Mazeduino

André Osório
Faculdade de Engenharia
Universidade do Porto
Porto, Portugal
up202004732@up.pt

Miguel Magalhães
Faculdade de Engenharia
Universidade do Porto
Porto, Portugal
up201708985@up.pt

Rodrigo Oliveira
Faculdade de Engenharia
Universidade do Porto
Porto, Portugal
up202008600@up.pt

***Abstract*—This report describes the development process of the Mazeduino game, a real-time system that implements the classic wood maze game on a microcontroller. The project utilises an ESP32 microcontroller with the Arduino framework to manage multiple tasks, including input handling, output control, and game logic. The system employs Adafruit Flora LSM303DLHC accelerometers to capture real-time tilt data that serves as the primary control input for the maze game, push buttons for the selection process and Nokia 5110 LCDs for visualisation.**

I. General Explanation

Introduction

This report describes the process behind the development of the digital maze game project, undertaken as part of the Embedded Systems course. The main objective of the project was to apply the theoretical concepts of Real-Time Operating Systems (RTOS), discussed during course lectures, to a real-world application. By designing and implementing a system capable of managing concurrent tasks with precise timing, the project aimed to demonstrate the practical use of RTOS principles in embedded systems. This report begins by discussing the hardware used and the objectives of the work carried out. Then comes the information regarding the software implementation and according tasks as well as the results. Finishing by presenting an adequate conclusion and the weights for each team members' contribution.

Hardware Specifications

The hardware setup for the project consisted of an ESP32 microcontroller (fig. 1) running the Arduino framework, which is built on top of FreeRTOS, a widely used real-time operating system. The ESP32 was chosen over alternative ESP8266 due to its ability to support both FreeRTOS and the Arduino platform libraries simultaneously, providing greater flexibility, simplicity and computational power for real-time applications.

FreeRTOS simplifies the development of real-time applications by providing advanced task management features, such as scheduling and prioritization. This, in combination with the ESP32's computational power and a clock frequency ranging from 80 MHz to 240 MHz, provides an ideal environment for developing parallel tasks with precise timing requirements.

For the purposes of versus mode two-player multiplayer, two controllers where built using veroboards. Each controller has one accelerometer to gather the necessary data to make the ball move. The devices are Adafruit Flora LSM303DLHC - fig. 2.

Fig. 1. ESP 32 microcontroller

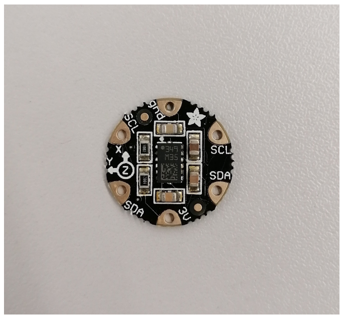

Fig. 2. Adafruit Flora LSM303DLHC accelerometer

Each controller also has its own Nokia 5110 LCD display (fig. 3) where the game will be rendered.

The breadboard, where the microcontroller is placed, also has two control buttons, which are simple push buttons - fig. 4.

The following fig. 5 shows all of the components used and their according connections. The following figure (fig. 6 illustrates these connections via a diagram.

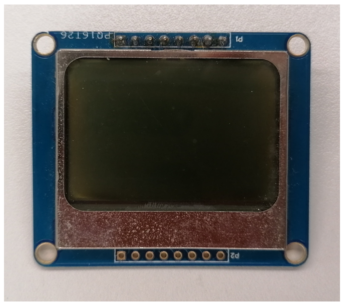

Fig. 3. Nokia 5110 LCD display

Fig. 4. Push Buttons

Objectives

The primary goal of the project was to create an interactive real-time application, capable of managing concurrent tasks, including frame rendering for animations, user input handling, and game logic execution, while applying the following basic RTOS principles:

- Task Scheduling: Ensure that different tasks are executed at precise intervals and within their timing constraints.
- Task Prioritisation: Assigning higher priorities to critical tasks, such as the screen refresh, to maintain system responsiveness.
- Timing and Synchronisation: Utilising FreeRTOS's mechanisms to maintain smooth transitions and consistent animations.

The main overall objective set to classify whether the project's real-time system implementation was successful or not was whether both displays were capable of adequately refreshing at a minimum of 30 frames per second (fps), which is equivalent to a 30 Hz frequency. This decision was made since the least amount a time a human eye will start to perceive frame jitter-ness is that of 30 ms, any value below it will compromise the user experience of the game.

