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

Подключение внешнего кода

Объявление типов и констант

Список инициализаций

Суперцикл

```c
// 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 STDIO
    stdio_init_all();
    // Инициализация GPIO вывода
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    // Бесконечный цикл, чтобы светодиод мига 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 типы!
                    └── sys/_stdint.h             ← Здесь typedef uint8_t, uint32_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


```c
#include "pico/stdlib.h"
#include "hardware/gpio.h"

// Глобальная константа
const uint LED_PIN = 25;

// Глобальная переменная (инициализированная)
uint32_t blink_count = 0;

// Глобальная переменная (неинициализированная)
uint32_t last_time;

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++;
    }
}
```

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 хранится именно там, поскольку не будет изменятся 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.

Схема секций памяти

FLASH 0x10000000 - 0x101FFFFF			
.boot2	0x10000000	256 байт	Загрузчик 2
.text	0x10000100	~7 КБ	Код main(), SDK
.rodata	0x10001E28	~300 байт	LED_PIN = 25
.data (LMA)	0x10001F74	~384 байт	Копия для blink_count

SRAM 0x20000000 - 0x20041FFF			
.data (VMA)	0x200000C0	~384 байт	blink_count = 0
.bss	0x20000240	~1 КБ	last_time (= 0)
Heap ↓	0x20000640	растёт ↓	malloc(), etc.
...			
Stack ↑	0x20042000	растёт ↑	delay = 250

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 C и C++, 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

```c
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

Объявления типа add_ptr_t - указателя на функцию, принимающую два int и возвращающую int

Явное приведение типа

```
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);
}
```

