

# Как начать писать код для микроконтроллера?

# Цель занятия

- Узнать какой код вы запустили, что он значит и как конкретно он исполнялся внутри микроконтроллера.

# ВПО как исходный код

# Структура кода ВПО

```
// подключаем библиотеки из pico SDK
#include "pico/stdlib.h"
#include "hardware/gpio.h"

// задаем константу с номером ножки, к которой подключен светодиод
// номер можно узнать и проверить в документации к плате
const uint LED_PIN = 25;

int main()
{
    // Инициализация STUDIO
    stdio_init_all();
    // Инициализация GPIO вывода
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    // Бесконечный цикл, чтобы светодиод мигал все время
    while (1)
    {
        gpio_put(LED_PIN, 0);
        sleep_ms(250);
        gpio_put(LED_PIN, 1);
        sleep_ms(1000);
    }
}
```

# Подключение внешнего кода

Вставками кода занимается препроцессор. И после окончания его работы файл содержит в себе **3622 строки!**

```
main.c (27 строк)
└ pico/stdlib.h
  └ pico.h
    └ pico/types.h
      └ pico/assert.h
        └ stdbool.h (компилятор)
      └ assert.h (newlib)
        └ _ansi.h
          └ newlib.h
          └ sys/config.h
      └ stdint.h (компилятор)
        └ stdint.h (newlib)
          └ machine/_default_types.h ← Здесь определяются базовые
            └ sys/_stdint.h           ← Здесь typedef uint8_t,
          └ pico/version.h
          └ pico/config.h
    └ hardware/gpio.h
      └ hardware/structs/sio.h ← Структура для работы с GPIO
      └ hardware/structs/iobank0.h
    └ hardware/uart.h
  └ pico/time.h
```

# Объявление типов и констант

- Все, что написано в «\*.c» файлах вне кода функций по умолчанию видно лишь функциям внутри этого файла (это ограничение можно обойти, но об этом в другой раз). Это похоже на использование модификатора доступа **«private»**
- Если вы хотите, чтобы ваши типы и константы были видны другом участкам кода, как будто они **«public»**, то необходимо объявлять их внутри «\*.h» файлов. Тогда при подключении директивой **«include»** они автоматически попадут в другие «\*.c» файлы.
- **Если названия таких констант или типов совпадут, то появится ошибка, которую придется искать.**

# Список инициализаций

- По мере усложнения проекта список инициализаций будет расти: настройка тактирования, таймеров, интерфейсов связи (UART, SPI, I2C), прерываний и других модулей.
- Нужно иметь в виду, что некоторые аппаратные модули опираются на другие и важно соблюдать порядок. Первым обычно настраивается тактирование, так как оно требуется всем модулям. Если вы хотите использовать «printf» для вывода, то сначала нужно инициализировать подсистему вывода.

# Суперцикл

- Поскольку, если модуль дойдёт до конца функции main, то исполнение программы завершится, в ВПО существует концепция **суперцикла**.
- **Суперцикл** (англ. super loop, main loop, infinite loop) — архитектурный паттерн встроенного программного обеспечения, при котором основная логика приложения выполняется внутри бесконечного цикла, обеспечивающего непрерывную работу программы до отключения питания или перезагрузки системы.
- Внутри суперцикла выполняются регулярные задачи опроса модулей, отслеживания их состояний и передачи информации.

# Цепочка загрузки RP-2040

- Мы узнали, что main это примерно 0.1% нашего кода. Сейчас мы узнаем, что на самом деле это далеко не первые инструкции исполняемые кодом.

# Цепочка загрузки RP2040

# Загрузчик первой стадии

**Загрузчик первой стадии (Boot ROM)**

**Адрес: 0x00000000**

**Размер: ~16 КБ**

**~несколько тысяч инструкций**

- Инициализация минимальной периферии
- Проверка кнопки BOOTSEL:
  - Нажата -> USB режим, ждем .uf2 файл
  - Не нажата -> проверка FLASH по SPI
- Копирование boot2 из FLASH в SRAM
- Передача управления загрузчику второй стадии