Game Objective

- Each player tilts their controller to navigate a virtual ball through a maze displayed on-screen, racing to reach the goal first and/or to beat their given high-score.

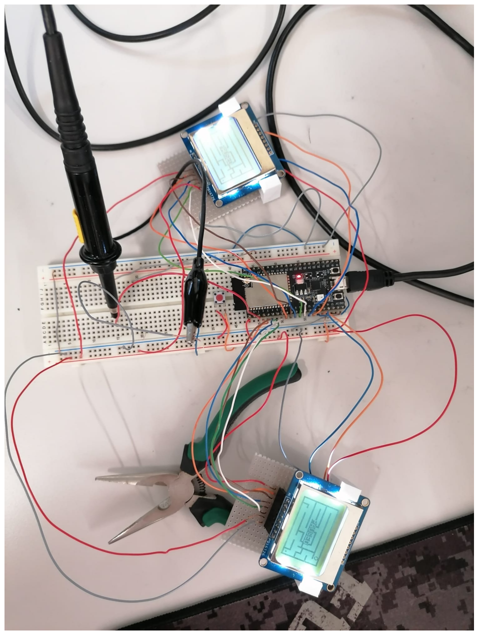

Fig. 5. Full component implementation

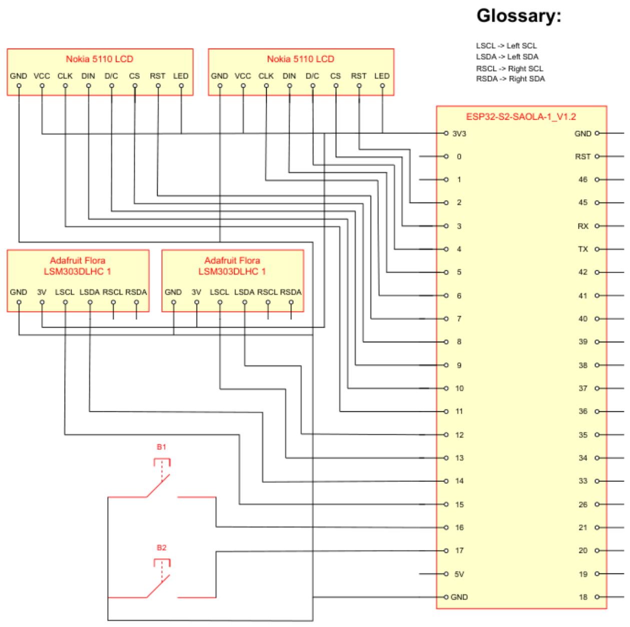
Glossary:
LSCL -> Left SCL
LSDA -> Left SDA
RSCL -> Right SCL
RSDA -> Right SDA
Nokia 5110 LCD
GND VCC CLK DIN D/C CS RST LED
Nokia 5110 LCD
GND VCC CLK DIN D/C CS RST LED
ESP32-S2-SAOLA-1_V1.2
3V3 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 5V GND
GND RST 46 45 RX TX 42 41 40 39 38 37 36 35 34 33 26 21 20 19 18
Adafruit Flora LSM303DLHC 1
GND 3V LSCL LSDA RSCL RSDA
Adafruit Flora LSM303DLHC 1
GND 3V LSCL LSDA RSCL RSDA
B1
B2

Fig. 6. Diagram of all component connections

II. Implementation and Software Architecture

Task Description

The system was divided into three main tasks, each responsible for a specific functionality:

- Accelerometer Task:
  - Periodically reads both accelerometers over the I2C bus, filters the raw measurements, and computes the board orientation used as input to the game logic.

- Display Task:
  - Updates both displays at a fixed refresh rate, drawing the maze, player positions, and game status while avoiding visible frame drops.
- Game logic & main screen Task:
  - Handles the state machine that determines the state of the game. There are two possible options from the MENU, PLAY or CREDITS. Inside the PLAY mode, there is another state machine to handle the button operation inside this state, which is also handled by this task.
- Game physics task:
  - Computes positions and velocities based on the accelerometer data. Handles the collisions with the walls and the determines if the player reached the finish area.
- Button Task:
  - Monitors the physical buttons, debounces their signals, and generates events for menu navigation, pause/ resume functions, and select.

The period (and deadline) of each task is summarized in table I.

