Федеральное государственное автономное образовательное учреждение высшего образования

Университет ИТМО

Дисциплина: Проектирование вычислительных систем

Лабораторная работа 2

Вариант 4

Выполнили:
Марков Петр Денисович
Кривоносов Егор Дмитриевич

Группа: P34111

Преподаватель:
Пинкевич Василий Юрьевич

2022 г.

Санкт-Петербург

Оглавление

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.

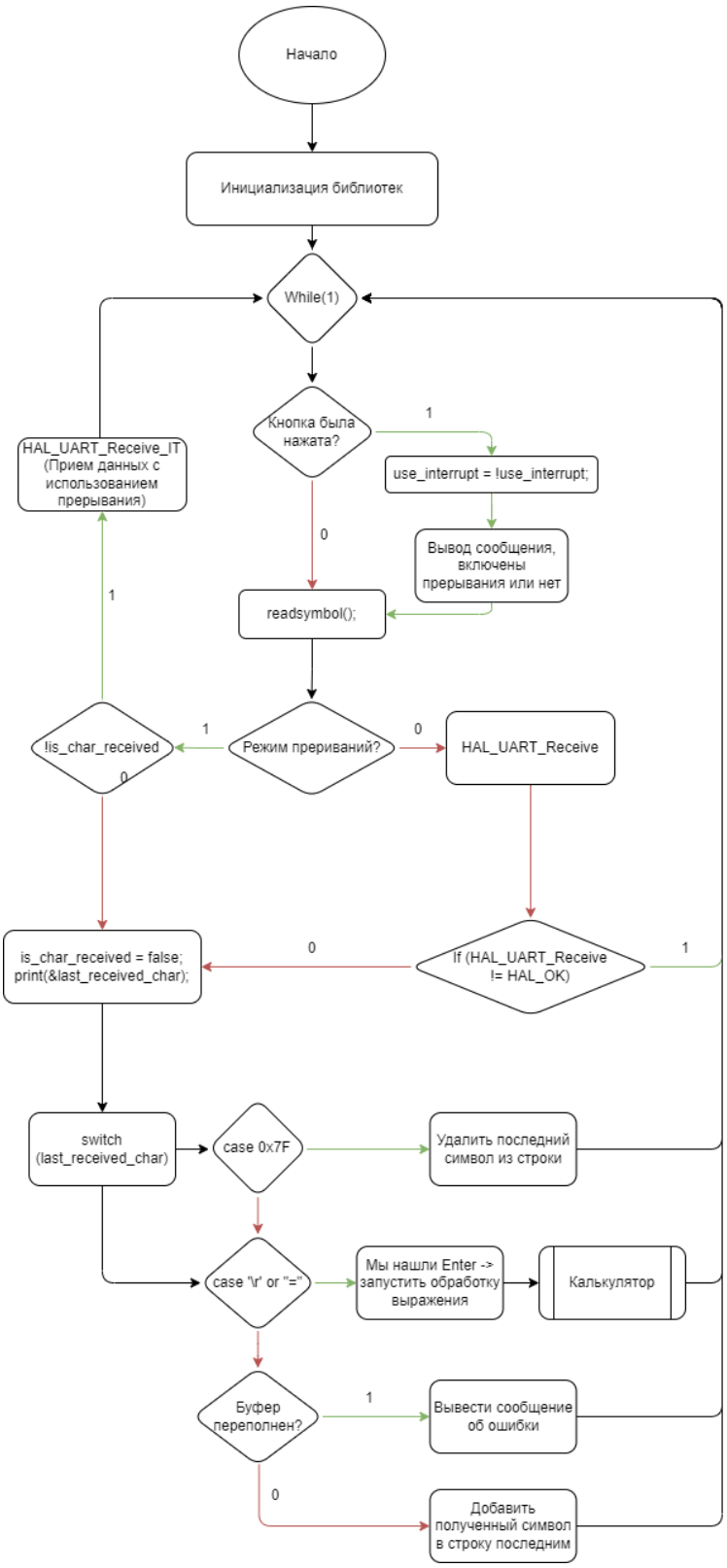
Вариант 4

Разработать программу-калькулятор. Ввод значений производится с компьютера через UART: xx…xyxx…x=, где x – десятичные цифры, y – 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).

