Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное образовательное учреждение

высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчет

по лабораторной работе №3

по дисциплине «**Проектирование вычислительных систем**»

Вариант 3

Авторы: Сенина М.М.

Никонова Н.И.

Факультет: ПИиКТ

Группа: P34102

Преподаватель: Пинкевич В.Ю.

Санкт-Петербург, 2023

Цель работы

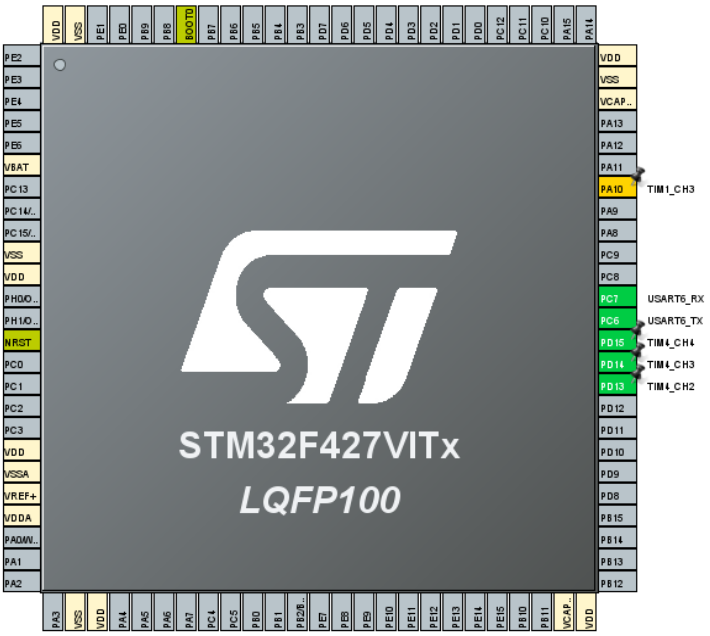
1. Получить базовые знания об устройстве и режимах работы таймеров в микроконтроллерах.
2. Получить навыки использования таймеров и прерываний от таймеров.
3. Получить навыки использования аппаратных каналов ввода-вывода таймеров.

Задачи

Разработать программу, которая использует таймеры для управления яркостью светодиодов и излучателем звука (по прерыванию или с использованием аппаратных
каналов). Блокирующее ожидание (функция HAL_Delay()) в программе использоваться не должно.
Стенд должен поддерживать связь с компьютером по UART и выполнять указанные
действия в качестве реакции на нажатие кнопок на клавиатуре компьютера. В данной
лабораторной работе каждая нажатая кнопка (символ, отправленный с компьютера на стенд)
обрабатываются отдельно, ожидание ввода полной строки не требуется.
Для работы с UART на стенде можно использован один из двух вариантов драйвера (по прерыванию и по опросу) на выбор исполнителя. Поддержка двух вариантов не требуется.

Реализация

Используемые контакты



- **PD13 (TIM4_CH2)** – на 2ом канале стоит PWM 4го таймера. Он управляет зеленым светодиодом.
- **PD14 (TIM4_CH3)** – на 3ем канале стоит PWM 4го таймера. Управляет желтым светодиодом.
- **PD15 (TIM4_CH4)** – на 4ом канале стоит PWM 4го таймера. Управляет красным светодиодом.
- **PA10 (TIM1_CH3)** – получает прерывания от таймера
- **PC7 (USART_RX)** – получает прерывание RX о том, что uart завершил прием данных
- **PC6 (USART_TX)** – получает прерывание TX о том, что uart завершил отправку данных
- **PB3, PB4, PA13, PA14, PA15** – J-Tag для отладки

Описание алгоритма

Общая идея воспроизведения режимов гирлянды

Для воспроизведения режимов гирлянды мы по прерываниям таймера включаем или выключаем светодиодны на нужные нам цвета. Для конфигурации того, как сейчас должен себя вести светодиод мы используем два массива конфигураций:

```
Tick green[TICK_BUFF_SIZE] = {};
Tick red_yellow[TICK_BUFF_SIZE] = {}
```

Это массивы из структур, относящихся либо к светодиоду горящему красными или желтым, либо к светодиоду горящему зелёным.

```
typedef struct {
 uint8_t color; // 0 для красного и зелёного, 1 для жёлтого
 uint8_t duration; // для свечения в такте 0..9
} Tick;
```

`color` описывает каким цветом должен гореть светодиод, а `duration` говорит, что светодиод должен гореть `duration/10` долю такта. Это реализуется через запись регистра `CCR` в таймере, чей канал привязан к этому светодиоду. Чем большее значение записано в этот регистр тем ярче будет гореть светодиод. Если светодиод гореть в этом тике не должен мы записываем в этот регистр 0.

