

# Умный город

## 1 Состояние кнопки

На микроконтроллере есть *кнопка*, с которой считывается *дискретный* сигнал:

- 1) Сервер запрашивает с некоторой частотой данные у микроконтроллера.
- 2) Микроконтроллер отправляет *бит состояния* кнопки по протоколу TCP в среде Wi-Fi на сервер.
- 3) Web-сервер посылает *GET-запрос* состояния кнопки на сервер, дожидается ответа.
- 4) Web-сайт отображает состояние кнопки (*ON/OFF*).

### Дополнительно:

- *динамическое* обновление состояния кнопки
- считывать состояния *нескольких* кнопок
- перевести кнопку в состояние *switch*
- хранить состояния кнопки в *log-файле*
- *графическое* отображение состояния кнопки

## 1.1 Fetch API

Взаимодействие с web-сервером происходит с помощью *HTTP-запросов*.

**Fetch API** — функция JavaScript, которая помогает web-странице посылать такие запросы на web-сервер.

К примеру, без *GET-запроса* web-сервер отправлял *целый HTML-файл* с форматированным значением целевой переменной, чтобы обновить информацию на web-странице.

Чтобы сэкономить трафик, достаточно отправлять лишь *значение целевой переменной* в теле *GET-запроса*, которое будет форматировано на самом клиенте.

## 2 Дискретный светодиод

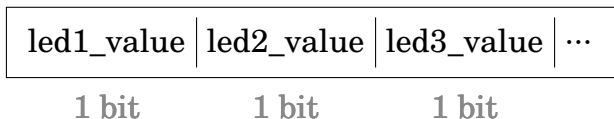
На микроконтроллере есть *светодиод*, который управляется *дискретным* сигналом:

- 1) Микроконтроллер с определённой частотой запрашивает *управляющий бит* у сервера.
- 2) Web-сайт по нажатию на кнопку отправляет *POST-запрос* с управляющим битом на сервер.

### Дополнительно:

- посылать состояние *нескольким* светодиодам
- включить в тело HTTP-пакета информацию в *JSON*
- хранить состояния светодиода в *log-файле*

### Структура TCP-пакета:



## 2.1 JSON

**JSON** — текстовый формат обмена данными, основанный на JavaScript. Чаще всего применяется в *RESTful-сервисах*.

**REST API** — это способ взаимодействия web-приложений с сервером, для которого характерно чёткое разделение между функциями сервера и клиента:

- *сервер* — код для доступа к данным и их обработки;
- *клиент* — пользовательский интерфейс.

**Масштабируемость** — главное преимущество.

## 2.2 PWM-модуляция

Рассказать здесь про `ledcSetup()` и прочее.

## 3 Аналоговый светодиод

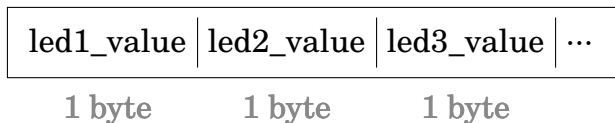
На микроконтроллере есть *светодиод*, который управляется *аналоговым* сигналом:

- 1) Микроконтроллер с определённой частотой запрашивает *управляющий бит* у сервера.
- 2) Web-сайт по нажатию на кнопку отправляет *POST-запрос* с JSON на сервер.

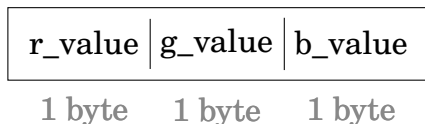
## Дополнительно:

- заменить обычный светодиод на *rgb-светодиод*
- хранить состояния светодиода в *log-файле*

**Структура** TCP-пакета для *обычных* светодиодов:



**Структура** TCP-пакета для *rgb-светодиода*:



## 3.1 Task

**Структура** создания таска:

```
TaskHandle_t task_h;  
void function(void *params) {...}  
xTaskCreatePinnedToCore(function, // function  
                        "Name",   // name  
                        10000,     // stack size  
                        NULL,      // func params  
                        1,         // priority  
                        &task_h,   // task_handle  
                        0);        // core_id
```

Точный размер **стека** можно узнать по функции, вызванной в таске:

```
uxTaskGetStackHighWaterMark(NULL)
```

Вместо *NULL* можно передать *task\_handler*, а саму функцию вызвать из любого места.

