Laboratory of Optoelectronics and Photonics
Wrocław University of Science and Technology
Chair of Electronic and Photonic Metrology

Optical text transmission

Optoelectronics - project

Group Number: 10

Name	Surname	Album number
Jakub	Kaszowski	263544
Mikołaj	Pastucha	264353
Fryderyk	Leszczyński	263553

Wrocław 2024

Wrocław University
of Science and Technology

Contents

1 Introduction

The purpose of this project was to build a device capable of transmitting text using light. It should provide a Physical Layer for higher level arbitrary protocols. Such a device could be used in a variety of applications, such as remote control of any device that has a serial port. Any encoding could be used. Some of them could involve checksums and retransmission. Our device will however send a message in one direction, so even if the error is caught, there is no means to request a retransmission. Having in mind this assumption, we decided to use a morse code for encoding the information. This report elucidates a project that employs Morse code for intercommunication between two Printed Circuit Board (PCB) assemblies. The communication is facilitated by a light-emitting diode (LED) transmitter and an LED receiver. The messages, originating from a computer terminal, are transmuted into Morse code, transmitted, received, and then demodulated back into alphanumeric characters for display.

This report is a technical documentation of the project. It describes the initial assumptions, selected solutions and final result.

2 Theoretical Introduction

The project is situated within the realm of digital electronics and optical communication. It leverages the principle of optical transmission and reception of signals. The Morse code, a method of encoding text characters as standardized sequences of two different signal durations, is used as the medium of communication.

The hardware components employed include LEDs and STM32 microprocessors. The system's performance is contingent on parameters such as signal strength, transmission distance, and ambient light interference. The mathematical model for the Morse code encoding and decoding process will be presented. Since the aim of the project is transmitting information using light, elements capable of transmitting and receiving light are needed. They will be core parts of respectively a transmitter and receiver.

2.1 Transmitter

From available sources of light, we have decided to use a standard infrared diode used in any TV remote control. It can generate a light with wavelength of 940nm. Such diodes have many advantages and let us mitigate problems that we could have when using more sophisticated ligth source. One of our alternatives were lasers that can be found in our lab. The provide more directed and more intense light, but require power supply and the modulation of such light would be complex. The another factor for such choice is price and availability. The used infrared led was purchased at local electronic shop and is fairly inexpensive - costs less than 1$.

Figure 1: Infrared LED

Figure 2: Photodiode

2.2 Receiver

The crucial element of the setup is the sensing element measuring the light sent by transmitter. It should be chosen in such a way to maximise the useful signal and minimise the noise. A most common part used to measure light intensity in a hobbyist projects is a fotoresistor. Such element changes its resistance with the change of light conditions. It can't be used in our device since it would be too prone to the interferences, as it has a wide sensing specturm. Since we decided to use a standard infrared diode, we chose to use a phototransistor made for the same light

3 Assumptions

3.1 Functional Assumptions

The system is architected to transmit alphanumeric messages from one terminal to another using Morse code as the communication protocol. The user inputs a message into the transmitting terminal. This message is then encoded into Morse code and transmitted via the LED transmitter. The LED receiver picks up the transmitted Morse code signal, decodes it back into alphanumeric characters, and displays it on the terminal screen.

The transmitter should be able to broadcast a message encoded as a morse code. It should take the input data from a serial port that should be easy to connect to a standard PC without specialized converter. The device should take a text message, turn in into a morse code and transmit.

The receiver should listen continuously wait for any message to come. When it arrives, it should decode it into a useful form - back into a human readable text. In case a timing error happens, the user should be notified. Both of them should be easy to carry. Moreover they should be powered from a USB port with maximum current of 500mA.

The functional requirements can be summarized as follows:

- Transmit alphanumeric messages from one terminal to another using light
- Encode a message with morse code.
- Decode morse code into a useful form.
- Detect idle line or incorrect timings.
- Provide USB interface for sending/receiving data.
- Does not need external power supply.

3.2 Design Assumptions

The system is implemented on the Arduino Black Pill platform, powered by STM32 microprocessors. The firmware, written in C, provides low-level control over the hardware peripherals and system resources. The design assumes a certain level of proficiency in digital electronics, embedded systems, and C programming. The firmware handles tasks such as Morse code encoding/decoding, signal transmission/reception, and user interface management. We want our design to be easy to make at home. Thus, inexpensive off the shelf components are used. Also, the setup does not have a custom PCB - this makes it easy to reproduce for everyone, even using a breadboard. The case is optional for the device operation. In our case it will be 3D printed. Our design should meet the following criteria:

- Off the shelf components are used.
- The costs should be minimized.
- The source code should be open source.
- The case is 3D printed.

4 Description of the Hardware Part

This section presents how the physical device was built. There are two modules: transmitter and receiver. The electronics part for both is described in separate sections. Since they have the same dimensions, they both used the same case which is described at the end.

In order to make the transmission work, the transmitter and receiver need to be in correct position, turned towards each other with their optical parts.

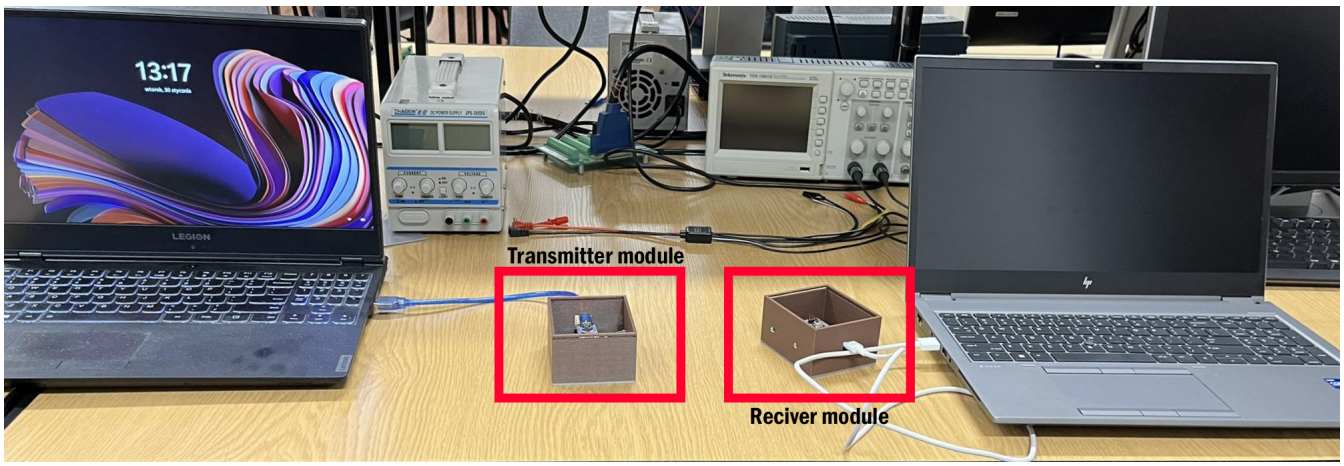


Figure 3: Overview of system during the operation.

