

# LcTyper

LC 2021/2022

T7G02

up202007398@fe.up.pt

up202008866@fe.up.pt

up202008867@fe.up.pt

up202008462@fe.up.pt

André Filipe Cardoso Barbosa

Guilherme Cunha Seco Fernandes de Almeida

João Tomás Marques Félix

José Luís Cunha Rodrigues

June 2022

## Table of contents

<b>1</b>	<b>User instructions</b>	<b>3</b>
1.1	Context . . . . .	3
1.2	How to play . . . . .	3
1.3	Main menu . . . . .	3
1.4	Game State . . . . .	4
1.5	Game Over State . . . . .	4
1.6	Leaderboard . . . . .	5
<b>2</b>	<b>Project status</b>	<b>6</b>
2.1	Timer . . . . .	6
2.2	Keyboard . . . . .	6
2.3	Video card . . . . .	6
2.4	Mouse . . . . .	6
2.5	Real-Time Clock (RTC) . . . . .	7
<b>3</b>	<b>Code organization/structure</b>	<b>8</b>
3.1	Drivers . . . . .	8
3.2	States . . . . .	8
3.3	Typer . . . . .	9
<b>4</b>	<b>Implementation details</b>	<b>10</b>
4.1	XPM's . . . . .	10
4.2	Event driven code . . . . .	10
4.3	State Machine . . . . .	10
4.4	Object orientation . . . . .	10
4.5	Assets . . . . .	10
<b>5</b>	<b>Conclusions</b>	<b>11</b>

# 1 User instructions

## 1.1 Context

The theme of our project is a typing tutor. A tool designed to help the user to enhance their typing skills. The main idea is to provide an environment where the user can type and get feedback on their typing speed.

## 1.2 How to play

To start the game just press the play button located on the menu.

When in the game environment a random phrase is displayed and your goal is to write the full sentence as quickly and as accurate as possible. Once you start typing you'll get live feedback of your typing speed (words per minute).

Every time you misspell a letter, that same letter turns red and you can't move on unless you fix your mistake.

## 1.3 Main menu

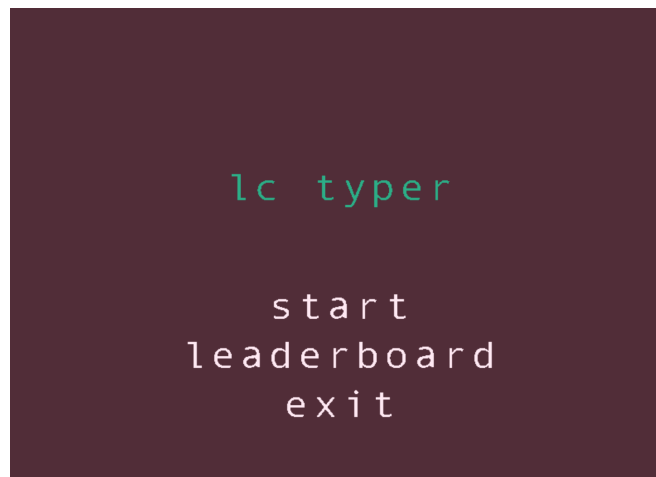


Figure 1: Main menu

The user can select one of the available options using the mouse:

- **Start:** Enter game state and start playing.
- **Leaderboard:** Check the top five all time scores.
- **Exit:** Exit the game.

## 1.4 Game State

As previously mentioned the user is prompted with a random phrase and needs to type the sentence correctly.

