ISTANBUL TECHNICAL UNIVERSITY
Faculty of Computer Science and Informatics

Embedded Software Development on STM32 and ESP32 Microcontrollers
Implementation of the Bluetooth Human Interface Device Profile

INTERNSHIP PROGRAM REPORT

Denıs Iurıe Davıdoglu
150200916

Summer / 2023

Istanbul Technical University

Faculty of Computer Science and Informatics

INTERNSHIP REPORT

Academic Year:	2022/2023
Internship Term:	☒ Summer ☐ Spring ☐ Fall

Student Information

Name Surname:	Denıs Iurıe Davıdoglu
Student ID:	150200916
Department:	Computer Engineering
Program:	100% English
E-Mail:	davidoglu21@itu.edu.tr
Mobile Phone:	+90 (543) 462 6407
Pursuing a Double Major?	☐ Yes (Faculty/Department of DM: _____) ☒ No
In the Graduation Term?	☐ Yes ☒ No
Taking a class at Summer School?	☐ Yes (Number of Courses:_____) ☒ No

Institution Information

Company Name:	ENTES Elektronik Cihazlar İmalat ve Ticaret A.Ş.
Department:	Research & Development
Web Address:	https://www.entes.com.tr/
Postal Address:	Dudullu OSB, 1. Cadde, No:23 34776, Ümraniye, İstanbul

Authorized Person Information

Department:	Research & Development
Title:	Hardware and Software Engineer
Name Surname:	Mehmet Duman
Corporate E-Mail:	mduman@entes.com.tr
Corporate Phone:	+90 216 313 0110

Internship Work Information

Internship Location:	☒ Turkey ☐ Abroad
Internship Start Date:	03.07.2023
Internship End Date:	28.07.2023
Number of Days Worked:	20
During your internship, did you have insurance?	☒ Yes, I was insured by İTÜ. ☐ Yes, I was insured by institution. ☐ No, I did my internship abroad. ☐ No.

Table of Contents

1 INFORMATION ABOUT THE INSTITUTION

ENTES Elektronik was founded in 1980 withing the context of global energy crises, aiming to develop power efficiency solutions[1]. The name "ENTES" comes from Turkish "**EN**düstriyel **TES**isat", meaning "industrial installation". Just after the first five years of their activity, their products achieved wide recognition in the energy sector of Turkey, and started selling on the international market. Other companies emerged from ENTES Elektronik, such as ENTPA and NETA Technologies. Today, the company's headquarters and 12000 m^2 factory are located in Dudullu Organized Industrial Zone, Asian side of Istanbul.

The building is well-organized and has a variety of departments, which can be told from the production perspective. A new product idea starts in the Research & Development department, where hardware engineers, embedded software engineers, test automation engineers, industrial designers, web developers and technicians work together on every aspects of the device. In the R&D's large territory there is a 3D printer, a PCB milling machine, soldering and measurement equipment, boxes with electronic components and microcontrollers, workspaces with computers and dedicated areas for testing the prototypes. Before going to mass production, prototypes are handed to ENTES's own Test and Certification center, where they are further stress tested and examined to meet the international quality and safety standards. Even during the mass-production, the devices are tested both by company's engineers, and by independent specialists in a second center. The manufacture area itself is divided into SMD Assembly, Manual Assembly, Transformer Production etc. In case of a malfunction or any defect related to the products, Customer Support department gets involved, trying to recreate and solve the problem. IT department operates the local internet and maintains company's energy monitoring servers. Marketing department, Sales department and Strategy Management Office collaborate in order to analyze the market's demands, reach out potential customers at home and abroad, regulate prices and keep track of the company development. Unlike outsource software companies, ENTES Elektronik manages the whole production chain with over 400 employees and generates more added value.

ENTES Elektronik's areas of activity are power and energy efficiency, energy management solutions, electrical measurement, compensation, protection and control, monitoring and Internet of Things (IoT)[2]. Their innovative Research and Development department, over the course of more than 40 years, designed and pioneered in many hardware and software projects[1]:

- Turkey's first digital power factor controller
- Turkey's first digital network analyzer (MPR-53)
- World's first 3-phase power factor controller (RG3-12T)
- Turkey's first web-based energy monitoring software (ENTBUS)

ENTES Research & Development is still experimenting with integrating IoT into their devices and they welcomed interns like me to join the development. Human Resources department organized a creative and competitive environment for the interns, making us come up with our own ideas, discuss them with engineers, learn new skills and in the end offer complete solutions, which can be used by the company. In this context, my internship work was a valuable experience both for me and for the institution.