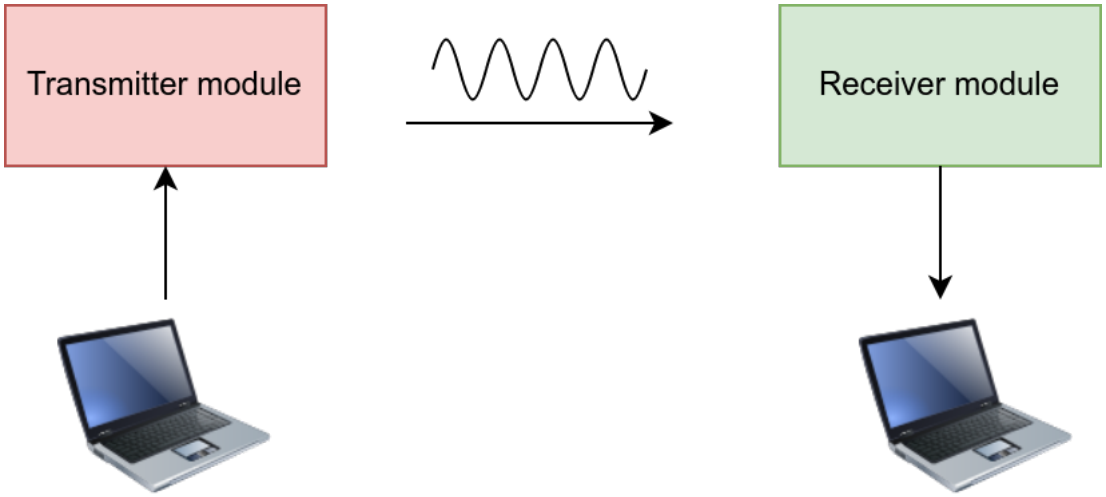


Figure 4: Overview of system during the operation - block diagram.

4.1 Transmitter module

The main componenet of the transmitter module are:

- BlackPill development board equipped with USB connetor, capable of communication with PC.
- Sensing circuit - photodiode for 940nm wavelength.
- LED indicator.

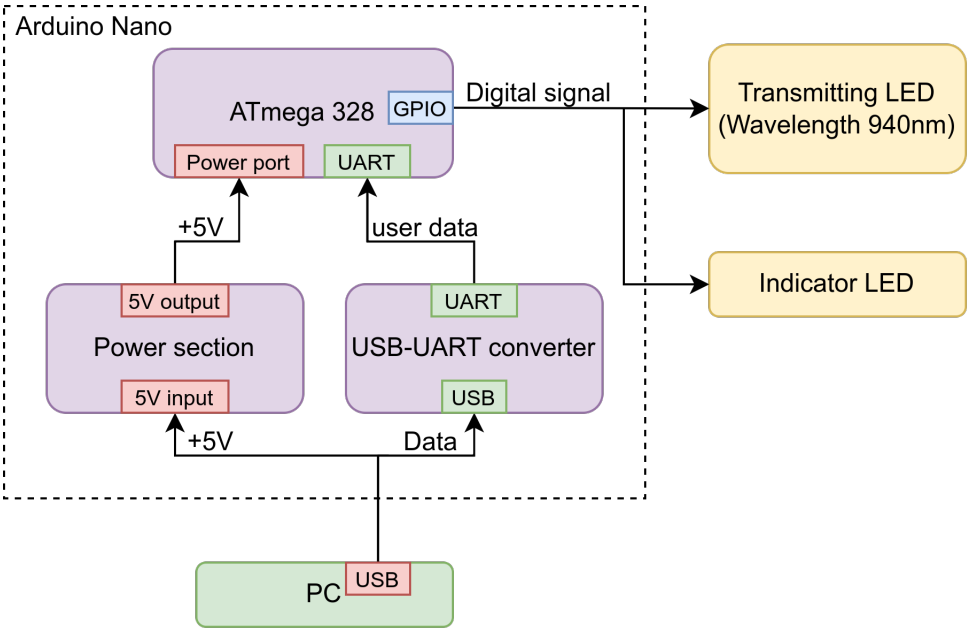


Figure 5: Transmitter module overview.

Full schematic is available in Appendix B.

4.2 Receiver module

The main componenet of the receiver module are:

- Arduino Nano development board equipped with USB connetor, capable of communication with PC.
- Transmitting circuit - infrared LED for 940nm wavelength.
- LED indicator.

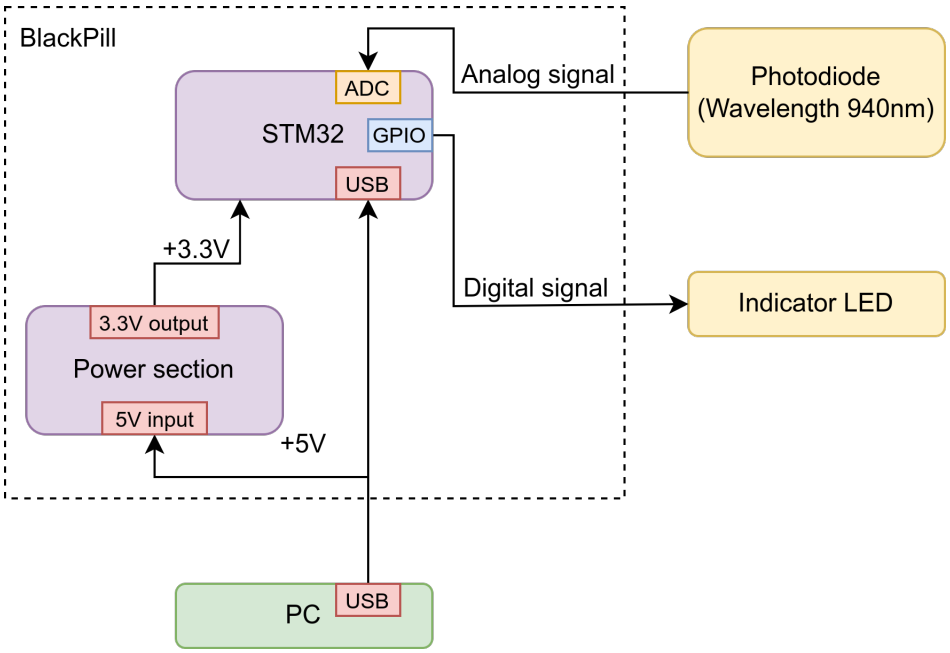


Figure 6: Receiver module overview.

Full schematic is available in Appendix B.

4.3 Case

The cover part was designed in Autodesk Inventor. It's two identical boxes with 4 square cubes in which there are six-sided holes for metal nuts of size M3. These nuts are meant to mount the pcb to the case. Thanks to this, our two devices will be on the same height and we prevent any unwanted movements. In each box, there are three holes - two for our transmitter/receiver diode (with separate indicator). The third one is for USB type-C cable to get power and signal from the computer. It's closed by slide-in plate. The box was printed with SLA, using 3D printer avaialbe at the robotics class.