Чтобы переключать режимы мы копируем массивы-режимы из структуры-коллекции режимов `modes` в массивы `green` и `red_yellow`. Соответственно в режимах по умолчанию мы используем заранее заготовленные массивы инициируемые функцией void init_modes(Mode *modes); . Пользовательский режим мы заполняем на лету.

Коллекция режимов хранится в глобальной переменной:
```
Mode modes[MODES_COUNT] = {0};
```

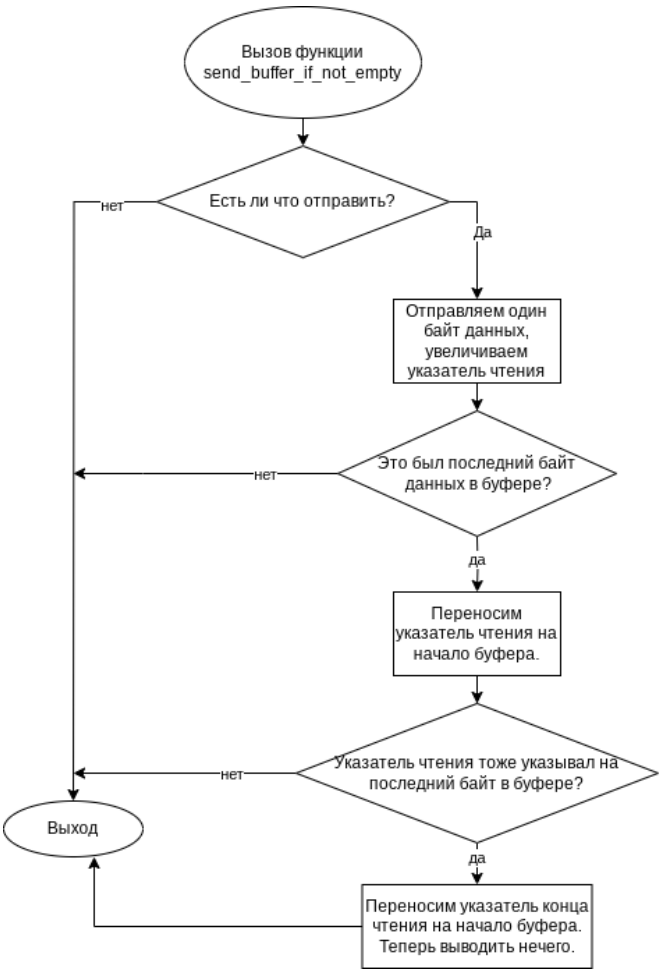
В каждом элементе этого массива находится структура, в которой хранятся массивы, которые надо скопировать в, для того чтобы они начали производиться.

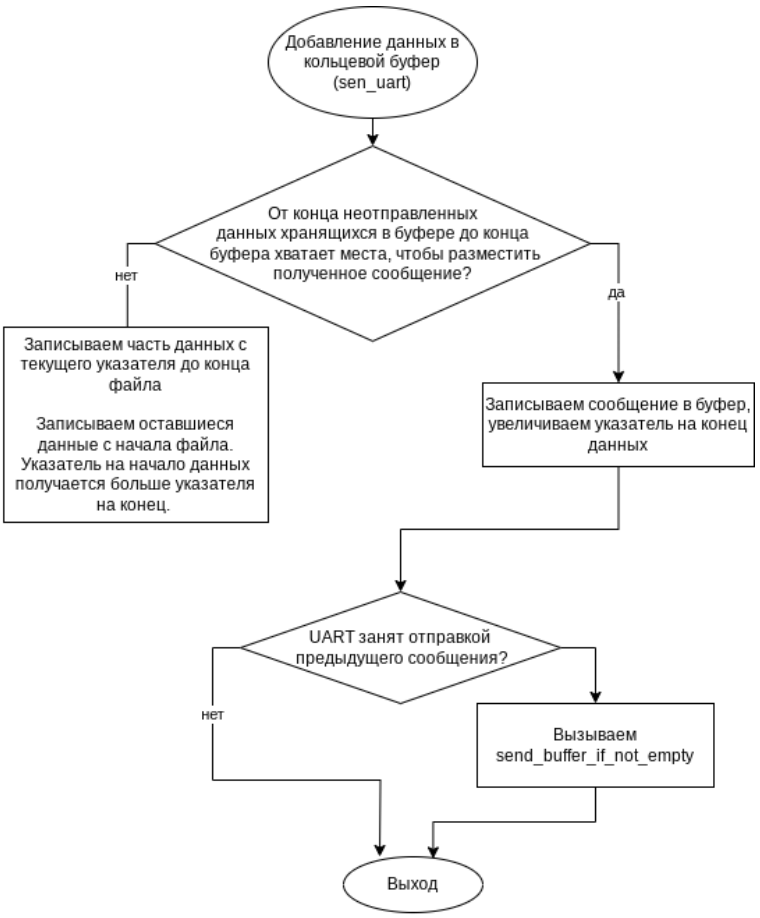
```
typedef struct {
 Tick *green;
 Tick *red_yellow;
 uint8_t len;
} Mode;
```

