other player will know that someone is waiting to connect.

When entering the online state both players will be playing. If a player tries to connect and nobody is there then he will see a message that no one can play. In chapter 4.5 we will talk about this in a more detailed way.

Further implementation details can be checked in /proj/src/game/int_handlers/dependent/dependent_ih.c .

4.3 - RTC

The RTC is set to give an interrupt every second. Thus, every second, the IH goes and reads the updated values of the date and hour from the RTC and stores them in an array, *rtc_data[6]*. Some precautions were made to assure a good implementation of this device driver. Namely, waiting for the RTC to finish a pending update and disabling & reenabling interrupts due to preemption.

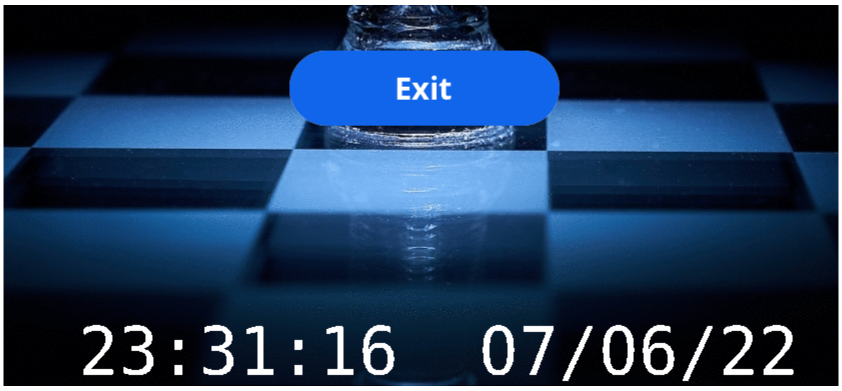


Image 13. RTC values in the start menu

A more applicable and elegant use of the RTC is for the chess game clocks. Each time the RTC raises an interrupt, a second passes, and hence, a second is decremented from the currently active player's clock. If a clock reaches zero, the game ends and the other player wins.

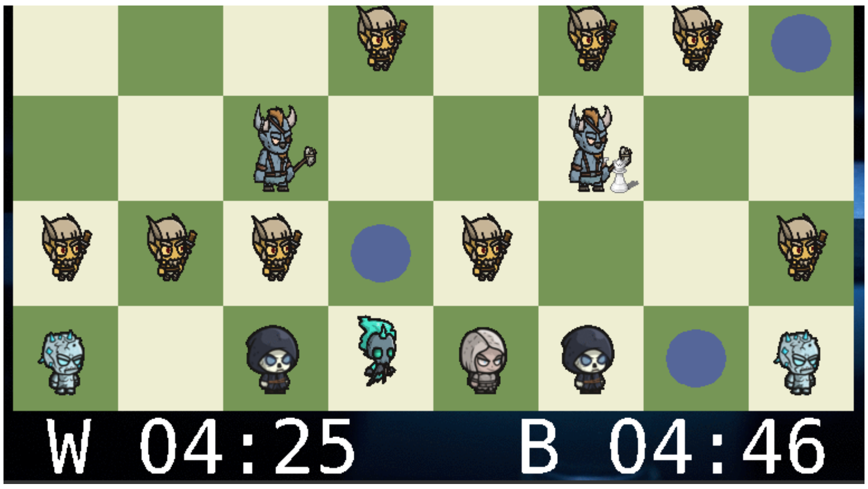


Image 14. Game clocks using RTC to work

4.4 - UART

As time was counted we decided not to implement the FIFO. However, we noted that something similar to the FIFO behavior was happening. If we send two bytes from one machine to the other (with only one **Transmitter Holding Register Empty interrupt**), with the other not being available, we noted that when the not available one was turned on it could read both the bytes.

Of course that this approach is not as independent as one might wish, in order to maintain synchronization we will have to know which information we are receiving and pass it not through an interface(queue) but directly to the device.

Another problem that arose is that some information was being lost, so we resolved that with a simple trick, check /proj/src/game/int_handlers/dependent/dependent_ih.c**.**

When receiving a T.H.R.E interrupt a flag will be raised that basically allows the dependent IH to know if it can send information. The first event that grabs that flag can send the information (others need to wait for another interrupt), this behavior is almost the same as a **queue**.

The difference is that we don't prevent ourselves from starvation, which is hard to occur, and we don't secure the **F.I.F.O** behavior, although it happens due to the rarity of the event (check ***handle_mouse_evt*** on /proj/src/game/int_handlers/dependent /dependent_ih.c to see what we are talking about).

4.5 - Communication Protocol

Move ID	**1**	**0/1**	**-**	**-**	**-**	**-**	**-**	**-**
Com. status	**0**	**1**	**-**	**-**	**-**	**-**	**-**	**-**
Message	**0**	**0**	**-**	**-**	**-**	**-**	**-**	**0/1**

In order to have a way to communicate between the two processes we had to develop a communication protocol.

To signal a **piece move** we used bit 7 to indicate that a move is pending and we used bit 6 to indicate if that move is a column move or a line move. Then in the next 6 bits, we pass the **original piece col/lin** and the **final col/lin**, reserving 3 bits for each. There was no way of optimizing this, as a move requires knowing the position of the piece and each takes 6 bits.