# Загрузчик второй стадии

**Загрузчик второй стадии (Boot2)**

**Адрес: 0x10000000**

**Размер: 256 байт**

**~64 инструкции**

- Выполняется из SRAM (скопирован загрузчиком первой стадии)
- Знает параметры конкретной микросхемы Flash
- Настраивает режим XIP (eXecute In Place)
- Передача управления стартап-коду
- Проверка кнопки BOOTSEL

# Стартап-код

## СТАРТАП-КОД

Адрес: 0x10000100

~110 инструкций

- Таблица векторов (адреса обработчиков прерываний)
- Reset Handler (точка входа после сброса)
- Инициализация ядра, памяти и периферии
- Копирование .data из Flash в RAM
- Обнуление .bss
- Передача управления в main()

# Функция main

Функция main  
Адрес: 0x100002D0

- Инициализация модулей (stdio, GPIO, ...)
- Суперцикл — бесконечный цикл работы приложения

# Память в ВПО

# Код в памяти

```

#include "pico/stl.h"
#include "hardware/gpio.h"

.rodata (FLASH) {
    // Глобальная константа
    const uint LED_PIN = 25;
}

.data (SRAM) {
    // Глобальная переменная (инициализированная)
    uint32_t blink_count = 0;
}

.bss (SRAM) {
    // Глобальная переменная (неинициализированная)
    uint32_t last_time;
}

.text (FLASH) {
    int main()
    {
        // Локальная переменная
        uint32_t delay = 250;

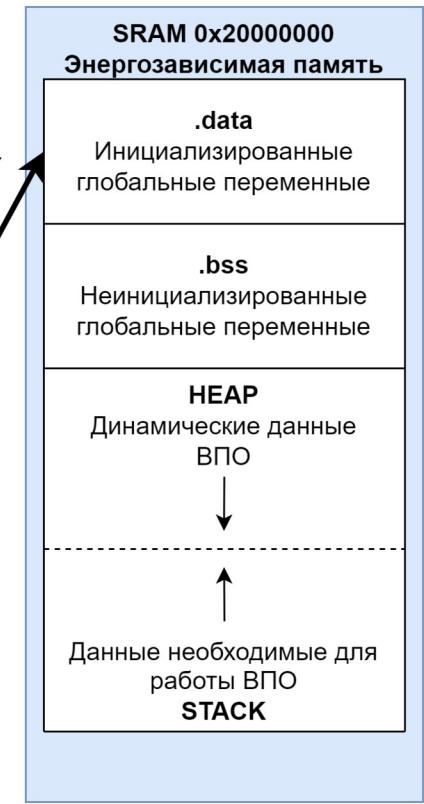
        stdio_init_all();
        gpio_init(LED_PIN);
        gpio_set_dir(LED_PIN, GPIO_OUT);

        while (1)
        {
            gpio_put(LED_PIN, 0);
            sleep_ms(delay);
            gpio_put(LED_PIN, 1);
            sleep_ms(delay * 4);
            blink_count++;
        }
    }
}

```



Инициализация глобальных переменных



# Разделы памяти

- Секция **.boot2** определяет расположение загрузчика второй стадии во **FLASH**.
- Секция **.text**, в ней хранится код main и код SDK, эта секция также располагается во **FLASH**.
- Секция **.rodata**, в ней хранятся константы. Созданная нами переменная **LED\_PIN** может храниться именно там, поскольку не будет изменяться в ходе программы, она размечена в **FLASH**.
- Секция **.data**, в ней описано, где будут храниться глобальные переменные. Обычно размечена в **SRAM**, поскольку значение глобальных может меняться в процессе работы. У этих переменных будет постоянный адрес во время всей работы программы.
- Секция **.bss**, в ней хранятся неинициализированные глобальные переменные. Она нужна для того, чтобы перед стартом программы стартап-код записал туда нулевые значения.

# Стек

- Стек (англ. Stack) — область памяти для хранения локальных переменных, адресов возврата из функций и сохранённых регистров. Работает по принципу LIFO (Last In, First Out).
- Начало стека и его границы определяются в линкер-скрипте.
- Стек растёт с максимального адреса до минимального.

# Куча

- Куча (англ. Heap) — область памяти для динамического выделения во время выполнения программы. Управляется программистом через `malloc()/free()`.
- Кучу контролирует аллокатор памяти, который находит свободные места в куче и удаляет ненужные данные.