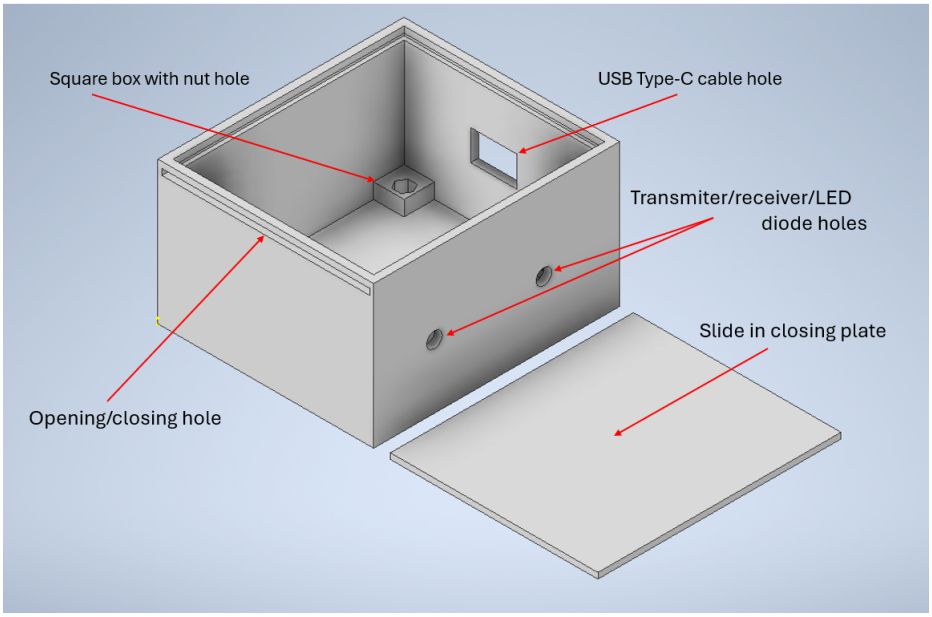


Figure 7: The case rendered in Autodesk Inventor. Most importants parts are shown.

Full technical drawing is available in Appendix A.

5 Software

5.1 Receiver module

The software on the microcontroller is responsible for following tasks:

1. Measure current signal level using builtin ADC.
2. Quantize an analog signal into binary voltage levels.
3. Continuously run converting algorightm.
4. Recognize errors.

5.1.1 MCU selection

The selected MCU is STM32F411 that can be found onboard breakout module Blackpill. It contains USB periperial enabling us to communicate with serial emulators without a dedicated USB to UART converter. It also has a builtin ADC with configurable resolution (in our case it is set to 12 bits) that runs in a continuous mod) that runs in a continuous mode. Additionally, it is equipped with a Floating Point Unint which significantly increases performance while performing math operations. The alternative for STM microcontrollers could be Arduino. It is also used for simliar projects, however it comes with much lower capabilites at the same price - there is no builtin serial to USB converter. Moreover, the debugging is supported only on STM, which proved crucial during implementation.

5.1.2 Toolchain

The STM32 can be programmed using both C and C++. The ARM company provides fork of gcc for the Cortex M4 cores that are used in the Blackpill, which use C++ 17 standard. We have decided to use STM32CubeIDE. It is an Eclipse based IDE that comes with proper compiler and visual configurator of the peripherals and clock.

With the complexity of an Cortex M4 core, the manual configuration of every peripheral and clock using registers is pretty tedious and error prone. In order to write a portable code, the Hardware Abstraction Layer libraries from the manufacturer were used. They significantly speed up the process of writing hardware drivers and lets us focus on the algorithm and achieving requirements. Moreover, the graphical tool lets us generate appropriate code for setting proper settings of peripherals and clocks. A very useful feature of all arm microcontrollers is SysTick interrupt [2]. It is called each millisecond and increments a time variable. It enables us to measure time easily, without setting up a dedicated timer.

5.1.3 Analog to Digital Conversion

The selected microcontroller has a powerful successive approximation analog-to-digital converter onboard, capable of achieving 12 bits resolution. It can run in a single or continuous mode and optionally generate a DMA request to offload main CPU.

In our case we use a single measurement mode. It means that each time that we want to get the value of analog voltage in digital form, we have to request it and wait for the event. It is suitable in our case because of low timing requirements. Our algorithm should work in such a way that a human operator could see the transmission and understand it. The timings of the ADC converter are shown below:

By setting the SWSTART bit, we can trigger a start a conversion process. Once it finishes, the End Of Conversion (EOC) bit is set. This process takes usually 15 clock cycles of ADC, which in our case is a few MHz. Thanks to such a speed of the conversion, we are able to probe the channel with frequency well above the frequency of changes of the signal during normal operation. Thanks to that, the shannon-nyquist theorem is not violated in our case.

5.1.4 Converting algorithm

The converting algorithm uses a preconfigured timing constraints that include times of: dash, dot, space between two signs, space between two letters, space between two words, timeout.

The main flow of the algorithm consists of following steps:

1. Calibrate reference value.
2. Wait for start of transmission.
3. Get chain of dash, dot and space.
4. Analyze the chain and present human readable form.

The schematic of the algorithm is visualized on fig.8. The full algorithm is available in Appendix C.1 .

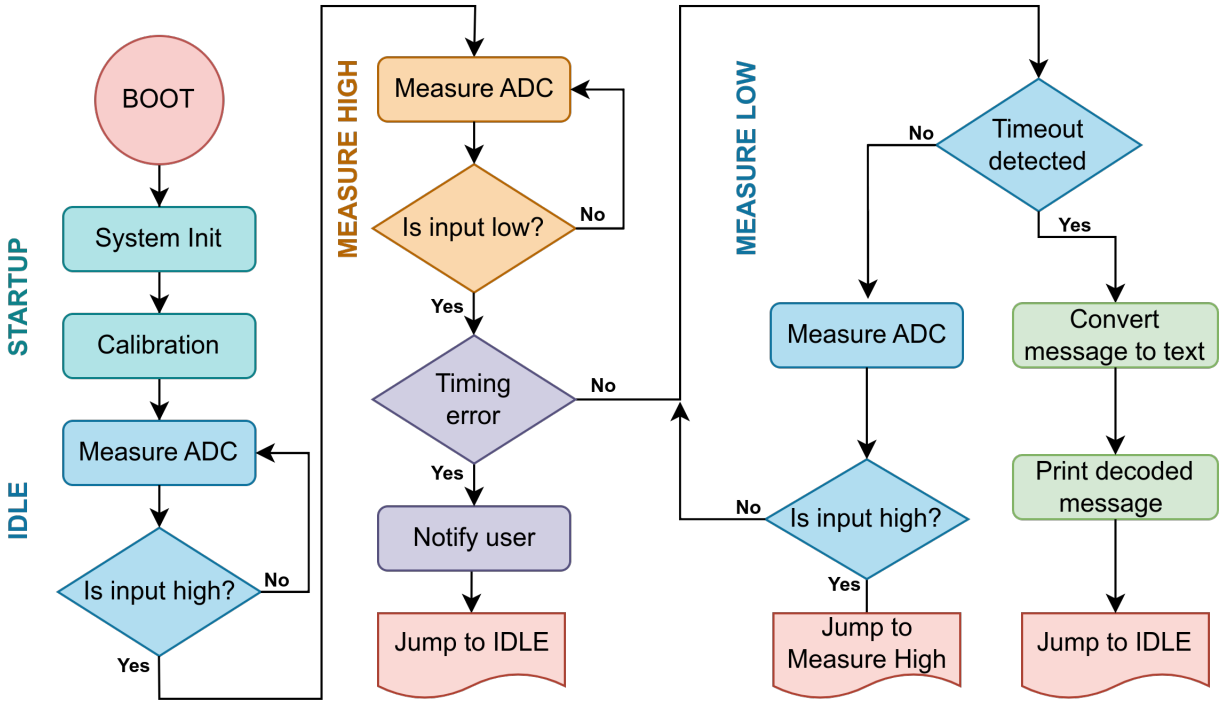


Figure 8: Algorithm of the receiver.