2 INTRODUCTION

I did my internship in embedded software at the Research & Development department, using Bluetooth Low Energy (BLE) feature of an STM32 and later an ESP32 microcontroller. I aimed to achieve data transmission between the microcontroller and any Bluetooth capable master device, mainly for the purpose of saving accumulated data or error messages from the slave device. This becomes important when a device does not have a dedicated display or other means of delivering detailed information to the user. If, for example, a software bug causes a problem on a device with a 7-segment display, an engineer attaches a debug interface such as JTAG, SWD or STLINK to find out the detailed state of the program. However, in a massly-produced device, the debug port is removed for the security reasons, and even an engineer would not be able to read the register values that describe the full state of the processor. Hence, the wireless protocols come into place, and Bluetooth is convenient because it does not require internet connection. As of profile, I chose the Human Interface Device (HID) Profile, because it was more interesting and not as easy as the dedicated Serial Port Profile (SPP). With the microcontroller acting like a standard Bluetooth keyboard, there is no need to install additional drivers or programs for it to work with anything. By aquiring a Bluetooth port, the device becomes capable of more interaction.

3 DESCRIPTION AND ANALYSIS OF THE INTERNSHIP PROJECT

3.1 Developing with STM32 Platform

After discussing with engineers at the company, we decided that I could experiment with writing a Bluetooth application on an STM32 Nucleo board they had recently purchased. In its datasheet, STM32WBACG is described as a 32-bit ARM Cortex-M33 microcontroller with multiple wireless protocols support including Bluetooth [3]. At the moment of internship, the development board was out for only 3 months, and because of the bugs it had, I switched later to an ESP32 board (Section 3.3).

Staring from the basics, I downloaded the recommended development environment STM32CubeIDE, chose the correct board in the project configurator and began examining the microcontroller peripherals. I configured pin PB9 to be in output push-pull mode and tested toggling an LED every 500 miliseconds (Code snippet 1). The code did not work, at least not with the pin PB9. An embedded software engineer responsible for me could not make some pins work in output mode either, although the datasheet stated it was possible. Fortunately, it was not a big issue since I was going to interact with the microcontroller wirelessly.

I continued learning the HAL API using the onboard LEDs, push-buttons, UART communication. When I started to feel confident in writing interrupt service routines for UART and push-buttons, I started reading the Bluetooth Low Energy API manual for my microcontroller[4] and looked at some example Bluetooth projects.

3.2 Defining Bluetooth HID Profile

My STM32WBA microcontroller supported only BLE technology, rather than the classical Bluetooth, which is good because its low power consumption guarantees that my application can run even off a coin cell, making it appliable everywhere. The data is transmitted in small packets and there had been defined many standards for kinds of data exchanged[4].

In the case of my application, data is transmitted via Human Interface Device (HID) Profile. Bluetooth SIG's *HID over GATT Profile Specification* describes the requirements of a HID profile, which is in fact an adaptation of USB HID specification. My device is the one that sends data and is recognized as a wireless keyboard, offering a HID Service. Both Bluetooth and USB share this term, however Bluetooth additionally requires a Battery Service, Device Information Service and optionally Scan Parameters Service. A device can have other optional services as well: Figure 1 [5].

Before setting up Bluetooth, STM32CubeMX configuration software demands some peripherals to be enabled. In the Figure 2, it can be seen that STM32_WPAN peripheral is activated as a result of all other components having been configured. Especially the RF, the radio frequency receiver-transmitter is crucial. I created four services with characteristics and entered their corresponding 16-bit Universally Unique Identifiers (UUID). HID Service is 0x1812 with its Report Map of 0x2A4B; Battery Service is 0x180F and Battery level characteristic is 0x2A19; Device Information Service with PnP ID characteristic is 0x180A and 0x2A50, respectively; and finally Scan Parameters Service has an UUID of 0x1813 [6]. CubeMX recognized the hexadecimal codes properly and shows the full names instead of the codes: Figures 4a-4d.