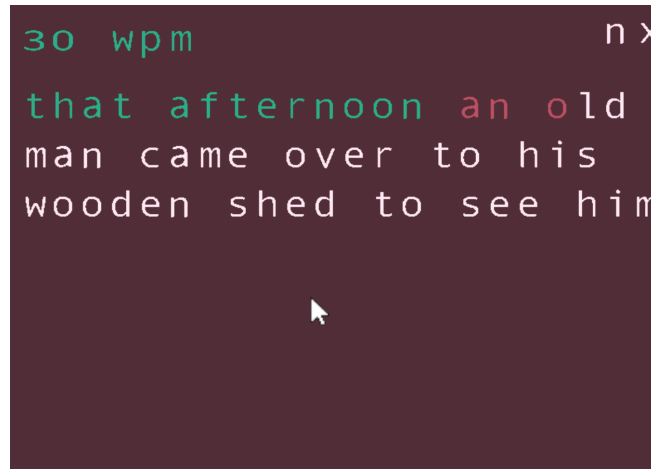


Figure 2: Game State

Using the mouse the user can generate a new sentence by clicking the 'n' shaped button. It's also possible to leave the game by pressing the 'x' button or by pressing ESC. If the user wishes to restart the current test, they can do so by pressing TAB.

## 1.5 Game Over State

The game ends when the user writes the full sentence correctly. In this screen the user gets information about the final words per minute score and accuracy.

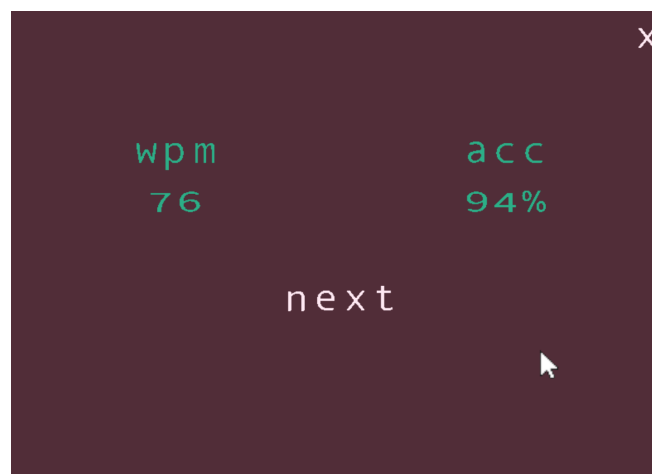
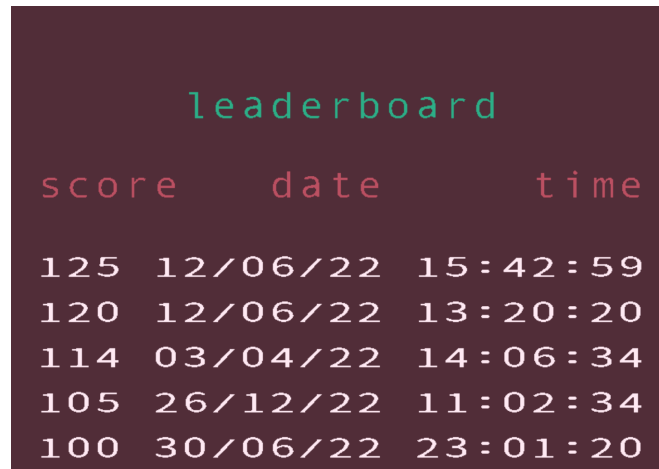


Figure 3: Game Over State

By pressing the 'next' button the user starts a new game with a new random phrase. To go back to the main menu the user can press the 'x' button with the mouse or press ESC on the keyboard.

## 1.6 Leaderboard

The leaderboard displays the top five all time scores and the respective date and time they were set. After each game the scoreboard is updated and ordered if necessary.



score	date	time
125	12/06/22	15:42:59
120	12/06/22	13:20:20
114	03/04/22	14:06:34
105	26/12/22	11:02:34
100	30/06/22	23:01:20

Figure 4: Leaderboard

The user can exit this state by pressing ESC.

## 2 Project status

The usage of each device can be found in the table below. Each device has its own interrupt handler. So the information was of easy access across the program, most interrupts are parsed into an Event struct, that contains essential informations about I/O.

Device	Usage	Method
Timer	Frame rate	Interrupts
Keyboard	Writing text, game navigation	Interrupts
Video card	Rendering the game	None
Mouse	Navigation in the UI	Interrupts
RTC	Periodic interrupts to measure speed and date to save scores	Interrupts

### 2.1 Timer

Timer 0 was used to control the screen refresh rate. Instead of changing the timer's frequency, we worked with Minix's default. This way, screen refresh is done at half the rate of timer. The timer is also used for some animation.

