Факультет программной инженерии и компьютерной техники

Лабораторная работа №2

«Последовательный интерфейс UART»

по дисциплине «Проектирование вычислительных систем»

Вариант – 1

Выполнили:

Студенты группы P3432

Чмурова М.В.

Комягин Д.А.

Преподаватель:

Пинкевич Василий Юрьевич

Санкт-Петербург

2025

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 «ОК», 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 1:

Доработать программу «светофор», добавив возможности отключения кнопки и задания величины тайм-аута (период, в течение которого горит красный).

Должны обрабатываться следующие команды, посылаемые через UART:

- *?* – в ответ стенд должен прислать
    - состояние, которое отображается в данный момент на светодиодах: red, yellow, green, blinking green,
    - режим – mode 1 или mode 2 (см. 78 далее),
    - величину тайм-аута (сколько горит красный) – timeout …,
    - и задействованы ли прерывания – символ I (interrupt) или P (polling);
- *set mode 1* или *set mode 2* – установить режим работы светофора, когда обрабатываются или игнорируются нажатия кнопки;
- *set timeout X* – установить тайм-аут (X – длина периода в секундах);
- *set interrupts on* или *set interrupts off* – включить или выключить прерывания.

Скорость обмена данными по UART – 57600 бит/с.

Описание организации программы и структуры драйверов

Для реализации работы драйвера UART без прерываний воспользуемся переменной:

```
#define UART_TIMEOUT 10
```

Это максимальное время, которое будет возможно для передачи байта. Этого числа должно быть достаточно для использования всеми командами в режиме без прерываний.

Для хранения полученных данных (receive) или данных, которые будут отправлены на ПК (transmit), будут использованы кольцевые буферы, с указателем на начало (head – адрес, по которому положить байты в буфер) и конец (tail – адрес, с которого брать байты из буфера):

```
static uint8_t rx_buf[UART_RX_BUF_SIZE];
static uint8_t tx_buf[UART_TX_BUF_SIZE];

static volatile uint16_t rx_head = 0, rx_tail = 0;
static volatile uint16_t tx_head = 0, tx_tail = 0;
```

Таким образом, байты из буфера извлекаются по принципу FIFO. В случае переполнения буферов будет также использован подход FIFO: первый положенный в буфер байт будет потерян.

Для управления режимами с прерываниями/без прерываний будет использована глобальная переменная:

```
static uint8_t interrupts_enabled = 0;
```

Которая изменяется в случае смены режима.

Разбор функций, реализованных в драйвере:

```
static inline uint32_t uart_critical_enter(void) {
    uint32_t pmask = __get_PRIMASK();
    __disable_irq();
    return pmask;
}

static inline void uart_critical_exit(uint32_t pmask) {
    __set_PRIMASK(pmask);
```

Использование критических секций необходимо для того, чтобы обеспечить атомарность указателей tail и head в функциях *uart_transmit_char()* и *uart_receive_char()*.

```
void uart_set_interrupts(uint8_t enabled)
{
```

```
    interrupts_enabled = enabled;
    if (enabled)
    {
        rx_head = rx_tail = 0;
        tx_head = tx_tail = 0;
         HAL_NVIC_EnableIRQ(USART6_IRQn);
        HAL_UART_Receive_IT(&huart6, &rx_byte, 1);
    }
    else
    {
        HAL_UART_AbortReceive_IT(&huart6);
        HAL_UART_AbortTransmit_IT(&huart6);
        HAL_NVIC_DisableIRQ(USART6_IRQn);
    }
}
```

