

Федеральное государственное автономное образовательное  
учреждение высшего образования  
«Национальный исследовательский университет ИТМО»

Дисциплина: Проектирование вычислительных систем

**Лабораторная работа №2**

Вариант 4

**Выполнили:**

Барсуков Максим Андреевич,  
группа Р3415

Стригалев Никита Сергеевич,  
группа Р3412

**Преподаватель:**

Пинкевич Василий Юрьевич

2025 г.

Санкт-Петербург

## **Содержание**

Задание.....	3
Вариант: 4.....	4
Блок-схема.....	5
Описание работы алгоритма.....	6
Исходный код.....	7
Вывод.....	17

## **Задание**

Разработать и реализовать два варианта драйверов UART для стенда SDK-1.1M: с использованием и без использования прерываний. Драйверы, использующие прерывания, должны обеспечивать работу в «неблокирующем» режиме (возврат из функции происходит сразу же, без ожидания окончания приема/отправки), а также буферизацию данных для исключения случайной потери данных. В драйвере, не использующем прерывания, функция приема данных также должна быть «неблокирующей», то есть она не должна зависать до приема данных (которые могут никогда не поступить). При использовании режима «без прерываний» прерывания от соответствующего блока UART должны быть запрещены.

Написать с использованием разработанных драйверов программу, которая выполняет определенную вариантом задачу. Для всех вариантов должно быть реализовано два режима работы программы: с использованием и без использования прерываний. Каждый принимаемый стендом символ должен отсылаться обратно, чтобы он был выведен в консоли (так называемое «эхо»). Каждое новое сообщение от стенда должно выводиться с новой строки. Если вариант предусматривает работу с командами, то на каждую команду должен выводиться ответ, определенный в задании или «OK», если ответ не требуется. Если введена команда, которая не поддерживается, должно быть выведено сообщение об этом.

Скорость работы интерфейса UART должна соответствовать указанной в варианте задания.

## **Вариант: 4**

Разработать программу-калькулятор. Ввод значений производится с компьютера через UART:  $xx...xuxx...x=$ , где  $x$  – десятичные цифры,  $u$  – знак (+, -, \*, /). Ввод чисел завершается либо знаком операции (для первого числа), либо знаком «равно» (для второго числа), либо после ввода пяти цифр числа. Обратите внимание, что операнды не могут быть отрицательными, а ответ может.

Размерность результата и обоих операндов должна быть `short int` (16-битовое знаковое число), и должна быть предусмотрена защита от переполнения. В случае выполнения недопустимых операций (ответ или вводимые числа больше, чем размер переменных в памяти) должен загораться красный светодиод, а в последовательный канал вместо ответа выводиться слово `error`.

Включение/отключение прерываний должно осуществляться нажатием кнопки на стенде и сопровождаться отправкой в последовательный порт сообщения произвольного содержания, сообщающего, какой режим включен (с прерываниями или без прерываний). При смене режима необходимо сбрасывать все введенные пользователем данные и выводить приглашение начать ввод заново.

Скорость обмена данными по UART – 19200 бит/с.