After laboriously reading and collecting all puzzle pieces from documents, the configuration was complete and STM32CubeMX generated a skeleton code for my project. Unfortunately, the auto-generated code, in which I didn't add anything myself, failed to build with nine errors. I could correct only the parts where UUIDs where written as decimals instead of hexadecimals, by adding **0x** in the front, but this alone didn't make much difference. Failing to understand thousands of generated lines of code and receiving no help from the engineers, I decided start from scratch with another microcontroller.

3.3 Developing with ESP-IDF

Espressif IoT Development Framework (ESP-IDF) is the official tool for programming low-power ESP32 system on a chip microcontrollers. Although their Xtensa instruction set architecture is totally different from the ARM microcontroller I worked with, ESP32 supports Wi-Fi and Bluetooth. In fact, it is an established chip in field of the Internet of Things and has an extensive API [10].

ESP-IDF's Integrated Development Environment is limited in features such as configuring the peripherals with the help of a GUI program. As with STM32, I first wanted to get confortable with setting up the microcontroller. This time, I had an idea of connecting an old PS/2 keyboard to the MCU, so that I can practice reading from the GPIO pins. PS/2 was a primitive serial communication protocol for mouse and keyboard, using two signals: data and clock. Figure 3 summarizes the protocol. Data is read on the clock's rising edge, with a starting bit always 0, eight bits of information, a parity bit for correction and a finish bit, which is always 1.

I set up two pins of the microcontroller accordingly (Code snippet 2). In the pin configuration, I am setting the interrupt mode for the Clock pin as negative edge (line 10), while it is positive in the timing diagram. Also, in the ISR, I perform a bitwise NOT operation on the registered data (line 19). These are due to the peculiarities of the prototype that will be explained in the last Subsection 3.4.

I used the queue component of FreeRTOS [11] to send data from the ISR (line 22). In the main function, I initialize the GPIO, queue, and create a new FreeRTOS task that runs in an infinite loop, after which main is terminated. See Code snippet 3.

As soon as the new task is created, it waits to receive data from queue, which in turn will be filled when an interrupt is fired. In other words, when a key on the keyboard is pressed, the task receives a decoded value of it, and the application can proceed with sending the key via

Bluetooth. The Bluetooth part of the project is overwhelmingly large and I was not allowed to share the code. Instead, I will talk about important constants and function signatures I implemented.

The array below keeps the report map descriptors for specifiying a standard USB Human Interface Device, compiled from official documents [7][8]. As I mentioned in Subsection 3.2, Report Map is common between USB and Bluetooth. There I specify a generic keyboard device with the smallest packet size, achieved by eliminating the ability of pressing multiple keys at the same time.

```
const unsigned char keyboardReportMap[63];
```

USB defines its own scancodes, so ASCII characters must be converted to a correct format. This is achieved in O(1) through a lookup table:

```
static const uint8_t usb_scancodes[128];
```

An ASCII character might be obtained using modifier keys like Shift or even Alt Gr key, in the case of Turkish characters. To save space on the microcontroller's flash, I took advantage of the fact that I only needed to know a boolean value, 0 or 1, with respect to each modifier key. I manually created the tables for modifier keys, then by combining the values into the groups of 16 bits, I squeezed the 128 elements of array into just 8. I also defined a macro which accepts a modifier key array and a **char**, that yields the boolean value.

```
static const uint16_t usb_modifier_shift[8];
static const uint16_t usb_modifier_alt[8];
#define HAS_MODIFIER(a, i) (((a)[(int)(i)/16]&(1<<(15-(int)(i)%16))) != 0)
```

I dedicated a special array for Turkish characters, the length of which is 2 bytes and which would not fit into a reasonably-sized lookup table. Each of the 12 rows contains two char spaces for a Turkish character, one space for null terminator, one USB scancode and two booleans for modifier keys. When a character is being searched in the list, the C standard **strcmp** function compares the character with an entry of the array, and thanks to the null terminator in the third column, it stops and can find a match. The result of the search is an entry with the initial character plus additional information described above.

```
static const char usb_scancodes_special[12][6];
```