# Память как инструмент

# Указатели на переменные и константы

Операция взятия адреса

```
&LED_PIN
```

Сохранение указателя на константу

```
const uint32_t* ptr = &LED_PIN;
```

Запись значения лежащего по адресу в переменную

```
uint32_t value = *ptr;
```

Запись в переменную значения, лежащего по явно указанному адресу

```
uint32_t value = *(uint32_t*)0x10001E28;
```

# Квалификатор volatile

- volatile— это квалификатор типа (type qualifier) в языках С и С++, который говорит компилятору: «Не оптимизируй доступ к этой переменной. Её значение может измениться в любой момент без ведома компилятора».
- Компиляторы оптимизируют код: если переменная читается несколько раз подряд, компилятор может сохранить её значение в регистр процессора и не читать из памяти повторно. Обычно это ускоряет работу. Но иногда такое поведение ломает логику программы.
- Такое может происходить в регистрах периферии

```
uint32_t reg_value = *(volatile uint32_t*)0x40014000;
```

# Указатели на структуру.

## Пример с выравниванием

- Компилятор автоматически добавляет между полями **выравнивание (padding)** — пустые байты, чтобы каждое поле начиналось с подходящего адреса. В результате размер структуры может оказаться больше суммы размеров полей
- Функция `sizeof(pins_t)` возвращает итоговый размер с учётом паддинга
- Макрос `offsetof(pins_t, LED_PIN)` — смещение поля относительно начала структуры

```
typedef struct {
    uint32_t LED_PIN;
    uint32_t BUTTON_PIN;
} pins_t;

pins_t my_pins = { .LED_PIN = 25, .BUTTON_PIN = 12 };
```

# Указатели на структуру. Пример с оператором ->

```
pins_t* p = &my_pins;  
gpio_put(p->LED_PIN, 1);  
gpio_init(p->BUTTON_PIN);
```

# Указатели на структуру.

## Отключение padding

- Если структура должна точно соответствовать размерам полей (например, при обмене по протоколу или при работе с регистрами периферии), используют директиву `#pragma pack`

```
#pragma pack(push, 1)
typedef struct {
    uint8_t cmd;
    uint16_t value;
    uint32_t timestamp;
} packet_t;
#pragma pack(pop)
```

# Указатели на функции

Объявление типа add\_ptr\_t - указателя на  
функцию, принимающую два int и  
возвращающую int

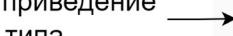


```
int add(int a, int b)
{
    return a + b;
}

typedef int (*add_ptr_t)(int, int);

int main()
{
    add_ptr_t add_func = (add_ptr_t)0x10001A00;
    int result = add_func(2, 2);
}
```

Явное приведение  
типа



# Передача функции как параметра (callback)

Объявление указателя на  
функцию  
типа callback\_t

Сигнатура функции с  
параметром в виде  
указателя на функцию

Защита от перехода  
по нулевому адресу

Передача функции в  
качестве аргумента

```
typedef void (*callback_t)(void);

void run_after_delay(uint32_t ms, callback_t callback)
{
    sleep_ms(ms);
    if (callback != NULL) {
        callback();
    }
}

void on_timeout(void)
{
    gpiو_put(LED_PIN, 1);
}

int main()
{
    run_after_delay(1000, on_timeout);
}
```

# Массив указателей на функции

Объявление указателя на  
функцию  
типа handler\_t

→ `typedef void (*handler_t)(void);  
handler_t handlers[] = { handle_idle, handle_run, handle_error };`

Объявление массива функций →

Защита от выхода  
за границы массива

```
void dispatch(uint8_t state)
{
    if (state < sizeof(handlers) / sizeof(handlers[0])) {
        handlers[state]();
    }
}
```

