MINIX IoT

Computer Labs

Bachelors in Informatics and Computer Engineering

Nelson Neto 202108117
Paulo Fidalgo 201806603
Simão Neri 202206370
Wagner Pedrosa 201908556

2023/2024

Contents

1 User Instructions

This project aims to control Internet of Things (IoT) devices using MINIX. A user interface was built to facilitate user control of these devices.

The welcoming page is shown in Figure 1. To navigate to the next page, simply use the mouse to select the desired option.

Figure 1: Home Page

In the Control Shell page, the video mode is turned off and text mode is activated. Here, the user can type 'help' to see the available commands. Once a command is typed, pressing enter will execute it (see Figure 2).

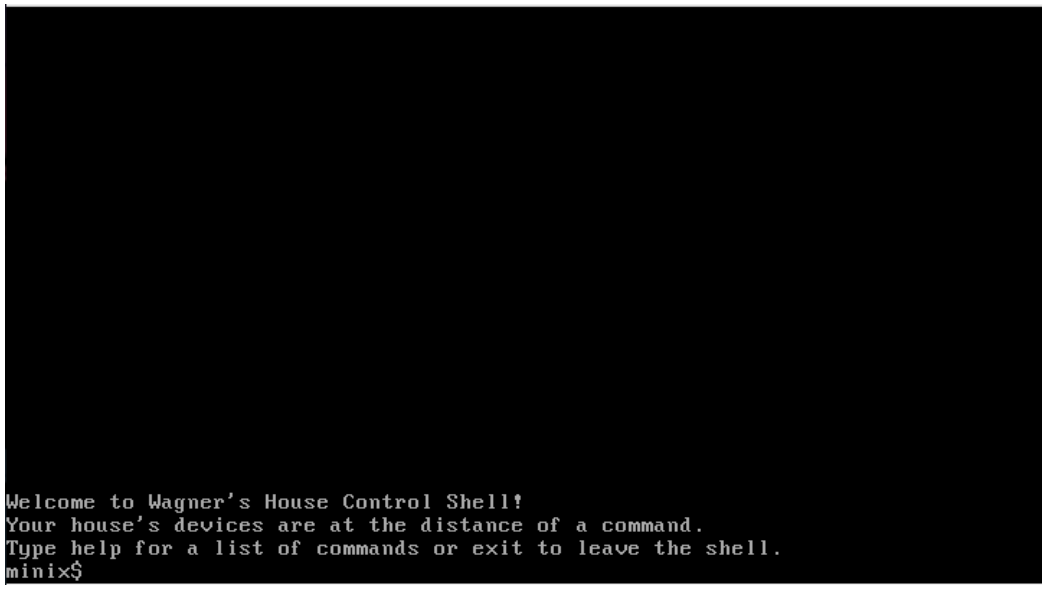


Figure 2: Control Shell Page

On the Resolution page, users can choose between two different screen resolutions: 800x600 and 1152x864 (see Figure 3).

Figure 3: Resolution Page

The Help page provides more information about how to use the application (see Figure 4).

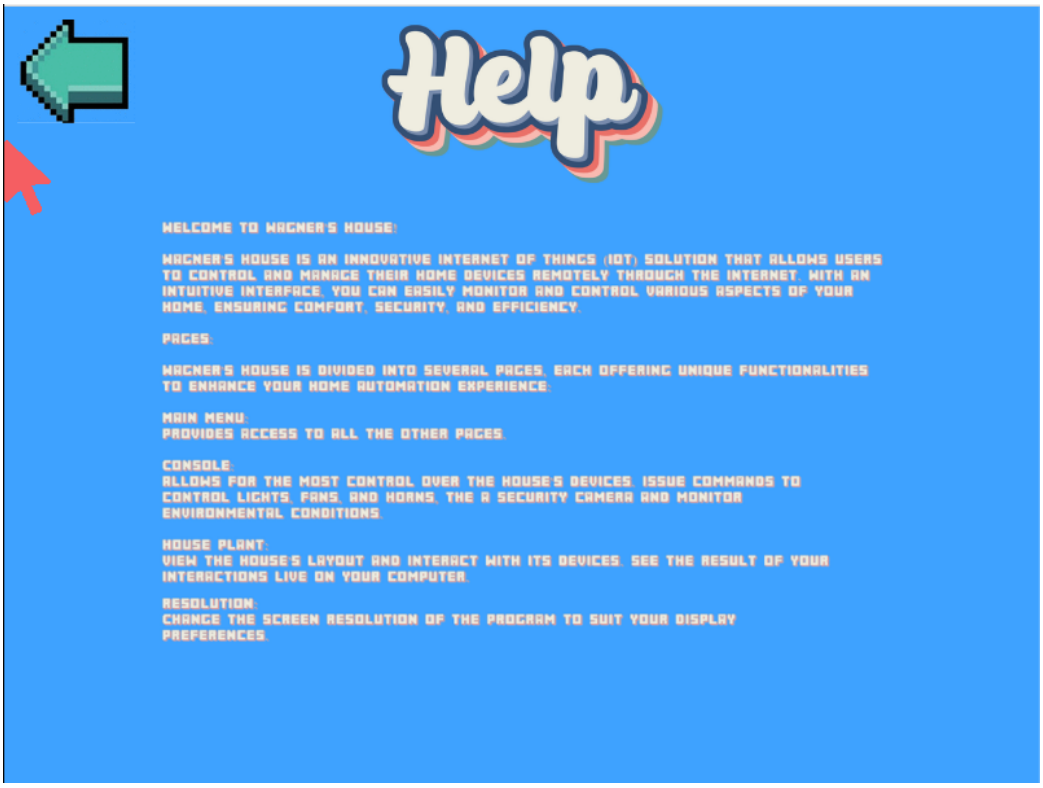


Figure 4: Help Page

In the House Plant page, users can control devices in a more intuitive manner using the presented buttons (see Figure 5).

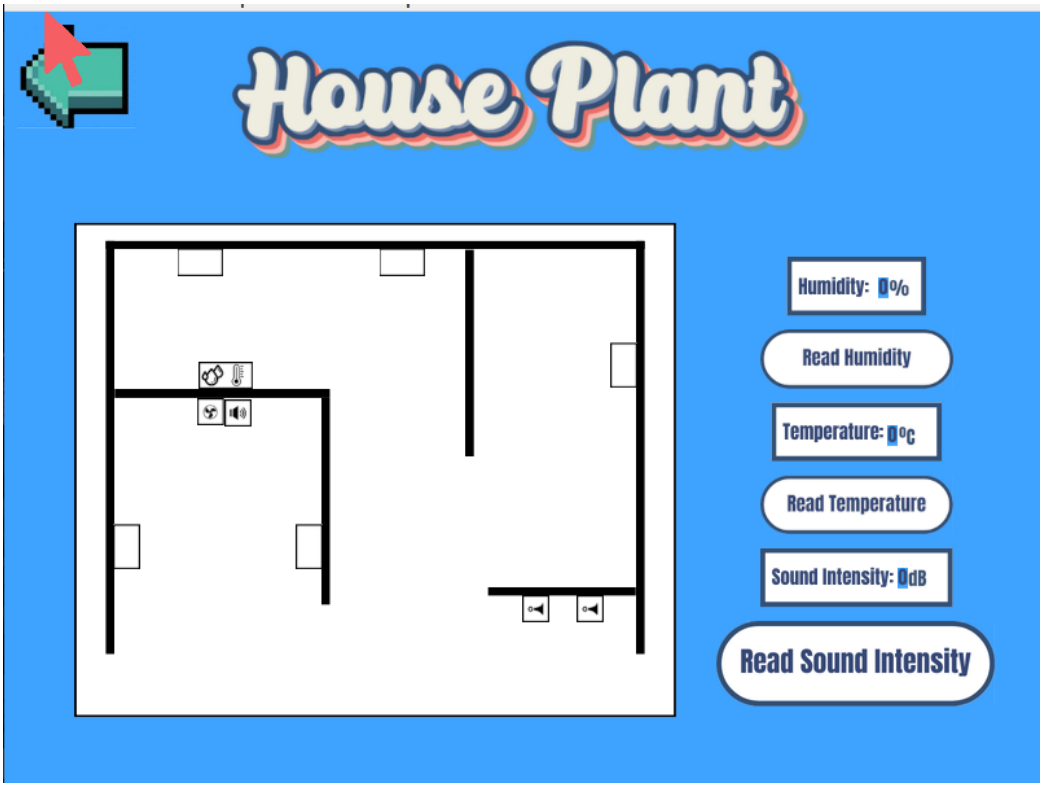


Figure 5: House Plant Page

When you are done, simply press the exit button to close the application.

2 Project Status

2.1 Functionalities

Functionality	**Devices**
Navigation between menus	Video Card, Mouse
Different Resolutions	Video Card
Mouse Animation	Mouse, Video Card
Control Shell	Keyboard, Serial Port, Timer
Turn on/off devices	Serial Port
Turn on/off devices for a certain duration	Serial Port, Timer
Blinking lights control	Serial Port, Timer
Request sensor information	Serial Port
Display current time and date	Real Time Clock, Video Card
Send message to display	Serial Port, Keyboard
Device status update	Serial Port, Video Card
Frame-rate control	Timer
Set up alarms	Real Time Clock, Video Card

Table 1: Functionality and Associated Devices

2.2 Devices Overview

Device	Functionalities
Timer	Used for frame rate control, blinking light periods, and managing the duration a device must remain on.
Mouse	Enables navigation through the application by selecting displayed buttons and making requests to the microcontroller via respective buttons on the plant.
Keyboard	Ensures the functionality of the control shell, allowing commands to be sent via text in a shell-like manner.
Video Card	Displays the user interface intuitively, highlighting "hovered" buttons, showing the current state of the house, and offering the app in two different sizes.
Serial Port	Establishes communication between MINIX and the host computer to send commands to the microcontroller and receive information from sensors and the camera.
Real Time Clock	Displays the current date and time, and allows configuration of alarms for specific hours.

