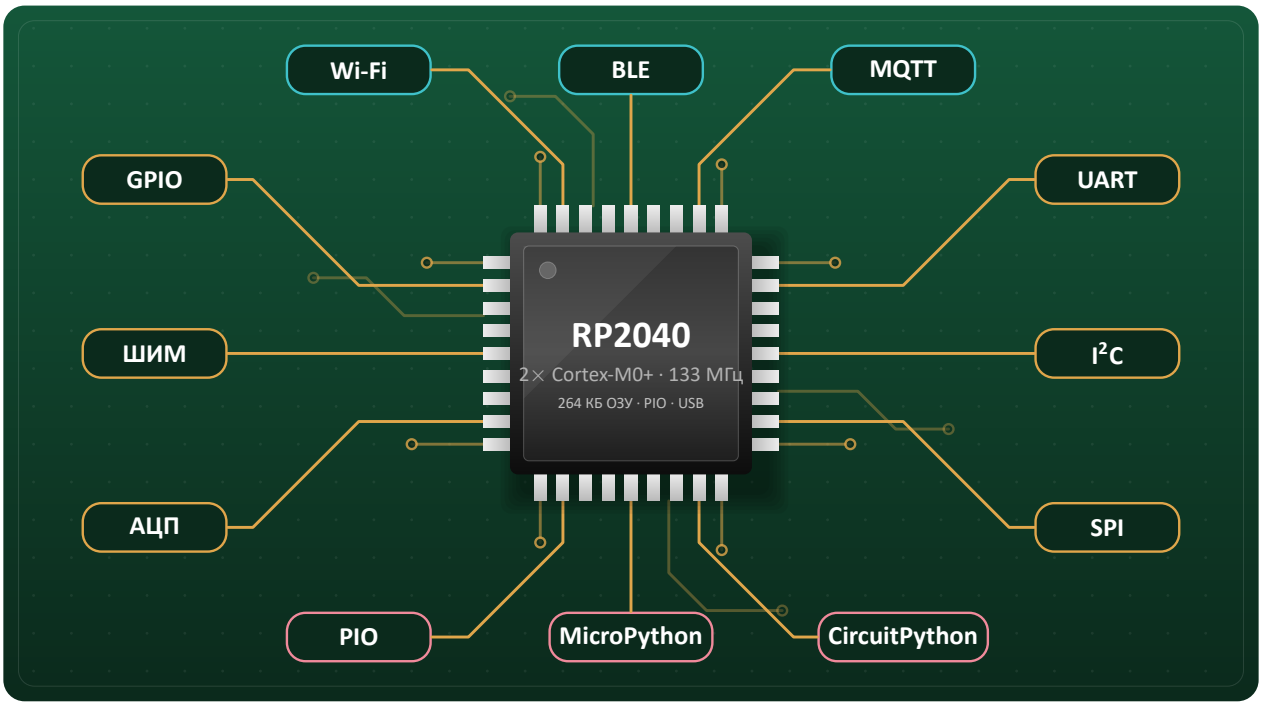
УЧЕБНОЕ ПОСОБИЕ

Raspberry Pi Pico

Практический курс MicroPython и CircuitPython

Pico, Pico W, Pico 2 и Pico 2 W: от первого светодиода до умного устройства



2026

Raspberry Pi Pico: практический курс MicroPython и CircuitPython : учебное пособие. — 2026. — 492 с.

Пособие содержит вводные главы, двадцать пять уроков и приложения по программированию плат Raspberry Pi Pico, Pico W, Pico 2 и Pico 2 W на языках MicroPython и CircuitPython: от техники безопасности, цифровых входов и выходов, ШИМ и АЦП до шин I^2C и SPI, датчиков, адресных светодиодов, файловой системы, многозадачности, программируемого ввода-вывода (PIO), Wi-Fi, веб-серверов, MQTT, Bluetooth Low Energy, моторов, звука, USB, отладки и итогового проекта. Каждый урок включает объяснение идеи, рисунки и схемы, готовые примеры для обоих языков, разбор типичных ошибок, вопросы для самопроверки и шесть заданий четырёх уровней сложности.

Пособие адресовано обучающимся объединений технической направленности и педагогам дополнительного образования. Примеры рассчитаны на MicroPython версии 1.2x и CircuitPython версии 9.x; перед использованием сверяйтесь с документацией вашей версии прошивки.

Содержание

Техника безопасности и рабочее место **11**

Макетная плата, схема и мультиметр **12**

Знакомство: микроконтроллер, Pico и семейство RP2 **15**

Выводы платы: распиновка, питание, правила **19**

1 Первый запуск: прошивка, Thonny, REPL **22**
1.1 Идея урока коротко . . . 22
1.2 Выбираем прошивку . . . 22
1.3 Режим BOOTSEL: загружаем прошивку . . . 23
1.4 MicroPython: Thonny и консоль . . . 23
1.4.1 Горячие клавиши REPL . . . 24
1.5 MicroPython: программа, которая запускается сама . . . 24
1.6 MicroPython: утилита mpremote . . . 25
1.7 CircuitPython: плата как диск . . . 25
1.8 Информация о вашей плате . . . 26
1.9 Сравнение подходов к работе . . . 27

2 Цифровой выход: светодиод **29**
2.1 Идея урока коротко . . . 29
2.2 Зачем нужен резистор: закон Ома . . . 29
2.2.1 Почему светодиод сгорает без резистора . . . 29
2.2.2 Рассчитываем резистор . . . 30
2.3 Как устроен выход GPIO . . . 31
2.4 Подключаем светодиод . . . 31
2.5 Классический Blink . . . 32
2.6 Разберём Blink по строкам . . . 33
2.7 Мигание без остановки программы . . . 33
2.8 Учимся программировать: цикл `for` и списки . . . 35
2.9 Практика: бегущий огонь . . . 35
2.10 Встроенный светодиод платы . . . 37
2.11 Самостоятельный опыт . . . 38

3 Цифровой вход: кнопка **40**
3.1 Идея урока коротко . . . 40
3.2 Что читает вход и почему нельзя оставлять его «в воздухе» . . . 40
3.3 Логические уровни: где проходит граница между 0 и 1 . . . 40
3.4 Подтяжка: pull-up и pull-down . . . 41
3.5 Подключаем кнопку . . . 42
3.6 Учимся программировать: функция, которая прячет «странную» логику . . . 44
3.7 Дребезг контактов . . . 45
3.7.1 Три способа победить дребезг . . . 46
3.8 Программный антидребезг без блокировки . . . 46
3.9 Аппаратный антидребезг . . . 49
3.10 Опрос или прерывание . . . 49
3.10.1 Как это в **CircuitPython** . . . 51
3.11 Переключатель-триггер: нажал — включил, нажал — выключил . . . 51
3.12 Несколько кнопок на одной плате . . . 53

3.12.1 Класс: образец кнопки . . . 54
3.13 Долгое нажатие . . . 56
3.14 Самостоятельный опыт . . . 58

4 Время и конечные автоматы 59
4.1 Идея урока коротко . . . 59
4.2 Часы микроконтроллера: тики . . . 59
4.2.1 Переполнение счётчика . . . 60
4.3 Планировщик «на тиках» . . . 61
4.4 Аппаратные таймеры: `machine.Timer` . . . 64
4.4.1 А как в **CircuitPython**? . . . 66
4.5 Учимся программировать: класс «Задача» . . . 67
4.6 Сколько времени занимает код: замер . . . 69
4.6.1 Загрузка процессора . . . 71
4.7 Конечные автоматы . . . 71
4.7.1 Светофор . . . 72
4.7.2 Учимся программировать: словарь как таблица . . . 72
4.7.3 Монтаж светофора . . . 73
4.7.4 Программа светофора . . . 73
4.7.5 Добавляем пешеходную кнопку . . . 75
4.8 Самостоятельный опыт . . . 78

5 ШИМ: яркость, звук, серво 80
5.1 Идея урока коротко . . . 80
5.2 Что такое ШИМ . . . 80
5.2.1 Среднее напряжение . . . 81
5.2.2 Почему светодиод «видит» среднее . . . 81
5.3 Как устроен ШИМ в RP2040 . . . 82
5.3.1 Слайсы: кто с кем связан . . . 82
5.4 Подключаем светодиод к ШИМ . . . 84
5.5 Гамма-коррекция: почему «линейное» затухание выглядит неправильно . . . 85
5.5.1 «Дыхание» светодиода . . . 86
5.6 Учимся программировать: вычислить заранее . . . 87
5.7 RGB-светодиод: смешиваем цвета . . . 87
5.8 Сервопривод . . . 90
5.8.1 Класс «Серво» и два привода . . . 94
5.9 Самостоятельный опыт . . . 95

6 АЦП: аналоговые входы 97
6.1 Идея урока коротко . . . 97
6.2 Квантование: измерение по ступенькам . . . 97
6.3 АЦП RP2040: каналы и «16 бит» . . . 98
6.3.1 Опорное напряжение и «16 бит» . . . 98
6.4 Читаем АЦП . . . 99
6.5 Делитель напряжения . . . 100
6.6 Практика: потенциометр . . . 102
6.7 Датчик света: фоторезистор . . . 103
6.7.1 Учимся программировать: гистерезис . . . 105
6.8 Датчик температуры: NTC-термистор . . . 106
6.9 Встроенный датчик температуры . . . 108
6.10 Шум и как с ним бороться . . . 108
6.11 Самостоятельный опыт . . . 111

7 UART и последовательный обмен 113
7.1 Идея урока коротко . . . 113
7.2 Как говорят без общих часов . . . 113
7.3 Кадр 8N1: как выглядит один байт . . . 113
7.4 Почему скорости должны совпадать . . . 114
7.5 Уровни напряжения: UART, 5 В и RS-232 . . . 114
7.6 UART в Pico: два блока и USB-консоль . . . 115
7.7 Подключаем USB-UART адаптер . . . 116
7.8 Учимся программировать: байты и строки . . . 118
7.9 Протокол из строк: команды и ответы . . . 118
7.10 USB-консоль: `print()` и `input()` . . . 121
7.11 Практика: разбор предложения GPS . . . 123
7.12 Bluetooth-модули HC-05 и HC-06 . . . 125

8 Шина I2C: OLED, датчики, часы 128
8.1 Идея урока коротко . . . 128
8.2 Как устроена шина . . . 128
8.2.1 Открытый сток и подтяжка . . . 129
8.2.2 Какой резистор выбрать . . . 129
8.3 Один обмен: старт, адрес, подтверждение, стоп . . . 130
8.4 Выводы I2C в Pico . . . 131
8.5 Сначала ищем устройства: scan() . . . 131
8.6 Монтажная схема: OLED и BME280 на общей шине . . . 132
8.7 Дисплей SSD1306 . . . 133
8.8 Датчик BME280: температура, влажность, давление . . . 135
8.9 Учимся программировать: шестнадцатеричные числа и биты . . . 136
8.10 Часы реального времени DS3231 . . . 136
8.11 Рисуем крупно: свой «шрифт» из framebuf . . . 138

9 Шина SPI: SD-карта и дисплеи 141
9.1 Идея урока коротко . . . 141
9.2 Четыре провода и ведущий . . . 141
9.3 Обмен как сдвиг . . . 142
9.4 Режимы SPI: полярность и фаза . . . 142
9.4.1 Как выглядит один байт . . . 143
9.5 Скорость и выводы в Pico . . . 143
9.6 Учимся программировать: биты, маски и `bytearray` . . . 145
9.7 Сдвиговый регистр 74HC595 . . . 145
9.8 SD-карта . . . 148
9.9 Цветные дисплеи ST7789 и другие . . . 151
9.10 I2C, SPI или UART: что выбрать . . . 153

10 Датчики: температура, расстояние, движение 155
10.1 Идея урока коротко . . . 155
10.2 Температура и влажность: DHT22 . . . 155
10.2.1 Протокол на пальцах . . . 156
10.2.2 Монтажная схема . . . 156
10.3 Много термометров на одном проводе: DS18B20 . . . 158
10.4 Расстояние: ультразвуковой HC-SR04 . . . 160
10.4.1 Делитель напряжения . . . 161
10.5 Движение: PIR-датчик HC-SR501 . . . 163
10.6 Аналоговые датчики: свет и звук . . . 164

10.7 Наклон и движение: акселерометр MPU6050 . . . 166
10.8 Сбор и обработка данных . . . 167
10.9 Учимся программировать: классы . . . 169

11 Адресные светодиоды WS2812 173
11.1 Идея урока коротко . . . 173
11.2 Как устроен WS2812 . . . 173
11.3 Протокол: один провод, 800 кГц . . . 173
11.3.1 Формат кадра: 24 бита на пиксель . . . 174
11.4 Питание: самая важная часть . . . 174
11.4.1 Защитные детали: конденсатор и резистор . . . 175
11.4.2 Уровень сигнала: 3,3 В или 5 В? . . . 175
11.5 Подключаем кольцо . . . 176
11.6 Первая программа . . . 177
11.7 Цвет: смешиваем свет . . . 178
11.7.1 Модель HSV . . . 179
11.8 Учимся программировать: функции и остаток от деления . . . 179
11.9 Радуга . . . 180
11.10 Бегущая «комета» . . . 182
11.11 Яркость и гамма-коррекция . . . 183
11.12 Эффект «дыхание» . . . 184
11.13 Эффект «огонь» . . . 185
11.14 Готовые анимации в CircuitPython . . . 187
11.15 Самостоятельный опыт . . . 187

12 Память и файлы 189
12.1 Идея урока коротко . . . 189
12.2 Какая память есть в Pico . . . 189
12.3 Как разделена flash-память . . . 190
12.4 Почему flash «изнашивается» . . . 191
12.4.1 Стратегии бережной записи . . . 191
12.5 Файловая система LittleFS . . . 192
12.6 Работа с файлами в Python . . . 192
12.7 Учимся программировать: контекстный менеджер `with` . . . 194
12.8 Форматы данных: текст, JSON и CSV . . . 194
12.9 Хранение настроек . . . 194
12.10 Журнал измерений с временем . . . 196
12.11 CircuitPython: диск принадлежит компьютеру . . . 198
12.11.1boot.py и перемычка выбора режима . . . 199
12.12 SD-карта . . . 201
12.13 Оперативная память и сборка мусора . . . 202
12.14 Копирование, удаление и резервные копии . . . 203
12.15 Самостоятельный опыт . . . 203

13 Многозадачность: два ядра, потоки, asyncio 205
13.1 Идея урока коротко . . . 205
13.2 Зачем нужна параллельность . . . 205
13.3 Два ядра RP2040 . . . 206
13.4 Модуль `_thread` в **MicroPython** . . . 206
13.5 Гонка данных и замок . . . 207
13.6 Кооперативная многозадачность: asyncio . . . 209
13.7 Практика: мигаем и опрашиваем кнопку . . . 210

13.8 Учимся программировать: корутины и `await` . . . 212
13.9 Обмен данными между задачами: «датчик, экран, светодиод» . . . 212
13.10 Замок в asyncio и когда он нужен . . . 215
13.11 Сравнение подходов и как выбрать . . . 216
13.12 Самостоятельный опыт . . . 216

14 PIO: программируемый ввод-вывод 218
14.1 Идея урока коротко . . . 218
14.2 Зачем нужен PIO . . . 218
14.3 Устройство PIO . . . 219
14.4 Тактирование: делитель частоты . . . 220
14.5 Девять команд PIO . . . 220
14.5.1 Задержки и side-set . . . 221
14.6 Первая программа: мигалка на PIO . . . 221
14.7 Учимся программировать: биты и сдвиги . . . 224
14.8 WS2812 на PIO . . . 224
14.9 Измеряем импульс . . . 226
14.10 UART-передатчик на PIO . . . 228
14.11 PIO и CircuitPython: `rp2pio` и `adafruit_pioasm` . . . 230
14.12 Ограничения PIO и RP2350 . . . 230

15 Питание, сон и сторожевой таймер 233
15.1 Идея урока коротко . . . 233
15.2 Откуда плата берёт энергию . . . 233
15.2.1 Диод-ИЛИ: два источника без драки . . . 234
15.3 Питание от батареек . . . 234
15.3.1 Литий-полимерный аккумулятор . . . 236
15.4 Сколько тратит плата . . . 236
15.5 Снижение тактовой частоты . . . 237
15.6 Сон: идея и режимы . . . 239
15.7 Лёгкий сон . . . 239
15.7.1 Как работает пробуждение . . . 240
15.8 Глубокий сон . . . 242
15.8.1 Работаем циклами: «проснулся — измерил — заснул» . . . 243
15.9 Измеряем заряд батареи . . . 245
15.9.1 А что с «встроенным» измерением VSYS? . . . 247
15.10 Учимся программировать: усреднение и фильтр . . . 248
15.11 Сторожевой таймер . . . 249
15.11.1Где «кормить» сторожа . . . 250
15.12 Причины сброса . . . 251
15.13 Практика: автономный датчик . . . 252

16 Wi-Fi: Pico W и Pico 2 W 254
16.1 Идея урока коротко . . . 254
16.2 Радиочип CYW43439 . . . 254
16.2.1 Диапазон 2,4 ГГц и каналы . . . 255
16.3 Два режима: станция и точка доступа . . . 255
16.4 Первое подключение . . . 256
16.4.1 Где хранить пароль . . . 256
16.4.2 Разбираем статус . . . 258
16.5 Надёжная функция подключения . . . 258
16.6 Сила сигнала и сканирование . . . 262

16.7 Адреса в сети . . . 263
16.8 Стек протоколов: от радиоволн до JSON . . . 264
16.9 HTTP-клиент . . . 265
16.9.1 Погода без регистрации . . . 265
16.9.2 Отправляем данные: POST и JSON . . . 268
16.10 Время по сети: NTP . . . 269
16.11 Учимся программировать: обработка ошибок и словари . . . 271
16.12 Практика: «сетевая метеостанция» . . . 271

17 Веб-сервер на Pico W 273
17.1 Идея урока коротко . . . 273
17.2 Клиент и сервер . . . 273
17.3 HTTP построчно . . . 273
17.4 Минимум HTML, CSS и JavaScript . . . 274
17.5 Сокеты: как сервер принимает клиентов . . . 275
17.6 Практика: панель управления светодиодом . . . 277
17.7 Учимся программировать: строки и словари . . . 282
17.8 JSON-интерфейс и `fetch()` . . . 282
17.9 Асинхронный сервер . . . 285
17.9.1 Готовые библиотеки . . . 288
17.10 Собственная сеть: режим точки доступа . . . 288
17.10.1Идея «captive portal» . . . 290
17.11 Безопасность . . . 291
17.12 Практика: «умная лампа» . . . 293

18 MQTT и интернет вещей 295
18.1 Идея урока коротко . . . 295
18.2 Модель «публикация — подписка» . . . 295
18.2.1 Темы и шаблоны . . . 296
18.3 Надёжность доставки: QoS, retained и «завещание» . . . 296
18.3.1 Retained-сообщения . . . 297
18.3.2 «Завещание» (Last Will) . . . 297
18.4 Брокер Mosquitto . . . 298
18.5 Формат пакета и почему MQTT экономен . . . 298
18.6 Практика: узел с датчиком и исполнителем . . . 299
18.6.1 Публикация данных . . . 300
18.6.2 Подписка на команды . . . 302
18.7 Учимся программировать: функции обратного вызова . . . 305
18.8 Надёжный узел . . . 305
18.9 Home Assistant: автоматическое обнаружение . . . 309
18.10 Безопасность . . . 310

19 Bluetooth Low Energy 313
19.1 Идея урока коротко . . . 313
19.2 BLE и «классический» Bluetooth . . . 313
19.2.1 Радиоканалы BLE . . . 313
19.3 Стек BLE: GAP и GATT . . . 314
19.3.1 Роли . . . 314
19.4 Реклама: как устройство «кричит» о себе . . . 315
19.5 GATT: как устроены данные . . . 316
19.5.1 Как закодирована температура . . . 317
19.6 Чтение, запись и уведомления . . . 317

19.7 MicroPython: низкоуровневый API и aioble . . . 318
19.8 Практика: термометр . . . 318
19.8.1 Свои сервисы: светодиод с телефона . . . 321
19.8.2 Учимся программировать: байты и `struct` . . . 323
19.9 Nordic UART Service: «последовательный порт» по BLE . . . 323
19.10 Центральное устройство: плата сканирует эфир . . . 326
19.11 Одновременная работа Wi-Fi и BLE . . . 327
19.12 Безопасность и приватность . . . 327
19.13 Практика: BLE-метеопульт . . . 328

20 Моторы, реле и драйверы 329
20.1 Идея урока коротко . . . 329
20.2 Почему мотор нельзя питать от вывода . . . 330
20.3 Индуктивная нагрузка и защитный диод . . . 330
20.4 Транзисторный ключ . . . 331
20.4.1 Расчёт резистора базы . . . 332
20.4.2 MOSFET: ключ, управляемый напряжением . . . 332
20.4.3 Монтажная схема: мотор через BC337 . . . 333
20.4.4 Управление скоростью . . . 334
20.5 Учимся программировать: классы и объекты . . . 336
20.6 Реле: выключатель с электромагнитом . . . 336
20.7 H-мост: направление и скорость . . . 338
20.7.1 Монтажная схема: DRV8833 и мотор . . . 340
20.8 Шаговый двигатель . . . 342
20.8.1 Мотор 28BYJ-48 и плата ULN2003 . . . 343
20.9 Сервопривод . . . 347
20.10 Питание и общая земля . . . 348
20.11 Помехи и конденсаторы . . . 349
20.12 Практика: машина-гусеничка . . . 349

21 Звук и музыка 351
21.1 Идея урока коротко . . . 351
21.2 Что такое звук . . . 351
21.3 Ноты и их частоты . . . 352
21.4 Чем издать звук: активный и пассивный излучатель . . . 353
21.5 Тон средствами ШИМ . . . 354
21.6 Гамма и номера нот . . . 355
21.7 Мелодии: список нот и длительностей . . . 356
21.8 Громкость: скважность . . . 358
21.9 Монтажная схема: зуммер и кнопки-«пианино» . . . 360
21.10 Учимся программировать: классы и неблокирующий проигрыватель . . . 362
21.11 Усиление: динамик и транзистор . . . 366
21.12 ШИМ как цифро-аналоговый преобразователь . . . 367
21.13 Цифровой звук: отсчёты, WAV и I2S . . . 367
21.13.1 Синусоида через I2S . . . 368
21.13.2 Воспроизведение WAV-файла . . . 369
21.14 Усилитель PAM8403 . . . 371
21.15 Практика: четыре шага к музыкальной шкатулке . . . 372

22 USB: клавиатура, мышь, диск, консоль 374
22.1 Идея урока коротко . . . 374
22.2 Как устроено USB . . . 374
22.2.1 Дескрипторы: паспорт устройства . . . 375
22.3 Классы: что умеет Pico . . . 375
22.4 USB-консоль (CDC) . . . 376
22.4.1 Перезагрузка в BOOTSEL из программы . . . 377
22.5 Режимы USB в CircuitPython: файл `boot.py` . . . 378
22.6 HID: клавиатура, мышь, кнопки громкости . . . 380
22.6.1 Клавиатура в CircuitPython . . . 380
22.6.2 Мышь . . . 381
22.6.3 Кнопки громкости и мультимедиа . . . 382
22.7 Учимся программировать: таблица «кнопка → действие» . . . 382
22.8 Практика: макропад из четырёх кнопок . . . 382
22.8.1 Макропад в MicroPython: честно о возможностях . . . 385
22.9 Этика и безопасность: BadUSB . . . 387
22.10 MIDI-контроллер . . . 387
22.11 Питание и безопасность USB . . . 389

23 Отладка и надёжность 391
23.1 Идея урока коротко . . . 391
23.2 Методика: ошибка — это загадка, а не наказание . . . 391
23.3 Читаем traceback . . . 392
23.4 try, except, finally: пусть ошибка не убивает программу . . . 395
23.5 Учимся программировать: декораторы . . . 398
23.6 print-отладка и журнал . . . 399
23.7 Память: куча, сборщик мусора и «утечки» . . . 402
23.8 Мультиметр: первый прибор . . . 404
23.9 Логический анализатор: видим то, чего не видит print . . . 405
23.10 Аппаратная отладка: SWD и Raspberry Pi Debug Probe . . . 407
23.11 Тесты: пусть компьютер проверяет вас . . . 409
23.12 Сторожевой таймер: «собака», которая перезагрузит плату . . . 411
23.13 CircuitPython: безопасный режим и восстановление . . . 413
23.13.1Полный сброс flash . . . 415
23.14 Чек-лист «ничего не работает» . . . 415

24 RP2350, Pico 2 и Pico 2 W 417
24.1 Идея урока коротко . . . 417
24.2 Что нового: RP2040 и RP2350 рядом . . . 418
24.3 Два типа ядер: Arm и RISC-V . . . 419
24.4 Новые возможности чипа . . . 420
24.4.1 Три блока PIO и HSTX . . . 420
24.4.2 Безопасность: TrustZone, безопасная загрузка, OTP . . . 421
24.5 Корпуса RP2350A и RP2350B . . . 422
24.6 Прошивки для Pico 2 и Pico 2 W . . . 422
24.7 Миграция проекта с Pico на Pico 2 . . . 423
24.8 Практика: бенчмарк на Pico и Pico 2 . . . 424
24.9 Эрратум E9: подтяжка входа GPIO . . . 426
24.10 Сравнение плат семейства . . . 429
24.11 Учимся программировать: определяем железо в коде . . . 430

25 Итоговый проект: метеостанция с экраном, Wi-Fi и MQTT 432
25.1 Постановка задачи . . . 432
25.2 Архитектура . . . 433
25.3 Детали и бюджет ресурсов . . . 434
25.4 Монтажная схема . . . 435
25.5 Разбиение на модули . . . 437
25.6 Конечный автомат режимов экрана . . . 437
25.7 Полный код: MicroPython . . . 438
25.7.1 Как устроен главный цикл . . . 446
25.8 Версия для CircuitPython . . . 447
25.9 Ошибки и переподключения . . . 449
25.10 Веб-страница, JSON и MQTT . . . 450
25.11 Журнал на flash . . . 450
25.12 Этапы сборки и тестирования . . . 451
25.13 Критерии оценивания . . . 452
25.14 Идеи расширения . . . 452
25.15 Защита проекта . . . 452

A Справочник 455
A.1 Распиновка Pico . . . 455
A.1.1 Функции выводов по таблице . . . 455
A.1.2 Отличия моделей . . . 458
A.2 Электрические характеристики . . . 458
A.3 MicroPython и CircuitPython: сравнение API . . . 458
A.4 Коды цветов резисторов . . . 461
A.5 Частоты нот . . . 461
A.6 Таблица ASCII . . . 462
A.7 Единицы измерения и приставки . . . 463
A.8 Основные формулы . . . 463
A.9 Команды инструментов . . . 464
A.9.1 mpremote (MicroPython) . . . 464
A.9.2 circup (CircuitPython) . . . 464
A.9.3 picotool (аналог esptool для Pico) . . . 464
A.10 Полезные ссылки . . . 465

B Детали и инструменты 466
B.1 Какую плату выбрать . . . 466
B.2 Каталог деталей . . . 466
B.3 Комплекты . . . 466
B.4 Инструменты . . . 468
B.5 Советы по покупке . . . 468
B.6 Организация рабочего места . . . 469

C Словарь терминов 469
C.1 Латинские обозначения и названия . . . 469
C.2 Русские термины . . . 471

D Python за один вечер 473
D.1 Как запускать программы . . . 473
D.2 Переменные и типы . . . 473
D.3 Строки, форматирование, ввод и вывод . . . 474
D.4 Условия . . . 475
D.5 Циклы . . . 476

D.6 Списки, кортежи и словари . . . 477
D.7 Функции . . . 478
D.8 Исключения . . . 478
D.9 Классы (основы) . . . 479
D.10 Чем MicroPython отличается от обычного Python . . . 480
D.11 Чем отличается CircuitPython . . . 481
D.12 Упражнения . . . 481

E Ответы на вопросы для самопроверки 482
Урок 1. Первый запуск . . . 482
Урок 2. Цифровой выход . . . 483
Урок 3. Цифровой вход . . . 483
Урок 4. Время и конечные автоматы . . . 483
Урок 5. ШИМ . . . 483
Урок 6. АЦП . . . 484
Урок 7. UART . . . 484
Урок 8. I^2C . . . 484
Урок 9. SPI . . . 484
Урок 10. Датчики . . . 484
Урок 11. WS2812 . . . 485
Урок 12. Память и файлы . . . 485
Урок 13. Многозадачность . . . 485
Урок 14. PIO . . . 485
Урок 15. Питание, сон, WDT . . . 486
Урок 16. Wi-Fi . . . 486
Урок 17. Веб-сервер . . . 486
Урок 18. MQTT . . . 486
Урок 19. BLE . . . 487
Урок 20. Моторы . . . 487
Урок 21. Звук . . . 487
Урок 22. USB . . . 487
Урок 23. Отладка . . . 487
Урок 24. RP2350 . . . 488
Урок 25. Итоговый проект . . . 488

F Для педагога: планирование и организация курса 488
F.1 Для кого и на сколько часов рассчитан курс . . . 488
F.2 Структура занятия . . . 489
F.3 Тематическое планирование (стандартный вариант, 72 часа) . . . 489
F.4 Оборудование на группу . . . 490
F.5 Подготовка класса: чек-лист . . . 491
F.6 Критерии оценивания . . . 491
F.7 Итоговый проект и защита . . . 491
F.8 Типичные трудности и что делать . . . 491
F.9 Работа с двумя языками . . . 491

Техника безопасности и рабочее место

Опыты этого курса идут на безопасном напряжении $3{,}3$ – 5 V. Ударить током оно не может, но испортить плату, обжечь палец о перегретую деталь или повредить аккумулятор можно за секунду. Правила ниже короткие. Прочитайте их до первой сборки и возвращайтесь к ним перед каждым новым уроком.

Десять правил

1. **Сначала отключите питание, потом меняйте схему.** Выньте USB-кабель, пересоберите, проверьте по рисунку и только после этого включайте.
2. **На выводы GPIO — не больше** $3{,}3$ **V.** Контакт VBUS (5 В) — это питание, а не сигнал.
3. **Не замыкайте 3V3 и VBUS на GND.** Оголённые концы проводов и металлические предметы на столе — частая причина короткого замыкания.
4. **Деталь греется или пахнет — сразу отключите питание.** Потом ищите ошибку: чаще всего перепутаны плюс и минус.
5. **Светодиод подключайте через резистор, а мотор и реле — через ключ или драйвер** (уроки 2 и 20).
6. **Сетевое напряжение** 220 V **в учебных опытах не используется.** Никогда не подключайте к макетной плате провода от розетки и не разбирайте блоки питания.
7. **Литиевые аккумуляторы требуют особой осторожности:** не замыкать, не прокалывать, не заряжать без платы защиты и без присмотра (урок 15). Вздувшийся аккумулятор отдайте педагогу.
8. **Паяльник — только с разрешения педагога.** Жало нагрето выше $300\,^{\circ}$C: кладите паяльник только на подставку, работайте в проветриваемом помещении, после пайки мойте руки.
9. **Берегите глаза.** Откусывая выводы деталей, придерживайте их и направляйте вниз, от себя и от соседей. Не смотрите на яркие светодиоды в упор.
10. **Порядок на столе.** Никакой еды и напитков рядом с электроникой; в конце занятия отключите питание и разложите детали по ячейкам.