# Ввод и вывод в микроконтроллере

# Путь от printf до терминала



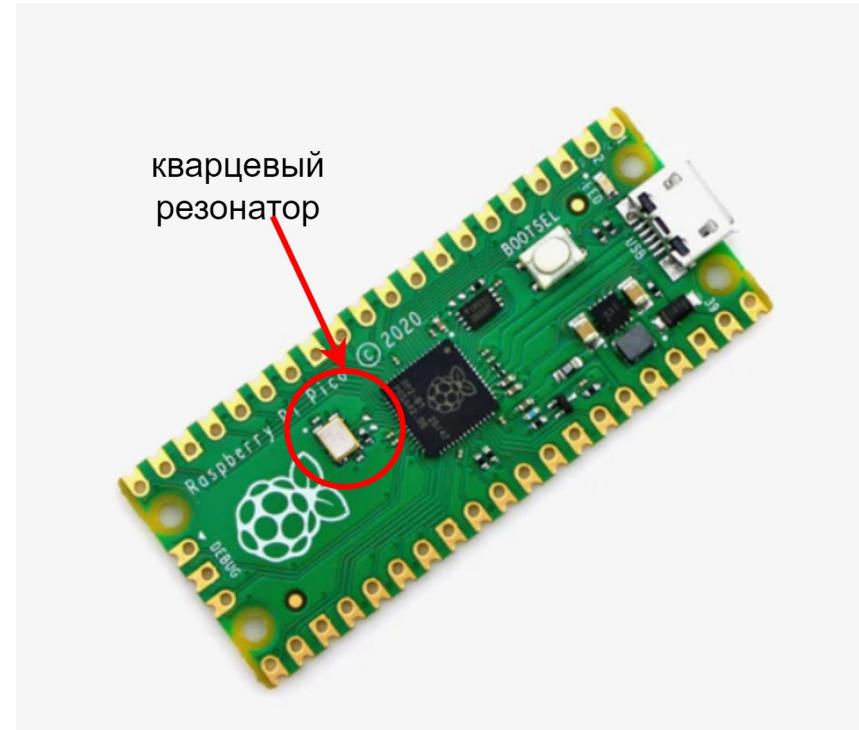
# Время внутри микроконтроллера

# Тактовый сигнал

- Тактовый сигнал (англ. *clock*) — периодическая последовательность электрических импульсов, которая синхронизирует все операции в микроконтроллере. Один такт — одна атомарная операция: одна инструкция процессора, один шаг счётчика таймера.
- API (**A**pplication **P**rogramming **I**nterface) — набор функций, скрывающих детали реализации и предоставляющих удобный интерфейс для программиста

# Осциллятор

- Кварцевый генератор – источник стабильных колебаний, который генерируется с помощью кварцевого резонатора
- Уже установлен на Pico, частота 12МГц
- Для изменения частоты колебаний используются PLL (Phase-Locked-Loop)
- Монотонное время – счётчик, который не уменьшается и не откатывается назад. Идеально подходит для измерения интервалов и проверки «прошло ли уже N единиц времени».



