Факультет программной инженерии и компьютерной техники

Проектирование вычислительных систем

Лабораторная работа №2

Вариант 2: Последовательный интерфейс UART

Преподаватель: Пинкевич Василий Юрьевич

Выполнили: Тарасов Александр Станиславович, Кульбако Артемий Юрьевич
P34112

Санкт-Петербург
2021

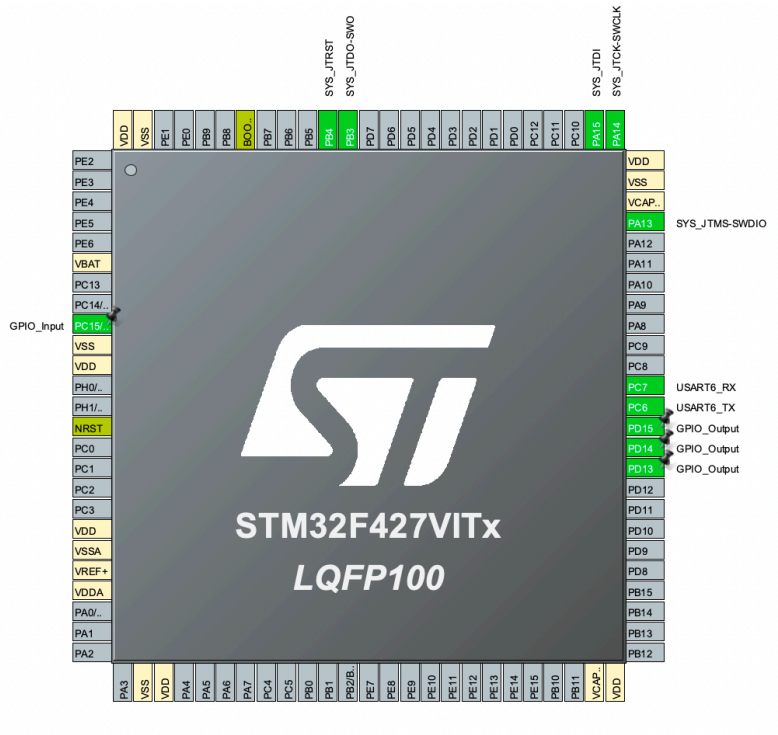
Задание

Доработать программу «гирлянда», реализовав возможность добавления четырёх новых
последовательностей миганий светодиодов с индивидуальной настройкой частоты переключения состояний для каждой последовательности. Каждая вводимая последовательность должна иметь от двух до восьми состояний. В один момент времени может гореть только один светодиод (или не гореть ни один). Смена отображаемой в данный момент последовательности должна осуществляться нажатием кнопки или командой, посылаемой через UART.
Должны обрабатываться следующие команды, посылаемые через UART:

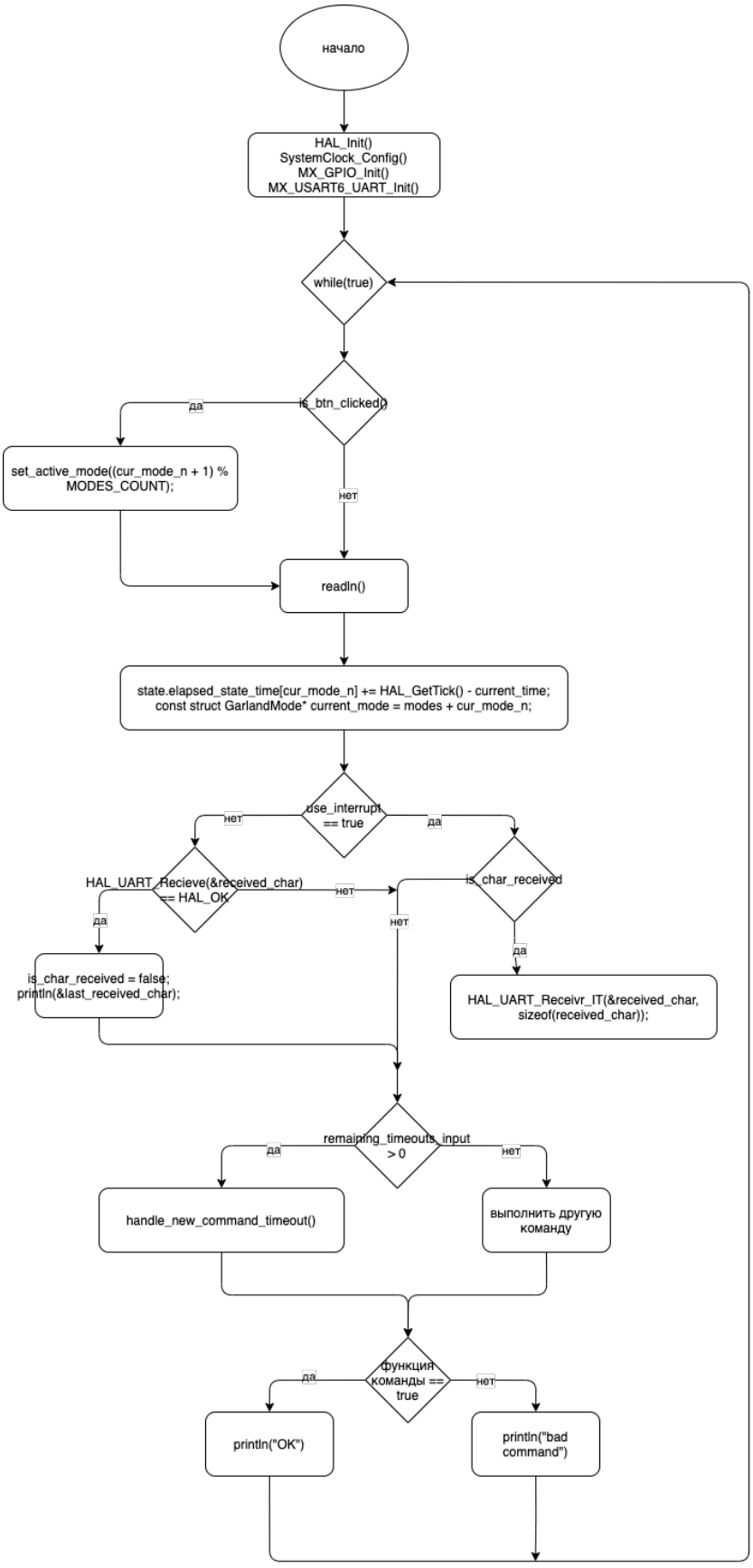
- – `new xx...` – ввести новую последовательность, где «x» – это одна из букв g, r, y, n («g» соответствует включению зелёного светодиода, «r» - красного, «y» - жёлтого, «n» означает, что ни один светодиод не горит); количество вводимых значений «x» может быть от двух до восьми, ввод завершается либо по нажатию enter, либо после ввода восьми значений; после окончания ввода последовательности мерцаний стенд должен послать сообщение произвольного содержания, приглашающее ввести частоту мерцаний светодиодов (должны предусматриваться минимум три градации); ввод частоты мерцаний заканчивается по нажатию *enter*; новой последовательности присваивается очередной свободный номер от 5 до 8; если уже есть 8 последовательностей, то переопределяется последовательность 5 и т.д.; номер новой
  сохраненной последовательности выводится в UART;

- – `set x` – сделать активной последовательность мерцаний *x*, где *x* – порядковый номер;

- – `set interrupts on` или `set interrupts off` – включить или выключить прерывания.