## Блок-схема

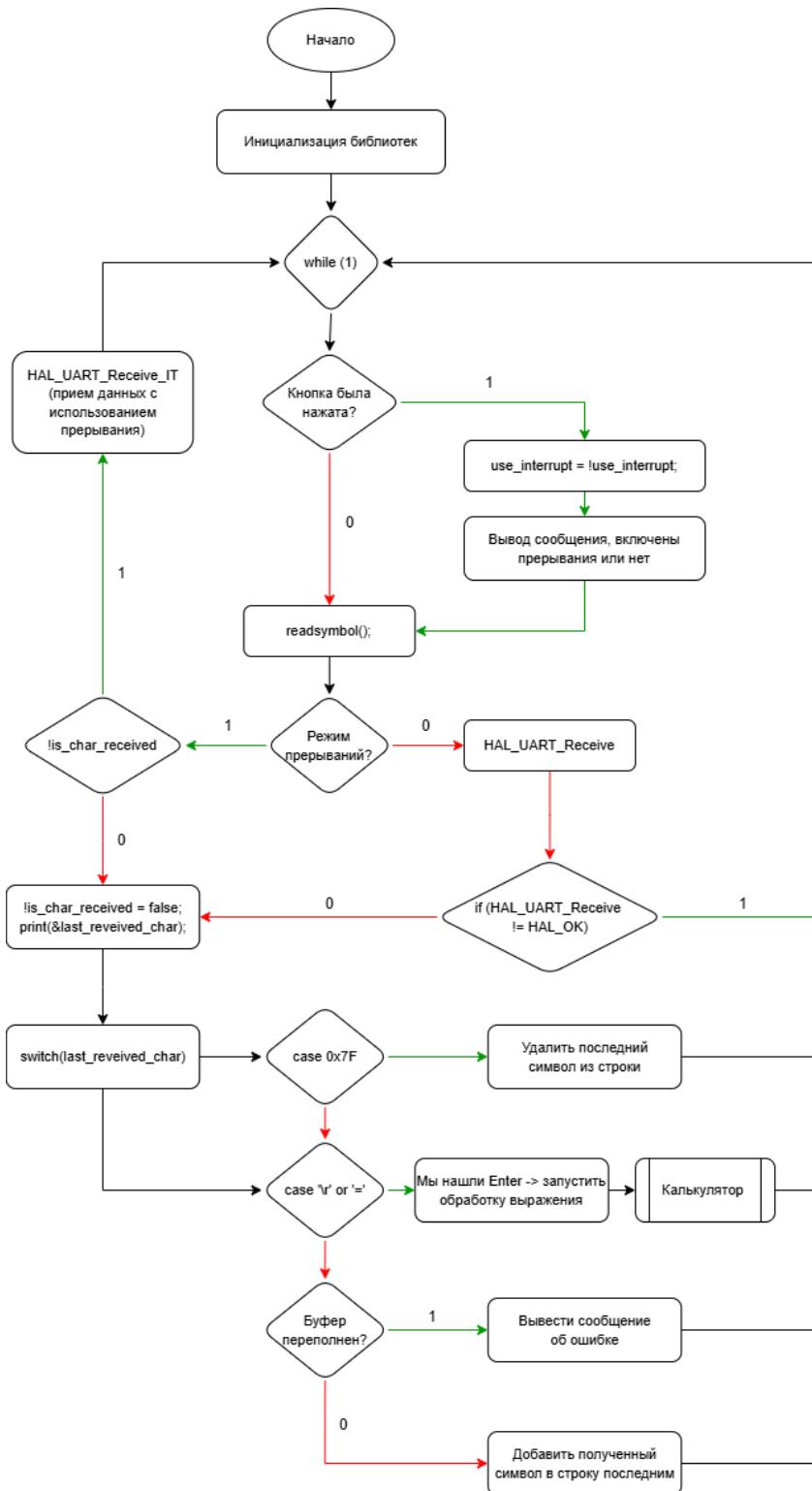


Рисунок 1 – Схема алгоритма

## **Описание работы алгоритма**

В нашей программе реализовано два режима работы: с использованием прерываний и без них.

В режиме с прерываниями приём символов осуществляется с помощью функции `HAL_UART_Receive_IT`, а в режиме без прерываний — посредством опроса с таймаутом через `HAL_UART_Receive`.

После получения данных выполняется посимвольная проверка ввода: контролируется допустимый диапазон значений, длина числа и наличие специальных управляющих символов (таких как +, -, \*, /, = и т.д.).

Если ввод содержит недопустимые символы, превышает допустимую длину или нарушает синтаксис выражения, программа выводит сообщение об ошибке. В противном случае выполняются арифметические вычисления, и на выход подаётся корректный результат.

## Исходный код

```
/* USER CODE BEGIN Header */
/**
 * @file          : main.c
 * @brief         : Main program body
 */
 * @attention
 *
 * Copyright (c) 2022 STMicroelectronics.
 * All rights reserved.
 *
 * This software is licensed under terms that can be found in the LICENSE file
 * in the root directory of this software component.
 * If no LICENSE file comes with this software, it is provided AS-IS.
 *
 */
/* USER CODE END Header */

/* Includes ----- */
#include "main.h"
#include "usart.h"
#include "gpio.h"

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

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

/* USER CODE END Includes */

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

/* USER CODE END PTD */

/* Private define ----- */
/* USER CODE BEGIN PD */
#define UART_TIMEOUT 10
#define BUF_SIZE 1024
#define MAX_DIGITS 5
/* 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 */
struct CircularQueue {
    char data[BUF_SIZE];
    uint16_t write_pos;
    uint16_t read_pos;
    bool empty;
};

struct ButtonStatus {
    bool pressed;
    bool acknowledged;
    uint32_t pressed_timestamp;
};

struct SystemState {
    bool interrupts_active;
    uint32_t interrupt_mask;
};

typedef struct CircularQueue CircularQueue;
static struct CircularQueue input_queue;
static struct CircularQueue output_queue;
static struct SystemState system_state;

static void queue_init(CircularQueue *queue) {
    queue->write_pos = 0;
    queue->read_pos = 0;
    queue->empty = true;
}

static void queue_push(CircularQueue *queue, char *element) {
    uint16_t element_size = strlen(element);

    if (queue->write_pos + element_size + 1 > BUF_SIZE) {
        queue->write_pos = 0;
    }

    strcpy(&queue->data[queue->write_pos], element);
    queue->write_pos += element_size + 1;
}