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

```c
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)
{
    gpio_put(LED_PIN, 1);
}

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

Массив указателей на функции

Объявление указателя на функцию типа handler_t

Объявление массива функций

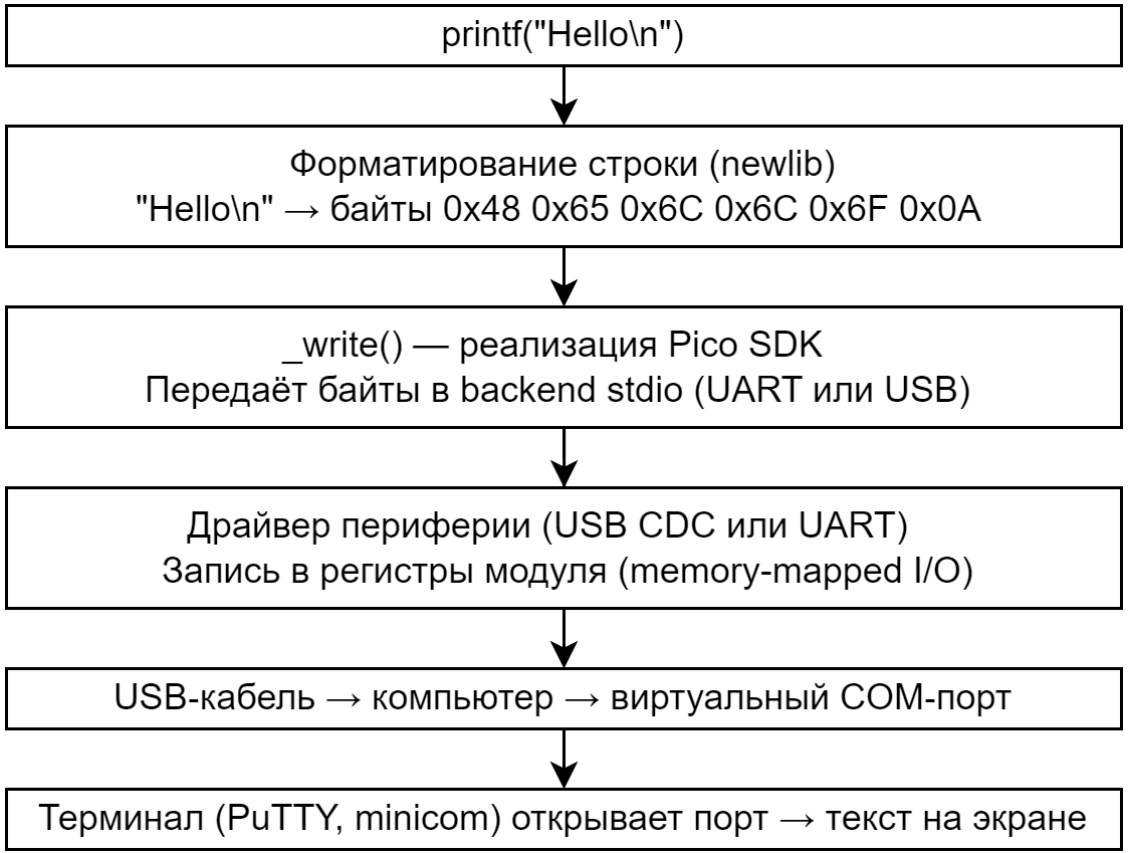
Защита от выхода за границы массива

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

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 и осциллятор

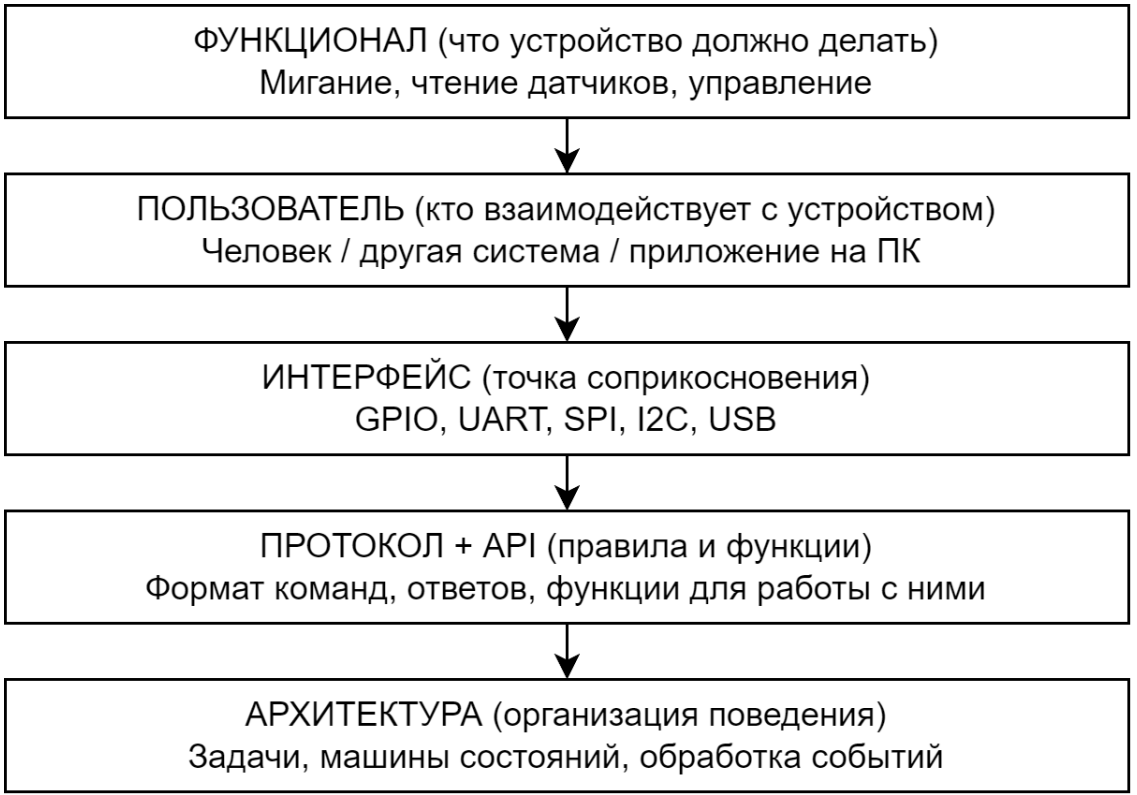
- Тактовый сигнал (англ. clock) — периодическая последовательность электрических импульсов, которая синхронизирует все операции в микроконтроллере. Один такт — одна атомарная операция: одна инструкция процессора, один шаг счётчика таймера.
- Кварцевый генератор — источник стабильных колебаний, который генерируется с помощью кварцевого резонатора
- 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)	sleep_ms() /sleep_us()
Измерение интервалов, проверка «прошло ли»	time_us_64() или get_absolute_time() + absolute_time_diff_us()
Ожидание без блокировки (можно делать что-то ещё)	Alarm'ы (add_alarm_in_us, add_alarm_in_ms)
Дата и время «как на часах»	RTC (hardware/rtc.h)

От ВПО к устройству

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

```c
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

```c
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 A по событию X перейти в состояние B»
  - 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 (код)

```c
typedef enum {
    STATE_OFF,
    STATE_ON,
    STATE_BLINK
} LedState;

LedState led_state = STATE_OFF;

void task_led_control(void) {
    switch (led_state) {
        case STATE_OFF:
            gpio_put(LED_PIN, 0);
            break;
        case STATE_ON:
            gpio_put(LED_PIN, 1);
            break;
        case STATE_BLINK:
            gpio_put(LED_PIN, !gpio_get(LED_PIN));
            break;
    }
}

void on_button_press(void) {
    switch (led_state) {
        case STATE_OFF:   led_state = STATE_ON;    break;
        case STATE_ON:    led_state = STATE_BLINK;  break;
        case STATE_BLINK: led_state = STATE_OFF;    break;
    }
}
```