Осторожно

Статическое электричество способно повредить микросхему незаметно для вас. Перед работой коснитесь заземлённого металлического предмета, не трогайте контакты модуля камеры и чипа пальцами и держите платы за края.

Если что-то пошло не так

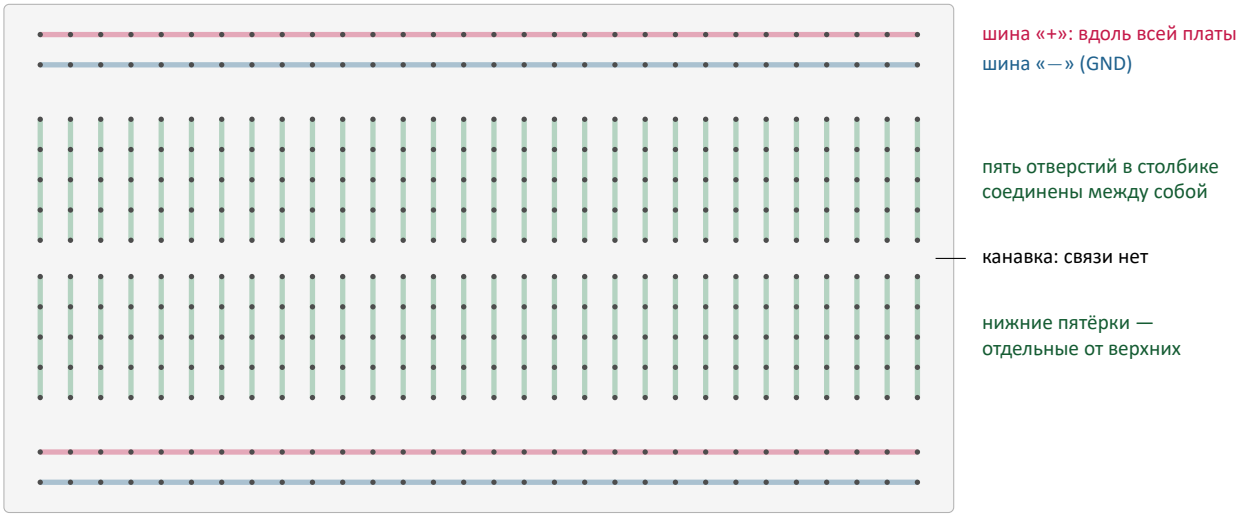
- **Дым, искры, запах гари** — отключите питание, отойдите, позовите педагога. Не пытайтесь потушить или разобрать устройство самостоятельно.
- **Ожог** — подержите обожжённое место под прохладной водой и сообщите педагогу.
- **Плата перестала определяться компьютером** — отключите всё от её выводов и проверьте кабель; чаще всего виновато короткое замыкание в схеме, а не «сгоревшая» плата.

Макетная плата, схема и мультиметр

Три инструмента понадобятся в каждом уроке: макетная плата, на которой собирают схему без пайки, принципиальная схема, по которой её собирают, и мультиметр, которым проверяют результат.

Как устроена макетная плата

Под отверстиями макетной платы спрятаны металлические полоски. Они соединяют отверстия группами, и это нужно запомнить один раз.



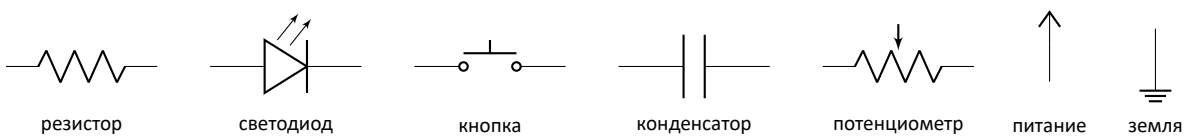
- **Шины питания** по краям соединены *вдоль* платы. К красной подключают 3V3, к синей — GND.
- **Рабочее поле** соединено *поперёк*, столбиками по пять отверстий. Если вставить две ножки детали в один столбик, они окажутся замкнуты.
- **Канавка** посередине разделяет верхние и нижние пятёрки. Плату Pico и микросхемы ставят над канавкой, чтобы ножки левого и правого ряда не соединялись.

Осторожно

На многих длинных макетных платах шины питания разорваны посередине: левая и правая половины не соединены. Если половина схемы «не получает питание», поставьте перемычки между половинами шин или проверьте их мультиметром в режиме прозвонки.

Как читать схему

Принципиальная схема показывает не внешний вид деталей, а их соединения. Линия — это провод, точка на пересечении линий — соединение. Чаще всего в книге встречаются такие обозначения:



Все значки «земля» на схеме соединены между собой, даже если линия между ними не нарисована. То же относится к одинаково подписанным питаниям (3,3 В). У светодиода треугольник показывает направление тока: от анода (длинная ножка) к катоду.

Номинал резистора можно прочитать по цветным полоскам: первые две — цифры, третья — число нулей.

чёрный — 0	коричневый — 1	красный — 2	оранжевый — 3	жёлтый — 4
зелёный — 5	синий — 6	фиолетовый — 7	серый — 8	белый — 9

Например, «красный, красный, коричневый» — это 2, 2 и один ноль: $220\,\Omega$. «Коричневый, чёрный, оранжевый» — 1, 0 и три нуля: $10\,\text{k}\Omega$. Золотая или серебряная полоска в конце показывает точность. Если сомневаетесь, измерьте резистор мультиметром.

Мультиметр: четыре измерения

Что измеряем	Режим	Как подключать щупы
Напряжение	V с прямой чертой (DC), предел 20 В	*параллельно*: чёрный щуп на GND, красный — в точку схемы; питание включено
Целостность цепи	прозвонка (значок звука или диода)	к двум концам провода или дорожки; **питание выключено**
Сопротивление	Ω	к выводам детали, вынутой из схемы; **питание выключено**
Ток	mA или A (DC)	*последовательно*, в разрыв цепи; красный щуп иногда переставляют в другое гнездо

Осторожно

В режиме измерения тока мультиметр ведёт себя как кусок провода. Если в этом режиме коснуться щупами плюса и минуса источника, получится короткое замыкание. После измерения тока сразу возвращайте переключатель и красный щуп в положение для напряжения.

Первое, что делают при любой неполадке, — измеряют питание: между 3V3 и GND должно быть около $3{,}3\,\text{V}$, между VBUS и GND — около $5\,\text{V}$. Если напряжения нет, искать ошибку в программе бесполезно.

Совет

Нет платы или деталей под рукой? Многие опыты первых уроков можно собрать в браузерном симуляторе Wokwi (`https://wokwi.com`): там есть Raspberry Pi Pico и Pico W, светодиоды, кнопки, дисплеи и консоль MicroPython и CircuitPython. Симулятор не заменит настоящую сборку, но помогает проверить программу и схему до занятия.

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Сформулируйте вслух три правила безопасности, которые нарушать нельзя, и объясните соседу, почему.
2. ★ Найдите на макетной плате ряды питания и покажите, какие отверстия в одном столбце соединены между собой.
3. ★★ Определите по полоскам номиналы трёх резисторов и проверьте их мультиметром. Совпали ли значения? Почему допустимо небольшое расхождение?
4. ★★ Измерьте напряжения 3V3–GND и VBUS–GND при питании от USB и запишите результаты: они пригодятся при поиске неисправностей.
5. ★★★ Соберите на макетной плате цепь «3V3 — резистор — светодиод — GND», нарисуй-

те её схему и объясните, в какую сторону включён светодиод.

6. ★★★★ Мультиметром в режиме прозвонки найдите, не разорваны ли шины питания посередине вашей платы, и составьте карту: какие участки соединены, а какие нет.

Знакомство: микроконтроллер, Pico и семейство RP2

Микроконтроллер и компьютер: в чём разница

Raspberry Pi Pico — это **микроконтроллерная плата**. Не путайте её с Raspberry Pi 5 или обычным компьютером. У компьютера есть операционная система, диск, экран, браузер, и он умеет запускать тысячи разных программ одновременно. У микроконтроллера всё иначе: одна программа, нет операционной системы, почти нет «лишнего». Зато он включается за доли секунды, потребляет миллиамперы, работает месяцами от батарейки и напрямую управляет лампами, моторами, датчиками.

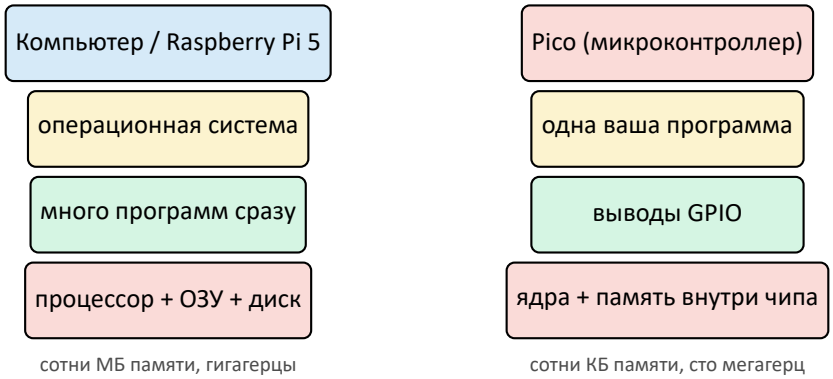


Рис. 0.1: Компьютер и микроконтроллер: слои, которые есть и которых нет

Главное достоинство микроконтроллера — **предсказуемость**: программа делает ровно то, что написана, и в нужный момент. Поэтому он управляет стиральной машиной, термостатом, пультом телевизора, дроном.

Из чего состоит Pico

Сердце платы — чип **RP2040** (Raspberry Pi разработала его сама). Рядом — микросхема внешней flash-памяти, в которой хранятся ваша программа и файлы, преобразователь питания и разъём micro-USB.

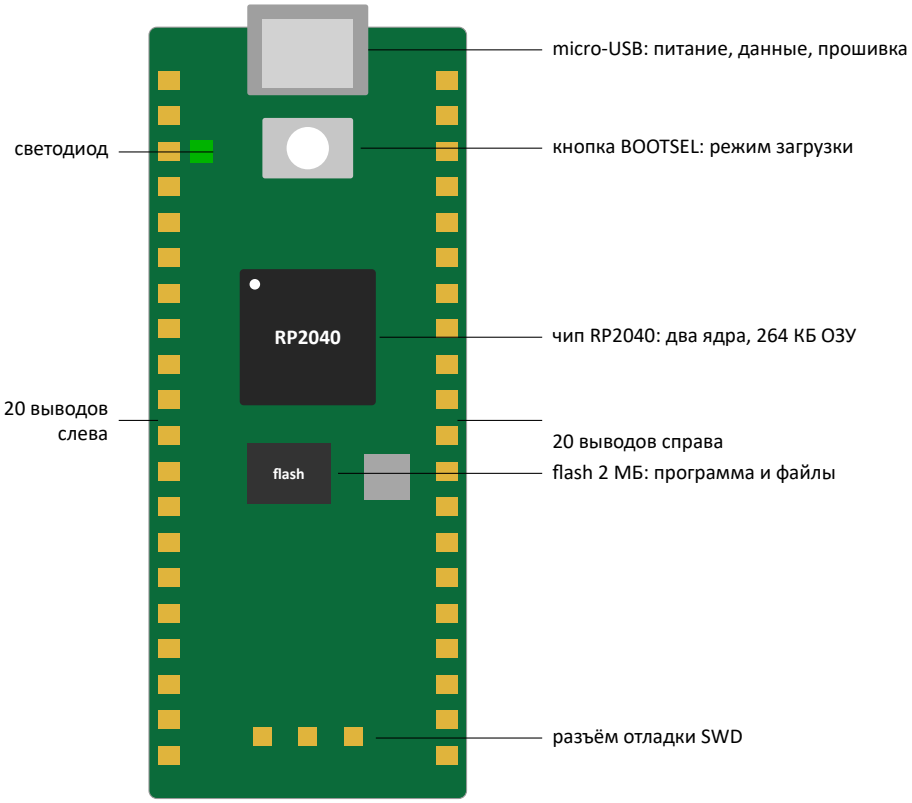


Рис. 0.2: Что где на плате Raspberry Pi Pico (вид сверху, USB вверху)

Блок-схема RP2040

Внутри чипа несколько независимых блоков. Они соединены общей шиной, поэтому ядра могут одновременно обращаться к памяти и периферии.

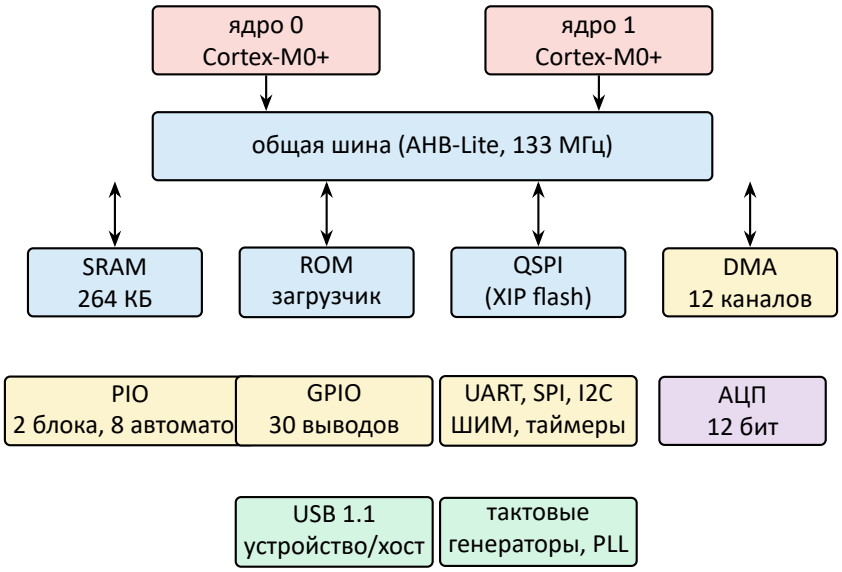


Рис. 0.3: Упрощённая блок-схема микроконтроллера RP2040

Ядра работают независимо: на одном можно читать датчики, на другом — рисовать на экране. **PIO** — особый блок: это маленькие программируемые автоматы, которые генерируют и принимают сигналы с точностью до одного такта, не нагружая ядра (урок 14).

Семейство Pico

Плата	Чип	Связь	Примечание
Pico	RP2040	нет	2 МБ flash; светодиод на GP25
Pico H	RP2040	нет	то же, с распаянными контактами и разъёмом SWD
Pico W	RP2040 + CYW43439	Wi-Fi, Bluetooth LE	светодиод подключён к радиочипу
Pico WH	RP2040 + CYW43439	Wi-Fi, Bluetooth LE	Pico W с распаянными контактами
Pico 2	RP2350	нет	4 МБ flash, быстрее, больше памяти
Pico 2 W	RP2350 + CYW43439	Wi-Fi, Bluetooth LE	Pico 2 с радиомодулем

Вы слышали про «RP2400»? Такого чипа нет: правильное название — **RP2040**. А его следующее поколение называется **RP2350**. Буква W в названии платы означает Wireless (беспроводная).

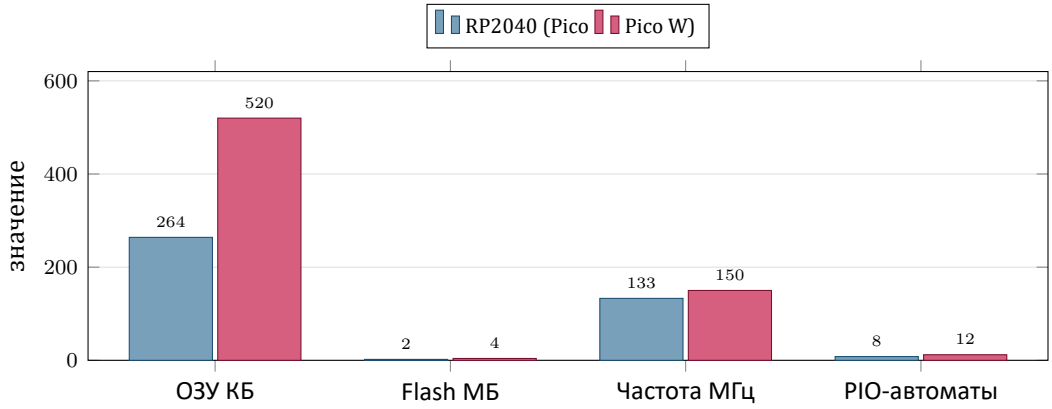


Рис. 0.4: Сравнение RP2040 и RP2350 (flash — объём на плате Pico соответствующего поколения)

	RP2040	RP2350
Ядра	2× Arm Cortex-M0+	2× Arm Cortex-M33 *или* 2× Hazard3 RISC-V
Частота	133 МГц (штатно)	150 МГц
ОЗУ	264 КБ	520 КБ
Вещественные числа	программно (медленно)	аппаратный FPU (одинарная точность)
PIO	2 блока, 8 автоматов	3 блока, 12 автоматов
Безопасность	нет	TrustZone, защищённая загрузка, OTP
Видео/быстрый вывод	нет	HSTX

Таблица 0.1: Главные отличия двух поколений (подробнее — в уроке 24)

Совет

Для обучения разница между Pico и Pico 2 почти незаметна: программы на MicroPython и CircuitPython переносятся практически без изменений. Если выбираете плату сейчас, берите **Pico 2 W** или **Pico W**: Wi-Fi и Bluetooth пригодятся почти всегда.

На каком языке программировать

Платы Pico понимают много языков. В этом курсе их два, и оба — это Python:

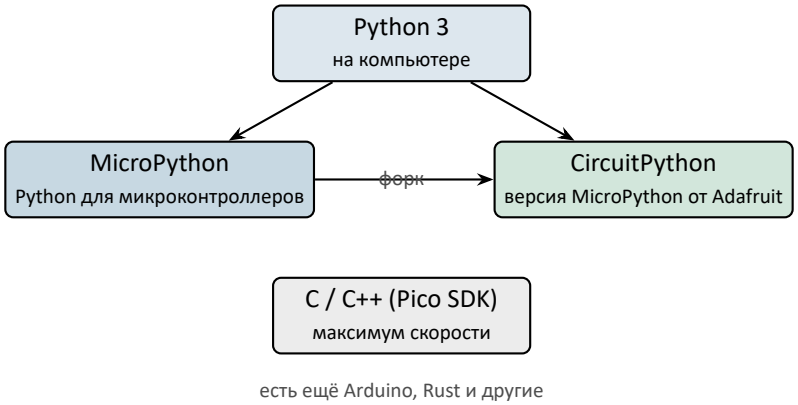


Рис. 0.5: Родословная языков для Pico

	MicroPython	**CircuitPython**
Как попадает код на плату	через Thonny, `mpremote` или другую IDE	копируется как файл на диск `CIRCUITPY`
Файл запуска	`main.py`	`code.py`
Выводы	`machine.Pin`	`board` и `digitalio`
Библиотеки	встроенные модули, `mip`	огромный набор Adafruit, `circup`
Многозадачность	потоки и `asyncio`	только `asyncio`
Сильная сторона	гибкость, PIO, скорость	простота и готовые драйверы

В каждом уроке код приводится для обоих языков. Выберите один и дойдите до конца курса, а второй изучайте по сравнению: основная логика одна и та же.

Как работает программа на Pico

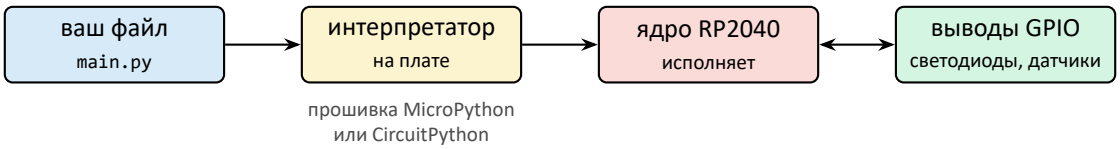


Рис. 0.6: Путь от файла с программой до светодиода

На плате лежат два слоя: **прошивка** (сам интерпретатор Python, заливается один раз файлом `.uf2`) и **ваша программа** (файл `main.py` или `code.py`, который вы меняете сколько угодно). Прошивку не нужно переустанавливать при каждом изменении программы.

Выводы платы: распиновка, питание, правила

Плата Pico имеет 40 контактов по двум длинным сторонам. Главное, что нужно запомнить сразу: в программах мы называем выводы **номерами GP** (GP0, GP1, …), а не номерами контактов на плате (1–40). Контакт 17 на плате — это GP13. Поэтому всегда держите перед глазами распиновку.

Полная распиновка

Читать рисунок просто: найдите нужный вывод и посмотрите на метки. Например, GP4 может быть передатчиком UART1, линией данных I^2C0 или входом данных SPI0 (но только *одним из них* в каждый момент).

Питание платы

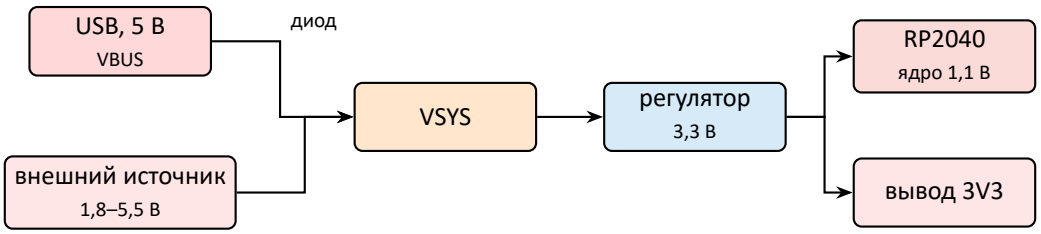


Рис. 0.8: Как питание попадает на плату: два источника объединяются через диод в линии VSYS

Контакт	Напряжение	Назначение
VBUS (40)	около 5 В	напряжение с USB-разъёма, когда кабель подключён
VSYS (39)	1,8–5,5 В	основной вход питания платы (от батареи или блока питания)
3V3 (36)	3,3 В	*выход* регулятора для питания датчиков (ток ограничен)
3V3_EN (37)	—	замкните на GND, чтобы выключить регулятор 3,3 В
RUN (30)	—	замкните на GND, чтобы перезапустить плату
ADC_VREF (35)	3,3 В	опорное напряжение аналого-цифрового преобразователя
AGND (33)	0 В	«чистая» земля для аналоговых измерений

Выводы, которые использовать не нужно

- **GP23, GP24, GP25** на обычной Pico заняты внутри платы (регулятор питания, определение USB, светодиод). На **Pico W и Pico 2 W** через них же работает радиочип Wi-Fi, поэтому к GP25 нельзя обращаться как к светодиоду — используйте имя `"LED"` (урок 2).
- **GP29** читает напряжение питания VSYS и тоже не выведен на контакт.
- GP0–GP22 и GP26–GP28 — свободные выводы, всего **26** штук.

Электрические правила

Осторожно

- Логика Pico — **3,3 В**. Выводы GPIO **не рассчитаны на 5 В**. Датчик с выходом 5 В подключайте через делитель напряжения или преобразователь уровней.
- Ток вывода по умолчанию ограничен примерно 4 мА, максимум (после настройки) — 12 мА. Суммарный ток всех выводов ограничен. Светодиоды — да; моторы, реле, длинные ленты — только через ключ или драйвер.

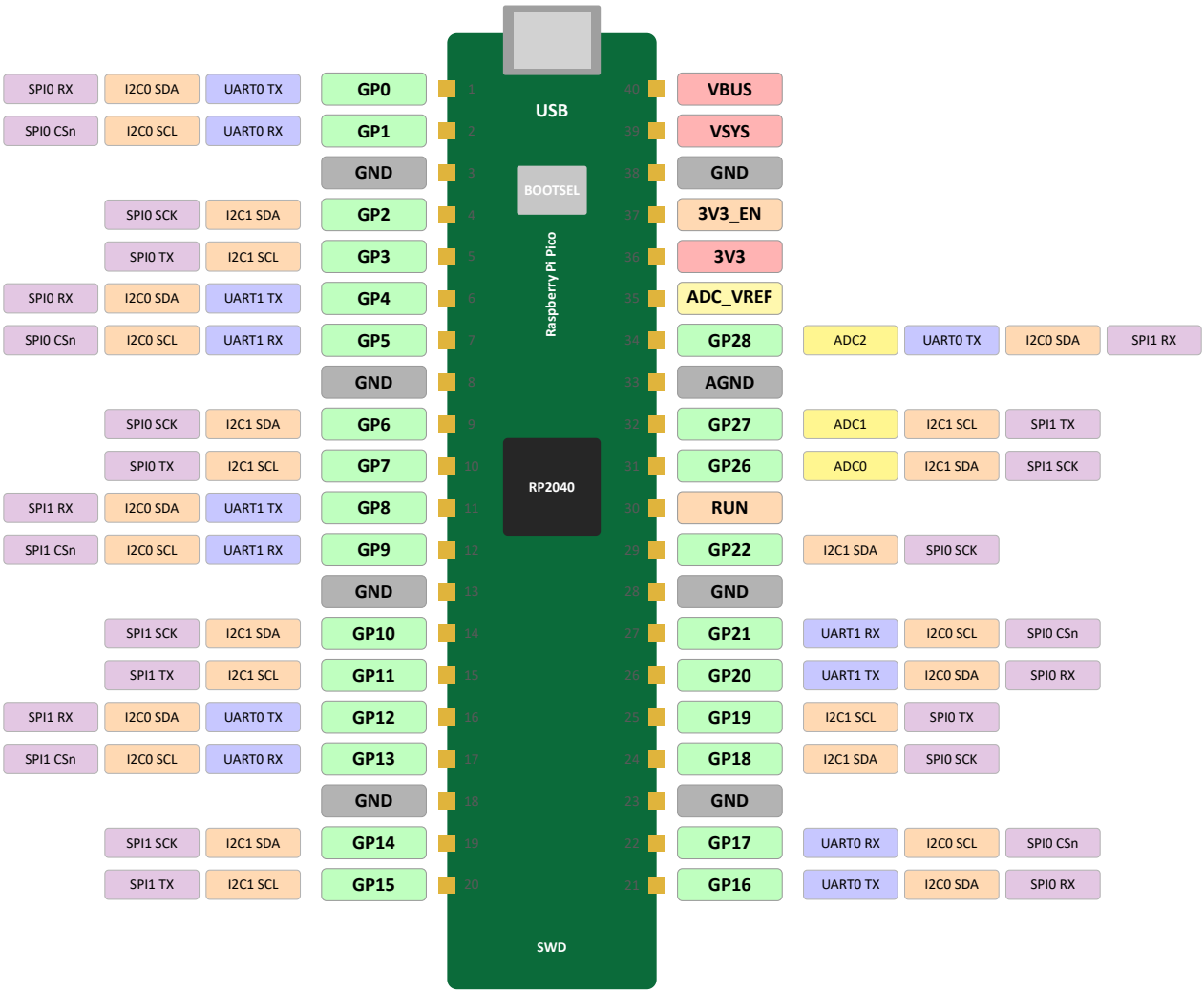


Рис. 0.7: Распиновка Raspberry Pi Pico. Зелёные — выводы общего назначения (любой из них можно использовать как цифровой вход/выход и ШИМ), красные — питание, чёрные — земля. Цветные метки показывают, какие интерфейсы можно подключить к выводу

- Вывод 3V3 может отдать лишь небольшой ток (порядка 300 мА вместе с платой). Мощные нагрузки питайте отдельно, соединив земли.
- Не подавайте напряжение на 3V3 и не замыкайте его на GND.

Для каких целей какие выводы выбирать

Задача	Выводы	Комментарий
Светодиоды, реле, зуммер	любые GP	всё, что не нужно для шин
Кнопки	любые GP	подтяжка включается программно
ШИМ	любой GP	8 «слайсов» по 2 канала, у пары общая частота (урок 5)
Аналоговые входы	GP26, GP27, GP28	каналы ADC0, ADC1, ADC2 (урок 6)
I^2C (удобные пары)	GP4/GP5 (I2C0), GP2/GP3 (I2C1)	SDA и SCL строго по таблице
SPI0	GP16 RX, GP17 CSn, GP18 SCK, GP19 TX	классический набор
UART0	GP0 TX, GP1 RX	удобные соседние выводы

Таблица 0.2: Рекомендуемые выводы для типичных задач

Совет

Заведите привычку: перед сборкой схемы выпишите, к какому GP подключён каждый провод. Ошибки в номерах выводов — самая частая причина «не работает».

Урок 1. Первый запуск: прошивка, Thonny, REPL

Цель	установить прошивку MicroPython или CircuitPython, подключиться к плате, выполнить первые команды и запустить программу, которая стартует сама
Опирается на	вводные главы: техника безопасности, плата и распиновка
Результат	на плате работает выбранная прошивка; вы умеете открыть консоль (REPL), сохранить `main.py` (`code.py`) и остановить программу
Понадобится	плата Pico или Pico W, кабель micro-USB *с данными*, компьютер с Windows, macOS или Linux

1.1 Идея урока коротко

Плата Pico при покупке «пустая»: в ней нет интерпретатора Python. Его нужно один раз загрузить — это и есть **прошивка**. Загрузка делается без программатора: плата сама притворяется USB-флешкой, и файл с прошивкой достаточно скопировать на неё. После этого вы пишете и запускаете программы на Python.



Рис. 1.1: Шесть шагов от новой платы до первой программы

1.2 Выбираем прошивку

В этом курсе используются два варианта прошивки. Выберите *один*, начните с него, а второй можно поставить позже — плата от этого не пострадает.

	MicroPython	**CircuitPython**
Сайт	`micropython.org/download`	`circuitpython.org/downloads`
Pico	`RPI_PICO`	Raspberry Pi Pico
Pico W	`RPI_PICO_W`	Raspberry Pi Pico W
Pico 2 / 2 W	`RPI_PICO2` / `RPI_PICO2_W`	Raspberry Pi Pico 2 / 2 W
Как работать	консоль и файлы через Thonny	плата видна как диск `CIRCUITPY`
Что выбрать новичку	если хотите гибкость и Wi-Fi-примеры из книги	если любите «просто скопировал и работает»

Осторожно

Прошивка для **Pico** и для **Pico W** *разная*. На Pico W с прошивкой для обычной Pico не заработают Wi-Fi и встроенный светодиод. Сверьте название платы на сайте и надпись на самой плате. Платы поколения RP2350 (Pico 2) требуют своих файлов — RP2040-прошивки на них не запустятся.