Блок-схема


начало
HAL_Init()
SystemClock_Config()
MX_GPIO_Init()
MX_USART6_UART_Init()
while(true)
is_btn_clicked()
да
set_active_mode((cur_mode_n + 1) % MODES_COUNT);
нет
readln()
state.elapsed_state_time[cur_mode_n] += HAL_GetTick() - current_time;
const struct GarlandMode* current_mode = modes + cur_mode_n;
use_interrupt == true
нет
да
HAL_UART_Recieve(&received_char) == HAL_OK
нет
да
is_char_received
нет
да
is_char_received = false;
println(&last_received_char);
HAL_UART_Receivr_IT(&received_char, sizeof(received_char));
remaining_timeouts_input > 0
да
нет
handle_new_command_timeout()
выполнить другую команду
функция команды == true
да
нет
println("OK")
println("bad command")

Драйвер

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2021 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "usart.h"
#include "gpio.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

#include <stdbool.h>
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include <stdarg.h>

/* USER CODE END Includes */

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

const char OK_MSG[] = "OK";
char cmd[256];
uint32_t last_pressed_time = 0;
uint8_t prev_mode_n = 7;
uint8_t cur_mode_n = 1;
bool is_char_received = false;
bool use_interrupt = false;
bool is_transmitted = true;
char last_received_char;
uint8_t remaining_timeouts_input = 0;
uint8_t MODES_COUNT = 4;

enum LED {
    LED_NO_ONE = 0,
    LED_RED = 1,
    LED_GREEN = 2,
    LED_YELLOW = 3,
};

struct LightState {
    enum LED color;
    uint32_t timeout;
};

struct GarlandMode {
    uint8_t light_states_count;
    struct LightState states[8];
};

struct State {
    uint8_t state[8];
    uint32_t elapsed_state_time[8];
};

typedef void (* set_led_function)(bool);

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
```

```c
#define UART_TIMEOUT 10
/* USER CODE END PD */

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

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */

/* USER CODE END PV */

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

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void set_green_led(bool on) { HAL_GPIO_WritePin(GPIOD, GPIO_PIN_13, on ? GPIO_PIN_SET :
GPIO_PIN_RESET); }

void set_yellow_led(bool on) { HAL_GPIO_WritePin(GPIOD, GPIO_PIN_14, on ? GPIO_PIN_SET :
GPIO_PIN_RESET); }

void set_red_led(bool on) { HAL_GPIO_WritePin(GPIOD, GPIO_PIN_15, on ? GPIO_PIN_SET :
GPIO_PIN_RESET); }

int is_btn_clicked() {
        // GPIO_PIN_RESET means pressed
        if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_15) == GPIO_PIN_RESET && HAL_GetTick() -
last_pressed_time > 1000) {
                last_pressed_time = HAL_GetTick();
                return 1;
        } else return 0;
}

void set_no_one_led(bool on) {}

const set_led_function led_functions[] = {
    [LED_NO_ONE] = set_no_one_led,
    [LED_RED] = set_red_led,
    [LED_GREEN] = set_green_led,
    [LED_YELLOW] = set_yellow_led,
};


struct GarlandMode modes[8] = {
                {
                        .light_states_count = 2,
            .states = {
                {
                    .color = LED_YELLOW,
                    .timeout = 250,
                },
                {
                    .color = LED_GREEN,
                    .timeout = 250,
                },
            },
        },
        {
            .light_states_count = 6,
            .states = {
                {
                    .color = LED_RED,
                    .timeout = 250,
                },
                {
                    .color = LED_NO_ONE,
                    .timeout = 250,
                },
                {
                    .color = LED_YELLOW,
                    .timeout = 250,
                },
                {
                    .color = LED_NO_ONE,
```