В *Arduino* стандартную функцию *loop()* нужно оставить *пустой* при использовании тасков, потому что она тоже является таском.

Полезно добавить в теле этой функции `vTaskDelay(100)`, что сократит время одной итерации цикла.

Другая функция — `vTaskDelete(NULL)` — удалит этот цикл, однако вместе с ним и код, который вызывал `setup()`.

## 3.2 Mutex

**Mutex** (*Mutual Exclusion*) — примитив синхронизации, который обеспечивает взаимно исключающий доступ к общему ресурсу:

```
SemaphoreHandle_t mutex = xSemaphoreCreateMutex();  
...  
// enter critical section  
xSemaphoreTake(mutex, ticksToWait);  
// leave critical section  
xSemaphoreGive(mutex);
```

`ticksToWait` — время ожидания возвращения мьютекса (*в тиках*):

- `portTICK_PERIOD_MS` — число тиков в миллисекунде
- `portMAX_DELAY` — бесконечность

## 4 Мультиядерность

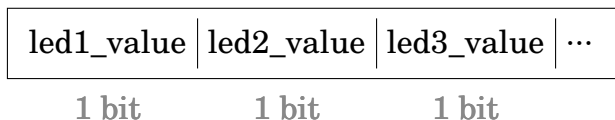
На микроконтроллере есть *дискретные светодиоды* которые мигают с разной частотой:

- 1) Микроконтроллер на одном ядре поддерживает мигание светодиодов.
- 2) Сервер с определённой частотой посылает сообщение микроконтроллеру.
- 3) Микроконтроллер на другом ядре запускает *TCP эхо-сервер*.

**Дополнительно:**

- асинхронный запрос состояний светодиодов (*mutex*)

**Структура** TCP-пакета для состояния светодиодов:



## 4.1 Бинарное представление

**Структура** дробного числа в *TCP-пакете*:



**Пример** перевода десятичной дроби:

$$36.6 = +10^{-1} \times 366 \Rightarrow \begin{cases} + \sim 0 \\ 10^{-1} \sim 1\ 000001 \\ 366 \sim 0000000000000000101101110 \end{cases}$$

$$36.6 = 010000010000000000000000101101110_2$$

При таком дизайне мантисса *не превышает*:

$$2^{24} - 1 = 16\ 777\ 215 \sim 3 \text{ байта (всего 4 байта)}$$

## 4.2 Виды датчиков температуры

**Основные** виды датчиков:

- прецизионный резистор (*NTC, PTC*)
- терморезистор
- цифровой датчик

Соотношение Стейнхарта-Харта, линейная зависимость, шины.

## 5 Определение температуры

На микроконтроллере есть *термистор*, с которого считывается *аналоговый* сигнал:

- 1) Сервер периодически отправляет *TCP-запрос* температуры на микроконтроллер.
- 2) Web-сайт периодически отправляет *TCP-запрос* температуры на сервер.

## Дополнительно:

- внедрить функцию *усреднения* аналоговых показаний
- *динамическое* обновление показаний термистора
- хранить показания термистора в *log-файле*

## 5.1 Long Polling

Если микроконтроллер выступает в роли *TCP-клиента*, он будет периодически запрашивать у сервера данные (*Regular Polling*).

Чтобы разгрузить сеть от лишних TCP-запросов (*когда состояние не поменялось с предыдущего запроса*), нужно установить *сеанс* между узлами (*Long Polling*).

**Алгоритм** установления сеанса:

- 1) Отправить запрос на сервер
- 2) Сервер не закрывает соединение
- 3) Появились ли данные:
  - > да  $\Rightarrow$  ответить запрос, закрыть соединение
  - > нет  $\Rightarrow$  ждать появления данных

Если сервер отвечает *таймаутом*, нужно повторить запрос.