1.3 Режим BOOTSEL: загружаем прошивку

Кнопка **BOOTSEL** переводит плату в режим загрузки. Делайте так:

1. Отсоедините USB-кабель от платы.
2. Нажмите и *держите* кнопку BOOTSEL.
3. Не отпуская кнопку, вставьте USB-кабель в компьютер. Теперь можно отпустить кнопку.
4. На компьютере появится новый диск **RPI-RP2** (для Pico 2 — **RP2350**).
5. Скопируйте на диск файл `.uf2` с прошивкой (перетащите мышью).
6. Диск исчезнет сам — плата перезагрузилась и готова к работе.

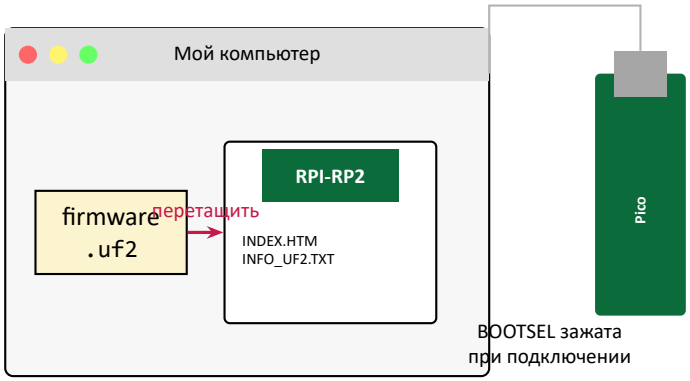


Рис. 1.2: В режиме BOOTSEL плата выглядит как USB-флешка. Файл прошивки копируется на неё обычным способом

Если не работает

- **Диск не появляется.** Кабель только для зарядки — возьмите другой. Или кнопку отпустили до подключения USB: повторите.
- **Скопировал файл, диск остался.** Файл не `.uf2` (например, скачали `.zip`) — распакуйте архив и скопируйте именно `.uf2`.
- **Плата «зависла» после неудачной программы.** Режим BOOTSEL работает всегда, он зашит в чип. Плату нельзя «окирпичить» программно.

1.4 MicroPython: Thonny и консоль

Thonny (`https://thonny.org`) — бесплатная среда разработки для Python, в которой удобно начинать. Установите её, подключите плату с прошивкой MicroPython и выберите интерпретатор:

1. Нажмите справа внизу на название интерпретатора (или *Инструменты* → *Параметры* → *Интерпретатор*).
2. В списке выберите **MicroPython (Raspberry Pi Pico)** и порт (обычно «Try to detect port automatically»).
3. Внизу окна появится **оболочка** (Shell) с приглашением `>>>` — это **REPL**.

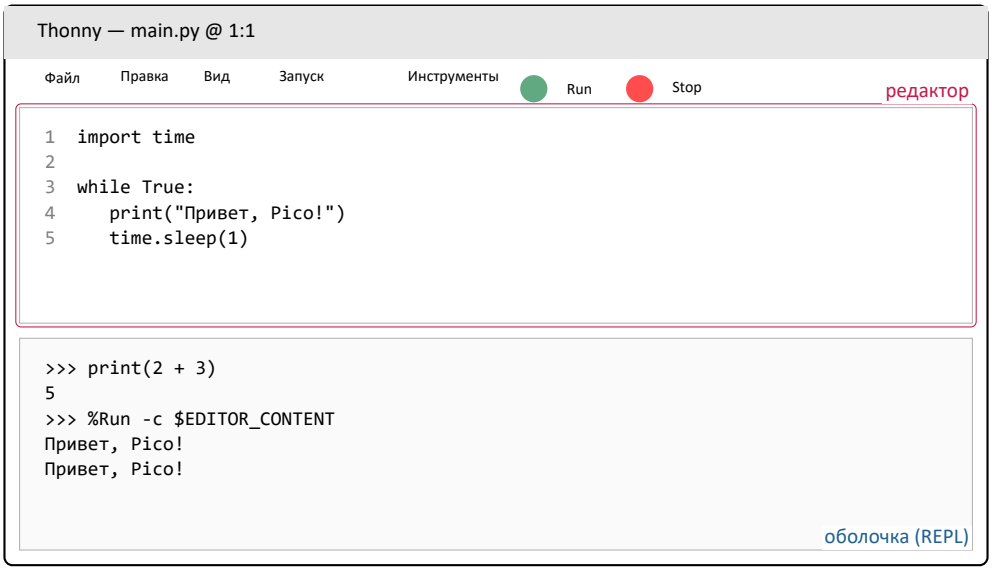


Рис. 1.3: Окно Thonny: сверху редактор программы, снизу оболочка REPL

В оболочке можно выполнять отдельные команды — плата отвечает сразу:

Листинг 1.1: MicroPython: первые команды в REPL (можно вводить по одной)

```
print("Привет, Pico!")
print("2 + 3 =", 2 + 3)

import sys
print(sys.implementation)     # какая прошивка и версия на плате
```

Горячие клавиши REPL

Клавиши	Действие
Ctrl+C	прервать выполняющуюся программу
Ctrl+D	мягкая перезагрузка (выполнится `main.py`)
Ctrl+E	режим вставки многострочного текста
Стрелка вверх	предыдущая команда

1.5 MicroPython: программа, которая запускается сама

Кнопка Run в Thonny выполняет код, но *не сохраняет его* на плате. Чтобы программа стартовала при включении, сохраните её на плате под именем **`main.py`**: *Файл* → *Сохранить как* → *Raspberry Pi Pico*.

Листинг 1.2: MicroPython: файл `main.py`

```
# main.py: запускается автоматически при включении платы
import time

n = 0
while True:
    n += 1
    print("Прошла секунда номер", n)
    time.sleep(1)
```

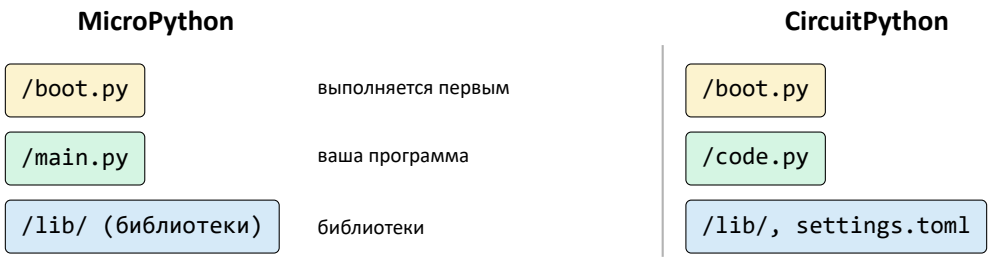


Рис. 1.4: Файлы, которые важны на плате: слева MicroPython, справа CircuitPython

1.6 MicroPython: утилита mpremote

Кому удобнее терминал и VS Code — пользуйтесь `mpremote`, официальным инструментом MicroPython:

Листинг 1.3: Основные команды `mpremote`

```
pip install mpremote             # один раз
mpremote connect list            # какие порты видит компьютер
mpremote                         # открыть REPL (Ctrl+X: выход)
mpremote fs ls                   # файлы на плате
mpremote fs cp main.py :main.py # копировать файл на плату
mpremote run test.py             # выполнить файл, не сохраняя на плате
mpremote reset                   # перезагрузить плату
```

1.7 CircuitPython: плата как диск

Установка проще. После копирования `.uf2` (способ тот же, что в режиме BOOTSEL) плата перезагрузится и появится диск **CIRCUITPY**. На нём лежат файл `code.py`, папка `lib` и другие. Откройте `code.py` в редакторе (Mu, Thonny с интерпретатором CircuitPython, VS Code), вставьте программу и *сохраните*: плата сама перезапустит её.

Листинг 1.4: CircuitPython: файл `code.py`

```
# code.py: запускается автоматически при каждом сохранении файла
import time

n = 0
while True:
    n += 1
    print("Прошла секунда номер", n)
    time.sleep(1)
```

Вывод `print()` смотрят в **последовательной консоли**: в Mu кнопка Serial, в Thonny — оболочка, в терминале — `screen` или PuTTY на порту платы.

Осторожно

Сохраняйте `code.py` *целиком и сразу*. Некоторые редакторы записывают файл частями; плата может перезапуститься на недописанном файле. Если что-то пошло странно, сохраните ещё раз.

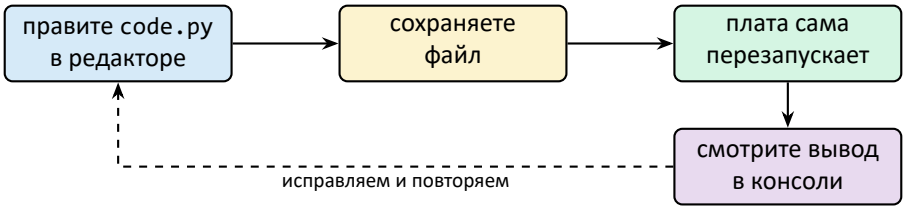


Рис. 1.5: Цикл разработки в CircuitPython: сохранили файл — плата перезапустилась

1.8 Информация о вашей плате

Запустите небольшую программу, которая расскажет, что за прошивка стоит на плате, на какой частоте работает процессор и сколько осталось памяти:

Листинг 1.5: MicroPython: сведения о плате и прошивке

```
import os
import sys
import machine
import gc

print("Прошивка: ", sys.implementation)
print("Версия:   ", sys.version)
print("Частота:  ", machine.freq() // 1_000_000, "МГц")
print("Свободно ОЗУ:", gc.mem_free(), "байт")
st = os.statvfs("/")
print("Свободно flash:", st[0] * st[3] // 1024, "КБ")
```

Листинг 1.6: CircuitPython: сведения о плате и прошивке

```
import os
import sys
import gc
import microcontroller

print("Прошивка: ", sys.implementation)
print("Частота:  ", microcontroller.cpu.frequency // 1_000_000, "МГц")
print("Температура чипа:", microcontroller.cpu.temperature, "°C")
print("Свободно ОЗУ:", gc.mem_free(), "байт")
st = os.statvfs("/")
print("Свободно flash:", st[0] * st[3] // 1024, "КБ")
```

Это интересно

Память Pico делится на два вида: **ОЗУ** (264 КБ у RP2040) — рабочая память программы, содержимое которой исчезает при выключении, и **flash** (2 МБ у Pico) — хранит прошивку и ваши файлы, её содержимое остаётся без питания.

1.9 Сравнение подходов к работе

	MicroPython	CircuitPython
Как попадает код на плату	Thonny/`mpremote` копируют файл по USB-консоли	вы копируете файл на диск `CIRCUITPY`
Файл запуска	`main.py`	`code.py`
Перезапуск программы	Ctrl+D или кнопка Run	сохранение файла (автоматически)
Остановка	Ctrl+C	Ctrl+C в консоли
Если плата не отвечает	BOOTSEL → `flash_nuke.uf2`	BOOTSEL → новая прошивка или `storage.erase_filesystem()`

Таблица 1.1: Как работать с каждой прошивкой

Если не работает

- **Thonny не видит плату.** Закройте другие программы, занимающие порт (терминалы, другой Thonny); отключите и подключите плату; выберите порт вручную в параметрах интерпретатора.
- **Плата не отвечает, программа «зависла».** Нажмите Ctrl+C несколько раз. Если не помогает — BOOTSEL и `flash_nuke.uf2`, затем прошивка заново.
- **Нет диска CIRCUITPY.** Возможно, в `boot.py` отключён USB-диск; перезагрузите плату в режиме BOOTSEL и залейте прошивку ещё раз.
- **Не тот язык.** Если после установки CircuitPython вы открыли Thonny в режиме MicroPython — переключите интерпретатор в параметрах.

Итоги урока

- Прошивка (MicroPython или CircuitPython) ставится копированием файла `.uf2` на диск в режиме BOOTSEL.
- Для Pico, Pico W и Pico 2 нужны *разные* файлы прошивки.
- REPL — консоль, где команды выполняются сразу; Ctrl+C прерывает, Ctrl+D перезапускает.
- `main.py` (MicroPython) и `code.py` (CircuitPython) стартуют автоматически.
- Режим BOOTSEL работает всегда: плату нельзя испортить программой.

Проверь себя

1. Для чего нужна кнопка BOOTSEL и как перевести плату в режим загрузки?
2. Чем прошивка отличается от вашей программы?
3. Почему нельзя использовать прошивку для Pico на Pico W?
4. Что такое REPL и чем он полезен?
5. Как программа может стартовать сама при включении платы в MicroPython и в CircuitPython?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Установите выбранную прошивку и убедитесь, что в REPL команда `print("OK")` отвечает.
2. ★ Определите по программе из урока версию прошивки, частоту процессора и свободную память вашей платы.
3. ★★ Сохраните программу-счётчик как `main.py` (`code.py`), отключите и снова включите плату: убедитесь, что она стартует сама.
4. ★★ Измените программу так, чтобы она выводила не номер секунды, а время работы в минутах и секундах.
5. ★★★ Поставьте на плату второй язык (если работали с MicroPython — CircuitPython и наоборот), запустите тот же счётчик и составьте список отличий в работе.
6. ★★★★ Установите `mpremote` и без Thonny загрузите на плату программу, запустите её и скопируйте с платы файл обратно на компьютер.

Урок 2. Цифровой выход: светодиод

Цель	научиться управлять выводом GPIO как выходом, правильно подключать светодиод через резистор и отмерять время, не останавливая программу
Опирается на	урок 1: прошивка, Thonny, REPL и запуск `main.py` / `code.py`
Результат	светодиод на **GP13** мигает; четыре светодиода образуют «бегущий огонь»; встроенный светодиод платы мигает на любой модели Pico
Понадобится	макетная плата, светодиоды, резистор $220 – 330\,\Omega$, провода

2.1 Идея урока коротко

GP13 создаёт логическую единицу (около 3,3 В) или ноль (около 0 В), а светодиод показывает этот сигнал светом. Току нужен замкнутый путь: GP13 → резистор → светодиод → GND. В **MicroPython** вывод описывает объект `Pin(13, Pin.OUT)`, а значение задают методом `value()`. В **CircuitPython** вывод называется `board.GP13`, им управляет `digitalio.DigitalInOut`.

2.2 Зачем нужен резистор: закон Ома

Электричество удобно сравнивать с водой в трубах:

- **напряжение** U (вольты, В) — это «давление», которое толкает воду;
- **ток** I (амперы, А) — сколько воды протекает за секунду;
- **сопротивление** R (омы, Ω) — насколько узкая труба: чем уже, тем меньше воды пройдёт.

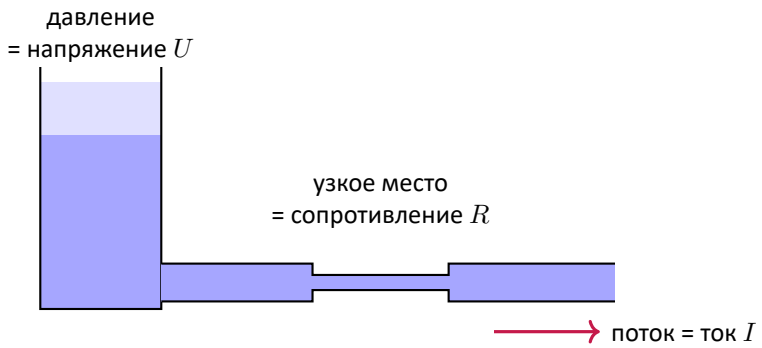


Рис. 2.1: «Водяная» модель электрической цепи

Эти три величины связаны **законом Ома** — главным законом электроники:

$$I = \frac{U}{R} \qquad \text{или} \qquad R = \frac{U}{I}.$$

Больше давление — больше поток. Уже труба — меньше поток.

Почему светодиод сгорает без резистора

Светодиод ведёт себя необычно: пока напряжение на нём маленькое, ток почти не течёт. Но стоит напряжению превысить порог — **прямое напряжение** U_F, для красного светодиода около 2 В, — ток начинает расти *очень* быстро:

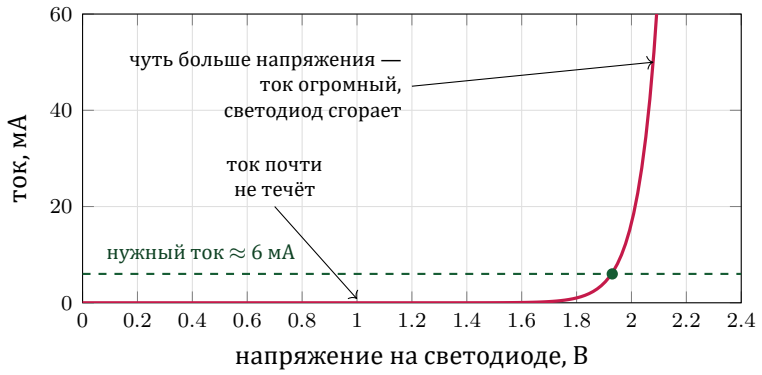


Рис. 2.2: Как ток через красный светодиод зависит от напряжения на нём (вольт-амперная характеристика)

Вывод платы выдаёт 3,3 В — намного больше, чем нужно светодиоду. Если подключить его напрямую, ток окажется огромным, и светодиод (а может, и вывод платы) сгорит. **Резистор** забирает на себя «лишнее» напряжение и ограничивает ток. Поэтому правило:

Светодиод подключается к выводу только через резистор!

Рассчитываем резистор

Из 3,3 В на светодиод придётся $U_F \approx 2{,}0$ В (для красного), значит, на резистор — оставшиеся $3{,}3 - 2{,}0 = 1{,}3$ В. Светодиоду достаточно тока около 6 мА = 0,006 А. По закону Ома:

$$R = \frac{U_{\text{питания}} - U_F}{I} = \frac{3{,}3\ \text{В} - 2{,}0\ \text{В}}{0{,}006\ \text{А}} \approx 217\ \Omega.$$

Резисторы выпускают только определённых номиналов, поэтому берём ближайший: $220\ \Omega$.

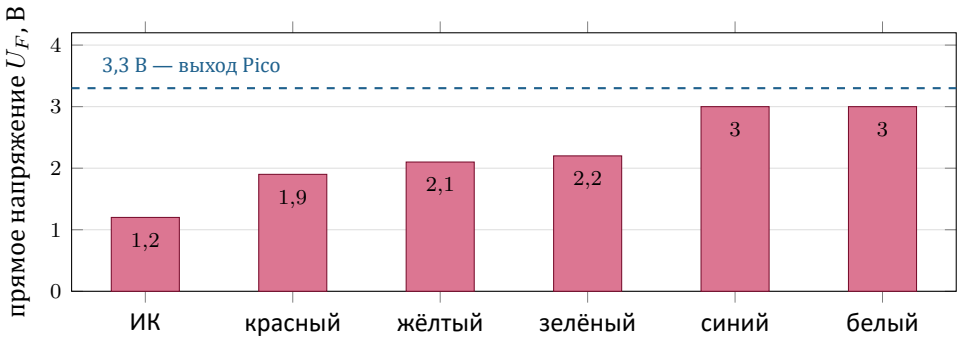


Рис. 2.3: Прямое напряжение светодиодов разных цветов (примерные значения)

Цвет светодиода	U_F, В	На резисторе, В	Резистор для 6 мА
Красный	1,9	$3{,}3 - 1{,}9 = 1{,}4$	$\approx 230\ \Omega \to 220\ \Omega$
Жёлтый, зелёный	2,1–2,2	1,1–1,2	$\approx 190\ \Omega \to 180\ \Omega$–$220\ \Omega$
Синий, белый	3,0	0,3	$\approx 50\ \Omega \to 47\ \Omega$

Таблица 2.1: Пример расчёта резисторов при питании от вывода 3,3 В

Синему и белому светодиоду от 3,3 В достаётся совсем небольшой запас, поэтому они светят слабее. Для опытов лучше всего подходят красные, жёлтые и зелёные светодиоды с резистором $220\ \Omega$.

2.3 Как устроен выход GPIO

Выход GPIO — это пара транзисторов (двухтактная схема, push-pull). Логическая `1` подключает вывод к $3{,}3$ V, логический `0` — к земле.

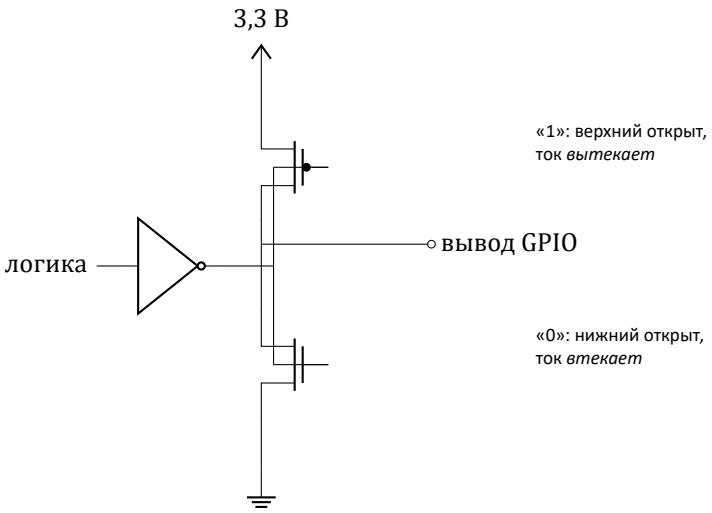


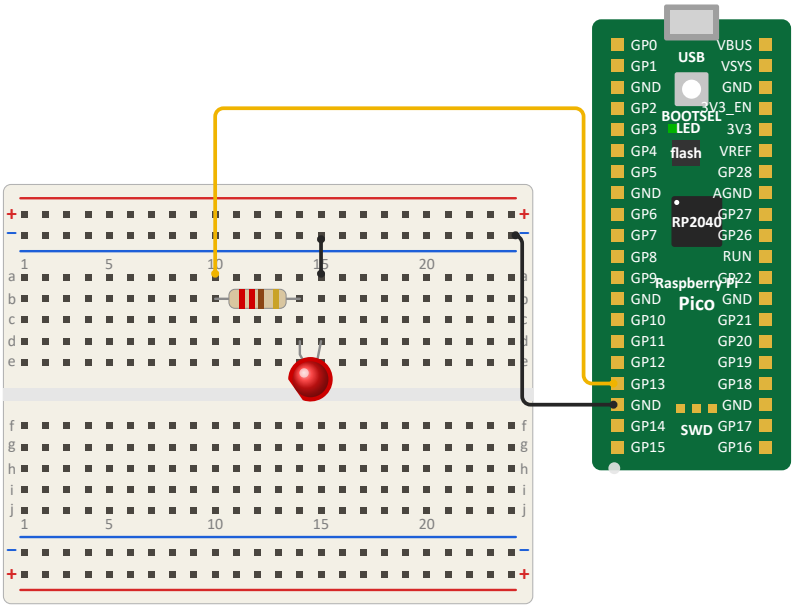
Рис. 2.4: Упрощённая схема выходного каскада GPIO

Осторожно

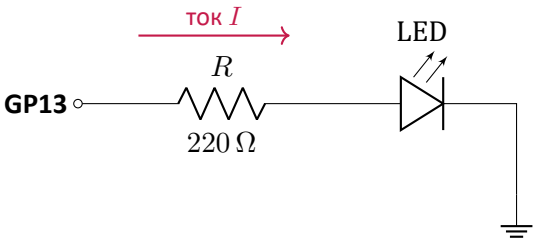
- Pico работает с логикой **3,3 В**. Подача 5 V на вывод может вывести чип из строя.
- Ток одного вывода по умолчанию ограничен примерно 4 мА (его можно поднять программно до 12 мА), а суммарный ток всех выводов не должен быть большим. Светодиоды — да, моторы и реле — только через ключ.
- В программе всегда указывается **номер GP**, а не номер контакта на плате: GP13 стоит на физическом контакте 17.

2.4 Подключаем светодиод

Соберём схему: вывод **GP13** $\rightarrow$ резистор $220\,\Omega$ $\rightarrow$ анод светодиода (длинная ножка); катод (короткая) $\rightarrow$ GND.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 2.5: Подключение светодиода к выводу GP13

Сравни две картинки: жёлтый провод на макетной плате — это линия от **GP13** на схеме, чёрные провода — значок земли. Проследи путь тока пальцем: от вывода платы через резистор и светодиод к земле и обратно в плату — цепь замкнута.

2.5 Классический Blink

Название пришло из мира Arduino: «Hello, world!» для электроники — мигающий светодиод.

Листинг 2.1: MicroPython: мигание светодиодом

```
from machine import Pin
import time

LED_PIN = 13                # номер GP, а не физический номер контакта
led = Pin(LED_PIN, Pin.OUT)

while True:
    led.value(1)            # 3,3 В: светодиод горит
    time.sleep(0.5)
    led.value(0)            # 0 В: светодиод погас
    time.sleep(0.5)
```

Листинг 2.2: CircuitPython: то же самое

```
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True          # 3,3 В: светодиод горит
    time.sleep(0.5)
    led.value = False         # 0 В: светодиод погас
    time.sleep(0.5)
```

Совет

MicroPython: сохраните файл на плате как `main.py`. CircuitPython: сохраните как `code.py` на диск `CIRCUITPY`. Программа стартует сама при включении платы и остановится по Ctrl+C.

2.6 Разберём Blink по строкам

- `LED_PIN = 13` не включает светодиод: это только удобное имя для номера GP13.
- `Pin(13, Pin.OUT)` сообщает плате, что GP13 теперь *выход*. В CircuitPython то же делают две строки: создание `DigitalInOut` и `direction = OUTPUT`.
- `value(1)` (в CircuitPython — `value = True`) даёт на GP13 3,3 В; светодиод загорается.
- `time.sleep(0.5)` ждёт полсекунды. Потом значение 0 — светодиод гаснет, и цикл `while True` начинается снова.

Если светодиод не мигает, сначала проверьте длинную ножку (анод), резистор и провод GND. Не меняйте программу, пока не убедились в схеме.

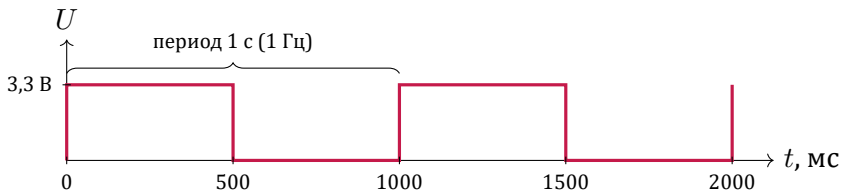


Рис. 2.6: Напряжение на выводе при работе Blink

2.7 Мигание без остановки программы

`time.sleep()` останавливает всю программу: пока идёт ожидание, она не может делать ничего другого. Настоящее устройство одновременно следит за кнопкой, датчиками, экраном и сетью, поэтому сразу привыкаем к другому приёму — сравнивать текущее время с моментом последнего действия. В MicroPython для этого есть `time.ticks_ms()` и `time.ticks_diff()`, в CircuitPython — `time.monotonic()`.

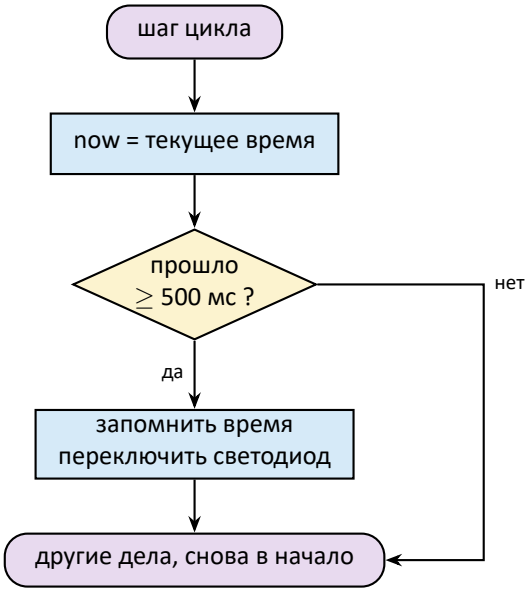


Рис. 2.7: Блок-схема неблокирующего мигания

Листинг 2.3: MicroPython: мигание без `sleep()`

```
from machine import Pin
import time

led = Pin(13, Pin.OUT)
INTERVAL = 500              # мс
last = time.ticks_ms()      # момент последнего переключения

while True:
    now = time.ticks_ms()
    if time.ticks_diff(now, last) >= INTERVAL:
        last = now
        led.toggle()
    # здесь можно спокойно заниматься другими делами
```

Листинг 2.4: CircuitPython: то же

```
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
INTERVAL = 0.5              # секунды
last = time.monotonic()

while True:
    now = time.monotonic()
    if now - last >= INTERVAL:
        last = now
        led.value = not led.value
    # здесь можно спокойно заниматься другими делами
```

Это интересно

Счётчик миллисекунд в **MicroPython** со временем переполняется и «закручивается» по кругу. Если вычитать числа обычным минусом, в момент переполнения получится огромное отрицательное число и программа собьётся. Функция `ticks_diff()` учитывает переполнение, поэтому ею нужно пользоваться всегда.

2.8 Учимся программировать: цикл `for` и списки

Зажечь четыре светодиода по очереди можно, написав четыре одинаковых блока. Но программисты не любят повторяться. Вместо этого их выходы собирают в **список** (в квадратных скобках), а цикл `for` перебирает элементы списка один за другим.