```
                    .timeout = 250,
                },
                {
                    .color = LED_GREEN,
                    .timeout = 250,
                },
                {
                    .color = LED_NO_ONE,
                    .timeout = 250,
                },
            },
        },
        {
            .light_states_count = 4,
            .states = {
                {
                    .color = LED_GREEN,
                    .timeout = 400,
                },
                {
                    .color = LED_NO_ONE,
                    .timeout = 250,
                },
                {
                    .color = LED_RED,
                    .timeout = 400,
                },
                {
                    .color = LED_NO_ONE,
                    .timeout = 250,
                },
            },
        },
        {
            .light_states_count = 6,
            .states = {
                {
                    .color = LED_RED,
                    .timeout = 3000,
                },
                {
                    .color = LED_NO_ONE,
                    .timeout = 1000,
                },
                {
                    .color = LED_YELLOW,
                    .timeout = 3000,
                },
                {
                    .color = LED_NO_ONE,
                    .timeout = 1000,
                },
                {
                    .color = LED_GREEN,
                    .timeout = 3000,
                },
                {
                    .color = LED_NO_ONE,
                    .timeout = 1000,
                },
            },
        }
    };

struct GarlandMode new_mode;

struct State state = {
    .state = { 0 },
    .elapsed_state_time = { 0 },
};

void print(const char * content) {
       if (use_interrupt) {
              while (!is_transmitted);
              is_transmitted = false;
              HAL_UART_Transmit_IT(&huart6, (void *) content, strlen(content));
       } else HAL_UART_Transmit(&huart6, (void *) content, strlen(content), UART_TIMEOUT);
}

void println(const char * message) {
       print(message);
```

```
        print("\r\n");
}

void print_format(const char * format, ...) {
        static char buffer[1024];
        if (use_interrupt) while (!is_transmitted);
        va_list ap;
        va_start(ap, format);
        vsnprintf(buffer, sizeof(buffer), format, ap);
        va_end(ap);
        println(buffer);
}

bool string_equals(const char * a, const char * b) { return strcmp(a, b) == 0; }

bool starts_with(const char * prefix, const char * str) { return strncmp(prefix, str,
strlen(prefix)) == 0; }

void set_active_mode(uint8_t mode_number) {
        led_functions[modes[cur_mode_n].states[state.state[cur_mode_n]].color](false);
        cur_mode_n = mode_number;
        if (modes[cur_mode_n].light_states_count > 0)
led_functions[modes[cur_mode_n].states[state.state[cur_mode_n]].color](true);
}

bool handle_set_command() {
        const char * const mode_idx_str = cmd + 4;
        uint32_t mode_idx;
        if ((sscanf(mode_idx_str, "%lu", &mode_idx) != 1) || (mode_idx < 1 || mode_idx >
MODES_COUNT)) return false;
        set_active_mode(mode_idx - 1);
        return true;
}

bool handle_new_command() {
        const char* const pattern = cmd + 4; // set pointer after 'new '
        const uint32_t pattern_length = strlen(pattern);
        if (pattern_length < 2 || pattern_length > 8) return false;
        new_mode.light_states_count = pattern_length;
        for (uint8_t i = 0; i < pattern_length; ++i)
                switch (pattern[i]) {
                        case 'n':
                                new_mode.states[i].color = LED_NO_ONE;
                                break;
                        case 'r':
                                new_mode.states[i].color = LED_RED;
                                break;
                        case 'g':
                                new_mode.states[i].color = LED_GREEN;
                                break;
                        case 'y':
                                new_mode.states[i].color = LED_YELLOW;
                                break;
                        default:
                                return false;
                }
        remaining_timeouts_input = pattern_length;
        print_format("print %d light on timeout in millis:\r\n", pattern_length);
        return true;
}

bool handle_new_command_timeout() {
        const uint8_t state_idx = new_mode.light_states_count - remaining_timeouts_input;
        if (sscanf(cmd, "%lu", &(new_mode.states[state_idx].timeout)) != 1) return false;
        --remaining_timeouts_input;
        if (remaining_timeouts_input == 0) {
                const uint8_t available_mode_idxs = sizeof(modes) / sizeof(*(modes)) -
MODES_COUNT;
                const uint8_t mode_idx = MODES_COUNT + (prev_mode_n - MODES_COUNT + 1) %
available_mode_idxs;
                if (MODES_COUNT <= mode_idx) MODES_COUNT = mode_idx + 1;
                prev_mode_n = mode_idx;
                memcpy(modes + mode_idx, &(new_mode), sizeof(new_mode));
                print_format("Written in mode %d\r\n", mode_idx + 1);
                return true;
        }
        print_format("%d timeouts remaining:\r\n", remaining_timeouts_input);
        return true;
}