At the beginning, the code performs calibration procedure. It should establish a reference value of an environmental noise. When finished, the algorithm enters idle state. Here it waits for a transmission to start. It continuously measures the input value and waits for the high state.

Once the transmission has started, it can measure the length of the high state. Next step is to compare measured value to predefined one and classify signal as a dash, dot or timing error. Following the high state, the low state is measured. If it is too long, the timeout is detected. The user is notified of an error and the algorithm returns to the idle state.

The other possiblity is that the low state is measured to be a pause between signs, letters or words. If, it is noted as such and the code goes back to measuring high state to get the next sign. The listing of a full code can be found at the end of this report.

5.1.5 Error detection

The first method error detection is performed by comparing measured times to the expected values. The second method is analysis of the chain of dots and dashes. If it happens that the combination does not match any letter in the standard, a question mark is inserted.

```
char getAscii(char *str) {
  for (int i = 0; i < 26; i++) {
    if (strcmp(str, morseCode[i]) == 0) {
      return i + 'a';
    }
  }
  return '?';
}
```

Figure 9: Code responsible for morse code decoding. Unknown sequence return question mark.

5.2 Transmitter module

The software on the microcontroller is responsible for following tasks:

1. Get data to send from the user.
2. Encode the data using morse code.
3. Transmit the data using IR LED.

5.2.1 MCU selection

As the requirements for transmitter are much simpler than for receiver, an ATmega328 was selected. It is a cheaper MCU than STM32 and can be found in popular Arduino modules. It does not have debugging capabilities but at least have a support from Arduino framework, enabling fasting prototyping.

5.2.2 Algorithm

The part of software that sends the data is much simpler than the software that receives the data. The tasks of the software is to get the data that user wants to send in a form of text, convert the text into a morsed code and generate an appropriate light using connected IR diode. The algorithm itself is much simpler than in the case of receiver. The code waits for data available in serial port buffer. When its available, it gets converted by using each character in the buffer as a key to a lookup table. The decoded code is immediately transmitted using predefined timing constraints. Full listing is available in Appendix C.2 .

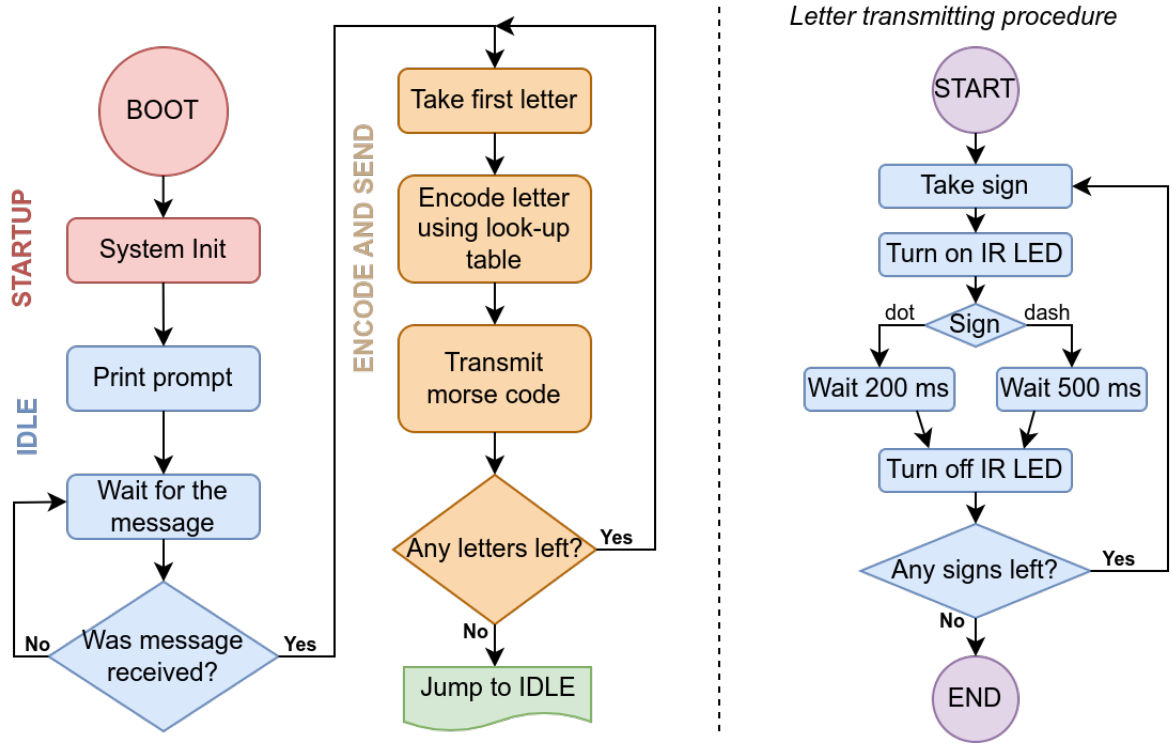


Figure 10: Transmitter module algorithm.

6 Start-up, Calibration

6.1 Receiver module

During the start-up, the calibration procedure is needed. It is impossible to know in what ligh conditions the device will operate, so the reference value of light intensity is not hardcoded. Instead, a short procedure is executed at each startup. It does not take more than 3 seconds which is usually negligible since the operator won't even have time to open serial connection in this time.

6.2 Transmitter module

The calibration of transmitter is not needed. It does not sample any data thus the reference value is not needed.

7 Test Measurements

The measurements were conducted to understand how lightning conditions constrain operation of the device. The most important part of the setup was an oscilloscope, measuring transmitter output and receiver input at the same time. This allowed us to directly compare the signal sent and received. Any kind of noise or change of amplitude would be visible immediately.