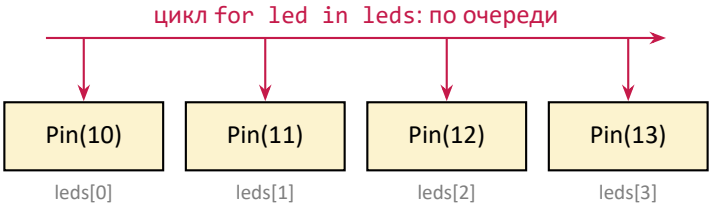
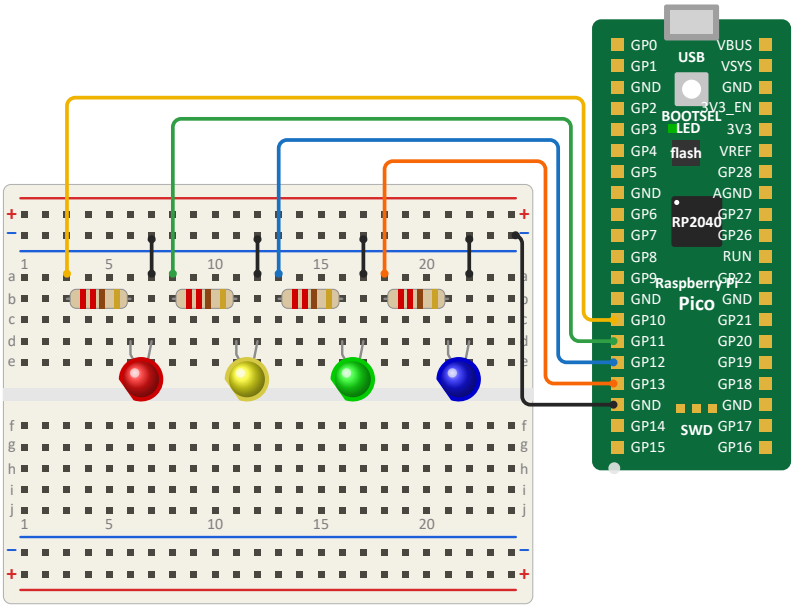


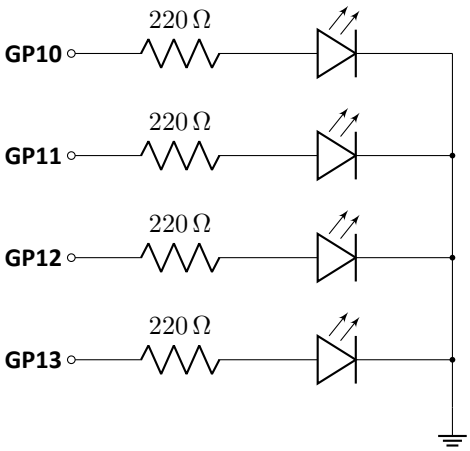
Рис. 2.8: Список из четырёх выходов и цикл по его элементам

2.9 Практика: бегущий огонь

Подключим четыре светодиода к GP10–GP13. Каждому нужен свой резистор $220\ \Omega$, а катоды всех четырёх идут на общую линию GND:



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 2.9: Четыре светодиода на выводах GP10–GP13

Листинг 2.5: MicroPython: бегущий огонь

```
from machine import Pin
import time

leds = [Pin(n, Pin.OUT) for n in (10, 11, 12, 13)]   # список из четырёх выходов

while True:
    for led in leds:              # по очереди зажигаем каждый
        led.value(1)
        time.sleep(0.15)
        led.value(0)
```

Листинг 2.6: CircuitPython: бегущий огонь

```
import time
import board
import digitalio

leds = []
for pin in (board.GP10, board.GP11, board.GP12, board.GP13):
    led = digitalio.DigitalInOut(pin)
    led.direction = digitalio.Direction.OUTPUT
    leds.append(led)

while True:
    for led in leds:
        led.value = True
        time.sleep(0.15)
        led.value = False
```

Обратите внимание: в CircuitPython список выходов строится циклом, потому что у `DigitalInOut` нет такого короткого конструктора, как у `Pin`. В MicroPython то же самое делает **списковое включение** `[Pin(n, Pin.OUT) for n in (10, 11, 12, 13)]`: короткая запись цикла, которая сразу собирает список.

2.10 Встроенный светодиод платы

У Pico и Pico 2 светодиод подключён к GP25, а у Pico W и Pico 2 W он подключён к радиочипу Wi-Fi и *не* лежит на обычном выводе. Поэтому писать `Pin(25)` нельзя: на беспроводных платах такой код не сработает. Универсальное имя работает на любой модели семейства:

Листинг 2.7: MicroPython: встроенный светодиод на любой плате

```
from machine import Pin
import time

led = Pin("LED", Pin.OUT)    # встроенный светодиод: работает на Pico, Pico W, Pico 2, Pico 2 W

while True:
    led.toggle()
    time.sleep(0.25)
```

Листинг 2.8: CircuitPython: встроенный светодиод

```
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.LED)   # одно имя для всех плат семейства
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = not led.value
    time.sleep(0.25)
```

Плата	MicroPython	CircuitPython
Pico, Pico 2	`Pin("LED")` или `Pin(25)`	`board.LED`
Pico W, Pico 2 W	только `Pin("LED")`	`board.LED`

Таблица 2.2: Встроенный светодиод на разных платах

2.11 Самостоятельный опыт

Подключите светодиод к GP13 через резистор и запустите Blink. Затем проверьте встроенный светодиод. Это два отдельных опыта: внешний показывает обычный GPIO, встроенный проверяет, что вы узнаёте свою плату.

Если не работает

- **Светодиод не горит** — проверьте полярность: длинная ножка (анод) — к резистору и GPIO, короткая — к GND.
- **Светодиод горит постоянно** — номер вывода в программе не совпадает с проводом; сверьте номер GP с распиновкой.
- **Ошибка `ValueError: invalid pin`** — такого номера GP нет на плате (бывают только GP0–GP22 и GP26–GP28).
- **Плата не видна в Thonny** — кабель только для зарядки или не тот порт; попробуйте другой кабель и разъём.
- **Мигает не то, что написано в программе** — на плате остался старый `main.py`/`code.py`. Остановите его Ctrl+C и сохраните новый файл.

Итоги урока

- `Pin(n, Pin.OUT)` и `value(0/1)` в MicroPython; `DigitalInOut` и `value = False/True` в CircuitPython.
- Светодиод всегда подключается через резистор; ток вывода ограничен.
- Для отсчёта времени без остановки программы используем `ticks_ms()` / `monotonic()`, а не `sleep()`.
- Встроенный светодиод на любой плате — `Pin("LED")` и `board.LED`.

Проверь себя

1. Зачем светодиоду резистор и как рассчитать его номинал?
2. Почему в программе указывают номер GP, а не номер контакта платы?
3. Почему `sleep()` мешает делать несколько дел одновременно?
4. Почему в **MicroPython** время вычитают через `ticks_diff()`, а не обычным минусом?
5. Как включить встроенный светодиод на Pico W?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите схему из урока и запустите Blink. Измените паузу так, чтобы светодиод мигал в два раза быстрее.
2. ★ Измените программу так, чтобы светодиод горел 100 мс, а гас 900 мс. Как называется отношение этих времён?
3. ★★ Замените `sleep()` на проверку времени по образцу урока и убедитесь, что светодиод мигает так же.
4. ★★ Измените бегущий огонь так, чтобы огонь бежал туда и обратно, не повторяя крайние светодиоды дважды.
5. ★★★ Соберите «светофор» из трёх светодиодов с правильной последовательностью и длительностями, без `sleep()`.
6. ★★★★ Сделайте мигание «сердцебиением»: две короткие вспышки и пауза, а второй светодиод одновременно мигает с частотой 3 Hz — оба без `sleep()`.

Урок 3. Цифровой вход: кнопка

Цель	научиться читать состояние кнопки, понимать подтяжку (pull-up и pull-down) и логические уровни, бороться с дребезгом контактов и выбирать между опросом и прерыванием
Опирается на	урок 2: вывод как выход, светодиод и резистор, неблокирующий отсчёт времени
Результат	кнопка на **GP12** зажигает и гасит светодиод на **GP13**; три кнопки управляют светодиодом; программа отличает короткое нажатие от длинного
Понадобится	макетная плата, 1–3 тактовые кнопки, светодиод, резистор $220\,\Omega$, провода

3.1 Идея урока коротко

В уроке 2 вывод GPIO *говорил*: выдавал 0 или 1. Теперь он будет *слушать*. Вход измеряет напряжение на выводе и сообщает программе: «ниже порога — это 0, выше порога — это 1». Главная трудность в том, что вход ничего не должен «додумывать»: если к нему *ничего* не подключено, он считывает случайные наводки. Поэтому кнопку всегда подключают вместе с **подтягивающим резистором**, который задаёт уровень по умолчанию. Вторая трудность — **дребезг**: механические контакты кнопки замыкаются не мгновенно, а «стучат» несколько миллисекунд, и микроконтроллер, быстрый как молния, успевает насчитать десяток нажатий вместо одного.

3.2 Что читает вход и почему нельзя оставлять его «в воздухе»

Вход GPIO устроен как очень чувствительный вольтметр с огромным сопротивлением (миллионы омов). Ему почти не нужен ток, поэтому достаточно крошечной наводки — от руки, от провода сети, от соседней дорожки, — чтобы напряжение на выводе изменилось. Вывод, ни к чему не подключённый, называют **плавающим** (floating). Его значение непредсказуемо: сегодня 0, завтра 1, а если поднести руку — то и вовсе будет «мигать».

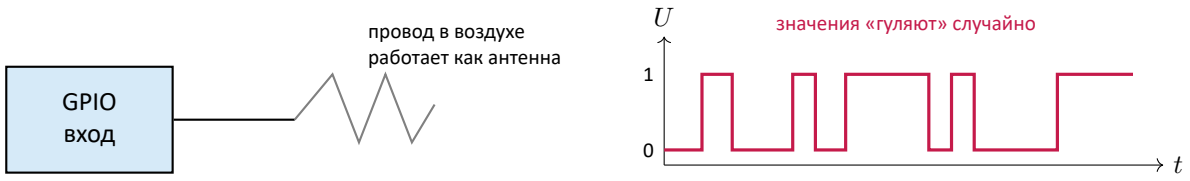


Рис. 3.1: Плавающий вход: напряжение на нём определяется наводками, а не вашей схемой

Это интересно

Если поднести палец к плавающему входу, а в программе печатать `value()`, можно увидеть, как по экрану побегут нули и единицы: вы своим телом наводите на провод напряжение сети 50 Гц из проводки в стенах. Такой «детектор прикосновений» работает без всяких датчиков.

3.3 Логические уровни: где проходит граница между 0 и 1

Цифровой вход решает, что считать нулём, а что единицей, сравнивая напряжение с порогами. Для RP2040 при питании выводов $3{,}3\,\text{V}$ по документации:

- напряжение ниже примерно $0{,}8\,\text{V}$ считается **логическим нулём**;
- выше примерно $2{,}0\,\text{V}$ — **логической единицей**;

- между ними — «серая зона»: результат не гарантирован.

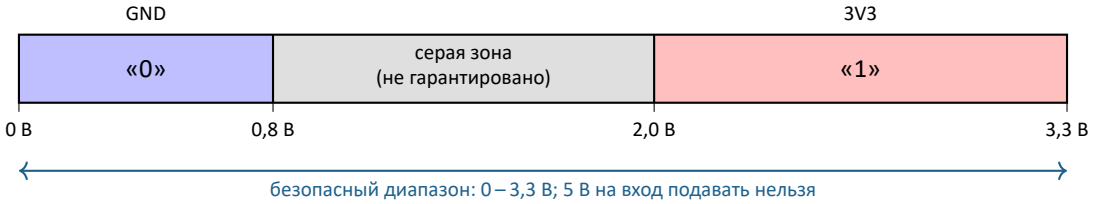


Рис. 3.2: Пороги цифрового входа RP2040 при питании 3,3 В

Чтобы дрожание сигнала вблизи порога не вызывало лишних переключений, входы RP2040 содержат **триггер Шмитта**: порог для переключения вверх чуть выше порога для переключения вниз. Эта небольшая разница называется **гистерезисом**. Она делает вход устойчивым к лёгким шумам, но не спасает от настоящего дребезга кнопки, о котором речь пойдёт ниже.

Осторожно

Входы Pico **не терпят 5 В**. Подавать на вывод напряжение выше $3{,}3$ V (например, сигнал от 5-вольтового датчика) можно только через делитель напряжения или специальный преобразователь уровня.

3.4 Подтяжка: pull-up и pull-down

Решение проблемы плавающего входа — заставить его «по умолчанию» быть нулём или единицей. Для этого вход соединяют с землёй или с питанием через резистор:

- **pull-down** (подтяжка вниз) — резистор к GND: пока кнопка не нажата, на входе **0**, нажатие подаёт на вход 3,3 В, и читается **1**;
- **pull-up** (подтяжка вверх) — резистор к 3,3 В: пока кнопка не нажата, на входе **1**, нажатие замыкает вход на землю, и читается **0**.



а) pull-up **б)** pull-down

Рис. 3.3: Две схемы подтяжки. Резистор R может быть внешним ($10\,\text{k}\Omega$) или встроенным в микросхему

В RP2040 у каждого вывода есть **встроенные** резисторы подтяжки, которые включаются программой. Их сопротивление около 50 – $80\,\text{k}\Omega$. Благодаря им внешний резистор не нужен: кнопку можно включить прямо между выводом и землёй. Почему значение такое большое? Посчитаем ток через подтяжку, когда кнопка нажата и вход замкнут на землю:

$$I = \frac{U}{R} = \frac{3{,}3\ \text{В}}{50\,000\ \Omega} \approx 0{,}066\ \text{мА}.$$

Это очень мало: батарейка почти не разряжается. Если взять резистор $1\,\mathrm{k}\Omega$, ток вырос бы до 3,3 мА — зачем тратить энергию впустую? А слишком большой резистор (мегаомы) делает вход чувствительным к наводкам. Золотая середина — $10 – 100\,\mathrm{k}\Omega$.

	pull-up	pull-down
Кнопка отпущена	1	0
Кнопка нажата	0	1
Кнопка между выводом и	землёй (GND)	питанием (3V3)
MicroPython	`Pin.IN, Pin.PULL_UP`	`Pin.IN, Pin.PULL_DOWN`
CircuitPython	`pull = Pull.UP`	`pull = Pull.DOWN`
Когда выбирают	чаще всего: удобно, кнопка на землю	когда нужна «нормальная» логика: нажато = 1

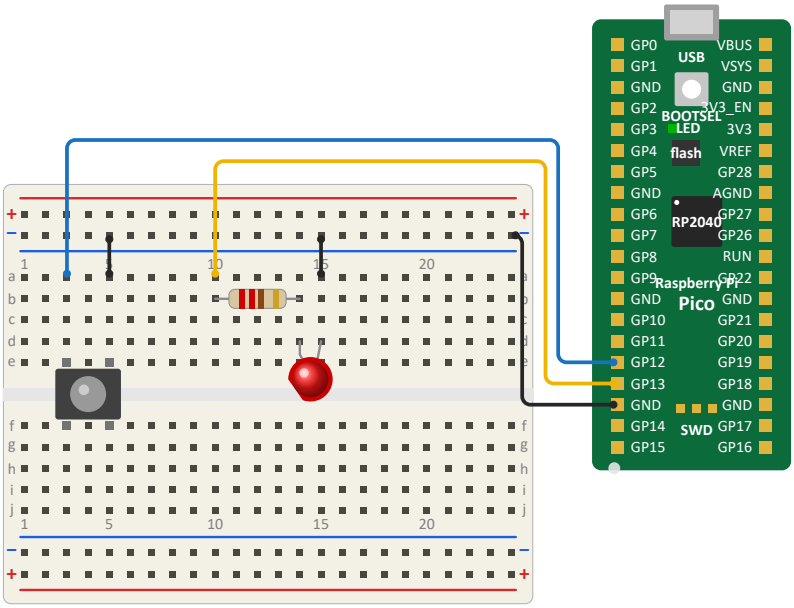
Таблица 3.1: Сравнение подтяжек

Совет

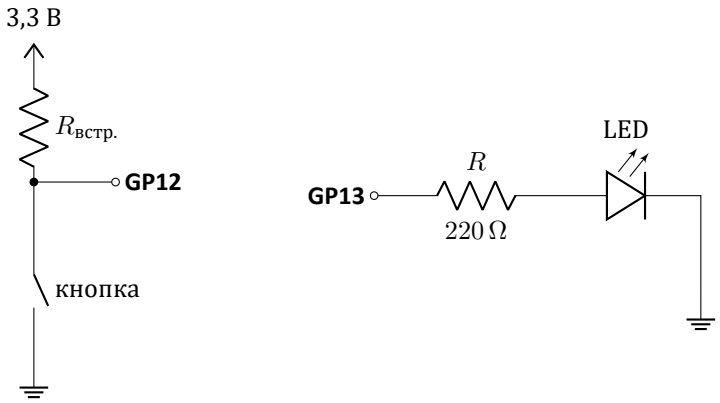
В этом курсе по умолчанию используем **pull-up**: кнопка соединяет вход с землёй. Это проще монтировать, а ошибка «забыл подтяжку» сразу видна. «Нажато = 0» поначалу кажется странным, но к инверсной логике быстро привыкаешь: см. функцию `is_pressed()` ниже.

3.5 Подключаем кнопку

Тактовая кнопка имеет четыре вывода, попарно соединённых внутри. На макетной плате она ставится поперёк центральной канавки; нажатие соединяет её левую и правую пару контактов. Подключим вывод **GP12** к одной стороне кнопки, GND — к другой, а на **GP13** повесим светодиод с резистором из прошлого урока.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 3.4: Кнопка на GP12 (внутренний pull-up) и светодиод на GP13

Обратите внимание на провода: синий идёт от **GP12** к левой стороне кнопки, а правая сторона кнопки через чёрный провод соединена с шиной GND. На схеме это резистор подтяжки $R_{встр.}$ внутри чипа и кнопка к земле.

Листинг 3.1: MicroPython: чтение кнопки и вывод состояния

```
from machine import Pin
import time

# GP12 — вход с внутренней подтяжкой к 3,3 В;
# кнопка при нажатии замыкает вход на землю
button = Pin(12, Pin.IN, Pin.PULL_UP)

while True:
    print(button.value())    # 1 — кнопка отпущена, 0 — нажата
    time.sleep(0.2)
```

Листинг 3.2: CircuitPython: то же самое

```
import time
import board
import digitalio

button = digitalio.DigitalInOut(board.GP12)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP      # внутренняя подтяжка к 3,3 В

while True:
    print(button.value)              # True — отпущена, False — нажата
    time.sleep(0.2)
```

Запустите программу и понажимайте кнопку: при отпущенной кнопке печатаются единицы, при нажатой — нули. В MicroPython `value()` возвращает 0 или 1, в CircuitPython `value` — логическое `True` или `False`.

3.6 Учимся программировать: функция, которая прячет «странную» логику

Нажатие читается как 0 — это неудобно держать в голове. Хороший приём: спрятать особенность схемы в **функцию** с понятным именем. Остальной код тогда читается почти как обычная речь: «если кнопка нажата...». Если завтра вы замените pull-up на pull-down, поправить придётся одну строку внутри функции, а не десять в программе.

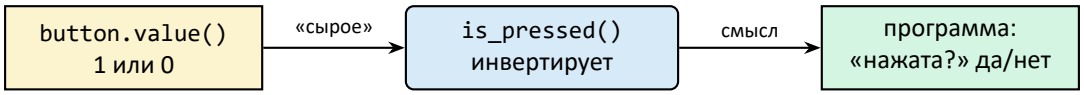


Рис. 3.5: Функция превращает «электрическое» значение вывода в понятное программе

Листинг 3.3: MicroPython: светодиод горит, пока кнопка нажата

```
from machine import Pin

led = Pin(13, Pin.OUT)
button = Pin(12, Pin.IN, Pin.PULL_UP)

def is_pressed():
    """Кнопка нажата? Из-за подтяжки к 3,3 В нажатие — это НОЛЬ."""
    return button.value() == 0

while True:
    led.value(is_pressed())      # True превращается в 1, False — в 0
```

Листинг 3.4: CircuitPython: то же

```python
import board
import digitalio

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

button = digitalio.DigitalInOut(board.GP12)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP

def is_pressed():
    """Кнопка нажата? Из-за подтяжки к 3,3 В нажатие — это False."""
    return not button.value

while True:
    led.value = is_pressed()
```

3.7 Дребезг контактов

Внутри кнопки лежит упругая металлическая пластинка. При нажатии она не «прилипает» к контакту, а несколько раз отскакивает, как мячик об пол. За время около $1 – 10$ ms (иногда и дольше у изношенных кнопок) напряжение на входе несколько раз переключается между 0 и 1, прежде чем успокоится. Это явление называется **дребезгом контактов**. Для человека оно незаметно, а для микроконтроллера, который проверяет вход миллионы раз в секунду, это серия настоящих нажатий.

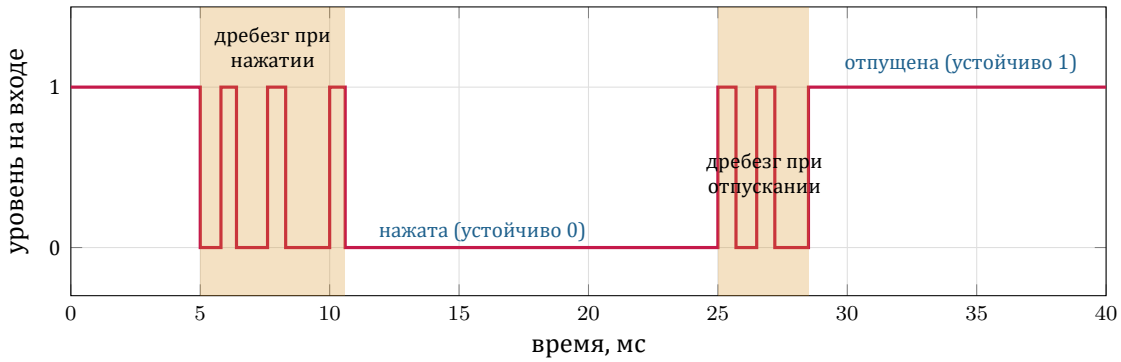


Рис. 3.6: Осциллограмма контакта кнопки с подтяжкой к 3,3 В: при нажатии сигнал падает не один раз, а несколько

Проверим на опыте. Программа ниже считает *падающие фронты* — моменты, когда значение меняется с 1 на 0. Наивно кажется, что одно нажатие даст один фронт.

Листинг 3.5: MicroPython: наивный подсчёт нажатий (покажет дребезг)

```python
from machine import Pin

button = Pin(12, Pin.IN, Pin.PULL_UP)

count = 0
prev = button.value()            # предыдущее чтение

while True:
```

```
    now = button.value()
    if prev == 1 and now == 0:   # падающий фронт: кнопку только что нажали
        count += 1
        print("нажатий:", count)
    prev = now                   # запоминаем для следующего шага
```

Листинг 3.6: CircuitPython: то же

```
import board
import digitalio

button = digitalio.DigitalInOut(board.GP12)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP

count = 0
prev = button.value

while True:
    now = button.value
    if prev and not now:         # падающий фронт: кнопку только что нажали
        count += 1
        print("нажатий:", count)
    prev = now
```

Нажмите кнопку 10 раз. На качественной кнопке счётчик, скорее всего, покажет 10–15, а иногда и больше: ровно столько раз сигнал «упал», пока контакт стучал. У кнопок бывает по-разному: одни почти не дребезжат, другие — как отбойный молоток. Рассчитывать на удачу нельзя.

Три способа победить дребезг

Способ	Идея	Плюсы и минусы
Программный: пауза	после первого фронта игнорируем вход 20–50 мс	просто; но `sleep()` блокирует программу
Программный: «стабильность»	принимаем новое значение, только если оно не менялось ≥ 30 мс	не блокирует, надёжно; чуть сложнее
Аппаратный	RC-фильтр и триггер Шмитта сглаживают дребезг до входа	не нагружает процессор; нужны детали

Таблица 3.2: Способы подавления дребезга

3.8 Программный антидребезг без блокировки

Идея: пока сигнал дёргается, мы не доверяем ему. Новое значение принимаем только после того, как оно *продержалось без изменений* заданное время (обычно 20–50 мс). Для отсчёта времени используем приём из урока 2: сравнение `ticks_ms()` с моментом последнего изменения.

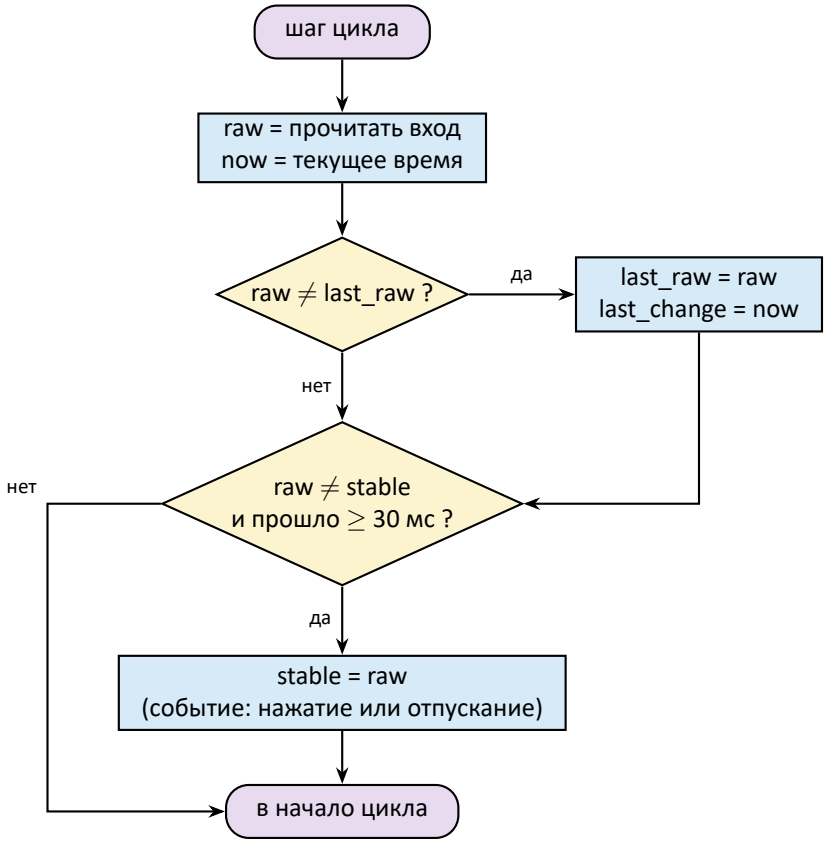


Рис. 3.7: Блок-схема антидребезга по «времени стабильности»

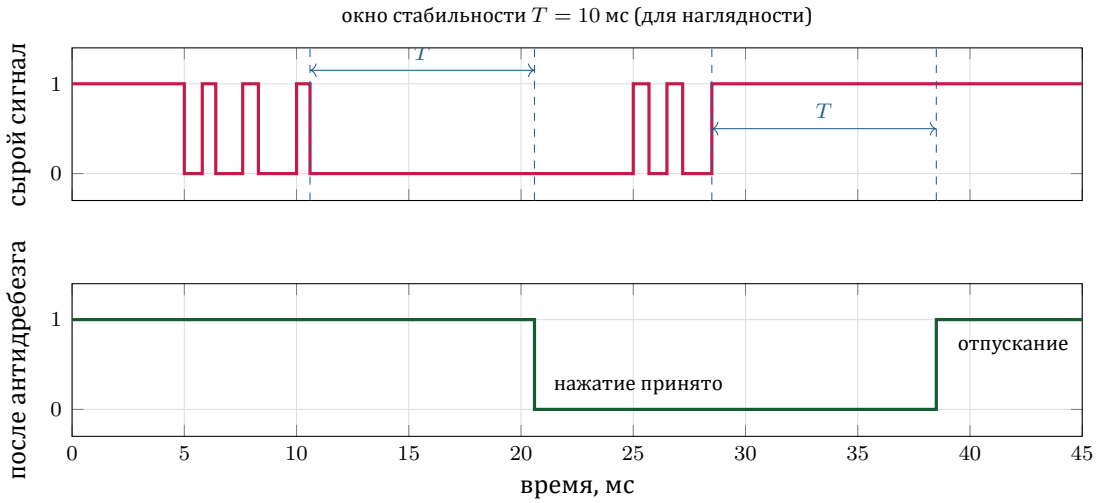


Рис. 3.8: Сырой сигнал и результат антидребезга: короткие «дёрганья» отброшены, изменение принято через T после последнего скачка

Листинг 3.7: MicroPython: счётчик нажатий с антидребезгом

```micropython
from machine import Pin
import time

DEBOUNCE_MS = 30                # сколько миллисекунд сигнал должен «молчать»

button = Pin(12, Pin.IN, Pin.PULL_UP)

stable = button.value()         # подтверждённое состояние кнопки
last_raw = stable               # последнее «сырое» чтение
last_change = time.ticks_ms()   # когда «сырое» значение менялось в последний раз
count = 0

while True:
    raw = button.value()
    now = time.ticks_ms()
    if raw != last_raw:                 # сигнал дёрнулся — отсчёт начинаем заново
        last_raw = raw
        last_change = now
    if raw != stable and time.ticks_diff(now, last_change) >= DEBOUNCE_MS:
        stable = raw                    # значение продержалось 30 мс — принимаем
        if stable == 0:                 # и это нажатие, а не отпускание
            count += 1
            print("нажатий:", count)
```

Листинг 3.8: CircuitPython: то же

```circuitpython
import time
import board
import digitalio

DEBOUNCE = 0.030                # секунды, в течение которых сигнал «молчит»

button = digitalio.DigitalInOut(board.GP12)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP

stable = button.value           # подтверждённое состояние
last_raw = stable               # последнее «сырое» чтение
last_change = time.monotonic()  # когда сигнал менялся в последний раз
count = 0

while True:
    raw = button.value
    now = time.monotonic()
    if raw != last_raw:                   # сигнал дёрнулся — отсчёт заново
        last_raw = raw
        last_change = now
    if raw != stable and now - last_change >= DEBOUNCE:
        stable = raw                      # продержался 30 мс — принимаем
        if not stable:                    # False = кнопка нажата
            count += 1
            print("нажатий:", count)
```

Теперь одно нажатие — один счёт. Разберём главную идею: переменная `stable` хранит *подтверждённое* состояние, а `last_raw` и `last_change` отслеживают, как давно сырой сигнал «молчит». Пока `raw` прыгает, `last_change` обновляется, и условие «прошло 30 мс» не выполняется.

Это интересно

В **CircuitPython** для простоты использована `time.monotonic()` — число секунд с плавающей точкой. Оно со временем теряет точность (через несколько часов шаг становится заметно больше миллисекунды), поэтому для долгих программ используют `supervisor.ticks_ms()` или библиотеку `adafruit_ticks`. Подробнее — в уроке 4.

3.9 Аппаратный антидребезг

Дребезг можно убрать и без программы. Простейший способ — RC-фильтр: резистор и конденсатор сглаживают быстрые скачки, как амортизатор сглаживает толчки на неровной дороге. Затем сигнал проходит через вход с триггером Шмитта, который и так есть в RP2040.