Блок-схемы


Начало
Инициализация библиотек
While(1)
Кнопка была нажата?
1
use_interrupt = !use_interrupt;
Вывод сообщения, включены прерывания или нет
0
readsymbol();
HAL_UART_Receive_IT (Прием данных с использованием прерывания)
1
!is_char_received
1
Режим прерываний?
0
HAL_UART_Receive
0
is_char_received = false; print(&last_received_char);
0
If (HAL_UART_Receive != HAL_OK)
1
switch (last_received_char)
case 0x7F
Удалить последний символ из строки
case '\r' or '='
Мы нашли Enter -> запустить обработку выражения
Калькулятор
Буфер переполнен?
1
Вывести сообщение об ошибки
0
Добавить полученный символ в строку последним

Описание работы алгоритма

В нашей программе есть 2 режима работы - с прерываниями или без них. При работе с ними мы используем HAL UART Recieve при вводе чисел, иначе мы просто проверяем по таймауту. Затем мы делаем проверку каждого символа и проверяем размер вводимого числа и уникальные командные символы (=, + и тд). При возникновении некорректных данных выводится error, иначе - происходят вычисления и выводится результат.

Исходный код

```
struct RingBuffer {
  char data[BUF_SIZE];
  uint8_t head;
  uint8_t tail;
  bool empty;
};

typedef struct RingBuffer RingBuffer;

static void buf_init(RingBuffer *buf) {
  buf->head = 0;
  buf->tail = 0;
  buf->empty = true;
}

static void buf_push(RingBuffer *buf, char *el) {
  uint64_t size = strlen(el);

  if (buf->head + size + 1 > BUF_SIZE) {
    buf->head = 0;
  }

  strcpy(&buf->data[buf->head], el);
  buf->head += size + 1;

  if (buf->head == BUF_SIZE) {
    buf->head = 0;
  }

  buf->empty = false;
}

static bool buf_pop(RingBuffer *buf, char *el) {
  if (buf->empty) {
    return false;
  }

  uint64_t size = strlen(&buf->data[buf->tail]);
```

```
  strcpy(el, &buf->data[buf->tail]);
  buf->tail += size + 1;

  if (buf->tail == BUF_SIZE || buf->tail == '\0') {
    buf->tail = 0;
  }

  if (buf->tail == buf->head) {
    buf->empty = true;
  }

  return true;
}

static struct RingBuffer ringBuffer;
static struct RingBuffer ringBufferTx;

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

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

static void set_green_led(bool on) { HAL_GPIO_WritePin(GPIOD, GPIO_PIN_13, on ? GPIO_PIN_SET
: GPIO_PIN_RESET); }

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

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


struct ButtonState {
    bool is_pressed;
    bool signaled;
    uint32_t press_start_time;
};

static bool update_button_state(struct ButtonState *state) {
    if (state->is_pressed) {
        state->is_pressed = is_button_active();

        if (state->signaled) {
            return false;
        }

        if ((HAL_GetTick() - state->press_start_time) > 20 /* ms */) {
            state->signaled = true;
            return true;
        }
        return false;
    }
```

```
    if (is_button_active()) {
        state->press_start_time = HAL_GetTick();
        state->is_pressed = true;
        state->signaled = false;
    }

    return false;
}

struct Status {
    bool interrupt_enable;
    uint32_t pmask;
};

static struct Status status;

bool transmit_busy = false;

void enable_interrupt(struct Status *status) {
    HAL_NVIC_EnableIRQ(USART6_IRQn);
    status->interrupt_enable = true;
}

void disable_interrupt(struct Status *status) {
    HAL_UART_AbortReceive(&huart6);
    HAL_NVIC_DisableIRQ(USART6_IRQn);
    status->interrupt_enable = false;
}

void transmit_uart(const struct Status *status, char *buf, size_t size) {
  if (status->interrupt_enable) {
    if (transmit_busy) {
      buf_push(&ringBufferTx, buf);
    } else {
      HAL_UART_Transmit_IT(&huart6, buf, size);
      transmit_busy = true;
    }
    return;
  }
  HAL_UART_Transmit(&huart6, buf, size, 100);
}

void transmit_uart_nl(const struct Status *status, char *buf, size_t size) {
  transmit_uart(status, buf, size);
  transmit_uart(status, "\r\n", 2);
}

void receive_uart(const struct Status *status) {
  if (status->interrupt_enable) {
    HAL_UART_Receive_IT(&huart6, el, sizeof(char));
    return;
  }
  HAL_StatusTypeDef stat = HAL_UART_Receive(&huart6, el, sizeof(char), 0);
  switch (stat) {
  case HAL_OK: {
```

```
    buf_push(&ringBuffer, el);
    transmit_uart(status, el, 1);
    break;
  }
  case HAL_ERROR:
  case HAL_BUSY:
  case HAL_TIMEOUT:
    break;
  }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
  buf_push(&ringBuffer, el);
  transmit_uart(&status, el, 1);
}

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
  char buf[1024];
  if (buf_pop(&ringBufferTx, buf)) {
    HAL_UART_Transmit_IT(&huart6, buf, strlen(buf));
  } else {
    transmit_busy = false;
  }
}

enum ValueDefenition {
    LeftValue,
    LeftValueWithOperand,
    RightValue,
    RightValueWithEquals,
    ResultValue,
    ErrorValue
};

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

  HAL_Init();

  SystemClock_Config();

  MX_GPIO_Init();
  MX_USART6_UART_Init();

  char interrupt_enabled_msg[] = {"Interrupts ON"};
  char interrupt_disabled_msg[] = {"Interrupts OFF"};
  char error_msg[] = {"\r\nerror\n"};

  struct ButtonState buttonState = {.press_start_time = 0, .signaled = false, .is_pressed =