TABLE I
TASK PERIODS AND DEADLINES

Task	Period = Deadline (ms)
Accelerometer Task	33
Display Task	33
Game logic & main screen Task	100
Game physics Task	33
Button Task	20

We display 5 tasks only but the system had a total of 7 tasks. This is because each player has one Accelerometer Task and one Display Task, making it two extra tasks in total.

Tasks Execution Times

In order to properly quantize the effectiveness of the real-time system developed, the execution times of each task where measured. This measurement was made with only one task operational - as far as the microcontroller was aware, other tasks were not even created. The listing 1 illustrates how these measurements were made. Within the tasks pre-existing code, additional variables, such as *beginTime* and *endTime*, and calculations, such as *print_resuslts()* function, were added in order to get the values in table II.

Listing 1. Task execution time measurement example

```
void function_name()
{
    // Local function variable (including time variables) go here

    for(;;)
    {
        beginTime = micros();

        // Execute task relevant code here

        endTime = micros();
        diffTime = endTime - beginTime;
        print_results(); // Function to get min/avg/max and log results to terminal
    }
}
```

The following table (table II) illustrates the execution times for each task, including the 'duplicate' ones. The measurements for the averages were made using 2002 iterations rounding up to the third decimal case.

TABLE II
TASKS EXECUTION TIMES

Task	Execution Times (μs)		
	Min	Avg	Max
Accelerometer Task 1	1084	1087.699	1098
Accelerometer Task 2	1083	1087.682	1098
Display Task 1	2142	2148.965	2160
Display Task 2	2135	2142.605	2151
Game logic & main screen Task	18	19.998	20
Game physics Task	15	16.350	22
Button Task	2	4.437	10

As to be expected, the tasks that require querying sensors (accelerometers and displays) lead to noticeably higher overall execution times, whilst purely computational tasks (Game logic & main screen and Game physics) and simple electrical signal sensing tasks (Buttons) have a lower overall execution time. This is logical, as the additional hardware processing, library calculations and the communication back to the microcontroller add significant overhead - the most likely one to cause a bottle-neck is precisely the task that gives this project its main real-time constraint, the display tasks.

With the previous table, it is also possible to conclude that, if no extraordinary actions occur to the system, it will 100% of the time reach its goal of sub 33 ms display refresh rate, as the sum of the worst-case execution time of all tasks is a mere 6.559 ms.

Additionally, tests with an oscilloscope, to further guarantee the adequate refresh rate by the displays, was performed. The test performed was similar to that of the listing 1, there was however a substitution of the time related variables for a *digitalWrite(PIN,HIGH)* at the beginning of the for-loop and a *digitalWrite(PIN,LOW)* at its end. This way, the oscilloscope, measuring this pin's value would display the frequency of the display task with a slight margin of error (as writing a value to the pin adds a slight overhead). The following fig. 7 illustrates this.

Rate Monotonic Scheduling

To ensure that all tasks meet their timing requirements, the system uses Rate Monotonic Scheduling (RMS), a fixed-priority scheduling algorithm. In RMS, tasks with shorter periods are assigned higher priorities:

- The priority order for the tasks is as follows:
  - 1^{st} Button Task - shortest period (20 ms);
  - 2^{nd} Game physics Task - 33 ms period;
  - 3^{rd} LCD Task 1 - 33 ms period;
  - 4^{th} LCD Task 2 - 33 ms period;
  - 5^{th} Accelerometer Task 1 - 33 ms period;
  - 6^{th} Accelerometer Task 2 - 33 ms period;
  - 7^{th} Game logic & main screen Task - highest period (100 ms).

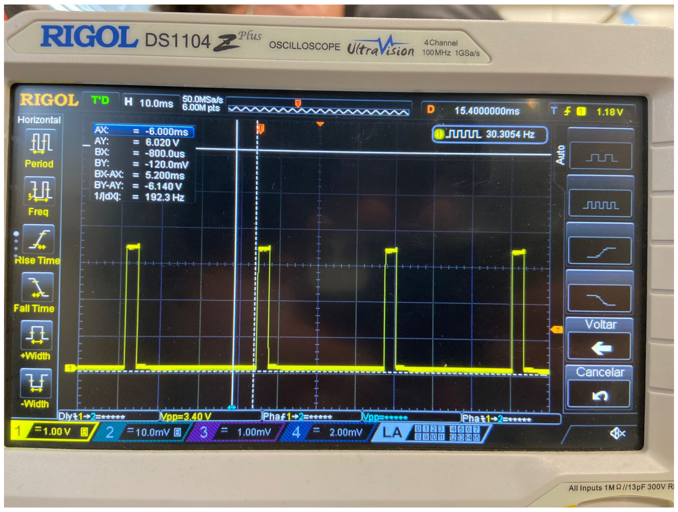

Fig. 7. Oscilloscope Measurements

Note that some tasks, even though they have the same period, hold higher priority. This is because the priority assignment is done through an array that merely organizes the tasks according to their period, tasks with the same period will organized by order of selection, in other words, the lower the array index the higher their priority if periods match.