Модуль ввода-вывода UART

За счёт функций этого модуля мы общались с пользователем через протокол USART. В этом модуле всего 4 функции:

1. Добавление данных в кольцевой буфер на отправку.
2. Отправка сообщения по USART в режиме прерываний.
3. Обработчик прерываний отправки сообщения по USART.
4. Обработчик прерываний получения сообщения по USART.





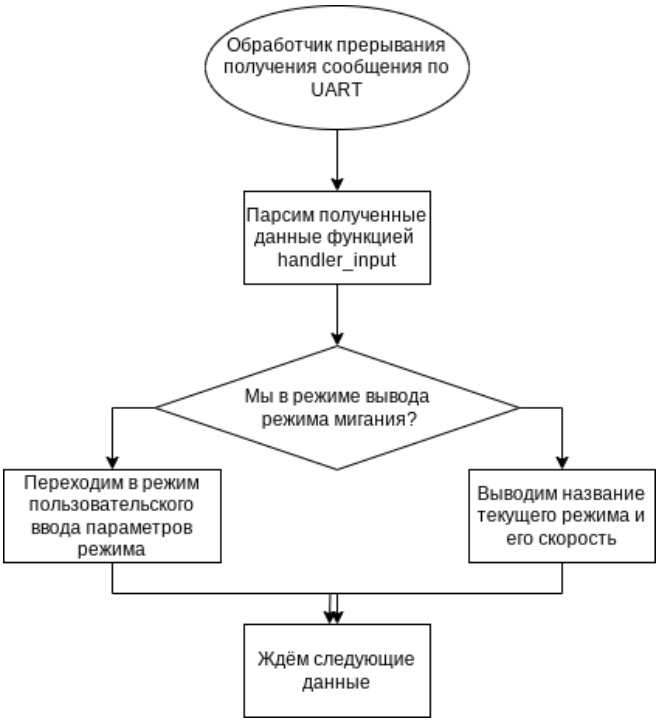

```
void send_buffer_if_not_empty_IT(UART_HandleTypeDef *huart){
 if (start_write != end_write) {
   HAL_UART_Transmit_IT(huart, (uint8_t*) (buffer_to_write + start_write), 1);
   start_write++;
   if (start_write == WRITE_BUFFER_SIZE) {
     start_write = 0;
     if (end_write == WRITE_BUFFER_SIZE) {
       end_write = 0;
     }
   }
 }
}
```

```
void send_uart(UART_HandleTypeDef *huart, uint8_t* buffer, size_t buf_size) {
 char state = HAL_UART_GetState(huart);
 if (buf_size > WRITE_BUFFER_SIZE - end_write) {
   size_t first_size = WRITE_BUFFER_SIZE - end_write;
   if (first_size > 0) {
     memcpy(buffer_to_write + end_write, buffer, first_size);
   }
   memcpy(buffer_to_write, buffer, buf_size - first_size);
   end_write = buf_size - first_size;
 } else {
   memcpy(buffer_to_write + end_write, buffer, buf_size);
   end_write += buf_size;
 }
```

```
 if (state != HAL_UART_STATE_BUSY_TX) {
   send_buffer_if_not_empty_IT(huart);
 }
}
```