7.1 Normal operation

Firstly, the signals during normal operation were measured. The devices were put in front of each other with a distance of 10cm between them. Then, a message was sent. The following signals were measured: As we can see, the received signal very well resembles the original one. The normal voltage at the analog

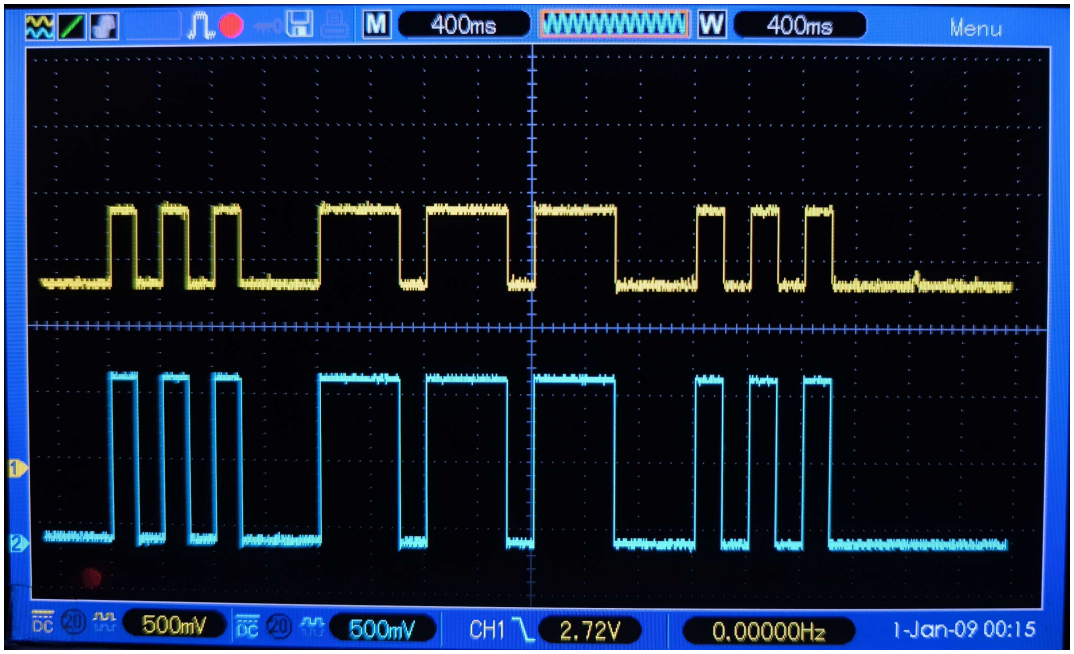


Figure 11: Signal levels during normal device operation. Blue signal is voltage on transmitting LED, yellow is voltage level at the analog input of the receiver.

output is 1.25V, rising to almost 2V when the signal from Arduino is received. The signal to noise ratio is very good, we can barely see any distortions to the signal.

7.2 Extreme light conditions

In the second case, the setup was put in direct ligh of a desk lamp, equipped with a 6.3W LED lightbulb. In this case, a signal to noise ratio is very poor. Clearly the increased light from lightbulb shifts the

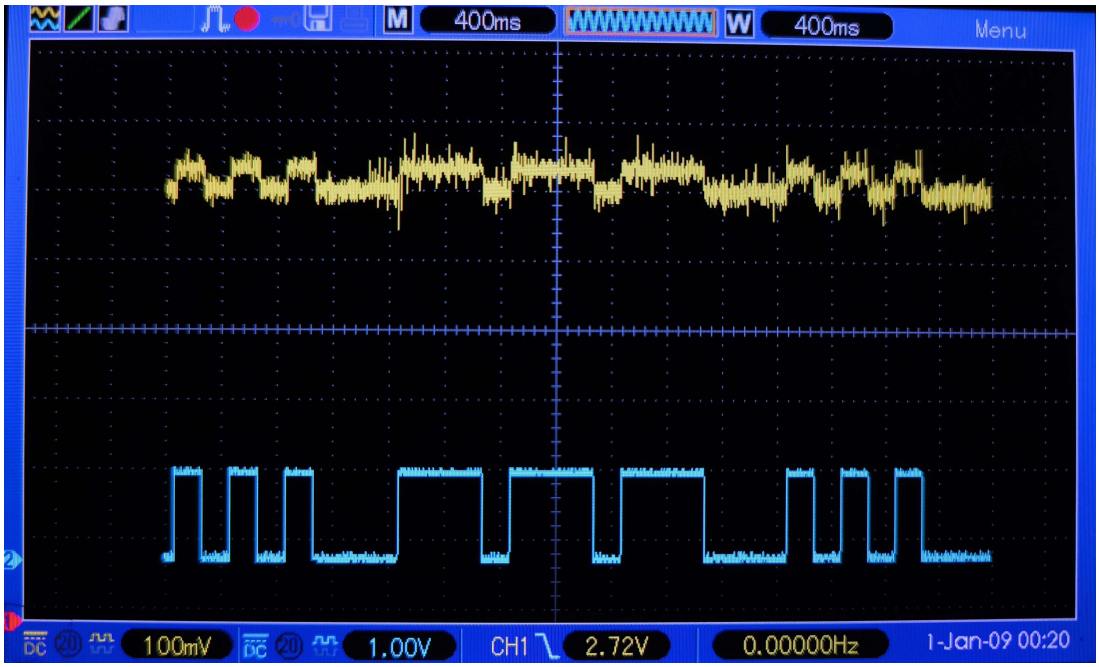


Figure 12: Signal levels during device operation in extreme light conditions. Blue signal is voltage on transmitting LED, yellow is voltage level at the analog input of the receiver.

reference value so high, that when the transmitter diode is turned on, it's barely visible. The implemented algorithm recognizes whether a certain sample is a 0 or 1 based on two values: reference and threshold. Looking at these signals, we can conclude that in harsh lightning conditions the transmission is not possible due to SNR and issues with selecting proper threshold.

7.3 Maximum distance

We have also tried to find the maximum distance for which the transmission works. In general, the value measured when transmitting LED is on decreases as the devices are moved apart from each other. This means that in order to perform the communication at long distances, the threshold has to be adjusted. However, its value highly depends on the lighthing conditions. Therefore, we can only conclude that for higher distances the threshold should be lowered. During our tests, the maximum distance between transmitter and receiver was 50cm for the transmission to occur.

8 User Manual

Step 1: Powering On the Devices

a). Transmitter (Sender):

Start the Arduino with the transmitter code, and optionally connect to the first PC. Ensure that the LED on the transmitter is blinking, indicating readiness for transmission.

b). Receiver (Recipient):

Power on the Blackpill with the receiver code and connect to the second PC. Ensure that the receiver is ready (e.g., an LED indicating readiness to receive signals).

Step 2: Entering Messages

1. Transmitter (Sender):

   Enter the message in a natural way using a keyboard or another input device.

2. Transmitter (Sender - Continued):

   Press the button to send the message.

Step 3: Receiving and Displaying Messages

1. Receiver (Recipient):

   Point the receiver towards the transmitter. Observe the LED on the receiver, which should be blinking or lit, indicating the reception of signals.

2. Receiver (Recipient - Continued):

   Wait for the completion of the transmission. View the received messages, which are displayed on the computer.

Step 4: Powering Off the Device

1. Transmitter (Sender):

   Finish using the device by turning off the transmitter.

2. Receiver (Recipient): Finish using the device by turning off the receiver.

Tips and Notes:

- Distance: Maintain a moderate distance between the transmitter and receiver for effective transmission.
- Ambient Light: Avoid strong ambient light that may interfere with the transmission.
- Readiness to Receive: Ensure the receiver is ready to receive before starting the transmission.
- Safety: Avoid direct exposure to LED lights, especially with strong light sources.