При установке режима с прерываниями значения кольцевых буферов сбрасываются, непосредственно разрешается доступ к прерываниям для USART6, и ожидается поступление одного байта с ПК (это включает RXNEIE бит, который позволяет вызывать прерывания при поступлении бита с линии RX – по завершении передачи кадра байт складывается RDR, а RXNE = 1). При установке режима без прерывания прерываются текущие RX и TX-сессии, устанавливая состояния TX и RX в READY. А также NVIC перестает пропускать любые прерывания от USART6 в ядро.

```
void uart_transmit_char(char ch)
{
    if (!interrupts_enabled) {
        HAL_UART_Transmit(&huart6, (uint8_t *)&ch, 1, UART_TIMEOUT);
        return;
    }

    uint32_t pmask = uart_critical_enter();

    tx_buf[tx_head] = (uint8_t)ch;
    tx_head = (tx_head + 1) % UART_TX_BUF_SIZE;

      if (tx_head == tx_tail)
            tx_tail = (tx_tail + 1) % UART_TX_BUF_SIZE;

    uint8_t need_start = (__HAL_UART_GET_FLAG(&huart6, UART_FLAG_TXE) != 0)
&& (tx_head != tx_tail);

    uart_critical_exit(pmask);

    if (need_start) {
        HAL_UART_Transmit_IT(&huart6, &tx_buf[tx_tail], 1);
    }
}

void uart_transmit_string(char *str) {
     while (*str) uart_transmit_char(*str++);
}
```

```
void uart_transmitln_string(char *str)
{
      uart_transmit_string(str);
      uart_transmit_string("\r\n");
```

Transmit используется для передачи байта с МК на ПК. В режиме без прерываний потог блокируется на тайм-аут до передачи байта. В режиме с прерываниями байт на отправку кладется в буфер, смещается указатель head, проверяется переполнение буфера и в случае наличия флага TXE (регистра передачи данных TDR пуст и возможна отправка), байт передается на передачу в HAL_UART_Transmit_IT().

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance != USART6) return;
    if (!(tx_head == tx_tail))
    {
        tx_tail = (tx_tail + 1) % UART_TX_BUF_SIZE;
        if (!(tx_head == tx_tail))
            HAL_UART_Transmit_IT(&huart6, &tx_buf[tx_tail], 1);
    }
}
```

После передачи байта происходит callback. Он сдвигает хвост после отправки (подтверждение отправки) и если в буфере еще остались данные, то запускает передачу следующего байта.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance != USART6) return;

    rx_buf[rx_head] = rx_byte;
    rx_head = (rx_head + 1) % UART_RX_BUF_SIZE;

    if (rx_head == rx_tail)
        rx_tail = (rx_tail + 1) % UART_RX_BUF_SIZE;

    HAL_UART_Receive_IT(&huart6, &rx_byte, 1);
}
```

После поступления прочитанного байта из RDR в rx_byte, вызывается callback – это точка программы, где полученный байт перекладывается из rx_byte в rx_buf и стартует прием следующего байта.

```
int uart_receive_char(char *ch)
{
    if (!interrupts_enabled) {
        return (HAL_UART_Receive(&huart6, (uint8_t *)ch, 1, 0) == HAL_OK) ? 1
: 0;
    }
    else
    {
```

```
        int is_byte_available = 0;
        uint32_t pmask = uart_critical_enter();

        if (rx_head != rx_tail) {
            *ch = rx_buf[rx_tail];
            rx_tail = (rx_tail + 1) % UART_RX_BUF_SIZE;
            is_byte_available = 1;
        }

        uart_critical_exit(pmask);
        return is_byte_available;
    }
}
```

В случае режима работы без прерываний происходит чтение байта в ch и возврат 1, если он уже лежал в RDR и был получен – иначе 0. В режиме с прерываниями, если был получен байт, то читаем его из rx_buf и возвращаем 1. Иначе возвращаем 0.

Для обработки команд, поступаемых от ПК, использовался следующий enum:

```
enum {
      CMD_UNKNOWN,
      CMD_STATUS,
      CMD_SET_MODE,
      CMD_SET_TIMEOUT,
      CMD_SET_INTERRUPTS
      } command_type = CMD_UNKNOWN;
```

Он необходим для корректной обработки каждой поступающей команды и присваивания ей типа.