Table 2: Devices Functionality

2.3 Devices

2.3.1 Timer

The timer in our application is a crucial component that manages various time-dependent functionalities. It is responsible for handling operations that require precise timing, such as turning lights ON for a specific duration (e.g., 10 seconds), controlling the blinking period of the lights, and maintaining the animation frame-rate at 30 frames per second. The timer ensures that these tasks are executed accurately and efficiently.

The implementation of the timer includes several key functions, which are defined in the `timer.c` file located in the `drivers/timer` directory.

The `i8254.h` file, also located in the `drivers/timer` directory, defines various constants and macros used by the timer. This header file includes definitions for the timer control words, timer modes, and other relevant parameters that are essential for configuring and operating the timer.

2.3.2 Keyboard

The application includes a built-in control shell, developed by the group, which allows users to send control signals in the form of text, similar to a

typical shell. The keyboard was essential for this functionality.

To implement the control shell, we needed to map every scancode to its corresponding ASCII character. This required meticulous handling of backspaces and arrow keys to ensure these functions were available and functional. We tracked the cursor position and managed the text display on the screen, especially when users edited text in the middle of a line.

Each key press generates a scancode, which we mapped to its corresponding ASCII character. This process was complicated by the need to handle special keys like backspaces and arrow keys.

When a backspace is pressed, the cursor position must shift left, and the displayed text must be updated to reflect the deletion. This involved managing the text buffer and ensuring the correct display of the remaining characters. Arrow keys allow users to navigate through the text. Handling these keys required maintaining the current cursor position and updating the display accordingly. When characters are added in the middle of a line, the text must shift to the right. This required dynamically managing the text buffer and ensuring the display accurately reflected the inserted character.

The implementation details are available in the `keyboard.c` file located in the `drivers/mouse_keyboard` folder. Since the keyboard and mouse both use the KBC (Keyboard Controller), the related functions are shared and defined in the same folder within the `KBC.c` file. Additionally, the `i8042.h` file in the same folder defines the relevant headers.

2.3.3 Mouse

To enable interaction with the graphical interface, the mouse functionality was implemented. This allows users to navigate through the application's pages by clicking the left mouse button on the displayed buttons. Since the mouse has a custom design and we needed to track its position to determine if it was over an element, it was necessary to maintain Cartesian coordinates. On each movement, the page is redrawn with the new mouse position. When the mouse hovers over a button, there is an animation or color change to indicate the interaction.

The implementation details are available in the `mouse.c` file located in the `drivers/mouse_keyboard` folder. Since the mouse and keyboard both use the KBC (Keyboard Controller), the related functions are shared and defined in the same folder within the `KBC.c` file. Additionally, the `i8042.h` file in the same folder defines the relevant headers and constants.

2.3.4 Video Card

To enhance user experience, the Video Card was implemented to support the application's graphical interface. The application offers two different resolutions: 800x600 and, 1152x864, both using direct color mode, which can be changed in the resolution page.

To ensure smooth and responsive mouse movement on the screen, we utilized the technique of triple buffering. Triple buffering involves maintaining three frame buffers: one being displayed, and the others being filled with the next frames. This approach allows for faster initialization and seamless buffer transitions, resulting in smoother animations and a more fluid user interface. In particular, the drawing of the mouse movement without this approach makes the screen blink.

The graphical interface of the application is built using sprites, which include the title of each page, the houseplant, buttons, and other visual elements. These sprites are in XPM format, as discussed in the lectures. By leveraging these graphical assets, we created an intuitive and visually appealing interface for users.

The implementation details for the Video Card, including the setup of resolutions, triple buffering, and sprite management, are contained in the files located in the `drivers/video` directory. The `VBE.h` file in this directory defines the necessary headers and constants for handling video operations.

2.3.5 Real Time Clock

The Real Time Clock (RTC) provides information about the current time of day and the day of the year, enabling the display of this information on the main menu and the setup of alarms for specific times and dates.

The implementation of the real-time clock is available in the file `real_time_clock.c` located in the `driver/real_time_clock` directory. The file `RTC.h`, also located in the same directory, defines the appropriate headers for the device.

2.3.6 Serial Port

The serial port serves as the critical communication channel that enables MINIX to connect to the internet, a primary necessity for interacting with the microcontroller via the MQTT broker. To facilitate this communication, the application running in MINIX sends commands through the serial port, which is emulated by VirtualBox. From the MINIX perspective, the implementation mirrors a standard serial port, as covered in lectures. However, due to the absence of a physical connection between the host and MINIX,

VirtualBox maps the serial communication to a socket or a named pipe. For this project, a TCP connection was chosen.

This setup implies that from the host side, messages sent by the emulated serial port are received as messages over the TCP protocol. Conversely, to send messages to MINIX (e.g., to retrieve temperature and humidity data from sensors), the same logic applies: the host writes to the socket, and on the MINIX side, the standard interrupt on the serial port IRQ line is triggered, prompting MINIX to handle the received byte.

The serial port thus becomes the main component of the project, bridging the communication between MINIX and the microcontroller. All commands from MINIX need to be delivered to the microcontroller, and responses from the microcontroller to MINIX are handled similarly through this serial-to-TCP interface.

The implementation of the serial port includes several key functions, which are defined in the `serial.c` file located in the `drivers/serial_port` directory. The file `serial_headers.h`, located in the same directory, contains the relevant definitions of the ports and masks to facilitate the readability of the code.

3 Code Organization/Structure

3.1 Files

3.1.1 Timer

3.1.1.1 Timer Implementation

Code Weight - 7.5%

This module was developed in Lab2. It includes functions to change the frequency of the timer, to set up the timer, and to handle timer interrupts. The corresponding header file, which outlines these functions, is depicted in Figure 6.

```
int (timer_set_frequency)(uint8_t timer, uint32_t freq);

int (timer_unsubscribe_int)();

int (timer_subscribe_int)(uint8_t *bit_no);

void (timer_int_handler)();

int (timer_get_conf)(uint8_t timer, uint8_t *st);

int (timer_display_conf)(uint8_t timer, uint8_t st,enum timer_status_field field);
```

Figure 6: Functions of Timer.h

Contributors:

- Paulo Fidalgo

3.1.1.2 Timer Handlers

Code Weight - 7.5%

This module defines all the handlers for the different states of the application. The behavior of the timer depends on the current state of the app. The file **timer_handler.h**, shown in Figure 7, illustrates the different handlers available.

```
void timer_main_menu_handler ();
void timer_control_shell_handler ();
void timer_house_plant_handler ();
void timer_settings_handler ();
void timer_help_handler ();
```

Figure 7: Functions of timer_handler.h

Contributors:

- Nelson Neto
- Paulo Fidalgo
- Simão Neri
- Wagner Pedrosa

3.1.2 Keyboard

3.1.2.1 Keyboard Implementation

Code Weight - 5%

This module was developed during the lab classes, in Lab 3. It configures the keyboard, subscribes to interrupts in exclusive mode so that MINIX does not "grab" the keyboard interrupts, handles the keyboard interrupts, and restores the previous configuration at the end. The corresponding header file, which details these functions, is shown in Figure 8.

```
int (kbc_subscribe_int)(uint8_t *bit_no);

int (kbc_unsubscribe_int)();

void (kbc_int_handler)();

int (kbc_restore)();
```

Figure 8: Functions of keyboard.h

Contributors:

- Paulo Fidalgo

3.1.2.2 Keyboard Handlers

Code Weight - 7.5%

This module defines all the handlers for the different states of the application. The behavior of the keyboard depends on the current state of the app. The file **keyboard_handler.h**, shown in Figure 9, illustrates the different handlers available.

```
void keyboard_main_menu_handler ();
void keyboard_control_shell_handler ();
void keyboard_house_plant_handler ();
void keyboard_settings_handler ();
void keyboard_help_handler ();
```

Figure 9: Functions of keyboard_handler.h

Contributors:

- Nelson Neto
- Paulo Fidalgo
- Simão Neri
- Wagner Pedrosa

3.1.3 Mouse

3.1.3.1 Mouse Implementation

Code Weight - 6%

This module was developed during the lab classes, in Lab 4. It configures the mouse by enabling data reporting, since MINIX by default has that disabled, subscribes and unsubscribes to interrupts, and parses the received bytes into a packet, defined as `mouse_packet`. The corresponding header file, which details these functions, is shown in Figure 10.

```c
struct mouse_packet;
typedef struct mouse_packet mouse_packet_t;

struct mouse_packet
{
    uint8_t right_button;
    uint8_t left_button;
    int16_t x;
    int16_t y;
};

mouse_packet_t mouse_packet;

int (mouse_subscribe_int)(uint8_t *bit_no);

int (mouse_unsubscribe_int)();

void (mouse_ih)();

void (parse_bytes_to_packet)();

int (write_mouse_cmd)(uint8_t cmd);
```

Figure 10: Functions of mouse.h

Contributors:

- Paulo Fidalgo

3.1.3.2 Mouse Handlers

Code Weight - 7.5%

This module defines all the handlers for the different states of the application. The behavior of the mouse depends on the current state of the app. The file **mouse_handler.h**, shown in Figure 11, illustrates the different handlers available.

```
void mouse_main_menu_handler ();
void mouse_control_shell_handler ();
void mouse_house_plant_handler ();
void mouse_settings_handler ();
void mouse_help_handler ();
```

Figure 11: Functions of mouse_handler.h

Contributors:

- Nelson Neto
- Paulo Fidalgo
- Simão Neri
- Wagner Pedrosa

3.1.4 KBC Module

Code Weight - 2%

This module defines helper functions for handling the mouse and keyboard, as both use the same controller. It was developed during the lab classes and includes functions to read status, read, and write KBC commands. The corresponding header file, which details these functions, is shown in Figure 12.

```
int (get_kbc_status)(uint8_t *status);

int (write_kbc_cmd)(uint8_t port, uint8_t cmd);

int (read_kbc_output)(uint8_t port, uint8_t *of, bool mouse);
```

Figure 12: Functions of KBC.h

Contributors:

- Paulo Fidalgo

3.1.5 Video Card

Code Weight - 10%

Part of this module was developed during the lab classes, specifically in Lab 5, which focused on the Video Card. This module configures the graphics mode, maps the video memory, and draws pixels on the screen by altering the content of the buffer. It also provides functions to swap buffers; for example, we use 2 buffers for one resolution and 3 buffers for larger resolutions. Additionally, it unmaps the frame buffer to change the resolution and utilizes a technique called page flipping to enable smooth transitions. The corresponding header file, which details these functions, is shown in Figure 13.

```
vbe_mode_info_t vmi_p;

int(vg_set_graphics_mode)(uint16_t mode);
int(map_frame_buffer_page_flipping)(uint16_t mode);
int(map_frame_buffer_triple_buffering)(uint16_t mode);
int(vg_draw_pixel)(uint16_t x, uint16_t y, uint32_t color);
int(clear_back_buffer)();
int(set_display_start_page_flipping)();
int(set_display_start_triple_buffering)();
int(swap_buffers)();
int(change_resolution)(int res);
int(unmap_frame_buffer)();
int(triple_copy)();
int(map_frame_buffer)(int res);
int(buffering_method)();
int(wait_display_start_change)();
```

Figure 13: Functions of graphics.h

Contributors:

- Paulo Fidalgo
- Simão Neri

3.1.6 Real Time Clock

3.1.6.1 Real Time Clock Implementation

Code Weight - 5%

This module was developed based on the work proposed in Lab6, which covered the real-time clock. It configures the device, subscribes and dispatches interruptions, sets up alarms, and provides the current date and

time. The corresponding header file, which details these functions, is shown in Figure 14.

```
struct real_time;
typedef struct real_time real_time_info;

int rtc_subscribe_int(uint8_t *bit_no);

int rtc_unsubscribe_int();

int rtc_get_config(uint8_t reg, uint8_t *config);

int rtc_set_config(uint8_t reg, uint8_t config);

int rtc_get_time();

int rtc_set_alarm(uint8_t hours, uint8_t minutes, uint8_t seconds);

int rtc_activate_interrupt(uint8_t interrupt);

int rtc_deactivate_interrupts();

void rtc_ih();
```

Figure 14: Functions of RTC.h

Contributors:

- Nelson Neto

3.1.6.2 Real Time Clock Handlers

Code Weight - 1%

This module defines all the handlers for the different states of the application. The behavior of the real time clock depends on the current state of the app. The file **real_time_clock_handler.h**, shown in Figure 15, illustrates the different handlers available.

```
void real_time_clock_main_menu_handler ();
void real_time_clock_control_shell_handler ();
void real_time_clock_house_plant_handler ();
void real_time_clock_settings_handler ();
void real_time_clock_help_handler ();
```

Figure 15: Functions of real_time_clock_handler.h

Contributors:

- Nelson Neto
- Paulo Fidalgo
- Simão Neri
- Wagner Pedrosa

3.1.7 Serial Port

3.1.7.1 Serial Implementation

Code Weight - 10%

This module was developed based on the work proposed in Lab7 (available at moodle), which covered the serial port device. It configures the serial port registers, initializes the port, handles received messages by enqueuing them into the queue, handles sending messages, manages interrupts, and clears interruptions. The corresponding header file, which details these functions, is shown in Figure 16.

```
int serial_initial_config();

int serial_port_subscribe_int(uint8_t *bit_no);

int serial_port_unsubscribe_int();

void serial_port_int_handler();

int send_serial_port_msg(uint8_t msg);

int read_serial_port_msg();

int serial_port_clear_int();

void serial_port_clear_all();

queue_t *get_queue();
```

Figure 16: Functions of Serial.h

Contributors:

- Paulo Fidalgo

3.1.7.2 Serial Handlers

Code Weight - 10%

This module defines all the handlers for the different states of the application. The behavior of the serial port depends on the current state of the

app. The file **serial_port_handler.h**, shown in Figure 17, illustrates the different handlers available.

```
void serial_port_main_menu_handler ();
void serial_port_control_shell_handler ();
void serial_port_house_plant_handler ();
void serial_port_settings_handler ();
void serial_port_help_handler ();
```

Figure 17: Functions of serial_port_handler.h

Contributors:

- Nelson Neto
- Paulo Fidalgo
- Simão Neri
- Wagner Pedrosa

3.1.8 Queue

Code Weight - 2%

This module was developed as an auxiliary module to handle the incoming messages from the Serial Port. It includes functions to create and destroy the queue, push and pop elements, and resize if needed. The corresponding header file is shown in Figure 18.

```
struct queue;
typedef struct queue queue_t;

queue_t *new_queue();

void delete_queue(queue_t *q);

int push(queue_t *q, int n);

int pop(queue_t *q, int* n);

void clear_queue(queue_t *q);

bool is_empty(queue_t *q);
```

Figure 18: Functions of queue.h

Contributors:

- Paulo Fidalgo

3.1.9 Main Module

Code Weight - 5%

This module was designed to manage the initialization, execution, and termination of the project that involves handling various hardware interrupts and drawing graphical pages. The corresponding header file, which details these functions, is shown in Figure 19.

```
struct handler;
typedef struct handler handler_t;

int (project_start)();
int (project_loop)();
int (project_stop)();
```

Figure 19: Functions of project.h

Contributors:

- Paulo Fidalgo

- Wagner Pedrosa
- Nelson Neto
- Simão Neri

3.1.10 Arduino Module

Code Weight - 5%

This module deals with the code running on the microcontrollers. It sets up the Wi-Fi and Mosquitto connection, defines the handlers for each topic, and dispatches the received messages.

Contributors:

- Paulo Fidalgo
- Wagner Pedrosa

3.1.11 Command Line Module

Code Weight - 2%

This module was developed as an auxiliary module to handle input commands to the shell. It allows inputting a command into the shell, translating a scancode, and dispatching a command to the corresponding handler. The corresponding header file, which details these functions, is shown in Figure 20.

```
void input_to_command_line(uint8_t scancode);
void translate_scancode(uint8_t scancode);
void handle_command(char* line_buffer);
```

Figure 20: Functions of command_line.h

Contributors:

- Nelson Neto

3.1.12 Light Module

Code Weight - 2%

This module was developed as an auxiliary module to handle the control and management of lighting systems. It provides functions to turn lights on and off, set timers, and enable blinking modes. The corresponding header file, which details these functions, is shown in Figure 21.

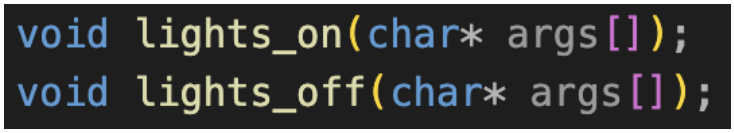


Figure 21: Functions of lights.h

Contributors:

- Nelson Neto

3.1.13 Draw Module

Code Weight - 2%

This module was developed as an auxiliary module to draw the pages. It includes functions to render various pages and elements on the screen based on the current state of the application. It handles the drawing of the main menu, security camera, settings, and placeholder pages for additional features. The corresponding header file, which details these functions, is shown in Figure 22.

```
int draw_page();

int draw_main_menu();

int draw_control_shell();

int draw_display_message();

int draw_security_camera();

int draw_settings();

int draw_house_plant();

int draw_help();

int draw_main_buttons();

int draw_settings_buttons();
```

Figure 22: Functions of draw.h

Contributors:

- Simão Neri

3.1.14 Sprite Module

Code Weight - 2%

This module was developed as an auxiliary module to manage and render sprites within the application. It includes functions for loading, updating, and drawing sprites on the screen, ensuring smooth animations and interactions. The module handles various sprite-related tasks such as collision detection with mouse and state management. The corresponding header file, which details these functions, is shown in Figure 23.

```
//Creates a new sprite drom XPM "pic" in the specified position
Sprite* (create_sprite)(xpm_map_t pic, int x, int y);

//Destroys a sprite
int (destroy_sprite)(Sprite *sp);

//Destroys all sprites used
int (destroy_all_sprites)();

//Draws a specific sprite
int (draw_sprite)(Sprite *sp);

//Loads the sprites
int (load_sprites)(int res);

int (load_sprites_1152x864)();
int (load_sprites_800x600)();
```

Figure 23: Functions of sprite.h

Contributors:

- Simão Neri

3.2 Function Calls

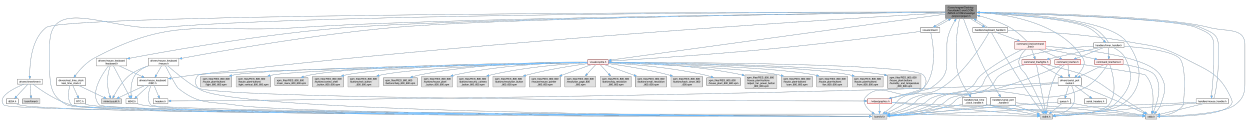


Figure 24: Function Calls Graph

4 Implementation Details

4.1 Topics Covered in Lectures

4.1.1 State Machine

The core logic of our application is driven by a state machine, which allows us to manage the behavior of the application based on its current state. Each page of the application corresponds to at least one state, and each state has specific handlers for different devices such as the timer, keyboard, mouse, serial port, and real-time clock. By utilizing a state machine, we can dynamically change the behavior of these devices according to the current state of the program.