This function initializes Bluetooth transmission of a single one byte long character. The format of data is specified in the Report Map, and in this case consists of three bytes: modifier keys byte, reserved byte and regular key byte.

```
  static void hid_send_ascii_char(char);
```

This function accepts a single character longer than one byte and sends it via Bluetooth.

```
static int hid_send_nonascii_char(char*);
```

This function presses the Caps Lock key on the paired device.

```
static void hid_send_capslock();
```

Callback function for managing Bluetooth events such as start, connect and disconnect.

```
static void bt_hidd_event_callback(...);
```

And this function is the ultimate piece of my application. I intended this function to be used as an API for sending any string data via Bluetooth. Data is sent at the average speed of one character every 15 miliseconds, that is 533 bits/s. Assuming that English words are 6.5 letters long, just 10 words can be trasmitted in a second. This limitation arises from the fact that the HID profile was designed for slow-typing humans.

```
static void hid_send_string(char*);
```

3.4 Prototyping

In this last subsection, I only provide details about my work not related to the software. Employed embedded software engineers are not required to design electronic circuits, but at least some prototyping skills are extremely helpful. My motivation was to experience the effect of hardware on software and software on hardware during the product development cycle.

I tested a PS/2 keyboard with an ESP32 microcontroller on a breadboard at first. Figure 6 shows the keyboard plugged into a PS/2 port (right bottom corner), from which four wires go to the breadboard. Because ESP32 works on 3.3V logic level, and the keyboard needs 5V, I had to build a level shifter using MOSFETs. I happened to have used exactly the same circuit schematic and components as in the Figure 5 I found on internet [12]. In my case, the node labled HV in the figure is 3.3V, and INPUT is a wire coming from keyboard, and the OUTPUT connects to the microcontroller. This basic circuit consisting of an an N-channel MOSFET and a 10k pullup resistor assures the microcontroller gets the safe amount of voltage, but it also makes the output inverted, a caveat to be addressed when writing software.

After testing the transistor circuit and my PS/2 driver software, I assembled two of them for data and clock signals and soldered on a perfboard: Figure 7. I was more than happy with this prototype, I could plug any old keyboard and it would suddenly gain Bluetooth capabilities. My code worked well and I was left with only preparing the scancode tables (Subsection 3.3).

Unfortunately, for unknown reasons, the prototype stopped functioning. I got permission to sacrifice the keyboard by directly soldering it to the perfboard. Later, again with the help of technicians, I coated the bottom part with silicone for the connections to be stable (Figure 8, left). After working for a while it failed one more time. I became suspicios of the MOSFETs after checking them with a multimeter, and I replaced them with a rather inelegant resistor voltage divider circuit (Figure 8, right).

Lastly, I got rid of the real keyboard and replaced it with eight switches in a dual inline package, meant to be used for choosing an ASCII character, and a push-button for sending data: Figure 9. It was enough for demonstrating both generic keyboard capabilities and my feature of continuously sending character strings. The codes 0x00-0x79 were mapped to ASCII, while the remainig space between 0x80 and 0xFF was reserved for testing strings, which were sent through the function `static void hid_send_string (char *)` (Described in Subsection 3.3).

4 CONCLUSIONS

This internship gave me the opportunity to get started in embedded systems, see the differences and similarities between STM32 and ESP32 microcontrollers, understand the simultaneous development cycle of hardware and software, and to get a hands-on experience in prototyping, soldering and utilizing a multimeter. I deepened my knowledge of the C programming language by managing and optimizing memory, performing many bitwise operations, reading wired and radio frequency signals from the external environment,

communicating with a real-time operating system and understanding that the language is versatile with both high-level APIs and complete low-level freedom. I exercised creativity to begin and finish my project, but to be fair, I consider it to be just an experiment with no chances of further development, because the device I created suffers from the speed limitations. At least, I realized there are better protocols dedicated to data transmission.

As of the institution which offered me all of these, it was aware of the impossibility of developing a big and useful project for the company under just twenty working days, so they left the interns to freely explore any desired topic. ENTES Elektronik's R&D is trully an engineering powerhouse and I wish I could stay more to get involved into real projects, which take around two years to be completed.