*Long Polling* возможен только если сервер **асинхронный**:

- *первичные* данные поступают из серверной переменной
- *обновлённые* данные поступают из `asyncio.Future()`

## 6 Контроль дистанции

На МК есть *дальномер*, с которого считывается аналоговый сигнал, и *пьезоизлучатель*, на который подаётся аналоговый сигнал:

- 1) МК периодически считывает показания дальнометра и *усредняет* их.
- 2) МК отправляет усреднённое значение в *UDP-пакете* серверу.
- 3) Сервер в случае превышения *threshold* отправляет *UDP-пакет* МК.

- 4) МК на другом ядре ожидает *UDP-пакет*, получив который, издаёт короткий сигнал пьезоизлучателем.

**Структура** UDP-пакета для дальнометра:

distance

4 bytes

## 6.1 Interrupt

Когда на МК есть дискретный датчик, нужно знать его показания. Обычно достаточно считывать их функцией `digitalRead()` с некоторой частотой.

По аналогии с *Long Polling*, чтобы разгрузить сеть от лишних вызовов функции, нужно использовать *обработку прерываний (GPIO Interrupt)*:

```
void IRAM_ATTR function() {...}

void setup() {
  attachInterrupt(pin, function, event);
}
```

`event` — событие, которое *активирует* прерывание:

- *LOW* — *низкие* показания
- *HIGH* — *высокие* показания
- *CHANGE* — *изменились* показания
- *FALLING* — *уменьшились* показания
- *RISING* — *увеличились* показания

Функция прерывания не должна быть *блокирующей*, а именно:

- использовать `delay()`
- выводить текст в *Serial*
- работать с *мьютексами*

Иначе сработает *WDT*, и МК перезагрузится.

## 6.2 Queue

**Очередь** — объект, при помощи которого задачи *взаимодействуют* между собой и прерываниями.

Создание очереди типа *int*:

```
QueueHandle_t queue;  
void setup() {  
    queue = xQueueCreate(capacity, sizeof(int));  
}
```

Добавление *int*-элемента в очередь:

```
int element = 8;  
xQueueSend(queue, &element, ticksToWait);  
// BaseType_t status = pdFalse; // MUST-HAVE  
// xQueueSendFromISR(queue, &element, &status)
```

Перезапись *единичной int*-очереди:

```
int element = 8;  
xQueueOverwrite(queue, &element);  
// BaseType_t status = pdFalse; // MUST-HAVE  
// xQueueOverwriteFromISR(queue, &element, &status);
```

Чтение *int*-элемента в очереди:

```
int element;  
xQueuePeek(queue, &element, ticksToWait);
```

Извлечение *int*-элемента из очереди:

```
int element;  
xQueueReceive(queue, &element, ticksToWait);
```

Получение количества элементов в очереди:

```
int count = uxQueueMessagesWaiting(queue);  
// int count = uxQueueMessagesWaitingFromISR(queue);
```

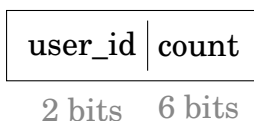


## 7 Учёт пользователей

На МК есть *две кнопки*, с которых считывается дискретный сигнал:

- 1) МК считывает *обработкой прерываний* нажатие на *первую* кнопку, инкрементирует счётчик
- 2) МК считывает *обработкой прерываний* нажатие на *вторую* кнопку, отправляет *UDP-пакет* серверу и меняет *id* пользователя
- 3) Сервер вносит изменения в базу данных при помощи *pandas*
- 4) Web-сервер по *long-polling* получает актуальные данные для отправки (*считывает их с базы данных*)

**Структура UDP-пакета:**



### 7.1 Wi-Fi Hotspot

**Создание *wi-fi*-соединения:**

```
| nmcli con add type wifi ifname * con-name * ssid *
```

**Отключение от *wi-fi*-соединения:**

```
| nmcli con down/up SSID
```

**Отключение *wi-fi*-карты:**

```
| nmcli radio wifi off/on
```

**Изменение** соединения возможно по редактированием файла по пути:

```
| vi /etc/NetworkManager/system-connections/NAME
```

После внесения изменений нужна *перезагрузка*:

```
| sudo service network-manager restart
```

## Параметры точки доступа:

— `autoconnect` — подключаться после включения к ней?  
(yes/no)