4.1.1.1 State Handlers

For each device, we define an array of structs, where each struct contains the handlers for the corresponding devices for each state. These handlers are invoked upon receiving any interrupt on the respective IRQ line. This modular approach ensures that each device operates correctly based on the current state of the application.

```
static const handler_t timer_handler[] = {
    {timer_main_menu_handler},
    {timer_control_shell_handler},
    {timer_house_plant_handler},
    {timer_settings_handler},
    {timer_help_handler}
};
```

(a) Timer Handlers

```
static const handler_t keyboard_handler[] = {
    {keyboard_main_menu_handler},
    {keyboard_control_shell_handler},
    {keyboard_house_plant_handler},
    {keyboard_settings_handler},
    {keyboard_help_handler}
};
```

(b) Keyboard Handlers

```
static const handler_t mouse_handler[] = {
    {mouse_main_menu_handler},
    {mouse_control_shell_handler},
    {mouse_house_plant_handler},
    {mouse_settings_handler},
    {mouse_help_handler}
};
```

(c) Mouse Handlers

```
static const handler_t serial_port_handler[] = {
    {serial_port_main_menu_handler},
    {serial_port_house_plant_handler},
    {serial_port_settings_handler},
    {serial_port_help_handler}
};
```

(d) Serial Port Handlers

```
static const handler_t real_time_clock_handler[] = {
    {real_time_clock_main_menu_handler},
    {real_time_clock_control_shell_handler},
    {real_time_clock_house_plant_handler},
    {real_time_clock_settings_handler},
    {real_time_clock_help_handler}
};
```

(e) Real-Time Clock Handlers

Figure 25: Arrays of structs with handlers for each state

4.1.1.2 State Management

The state of the application is represented as an integer, starting from 0. This integer serves as an index into the array of structs, mapping to the appropriate handler for each device. When an interrupt is received, the dispatcher simply invokes the current handler based on the application's state.

```
while (running) {
  if( (r =  driver_receive(ANY, &msg, &ipc_status)) != 0 ) {
    printf("Error %s\r", r);
    continue;
  }

  if(is_ipc_notify(ipc_status)) {
    switch(_ENDPOINT_P(msg.m_source)){
      case HARDWARE:
        if (msg.m_notify.interrupts & irq_timer) {

          timer_handler[page_state].handler();
        }

        if (msg.m_notify.interrupts & irq_keyboard) {
          keyboard_handler[page_state].handler();
        }

        if (msg.m_notify.interrupts & irq_mouse) {
          mouse_handler[page_state].handler();
        }

        if (msg.m_notify.interrupts & irq_real_time_clock) {
          real_time_clock_handler[page_state].handler();
        }

        if (msg.m_notify.interrupts & irq_serial_port) {
          serial_port_handler[page_state].handler(serial_port_handler);
        }
```

Figure 26: State Machine Dispatcher

4.1.1.3 Event-Driven Behavior

By using a state machine, our application follows an event-driven paradigm. This means that the internal flow of the application depends on external inputs, such as user interactions or device interrupts. The state machine allows for a clear and organized way to handle these events, ensuring that the application responds correctly to each input.

4.1.1.4 Advantages of Using a State Machine

Implementing a state machine provides several advantages:

- **Modularity:** Each state is handled independently, making the code easier to manage and extend.
- **Clarity:** The flow of the application is clear and well-defined, with specific handlers for each state.
- **Scalability:** New states and handlers can be added without disrupting the existing code.

- **Maintainability:** Changes to the behavior of a specific state can be made without affecting other parts of the application.

By leveraging the state machine architecture, our application maintains a robust and flexible structure, capable of handling complex interactions and ensuring a smooth user experience.

4.1.2 Event-Driven Code

The paradigm of event-driven programming centers around the concept that the flow of the program is determined by external inputs or events. This approach contrasts with traditional sequential programming, where the flow is predetermined by the sequence of instructions.

In the context of our project, event-driven programming plays a pivotal role. Both the behavior of the IoT devices and the main application depend entirely on external inputs provided by the end user. The application itself is not autonomous; it requires user interactions to control and manage the devices. This design ensures that the system remains responsive and adaptable to real-time user commands.

4.1.2.1 Event Handling in Our Project

Our implementation uses event-driven code to handle various types of inputs and interactions:

- **User Commands:** The primary events in our system are user commands received through the interface. These commands are processed and translated into actions that control the IoT devices. For instance, turning lights on or off, adjusting the thermostat, or initiating a camera feed.

- **Sensor Data:** Events also include data received from sensors connected to the microcontroller. This data is processed and can trigger further actions, such as sending alerts or updating the display with real-time information.

- **Network Communication:** The application also handles network-based events, such as incoming TCP requests from the MINIX system or MQTT messages from the Mosquitto broker. These events are crucial for maintaining synchronization between the host system and the microcontroller.

4.1.2.2 Advantages of Event-Driven Code

The event-driven approach offers several advantages for our project:

- **Responsiveness:** The system remains highly responsive to user actions and real-time data changes, providing immediate feedback and control.
- **Scalability:** New devices and features can be integrated seamlessly, as the event-driven model allows for easy addition of new event handlers without disrupting the existing code.
- **Modularity:** Each component (user interface, sensors, network communication) operates independently and interacts through well-defined events, enhancing the modularity and maintainability of the code.

In our implementation, the core event loop listens for events and dispatches them to the appropriate handlers. This design is evident in the handling of TCP requests, as described in the previous sections. The server thread, for instance, listens for incoming TCP connections (an event) and processes each connection by reading the message, interpreting it, and sending the corresponding command to the microcontroller.

4.1.2.3 Example: User Interaction Event

When a user interacts with the interface, such as clicking a button to turn on a light, the following sequence occurs:

1. **Event Generation:** The button click generates an event, which is captured by the event loop.
2. **Event Handling:** The event loop dispatches this event to the corresponding handler function.
3. **Command Dispatch:** The handler function translates the event into a command that is sent to the microcontroller via the serial port.
4. **Device Action:** The microcontroller receives the command and activates the light.
5. **Feedback Loop:** If necessary, the microcontroller sends a status update back to the host, which is processed and displayed to the user.

This process illustrates how event-driven programming allows the system to react dynamically to user inputs and other external events, ensuring efficient and effective control over the IoT devices.

4.1.3 Real Time Clock

The implementation of the Real Time Clock (RTC) module was developed to handle various time-related functionalities such as retrieving the current date and time, setting up alarms, and managing RTC interrupts.

RTC Configuration Functions: The functions used to read from and write to the RTC registers facilitate the configuration of the RTC settings.

Interrupt Handling: Functions enable and disable RTC interrupts by modifying the configuration register `RTC_REG_B`. The interrupt handler function checks and handles alarm, update, and periodic interrupts.

Binary and BCD Conversion: The RTC can operate in either binary or BCD (Binary-Coded Decimal) mode. Helper functions are used for conversions between binary and BCD formats.

Setting Alarms: This functionality sets an alarm time by configuring the corresponding alarm registers. It handles both binary and BCD formats based on the RTC's current configuration.

Reading Time: The function reads the current time from the RTC registers. It ensures that the data is consistent by checking the `RTC_UIP` flag and then reads the values for seconds, minutes, hours, day, month, and year. The values are converted from BCD to binary if necessary.

Interrupt Subscription: Functions manage the subscription and unsubscription of RTC interrupts. These functions are essential for enabling the RTC to trigger interrupts for time-related events.

This implementation allows the system to manage time accurately and perform scheduled tasks such as setting alarms and handling periodic updates, which are critical for time-sensitive applications.

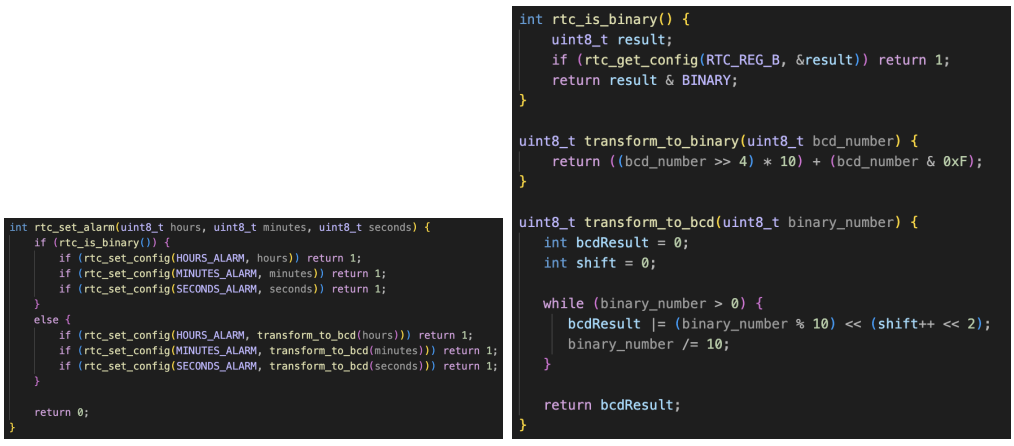
To better understand the RTC module's functionality, we include the following figures:

```c
int rtc_get_config(uint8_t reg, uint8_t *config) {
    if (sys_outb(RTC_ADDR_REG, reg)) return 1;
    if (util_sys_inb(RTC_DATA_REG, config)) return 1;
    return 0;
}

int rtc_set_config(uint8_t reg, uint8_t config) {
    if (sys_outb(RTC_ADDR_REG, reg)) return 1;
    if (sys_outb(RTC_DATA_REG, config)) return 1;
    return 0;
}

int rtc_activate_interrupt(uint8_t interrupt) {
    uint8_t config;
    if (rtc_get_config(RTC_REG_B, &config)) return 1;
    config |= interrupt;
    if (rtc_set_config(RTC_REG_B, config)) return 1;
    return 0;
}
```