```

```

    if (queue->write_pos == BUF_SIZE) {
        queue->write_pos = 0;
    }

    queue->empty = false;
}

static bool queue_pop(CircularQueue *queue, char *element) {
    if (queue->empty) {
        return false;
    }

    uint16_t element_size = strlen(&queue->data[queue->read_pos]);

    strcpy(element, &queue->data[queue->read_pos]);
    queue->read_pos += element_size + 1;

    if (queue->read_pos == BUF_SIZE || queue->data[queue->read_pos] == '\0') {
        queue->read_pos = 0;
    }

    if (queue->read_pos == queue->write_pos) {
        queue->empty = true;
    }

    return true;
}

static char input_char[2] = {"\0"};

static bool check_button_pressed() {
    return HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_15) == GPIO_PIN_RESET;
}

static void control_led(GPIO_TypeDef* port, uint16_t pin, bool state) {
    HAL_GPIO_WritePin(port, pin, state ? GPIO_PIN_SET : GPIO_PIN_RESET);
}

static bool process_button_status(struct ButtonStatus *status) {
    if (status->pressed) {
        status->pressed = check_button_pressed();

        if (status->acknowledged) {
            return false;
        }

        if ((HAL_GetTick() - status->pressed_timestamp) > 20) {
            status->acknowledged = true;
            return true;
        }
        return false;
    }

    if (check_button_pressed()) {
        status->pressed_timestamp = HAL_GetTick();
    }
}

```

```

        status->pressed = true;
        status->acknowledged = false;
    }

    return false;
}

bool transmit_busy = false;

void activate_interrupts(struct SystemState *state) {
    HAL_NVIC_EnableIRQ(USART6_IRQn);
    state->interrupts_active = true;
    HAL_UART_Receive_IT(&huart6, (uint8_t*)input_char, 1);
}

void deactivate_interrupts(struct SystemState *state) {
    HAL_UART_AbortReceive(&huart6);
    HAL_NVIC_DisableIRQ(USART6_IRQn);
    state->interrupts_active = false;
}

void send_uart_data(const struct SystemState *state, char *buffer, size_t length) {
    if (state->interrupts_active) {
        if (transmit_busy) {
            queue_push(&output_queue, buffer);
        } else {
            HAL_UART_Transmit_IT(&huart6, (uint8_t*)buffer, length);
            transmit_busy = true;
        }
        return;
    }
    HAL_UART_Transmit(&huart6, (uint8_t*)buffer, length, 100);
}

void send_uart_line(const struct SystemState *state, char *buffer, size_t length) {
    send_uart_data(state, buffer, length);
    send_uart_data(state, "\r\n", 2);
}

void receive_uart_data(const struct SystemState *state) {
    if (state->interrupts_active) {
        return;
    }

    HAL_StatusTypeDef result = HAL_UART_Receive(&huart6, (uint8_t*)input_char, 1, 0);
    if (result == HAL_OK) {
        queue_push(&input_queue, input_char);
        send_uart_data(state, input_char, 1);
    }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart_instance) {
    queue_push(&input_queue, input_char);
    send_uart_data(&system_state, input_char, 1);
    HAL_UART_Receive_IT(&huart6, (uint8_t*)input_char, 1);
}

```

```

}

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart_instance) {
    char output_buffer[1024];
    if (queue_pop(&output_queue, output_buffer)) {
        HAL_UART_Transmit_IT(&huart6, (uint8_t*)output_buffer,
        strlen(output_buffer));
    } else {
        transmit_busy = false;
    }
}

enum CalculationStage {
    FirstOperandInput,
    FirstOperandComplete,
    SecondOperandInput,
    SecondOperandComplete,
    ComputeResult,
    ErrorOccurred
};
/* 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 */

    /* 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 */

```

```

    char interrupts_enabled_msg[] = {"Interrupts ENABLED"};
    char interrupts_disabled_msg[] = {"Interrupts DISABLED"};
    char error_msg[] = {"\r\nerror\r\n"};

    struct ButtonStatus button_status = {.pressed_timestamp = 0, .acknowledged =
false, .pressed = false};

    activate_interrupts(&system_state);
    queue_init(&input_queue);

    uint32_t first_operand = 0;
    uint32_t second_operand = 0;
    uint8_t first_digit_count = 0;
    uint8_t second_digit_count = 0;
    int64_t calculation_result = 0;
    char result_string[16];
    char operation_symbol;
    enum CalculationStage current_stage = FirstOperandInput;
    control_led(GPIOID, GPIO_PIN_15, false);

    while (1) {
        if (process_button_status(&button_status)) {
            if (system_state.interrupts_active) {
                deactivate_interrupts(&system_state);
                send_uart_line(&system_state, interrupts_disabled_msg,
sizeof(interrupts_disabled_msg));
            } else {
                activate_interrupts(&system_state);
                send_uart_line(&system_state, interrupts_enabled_msg,
sizeof(interrupts_enabled_msg));
            }
        }

        first_operand = 0;
        second_operand = 0;
        calculation_result = 0;
        first_digit_count = 0;
        second_digit_count = 0;
        current_stage = FirstOperandInput;
        control_led(GPIOID, GPIO_PIN_15, false);
    }

    receive_uart_data(&system_state);

    char input_char_buffer[2];

    if ((current_stage != ComputeResult && current_stage != ErrorOccurred) &&
!queue_pop(&input_queue, input_char_buffer)) {
        continue;
    }

    switch (current_stage) {
        case FirstOperandInput: {
            if (!isdigit(input_char_buffer[0])) {
                current_stage = ErrorOccurred;
                break;
            }
        }
    }
}