In conclusion, I loved researching, implementing, writing documentation and just to be present in an environment were innovation takes place. I reassured myself that embedded systems is the area to which I want to dedicate my life, because it is concerned with every topic we learn as computer engineering students.

5 REFERENCES

[1] ENTES Elektronik, *About Us*, https://www.entes.eu/about-us/.

[2] ENTES Elektronik (2023), *Product Catalogue*, https://www.entes.eu/uploads/contents/file/en-katalog-2023-64db70c22016a.pdf.

[3] STMicroelectronics (2023), *STM32WBA52xx*, Rev 4.

[4] STMicroelectronics (2023), *Guidelines for Bluetooth® Low Energy stack programming on STM32WB/STM32WBA MCUs*, Rev 7.

[5] Bluetooth SIG, Inc. (2011), *HID over GATT Profile Specification*.

[6] Bluetooth SIG, Inc. (2023), *Assigned Numbers*.

[7] USB Implementers' Forum (2001), *Device Class Definition for Human Interface Devices (HID)*, Version 1.11.

[8] USB Implementers' Forum (2022), *HID Usage Tables for Universal Serial Bus (USB)*, Version 1.3.

[9] Adam Chapweske (2003), *The PS/2 Mouse/Keyboard Protocol*.

[10] Espressif Systems (Shanghai) Co., Ltd (2023), *ESP-IDF API Reference*.

[11] Amazon.com, Inc. (2017), *Reference Manual for FreeRTOS*, version 10.0.0, issue 1.

[12] Open Impulse, *Unidirectional voltage level translators*, https://www.openimpulse.com/blog/2012/06/unidirectional-voltage-level-translators/

6 APPENDIX

6.1 Figures

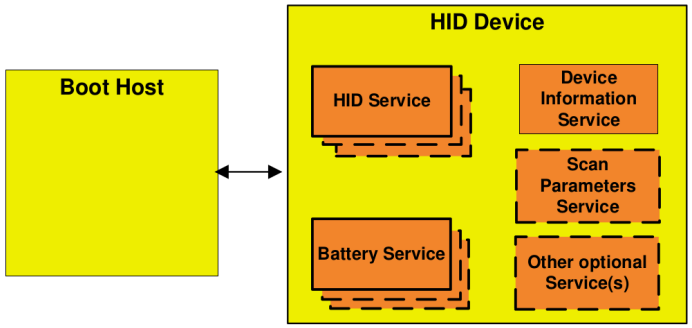


Figure 1: Bluetooth HID device services

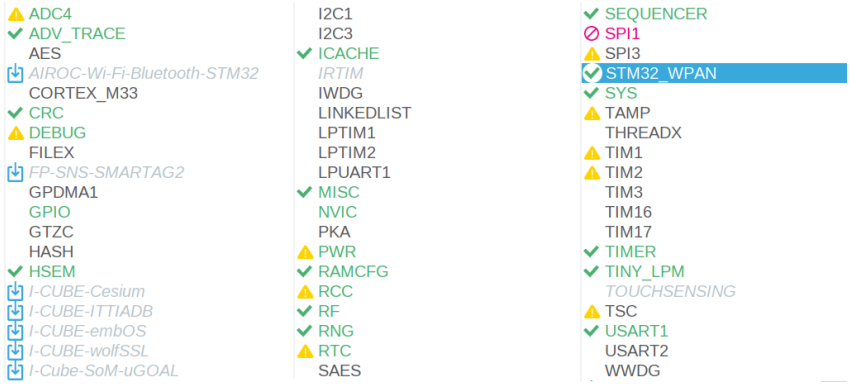


Figure 2: Minimal configuration of peripherals for using Bluetooth on an STM32WBA52CG