Figure 27: RTC Configuration Registers



```c
int rtc_set_alarm(uint8_t hours, uint8_t minutes, uint8_t seconds) {
    if (rtc_is_binary()) {
        if (rtc_set_config(HOURS_ALARM, hours)) return 1;
        if (rtc_set_config(MINUTES_ALARM, minutes)) return 1;
        if (rtc_set_config(SECONDS_ALARM, seconds)) return 1;
    }
    else {
        if (rtc_set_config(HOURS_ALARM, transform_to_bcd(hours))) return 1;
        if (rtc_set_config(MINUTES_ALARM, transform_to_bcd(minutes))) return 1;
        if (rtc_set_config(SECONDS_ALARM, transform_to_bcd(seconds))) return 1;
    }

    return 0;
}
```

```c
int rtc_is_binary() {
    uint8_t result;
    if (rtc_get_config(RTC_REG_B, &result)) return 1;
    return result & BINARY;
}

uint8_t transform_to_binary(uint8_t bcd_number) {
    return ((bcd_number >> 4) * 10) + (bcd_number & 0xF);
}

uint8_t transform_to_bcd(uint8_t binary_number) {
    int bcdResult = 0;
    int shift = 0;

    while (binary_number > 0) {
        bcdResult |= (binary_number % 10) << (shift++ << 2);
        binary_number /= 10;
    }

    return bcdResult;
}
```

Figure 28: (Left) RTC Alarm Setup, (Right) RTC Time Retrieval

These images help illustrate how the RTC module is configured and used

```c
int rtc_get_time() {

    uint8_t out;

    do {
        if (rtc_get_config(RTC_REG_A, &out)) return 1;
    } while (out & RTC_UIP);

    if (rtc_is_binary()) {
        if (rtc_get_config(SECONDS, &out)) return 1;
        time_info.seconds = out;

        if (rtc_get_config(MINUTES, &out)) return 1;
        time_info.minutes = out;

        if (rtc_get_config(HOURS, &out)) return 1;
        time_info.hours = out;


        if (rtc_get_config(DAY_OF_THE_MONTH, &out)) return 1;
        time_info.day = out;


        if (rtc_get_config(MONTH, &out)) return 1;
        time_info.month = out;


        if (rtc_get_config(YEAR, &out)) return 1;
        time_info.year = out;
    }
    else {
        if (rtc_get_config(SECONDS, &out)) return 1;
        time_info.seconds = transform_to_binary(out);

        if (rtc_get_config(MINUTES, &out)) return 1;
        time_info.minutes = transform_to_binary(out);

        if (rtc_get_config(HOURS, &out)) return 1;
        time_info.hours = transform_to_binary(out);


        if (rtc_get_config(DAY_OF_THE_MONTH, &out)) return 1;
        time_info.day = transform_to_binary(out);


        if (rtc_get_config(MONTH, &out)) return 1;
        time_info.month = transform_to_binary(out);


        if (rtc_get_config(YEAR, &out)) return 1;
        time_info.year = transform_to_binary(out);
    }

    return 0;
}
```

Figure 29: RTC get time function

to handle alarms and retrieve the current time.

4.1.4 Double Buffering

To improve the smoothness and performance of our animations and visual updates on the screen, we implemented double buffering.

Double buffering involves the use of two buffers: a front buffer and a back buffer. The front buffer is the one currently being displayed on the screen, while the back buffer is used for drawing the next frame. Once the drawing is complete, the buffers are swapped.

In our code, the "`map_frame_buffer_page_flipping`" function sets up the front and back buffers, while "`set_display_start_page_flipping`" updates the display to start showing the new frame. To switch between the front and back buffers, we use the "`swap_buffers`" function.

When using double buffering, sometimes it may not completely eliminate visual artifacts. To solve this, we can use vertical retrace, the interval between rendering the rightmost pixel of the bottom line of one frame, and the rendering of the leftmost pixel of the top line of the next frame. In our case, we change the `r.bh` and `r.bl` to 0x80.

```c
int (set_display_start_page_flipping)() {
  reg86_t r;
  memset(&r, 0, sizeof(r));

  r.intno = BIOS_VIDEOCARD_SERV;


  r.ax = 0x4F07;

  r.bh = 0x00;
  r.bl = 0x80;

  r.cx = 0x00;
  r.dx = buffer ? 0x00: vmi_p.YResolution;


  if (sys_int86(&r)) {
    printf("vg_set_graphics_mode: sys_int86() failed\n");
    return 1;
  }

  buffer = (buffer == 0 ? 1 : 0);

  return 0;
}
```

(a) Set page flipping

```c
int(map_frame_buffer_page_flipping)(uint16_t mode) {
  memset(&vmi_p, 0, sizeof(vmi_p));

  int attempts = 10;
  while (attempts > 0 && vbe_get_mode_info(mode, &vmi_p)) {
    attempts--;
  }
  if (attempts < 0)
    return 1;

  struct minix_mem_range mr;
  unsigned int vram_base = vmi_p.PhysBasePtr;
  vram_size = vmi_p.XResolution * vmi_p.YResolution * ((vmi_p.BitsPerPixel + 7) / 8);
  mr.mr_base = vram_base;
  mr.mr_limit = mr.mr_base + vram_size * 2;

  if (sys_privctl(SELF, SYS_PRIV_ADD_MEM, &mr)) {
    panic("sys_privctl (ADD_MEM) failed\n");
    return 1;
  }

  video_mem = vm_map_phys(SELF, (void *) mr.mr_base, vram_size * 2);
  if(video_mem == NULL) {
    panic("couldn't map video memory");
    return 1;
  }
  front_buffer = video_mem;
  back_buffer = video_mem + vram_size;

  printf("Video Mem Address: %p, Size: %u bytes\n", (void*)video_mem, vram_size);
  printf("Front Buffer Address: %p, Size: %u bytes\n", (void*)front_buffer, vram_size);
  printf("Back Buffer Address: %p, Size: %u bytes\n", (void*)back_buffer, vram_size);
  printf("Extra Buffer Address: %p, Size: %u bytes\n", (void*)extra_buffer, vram_size);

  return 0;
}
```

(b) Map page flipping

```c
int (swap_buffers)() {
  if(set_display_start_page_flipping() != 0) {
    printf("Error: Problems occured while trying to set display start! \n");
    return 1;
  }

  uint8_t *temp = front_buffer;
  front_buffer = back_buffer;
  back_buffer = temp;

  if(clear_back_buffer() != 0) {
    printf("Error: Problems occured while trying to clean back buffer! \n");
    return 1;
  }

  return 0;
}
```

(c) Swap buffers

Figure 30: Double buffering

4.1.5 Page Flipping

While double buffering typically involves two buffers and may involve copying the frame from the back buffer to the front buffer, page flipping swaps the front and back buffers by changing the display pointer, making the process more efficient. As mentioned in the double buffering section, we implemented double buffering with page flipping to ensure smooth performance of our animations and visual updates on the screen.

4.1.6 Triple Buffering

Triple buffering extends the concept of double buffering by adding an extra buffer, known as the extra buffer. This technique provides an additional buffer to work with, which can further reduce latency and improve performance. With triple buffering, while one buffer is being displayed, another

buffer is being prepared for the next frame, and the third buffer is available for immediate use if needed.

In our code, the '`wait_display_start_change`' function waits until the last scheduled change occurs, while '`set_display_start_triple_buffering`' updates the display to start showing the new frame, taking into account the additional buffer. The '`triple_copy`' function handles the rotation of the buffers and clears the new back buffer for the next frame.

4.1.7 Object-Oriented C

The use of object-oriented principles in C significantly enhances the readability and reusability of the code. Although C does not inherently support object-oriented programming, we emulated these concepts using structs. This approach allows us to create modular and encapsulated code, which is easier to manage and extend.

4.1.7.1 Encapsulation with Structs

In our implementation, we encapsulate the details of the struct to hide its internal structure from the user, promoting modularity and encapsulation. This is achieved by declaring the struct in the header file (`queue.h`) and defining its implementation in the source file (`queue.c`). By doing so, we separate the interface from the implementation, adhering to the principles of object-oriented design. The same logic applies to other structs declarations, like the real time clock struct (fig. 32).

```c
#ifndef _QUEUE_H_
#define _QUEUE_H_

#include <lcom/lcf.h>

#include <stdint.h>
#include <stdio.h>

struct queue;
typedef struct queue queue_t;
```

(a) Queue.h

```c
#include "queue.h"

struct queue
{
    int *buf;
    int in, out;
    unsigned int size, count;
};
```

(b) Queue.c

Figure 31: Example of object-oriented C - Queue

```c
#ifndef _LCOM_REAL_TIME_CLOCK_H_
#define _LCOM_REAL_TIME_CLOCK_H_

#include <minix/sysutil.h>
#include <lcom/lcf.h>
#include "RTC.h"

struct real_time;
typedef struct real_time real_time_info;
```

(a) RTC.h

```c
struct real_time {
    uint8_t year;
    uint8_t month;
    uint8_t day;
    uint8_t hours;
    uint8_t minutes;
    uint8_t seconds;
};
```

(b) RTC.c

Figure 32: Example of object-oriented C - RTC

4.1.7.2 Advantages of Object-Oriented C

Emulating object-oriented programming in C provides several benefits:

- **Modularity:** By hiding the internal details of structs, we create a modular codebase where each module can be developed and tested independently.
- **Reusability:** Encapsulated code can be reused across different parts of the project or even in other projects, reducing redundancy and improving maintainability.
- **Maintainability:** Changes to the implementation details of a module do not affect other parts of the code that use the module, as long as the interface remains consistent. This separation of concerns makes the code easier to maintain and evolve.