9 Summary

As a result of the project, two separate devices were built. Both of them fullfill the initial assumptions. The transmitter gets data from serial port emulated in USB, encodes it as a morse code and transmits. The receiver perfroms calibration, classification of the current state of the line and performs algorithm for decoding the morse code into the text message. It also detects and notifies user about timing errors.

The biggest challenge was definitely implementing the receiving algorithm and calibration. Other than that, we have learned how to make a device with a full documentation from scratch and how to work as a team.

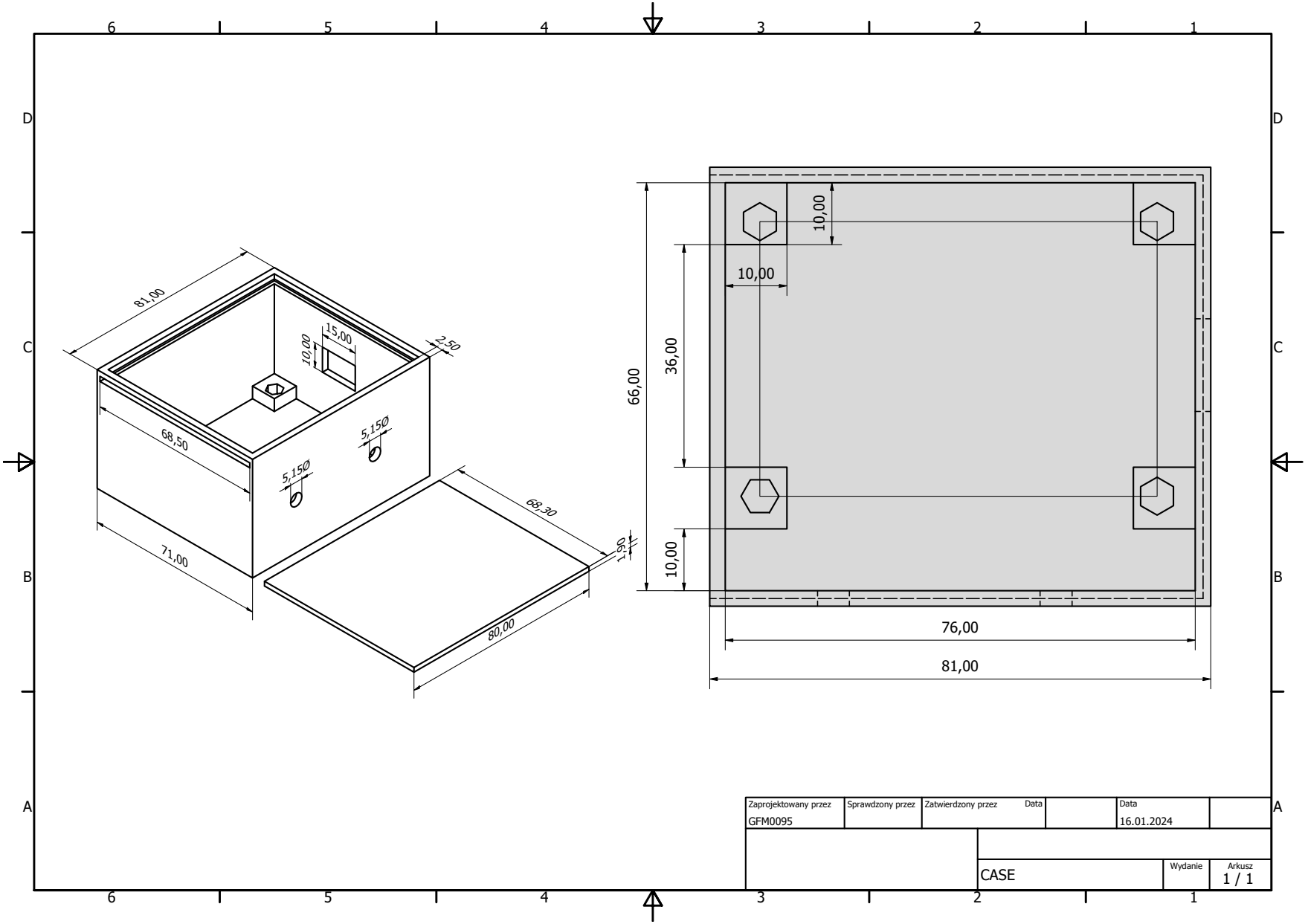
In the future, the project could be extended by introducing calibration of the transmitter/receiver pair to support custom timings. Also, an algorithm of receiver could be explored to provide ability to decode a morse code without predefined timings, judging purely by relative length of pulses.

References

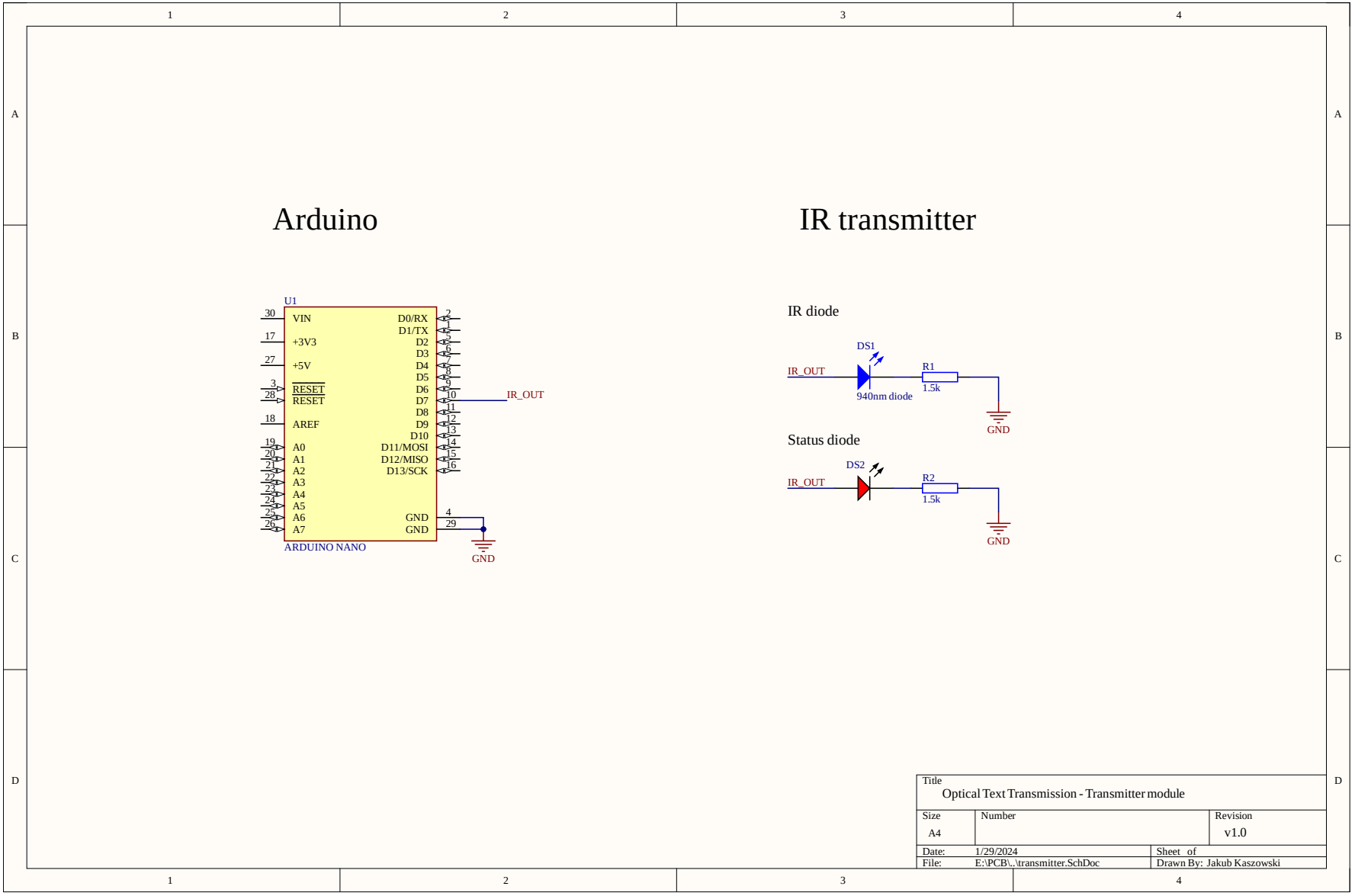
[1] STM32F412 Reference Manual - RM0402 (2020), *ST Microelectronics*