To signal a **communication change** we used bit 6 to indicate that. When the signal is read on the "other side", the communication status of the other machine is changed. This approach has a problem. It can happen that no one is listening, if a player is waiting and nobody is listening and then a player enters the game, the player that was waiting will have to go to the menu and reenter the online status. We could have done a **PIO** until someone was ready to play, however, due to time limitations we are happy with this solution.

The message is simpler to explain. Bit 0 will indicate if, after the character, that goes from bit 1-5, there are more characters. The Character passed corresponds to letters from A-Z.

There are more details on this. To further understand the mechanics of the communication protocol refer to /proj/src/game/protocol/communication_protocol.c**,** where you can see how we decoded the information and encoded it. The interrupt handling of this protocol can be found on /proj/src/game/int_handlers/dependent/dependent_ih.c **.**

4.6 - Menu Drawing

Our menus have a background and buttons. When hovering with the mouse or selecting buttons with the keyboard, a subtle glow effect is applied to the button in question. To avoid drawing multiple sprites in each frame, we have 3 different backgrounds with the buttons corresponding to each menu “built-in”. Thus, a single image is required instead of drawing a background + 3 or 4 buttons. When hovering a button, the correct sprite is drawn to cover the background, giving the illusion of a more interactive menu without much extra CPU being wasted on extra drawings every frame.

4.7- Debug

Debugging is a crucial part of writing a program in c. A very important part is to always check the return values of functions. It is also important to check if a pointer is not a null pointer and if values are in the range of what we expect. Doing this for every function call and for every parameter in which we need to check a range or the nullity of the program is tedious and takes precious time from the programmer.

Taking that in mind we decided very early in the unit course to make our own debugger. In **/proj/src/drivers/utils/handlers.h** you can find a useful set of tools used by us to somehow understand what is going on in the program.

Those tools are:

- **CHECKCall** - To be enabled you must define the **_DEBUG_MACRO_**, verifies if the return value is ok, and if not a message indicating **where(line)** in the file the error happened and the **file** in which it happened is printed and the program stops executing.
- **NullSafety** - Verifies if a pointer is not null. Making also a log message too.
- **Assert** - Verifies if expression is true, terminating the program if not.
- **Assert_cbet** - Verifies if value is in a closed interval. Make a log message if not.

These functions are great to understand what happened before the program stopped executing.

4.8 - Event-Driven Design and Layering

Our project follows an event-driven design. The **independent interrupt handlers** will signal that certain events have occurred, the independent handlers are responsible for the communication with devices, using the **memory-mapped I/0 addresses**. Then an event **handler,** which is basically our **program-dependent handlers,** will use the values retrieved by the independent interrupt handler for the purpose that it pursues. For example, when a Keyboard event is thrown then the keyboard event handler will check if the KBC got any error, if not then it checks if the KBC is ready to inform us about the packet received. Then by calling an appropriate function we get the scancode received by the interrupt handler, **kbc_get_scancode(scan, &scan_size)**, which makes our program more modular by **layering** the different modules and defining a connection between them, respecting the **SRP**.

4.8.1 - The KBC, how we design it

When we first developed the keyboard interrupt handler we were naive to think that we were only going to use it with the keyboard. However, when designing the mouse we noticed that we could actually make a single module for the KBC and then reuse it in both the kbd and mouse modules, and so did we. In this way, we needn't make a new interrupt handler and new issue commands function. In fact, we did a **mouse_ih()**, however, it is just a stub that calls the **kbc_ih()**.

For more information refer to /proj/src/drivers/kbc**,** /proj/src/drivers/mouse and /proj/src/drivers/keyboard**.**

5. Conclusion

During the development of this coding project, the most demanding part was the little time we had to complete all the envisioned features. We would like to make our chat more sophisticated, adding a scroll behavior and a history of messages, however, we had to contend with this one as we did not have time to explore this.

If time wasn't bound, our future plans would also include the addition of game mode selection, move/attack pieces animations, and other minor quality of life tweaks and performance improvements.

Furthermore, many features took greater effort to implement than previously expected. Namely, the setting up of all the devices, creation of the state machines, implementation of the event-driven design, and the serial port.

Going past the constraints, we strived to make a good, well-designed, structured coded game using all the available devices in a useful manner. Hence, in that sense, we consider our project complete and meeting the goals that were to us proposed as well as the goals we set for ourselves.

As a worthy mention, the implementation of the serial port and the well-thought communication protocol that allowed for online gaming was our biggest accomplishment.

We are also proud to have implemented all the device drivers that were presented in the unit course. We have to admit, however, that, regarding the last 2 devices, we sadly had not had the time to try everything described in the labs.

The LC classes covered most things we found needed to create any kind of program as a final project. However, we think that topics like collision detection, the serial port and RTC, event-driven design and state machines should have been given extra importance and been addressed earlier in the course as these are crucial points to the project.

Furthermore, we hoped that the time available, and the resources for this project were not so restricted as that had a great impact on the number of features that we could implement.

Finally, as the group members couldn't reach a satisfying agreement concerning the project results and participation, we opted to keep our final thoughts as part of the final individual self-evaluation form.