4.1.7.3 Example: Queue Module

The Queue Module in our project demonstrates these principles effectively. The declaration of the struct in `queue.h` provides the interface for creating and manipulating queues, while the definition in `queue.c` contains the implementation details.

- **Queue.h:** This file declares the queue struct and the functions that operate on it, such as `push`, `pop`, and `is_empty`. By including only the necessary declarations, we ensure that users of the Queue Module have access to the interface without knowing the underlying implementation.

- **Queue.c:** This file defines the actual implementation of the queue functions. By separating the implementation from the declaration, we can modify the internal workings of the queue without affecting the code that uses it.

```h
#ifndef _QUEUE_H_
#define _QUEUE_H_

#include <lcom/lcf.h>

#include <stdint.h>
#include <stdio.h>

struct queue;
typedef struct queue queue_t;
```

(a) Queue.h

```c
#include "queue.h"

struct queue
{
    int *buf;
    int in, out;
    unsigned int size, count;
};
```

(b) Queue.c

Figure 33: Example of object-oriented C - Queue

By incorporating object-oriented principles in C, we enhance the structure and clarity of our code. The Queue Module is just one example of how encapsulation and modularity can be achieved in C, leading to a more maintainable and reusable codebase. This approach allows us to manage complexity effectively and ensures that our system remains robust and adaptable to future changes.

4.1.8 Serial Port

The implementation of the serial port plays a crucial role in the project, since MINIX cannot connect to the internet directly, let alone to the Mosquitto broker. To send and receive various control commands and status information, a socket was used for the serial port implementation instead of a named pipe. The objective was not to communicate with another virtual machine but with the host computer, where the Mosquitto broker would be running. Although the goal differed, the implementation of the serial port on the MINIX side was standard, as it would be if the purpose were to communicate between two virtual machines.

4.1.8.1 Serial Port Initialization

The serial port initialization process begins with the `serial_initial_config()` function (fig. 34). This function encompasses several key tasks, including the creation of a queue for message handling. Additionally, it configures the serial port by adjusting the settings of the Interrupt Enable Register (IER) to enable interrupts. Moreover, it sets up the Line Control Register to ensure that each character comprises 8 bits, with bits 0 and 1 set to 1, establishing the use of 1 stop bit (with bit 2 set to 0). Furthermore, the parity control is configured to odd (bits 5, 4 and 3 set to 0, 0, and 1, respectively), signifying that the message expects an odd number of "1" bits to maintain parity.

```c
int serial_initial_config() {
  if ((queue = new_queue()) == NULL) {
    printf("Failed to create queue\n");
    return 1;
  }

  uint8_t ier;
  if (util_sys_inb(COM1_BASE_REG + IER, &ier)) {
    printf("Failed sys_inb IER\n");
    return 1;
  }

  ier &= IER_LS4B_MASK;

  if (sys_outb(COM1_BASE_REG + IER, ier | BIT(0))) {
    printf("Failed sys_outb IER\n");
    return 1;
  }

  uint8_t lcr = 0;
  if (util_sys_inb(COM1_BASE_REG + LCR, &lcr)) {
    printf("Failed sys_inb LCR\n");
    return 1;
  }

  lcr |= 0x0B;
  lcr &= 0xCB;
  if (sys_outb(COM1_BASE_REG + LCR, lcr)) {
    printf("Failed sys_outb LCR\n");
    return 1;
  }

  return 0;
}
```

Figure 34: Serial Port Initialization

4.1.8.2 Subscribing and Unsubscribing Interrupts

To handle interrupts generated by the serial port, the system subscribes to the serial port interrupts using `serial_port_subscribe_int()` which makes use of the IRQ Line 4, as we are using COM1, and unsubscribes using `serial_port_unsubscribe_int()`. These functions enable and disable interrupt handling for the serial port, respectively (fig. 35).

```c
int serial_port_subscribe_int(uint8_t *bit_no) {
  if (bit_no == NULL)
    return 1;

  *bit_no = BIT(serial_port_hook_id);

  if (sys_irqsetpolicy(COM1_IRQ, IRQ_REENABLE | IRQ_EXCLUSIVE, &serial_port_hook_id)) {
    printf("Failed to subscribe Serial Port Interrupts\n");
    return 1;
  }

  return 0;
}

int serial_port_unsubscribe_int() {
  return sys_irqrmpolicy(&serial_port_hook_id);
}
```

Figure 35: Subscribing and Unsubscribing Interrupts

4.1.8.3 Sending Messages

The `send_serial_port_msg()` function is responsible for sending messages through the serial port. It ensures that the transmitter is empty before transmitting data, by checking the bit 6 of the status byte, and tries at most 10 times before returning error.

```c
int send_serial_port_msg(uint8_t msg) {
  uint8_t st;
  uint8_t att = ATTEMPTS;

  do {
    if (get_serial_port_status(&st)) {
      printf("Failed to get status of the serial port\n");
      return 1;
    }

    if (st & TRANSMITTER_EMPTY) {
      return sys_outb(COM1_BASE_REG + THR, msg);
    }

  } while (att-- > 0);

  return 1;
}
```

Figure 36: Sending Messages

4.1.8.4 Receiving Messages

Conversely, the `read_serial_port_msg()` function reads incoming messages from the serial port. It checks the receiver status to ensure that data is available for reading and handles any errors that may occur during reception, such as an overrun error (bit 1), a parity error (indicating that the odd number of "1" bits was different from expected), or a frame error. Subsequently, the received message is pushed to the queue for further analysis.

```c
int read_serial_port_msg() {
  uint8_t st;
  uint8_t msg;

  if (get_serial_port_status(&st)) {
    printf("Failed to get status of the serial port\n");
    return 1;
  }

  if (st & RECEIVER_READY) {
    if (util_sys_inb(COM1_BASE_REG + RBR, &msg)) {
      printf("Failed sys_inb RBR\n");
      return 1;
    }

    if (st & SER_OVERRUN_ERR) {
      printf("Error reading serial port Overrun Error\n");
      return 1;
    }

    if (st & SER_PARITY_ERR) {
      printf("Error reading serial port Parity Error\n");
      return 1;
    }

    if (st & SER_FRAME_ERR) {
      printf("Error reading serial port Frame Error\n");
      return 1;
    }

    while (push(queue, msg))
      ;
    return 0;
  }

  return 1;
}
```

Figure 37: Receiving Messages

4.1.8.5 Interrupt Handling

Interrupts generated by the serial port are handled by the `serial_port_int_handler()` function. This function identifies the type of interrupt and processes it ac-

cordingly. In particular, it handles character timeout interrupts by reading and processing incoming messages.

```
void serial_port_int_handler() {
  uint8_t interrupt_ident;
  if (util_sys_inb(COM1_BASE_REG + IIR, &interrupt_ident))
    return;
  if ((interrupt_ident & NO_INT_PENDING)) {
    printf("No interrupt pending\n");
    return;
  }
  if (interrupt_ident & INT_PENDING == CHAR_TIMEOUT_FIFO) {
    while (read_serial_port_msg())
      ;
  }
}
```

Figure 38: Interrupt Handling

4.1.8.6 Clearing Interrupts and Resources

The `serial_port_clear_int()` function clears any pending interrupts and flushes the FIFO buffer to ensure proper operation. Additionally, the `serial_port_clear_all()` function clears all resources associated with the serial port, including the message queue.

```
int serial_port_clear_int() {
  if (sys_outb(COM1_BASE_REG + FCR, FIFO_CLEAR)) {
    printf("Failed sys_out FCR\n");
    return 1;
  }

  clear_queue(queue);

  return 0;
}
```

```
void serial_port_clear_all() {
  delete_queue(queue);
}

queue_t *get_queue() {
  return queue;
}
```

Figure 39: Clearing Interrupts and Resources

4.1.8.7 Message Queue

A message queue, implemented using a FIFO (First-In-First-Out) data structure, is utilized to store incoming messages temporarily. This queue enables efficient handling of received messages and facilitates communication between the serial port and other system components.

For an illustrative representation of the serial port implementation code, please refer to the provided screenshots.

4.1.9 Our API

To facilitate communication between the different devices—specifically, the MINIX system, which acts as the "brain" of the project, the host, which communicates with MINIX and serves as an intermediary between MINIX and the IoT devices—a specific API was defined by the group. Given that the serial port is configured to send messages with 8 bits (1 byte), the standard message in our API adheres to this length (the "command byte"). These 8 bits are divided as follows: the 3 most significant bits designate the device being controlled (refer to Table 3 for more details), while the remaining 5 bits are specific to the type of device.

Device Code	**Device Type**
000	Lights
001	Buzzer
010	Display
011	Motor
100	Humidity and Temperature Sensor
101	Decibel Sensor
110	Camera

Table 3: Device Codes (3 most significant bits of the message)

For the **Lights**, the 5 bits represent the state of each light. Since we have a maximum of 5 lights, each one has a separate bit in the command byte. For example, if we want to turn on light 1, we need to send bit 0 active (1), or in case we want to turn on lights 1 and 2, we should activate bits 1 and 0.

For the **Buzzer**, the same logic as the lights applies. We have 2 buzzers, so bits 0 and 1 should be used. A key difference from the lights is that the light remains active until a 0 is sent (in the corresponding bit), but for the buzzer, a timer is set to 1 second.

Regarding the **Display**, the 5 least significant bits correspond to the number of packets that we want to send, allowing a maximum of 32 packets (0 meaning that we want to send 1 packet), each one corresponding to an ASCII code. Essentially, we are allowed to send messages of size 32.

For the **Motor**, the 5 least significant bits are scaled from 0 to 31 to 0 to 255, meaning that 255 is the maximum thrust applied (3 volts) and 0 is the minimum (0 volts).