[2] Arduino Core Refernce at `https://www.arduino.cc/reference/en/`

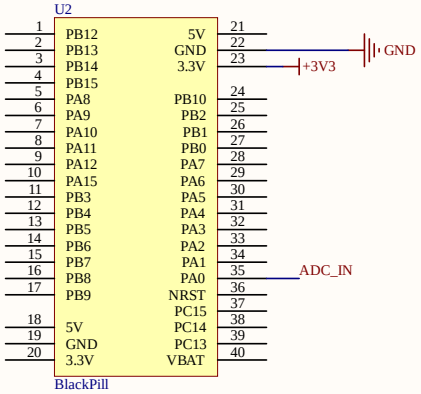
Appendix A Technical drawing of case



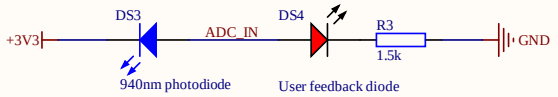
Appendix B Electrical schematic



BlackPill



IR receiver



Title: Optical Text Transmission - Receiver module		
Size: A4	Number:	Revision: v1.0
Date: 1/29/2024		Sheet of
File: E:\PCB\..\receiver.SchDoc		Drawn By: Jakub Kaszowski

Appendix C Full code listing

C.1 Receiver module

```
/* USER CODE BEGIN Header */
// ##############################################################################
//   Project: Optical Transmission of Text
//   Module:  Receiver
//   Author:  Jakub Kaszowski
//   Date:    09.01.2024
//   MCU:     BlackPill (STM32)
// ##############################################################################
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "usb_device.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "usbd_cdc_if.h"
#include <stdarg.h>
#include <string.h>

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
const char *morseCode[] = {
    ".-",   // A
    "-...", // B
    "-.-.", // C
    "-..",  // D
    ".",    // E
    "..-.", // F
    "--.",  // G
    "....", // H
    "..",   // I
    ".---", // J
    "-.-",  // K
    ".-..", // L
    "--",   // M
    "-.",   // N
    "---",  // O
    ".--.", // P
    "--.-", // Q
    ".-.",  // R
    "...",  // S
    "-",    // T
    "..-",  // U
    "...-", // V
    ".--",  // W
    "-..-", // X
    "-.--", // Y
    "--.."  // Z
};