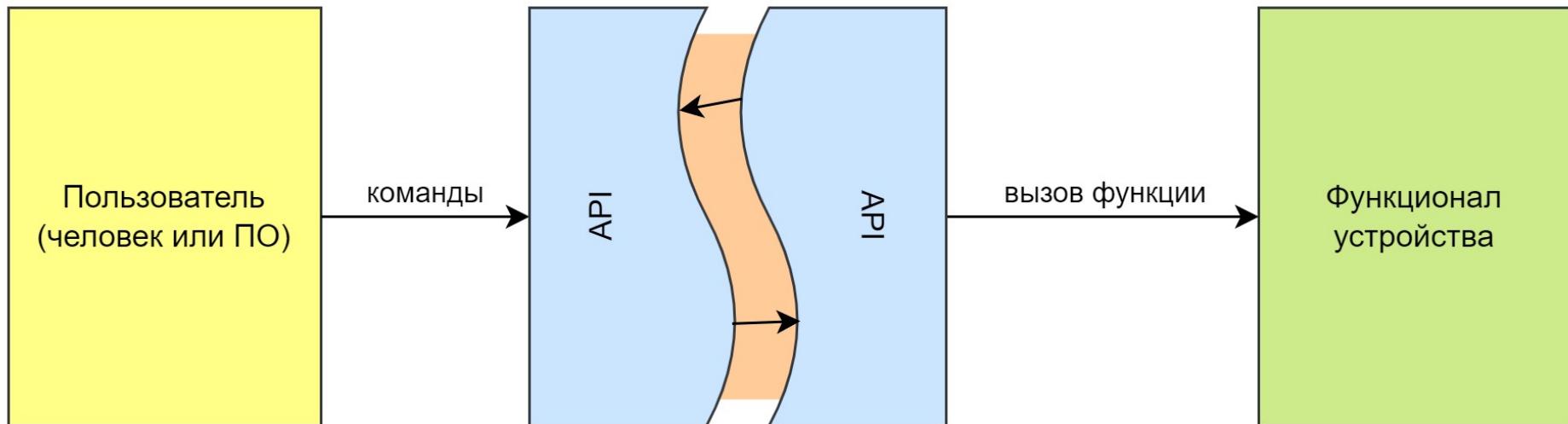
# Работа со временем

Задача	Инструмент
Простая задержка (мигание, пауза)	<code>sleep_ms()</code> / <code>sleep_us()</code>
Измерение интервалов, проверка «прошло ли»	<code>time_us_64()</code> или <code>get_absolute_time() + absolute_time_diff_us()</code>
Ожидание без блокировки (можно делать что-то ещё)	Alarm'ы ( <code>add_alarm_in_us</code> , <code>add_alarm_in_ms</code> )
Дата и время «как на часах»	RTC ( <code>hardware/rtc.h</code> )

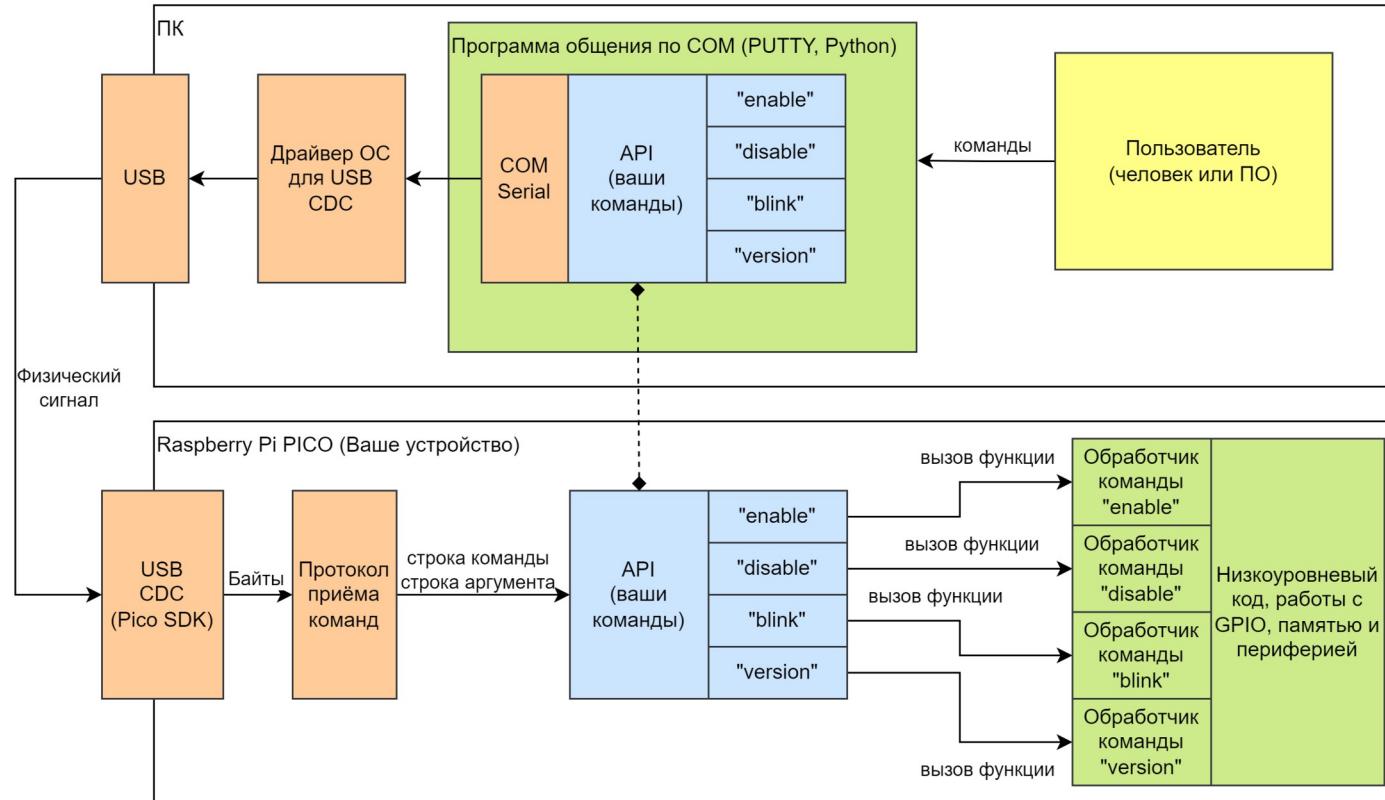
# От ВПО к устройству

# От ВПО к устройству

- API (Application Programming Interface) – набор функций, скрывающих детали реализации и предоставляющих удобный интерфейс для программиста