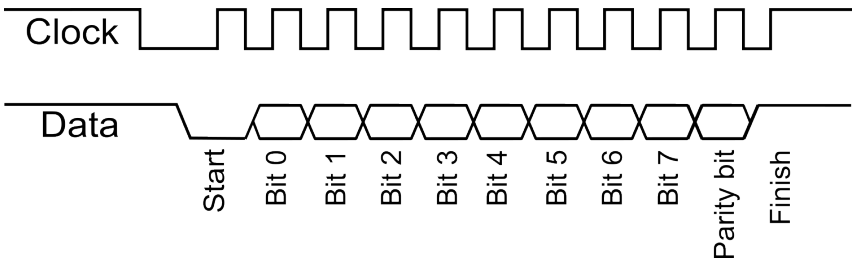
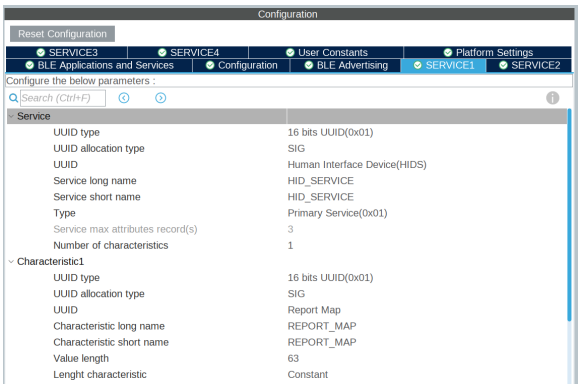
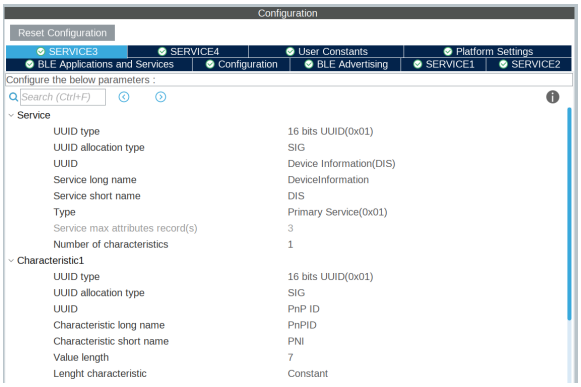


Figure 3: PS/2 serial protocol timing [9]