void handle_command_line() {
```

```
        bool cmd_exec_stat = false;
        if (strlen(cmd) != 0) {
                if (string_equals("set interrupts on", cmd)) {
                        use_interrupt = true;
                        cmd_exec_stat = true;
                }
                else if (string_equals("set interrupts off", cmd)) {
                        use_interrupt = false;
                        cmd_exec_stat = true;
                }
                else if (starts_with("set ", cmd)) cmd_exec_stat = handle_set_command(state);
                else if (starts_with("new ", cmd)) {
                        handle_new_command(state);
                        return;
                }
                else if (remaining_timeouts_input > 0) cmd_exec_stat =
handle_new_command_timeout();
                else cmd_exec_stat = false;
                println((cmd_exec_stat) ? OK_MSG : "bad command");
        }
}

void readln() {
    if (use_interrupt) {
         if (!is_char_received) {
             HAL_UART_Receive_IT(&huart6, (void *) &last_received_char,
sizeof(last_received_char));
             return;
         }
    } else if (HAL_UART_Receive(&huart6, (void *) &last_received_char,
sizeof(last_received_char), UART_TIMEOUT) != HAL_OK) return;
        is_char_received = false;
    print(&last_received_char);
    switch (last_received_char) {
        case '\b':
        case 0x7F: {
            const uint8_t cmd_len = strlen(cmd);
            if (cmd_len > 0) cmd[cmd_len - 1] = '\0';
                return;
        }
        case '\r':
            println("\n");
                handle_command_line(state);
                memset(cmd, '\0', sizeof(cmd));
                return;
    }
    const uint32_t command_line_length = strlen(cmd);
    // overflow
    if (command_line_length == sizeof(cmd) - 1) {
        println("\r\n invalid command");
        memset(cmd, '\0', sizeof(cmd));
        return;
    }
    cmd[command_line_length] = last_received_char;
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) { is_char_received = true; }

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) { is_transmitted = true; }

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

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

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */
```

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

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART6_UART_Init();
  /* USER CODE BEGIN 2 */
  /* USER CODE END 2 */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
    uint32_t current_time = HAL_GetTick();
    while (1) {
        if (is_btn_clicked()) set_active_mode((cur_mode_n + 1) % MODES_COUNT);
        readln();
        state.elapsed_state_time[cur_mode_n] += HAL_GetTick() - current_time;
        current_time = HAL_GetTick();
        const struct GarlandMode* current_mode = modes + cur_mode_n;
        if (current_mode->light_states_count == 0) continue;
        const struct LightState* current_state = current_mode->states + state.state[cur_mode_n];
        if (state.elapsed_state_time[cur_mode_n] >= current_state->timeout) {
            led_functions[current_state->color](false);
            state.elapsed_state_time[cur_mode_n] = 0;
            state.state[cur_mode_n] = (state.state[cur_mode_n] + 1) % current_mode-
>light_states_count;
            led_functions[current_mode->states[state.state[cur_mode_n]].color](true);
        }
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE3);
  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
  {
    Error_Handler();
  }
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
```

```
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */

  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     tex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

Вывод

В ходе выполнения лабораторной работы мы научились работать с светодиодами и кнопкой стенда SDK-1.1M на базе ARM-процессора STM32F427VI, разработали драйвер для управления им через терминал компьютера посредством интерфейса UART (с прерываниями и без). Главная сложность работы заключалась в считывании команд с клавиатуры и работы со строками в C (весьма сложная задача в сравнении в высокоуровневыми языками).