# От ВПО к устройству



# Архитектура ВПО

# Задачи (Tasks)

- Задача (англ. task) — независимая программная единица, которая выполняет одну конкретную функцию в системе. Задачи работают псевдопараллельно: на одном ядре процессора они выполняются по очереди, но достаточно быстро, чтобы казалось, что всё происходит одновременно.

**Свойства задачи:**

- Самостоятельность — у задачи своя логика работы
- Одна ответственность — каждая задача отвечает за одну функцию системы
- Минимальная связность — задачи как можно меньше зависят друг от друга

**Термостат (пример):**

- читать температуру с датчика,
- обновлять дисплей,
- реагировать на нажатия кнопок,
- управлять нагревателем,
- отправлять данные по сети.

# Суперцикл

Вызов нескольких задач по очереди в бесконечном цикле

## Достоинства:

- Простота
- Нет накладных расходов на переключение между задачами
- Предсказуемость порядка выполнения

## Недостаток:

- Зависание одной задачи блокирует выполнение остальных

```
int main() {
    init_system();

    while (1) {
        task_button_check();      // Задача: опрос кнопок
        task_led_control();      // Задача: управление светодиодом
        task_sensor_read();      // Задача: чтение датчика
        task_uart_process();     // Задача: обработка UART
    }
}
```

# Неблокирующий подход

- Вместо использования блокирующей функции sleep\_ms, проверять прошло ли нужное время и если нет, идти дальше
- Паттерн «проверь время → если пора, выполни → обнови метку» — основа неблокирующего программирования на микроконтроллерах. Вы будете встречать его повсюду: в библиотеках, в примерах SDK, в промышленном коде.

```
uint64_t last_blink_time = 0;  
|  
void task_blink(void) {  
    uint64_t now = time_us_64();  
    if (now - last_blink_time >= 250000) { // 250 мс = 250000 мкс  
        gpio_put(LED_PIN, !gpio_get(LED_PIN));  
        last_blink_time = now;  
    }  
    // Если 250 мс ещё не прошли – сразу выходим, не блокируя  
}
```

# Кооперативная многозадачность

Управление задачами  
происходит через планировщик

```
typedef struct {
    void (*function)(void);    // Функция задачи
    uint32_t period_ms;        // Период запуска (мс)
    uint32_t last_run_ms;      // Время последнего запуска
} Task;

Task tasks[] = {
    {task_button_check, 10, 0}, // Кнопки – каждые 10 мс
    {task_led_control, 250, 0}, // Светодиод – каждые 250 мс
    {task_sensor_read, 1000, 0}, // Датчик – каждую секунду
    {task_uart_process, 5, 0}, // UART – каждые 5 мс
};

void scheduler(void) {
    uint32_t now = to_ms_since_boot(get_absolute_time());
    for (int i = 0; i < 4; i++) {
        if (now - tasks[i].last_run_ms >= tasks[i].period_ms) {
            tasks[i].function();           // выполняем задачу
            tasks[i].last_run_ms = now;    // обновляем метку времени
        }
    }
}

int main() {
    stdio_init_all();
    // ... инициализация периферии ...

    while (1) {
        scheduler();
    }
}
```

# Машина состояний

- **Машина состояний** (англ. Finite State Machine, FSM) — модель поведения, в которой система может находиться в одном из конечного числа **состояний**. Переход между состояниями происходит при наступлении определённых **событий**. В каждом состоянии система ведёт себя по-разному.