```

```c
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;

/* USER CODE BEGIN PV */
// Data used by USB driver
extern int usb_rx_flag;
extern char usb_rx_buffer[];

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

// Measure current ADC value in range 0-4096
int getADCvalue() {
  int PomiarADC;
  if (HAL_ADC_PollForConversion(&hadc1, 10) == HAL_OK) {
    PomiarADC = HAL_ADC_GetValue(&hadc1);
    HAL_ADC_Start(&hadc1);
    return PomiarADC;
  }
  return 0;
}

// Send message to the user via USB
void log(const char *msg) {
  char buffer[256];
  snprintf(buffer, 256, "%s\r\n", msg);
  CDC_Transmit_FS(buffer, strlen(buffer));
}

void log_dec(const char *msg, int val) {
  char buffer[256];
  snprintf(buffer, 256, "%s, val=%d\r\n", msg, val);
  CDC_Transmit_FS(buffer, strlen(buffer));
}

// Convert a strign of dashes and dots to letter
char getAscii(char *str) {
  for (int i = 0; i < 26; i++) {
    if (strcmp(str, morseCode[i]) == 0) {
      return i + 'a';
    }
  }
  return ' ';
}
```

```
119
120 char human_readable[128]; // place to store human readable text after the
121                           // conversion
122 char *ptr;                // temporary value for conversion
123
124 void convertWord(char *word) {
125   char korektor[] = " ";
126   char *schowek;
127   schowek = strtok(word, korektor);
128   *(ptr++) = getAscii(schowek);
129   while ((schowek = strtok(NULL, korektor)) != NULL) {
130     *(ptr++) = getAscii(schowek);
131   }
132   *(ptr++) = ' ';
133 }
134
135 void convertBuffer(char *stack) {
136   char korektor[] = " ";
137   char *schowek;
138   ptr = human_readable;
139   schowek = strtok(stack, korektor);
140   *(ptr++) = getAscii(schowek);
141   while ((schowek = strtok(NULL, korektor)) != NULL) {
142     *(ptr++) = getAscii(schowek);
143   }
144   *(ptr++) = 0;
145 }
146
147 // possible states of main FSM
148 typedef enum {
149   NORMAL,
150   MEASURING_HIGH,
151   MEASURING_LOW,
152   FINISH,
153   CALIBRATION
154 } TRANSMISSION_STATE;
155
156 // Helper variables for calibration
157 int calibration_counter = 0;
158 int calibration_buffer = 0;
159 const int calibration_samples = 8;
160
161 // Stores calibrated value of reference signal
162 int reference_value = 0;
163
164 // Stores the threshold over refrence value to classify signal as logic HIGH
165 int threshhold = 100;
166
167 // Timing constraints
168 const int DOT_DURATION = 200;
169 const int DASH_DURATION = 600;
170 const int PAUSE_DURATION = 200;
171 const int MIDWORDPAUSE_DURATION = 600;
172 const int BETWEENWORDPAUSE_DURATION = 2200;
173 const int ERR = 50;
174 const int ERR_PAUSE = 300;
175 const int PAUSE_DURATION_LOW = (PAUSE_DURATION - ERR_PAUSE);
176 const int PAUSE_DURATION_HIGH = (PAUSE_DURATION + ERR_PAUSE);
177 const int DOT_DURATION_LOW = (DOT_DURATION - ERR_PAUSE);
178 const int DOT_DURATION_HIGH = (DOT_DURATION + ERR_PAUSE);
179 const int DASH_DURATION_LOW = (DASH_DURATION - ERR_PAUSE);
```

```
const int DASH_DURATION_HIGH = (DASH_DURATION + ERR_PAUSE);
const int MIDWORDPAUSE_DURATION_LOW = (MIDWORDPAUSE_DURATION - ERR_PAUSE);
const int MIDWORDPAUSE_DURATION_HIGH = (MIDWORDPAUSE_DURATION + ERR_PAUSE);
const int BETWEENWORDPAUSE_DURATION_LOW =
    (BETWEENWORDPAUSE_DURATION - ERR_PAUSE);
const int BETWEENWORDPAUSE_DURATION_HIGH =
    (BETWEENWORDPAUSE_DURATION + ERR_PAUSE);

#define DASH '-'
#define DOT '.'
#define SPACE ' '

// Space for stroing dashes and dots
static char stack[512];
char *stack_ptr = stack;

/* USER CODE END 0 */

/**
 * @brief  The application entry point.
 * @retval int
 */
int main(void) {
  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick.
   */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC1_Init();
  MX_USB_DEVICE_Init();
  /* USER CODE BEGIN 2 */
  HAL_ADC_Start(&hadc1);

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  TRANSMISSION_STATE state = CALIBRATION;
  int last_timestamp = 0, duration, measured;
  while (1) {
    switch (state) {
    case NORMAL:
      measured = getADCvalue();
      if (measured > (reference_value + threshhold)) {
        last_timestamp = HAL_GetTick();
        state = MEASURING_HIGH;
        stack_ptr = stack;
        log("Transition: NORMAL -> MEASURING_HIGH");
      }
      break;
    case MEASURING_HIGH:
      if (getADCvalue() < reference_value + threshhold) {
        duration = HAL_GetTick() - last_timestamp;
        if ((DOT_DURATION_LOW < duration) && (duration < DOT_DURATION_HIGH)) {
          *(stack_ptr++) = DOT;
        } else if ((DASH_DURATION_LOW < duration) &&
                   (duration < DASH_DURATION_HIGH)) {
```

```
          *(stack_ptr++) = DASH;
        }
        state = MEASURING_LOW;
        log_dec("Measured high state ", duration);
        last_timestamp = HAL_GetTick();
      }
      break;
    case MEASURING_LOW:
      duration = HAL_GetTick() - last_timestamp;

      if (duration > 2 * BETWEENWORDPAUSE_DURATION_HIGH) {
        log("Timeout");
        state = FINISH;
      }

      if (getADCvalue() > reference_value + threshhold) {
        state = MEASURING_HIGH;
        last_timestamp = HAL_GetTick();
        if ((PAUSE_DURATION_LOW < duration) &&
            (duration < PAUSE_DURATION_HIGH)) {
          log("Found pause between dot and dash");
        } else if ((BETWEENWORDPAUSE_DURATION_LOW < duration) &&
                   (duration < BETWEENWORDPAUSE_DURATION_HIGH)) {
          *(stack_ptr++) = SPACE;
          *(stack_ptr++) = 'X';
          *(stack_ptr++) = SPACE;
          log("Found pause between words");
        } else if ((MIDWORDPAUSE_DURATION_LOW < duration) &&
                   (duration < MIDWORDPAUSE_DURATION_HIGH)) {
          *(stack_ptr++) = SPACE;
          log("Found pause between letters");
        } else {
          log_dec("ERROR, low state too long, going back to NORMAL", duration);
          state = NORMAL;
        }
      }
      break;
    case FINISH:
      *(stack_ptr++) = 0;
      convertBuffer(stack);
      HAL_Delay(20);
      log(human_readable);
      state = NORMAL;
      break;
```

C.2 Transmitter module

```
// ##############################################################################
//   Project: Optical Transmission of Text
//   Module:  Transmitter
//   Author:  Mikoaj Pastucha
//   Date:    25.11.2023
//   MCU:     Arduino Nano
// ##############################################################################

// Define the mapping between the alphabet and Morse code symbols
const char* morseCode[] = {
  ".-",   // A
  "-...", // B
  "-.-.", // C
  "-..",  // D
  ".",    // E
  "..-.", // F
  "--.",  // G
  "....", // H
  "..",   // I
  ".---", // J
  "-.-",  // K
  ".-..", // L
  "--",   // M
  "-.",   // N
  "---",  // O
  ".--.", // P
  "--.-", // Q
  ".-.",  // R
  "...",  // S
  "-",    // T
  "..-",  // U
  "...-", // V
  ".--",  // W
  "-..-", // X
  "-.--", // Y
  "--.."  // Z
};

const int ledPin = 7; // Pin number for the LED
const int buttonPin = 8; // Pin number for the button
const int dotTime = 200;      // Time for a dot (in milliseconds)
const int dashTime = dotTime * 3; // Time for a dash (in milliseconds)
const int interElementSpace = dotTime; // Time between dots and dashes within a
    letter
const int interLetterSpace = dotTime * 3; // Time between letters
const int interWordSpace = dotTime * 7; // Time between words

void setup() {

   // Set up the serial communication
  Serial.begin(9600);
   pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT); // Set the button pin as input
}

void loop() {
  // Capture the text input from the user
  String input;
  Serial.println("Enter text to convert to Morse code:");
```

```
  while (!Serial.available()) {
    // Wait for user input
  }
  input = Serial.readString();
  input.trim(); // Remove leading and trailing whitespaces

  // Convert the input to Morse code
  String morseCodeOutput = "";
  for (int i = 0; i < input.length(); i++) {
    char c = input.charAt(i);
    if (c == ' ') {
      morseCodeOutput += "   "; // Use three spaces to separate words in Morse
    code
    } else if (c >= 'A' && c <= 'Z') {
      morseCodeOutput += String(morseCode[c - 'A']) + " ";
    } else if (c >= 'a' && c <= 'z') {
      morseCodeOutput += String(morseCode[c - 'a']) + " ";
    }
  }

  // Output the Morse code and original text to the serial monitor
  Serial.println("Original text: " + input);
  Serial.println("Morse code: " + morseCodeOutput);

  // Output the Morse code using the LED
  for (int i = 0; i < input.length(); i++) {
    char c = input.charAt(i);
    if (c == ' ') {
      delay(interWordSpace);
    } else if (c >= 'A' && c <= 'Z') {
      outputMorseCode(morseCode[c - 'A']);
      delay(interLetterSpace);
    } else if (c >= 'a' && c <= 'z') {
      outputMorseCode(morseCode[c - 'a']);
      delay(interLetterSpace);
    }
  }
}

void outputMorseCode(const char* code) {
  for (int i = 0; i < strlen(code); i++) {
    if (code[i] == '.') {
      digitalWrite(ledPin, HIGH);
      delay(dotTime);
      digitalWrite(ledPin, LOW);
      delay(interElementSpace);
    } else if (code[i] == '-') {
      digitalWrite(ledPin, HIGH);
      delay(dashTime);
      digitalWrite(ledPin, LOW);
      delay(interElementSpace);
    }
  }
}
```