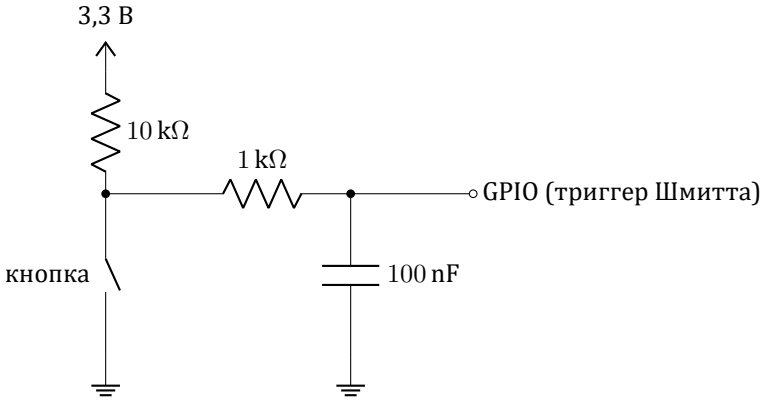


Рис. 3.9: Аппаратный антидребезг: постоянная времени $\tau = RC = 1\,\text{k}\Omega \cdot 100\,\text{nF} = 0{,}1\,\text{ms}$ сглаживает быстрые скачки

Постоянная времени цепи $\tau = RC$ показывает, за какое время конденсатор «успевает» заряжаться. Для $R = 1\,\text{k}\Omega$ и $C = 100\,\text{nF}$ получаем $\tau = 1000 \cdot 10^{-7} = 10^{-4}$ с = 0,1 мс. Для настоящего дребезга в 5–10 мс нужна постоянная времени порядка миллисекунд, например $10\,\text{k}\Omega$ и $1\,\mu\text{F}$. В учебных проектах чаще всего ограничиваются программным способом: он бесплатный.

3.10 Опрос или прерывание

До сих пор программа сама, снова и снова, спрашивала кнопку: такой приём называется **опросом** (polling). Он прост и надёжен, но программа должна заходить в опрос достаточно часто. Если она надолго уйдёт «рисовать на экране», короткое нажатие можно пропустить.
Другой подход — **прерывание** (interrupt). Контроллер *сам* следит за выводом аппаратно, и при изменении уровня приостанавливает основную программу, выполняет короткую функцию-обработчик и возвращается обратно. Аналогия: можно каждые пять минут подходить к двери и смотреть, не пришёл ли гость (опрос), а можно поставить звонок и заниматься своими делами (прерывание).

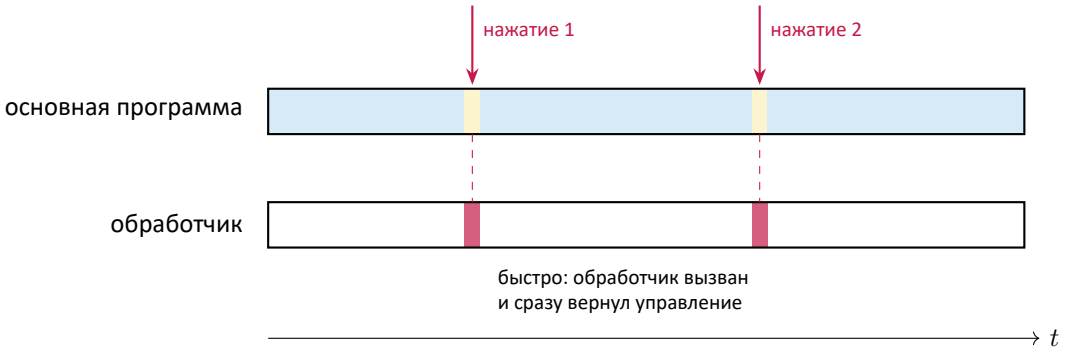


Рис. 3.10: Прерывание: основная программа на мгновение приостанавливается, пока работает обработчик

Листинг 3.9: MicroPython: прерывание по падающему фронту с защитой от дребезга

```
from machine import Pin
import time

DEBOUNCE_MS = 50

button = Pin(12, Pin.IN, Pin.PULL_UP)

presses = 0          # счётчик нажатий (его увеличивает обработчик)
last_irq = 0         # момент последнего принятого нажатия

def on_press(pin):
    """Обработчик прерывания: короткий, без print и sleep."""
    global presses, last_irq
    now = time.ticks_ms()
    if time.ticks_diff(now, last_irq) > DEBOUNCE_MS:   # отсекаем дребезг
        last_irq = now
        presses += 1

button.irq(trigger=Pin.IRQ_FALLING, handler=on_press)

shown = 0
while True:
    # основная программа занята «своими делами» и лишь следит за счётчиком
    if presses != shown:
        shown = presses
        print("нажатий:", shown)
    time.sleep_ms(10)
```

В MicroPython метод `irq()` принимает `trigger` (`Pin.IRQ_FALLING`, `Pin.IRQ_RISING` или их сумму через `|`) и `handler` — функцию с одним аргументом (объект `Pin`). Нажатие считается в обработчике, а печать и остальные медленные действия выполняет основной цикл.

Осторожно

Обработчик прерывания должен быть **очень коротким**:

- не вызывайте в нём `sleep()`, не ждите ничего, не печатайте большие тексты;
- не создавайте новые объекты и большие списки (в «жёстких» прерываниях `hard=True` это запрещено вовсе);
- общие переменные объявляйте через `global`; если нужен сложный расчёт — поставьте

флаг и сделайте расчёт в основном цикле;

- дребезг *всё равно* нужно подавлять: каждый скачок — это отдельное прерывание.

Как это в CircuitPython

В CircuitPython (на 9.x) нет метода `irq()`: готового пользовательского обработчика прерывания у вывода нет. Вместо него есть два удобных модуля:

- `keypad` — фоновый опрос кнопок: сам убирает дребезг (интервал сканирования по умолчанию 20 ms) и складывает события «нажали»/«отпустили» с метками времени в очередь;
- `countio` — аппаратный счётчик фронтов, который считает без участия программы.

Листинг 3.10: CircuitPython: `keypad` — очередь событий кнопки

```
import board
import keypad

# keypad сам опрашивает вывод в фоне и убирает дребезг (интервал 20 мс);
# нажатия и отпускания складываются в очередь событий
keys = keypad.Keys((board.GP12,), value_when_pressed=False, pull=True)

count = 0
while True:
    event = keys.events.get()        # одно событие или None
    if event and event.pressed:      # event.released — отпускание
        count += 1
        print("нажатий:", count)
    # здесь может быть любая другая работа — очередь не потеряет событий
```

Листинг 3.11: CircuitPython: `countio` — аппаратный счётчик фронтов

```
import time
import board
import countio
import digitalio

# аппаратный счётчик фронтов: считает без участия программы
counter = countio.Counter(board.GP12, edge=countio.Edge.FALL, pull=digitalio.Pull.UP)

while True:
    print("падающих фронтов:", counter.count)
    if counter.count > 100:
        counter.reset()              # обнуляем счётчик
    time.sleep(1)
```

Очередь событий `keypad` работает примерно как «почтовый ящик»: нажатие кладёт в него письмо, а программа забирает письма, когда ей удобно. Событие не теряется, даже если программа была занята. (Размер очереди ограничен: по умолчанию 64 события, при переполнении самые новые теряются.) Счётчик `countio` годится, когда нужно только *сколько*, а не *когда*: например, считать обороты колеса.

3.11 Переключатель-триггер: нажал — включил, нажал — выключил

Обычная кнопка «живёт», пока её держат. Часто нужно другое поведение — как у настольной лампы: одно нажатие включает, следующее выключает. Это **триггер** (toggle): программа долж-

на *помнить* состояние между нажатиями. Состояние описывается булевой переменной, а смена состояния — операцией not.



Рис. 3.11: Диаграмма состояний переключателя: каждое подтверждённое нажатие меняет состояние на противоположное

Важно реагировать на *событие* нажатия (переход «отпущена → нажата»), а не на состояние «кнопка нажата». Иначе пока вы держите кнопку, состояние будет меняться на каждом шаге цикла.

Листинг 3.12: MicroPython: триггер на светодиоде

```
from machine import Pin
import time

led = Pin(13, Pin.OUT)
button = Pin(12, Pin.IN, Pin.PULL_UP)

DEBOUNCE_MS = 30
stable = 1                      # подтверждённое состояние (1 = отпущена)
last_raw = 1
last_change = time.ticks_ms()

def was_pressed():
    """Вернуть True ровно один раз — в момент подтверждённого нажатия."""
    global stable, last_raw, last_change
    raw = button.value()
    now = time.ticks_ms()
    if raw != last_raw:
        last_raw = raw
        last_change = now
    if raw != stable and time.ticks_diff(now, last_change) >= DEBOUNCE_MS:
        stable = raw
        return stable == 0
    return False

state = False                   # включён ли светодиод
while True:
    if was_pressed():
        state = not state       # триггер: каждое нажатие меняет состояние
        led.value(state)
```

Листинг 3.13: CircuitPython: триггер с библиотекой `adafruit_debouncer`

```
import board
import digitalio
from adafruit_debouncer import Debouncer   # библиотека ставится через circup

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

pin = digitalio.DigitalInOut(board.GP12)
pin.direction = digitalio.Direction.INPUT
pin.pull = digitalio.Pull.UP

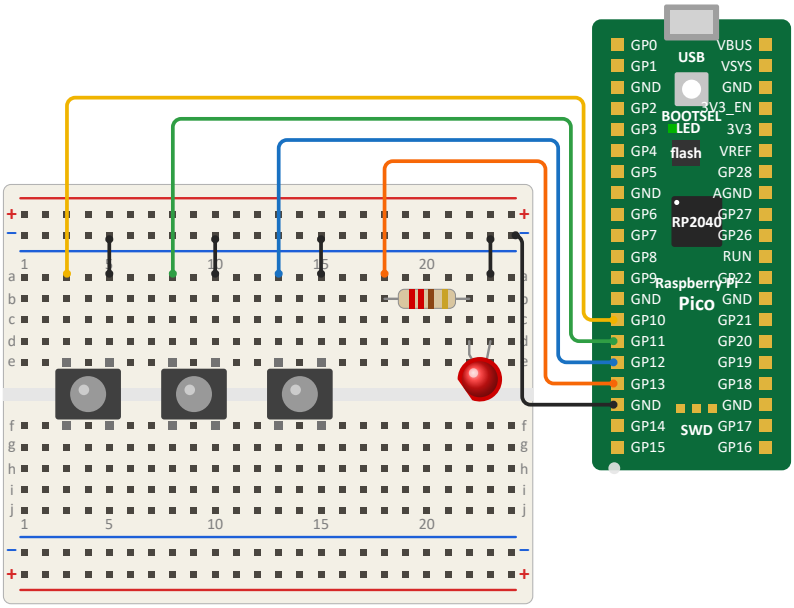
button = Debouncer(pin)          # обёртка с антидребезгом

while True:
    button.update()              # обязательно вызывать в каждом шаге цикла
    if button.fell:              # True ровно один раз: кнопка только что нажата
        led.value = not led.value
```

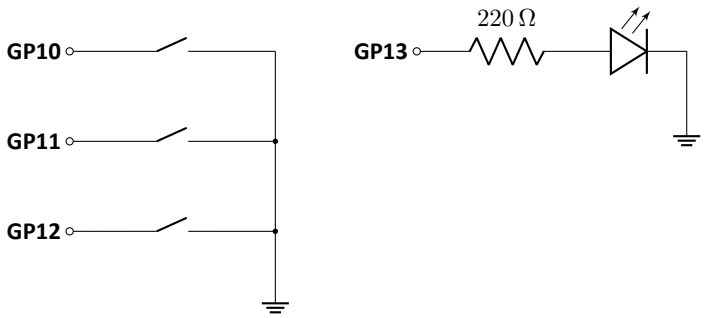
Здесь в MicroPython применена функция `was_pressed()`, которая вернёт `True` *ровно один раз* за нажатие. В CircuitPython то же делает свойство `fell` объекта `Debouncer`. Библиотеку нужно установить на плату (например, командой `circup install adafruit_debouncer`).

3.12 Несколько кнопок на одной плате

Добавим ещё две кнопки: одна включает светодиод, другая выключает, третья переключает. Каждой кнопке нужны отдельный вывод и свои данные для антидребезга (`stable`, `last_raw`, `last_change`). Копировать код три раза — неудобно: пора познакомиться с классами.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 3.12: Три кнопки на GP10, GP11, GP12 и светодиод на GP13

Проследите по рисунку: три провода (жёлтый, зелёный, синий) идут к трём кнопкам, оранжевый — к резистору светодиода. У каждой кнопки есть свой чёрный провод к шине GND. Провода не пересекаются: чем выше вывод на плате, тем ближе к плате Pico проходит его «полоса» и тем левее он спускается к детали.

Класс: образец кнопки

Класс — это «чертёж» объекта. По одному чертежу можно сделать сколько угодно одинаковых объектов, у каждого — свои данные. Класс `Button` объединяет вывод, настройки антидребезга и метод `pressed()`. Создать три кнопки — три строки.

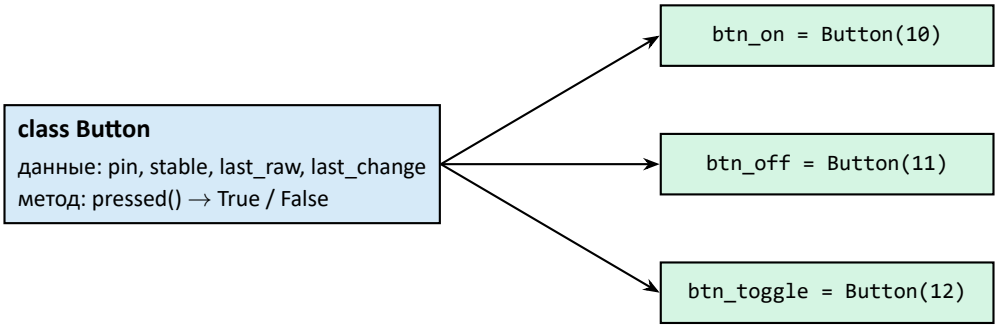


Рис. 3.13: Один класс — три независимых объекта-кнопки

Листинг 3.14: MicroPython: класс `Button` и три кнопки

```micropython
from machine import Pin
import time

class Button:
    """Кнопка с антидребезгом. pressed() даёт True один раз за нажатие."""
    def __init__(self, gp, debounce_ms=30):
        self.pin = Pin(gp, Pin.IN, Pin.PULL_UP)
        self.debounce_ms = debounce_ms
        self.stable = self.pin.value()
        self.last_raw = self.stable
        self.last_change = time.ticks_ms()

    def pressed(self):
        raw = self.pin.value()
        now = time.ticks_ms()
        if raw != self.last_raw:
            self.last_raw = raw
            self.last_change = now
        if raw != self.stable and time.ticks_diff(now, self.last_change) >= self.debounce_ms:
            self.stable = raw
            return raw == 0
        return False

led = Pin(13, Pin.OUT)
btn_on = Button(10)              # включить
btn_off = Button(11)             # выключить
btn_toggle = Button(12)          # переключить

while True:
    if btn_on.pressed():
        led.value(1)
    if btn_off.pressed():
        led.value(0)
    if btn_toggle.pressed():
        led.toggle()
```

Листинг 3.15: CircuitPython: три кнопки через `keypad.Keys`

```circuitpython
import board
import digitalio
import keypad

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

# один объект Keys следит сразу за тремя кнопками
keys = keypad.Keys((board.GP10, board.GP11, board.GP12),
                   value_when_pressed=False, pull=True)

while True:
    event = keys.events.get()
    if event and event.pressed:
        if event.key_number == 0:        # GP10: включить
            led.value = True
        elif event.key_number == 1:      # GP11: выключить
```

```
            led.value = False
        elif event.key_number == 2:      # GP12: переключить
            led.value = not led.value
```

В **CircuitPython** нужды в собственном классе нет: один объект `keypad.Keys` следит за несколькими выводами и сообщает номер кнопки в `event.key_number`. Выводы можно расположить как угодно, главное — перечислить их кортежем в нужном порядке.

3.13 Долгое нажатие

Одной кнопкой можно управлять несколькими функциями, если различать *короткое* и *долгое* нажатие, как на кнопке питания телефона. Для этого при нажатии запоминаем время и следим, сколько кнопка остаётся нажатой. Если прошло больше порога (скажем, 800 мс) — это долгое нажатие; если кнопку отпустили раньше — короткое.

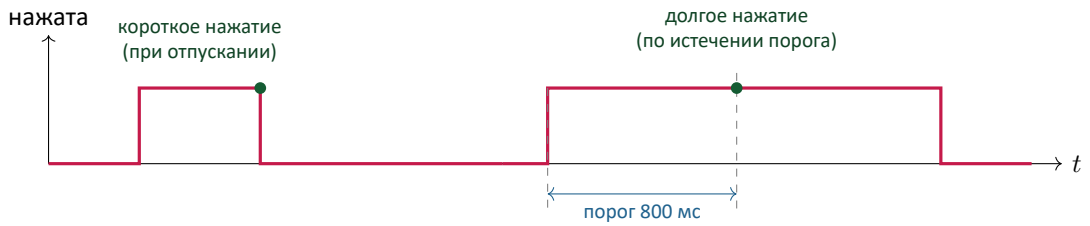


Рис. 3.14: Короткое действие выполняется при отпускании, длинное — как только пройден порог времени

Листинг 3.16: **MicroPython**: короткое нажатие переключает, длинное гасит

```
from machine import Pin
import time

led = Pin(13, Pin.OUT)
button = Pin(12, Pin.IN, Pin.PULL_UP)

DEBOUNCE_MS = 30
LONG_MS = 800                    # дольше этого нажатие считается «длинным»

stable = 1
last_raw = 1
last_change = time.ticks_ms()
t_press = 0                      # когда кнопку нажали
long_sent = False                # длинное нажатие уже обработано?

def poll():
    """Вернуть "down" (нажали), "up" (отпустили) или None."""
    global stable, last_raw, last_change
    raw = button.value()
    now = time.ticks_ms()
    if raw != last_raw:
        last_raw = raw
        last_change = now
    if raw != stable and time.ticks_diff(now, last_change) >= DEBOUNCE_MS:
        stable = raw
        return "down" if stable == 0 else "up"
    return None

while True:
```

```
    event = poll()
    now = time.ticks_ms()
    if event == "down":
        t_press = now
        long_sent = False
    elif event == "up" and not long_sent:
        print("короткое нажатие: переключаем светодиод")
        led.toggle()
    # кнопка всё ещё удерживается и порог времени пройден
    if stable == 0 and not long_sent and time.ticks_diff(now, t_press) >= LONG_MS:
        long_sent = True
        print("длинное нажатие: гасим светодиод")
        led.value(0)
```

Листинг 3.17: CircuitPython: то же через keypad

```
import time
import board
import digitalio
import keypad

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

keys = keypad.Keys((board.GP12,), value_when_pressed=False, pull=True)

LONG = 0.8                      # секунды
down = False                    # кнопка сейчас удерживается?
t_press = 0
long_sent = False

while True:
    event = keys.events.get()
    now = time.monotonic()
    if event:
        if event.pressed:
            down = True
            t_press = now
            long_sent = False
        else:                                   # отпускание
            down = False
            if not long_sent:
                print("короткое нажатие: переключаем светодиод")
                led.value = not led.value
    if down and not long_sent and now - t_press >= LONG:
        long_sent = True
        print("длинное нажатие: гасим светодиод")
        led.value = False
```

Проверьте логику на опыте: при длинном нажатии светодиод гаснет *один раз*, а при отпускании не переключается повторно. Именно для этого служит флаг `long_sent`. Заметьте: в этой программе вообще нет `sleep()` — можно параллельно выполнять и другие дела.

Совет

В библиотеке `adafruit_debouncer` есть класс `Button`, который сам различает короткое, длинное нажатие и подсчитывает серии кликов (`short_count`, `long_press`); это удобно, когда не нужно писать свой код.

3.14 Самостоятельный опыт

Подключите кнопку на **GP12** и светодиод на **GP13**. Выполните по порядку: (1) прочитайте состояние кнопки и убедитесь, что при нажатии выводится 0; (2) запустите наивный счётчик и посмотрите на дребезг; (3) подключите антидребезг и убедитесь, что одно нажатие — один счёт; (4) соберите триггер. Запишите в тетрадь, сколько «лишних» нажатий насчитала ваша кнопка без антидребезга.

Если не работает

- **Всегда читается 0** — вход, скорее всего, замкнут на землю: проверьте провода и положение кнопки на плате. Контакты кнопки, соединённые внутри, нужно ставить так, чтобы нажатие соединяло *разные* стороны, а не одну и ту же пару.
- **Значение «прыгает» без нажатия** — не включена подтяжка (забыли `Pin.PULL_UP` или `pull = Pull.UP`).
- **Нажатие считается несколько раз** — нет антидребезга или слишком маленькое окно; увеличьте до 30–50 мс.
- **Кнопка работает «наоборот»** — не учтена инверсная логика pull-up: нажато = 0.
- **Обработчик `irq` молчит** — проверьте `trigger` и не удерживает ли основной код процессор в долгом `sleep()`.
- **`ImportError: adafruit_debouncer`** — библиотека не установлена на `CIRCUITPY/lib`.

Итоги урока

- Вход нельзя оставлять плавающим: нужна подтяжка pull-up или pull-down (внутренняя $50 – 80\,\text{k}\Omega$ или внешняя $10\,\text{k}\Omega$).
- С pull-up нажатие читается как 0, с pull-down — как 1; логику прячем в функцию `is_pressed()`.
- Пороги входа RP2040: около $0{,}8\,\text{V}$ и $2{,}0\,\text{V}$; на вход нельзя подавать больше $3{,}3\,\text{V}$.
- Дребезг контактов длится миллисекунды; подавляется программой (окно стабильности 20–50 мс) или RC-фильтром.
- Опрос прост; прерывание (`Pin.irq`) быстрее реагирует, но обработчик должен быть коротким. В **CircuitPython** вместо `irq` используют `keypad` и `countio`.
- Триггер хранит состояние в переменной, долгое нажатие измеряется временем между нажатием и отпусканием.

Проверь себя

1. Что такое плавающий вход и чем он опасен?
2. Чем pull-up отличается от pull-down? Что прочитает программа при нажатой кнопке в каждом случае?
3. Что такое дребезг контактов и как он выглядит на осциллограмме?
4. Как работает антидребезг «по времени стабильности»?
5. Почему обработчик прерывания должен быть коротким и что нельзя в нём делать?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите схему с кнопкой и светодиодом: светодиод горит, пока кнопка нажата. Затем измените программу так, чтобы светодиод, наоборот, гас при нажатии.
2. ★ Запустите наивный счётчик и антидребезговый счётчик и сравните результаты на 20 нажатиях. Запишите, сколько раз дребезг «обманул» первую программу.
3. ★★ Замените внутренний pull-up на внешний резистор 10 kΩ (или на pull-down) и измените программу. Что изменится в `is_pressed()`?
4. ★★ Подберите минимальное время антидребезга для вашей кнопки: уменьшайте окно с 50 мс, пока не появятся ложные срабатывания.
5. ★★★ Сделайте «замок»: светодиод загорается на 3 секунды только после правильной комбинации из трёх кнопок (например, 1-3-2), неверный порядок сбрасывает ввод.
6. ★★★★ Сделайте «игру на реакцию»: светодиод загорается через случайное время (`random`), игрок жмёт кнопку, а программа выводит время реакции в миллисекундах. Используйте прерывание, а время измеряйте через `ticks_us()`.

Урок 4. Время и конечные автоматы

Цель	научиться отсчитывать время без `sleep()`, пользоваться таймером, планировать несколько задач «на тиках» и описывать поведение устройства конечным автоматом
Опирается на	урок 2: неблокирующее мигание; урок 3: чтение кнопки и антидребезг
Результат	светофор с пешеходной кнопкой работает по таблице состояний; несколько задач выполняются одновременно в одном цикле; вы умеете измерить время выполнения кода
Понадобится	макетная плата, три светодиода (красный, жёлтый, зелёный), кнопка, три резистора 220 Ω, провода

4.1 Идея урока коротко

Настоящее устройство делает много дел сразу: мигает, слушает кнопку, считает, пишет на экран. Но процессор один, он выполняет команды по очереди. «Одновременность» достигается хитростью: каждое дело выполняется маленькими порциями, а программа постоянно спрашивает часы: «не пора ли?». Чтобы такая программа не превратилась в кашу из `if`, поведение устройства описывают **конечным автоматом**: набором состояний («красный», «зелёный»...) и правил перехода между ними. В этом уроке вы научитесь работать с часами, таймерами, планировщиком и автоматами, а в конце соберёте настоящий светофор с кнопкой для пешеходов.

4.2 Часы микроконтроллера: тики

Внутри чипа работает аппаратный счётчик, который увеличивается равномерно (у RP2040 — один раз в микросекунду). Программа читает его показания и вычитает: так получается интервал времени. В MicroPython для этого служат:

- `time.ticks_ms()` — «тики» в миллисекундах, `time.ticks_us()` — в микросекундах;

- `time.ticks_diff(новое, старое)` — правильная разность двух значений;
- `time.ticks_add(t, ms)` — «прибавить» миллисекунды к моменту времени.

В **CircuitPython**:

- `time.monotonic()` — секунды с плавающей точкой (удобно, но точность падает при долгой работе);
- `time.monotonic_ns()` — целое число наносекунд (точное, но «тяжелее» для расчётов);
- `supervisor.ticks_ms()` — миллисекундные тики, как в **MicroPython**, и библиотека `adafruit_ticks`.

Переполнение счётчика

Счётчик тиков не бесконечный: в **MicroPython** он имеет период 2^{30} миллисекунд (около 12,4 суток), в `supervisor.ticks_ms()` — 2^{29} (около 6,2 суток). Дойдя до максимума, он «закручивается» и начинает сначала, как стрелки часов после 12. Если вычитать значения обычным минусом, в момент перехода получится огромное отрицательное число. Функция `ticks_diff()` считает разность *по модулю периода*, поэтому всегда даёт правильный интервал.

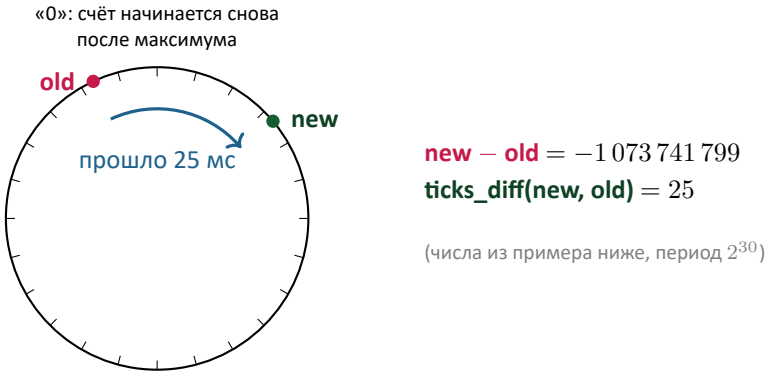


Рис. 4.1: Счётчик тиков — это циферблат: после максимума он снова идёт с нуля, и обычное вычитание ошибается

Листинг 4.1: MicroPython: `ticks_diff` спасает при переполнении

```
import time

# ticks_add(0, -10) — «за 10 мс до переполнения счётчика»: число около 2**30
old = time.ticks_add(0, -10)
# прошло ещё 25 мс: счётчик перескочил через максимум и начал с нуля
new = time.ticks_add(old, 25)

print("old =", old, " new =", new)
print("обычный минус:", new - old)                 # огромное отрицательное число
print("ticks_diff   :", time.ticks_diff(new, old)) # 25 — правильный ответ
```

Листинг 4.2: CircuitPython: то же с собственной функцией `ticks_diff`

```
import supervisor

PERIOD = 2 ** 29                # supervisor.ticks_ms() «закручивается» через 2**29 мс

def ticks_diff(new, old):
    """Разность тиков с учётом переполнения (как time.ticks_diff в MicroPython)."""
    d = (new - old) & (PERIOD - 1)       # вычитаем по модулю PERIOD
    if d >= PERIOD // 2:                 # «слишком большая» разность — на самом деле
    ↪ отрицательная
        d -= PERIOD
    return d

old = PERIOD - 10               # за 10 мс до переполнения
new = (old + 25) % PERIOD       # прошло 25 мс: счётчик начал с нуля

print("old =", old, " new =", new)
print("обычный минус:", new - old)
print("ticks_diff   :", ticks_diff(new, old))     # 25
print("сейчас тиков :", supervisor.ticks_ms())
```

В CircuitPython готовой `ticks_diff()` в ядре нет (она есть в библиотеке `adafruit_ticks`), поэтому в примере она написана вручную: вычитаем по модулю 2^{29} и отбрасываем «слишком большие» результаты как отрицательные. Запомните эту формулу: она пригодится везде, где есть циклический счётчик.

Это интересно

Если оставить Pico включённой без перезагрузки, `ticks_ms()` «перекрутится» примерно через 12 дней. Сколько программ в мире сломалось ровно в этот момент — не сосчитать. Именно поэтому вычитание тиков — всегда только через `ticks_diff()`.

4.3 Планировщик «на тиках»

Расширим приём из урока 2. Каждой задаче заведём переменную «когда запускать в следующий раз» и в бесконечном цикле будем проверять все задачи по очереди. Такая схема называется **кооперативной многозадачностью**: задачи сами уступают друг другу место, возвращаясь из функции как можно быстрее.