Блок-схема прикладного алгоритма

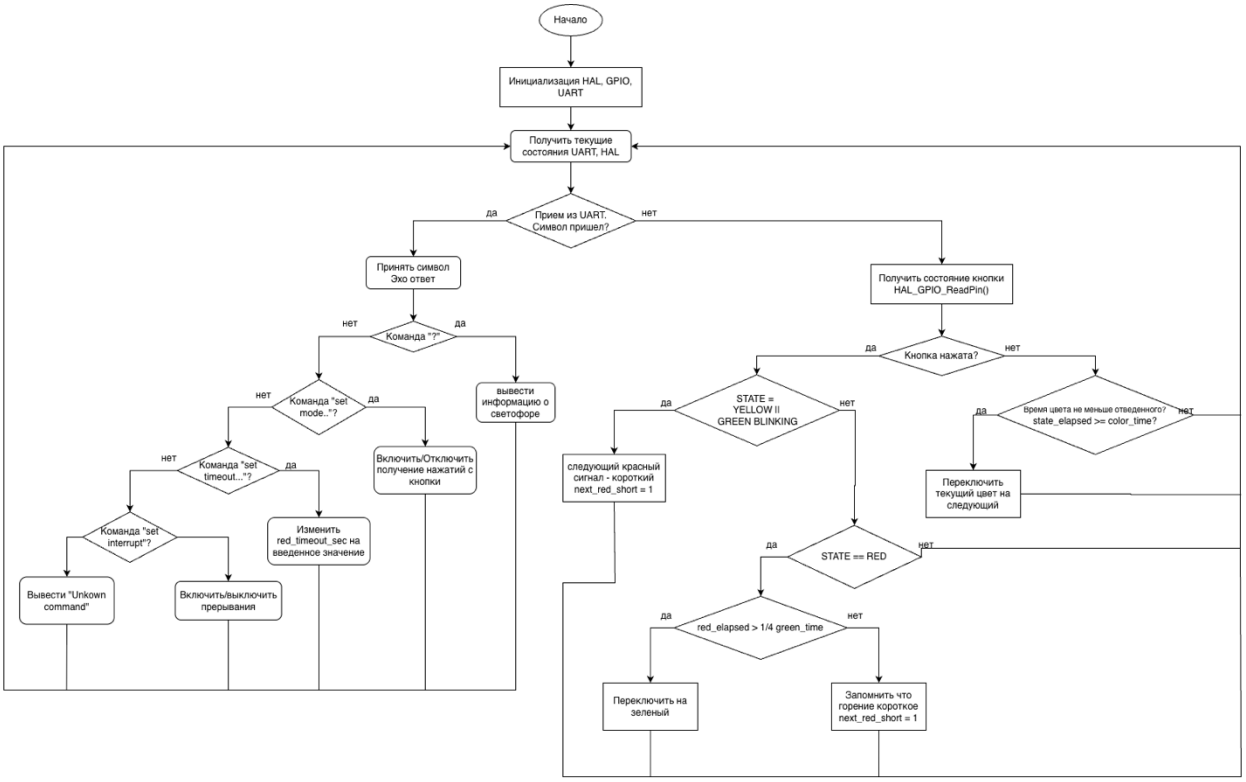

Рисунок 1. Блок-схема алгоритма

Исходный код

В последующем разделе приведён разбор только исходных файлов с расширением *.c*. Заголовочные файлы *.h* не рассматриваются, поскольку не содержат полезной информации, описания каких-либо структур или же они были рассмотрены выше.