To work with timer interrupts, we implemented `timer_subscribe_int` and `timer_unsubscribe_int`. After each interrupt, `timer_ih` is called and the interrupt is parsed into an Event by `timer_get_event`.

### 2.2 Keyboard

Due to the game's nature, the keyboard is a key component in the program. Key presses generate interrupts and the input drives the state of the game. After a keyboard interrupt, it is parsed into an `Event`. In this conversion, the makecode is transformed into a character (if possible) so it can be interpreted by the game. Additional keys can be used for navigation (e.g. restarting the game).

To handle interrupt subscription `keyboard_subscribe_int` and `keyboard_unsubscribe_int` were implemented. When an interrupt occurs, `kbc_ih` is called and if the read is complete `keyboard_get_event` is used to transform it into an event.

### 2.3 Video card

The video card is a key component of the program, as it is involved in every major aspect of the game. The video mode chosen was `0x115`, as its dimensions better suited our needs. To acquire all information about this mode, we implemented our own `vbe_get_mode_info` function. Because the mode is direct color, it allows for a big variety of colors.

To optimize rendering we used double buffering with page flipping. This required implementing `vbe_set_display_start`. This uses a VBE function to change the first pixel position (flipping the page).

Rendering letters is essential to the game, thus we implement a `draw_letter` function. When the first letter is drawn, a xpm font is loaded to the program and is used in all the following calls to the function. The letters' xpm keeps only two colors, so we can decide the letter's color at runtime. The downside of this technique was not being able to use anti-aliasing.

Due to the nature of the game, animations do not really come into play when designing the GUI. Because we wanted to play with this concept, a blinking cursor was added.

### 2.4 Mouse

The mouse is mainly used to interact with the game's UI. The program has an instance of a `mouse_sprite` that indicates the mouse position in the screen. For every mouse interrupt, this position is updated. This

information can be later used by the different buttons to check for collision and change the game state if needed. The program only checks for collisions in the event of a mouse interrupt, to save unnecessary steps.

To work with mouse interrupts the following functions were implemented: `mouse_subscribe_int`, `mouse_unsubscribe_int` and `mouse_set_data_reporting`. The latter is used to enable and disable mouse data reporting. Both functionalities were implemented by us. In the case of an interrupt, `parse_packets` is called to interpret mouse information. When the read is complete, `mouse_get_event` can be used to get an Event with information about the interrupt.

## 2.5 Real-Time Clock (RTC)

In the early development stage, typing speed was being calculated with the usage of the timer. However, this proved to be unreliable, thus making us switch to the RTC. This was first used for periodic interrupts, and made it much easier to keep track of time. Interrupts are generated every 500ms, when `rtc_ih` is called to handle them. Afterwards, if the time elapsed value is needed, a call to `rtc_get_time_elapsed` can be made.

Later on, as scores and leaderboard were being implemented, the RTC acquired a new functionality. It allows the program to get the current date and time to keep track of user score history.

The usage of RTC's interrupts is implemented by `rtc_subscribe_interrupts` and `rtc_unsubscribe_interrupts`. `rtc_ih` should then be called to deal with every interrupt.

### 3 Code organization/structure

The project was organized in an event-driven format and utilized a state machine to separate different sections of the program. The **driver** module holds the low level code responsible for interacting with the I/O. All the different states' code is kept under **states** and **typer** holds the program's main loop and some UI elements.

#### 3.1 Drivers

This module is responsible for the interaction with the I/O devices. The way the information is passed around to higher levels is different for each device and discussed further below.

**Driver** Groups all the functions related to the different drivers. The **driver.c** file contains the function **interrupt\_handler**, which handles and parses every interrupt as needed. To facilitate communication with other parts of the program, **Event** struct is created after the interrupts to accommodate relevant information.

**Keyboard** Manages keyboard interrupts and subscriptions as well as converting makecodes into chars.

**Mouse** Manages mouse interrupts, subscriptions and packet parsing.

**RTC** Manages RTC (real time clock) subscriptions and interrupts. Provides getters for current date and time, as well as functions to subscribe to periodic interrupts and retrieve the elapsed time.