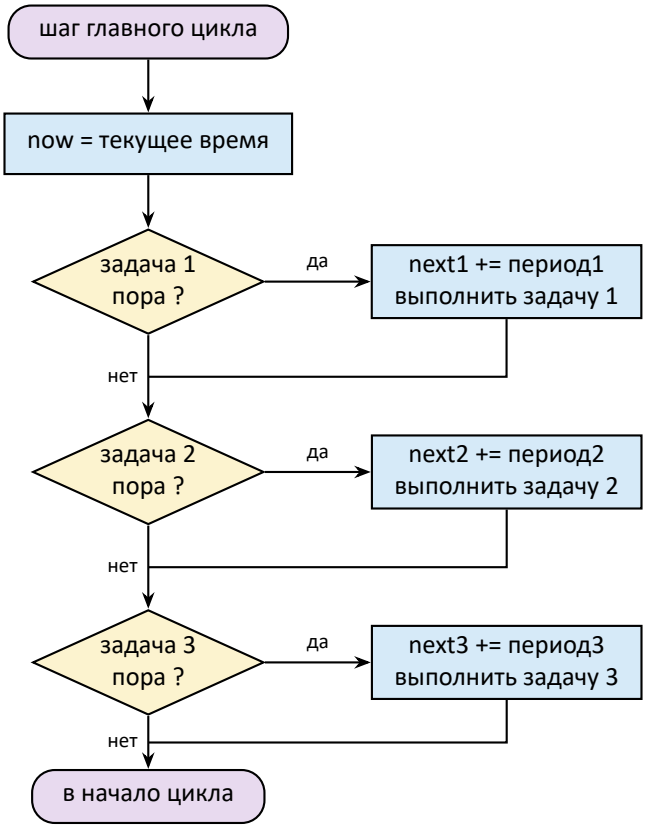


Рис. 4.2: Блок-схема планировщика: каждая задача проверяется по очереди и запускается, только когда пришло её время

Листинг 4.3: MicroPython: три задачи в одном цикле (мигание, кнопка, отчёт)

```
from machine import Pin
import time

led = Pin(13, Pin.OUT)
button = Pin(12, Pin.IN, Pin.PULL_UP)

BLINK_MS = 500                  # задача 1: мигать светодиодом
PRINT_MS = 2000                 # задача 2: печатать отчёт
SCAN_MS = 10                    # задача 3: опрашивать кнопку

t_blink = t_print = t_scan = time.ticks_ms()
presses = 0
prev = 1

while True:
    now = time.ticks_ms()

    if time.ticks_diff(now, t_blink) >= BLINK_MS:
        t_blink = time.ticks_add(t_blink, BLINK_MS)    # строго по сетке времени
        led.toggle()

    if time.ticks_diff(now, t_scan) >= SCAN_MS:
        t_scan = time.ticks_add(t_scan, SCAN_MS)
        cur = button.value()
        if prev == 1 and cur == 0:
            presses += 1
        prev = cur
```

```
28
29      if time.ticks_diff(now, t_print) >= PRINT_MS:
30          t_print = time.ticks_add(t_print, PRINT_MS)
31          print("работаю, нажатий:", presses)
```

Листинг 4.4: CircuitPython: то же

```
1  import board
2  import digitalio
3  import supervisor
4
5  PERIOD = 2 ** 29
6
7  def ticks_diff(new, old):
8      d = (new - old) & (PERIOD - 1)
9      return d - PERIOD if d >= PERIOD // 2 else d
10
11 def ticks_add(t, ms):
12     return (t + ms) % PERIOD
13
14 led = digitalio.DigitalInOut(board.GP13)
15 led.direction = digitalio.Direction.OUTPUT
16 button = digitalio.DigitalInOut(board.GP12)
17 button.direction = digitalio.Direction.INPUT
18 button.pull = digitalio.Pull.UP
19
20 BLINK_MS = 500                  # задача 1: мигать светодиодом
21 PRINT_MS = 2000                 # задача 2: печатать отчёт
22 SCAN_MS = 10                    # задача 3: опрашивать кнопку
23
24 t_blink = t_print = t_scan = supervisor.ticks_ms()
25 presses = 0
26 prev = True
27
28 while True:
29     now = supervisor.ticks_ms()
30
31     if ticks_diff(now, t_blink) >= BLINK_MS:
32         t_blink = ticks_add(t_blink, BLINK_MS)
33         led.value = not led.value
34
35     if ticks_diff(now, t_scan) >= SCAN_MS:
36         t_scan = ticks_add(t_scan, SCAN_MS)
37         cur = button.value
38         if prev and not cur:
39             presses += 1
40         prev = cur
41
42     if ticks_diff(now, t_print) >= PRINT_MS:
43         t_print = ticks_add(t_print, PRINT_MS)
44         print("работаю, нажатий:", presses)
```

Обратите внимание на строку `t_blink = ticks_add(t_blink, BLINK_MS)`. Мы прибавляем период к *прошлому плановому моменту*, а не к текущему времени. Разница принципиальна: если писать `t_blink = now`, то каждый запуск «опаздывает» на микросекунды, ошибка накапливается, и через час мигание заметно отстанет от настоящих часов. При записи «по сетке» опоздания не накапливаются.

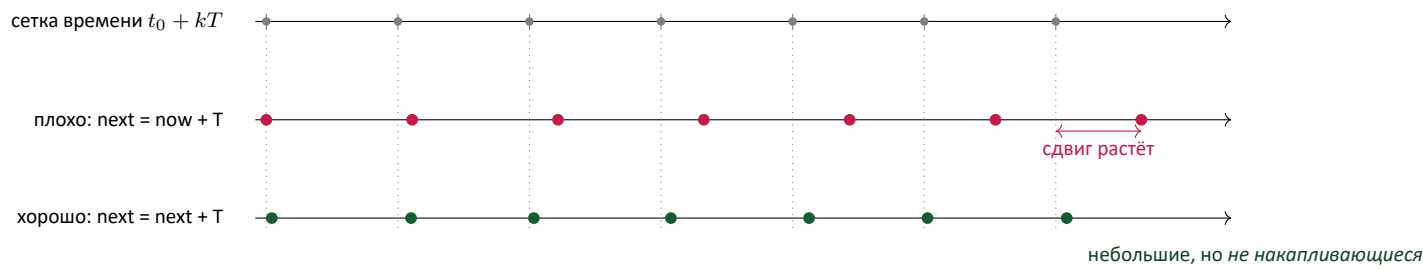


Рис. 4.3: Накопление ошибки при выборе «следующего момента» неправильным способом

Осторожно

У планировщика на тиках есть главное условие: **каждая задача обязана возвращаться быстро**. Если одна из них вызовет `sleep(1)` или будет долго считать, остальные пропустят свои сроки. Всё, что длится дольше нескольких миллисекунд, нужно разбивать на шаги (это и есть автомат) или выносить на второе ядро (урок 13).

4.4 Аппаратные таймеры: `machine.Timer`

Постоянно спрашивать часы неудобно. В MicroPython есть объект `machine.Timer`, который сам вызывает вашу функцию через заданное время. Таймер бывает:

- **периодический** (`Timer.PERIODIC`) — вызывает функцию снова и снова;
- **одноразовый** (`Timer.ONE_SHOT`) — вызывает функцию один раз и останавливается.

Период задают в миллисекундах (`period=500`) или частотой в герцах (`freq=2`). Аргумент `-1` в `Timer(-1)` означает программный таймер; на RP2040 других и нет, а в новых версиях MicroPython `id` можно вообще не указывать.

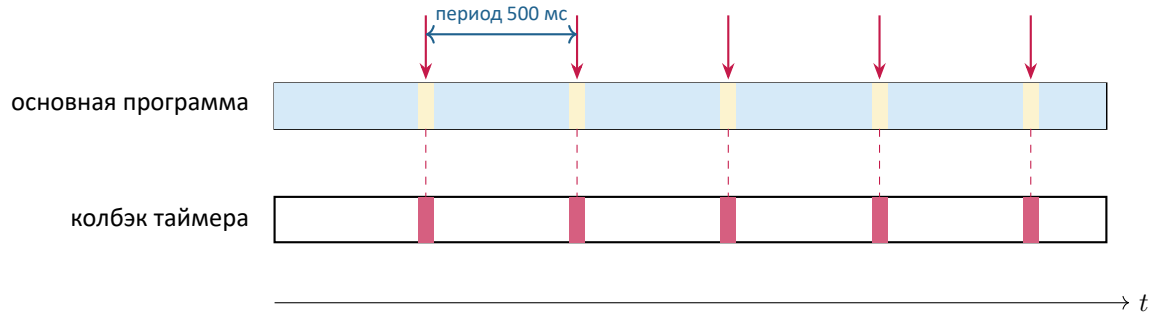


Рис. 4.4: Периодический таймер: каждые 500 мс основная программа ненадолго приостанавливается ради колбэка

Листинг 4.5: MicroPython: два периодических таймера

```
from machine import Pin, Timer
import time

led = Pin(13, Pin.OUT)
report = False                  # флаг: «пора напечатать отчёт»

def blink(t):
    """Колбэк таймера: быстрое действие, без print и sleep."""
    led.toggle()

def every_second(t):
```

```
    """Колбэк только поднимает флаг; печатать будет основной цикл."""
    global report
    report = True

tim_blink = Timer(-1)            # -1 — программный (виртуальный) таймер
tim_blink.init(freq=2, mode=Timer.PERIODIC, callback=blink)   # 2 переключения в секунду

tim_sec = Timer(-1)
tim_sec.init(period=1000, mode=Timer.PERIODIC, callback=every_second)

seconds = 0
while True:
    if report:                   # основной цикл свободен для любых дел
        report = False
        seconds += 1
        print("прошло секунд:", seconds)
    time.sleep_ms(20)
```

Функция, которую вызывает таймер, называется **колбэком** (callback, «перезвони мне»). Она получает один аргумент — сам объект таймера. Колбэк работает «поверх» основной программы, поэтому к нему те же требования, что и к обработчику прерывания (урок 3):

- быть коротким: не печатать длинные тексты, не вызывать `sleep()`, не ждать;
- не создавать лишних объектов (так можно быстро исчерпать память, а в «жёстких» прерываниях это вообще запрещено);
- общие данные менять через `global`; всё тяжёлое оставить основному циклу, передав сигнал флагом (как в примере: колбэк только поднимает `report = True`);
- помнить, что точность вызова ограничена: если основной код надолго «застрял» в одной операции, колбэк подождёт, поэтому для таймера разумно рассчитывать на миллисекунды, а не на микросекунды.

Листинг 4.6: MicroPython: одноразовый таймер — «лестничный» свет на 3 секунды

```
from machine import Pin, Timer
import time

led = Pin(13, Pin.OUT)
button = Pin(12, Pin.IN, Pin.PULL_UP)
off_timer = Timer(-1)

def turn_off(t):
    """Сработает один раз, через 3 секунды после последнего нажатия."""
    led.value(0)

def start_light():
    led.value(1)
    # повторный init() перезапускает отсчёт: свет «продлевается»
    off_timer.init(mode=Timer.ONE_SHOT, period=3000, callback=turn_off)

prev = 1
while True:
    now = button.value()
    if prev == 1 and now == 0:       # нажатие
        start_light()
    prev = now
    time.sleep_ms(20)                # заодно и грубый антидребезг
```

Это решение похоже на лампу в подъезде: нажатие включает свет, и он гаснет сам через три секунды; повторное нажатие заново «заводит» отсчёт, потому что `init()` перезапускает таймер.

А как в CircuitPython?

В CircuitPython нет класса `Timer` с колбэками: вся программа выполняется как один поток, а вмешательство «из-под руки» не поощряется. Привычные задачи решают иначе:

- опрос времени в цикле (как в планировщике выше);
- сопрограммы `asyncio` (урок 13);
- аппаратные средства: `pwmio` для импульсов, `countio` для подсчёта фронтов, модуль `alarm` для сна по таймеру.

Листинг 4.7: CircuitPython: аналог периодического таймера на тиках

```
import board
import digitalio
import supervisor

# В CircuitPython нет класса Timer с колбэками.
# Делаем то же самое «на тиках»: сравниваем текущее время с назначенным.
PERIOD = 2 ** 29

def ticks_diff(new, old):
    d = (new - old) & (PERIOD - 1)
    return d - PERIOD if d >= PERIOD // 2 else d

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

BLINK_MS = 500                  # переключение 2 раза в секунду
REPORT_MS = 1000

t_blink = supervisor.ticks_ms()
t_report = t_blink
seconds = 0

while True:
    now = supervisor.ticks_ms()
    if ticks_diff(now, t_blink) >= BLINK_MS:
        t_blink = (t_blink + BLINK_MS) % PERIOD    # прибавляем период, а не «now»
        led.value = not led.value
    if ticks_diff(now, t_report) >= REPORT_MS:
        t_report = (t_report + REPORT_MS) % PERIOD
        seconds += 1
        print("прошло секунд:", seconds)
```

Листинг 4.8: CircuitPython: аналог одноразового таймера

```
import board
import digitalio
import supervisor

PERIOD = 2 ** 29

def ticks_diff(new, old):
    d = (new - old) & (PERIOD - 1)
```

```
    return d - PERIOD if d >= PERIOD // 2 else d

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

button = digitalio.DigitalInOut(board.GP12)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP

LIGHT_MS = 3000
off_at = None                    # когда нужно погасить свет (None — таймер не запущен)
prev = True

while True:
    now = supervisor.ticks_ms()
    cur = button.value
    if prev and not cur:                          # нажатие
        led.value = True
        off_at = (now + LIGHT_MS) % PERIOD     # «заводим» одноразовый таймер заново
    prev = cur
    if off_at is not None and ticks_diff(now, off_at) >= 0:
        led.value = False                         # одноразовый таймер сработал
        off_at = None
```

4.5 Учимся программировать: класс «Задача»

Запись «`t_xxx = ...; if ticks_diff(...)`» повторяется для каждой задачи. Пора сделать её классом: объект хранит функцию, период и момент следующего запуска, а метод `run()` решает, пора ли. Заметьте красивую вещь: в атрибут `func` мы кладём *саму функцию* (без скобок). В Python функция — такое же значение, как число или строка, поэтому её можно хранить в переменной и передавать в другие функции. Так устроены все колбэки.

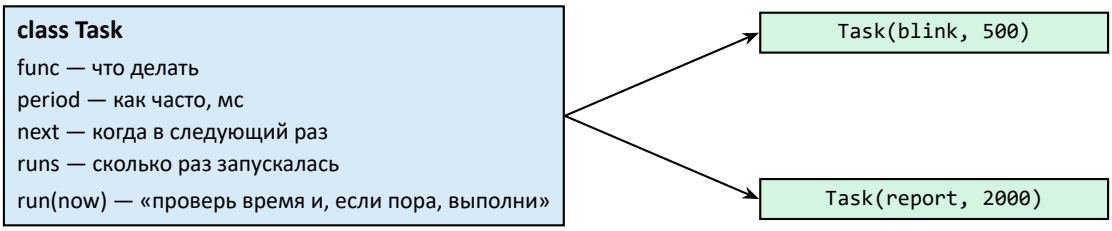


Рис. 4.5: Один класс — сколько угодно независимых задач

Листинг 4.9: MicroPython: класс `Task` и две задачи

```
from machine import Pin
import time

class Task:
    """Неблокирующая задача: вызывает func каждые period_ms миллисекунд."""
    def __init__(self, func, period_ms):
        self.func = func
        self.period = period_ms
        self.next = time.ticks_add(time.ticks_ms(), period_ms)  # когда запускать в следующий
        ↪  раз
        self.runs = 0                                           # сколько раз уже запускалась

    def run(self, now):
        if time.ticks_diff(now, self.next) >= 0:
```

```
            self.next = time.ticks_add(self.next, self.period)  # шаг по сетке, без накопления
            ↪  сдвига
            self.func()
            self.runs += 1

led = Pin(13, Pin.OUT)

def blink():
    led.toggle()

def report():
    print("мигнул раз:", t_blink.runs, " отчётов:", t_report.runs)

t_blink = Task(blink, 500)
t_report = Task(report, 2000)
tasks = [t_blink, t_report]      # сюда легко добавить новые задачи

while True:
    now = time.ticks_ms()
    for task in tasks:
        task.run(now)
```

Листинг 4.10: CircuitPython: тот же класс на `supervisor.ticks_ms()`

```
import board
import digitalio
import supervisor

PERIOD = 2 ** 29

def ticks_diff(new, old):
    d = (new - old) & (PERIOD - 1)
    return d - PERIOD if d >= PERIOD // 2 else d

class Task:
    """Неблокирующая задача: вызывает func каждые period_ms миллисекунд."""
    def __init__(self, func, period_ms):
        self.func = func
        self.period = period_ms
        self.next = (supervisor.ticks_ms() + period_ms) % PERIOD
        self.runs = 0

    def run(self, now):
        if ticks_diff(now, self.next) >= 0:
            self.next = (self.next + self.period) % PERIOD   # шаг по сетке времени
            self.func()
            self.runs += 1

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

def blink():
    led.value = not led.value

def report():
    print("мигнул раз:", t_blink.runs, " отчётов:", t_report.runs)

t_blink = Task(blink, 500)
```

```
t_report = Task(report, 2000)
tasks = [t_blink, t_report]

while True:
    now = supervisor.ticks_ms()
    for task in tasks:
        task.run(now)
```

Список `tasks` можно пополнять: добавить новую задачу — одна строка. Основной цикл при этом не меняется. Это и есть смысл абстракции: детали спрятаны внутри класса.

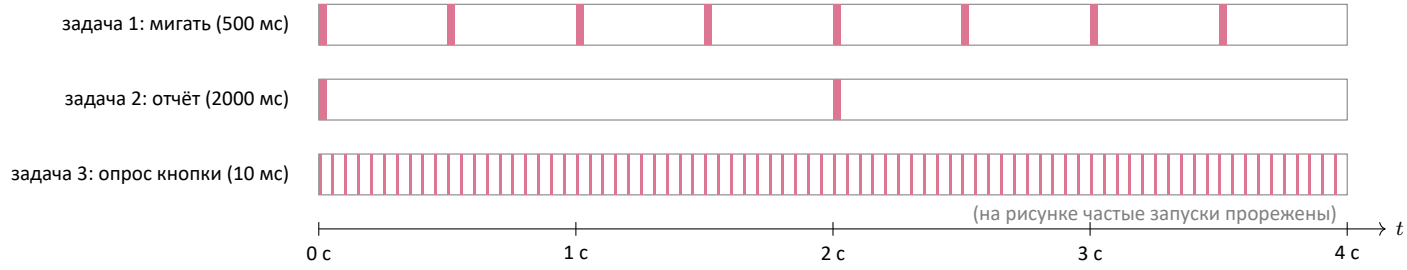


Рис. 4.6: Расписание трёх задач: у каждой свой период, а процессор почти всё время свободен

4.6 Сколько времени занимает код: замер

Прежде чем планировать задачи, полезно знать, сколько длится каждая. Для этого код «оборачивают» в две проверки времени: до и после.

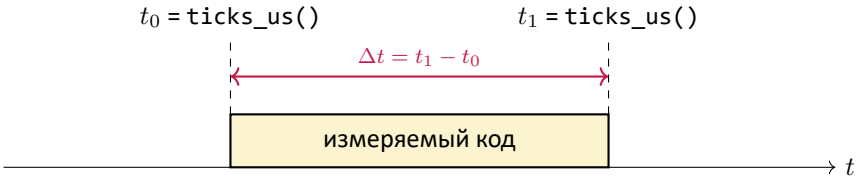


Рис. 4.7: Принцип измерения: время выполнения — разность показаний часов

Короткие операции выполняются быстрее, чем «тикают» часы, поэтому код повторяют много раз (1000) и делят общее время на число повторов. В **MicroPython** берём микросекунды (`ticks_us()`), в **CircuitPython** — целые наносекунды (`monotonic_ns()`), чтобы не терять точность дробных чисел.

Листинг 4.11: **MicroPython**: измеряем время выполнения функций

```
from machine import Pin
import math
import time

led = Pin(13, Pin.OUT)

def measure(func, repeats=1000):
    """Среднее время одного вызова func в микросекундах."""
    t0 = time.ticks_us()
    for _ in range(repeats):
        func()
    dt = time.ticks_diff(time.ticks_us(), t0)
    return dt / repeats

def empty():
```

```
    pass

def toggle():
    led.toggle()

def calc():
    return math.sin(1.234) * math.sqrt(2.0)

def loop100():
    s = 0
    for i in range(100):
        s += i
    return s

tests = (("пустая функция", empty),
         ("led.toggle()", toggle),
         ("sin * sqrt", calc),
         ("цикл на 100", loop100))

for name, func in tests:
    print("%-16s %8.1f мкс" % (name, measure(func)))
```

Листинг 4.12: CircuitPython: то же

```
import math
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

def measure(func, repeats=1000):
    """Среднее время одного вызова func в микросекундах."""
    t0 = time.monotonic_ns()         # целое число наносекунд: точность не теряется
    for _ in range(repeats):
        func()
    dt = time.monotonic_ns() - t0
    return dt / repeats / 1000       # наносекунды -> микросекунды

def empty():
    pass

def toggle():
    led.value = not led.value

def calc():
    return math.sin(1.234) * math.sqrt(2.0)

def loop100():
    s = 0
    for i in range(100):
        s += i
    return s

tests = (("пустая функция", empty),
         ("led.value = ...", toggle),
         ("sin * sqrt", calc),
```

```
        ("цикл на 100", loop100))

for name, func in tests:
    print("%-16s %8.1f мкс" % (name, measure(func)))
```

Запустите оба примера и сравните: у вас получатся свои числа (они зависят от модели платы, частоты, версии прошивки). Для ориентира на диаграмме показан типичный *характер* результатов: вызов вывода или пустая функция измеряются единицами микросекунд, а математика и циклы — десятками и сотнями.

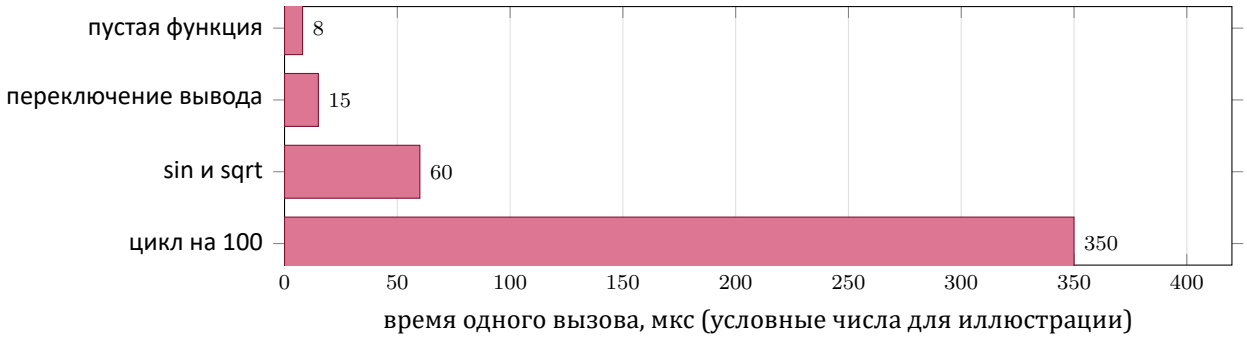


Рис. 4.8: Условный пример результатов замера: сравнивать нужно свои числа, но порядок «дешёвое/дорогое» узнаётся

Загрузка процессора

Зная время выполнения t_i каждой задачи и её период T_i, можно оценить, насколько занят процессор:

$$U = \sum_i \frac{t_i}{T_i}.$$

Если U близко к 1 (100 %), задачи не успевают, и планировщик начнёт пропускать сроки.

Задача	время t_i	период T_i	вклад t_i/T_i
Мигание светодиодом	20 µs	500 ms	0,004 %
Опрос кнопки	15 µs	10 ms	0,15 %
Печать отчёта	2 ms	2000 ms	0,1 %
Итого			0,25 %

Таблица 4.1: Пример расчёта загрузки (условные числа). Запас огромный: процессор простаивает

4.7 Конечные автоматы

Представьте светофор. В каждый момент он находится в *одном* из нескольких состояний: «красный», «красный+жёлтый», «зелёный», «жёлтый». Что он делает, зависит только от состояния и от событий: вышло время, нажата кнопка. Такое описание называется **конечным автоматом** (finite state machine, FSM): состояний конечное число, в каждый момент активно ровно одно.



Рис. 4.9: Схема работы автомата: события и текущее состояние определяют действия и следующее состояние

Преимущества автомата перед «лесом» из `if`:

- поведение видно на диаграмме: по ней можно проверить логику, не запуская программу;
- добавить новое состояние легко, и ничего не ломается;
- в каждый момент точно известно, «где мы находимся».

Светофор

Состояния и переходы светофора удобно рисовать **диаграммой состояний**: состояния — кружки, переходы — стрелки с подписью-условием.

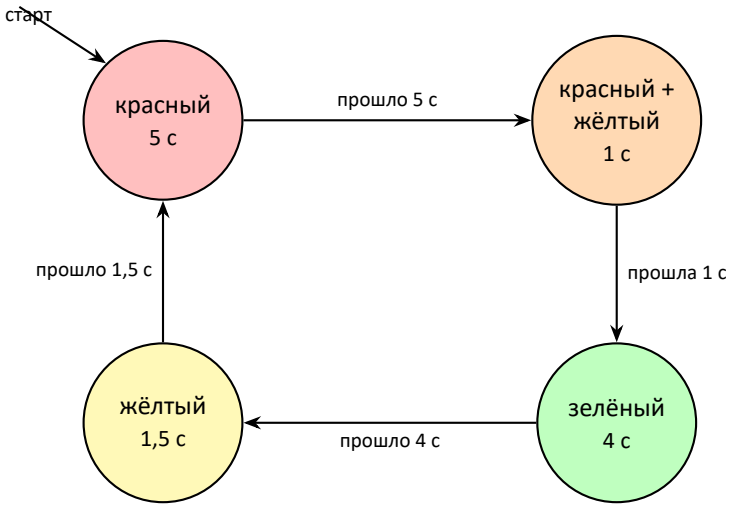


Рис. 4.10: Диаграмма состояний светофора: цикл из четырёх состояний, переход — по времени

Состояние	красный	жёлтый	зелёный	время, мс	дальше
RED	1	0	0	5000	RED_YEL
RED_YEL	1	1	0	1000	GREEN
GREEN	0	0	1	4000	YELLOW
YELLOW	0	1	0	1500	RED

Таблица 4.2: Таблица состояний светофора: каждая строка — одно состояние

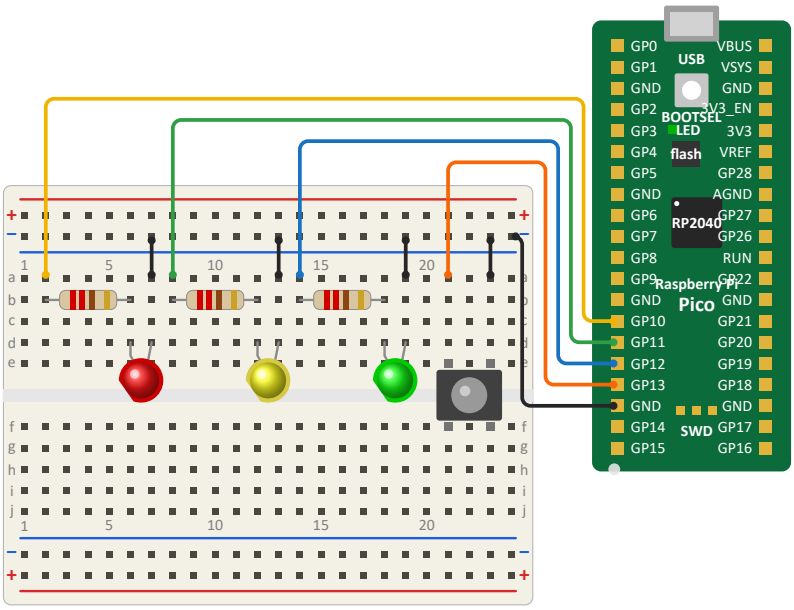
Учимся программировать: словарь как таблица

Таблицу из рисунка можно перенести в программу почти без изменений, в виде **словаря** (`dict`). Ключ — имя состояния, значение — кортеж с описанием. Строка `r, y, g, _, _ = STATES[name]` *распаковывает* кортеж по переменным; символ _ означает «это значение мне не нужно». Главное достоинство: чтобы изменить длительности или добавить состояние, не нужно трогать ло-

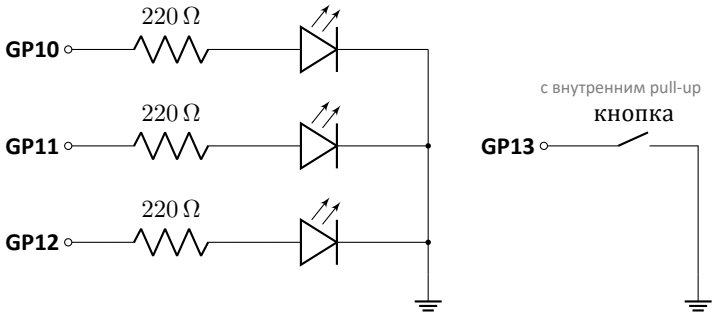
гику — меняется только таблица. Это называют **программированием, управляемым данными**.

Монтаж светофора

Красный, жёлтый и зелёный светодиоды подключены к **GP10**, **GP11**, **GP12**, кнопка пешехода — к **GP13** (с внутренним pull-up, как в уроке 3).



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 4.11: Светофор: светодиоды на GP10, GP11, GP12 и кнопка пешехода на GP13

Резистор у каждого светодиода свой, а катоды (короткие ножки) уходят чёрными проводами на шину GND. Кнопка стоит поперёк канавки: её правая пара контактов подключена к земле.

Программа светофора

Автомату достаточно двух переменных: текущего состояния и момента входа в него. Каждый шаг цикла проверяем: не истекло ли время состояния? Если да — переходим по таблице и выставляем выходы.

Листинг 4.13: MicroPython: светофор по таблице состояний

```
from machine import Pin
import time

red = Pin(10, Pin.OUT)
yellow = Pin(11, Pin.OUT)
green = Pin(12, Pin.OUT)

# таблица состояний: имя -> (красный, жёлтый, зелёный, длительность в мс, следующее состояние)
STATES = {
    "RED":     (1, 0, 0, 5000, "RED_YEL"),
    "RED_YEL": (1, 1, 0, 1000, "GREEN"),
    "GREEN":   (0, 0, 1, 4000, "YELLOW"),
    "YELLOW":  (0, 1, 0, 1500, "RED"),
}

def enter(name):
    """Войти в состояние: выставить выходы."""
    r, y, g, _, _ = STATES[name]
    red.value(r)
    yellow.value(y)
    green.value(g)

state = "RED"
enter(state)
t0 = time.ticks_ms()            # момент входа в текущее состояние

while True:
    now = time.ticks_ms()
    duration = STATES[state][3]
    if time.ticks_diff(now, t0) >= duration:   # время состояния вышло
        state = STATES[state][4]               # переход по таблице
        enter(state)
        t0 = now
    # здесь без всякого sleep() можно заниматься другими делами
```

Листинг 4.14: CircuitPython: тот же автомат

```
import board
import digitalio
import supervisor

PERIOD = 2 ** 29

def ticks_diff(new, old):
    d = (new - old) & (PERIOD - 1)
    return d - PERIOD if d >= PERIOD // 2 else d

def make_output(pin):
    out = digitalio.DigitalInOut(pin)
    out.direction = digitalio.Direction.OUTPUT
    return out

red = make_output(board.GP10)
yellow = make_output(board.GP11)
green = make_output(board.GP12)
```

```
19
20 # таблица состояний: имя -> (красный, жёлтый, зелёный, длительность в мс, следующее состояние)
21 STATES = {
22     "RED":     (1, 0, 0, 5000, "RED_YEL"),
23     "RED_YEL": (1, 1, 0, 1000, "GREEN"),
24     "GREEN":   (0, 0, 1, 4000, "YELLOW"),
25     "YELLOW":  (0, 1, 0, 1500, "RED"),
26 }
27
28 def enter(name):
29     r, y, g, _, _ = STATES[name]
30     red.value = bool(r)
31     yellow.value = bool(y)
32     green.value = bool(g)
33
34 state = "RED"
35 enter(state)
36 t0 = supervisor.ticks_ms()
37
38 while True:
39     now = supervisor.ticks_ms()
40     if ticks_diff(now, t0) >= STATES[state][3]:
41         state = STATES[state][4]
42         enter(state)
43         t0 = now
```

В цикле нет ни одного `sleep()`: программа свободна, и в место комментария можно поставить любые другие задачи — чтение датчиков, обновление экрана и так далее.