(a) HID service

(b) Battery service



(c) Device information service

(d) Scan parameters service

Figure 4: STM32 WPAN configuration

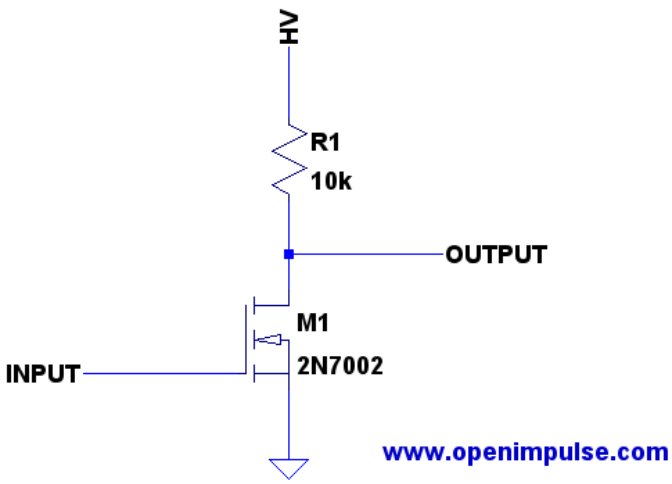


Figure 5: Level shifter using N-channel MOSFET and a pullup resistor

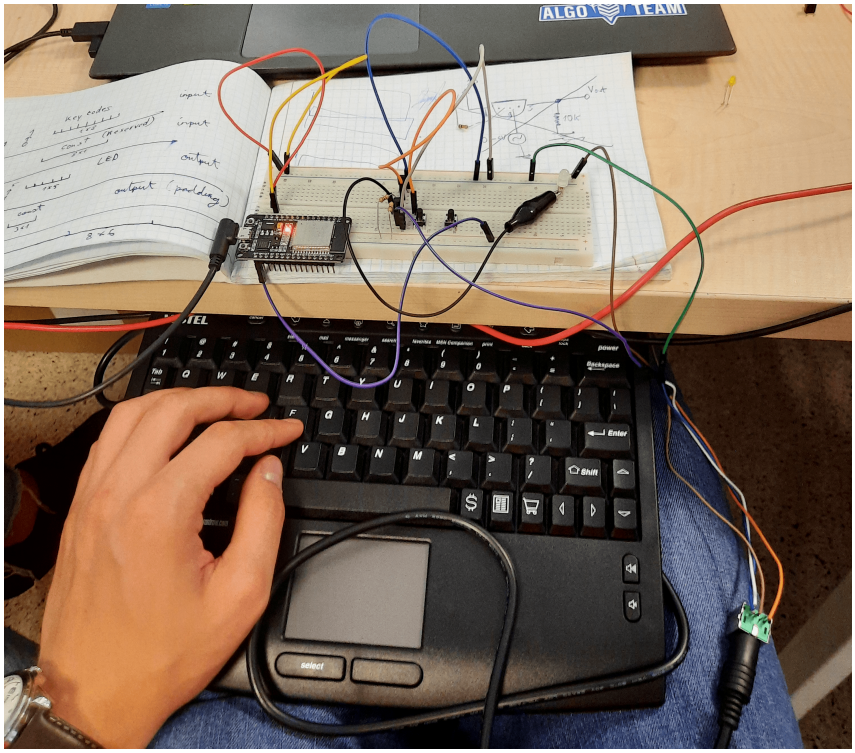
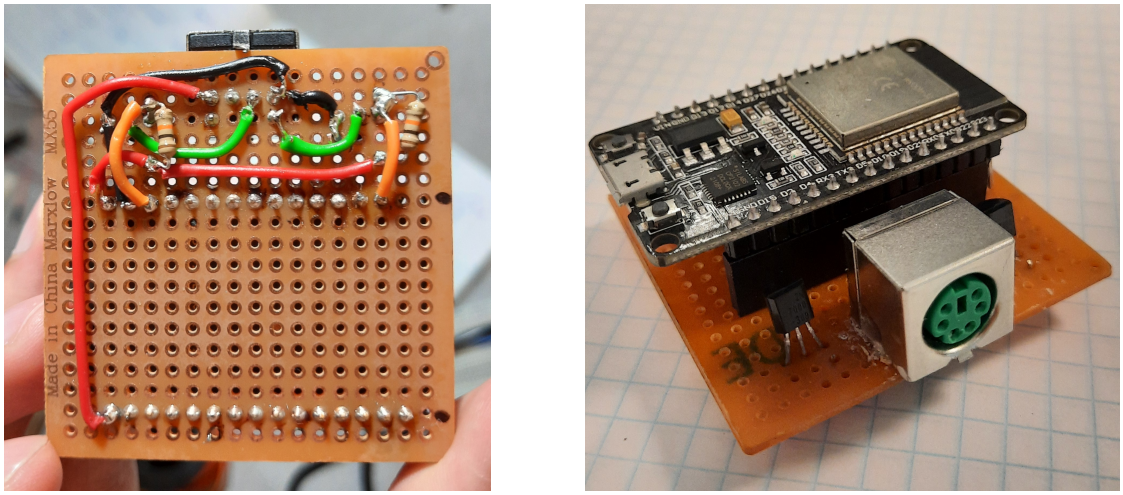


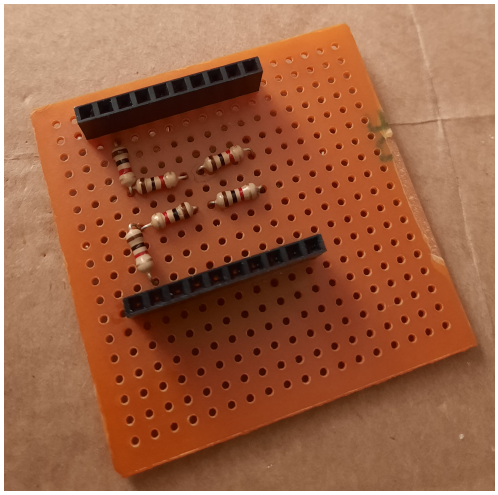
Figure 6: Breadboard prototype



(a) (b)

Figure 7: First soldered prototype

(a)



(b)