uart_driver.c

```
#include <drivers/uart_driver.h>

extern UART_HandleTypeDef huart6;

static uint8_t rx_buf[UART_RX_BUF_SIZE];
static uint8_t tx_buf[UART_TX_BUF_SIZE];

static volatile uint16_t rx_head = 0, rx_tail = 0;
static volatile uint16_t tx_head = 0, tx_tail = 0;

static uint8_t rx_byte;
static uint8_t interrupts_enabled = 0;

static char s_line_buf[UART_MAX_LINE];
static uint16_t s_line_len = 0;

static inline uint32_t uart_critical_enter(void) {
    uint32_t pmask = __get_PRIMASK();
    __disable_irq();
    return pmask;
}

static inline void uart_critical_exit(uint32_t pmask) {
    __set_PRIMASK(pmask);
}

void uart_set_interrupts(uint8_t enabled)
{
    interrupts_enabled = enabled;
    if (enabled)
    {
        rx_head = rx_tail = 0;
        tx_head = tx_tail = 0;
            HAL_NVIC_EnableIRQ(USART6_IRQn);
        HAL_UART_Receive_IT(&huart6, &rx_byte, 1);
    }
    else
    {
        HAL_UART_AbortReceive_IT(&huart6);
        HAL_UART_AbortTransmit_IT(&huart6);
        HAL_NVIC_DisableIRQ(USART6_IRQn);
    }
}

void uart_transmit_char(char ch)
{
    if (!interrupts_enabled) {
```

```
        HAL_UART_Transmit(&huart6, (uint8_t *)&ch, 1, UART_TIMEOUT);
        return;
    }

    uint32_t pmask = uart_critical_enter();

    tx_buf[tx_head] = (uint8_t)ch;
    tx_head = (tx_head + 1) % UART_TX_BUF_SIZE;

      if (tx_head == tx_tail)
            tx_tail = (tx_tail + 1) % UART_TX_BUF_SIZE;

    uint8_t need_start = (__HAL_UART_GET_FLAG(&huart6, UART_FLAG_TXE) != 0)
&& (tx_head != tx_tail);

    uart_critical_exit(pmask);

    if (need_start) {
        HAL_UART_Transmit_IT(&huart6, &tx_buf[tx_tail], 1);
    }
}

void uart_transmit_string(char *str) {
     while (*str) uart_transmit_char(*str++);
}

void uart_transmitln_string(char *str)
{
     uart_transmit_string(str);
     uart_transmit_string("\r\n");
}

int uart_receive_char(char *ch)
{
    if (!interrupts_enabled) {
        return (HAL_UART_Receive(&huart6, (uint8_t *)ch, 1, 0) == HAL_OK) ? 1
: 0;
    }
    else
    {
        int is_byte_available = 0;
        uint32_t pmask = uart_critical_enter();

        if (rx_head != rx_tail) {
            *ch = rx_buf[rx_tail];
            rx_tail = (rx_tail + 1) % UART_RX_BUF_SIZE;
            is_byte_available = 1;
        }

        uart_critical_exit(pmask);
        return is_byte_available;
    }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance != USART6) return;

    rx_buf[rx_head] = rx_byte;
    rx_head = (rx_head + 1) % UART_RX_BUF_SIZE;
```

```
    if (rx_head == rx_tail)
        rx_tail = (rx_tail + 1) % UART_RX_BUF_SIZE;

    HAL_UART_Receive_IT(&huart6, &rx_byte, 1);
}

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance != USART6) return;
    if (!(tx_head == tx_tail))
    {
        tx_tail = (tx_tail + 1) % UART_TX_BUF_SIZE;
        if (!(tx_head == tx_tail))
            HAL_UART_Transmit_IT(&huart6, &tx_buf[tx_tail], 1);
    }
}

void uart_receive_line_task(void) {
    char c;
    while (uart_receive_char(&c)) {
      uart_transmit_char(c);
        switch (c) {
        case '\r':
        case '\n':
            if (s_line_len > 0) {
                s_line_buf[s_line_len] = '\0';
                uart_transmitln_string("");
                process_command_line(s_line_buf);
                s_line_len = 0;
            }
            break;
        case '\b':
        case 0x7F:
            if (s_line_len > 0) s_line_len--;
            break;
        default:
            if (isprint((unsigned)c) && s_line_len < UART_MAX_LINE - 1)
                s_line_buf[s_line_len++] = c;
        }
    }
}
```