Добавляем пешеходную кнопку

Улучшим светофор. Пока никто не нажал кнопку, машинам горит зелёный сколько угодно. Пешеход нажимает кнопку, но переключение произойдёт не сразу: зелёный должен проработать минимум 3 секунды (чтобы светофор не «дёргался»). После этого цикл идёт обычным порядком, а на красный загорается встроенный светодиод — «можно идти». Появилась новая переменная состояния — **запрос** (`request`), и новое условие перехода.

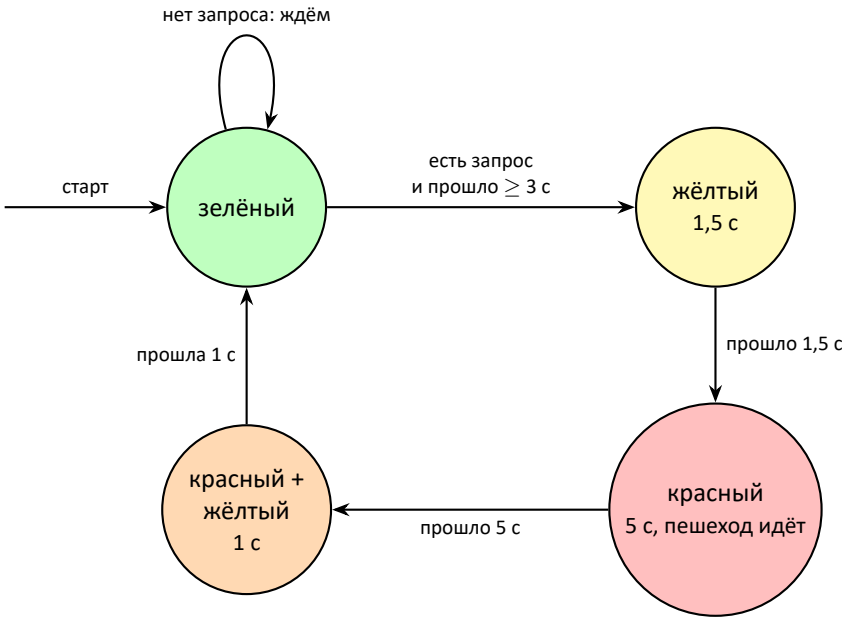


Рис. 4.12: Диаграмма состояний светофора с пешеходной кнопкой: в состоянии «зелёный» автомат может ждать сколько угодно

Листинг 4.15: MicroPython: светофор с запросом пешехода

```
from machine import Pin
import time

red = Pin(10, Pin.OUT)
yellow = Pin(11, Pin.OUT)
green = Pin(12, Pin.OUT)
walk = Pin("LED", Pin.OUT)       # встроенный светодиод: «пешеходам можно идти»
button = Pin(13, Pin.IN, Pin.PULL_UP)

MIN_GREEN = 3000                 # зелёный горит не меньше 3 с, даже если кнопку уже нажали

# состояние -> (красный, жёлтый, зелёный, пешеход)
OUTPUTS = {
    "GREEN":   (0, 0, 1, 0),
    "YELLOW":  (0, 1, 0, 0),
    "RED":     (1, 0, 0, 1),
    "RED_YEL": (1, 1, 0, 0),
}
DURATION = {"YELLOW": 1500, "RED": 5000, "RED_YEL": 1000}
NEXT = {"YELLOW": "RED", "RED": "RED_YEL", "RED_YEL": "GREEN"}

def next_state(state, elapsed, request):
    """Чистая функция: по состоянию, времени и запросу решает, куда идти."""
    if state == "GREEN":
        if request and elapsed >= MIN_GREEN:
            return "YELLOW"
        return "GREEN"                    # без запроса горим зелёным сколько угодно
    if elapsed >= DURATION[state]:
        return NEXT[state]
    return state

def show(state):
    r, y, g, w = OUTPUTS[state]
```

```
    red.value(r)
    yellow.value(y)
    green.value(g)
    walk.value(w)

state = "GREEN"
show(state)
t0 = time.ticks_ms()
request = False                  # пешеход нажал кнопку и ждёт

while True:
    now = time.ticks_ms()
    if button.value() == 0:      # дребезг не вредит: флаг можно «поднимать» много раз
        request = True
    new = next_state(state, time.ticks_diff(now, t0), request)
    if new != state:
        state = new
        t0 = now
        show(state)
        if state == "GREEN":
            request = False      # запрос выполнен
```

Листинг 4.16: CircuitPython: то же

```
import board
import digitalio
import supervisor

PERIOD = 2 ** 29

def ticks_diff(new, old):
    d = (new - old) & (PERIOD - 1)
    return d - PERIOD if d >= PERIOD // 2 else d

def make_output(pin):
    out = digitalio.DigitalInOut(pin)
    out.direction = digitalio.Direction.OUTPUT
    return out

red = make_output(board.GP10)
yellow = make_output(board.GP11)
green = make_output(board.GP12)
walk = make_output(board.LED)    # встроенный светодиод: «пешеходам можно идти»

button = digitalio.DigitalInOut(board.GP13)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP

MIN_GREEN = 3000

OUTPUTS = {
    "GREEN":   (0, 0, 1, 0),
    "YELLOW":  (0, 1, 0, 0),
    "RED":     (1, 0, 0, 1),
    "RED_YEL": (1, 1, 0, 0),
}
DURATION = {"YELLOW": 1500, "RED": 5000, "RED_YEL": 1000}
NEXT = {"YELLOW": "RED", "RED": "RED_YEL", "RED_YEL": "GREEN"}
```

```
35
36 def next_state(state, elapsed, request):
37     if state == "GREEN":
38         return "YELLOW" if (request and elapsed >= MIN_GREEN) else "GREEN"
39     if elapsed >= DURATION[state]:
40         return NEXT[state]
41     return state
42
43 def show(state):
44     r, y, g, w = OUTPUTS[state]
45     red.value = bool(r)
46     yellow.value = bool(y)
47     green.value = bool(g)
48     walk.value = bool(w)
49
50 state = "GREEN"
51 show(state)
52 t0 = supervisor.ticks_ms()
53 request = False
54
55 while True:
56     now = supervisor.ticks_ms()
57     if not button.value:          # кнопка нажата (подтяжка к 3,3 В)
58         request = True
59     new = next_state(state, ticks_diff(now, t0), request)
60     if new != state:
61         state = new
62         t0 = now
63         show(state)
64         if state == "GREEN":
65             request = False
```

Присмотритесь к функции `next_state()`: она ничего не включает и не выключает, а лишь отвечает на вопрос «какое состояние должно быть теперь?». Такие функции легко проверять в REPL: подайте любые значения и посмотрите ответ, не подключая никакого оборудования. Заодно видно, почему дребезг кнопки здесь не страшен: флаг `request` можно поднимать сколько угодно раз — результат тот же.

4.8 Самостоятельный опыт

Соберите светофор и запустите автомат без кнопки. Измерьте секундомером время красного: сходится ли оно с таблицей? Затем добавьте пешеходную кнопку. Попробуйте нажать её сразу после включения зелёного и убедитесь, что зелёный горит не менее трёх секунд.

Если не работает

- **Светофор «залип» в одном состоянии** — опечатка в имени следующего состояния в таблице: `KeyError` или, наоборот, цикл замкнулся не там.
- **`KeyError` при запуске** — ключ в таблице и в программе написан по-разному (регистр, пробелы).
- **Время состояний «плывёт»** — в цикле остался `sleep()` или тяжёлая операция; замерьте её время.
- **Мигание неточное** — в планировщике `next = now + период` вместо `ticks_add(next, период)`.

- **Таймер не срабатывает** — объект `Timer` создан внутри функции и был удалён «сборщиком мусора»; храните таймер в глобальной переменной.
- **Ошибка в колбэке таймера не видна** — вызовите колбэк вручную из REPL и посмотрите сообщение.

Итоги урока

- Время отсчитываем тиками: `ticks_ms()`, `ticks_diff()`, `ticks_add()` в MicroPython; `monotonic()`, `supervisor.ticks_ms()` в CircuitPython.
- Счётчик тиков переполняется, поэтому разность считается только через `ticks_diff()`.
- `machine.Timer` (только MicroPython) вызывает колбэк периодически или один раз; колбэк должен быть коротким.
- Планировщик «на тиках» запускает задачи по очереди; следующий момент прибавляем к плановому, а не к текущему.
- Конечный автомат: набор состояний, события и таблица переходов; удобно хранить её в словаре.
- Время выполнения кода измеряют `ticks_us()` или `monotonic_ns()`, многократно повторяя вызов.

Проверь себя

1. Почему `ticks_diff()` лучше обычного вычитания?
2. Чем `Timer.PERIODIC` отличается от `Timer.ONE_SHOT`? Что нельзя делать в колбэке?
3. Почему при планировании нужно писать `next = next + T`, а не `next = now + T`?
4. Что такое конечный автомат? Назовите его основные части.
5. Как измерить время выполнения короткой функции с достаточной точностью?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите светофор и запустите версию без кнопки. Измените длительности состояний в таблице, не трогая логику программы.
2. ★ Запустите пример с двумя таймерами в MicroPython и измените частоту мигания светодиода на 5 Гц.
3. ★★ Добавьте в светофор состояние «мигающий жёлтый» (ночной режим), в которое автомат переходит по долгому нажатию кнопки (идею возьмите из урока 3).
4. ★★ С помощью класса `Task` запустите три задачи с периодами 100 мс, 700 мс и 3 с. Выведите в консоль, сколько раз каждая выполнилась за минуту.
5. ★★★ Нарисуйте диаграмму состояний и напишите автомат для «умной лампочки»: короткое нажатие включает, ещё одно выключает; при включении она горит 10 минут и гаснет сама.
6. ★★★★ Сделайте перекрёсток из двух светофоров (шесть светодиодов). Составьте таблицу состояний, чтобы никогда не горели одновременно два зелёных. Рассчитайте загрузку процессора.

Урок 5. ШИМ: яркость, звук, серво

Цель	понять, как цифровой вывод «изображает» аналоговое напряжение с помощью широтно-импульсной модуляции, и применить ШИМ для регулировки яркости, цвета и положения сервопривода
Опирается на	урок 2: вывод как выход, светодиод и резистор; урок 4: неблокирующий отсчёт времени
Результат	светодиод плавно «дышит» с гамма-коррекцией; RGB-светодиод меняет цвет; серво поворачивается на заданный угол; вы можете назвать, какие выводы Pico делят один слайс ШИМ
Понадобится	макетная плата, 3 светодиода и резисторы $220\,\Omega$, сервопривод SG90 (или похожий) и внешний источник $5\,\text{V}$, провода

5.1 Идея урока коротко

Выводы Pico умеют выдавать только «0» или «1». А как плавно приглушить светодиод или повернуть серво на 37 градусов? Хитрость в том, чтобы переключать вывод *очень быстро*: то «1», то «0». Если доля времени с «1» мала — светодиод светит тускло, если велика — ярко. Глаз не успевает заметить мерцание и видит «среднюю» яркость. Такой приём называется **широтно-импульсной модуляцией**, ШИМ (по-английски PWM, Pulse Width Modulation). Ширина импульсов несёт информацию: в светодиоде — яркость, в серво — угол поворота, в динамике — звук.

5.2 Что такое ШИМ

ШИМ-сигнал — это прямоугольные импульсы, повторяющиеся с постоянным **периодом** T. У него две главные характеристики:

- **частота** $f = 1/T$ — сколько периодов в секунду (в герцах);
- **коэффициент заполнения** (duty cycle) $D = t_{\text{имп}}/T$ — какую часть периода сигнал равен «1». Его выражают в долях или процентах. В русскоязычной литературе часто говорят «скважность», хотя строго скважностью называют обратную величину $S = T/t_{\text{имп}}$.

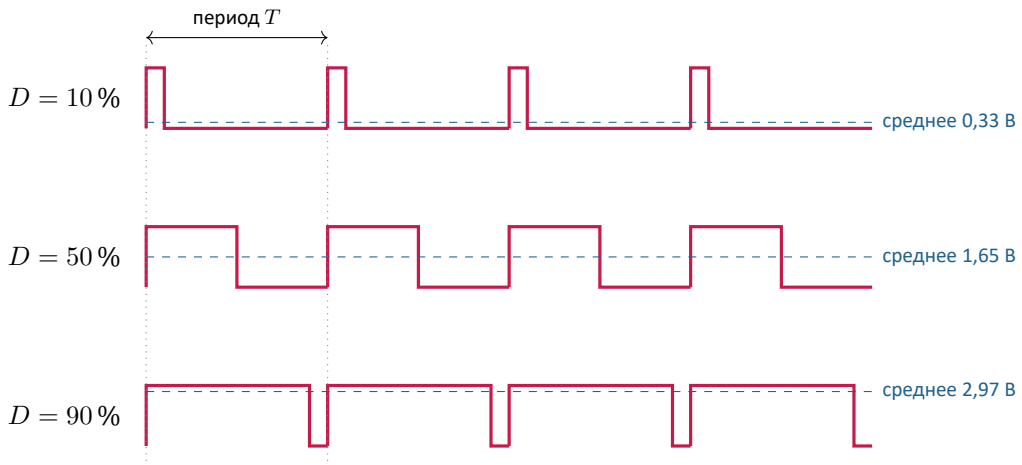


Рис. 5.1: Три ШИМ-сигнала одной частоты: чем больше D, тем выше среднее напряжение

Среднее напряжение

Пусть логическая «1» — это $U_{\text{пит}} = 3{,}3\,\text{V}$. За период сигнал равен 3,3 В долю D времени, а остальное время — нулю. Среднее напряжение — взвешенная сумма:

$$U_{\text{ср}} = D \cdot U_{\text{пит}}.$$

При $D = 25\,\%$ среднее напряжение равно $0{,}25 \cdot 3{,}3 \approx 0{,}83\,\text{V}$, при $D = 50\,\%$ — $1{,}65\,\text{V}$. Зависимость линейная: вдвое больше D — вдвое выше $U_{\text{ср}}$.

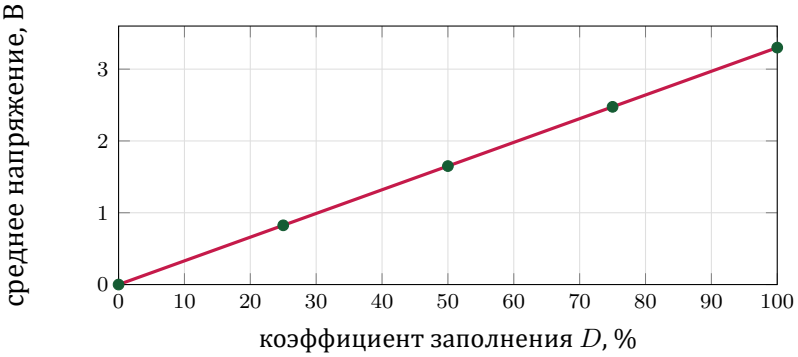


Рис. 5.2: Среднее напряжение ШИМ растёт линейно с коэффициентом заполнения

Почему светодиод «видит» среднее

У светодиода и у глаза есть «инерция»: свет не гаснет мгновенно, а глаз склеивает быстрые вспышки в сплошное свечение. Если частота выше примерно 100 Гц, мерцание не заметно; для светодиодов обычно берут $500 – 2000\,\text{Hz}$. Сам светодиод реагирует очень быстро, поэтому «усредняет» именно глаз. А вот электрическое среднее напряжение можно получить по-настоящему: достаточно пропустить сигнал через **RC-фильтр** — резистор и конденсатор.

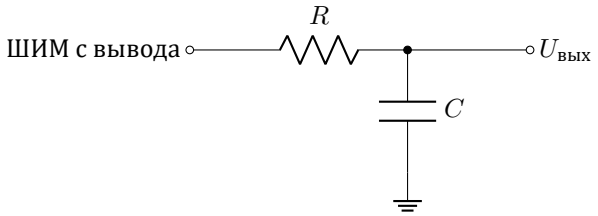


Рис. 5.3: RC-фильтр нижних частот: конденсатор сглаживает импульсы до почти постоянного напряжения

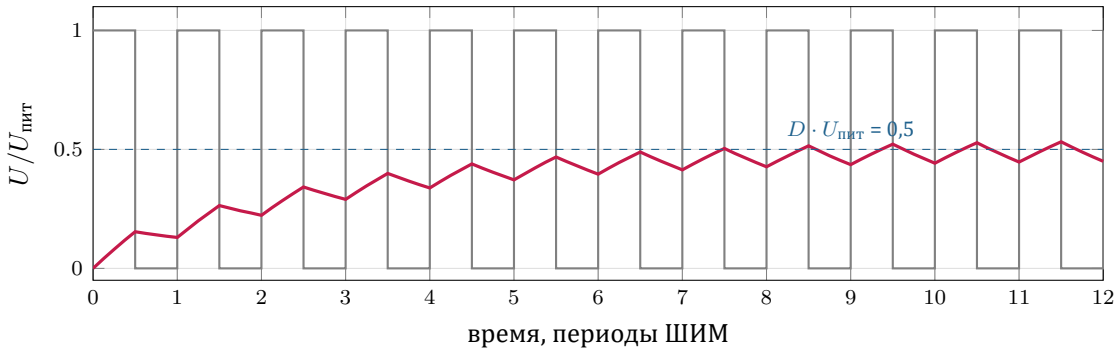


Рис. 5.4: Выход RC-фильтра при $D = 50\,\%$ и $RC = 3T$: напряжение выходит на среднее значение с небольшой «рябью»

Скорость сглаживания определяется постоянной времени $\tau = RC$ (мы встречались с ней в уроке 3). Чтобы рябь была мала, τ должна быть *много больше* периода ШИМ. Например, для $R = 10\,\text{k}\Omega$ и $C = 1\,\mu\text{F}$: $\tau = 10^{4} \cdot 10^{-6} = 10\,\text{ms}$, а частота среза $f_c = 1/(2\pi RC) \approx 16\,\text{Hz}$. Если подавать на такой фильтр ШИМ с частотой $10\,\text{kHz}$ (период $0{,}1\,\text{ms}$), рябь будет малой, и получится почти идеальный «цифро-аналоговый преобразователь» из одного вывода и двух деталей. Недостаток — медленная реакция на изменение D.

5.3 Как устроен ШИМ в RP2040

В RP2040 ШИМ делает не программа, а специальное оборудование. Оно ничего не отнимает у процессора: один раз задали частоту и D — сигнал идёт сам. Аппаратных блоков — **8 слайсов**, и у каждого **2 канала** (A и B), всего 16 независимых выходов. Каждый слайс состоит из:

- 16-битного **счётчика**, который бежит от 0 до значения `TOP` и сбрасывается — «пила»;
- двух **компараторов**: пока счётчик меньше заданного уровня (CC), выход канала равен «1», потом — «0».

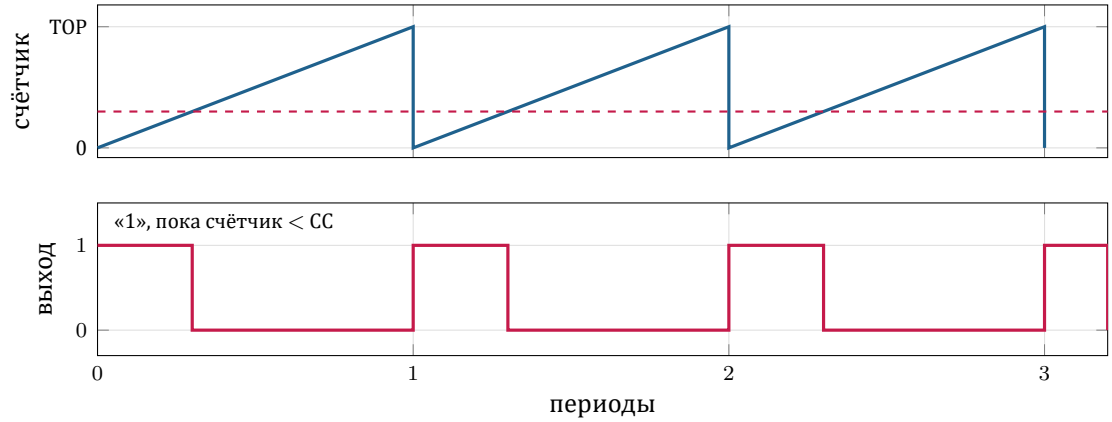


Рис. 5.5: Принцип работы канала ШИМ: счётчик-«пила» сравнивается с порогом CC. Здесь CC составляет 30 % от TOP, $D = 30\,\%$

Частота определяется скоростью счётчика и значением `TOP`: при тактовой частоте $125\,\text{MHz}$ (у Pico 2 — $150\,\text{MHz}$) число «ступенек» за период составляет $N = f_{\text{такт}}/f$. Чем выше частота ШИМ, тем меньше ступенек, то есть тем грубее регулировка D.

Частота ШИМ	ступеней за период (125 МГц)	разрядов
50 Hz (серво)	$\geq 65\,536$ (хватает 16 бит)	16
1 kHz (светодиод)	$\approx 125\,000$ (хватает 16 бит)	16
10 kHz	12 500	$\approx 13{,}6$
100 kHz	1 250	$\approx 10{,}3$
1 MHz	125	≈ 7

Таблица 5.1: Компромисс между частотой и разрешением ШИМ: функция `duty_u16()` принимает 16 бит, но реальное разрешение при высокой частоте меньше

Слайсы: кто с кем связан

Все выводы GPIO распределены по слайсам. Номер слайса и канал вычисляются по формулам:

$$\text{слайс} = \left\lfloor \frac{n}{2} \right\rfloor \bmod 8, \qquad \text{канал} = \begin{cases} \text{A}, & n \text{ чётно}, \\ \text{B}, & n \text{ нечётно}. \end{cases}$$

Например, **GP13**: слайс $\lfloor 13/2 \rfloor = 6$, канал B.

слайс 0 — общая частота	
A: GP0, GP16	B: GP1, GP17

слайс 1 — общая частота	
A: GP2, GP18	B: GP3, GP19

слайс 2 — общая частота	
A: GP4, GP20	B: GP5, GP21

слайс 3 — общая частота	
A: GP6, GP22	B: GP7

слайс 4 — общая частота	
A: GP8	B: GP9

слайс 5 — общая частота	
A: GP10, GP26	B: GP11, GP27

слайс 6 — общая частота	
A: GP12, GP28	B: GP13

слайс 7 — общая частота	
A: GP14	B: GP15

Рис. 5.6: Восемь слайсов по два канала. Выводы n и n+16 используют один и тот же канал (GP0 и GP16, GP10 и GP26...)

Осторожно

Общая частота слайса. У двух каналов A и B одного слайса *одна и та же* частота, а вот коэффициент заполнения — у каждого своя. Поэтому **GP14** и **GP15** (слайс 7) нельзя настроить на разные частоты: установка частоты на одном канале изменит её на другом. Кроме того, выводы n и n+16 (например, **GP0** и **GP16**) подключены к *одному и тому же* каналу и выдают один и тот же сигнал — их нельзя использовать независимо.

Листинг 5.1: MicroPython: два канала одного слайса и общая частота

```
from machine import Pin, PWM

# GP14 и GP15 — два канала (A и B) одного слайса номер 7
a = PWM(Pin(14))
b = PWM(Pin(15))

a.freq(1000)
a.duty_u16(16384)               # 25 %
b.duty_u16(49152)               # 75 %: скважность у каналов своя
print("частоты:", a.freq(), b.freq())

b.freq(2000)                    # меняем частоту «только у b»...
print("частоты:", a.freq(), b.freq())   # ...но она изменилась у обоих каналов слайса
```

Листинг 5.2: CircuitPython: слайс занят — другую частоту поставить нельзя

```
import board
import pwmio

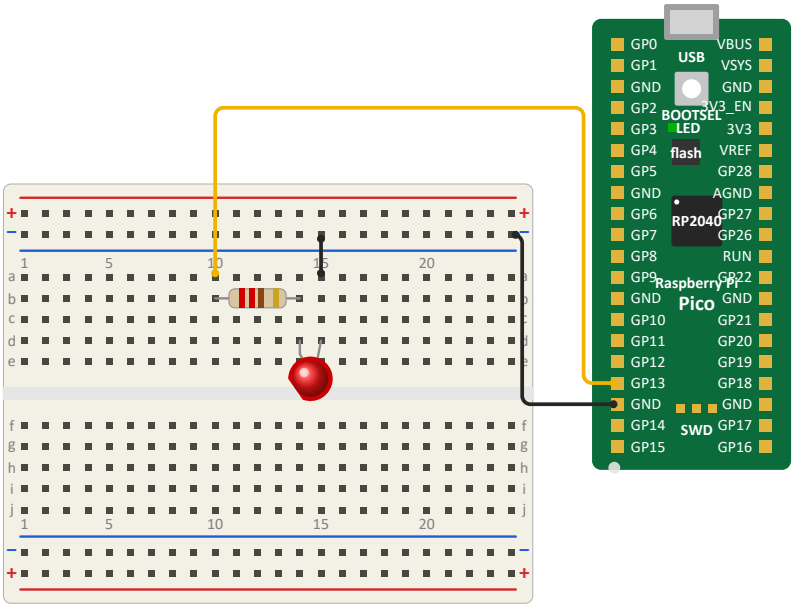
# GP14 и GP15 — два канала одного слайса: частота у них обязана совпадать
a = pwmio.PWMOut(board.GP14, frequency=1000, duty_cycle=16384)
b = pwmio.PWMOut(board.GP15, frequency=1000, duty_cycle=49152)   # та же частота: можно
print("два канала, частота", a.frequency, "Гц")

b.deinit()                      # освобождаем GP15 и пробуем другую частоту
try:
    b = pwmio.PWMOut(board.GP15, frequency=2000, duty_cycle=49152)
except ValueError as error:
    # слайс занят каналом GP14 с частотой 1000 Гц; текст ошибки зависит от версии
    print("нельзя:", error)
```

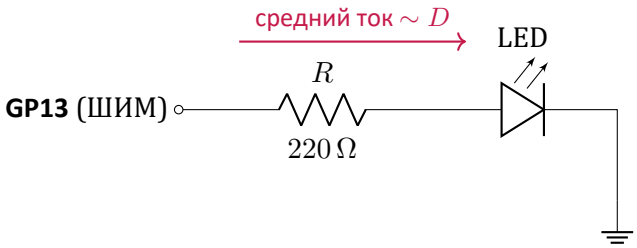
В **CircuitPython** ШИМ создаёт `pwmio.PWMOut(pin, frequency=..., duty_cycle=...)`; при несоответствии частоты на занятом слайсе конструктор выбросит `ValueError`. Коэффициент заполнения в **CircuitPython** всегда 16-битный: `duty_cycle` от 0 до 65535 (то же, что `duty_u16()` в **MicroPython**). Если нужно менять *частоту* после создания объекта, в конструктор добавляют `variable_frequency=True`.

5.4 Подключаем светодиод к ШИМ

Схема та же, что в уроке 2: вывод → резистор → светодиод → GND. Меняется только программа. Подключим светодиод к **GP13** (слайс 6, канал B).



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 5.7: Светодиод на GP13 для управления яркостью через ШИМ

Листинг 5.3: **MicroPython**: фиксированные уровни яркости

```
from machine import Pin, PWM
import time

led = PWM(Pin(13))
led.freq(1000)                      # 1 кГц: глаз мерцания не замечает

for percent in (0, 10, 25, 50, 100):
    led.duty_u16(percent * 65535 // 100)    # коэффициент заполнения: 0..65535
    print("яркость", percent, "%")
    time.sleep(1.5)

led.deinit()                        # освободить вывод и остановить ШИМ
```

Листинг 5.4: CircuitPython: то же

```
import time
import board
import pwmio

led = pwmio.PWMOut(board.GP13, frequency=1000, duty_cycle=0)

for percent in (0, 10, 25, 50, 100):
    led.duty_cycle = percent * 65535 // 100   # 16 бит: 0..65535
    print("яркость", percent, "%")
    time.sleep(1.5)

led.deinit()                          # освободить вывод
```

Строка `PWM(Pin(13))` создаёт ШИМ-объект на выводе, `freq(1000)` задаёт частоту 1 кГц, `duty_u16()` — D от 0 (всегда 0) до 65535 (почти всегда 1). Выражение `percent * 65535 // 100` переводит проценты в 16-битное число. Метод `deinit()` останавливает ШИМ и освобождает вывод.

5.5 Гамма-коррекция: почему «линейное» затухание выглядит неправильно

Запустите программу плавного разгорания.

Листинг 5.5: MicroPython: линейный разгон и затухание

```
from machine import Pin, PWM
import time

led = PWM(Pin(13))
led.freq(1000)

while True:
    # плавно разгораемся «в лоб»: коэффициент заполнения растёт линейно
    for duty in range(0, 65536, 512):
        led.duty_u16(duty)
        time.sleep_ms(10)
    # и так же гаснем
    for duty in range(65535, -1, -512):
        led.duty_u16(duty)
        time.sleep_ms(10)
```

Листинг 5.6: CircuitPython: то же

```
import time
import board
import pwmio

led = pwmio.PWMOut(board.GP13, frequency=1000, duty_cycle=0)

while True:
    # линейный разгон
    for duty in range(0, 65536, 512):
        led.duty_cycle = duty
        time.sleep(0.01)
    # линейное затухание
```

```
    for duty in range(65535, -1, -512):
        led.duty_cycle = duty
        time.sleep(0.01)
```

Вы увидите странное: почти вся «работа» происходит в самом начале, а потом светодиод почти не меняется. Дело в том, что **глаз воспринимает яркость нелинейно**: он очень чувствителен к малым яркостям и почти не различает большие. Примерно ощущение яркости пропорционально физической яркости в степени $1/2{,}2$ (так называемая гамма). Чтобы плавное изменение *казалось* равномерным, нужно подавать D не линейно, а возведённым в степень $\gamma \approx 2{,}2$:

$$D = b^{\gamma}, \qquad b \in [0;\,1] \text{ — желаемая «видимая» яркость.}$$

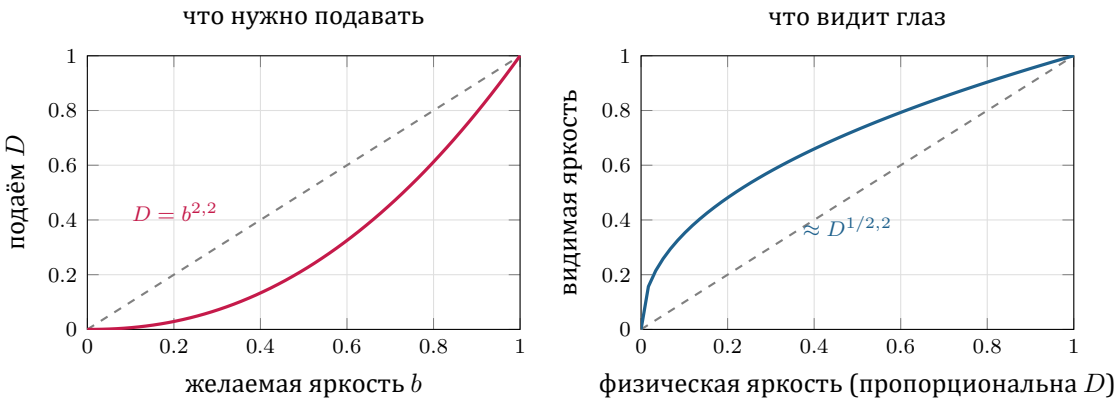


Рис. 5.8: Глаз «сжимает» яркости, поэтому для равномерной на вид регулировки D нужно подавать в степени 2,2

«Дыхание» светодиода

Теперь заставим светодиод «дышать» (плавно разгораться и гаснуть) и применим гамма-коррекцию. Чтобы не возводить в степень на каждом шаге, заранее считаем **таблицу** яркостей из 64 значений и затем просто берём их по очереди.