Figure 8: Attempts to fix the problems

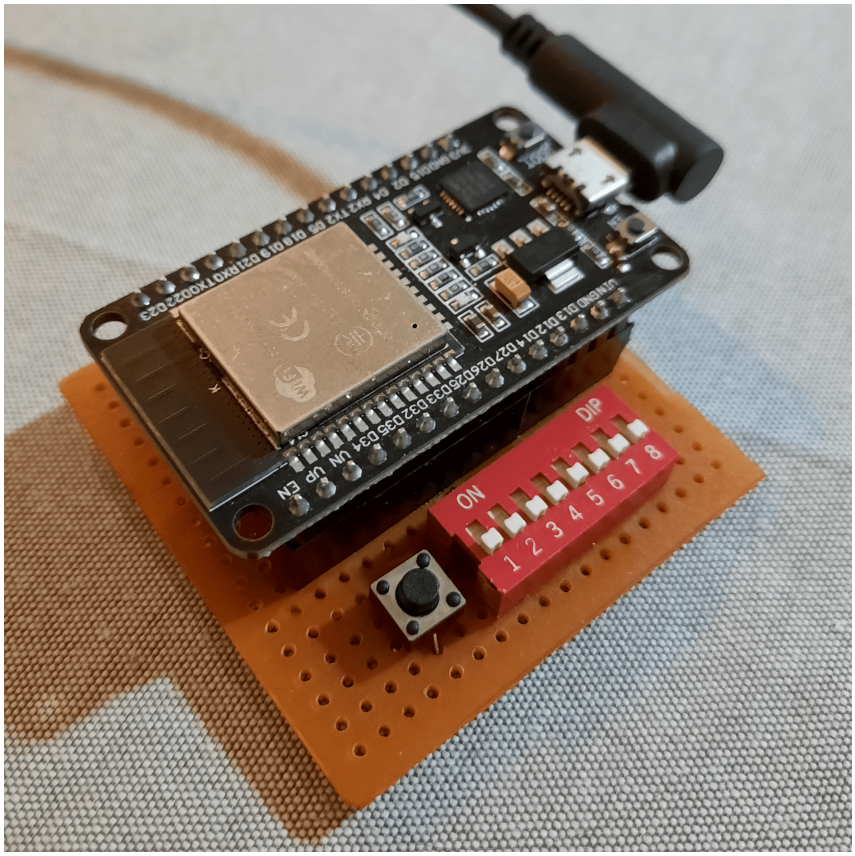


Figure 9: Final prototype

6.2 Code snippets

```
while (1) {
  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_9);
  HAL_Delay(500);
}
```

Code snippet 1: LED blink test

```
#define PS2_CLK_PIN 4
#define PS2_DATA_PIN 14

gpio_config_t ps2_pin_config = {
  (uint64_t)  ((1UL << PS2_DATA_PIN) |
              (1UL << PS2_CLK_PIN)), // pin_bit_mask
  (gpio_mode_t) GPIO_MODE_INPUT, // mode
  (gpio_pullup_t) 0, // pull_up_en
  (gpio_pulldown_t) 0, // pull_down_en
  (gpio_int_type_t) GPIO_INTR_NEGEDGE// intr_type
};

void ps2_interrupt_respond(void* arg) {
  static int ps2_bits_count = 0;
  static int ps2_data = 0;
  ps2_data |= (gpio_get_level(PS2_DATA_PIN) << ps2_bits_count);
  ps2_bits_count++;
  if (ps2_bits_count >= 11) {
    ps2_data = ~ps2_data;
    ps2_data &= 0x1FE;
    ps2_data >>= 1;
    xQueueSendFromISR(ps2_key_queue, &ps2_data, NULL);
    ps2_bits_count = 0;
    ps2_data = 0;
  }
}
```

Code snippet 2: Pin configuration and interrupt service routine

```
static QueueHandle_t ps2_key_queue = NULL;

void app_main(void) {
  gpio_config(&ps2_pin_config);
  gpio_set_intr_type(PS2_DATA_PIN, GPIO_INTR_DISABLE);
  gpio_install_isr_service(ESP_INTR_FLAG_LEVEL1);
  gpio_isr_handler_add(PS2_CLK_PIN, ps2_interrupt_respond, NULL);

  ps2_key_queue = xQueueCreate(3, sizeof(int));
  xTaskCreate(ps2_print, "PS/2 PRINT", 2048, NULL, 1, NULL);
  printf("App started\n");
}
```

Code snippet 3: Main function