process_comand.c:

```
#include "traffic_light_logic/command_process.h"

extern traffic_mode_t current_mode;
extern uint32_t red_timeout_sec;
extern traffic_light_state_t current_state;

void process_command_line(const char *line)
{
    char cmd[32];
    char arg1[32];
    char arg2[32];

    int count = sscanf(line, "%31s %31s %31s", cmd, arg1, arg2);

    if (count <= 0) {
      uart_transmitln_string("Empty command");
        return;
    }
```

```
    command_type = CMD_UNKNOWN;

    if (strcmp(cmd, "?") == 0)
        command_type = CMD_STATUS;
    else if (strcmp(cmd, "set") == 0 && count >= 2)
    {
        if (strcmp(arg1, "mode") == 0) command_type = CMD_SET_MODE;
        else if (strcmp(arg1, "timeout") == 0) command_type =
CMD_SET_TIMEOUT;
        else if (strcmp(arg1, "interrupts") == 0) command_type =
CMD_SET_INTERRUPTS;
    }

    switch (command_type) {

    case CMD_STATUS:
    {
        char buffer[64];
        char *color = "";

        switch (current_state) {
                case STATE_RED:             color = "red"; break;
                case STATE_GREEN:           color = "green"; break;
                case STATE_GREEN_BLINKING:  color = "blinking green"; break;
                case STATE_YELLOW:          color = "yellow"; break;
                default:                     color = "unknown"; break;
            }

        // Color
        uart_transmit_string("State: ");
        uart_transmitln_string((char *)color);

        // Mode
        uart_transmit_string("Mode: ");
        snprintf(buffer, sizeof(buffer), "%d", current_mode);
        uart_transmitln_string(buffer);

        // Timeout
        uart_transmit_string("Timeout (sec): ");
        snprintf(buffer, sizeof(buffer), "%lu", (unsigned
long)red_timeout_sec / 1000);
        uart_transmitln_string(buffer);

        // Interruption
        uart_transmit_string("Interruption: ");
        snprintf(buffer, sizeof(buffer), "%c", uart_get_interrupts() ? 'I' :
'P');
        uart_transmitln_string(buffer);

        break;
    }
    case CMD_SET_MODE:
    {
        if (count < 3) {
            uart_transmitln_string("Usage: set mode 1|2");
            break;
        }
        int mode = atoi(arg2);
        if (mode == 1 || mode == 2) {
            current_mode = mode;
```

```
            uart_transmitln_string("OK");
        } else {
            uart_transmitln_string("ERROR: Invalid mode. Usage: set mode
1|2");
        }
        break;
    }
    case CMD_SET_TIMEOUT: {
        if (count < 3) {
            uart_transmitln_string("Usage: set timeout X");
            break;
        }
        int t = atoi(arg2) * 1000;
        if (t > 0 && t > RED_TIME_SHORT_MS) {
            red_timeout_sec = (uint32_t)t;
            uart_transmitln_string("OK");
        } else {
            uart_transmitln_string("ERROR: Timeout must be more than
RED_TIME_SHORT");
        }
        break;
    }
    case CMD_SET_INTERRUPTS: {
        if (count < 3) {
            uart_transmitln_string("Usage: set interrupts on|off");
            break;
        }
        if (strcmp(arg2, "on") == 0) {
            uart_set_interrupts(1);
            uart_transmitln_string("Interrupts ON");
        } else if (strcmp(arg2, "off") == 0) {
            uart_set_interrupts(0);
            uart_transmitln_string("Interrupts OFF");
        } else {
            uart_transmitln_string("ERROR: Invalid parameter (use on|off)");
        }
        break;
    }
    default:
      uart_transmitln_string("Unknown command");
        break;
    }
}
```

С полной версией программы можно ознакомиться, перейдя на GitHub-репозиторий, расположенный по ссылке:

https://github.com/kkettch/design-of-computing-systems-semester-7

Вывод

В ходе выполнения работы были успешно разработаны и реализованы два варианта драйверов для UART на стенде SDK-1.1M, а также доработана программа «светофор» с расширенным функционалом управления через UART-интерфейс.

Драйвер с использованием прерываний обеспечил неблокирующий режим работы с буферизацией данных, что исключило потерю символов при интенсивном обмене. Были реализованы кольцевые буферы для приёма и передачи, обработчики прерываний для их заполнения и опустошения, а также функции асинхронной отправки и чтения данных.