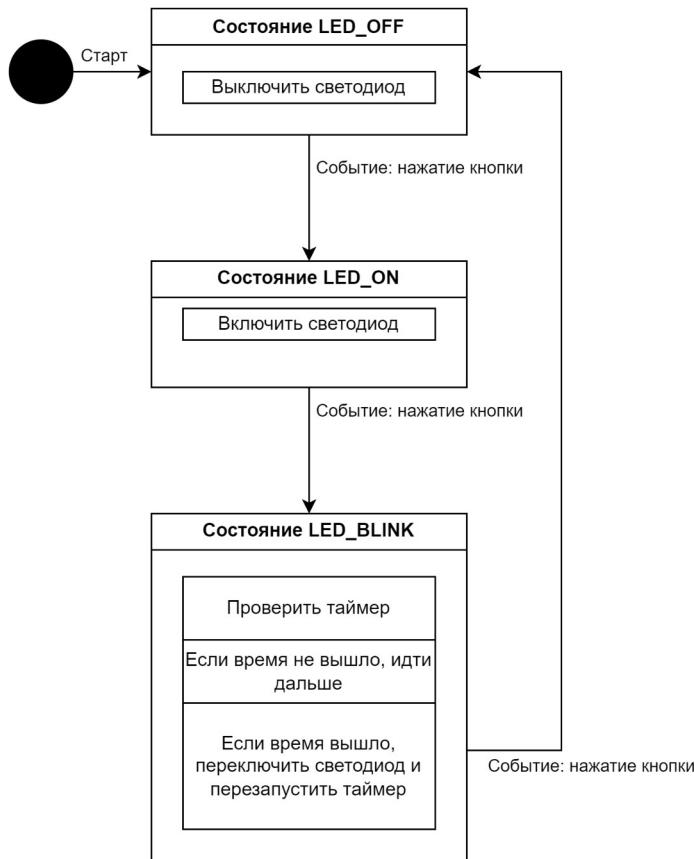
**Компоненты машины состояний:**

- Состояния (states) — дискретные режимы работы: «выключен», «горит», «мигает»
- События (events) — то, что вызывает переход: нажатие кнопки, истечение таймера, приход данных
- Переходы (transitions) — правила: «из состояния А по событию Х перейти в состояние В»
- Действия (actions) — что делать при переходе или в текущем состоянии

**Преимущества:**

- Чёткая структура — логика программы понятна и предсказуема
- Лёгкость отладки — всегда известно текущее состояние системы
- Масштабируемость — добавить новый режим = добавить `case` и переходы
- Неблокирующая работа — FSM проверяет состояние и мгновенно возвращается, идеально для кооперативной многозадачности

# Машина состояний



```

typedef enum
{
    LED_STATE_OFF   = 0,
    LED_STATE_ON    = 1,
    LED_STATE_BLINK = 2,
} led_state_t;

uint64_t led_ts = 0;
led_state_t led_state = LED_STATE_OFF;

void led_task_handle() {
    switch (led_state) {
        case LED_STATE_OFF:
            gpio_put(LED_PIN, 0);
            break;
        case LED_STATE_ON:
            gpio_put(LED_PIN, 1);
            break;
        case LED_STATE_BLINK:
            if (time_us_64() > led_ts)
            {
                led_ts = time_us_64() + (LED_BLINK_PERIOD_US / 2);
                bool state = gpio_get(LED_PIN);
                gpio_put(LED_PIN, !state);
            }
            break;
        default:
            led_state = LED_STATE_OFF;
    }
}

void on_button_press(void) {
    switch (led_state) {
        case LED_STATE_OFF:   led_state = LED_STATE_ON;      break;
        case LED_STATE_ON:    led_state = LED_STATE_BLINK;   break;
        case LED_STATE_BLINK: led_state = LED_STATE_OFF;     break;
    }
}
  
```

The provided C code defines the state transitions and event handling for the LED control system. It includes an enum for states (LED\_STATE\_OFF, LED\_STATE\_ON, LED\_STATE\_BLINK), variables for the current state (led\_state) and timestamp (led\_ts), and functions for task handling (led\_task\_handle) and button presses (on\_button\_press). The led\_task\_handle function uses a switch statement to handle each state. For the LED\_BLINK state, it checks if the current time is greater than the stored timestamp. If so, it updates the timestamp, toggles the LED state using gpio\_put, and then resets the timestamp. The on\_button\_press function also uses a switch statement to handle the state changes based on the current state.