All the **Sensors** and the **Camera** do not use the 5 least significant bits, as we are just requesting the data to be sent by the ESP to the Arduino.

4.2 Topics Extra

4.2.1 MQTT

Message Queuing Telemetry Transport (MQTT) is a standards-based messaging protocol used for machine-to-machine communication. Its lightweight nature and ease of implementation make it ideal for efficiently transmitting and receiving IoT data over resource-constrained networks with limited bandwidth.

The MQTT protocol operates on a publish/subscribe model, facilitating direct communication between clients and servers. To send a message, a client connects to a broker and publishes the message to a specific topic. When the broker receives the message, it redirects it to all clients subscribed to that topic.

In this project, Mosquitto MQTT was used as the broker. The objective was to control various devices from a MINIX system. The user specifies the device to be controlled, and upon receiving the command via the serial port, the host uses MQTT to send the message to a microcontroller—in this case, the ESP32-CAM—which then turns the specified device on or off.

For example, if a user wants to turn on a light, they click a button and the Interrupt handler will capture the click. The MINIX then sends this command to the host, using the Serial Port, which publishes it as a message on the topic "Lights." The broker, upon receiving the message, redirects it to all clients subscribed to the "Lights" topic. The ESP32-CAM, being one of these clients, receives the message, processes the command, and turns the light on.

4.2.2 Arduino

To control IoT devices, two ESP-32 CAM microcontrollers were used. Both were connected to the same network as the host program and to the Mosquitto broker. The code was implemented similarly to the main program running in MINIX, utilizing handlers and function pointers (as illustrated in figure 40). The microcontrollers subscribe to Mosquitto topics and redirect each request from the host program to the corresponding handler. The need for two microcontrollers arose from the insufficient number of GPIO pins for all the proposed devices. The first microcontroller controls the lights, buzzers, and humidity and temperature sensor, while the second one handles motor, display, and camera requests.

```
typedef struct {
  char *topic;
  void (*redirect_message)(byte *payload, unsigned int length);
} on_message_t;

static const on_message_t on_message[] = {
  {LIGHT_TOPIC, turn_light},
  {BUZZER_TOPIC, turn_buzzer},
  {MOTOR_TOPIC, turn_motor},
  {HUMIDITY_SENSOR_TOPIC, turn_humidity_sensor},
  {DECIBEL_SENSOR_TOPIC, turn_decibel_sensor},
  {PICTURE_TOPIC, take_picture}};
```

(a) Array of Structs

```
void callback(char *topic, byte *payload, unsigned int length) {
  Serial.print("Message arrived in topic: ");
  Serial.println(topic);
  Serial.println("-----------------------");

  size_t num_topics = sizeof(on_message) / sizeof(on_message[0]);

  for (size_t i = 0; i < num_topics; i++) {
    if (strcmp(topic, on_message[i].topic) == 0) {
      on_message[i].redirect_message(payload, length);
    }
  }
}
```

(b) Callback Function

Figure 40: Array of Struct Request and Callback Function

The code was developed in Arduino and is divided into two parts: the **setup** and the **loop**. In the **setup** phase, all the pins are configured as either OUTPUT or INPUT, and the internet and Mosquitto configurations are established. The **loop** phase is an infinite loop where the device operates and performs its tasks.

4.2.2.1 Setup Phase

The code begins by including several libraries necessary for the functioning of the ESP-32 CAM, WiFi, DHT sensor, MQTT communication, and camera functionality. Objects for WiFi and MQTT communication are initialized, as well as the DHT sensor object.

In the `setup` function, GPIO pins are configured for various devices such as lights, buzzers, and sensors. The Serial communication is initialized for de-

bugging purposes. The WiFi and MQTT configurations are also established in this phase.

A camera configuration structure is defined, specifying various pins and settings for the camera module. The initialization of the camera is currently commented out, but it includes error handling to ensure proper setup.

4.2.2.2 Message Handling

The `on_message_t` structure maps MQTT topics to their corresponding handler functions. The `callback` function listens for incoming messages on subscribed topics and redirects them to the appropriate handler based on the topic.

4.2.2.3 Handler Functions

Several handler functions are defined to control different devices:

- `take_picture`: Captures a photo using the camera and publishes it over MQTT.
- `turn_light`: Controls the state of multiple lights based on the received payload.
- `turn_buzzer`: Manages the buzzer's state and tone based on the payload.
- `turn_motor`: Adjusts the motor's speed.
- `turn_humidity_sensor`: Reads humidity and temperature data from the DHT sensor and publishes it.
- `turn_decibel_sensor`: Handles requests for decibel sensor data.

4.2.2.4 MQTT Connection Management

The `connect_mqtt` function establishes a connection to the MQTT broker and subscribes to the required topics. It also sets the callback function for incoming messages.

4.2.2.5 Main Loop

The `loop` function runs continuously, ensuring that the MQTT client stays connected and processes incoming messages. If the connection is lost, it attempts to reconnect.

4.2.3 Socket

The communication via the serial port in VirtualBox can be simulated using various methods, such as named pipes or TCP. Our group opted to use the TCP protocol to facilitate the exchange of information between MINIX and the host. This approach was chosen because it was not feasible to physically connect the host computer to the emulated environment running in VirtualBox. Consequently, a program was developed on the host to listen for TCP requests, which correspond to the requests sent by MINIX via the serial port. Additionally, the host needed to communicate with the microcontroller responsible for processing and responding to these requests. To achieve this, a multi-threaded approach was implemented. One thread listens for TCP requests, while the other listens for Mosquitto messages, which are then relayed through MINIX.

In the file ‘mosquitto.c‘, the function `server_thread(void *arg)` handles the tasks performed by the thread listening for requests on the TCP server. Initially, it ensures proper socket initialization and binds the socket, as illustrated in Figure 41. The main body of this function (Figure 42) continuously accepts connections on the server and reads incoming messages. These messages are then redirected to the function `mosquitto_command_byte()`, which is responsible for publishing the request to the appropriate topic, as shown in Figure 43.

```c
void *server_thread(void *arg) {
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    int client_socket;
    uint8_t buffer[1];
    ssize_t bytes_read;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(SERVER_IP);
    server_addr.sin_port = htons(SERVER_PORT);

    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    if (listen(sockfd, 3) < 0) {
        perror("Listen failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on %s:%d\n", SERVER_IP, SERVER_PORT);
```

Figure 41: Socket Initialization

```c
while (1) {
    client_socket = accept(sockfd, (struct sockaddr *)&client_addr, &client_addr_len);
    if (client_socket < 0) {
        perror("Accept failed");
        continue;
    }

    printf("Client connected\n");

    while ((bytes_read = read(client_socket, buffer, sizeof(buffer))) > 0) {
        if (bytes_read == 1) {
            uint8_t received_byte = buffer[0];
            printf("Received byte: 0x%02x\n", received_byte);
            mosquitto_commmand_byte_received(received_byte);
        }
    }

    if (bytes_read == -1) {
        perror("Read from client failed");
    }

    close(client_socket);
    printf("Client disconnected\n");
```

Figure 42: Socket Body

```c
void mosquitto_commmand_byte_received(uint8_t command) {
  uint8_t device = (command >> 5);
  uint8_t param = command & 0x1F;

  printf("Publishing to device: %x\n", device);
  printf("Parameter: %x\n", param);
  printf("Topic: %s\n", redirect[device].topic);
  mosquitto_publish(mosq, NULL, redirect[device].topic, 1, &param, 1, false);
}
```

Figure 43: Socket Redirect

4.2.3.1 Receiving Messages

The messages are received by the `read()` function in the main loop, as illustrated in Figure 42. This function reads from the socket 8 bits at a time. The information read here originates from the Serial Port implemented in MINIX.

4.2.3.2 Sending Messages

To send a message back to MINIX, such as the temperature and humidity data, the function `serial_port_send_message()` was created. This function writes a message of 8 bits (1 byte) to the socket file descriptor, aligning with the MINIX serial port's configuration to receive 8-bit packets. The implementation of this function is shown in Figure 44.

```c
int serial_port_send_message(int socketfd, const void *message) {
  if (send(socketfd, message, 1, 0) != 1) {
    perror("Send failed");
    return 1;
  }

  return 0;
}
```

Figure 44: Socket Send Message

5 Conclusions

We successfully achieved our ambitious goals and managed to implement some labs that were not fully covered in class, such as the real-time clock and the serial port.

The most challenging aspect of the project was implementing the serial port to establish communication with the host computer, enabling MINIX to connect to the internet. This task required extensive effort and problem-solving, but we were able to overcome the challenges through collaboration and perseverance.

In addition to the software implementation, we also embarked on a real-world implementation of the physical model with a proper lighting system, buzzers, motor, an LCD, and other devices. Integrating devices such as the real-time clock, video card, mouse, keyboard, and serial port allowed us to create a comprehensive IoT system. This practical application provided us with valuable insights into the complexities of hardware-software integration and system design.

As members of the group with a passion for embedded systems, this project was particularly engaging and rewarding. It provided us with valuable hands-on experience and allowed us to apply theoretical concepts in a practical setting. Despite its complexity, we thoroughly enjoyed working on the project and are proud of the results we achieved.

6 Appendix A: Installation Instructions

6.1 VirtualBox Configuration

Since this project does not use Named Pipe as the emulated Serial Port, as suggested by the instructor, it's necessary to adjust the virtual box settings. First, change the "Ports" to TCP and enter the TCP/IP used by the host program (normally localhost/4321 works), as shown in Figure 45. Then, check the port forwarding settings by navigating to the network settings, advancing to port forwarding, and configuring them accordingly as depicted in Figures 46 and 47.