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
 if (huart->Instance == USART6)
 {
   handler_input();
   if (output == 1) {
     bzero(middle_buffer,MIDDLE_BUFFER_SIZE);
     sprintf((char*) middle_buffer,
              "mode: %d, speed: %ld", mode,
       scaler_speed);
     send_uart(huart, middle_buffer,
              strlen((char*)middle_buffer));
   } else {
     send_user_mode_param();
   }
   HAL_UART_Receive_IT(huart, (uint8_t*)
&read_buffer, 1);
 }
}

void HAL_UART_TxCpltCallback(UART_HandleTypeDef
*huart){
 if (huart->Instance == USART6)
 {
   send_buffer_if_not_empty_IT(huart);
 }
}
```



Модуль управления переключения режимами

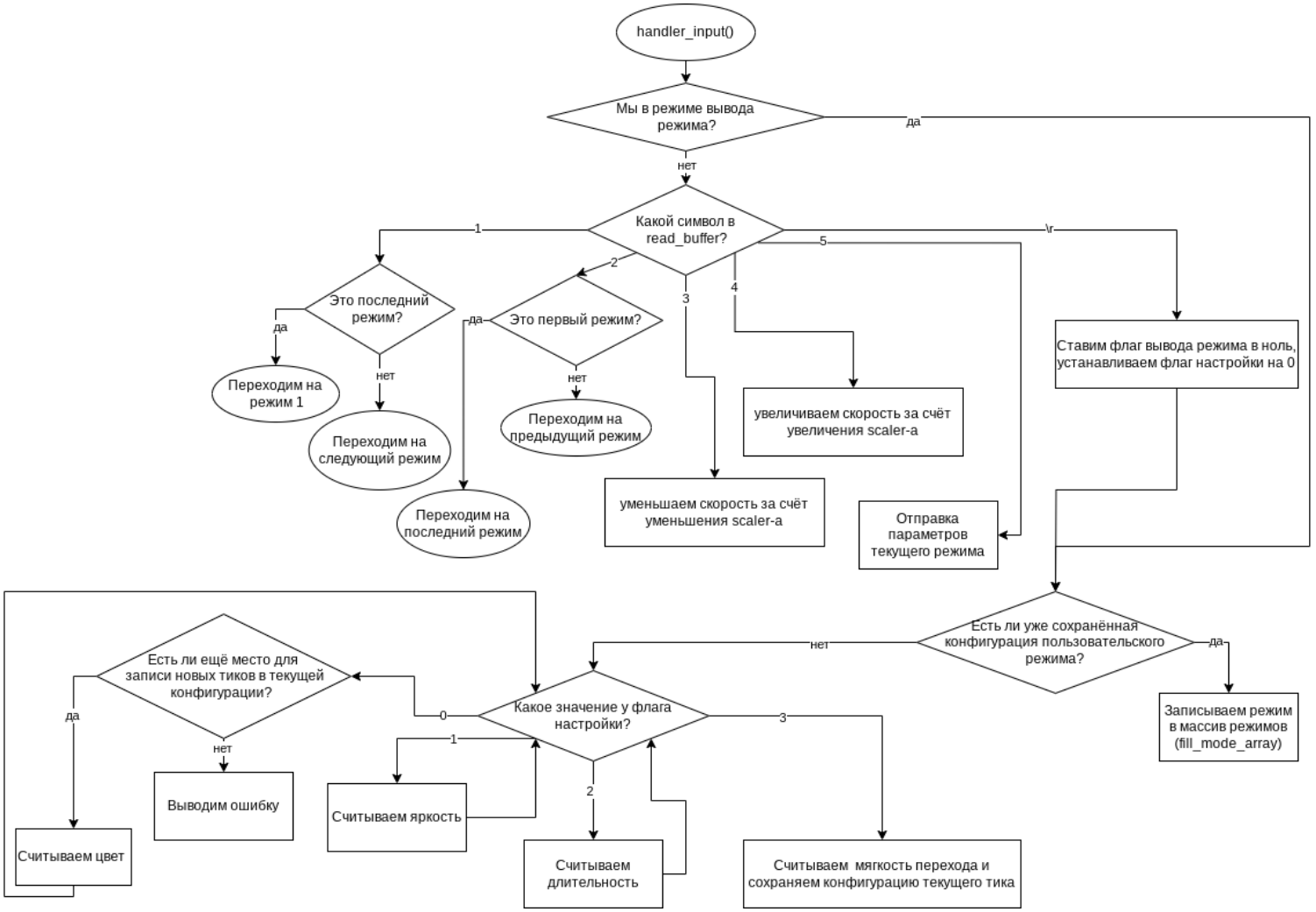
В этом модуле мы на основе данных, полученных по UART и записанных в буфер `read_buffer` переключаемся между режимами, увеличиваем скорость воспроизведения или выводим параметры текущего режима.

Функция работает на switch, который в зависимости от значений флагов выбирает разное поведение:

1. Переход к следующему режиму по циклу
2. Переход к предыдущему режиму по циклу
3. Увеличение скорости
4. Уменьшение скорости
5. Распечатка параметров текущего состояния
6. Переход к настройке пользовательского режима

Пункты 1-5 вроде должны быть понятны из описания.

В пункте 6 мы должны получить от пользователя конфигурацию пользовательского режима, мы это делаем заполняя массив данных о тиках полученных от пользователя (`Input_tick`), а потом конвертируем их в (`Tick`) используемые при выводе значений. Делаем это точно так же с помощью switch выбираем, какое значение сейчас вводит пользователь и пытаемся его прочитать.