## ДНСР-настройки точки доступа:

```
[wifi]
mode=ap

[wifi-security]
key-mgmt=wpa-psk
psk=00000000

[ipv4]
method=shared
address=172.16.2.129/25
gateway=172.16.2.1
```

## 7.2 IP Redirection

### Создание мостового соединения:

```
nmcli con add type bridge ifname br0 con-name *
stp no
```

### Настройки моста:

```
[bridge]
stp=no

[ipv4]
method=auto

[ipv6]
method=disabled
```

### Создание usb-ethernet:

```
nmcli con add type -bridgeslave ifname usb0 master br0
```

## Настройка wi-fi hotspot:

```
[connection]  
slave-type=bridge  
master=br0
```

После обновления настроек нужно *подождать*.

## 8 Зарядная станция

На МК есть *LCD-дисплей*, который управляется *дискретным* сигналом кнопки:

- 1) МК изменяет состояние зарядки с использованием *хендленра*
- 2) Сервер запрашивает состояние зарядки у МК по протоколу UDP.
- 3) МК поддерживает два UDP-сервера: для состояния зарядки и для статуса батареи.
- 4) Сервер отправляет эхо-ответ МК на сервер состояния зарядки.
- 5) Сервер отправляет число активных полосок МК на сервер статуса батареи

### Дополнительно:

— внедрить *long polling* на сервер состояния зарядки

## 9 Пьезоизлучатель

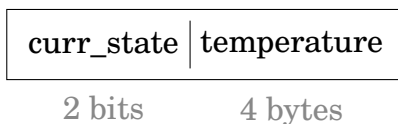
На МК есть *термистор*, с которого считывается аналоговый сигнал, и *пьезоизлучатель*, на который подаётся аналоговый сигнал:

- 1) МК периодически считывает показания термистора и переводит их в *температуру*.
- 2) МК сравнивает температуру с *threshold*, в случае превышения отправляет *UDP-пакет* серверу.
- 3) МК отправляет другой *UDP-пакет* и издаёт двойной короткий сигнал пьезоизлучателем, если температура остаётся повышенной в течение пяти секунд.
- 4) Web-сайт периодически отправляет *GET-запрос* на сервер.

## Дополнительно:

— присылать температурные показания на сервер

**Структура TCP-пакета для температурных показаний:**



## 10 Счётчик

**Идея.** На микроконтроллере сделать два счётчика нажатий, показания которых будут отображаться на LED-дисплее. Также их показания в JSON'е будут отправляться на web-сервер.

*(по сути, это практика по распаковке информации из JSON на web-сервере)*

### 10.1 Logging

Да.

### 10.2 Bluetooth

Да.

### 10.3 Net Interaction

КАК РЕАЛИЗОВАТЬ LONG POLLING НА МК?

**Идея.** Загрузить файл (*n-p, html*) на внешний сервер, а ESP32 считывает его и будет хостить.

### 10.4 Telegram Bot

Да?

### 10.5 Динамическое переподключение к Wi-Fi

Добавит надёжности системе + было у команды-победителя

# Заметки

## CSS- и JS-файлы

CSS- и JS-файлы не используются клиентом:

- веб-сервер отправил файлы
- клиент получил файлы
- клиент *не знает* тип файлов
- клиент не применяет файлы

**Решение** — указать на веб-сервере тип файлов при отправке клиенту (*поле `mime-type`*).

## 405 – Method Not Allowed

HTTP-запрос, отправленный на веб-сервер, не был принят им:

- веб-сервер работает
- веб-сервер обрабатывает адрес запроса
- веб-сервер *не обрабатывает* метод запроса

**Решение** — добавить в декоратор callback-функции используемый метод.

Если это обычное поведение веб-сервера, то пакет был отправлен *по неверному IP-адресу*.

## CORS

**CORS** (*Cross-Origin Resource Sharing*) — механизм, который по умолчанию *запрещает* пользоваться данными web-сервера сторонними источниками.

На TCP-сервере нет CORS-заголовков, поэтому их нужно дописать вручную в HTTP-response:

```
HTTP/1.1 200 OK\r\n
Access-Control-Allow-Origin: *\r\n\r\n
```