CPU Utilization Analysis

To evaluate the schedulability of the system under RMS, the utilization-based test Least Upper Bound was used. This test provides a sufficient condition to guarantee that all tasks meet their deadlines, ensuring that at least one instance of each task is executed within its respective period. The test is based on the following condition:

$$U(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{1}$$

Where:

- C_i: Execution time of task i.
- T_i: Period of task i.
- n: Number of tasks in the system.

Summing up the utilizations:

$$U(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} = X + Y + Z + ... + = 0.1988 \tag{2}$$

For $n = 7$ tasks, the theoretical utilization bound is given by:

$$n\left(2^{\frac{1}{n}} - 1\right) = 7\left(2^{\frac{1}{7}} - 1\right) = 0.7286 \tag{3}$$

Since the calculated utilization $U(n) = 0.1988$ is less than the theoretical upper bound 0.7286, it can be concluded that the system is schedulable under RMS. This means that all tasks will meet their deadlines, ensuring the proper operation of the real-time system.

Response Time Analysis

To further validate the schedulability of each task, we use an iterative technique to calculate the Response Time Worst-Case (RWC) for each task. The iterative process continues until the response time converges (i.e., $Rwc_i(m+1) = Rwc_i(m)$) or diverges (i.e., $Rwc_i(m+1) > D_i$, where D_i is the task's deadline that in this case is equal to the period).

The response time is calculated using the following iterative equations:

$$Rwc_i(0) = \sum_{k \in hp(i)} C_k + C_i$$

$$Rwc_i(m+1) = \sum_{k \in hp(i)} \left\lceil \frac{Rwc_i(m)}{T_k} \right\rceil \cdot C_k + C_i$$

where:

- C_k: Execution time of a higher-priority task k.
- C_i: Execution time of the current task i.
- T_k: Period of task k.
- $hp(i)$: Set of tasks with higher priority than task i.

The worst-case response times for each task, calculated using the Max values from table II, are summarized below.

TABLE III
WORST-CASE RESPONSE TIMES

Task	**Worst-Case Response Time (μs)**
Button Task	0.010
Game physics Task	0.032
Display Task 1	2.192
Display Task 2	4.343
Accelerometer Task 1	5.441
Accelerometer Task 2	6.539
Game logic & main screen Task	6.559

Since the worst-case response times for all tasks are less than their deadlines as provided in the analysis (table III), the system is schedulable under Rate Monotonic Scheduling.

III. CONCLUSIONS

This project successfully demonstrated the practical application of real-time embedded systems concepts in the development of an interactive game based on a microcontroller platform. The main objectives defined at the beginning of the work were achieved, namely the correct implementation of multiple concurrent tasks, the effective use of FreeRTOS scheduling mechanisms, and the fulfillment of strict timing constraints required for a smooth and responsive user experience. In particular, the system consistently met the target display refresh rate of at least 30 frames per second, validating the real-time nature of the solution.

From a technical standpoint, the use of Rate Monotonic Scheduling, along with utilization and response-time analyses, confirmed that the system is schedulable and robust under worst-case conditions. The separation of functionality into well-defined tasks (sensor acquisition, game logic, physics, display handling, and user input) resulted in a modular and scalable architecture, which simplifies debugging, testing, and future extensions of the project.

Beyond the technical results, this project proved to be a valuable learning experience for the students involved. It allowed them to bridge the gap between theoretical concepts presented in the Embedded Systems course—such as task prioritization, timing analysis, and synchronization—and their practical implementation on real hardware. Through hands-on development, testing, and performance evaluation, the students gained a deeper understanding of real-time system design challenges and constraints, as well as increased confidence in working with embedded platforms and RTOS-based applications.

Overall, the Mazeduino project met its proposed goals and provided meaningful insights into the design and analysis of real-time embedded systems, making it a successful and educational experience within this area of study.

IV. Dedication of each member

André Osório: (33.3%)
Miguel Magalhães: (33.3%)
Rodrigo Oliveira: (33.3%)