```

```

        current_stage = FirstOperandComplete;
        control_led(GPIOID, GPIO_PIN_15, false);
        first_operand = first_operand * 10 + (input_char_buffer[0] -
'0');

        first_digit_count++;
        uint16_t size_check = (uint16_t) first_operand;
        if (first_operand != size_check || first_digit_count >=
MAX_DIGITS) {
            current_stage = ErrorOccurred;
        }
        break;
    }

    case FirstOperandComplete: {
        if (!isdigit(input_char_buffer[0])) {
            switch (input_char_buffer[0]) {
                case '+':
                case '-':
                case '*':
                case '/': {
                    operation_symbol = input_char_buffer[0];
                    current_stage = SecondOperandInput;
                    break;
                }
                default:
                    current_stage = ErrorOccurred;
                    break;
            }
            break;
        }
        first_operand = first_operand * 10 + (input_char_buffer[0] -
'0');

        first_digit_count++;
        uint16_t size_check = (uint16_t) first_operand;
        if (first_operand != size_check || first_digit_count >=
MAX_DIGITS) {
            current_stage = ErrorOccurred;
        }
        break;
    }

    case SecondOperandInput: {
        if (!isdigit(input_char_buffer[0])) {
            current_stage = ErrorOccurred;
            break;
        }
        current_stage = SecondOperandComplete;
        second_operand = second_operand * 10 + (input_char_buffer[0] -
'0');

        second_digit_count++;
        uint16_t size_check = (uint16_t) second_operand;
        if (second_operand != size_check || second_digit_count >=
MAX_DIGITS) {
            current_stage = ErrorOccurred;
        }
        break;
    }
}

```

```

        }

    case SecondOperandComplete: {
        if (!isdigit(input_char_buffer[0])) {
            if (input_char_buffer[0] == '=') {
                current_stage = ComputeResult;
            } else {
                current_stage = ErrorOccurred;
            }
            break;
        }
        second_operand = second_operand * 10 + (input_char_buffer[0] -
        '0');

        second_digit_count++;
        uint16_t size_check = (uint16_t) second_operand;
        if (second_operand != size_check || second_digit_count >=
MAX_DIGITS) {
            current_stage = ErrorOccurred;
        }
        break;
    }

    case ComputeResult: {
        int32_t op1 = first_operand;
        int32_t op2 = second_operand;

        switch (operation_symbol) {
            case '+':
                calculation_result = op1 + op2;
                break;
            case '-':
                calculation_result = op1 - op2;
                break;
            case '*':
                calculation_result = op1 * op2;
                break;
            case '/':
                if (op2 == 0) {
                    current_stage = ErrorOccurred;
                    continue;
                } else {
                    calculation_result = op1 / op2;
                }
                break;
        }
    }

    if (calculation_result < -32768 || calculation_result > 32767) {
        current_stage = ErrorOccurred;
        break;
    }

    sprintf(result_string, "%d", (int)calculation_result);
    send_uart_line(&system_state, result_string,
strlen(result_string));
    first_operand = 0;
}

```

```

        second_operand = 0;
        calculation_result = 0;
        first_digit_count = 0;
        second_digit_count = 0;
        current_stage = FirstOperandInput;
        break;
    }

    case ErrorOccurred: {
        first_operand = 0;
        second_operand = 0;
        calculation_result = 0;
        first_digit_count = 0;
        second_digit_count = 0;
        control_led(GPIOID, GPIO_PIN_15, true);
        send_uart_line(&system_state, error_msg, sizeof(error_msg));
        current_stage = FirstOperandInput;
        break;
    }
}
/* 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.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 */
    __disable_irq();
    while (1)
    {
    }
    /* 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,
       ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
    /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

```

## **Вывод**

В ходе лабораторной работы был разработан простейший калькулятор, управляемый с клавиатуры. Реализация включала работу с UART как в режиме прерываний, так и без него, а также применение кольцевого буфера для обработки входных данных. Ошибки в вычислениях сигнализировались с помощью красного светодиода.