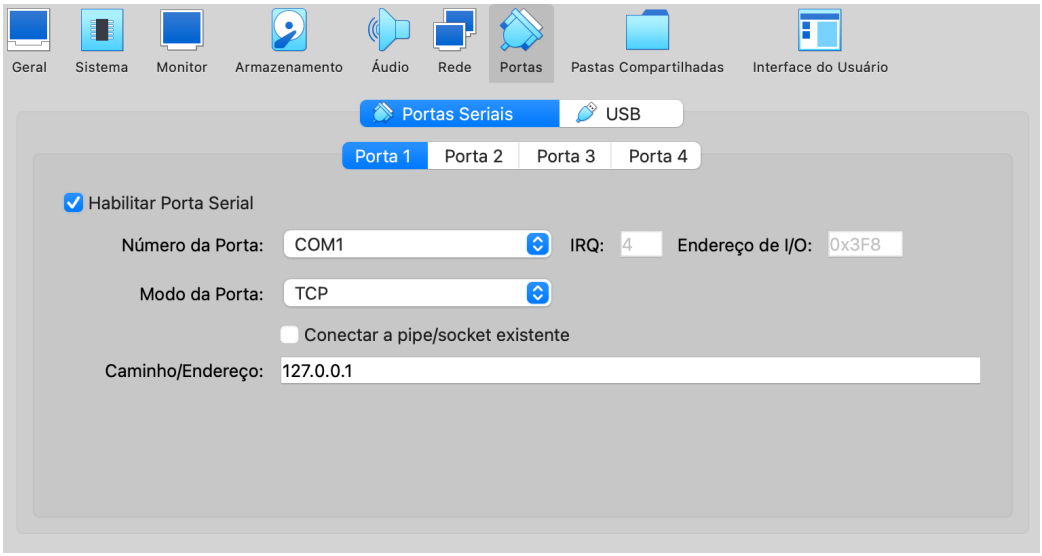


Figure 45: Serial Port Configuration - VirtualBox

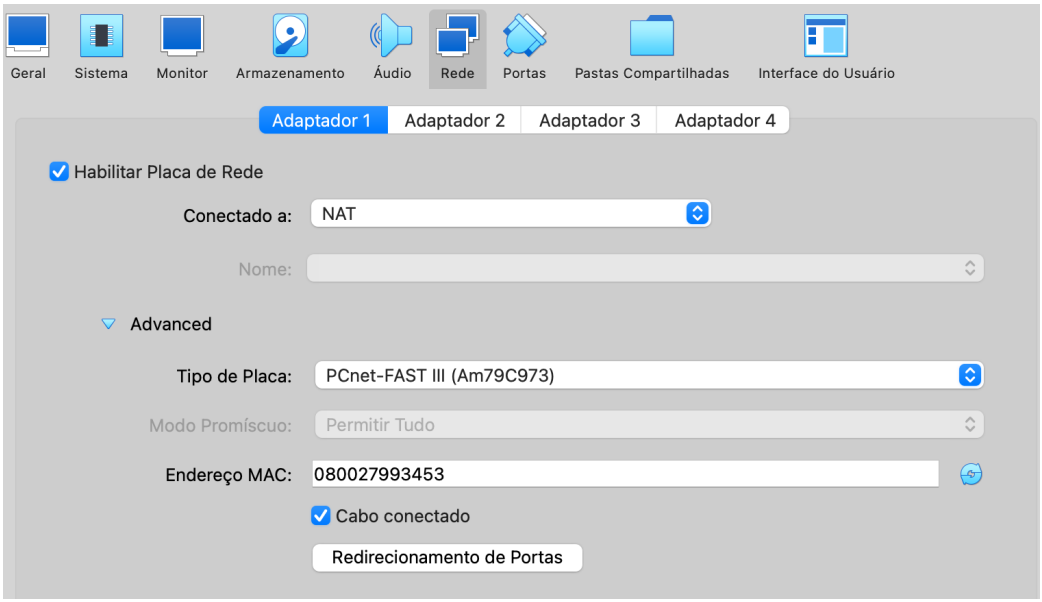


Figure 46: Network Configuration - VirtualBox

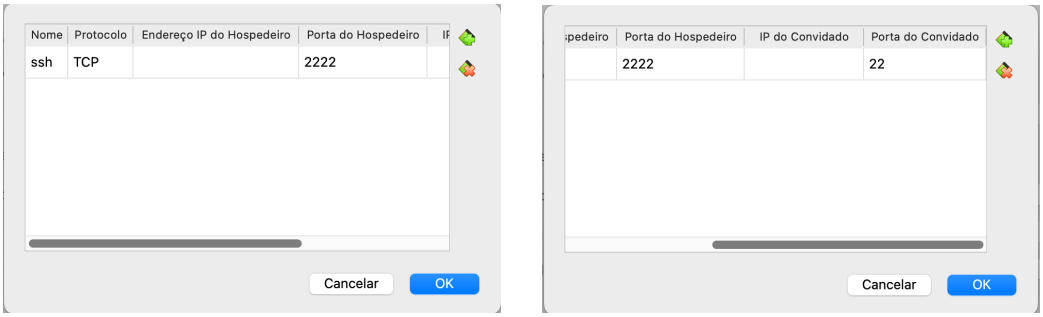


Figure 47: Port Forwarding Configuration - VirtualBox

6.2 Mosquitto Broker

First, ensure that Mosquitto is installed on your computer. To start the Mosquitto broker on the host computer, navigate to the **host_Server** directory, update the Mosquitto IP and Port to match your network settings (where your computer is turned on), and run the following command: `mosquitto -c mosquitto.conf`.

6.3 Host Program

This project requires a process running on the host computer. To achieve this, navigate to the `host` directory, compile the code by running `make`, and execute it using `./HOST`.

7 Appendix B: Maquette

As a visual aid, we built a maquette with 5 LEDs, an 8-port relay module, 2 buzzers, a 3V DC motor and fan, two ESP32-CAM modules, and one Arduino Uno R3. The photo of the maquette is shown in Figure 48.

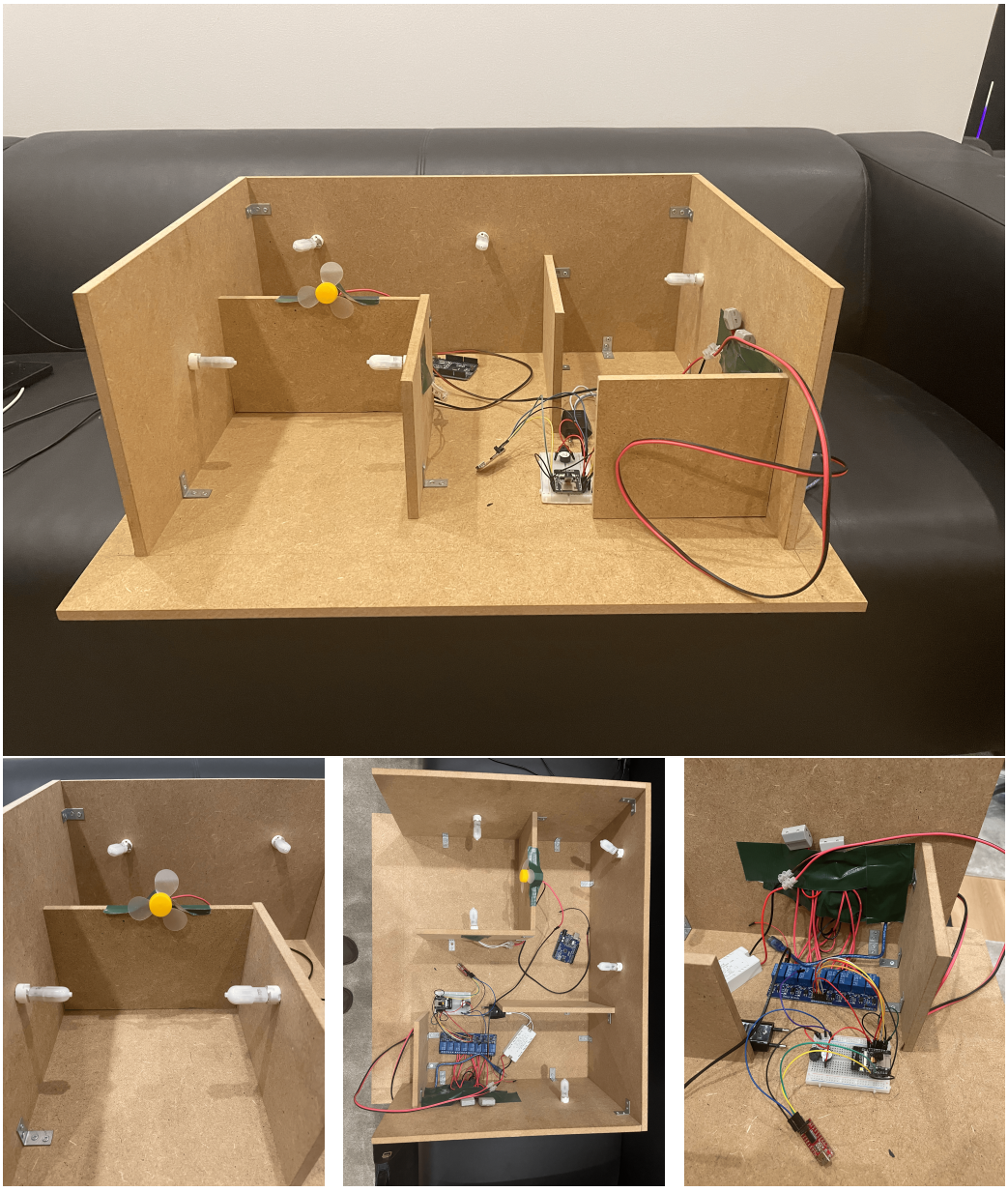


Figure 48: Maquette Images