false};

    enable_interrupt(&status);
    buf_init(&ringBuffer);
```

```
    uint32_t arg1 = 0;
    uint32_t arg2 = 0;
    int64_t res = 0;
    char res_str [8];
    char op;
    enum ValueDefenition exprState = LeftValue;
    set_red_led(false);

    while (1) {
        if (update_button_state(&buttonState)) {
                    if (status.interrupt_enable) {
                        disable_interrupt(&status);
                        transmit_uart_nl(&status, interrupt_disabled_msg,
sizeof(interrupt_disabled_msg));
                    } else {
                        enable_interrupt(&status);
                        transmit_uart_nl(&status, interrupt_enabled_msg,
sizeof(interrupt_enabled_msg));
                    }
                }

                receive_uart(&status);

                char c[2];

                if ((exprState != ResultValue && exprState != ErrorValue) &&
!buf_pop(&ringBuffer, c)) {
                    continue;
                }

                switch (exprState) {
                    case LeftValue: {
                        if (!isdigit(c[0])) {
                            exprState = ErrorValue;
                            break;
                        }
                        exprState = LeftValueWithOperand;
                        set_red_led(false);
                        arg1 = arg1 * 10 + (c[0] - 48);
                        uint16_t new_arg1 = (uint16_t) arg1;
                        if (arg1 != new_arg1) {
                            exprState = ErrorValue;
                        }
                        break;
                    }
                    case LeftValueWithOperand: {
                        if (!isdigit(c[0])) {
                            switch (c[0]) {
                                case '+':
                                case '-':
                                case '*':
                                case '/': {
                                    op = c[0];
                                    exprState = RightValue;
                                    break;
```

```
                }
                default:
                    exprState = ErrorValue;
                    break;
            }
            break;
        }
        arg1 = arg1 * 10 + (c[0] - 48);
        uint16_t new_arg1 = (uint16_t) arg1;
        if (arg1 != new_arg1) {
            exprState = ErrorValue;
        }
        break;
    }
    case RightValue: {
        if (!isdigit(c[0])) {
            exprState = ErrorValue;
            break;
        }
        exprState = RightValueWithEquals;
        arg2 = arg2 * 10 + (c[0] - 48);
        uint16_t new_arg2 = (uint16_t) arg2;
        if (arg2 != new_arg2) {
            exprState = ErrorValue;
        }
        break;
    }
    case RightValueWithEquals: {
        if (!isdigit(c[0])) {
            if (c[0] == '=') {
                exprState = ResultValue;
            } else {
                exprState = ErrorValue;
            }
            break;
        }
        arg2 = arg2 * 10 + (c[0] - 48);
        uint16_t new_arg2 = (uint16_t) arg2;
        if (arg2 != new_arg2) {
            exprState = ErrorValue;
        }
        break;
    }
    case ResultValue: {
        int32_t arg1_c = arg1;
        int32_t arg2_c = arg2;
        switch (op) {
            case '+':
                res = arg1_c + arg2_c;
                break;
            case '-':
                res = arg1_c - arg2_c;
                break;
            case '*':
                res = arg1_c * arg2_c;
                break;
```

```
                        case '/':{
                            if (arg2_c == 0) {
                                exprState = ErrorValue;
                                continue;
                            } else {
                                res = arg1_c / arg2_c;
                            }
                            break;
                        }
                        default:
                            break;
                    }

                    int16_t actual_res = res;
                    if (actual_res != res) {
                        exprState = ErrorValue;
                        break;
                    }

                    sprintf(res_str, "%d", res);
                    transmit_uart_nl(&status, res_str, strlen(res_str));
                    arg1 = 0;
                    arg2 = 0;
                    res = 0;
                    exprState = LeftValue;
                    break;
                }
                case ErrorValue: {
                    arg1 = 0;
                    arg2 = 0;
                    res = 0;
                    set_red_led(true);
                    transmit_uart_nl(&status, error_msg, sizeof(error_msg));
                    exprState = LeftValue;
                    break;
                }
                default:
                    break;
            }
    }
}
```

Вывод

В ходе выполнения лабораторной работы мы реализовали простейший калькулятор через клавиатуру, используя интерфейс UART в режиме прерываний и без, а также с использованием кольцевого буфера. Сигнализировали о неверном результате подсчетов с помощью красного светодиода.