**timer** Functions related to the i8254 Timer. Manages timer interrupts and subscriptions.

**Utils** Common functions for most and less significant bytes and **sys\_inb** function.

**Sprite** Manages letters, digits and symbols XPM's. Facilitates drawing of text.

**Video** Basic drawing functions, like drawing pixels, rectangles, lines as well as letter and digits. Manages screen buffering and VRAM.

The large part of the functions used in this module were re-utilized from the labs. These were developed by the group with pair programming, so the work was distributed equally. As for the others, Luís created the RTC and Guilherme developed the Sprite functions and file.

#### 3.2 States

States sections holds the code for each program's state. States use **proj\_set\_state** function to switch between program states.

**menu** Functions to related menu state.

**game** Functions related to the game state. Contains Game struct and additional functions related to game restart, text and wpm drawing and phrases selection.

**Game Score** Functions related to the **game\_over** state.

**Scoreboard** Functions related to scoreboard state including getter for scoreboard and function to update leaderboard.

The state model organization in our code was developed by Luís. The code behind game logic, game score and scoreboard was developed by André and Luís.



### 3.3 Typer

Typer section contains the core of the program, as well as other interface elements.

**Proj** Core of the project. Defines the main loop. Handles program states. This module defined the core of the project. It contains the main loop and manages program's state. In each step of the loop there are two key components. First, check for interrupts and take necessary actions. Lastly, execute the current state specific code.

**Mouse Sprite** The program keeps a unique mouse sprite responsible of rendering itself and interpret mouse information.

**Button** This module defines a generic use button. Button functionalities like collision detection are implemented, which makes it easy to use buttons across the project.

The main loop structure on Proj was developed by Luís, the button and Mouse Sprite modules were developed by João.

## 4 Implementation details

### 4.1 XPM's

Initially in order to store and display all letters and digits we tried creating a large xpm containing them all. This way we could avoid creating a xpm per char.

Another problem we encountered was the letters color changing during the game and the transparency of the letters. All xpm's have only two colors, a bright pink color (rarely used) for the background and black for the digit itself. In the `draw_letter` function the color we want to display is passed as an argument. This function interprets the xpm color. If the pink color is read, the function doesn't draw the pixel. If it reads the black color, draws the pixel with the color passed in the argument.

### 4.2 Event driven code

For every iteration of the main loop, an **Event** is generated from the latest interrupt. This information is then used to change the game's state according to it. In our project, the life cycle of an event is well defined and every stage is decoupled from the others. This allows for an independent development and update of each stage.

### 4.3 State Machine

Across the program, behavior changes depending on user input. So we could divide all the different sections of the game, a state machine was implemented. In the main loop, `proj_step_state` on every iteration. This is a function pointer that redirects the decision making to the current event. This way, when the state changes, the step function also changes.

### 4.4 Object orientation

Object oriented principles were followed all across the project. To implement it, we recurred to the use of `typedef struct` and separate each class related code through different files, where class methods were implemented. A good example of OOP practice in the project is the `button` class.

### 4.5 Assets

In assets we have the `font`, `XPM`, `phrases` and `scores`. We have several `phrases` and they are chosen randomly in the game, we use `XPM` to visually show characters and the `scores` will appear organized in the leaderboard.

## 5 Conclusions

To sum it up, most of the things we had set up to do at the beginning of the project were implemented. The game turned out to be very appealing and pretty fun to play. Our main achievement in this project was that we managed to make all the devices that were used work together in harmony. We were also able to present a visually simple game with an easy-to-understand interface and at the same time a complete game based on the chosen one.

Due to time limitations, we ended up not implementing the serial port as we had initially wanted. This meant that the multiplayer mode was not created. On another note, we were not able to create sentences with capital letters or dots as this would involve using "Shift" or "Caps Lock" and we were also unable to create more game modes, such as a time trial. That said, these were the most relevant features that ended up not being implemented. So if we could add something in the future, the serial port, capitalization and punctuation would be in our game.