```
void handler_input() {
 if (output == 1) {
   switch (read_buffer) {
   case '1':
     // to next mode
     if (mode == 5) {
       mode = 1;
     } else {
       mode++;
     }
     play_new_mode(&modes[mode - 1], green, red_yellow, &writing_ptr, &current_write_ptr);
     break;
   case '2':
     // to previos mode
     if (mode == 1) {
       mode = 5;
     } else {
       mode--;
     }
```

```
      play_new_mode(&modes[mode - 1], green, red_yellow, &writing_ptr, &current_write_ptr);
      break;
    case '3':
      // faster
      new_scaler_speed = scaler_speed - round((double) scaler_speed / 10);
      if (new_scaler_speed == scaler_speed) {
        new_scaler_speed--;
      }
      if (new_scaler_speed > min_scaler) {
        scaler_speed = new_scaler_speed;
      }
      break;
    case '4':
      // slower
      new_scaler_speed = scaler_speed + round((double) scaler_speed / 10);
      if (new_scaler_speed == scaler_speed) {
        new_scaler_speed++;
      }
      if (new_scaler_speed < max_scaler) {
        scaler_speed = new_scaler_speed;
      }
      break;
    case '5':
      // send current user mode
      send_user_mode_param();
      break;
    case '\r':
      // ввод в меню настройки
      output = 0;
      tick_len = 0;
      break;
    default:
      break;
    }
  } else {
    if (read_buffer == '\r') {
      if (tick_len != 0) {
        fill_mode_array(tick_buffer, tick_len, &modes[MODES_COUNT - 1]);
        tick_ptr = 0;
      }
      output = 1;
    } else {
      switch (tick_ptr) {
        case 0:
          if (tick_len == TICK_BUFF_SIZE) {
            bzero(middle_buffer, MIDDLE_BUFFER_SIZE);
            sprintf((char*) middle_buffer, "user mode buffer is full!");
            send_uart(&huart6, middle_buffer, strlen((char*)middle_buffer));
          } else {
            if (read_buffer == 'r' || read_buffer == 'g' || read_buffer == 'y') {
              tick_ptr++;
              cur_read_tick.color = read_buffer;
            }
          }
          break;
        case 1:
          if (read_buffer >= '1' && read_buffer <= '9'){
```

```
            tick_ptr++;
            cur_read_tick.brightness = read_buffer - '0';
          }
          break;
        case 2:
          if (read_buffer >= '1' && read_buffer <= '9'){
            tick_ptr++;
            cur_read_tick.duration = read_buffer - '0';
          }
          break;
        case 3:
          if (read_buffer == '+') {
            fill_softnes_and_save_input_tick(1);
          } else if (read_buffer == '-') {
            fill_softnes_and_save_input_tick(0);
          }
          break;
        default:
          tick_ptr = 0;
          break;
      }
    }
  }
}
```

Конвертация Input_tick-ов в Tick

Эта конвертация делается несколькими функциями в файле girland.c. Там есть функция-интерфейс вызываемая в handler_input(). Функция:

```
fill_mode_array(tick_buffer, tick_len, &modes[MODES_COUNT - 1]);
```

Она получает на вход массив “сырых” тиков, которые нужно конвертировать и записывает конвертированные тики в массив-коллекцию modes.

Внутри там для каждого тика вызывается функция-конвертер отвечающая за то, чтобы обеспечить правильную длительность горения светодиода и плавный переход к другому значению, если это было указано пользователем.

Плавный переход реализуется за счёт плавного наращивания значений duration записываемых для этого тика в итоговый массив.

Т.е. например, подряд идут два тика, где указаны следующий параметры:

{softness=0 и brightness=3, a duration=5} и {softness=1 и brightness=6, a duration=3}

Такие данные конвертируются в тики следующей длины:

Tick1: [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3]

Потому что мягкого перехода не требуется, а количество тиков получается по формуле: `duration * DEFAULT_LIGHT_DURATION`, где `DEFAULT_LIGHT_DURATION` = 5

Но в следующем тике, мы видим, что в предыдущем значение яркости было на 3 меньше и нам надо плавно перейти к этому значению. Соответственно, первая половина тика будет отведена под плавный переход.

Tick2: [3,4,5,6,6,6,6,6,6,6,6,6,6,6,6]

Рассчёт того, какая разница между яркостями соседних тиков, если softness стоит в 1 производится в `fill_mode_array`.

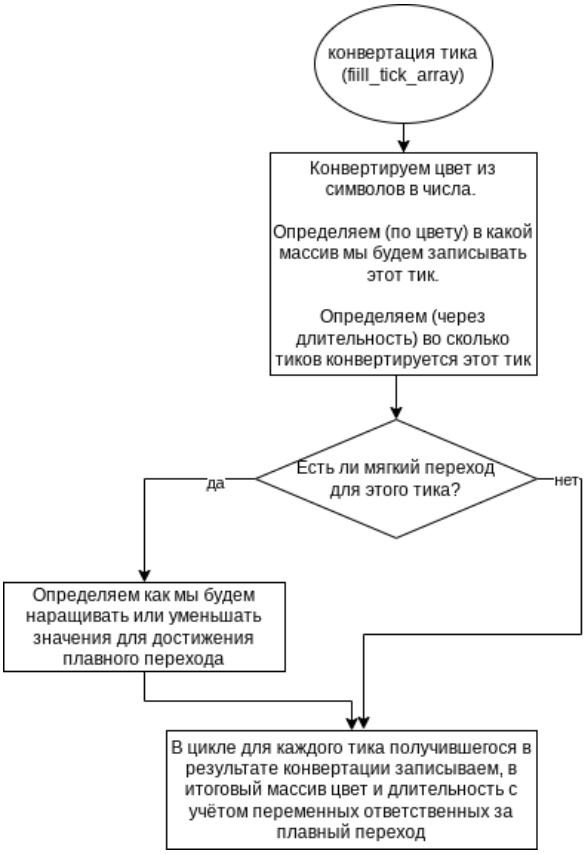
```
void fill_tick_array(char color, uint8_t brightness, uint8_t duration,
                     int8_t softness, Mode *mode) {
 uint8_t new_color;
 Tick *other_buffer;
 Tick *buffer = define_color(color, &new_color,
&other_buffer, mode);
 uint8_t n = duration * DEFAULT_LIGHT_DURATION; //
количество тиков, которое
                                                //
будет занимать этот цвет
 int each = 1;
 int left = abs(softness - (n / 2));
 int left_v = 0;

 if (softness > 0) {
   left_v = 1;
   if (abs(softness / (n / 2)) > 1) {
     each = 1;
   }
 }

 if (softness < 0) {
   each = -each;
   left_v = -left_v;
 }

 int cur_bright = brightness - softness;
 for (int i = 0; i < n; i++) {
   if (i < (n / 2)) {
     if (i < (n / 2) && abs(brightness - cur_bright)
>= abs(each)) {
       cur_bright += each;
     }

     if (abs(brightness - cur_bright) >= abs(left_v) && left > 0) {
       cur_bright += left_v;
       left--;
     }
   }
   fill_tick(&buffer[i + mode->len], new_color, cur_bright);
   zero_tick(&other_buffer[i + mode->len]);
 }
 mode->len += n;
}
```



Заключение

В ходе выполнения этой лабораторной работы мы попробовали использовать таймеры PWM, чтобы регулировать яркость горения светодиода. Разобрались с их настройкой. Мы спроектировали систему мигающую светодиодами по заданному режиму и позволяющую этот режим менять.