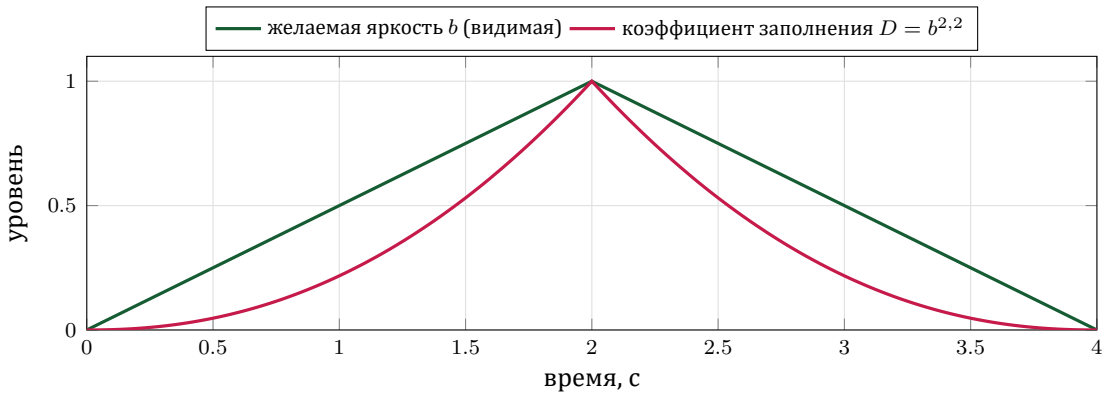


Рис. 5.9: Один цикл «дыхания»: чтобы яркость менялась равномерно, коэффициент заполнения меняется нелинейно

Листинг 5.7: MicroPython: «дыхание» с гамма-коррекцией

```python
from machine import Pin, PWM
import time

GAMMA = 2.2
STEPS = 64

# таблица яркостей считается один раз: i-я ступень -> коэффициент заполнения
table = [int(65535 * (i / (STEPS - 1)) ** GAMMA) for i in range(STEPS)]

led = PWM(Pin(13))
led.freq(1000)

while True:
    for i in range(STEPS):               # вдох
        led.duty_u16(table[i])
        time.sleep_ms(15)
    for i in range(STEPS - 1, -1, -1):   # выдох
        led.duty_u16(table[i])
        time.sleep_ms(15)
```

Листинг 5.8: CircuitPython: то же

```python
import time
import board
import pwmio

GAMMA = 2.2
STEPS = 64

# таблица яркостей считается один раз
table = [int(65535 * (i / (STEPS - 1)) ** GAMMA) for i in range(STEPS)]

led = pwmio.PWMOut(board.GP13, frequency=1000, duty_cycle=0)

while True:
    for i in range(STEPS):               # вдох
        led.duty_cycle = table[i]
        time.sleep(0.015)
    for i in range(STEPS - 1, -1, -1):   # выдох
        led.duty_cycle = table[i]
        time.sleep(0.015)
```

5.6 Учимся программировать: вычислить заранее

Строка `table = [int(65535 * (i / (STEPS - 1)) ** GAMMA) for i in range(STEPS)]` называется **списковым включением**: она одной строкой строит список, перебирая `i` от 0 до 63. Вычислять степень на каждом шаге цикла дорого (относительно): микроконтроллер тратит время на дробную арифметику. Если значения не меняются, их выгоднее посчитать *один раз* и положить в таблицу, как таблицу умножения. Такой приём называется **таблицей поиска** (lookup table) и встречается везде: в звуке, в датчиках, в графике.

5.7 RGB-светодиод: смешиваем цвета

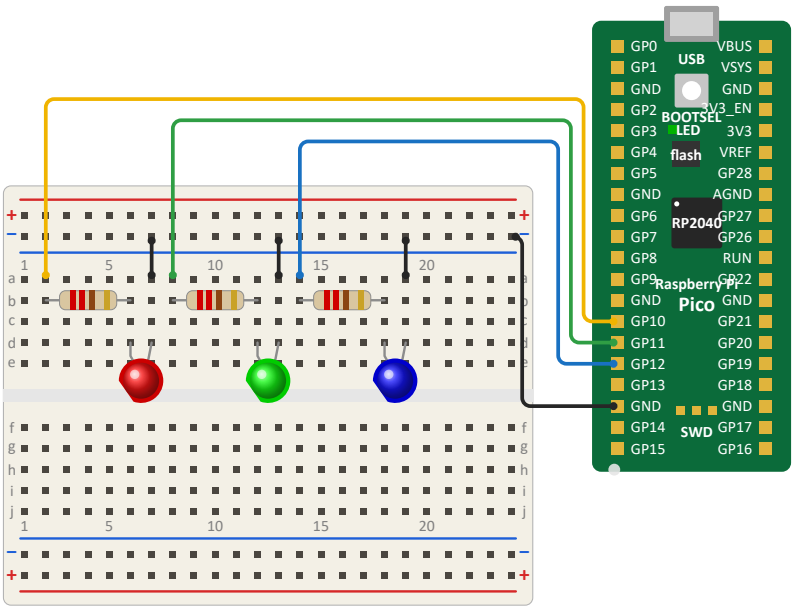
Внутри RGB-светодиода — три кристалла: красный, зелёный и синий, а выводов четыре: по одному на каждый цвет и общий. У модели с **общим катодом** (чаще всего) общая ножка — самая

длинная — подключается к GND, у модели с **общим анодом** — к 3,3 В (тогда яркость инвертируется: D нужно заменить на $1 - D$). Яркость каждого цвета регулирует свой ШИМ-канал.

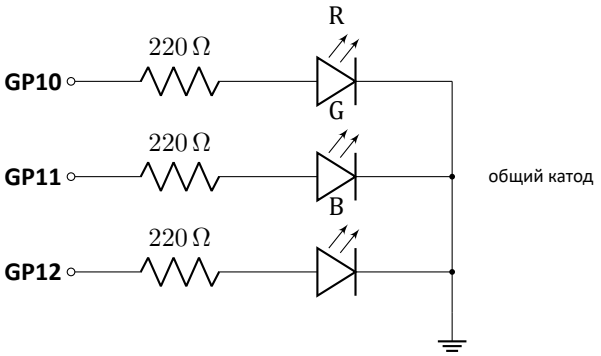


Рис. 5.10: Смешение света: три цвета в разной пропорции дают весь спектр. Подробнее о цвете — в уроке 11

Для опыта собираем схему из трёх отдельных светодиодов (красный, зелёный, синий): это то же самое, что один RGB-светодиод с общим катодом.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 5.11: Три светодиода (R, G, B) на GP10, GP11, GP12 — то же, что RGB-светодиод с общим катодом

Для настоящего RGB-светодиода вместо трёх отдельных используйте один с четырьмя ножками: красную, зелёную и синюю — через резисторы к **GP10**, **GP11**, **GP12**, длинную — к GND. У зелёного и синего кристаллов прямое напряжение выше (около 3 В), поэтому им часто ставят резисторы меньше (100 – $150\,\Omega$), чтобы цвета были одинаково яркими.

Листинг 5.9: MicroPython: смешивание цветов и «радуга»

```
from machine import Pin, PWM
import time

def make_pwm(gp):
    p = PWM(Pin(gp))
    p.freq(1000)
    return p

red = make_pwm(10)
green = make_pwm(11)
blue = make_pwm(12)

def gamma16(x):
    """x от 0 до 1 -> коэффициент заполнения с гамма-коррекцией."""
    return int(65535 * x ** 2.2)

def set_rgb(r, g, b):
    """Цвет в привычных числах 0..255."""
    red.duty_u16(gamma16(r / 255))
    green.duty_u16(gamma16(g / 255))
    blue.duty_u16(gamma16(b / 255))

def wheel(pos):
    """Цветовой круг: число 0..255 -> (r, g, b)."""
    pos %= 256
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3)

# несколько готовых цветов
for name, rgb in (("красный", (255, 0, 0)), ("зелёный", (0, 255, 0)),
                  ("синий", (0, 0, 255)), ("жёлтый", (255, 160, 0)),
                  ("белый", (255, 255, 255))):
    print(name)
    set_rgb(*rgb)
    time.sleep(1)

while True:                       # плавная смена цветов по кругу
    for pos in range(256):
        set_rgb(*wheel(pos))
        time.sleep_ms(20)
```

Листинг 5.10: CircuitPython: то же

```
import time
import board
import pwmio

red = pwmio.PWMOut(board.GP10, frequency=1000, duty_cycle=0)
green = pwmio.PWMOut(board.GP11, frequency=1000, duty_cycle=0)
blue = pwmio.PWMOut(board.GP12, frequency=1000, duty_cycle=0)

def gamma16(x):
    """x от 0 до 1 -> коэффициент заполнения с гамма-коррекцией."""
    return int(65535 * x ** 2.2)

def set_rgb(r, g, b):
    """Цвет в привычных числах 0..255."""
    red.duty_cycle = gamma16(r / 255)
    green.duty_cycle = gamma16(g / 255)
    blue.duty_cycle = gamma16(b / 255)

def wheel(pos):
    """Цветовой круг: число 0..255 -> (r, g, b)."""
    pos %= 256
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3)

for name, rgb in (("красный", (255, 0, 0)), ("зелёный", (0, 255, 0)),
                  ("синий", (0, 0, 255)), ("жёлтый", (255, 160, 0)),
                  ("белый", (255, 255, 255))):
    print(name)
    set_rgb(*rgb)
    time.sleep(1)

while True:                         # плавная смена цветов по кругу
    for pos in range(256):
        set_rgb(*wheel(pos))
        time.sleep(0.02)
```

Функция `wheel()` превращает одно число 0–255 в тройку `(r, g, b)`: так цвет плавно идёт по кругу красный → зелёный → синий → красный. Гамма-коррекция делает плавный переход смешивания равномерным.

5.8 Сервопривод

Сервопривод (серво) — это моторчик с редуктором и электроникой, которая поворачивает вал на нужный угол и удерживает его там. Вся «аналоговая» информация передаётся шириной импульса. Типовой стандарт для маленьких серво вроде SG90 (диапазон у разных моделей отличается):

- частота сигнала 50 Hz, период 20 ms;
- импульс $0{,}5$ ms – $2{,}5$ ms: примерно от 0° до 180° (часто считают 1 ms = 0°, $1{,}5$ ms = центр, 2 ms = 180°; у разных серво по-разному);

- серво само сравнивает ширину импульса с положением вала и «довинчивает» до нужного угла.



Рис. 5.12: Управление серво: частота всегда 50 Гц, угол задаётся шириной импульса

Связь угла и длины импульса — линейная: $t_{\text{имп}} = t_{\min} + (t_{\max} - t_{\min}) \cdot \alpha / 180°$.

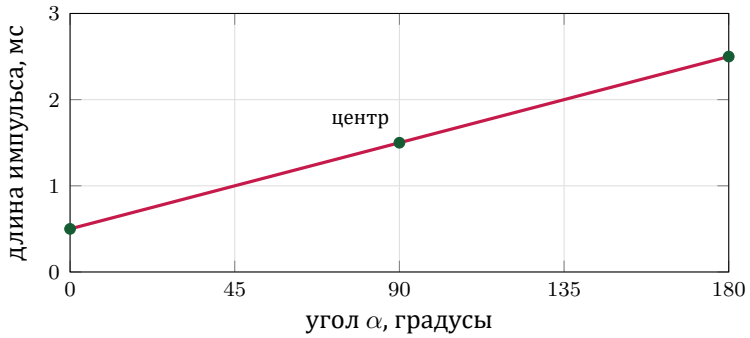


Рис. 5.13: Линейная зависимость длины импульса от угла (границы 0,5 и 2,5 мс задаются в программе)

В долях периода импульс $1{,}5$ ms при 50 Hz — это $1{,}5/20 = 7{,}5\,\%$. Приведём значения для 16-битного `duty_u16()`:

Угол	Импульс	Коэффициент заполнения	`duty_u16`
0°	0,5 ms	2,5 %	1638
45°	1,0 ms	5 %	3277
90°	1,5 ms	7,5 %	4915
135°	2,0 ms	10 %	6554
180°	2,5 ms	12,5 %	8192

Таблица 5.2: Значения для SG90 при частоте 50 Гц

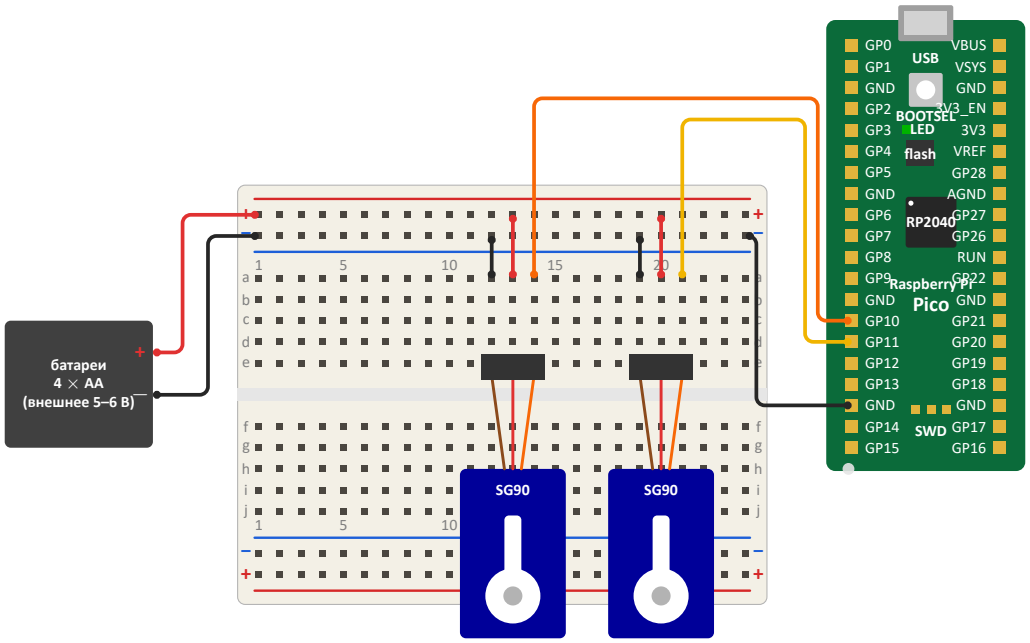
Осторожно

Питание серво. Серво — мотор: при движении и особенно при упоре он берёт ток сотни миллиампер (у SG90 при заклинивании — до 0,5–0,7 А). Это в разы больше, чем способен отдать вывод 3V3 Pico.

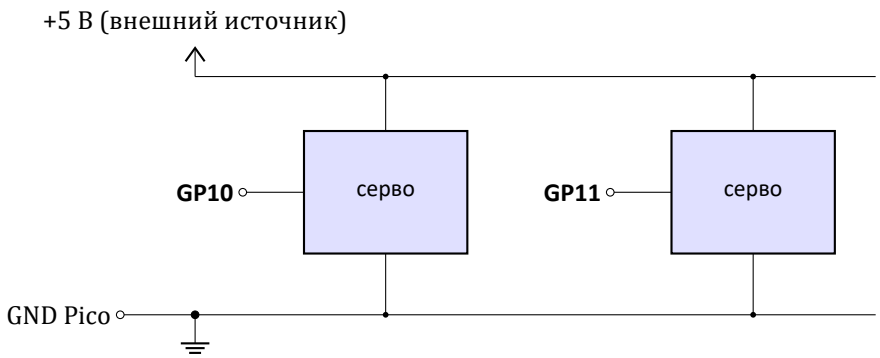
- **Не питайте серво от вывода 3V3 или от GPIO.** Красный провод — к 5 V: для одного маленького серво годится вывод VBUS (5 В с USB), для нескольких — внешний источник (4

батарейки AA дают около 6 V, подходит для SG90 с оговоркой на допустимое напряжение — смотрите документацию).

- **Земля у всех общая**: GND внешнего источника обязательно соединяют с GND Pico, иначе сигнал не будет «опираться» на ту же ноль-точку.
- Сигнальный провод (оранжевый или жёлтый) идёт напрямую на вывод Pico (логика 3,3 V для большинства серво достаточна). Коричневый или чёрный — земля, красный — питание.
- Параллельно питанию можно поставить конденсатор 220 – 470 µF: он сглаживает «просадки» при старте мотора.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 5.14: Два серво на GP10 и GP11 с внешним питанием 5 В и общей землёй

Рассмотрим монтаж: красные провода серво идут на шину «+» (к внешнему питанию), коричневые/чёрные — на шину GND, а GND этой шины соединён с GND Pico (чёрный провод справа). Оранжевый и жёлтый провода — сигналы, они идут от **GP10** и **GP11**.

Листинг 5.11: MicroPython: поворот серво на заданный угол

```
from machine import Pin, PWM
import time

MIN_US = 500                    # длина импульса для 0° (у разных серво 500...1000 мкс)
MAX_US = 2500                   # длина импульса для 180° (у разных серво 2000...2500 мкс)

servo = PWM(Pin(15))
servo.freq(50)                  # период 20 мс

def set_angle(angle):
    """Повернуть вал на угол 0..180 градусов."""
    angle = max(0, min(180, angle))                     # ограничиваем диапазон
    pulse_us = MIN_US + (MAX_US - MIN_US) * angle / 180  # линейная формула
    servo.duty_ns(int(pulse_us * 1000))                 # длина импульса в наносекундах

while True:
    for angle in range(0, 181, 5):       # от 0° до 180°
        set_angle(angle)
        time.sleep_ms(30)
    for angle in range(180, -1, -5):     # и обратно
        set_angle(angle)
        time.sleep_ms(30)
```

Листинг 5.12: CircuitPython: то же с библиотекой `adafruit_motor`

```
import time
import board
import pwmio
from adafruit_motor import servo      # библиотека adafruit_motor ставится через circup

pwm = pwmio.PWMOut(board.GP15, duty_cycle=0, frequency=50)   # 50 Гц для серво
my_servo = servo.Servo(pwm, min_pulse=500, max_pulse=2500)   # длины импульсов в мкс

while True:
    for angle in range(0, 181, 5):        # от 0° до 180°
        my_servo.angle = angle
        time.sleep(0.03)
    for angle in range(180, -1, -5):      # и обратно
        my_servo.angle = angle
        time.sleep(0.03)
```

В MicroPython функция `set_angle()` переводит угол в длину импульса в микросекундах и передаёт её в наносекундах методом `duty_ns()` — так не нужно считать проценты. В CircuitPython библиотека `adafruit_motor.servo` делает это сама: достаточно присвоить свойству `angle` число от 0 до 180. Установите её через `circup install adafruit_motor`.

Совет

Если серво упирается в край и «жужжит», сузьте диапазон: например, `MIN_US = 600`, `MAX_US = 2400`. Узнайте реальные границы своего серво экспериментом, увеличивая угол маленькими шагами. Жужжание при упоре нагревает мотор и быстро разряжает батарею.

Класс «Серво» и два привода

Для двух серво удобно написать класс — как кнопки в уроке 3. В нём прячется перевод угла в импульс, а главная программа читается как рассказ: «левое поверни на a, правое — на $180 - a$».

Листинг 5.13: MicroPython: класс `Servo` и зеркальное движение двух приводов

```
from machine import Pin, PWM
import time

class Servo:
    """Сервопривод на выводе GP: write(угол) и off()."""
    def __init__(self, gp, min_us=500, max_us=2500):
        self.pwm = PWM(Pin(gp))
        self.pwm.freq(50)
        self.min_us = min_us
        self.max_us = max_us

    def write(self, angle):
        angle = max(0, min(180, angle))
        us = self.min_us + (self.max_us - self.min_us) * angle / 180
        self.pwm.duty_ns(int(us * 1000))

    def off(self):
        self.pwm.duty_ns(0)          # импульсов нет: серво «расслабляется»

# GP10 и GP11 — один слайс ШИМ, поэтому частота у них общая — 50 Гц, это нам и нужно
left = Servo(10)
right = Servo(11)

while True:
    for a in range(0, 181, 5):
        left.write(a)                # левое серво поворачивается вперёд
        right.write(180 - a)         # правое — зеркально
        time.sleep_ms(30)
    for a in range(180, -1, -5):
        left.write(a)
        right.write(180 - a)
        time.sleep_ms(30)
```

Листинг 5.14: CircuitPython: то же

```
import time
import board
import pwmio
from adafruit_motor import servo

def make_servo(pin):
    pwm = pwmio.PWMOut(pin, duty_cycle=0, frequency=50)
    return servo.Servo(pwm, min_pulse=500, max_pulse=2500)

left = make_servo(board.GP10)
right = make_servo(board.GP11)       # GP10 и GP11 — один слайс, частота у них общая

while True:
    for a in range(0, 181, 5):
        left.angle = a               # левое серво поворачивается вперёд
        right.angle = 180 - a        # правое — зеркально
        time.sleep(0.03)
```

```
    for a in range(180, -1, -5):
        left.angle = a
        right.angle = 180 - a
        time.sleep(0.03)
```

Выводы **GP10** и **GP11** принадлежат одному слайсу (5), значит, у них общая частота. Здесь это не вредит: нужны как раз 50 Гц на обоих. А вот выбрать для серво выводы из слайсов, где потом понадобится другая частота (например, для звука), — плохая идея.

Это интересно

Звук — тоже ШИМ! Если подать на пьезопищалку ШИМ с $D = 50\,\%$ и частотой 440 Hz, прозвучит нота «ля» первой октавы. Частота определяет высоту тона. Подробнее — в уроке 21.

5.9 Самостоятельный опыт

Подключите светодиод к **GP13** и запустите по очереди: фиксированные уровни, линейное затухание и «дыхание» с гамма-коррекцией. Сравните последние два на глаз. Затем подключите серво (с отдельным питанием!) и убедитесь, что при угле 90° вал стоит посередине. Запишите границы импульсов `MIN_US` и `MAX_US`, при которых *ваш* серво не жужжит в крайних положениях.

Если не работает

- **Светодиод мерцает** — слишком низкая частота (меньше 100 Гц): поставьте `freq(1000)`.
- **Светодиод светит одинаково при любом** D — забыли `duty_u16()` или вывод остался обычным `Pin`.
- **Серво дёргается и перезагружает Pico** — питание от 3V3 или от слабого USB; перейдите на внешний источник 5 В и соедините земли.
- **Серво не слушается** — нет общей земли, перепутаны провода или частота не 50 Гц.
- **Серво гудит в крайних положениях** — импульс выходит за механический диапазон; сузьте `MIN_US` и `MAX_US`.
- **У двух выводов «загадочно» одинаковая частота** — они в одном слайсе (см. таблицу слайсов); возьмите выводы из разных слайсов.
- **`ValueError` в CircuitPython при создании `PWMOut`** — слайс уже занят другой частотой или вывод уже используется.

Итоги урока

- ШИМ — быстрое переключение вывода; среднее напряжение $U_{\text{ср}} = D \cdot U_{\text{пит}}$.
- Частота и коэффициент заполнения — две характеристики; для светодиода берут около 1 кГц, для серво — 50 Гц.
- В RP2040 восемь слайсов по два канала; каналы одного слайса делят частоту; выводы n и n+16 — один и тот же канал.
- MicroPython: `PWM(Pin(n))`, `freq()`, `duty_u16()`, `duty_ns()`; CircuitPython: `pwmio.PWMOut`, `duty_cycle`.
- Глаз нелинеен: для равномерной яркости нужна гамма-коррекция $D = b^{2{,}2}$, удобно использовать таблицу поиска.
- Серво: частота 50 Гц, импульс $0{,}5$ – $2{,}5$ ms, питание 5 В отдельно, земля общая.

Проверь себя

1. Что такое коэффициент заполнения и как по нему найти среднее напряжение?
2. Почему при частоте ШИМ около 1 кГц светодиод не мерцает на вид?
3. Что значит «выводы одного слайса делят частоту» и какие выводы это касается (приведите пример)?
4. Зачем нужна гамма-коррекция при регулировке яркости светодиода?
5. Как управляют серво с помощью ШИМ и почему его нельзя питать от вывода 3V3?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите светодиод на **GP13** и запустите фиксированные уровни яркости. Измерьте мультиметром среднее напряжение на выводе при D = 25, 50, 75 %: совпадает ли оно с расчётом?
2. ★ Измените программу «дыхания» так, чтобы полный цикл длился ровно 4 секунды (подберите шаг и паузу).
3. ★★ Сравните по очереди линейное «дыхание» и «дыхание» с гаммой 1,8 и 2,8. Какое значение даёт самое равномерное затухание на ваш взгляд?
4. ★★ Сделайте так, чтобы RGB-светодиод плавно переливался красным → синим и назад, а кнопка (урок 3) останавливала смену цвета.
5. ★★★ Сделайте «радар»: серво плавно качается от 0° до 180°; на концах делает паузу в 1 секунду. Не используйте `sleep()` внутри движения, а возьмите приём из урока 4.
6. ★★★★ Соберите RC-фильтр из резистора $10\,\text{k}\Omega$ и конденсатора $1\,\mu\text{F}$ на выводе ШИМ $10\,\text{kHz}$. Задайте D и измерьте напряжение мультиметром, затем постройте график зависимости $U(D)$ и сравните с теорией.

Урок 6. АЦП: аналоговые входы

Цель	понять, как аналоговое напряжение превращается в число, научиться читать АЦП, считать напряжение и физические величины и бороться с шумом измерений
Опирается на	урок 3: логические уровни и входы; урок 5: яркость светодиода через ШИМ
Результат	потенциометр управляет яркостью светодиода; фоторезистор включает «ночник»; термистор и внутренний датчик показывают температуру; шум подавлен усреднением и медианой
Понадобится	макетная плата, потенциометр $10\,\text{k}\Omega$, фоторезистор, резисторы $10\,\text{k}\Omega$, светодиод, NTC-термистор $10\,\text{k}\Omega$ (по желанию)

6.1 Идея урока коротко

Цифровой вход различает только «ниже порога» и «выше порога». Но мир аналоговый: яркость, температура, положение ручки, напряжение батареи меняются плавно. **Аналого-цифровой преобразователь** (АЦП, ADC) измеряет напряжение на выводе и возвращает число: чем выше напряжение, тем больше число. Дальше эту величину можно пересчитать в проценты, градусы или люксы. Главное, что нужно знать про АЦП: сколько в нём ступенек, какое напряжение считается «максимумом» и как не дать шуму испортить результат.

6.2 Квантование: измерение по ступенькам

АЦП разбивает диапазон напряжений от 0 до U_{ref} на 2^N одинаковых ступенек (для N бит). Число, которое он выдаёт, — номер ступеньки, на которую попало напряжение. Это называется **квантованием**. Ширина одной ступеньки:

$$\Delta = \frac{U_{\text{ref}}}{2^N}, \qquad \text{код} = \left\lfloor \frac{U}{\Delta} \right\rfloor .$$

Для 12-битного АЦП RP2040 и $U_{\text{ref}} = 3{,}3\,\text{V}$ получаем $\Delta = 3{,}3/4096 \approx 0{,}8\,\text{mV}$, а коды от 0 до 4095.

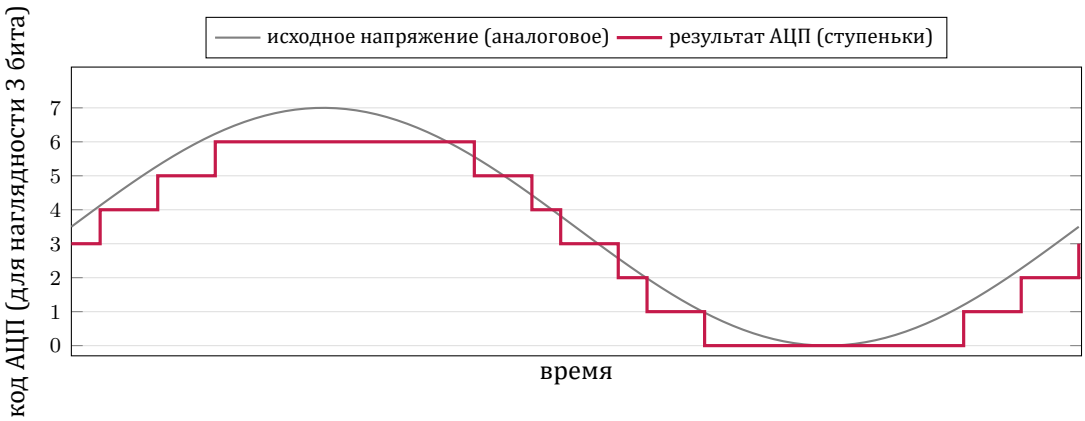


Рис. 6.1: Квантование: плавный сигнал превращается в ступенчатый. Чем больше разрядов, тем мельче ступеньки

Разрядность N	Число ступеней 2^N	Ширина ступеньки при 3,3 V
8	256	12,9 mV
10	1024	3,2 mV
12 (RP2040)	4096	0,8 mV
16	65536	0,05 mV

Таблица 6.1: Чем больше разрядов, тем точнее АЦП (теоретически)

6.3 АЦП RP2040: каналы и «16 бит»

В RP2040 один АЦП последовательного приближения (SAR) на 12 бит, который до 500 тысяч раз в секунду измеряет напряжение; к нему через **мультиплексор** по очереди подключаются пять входов.

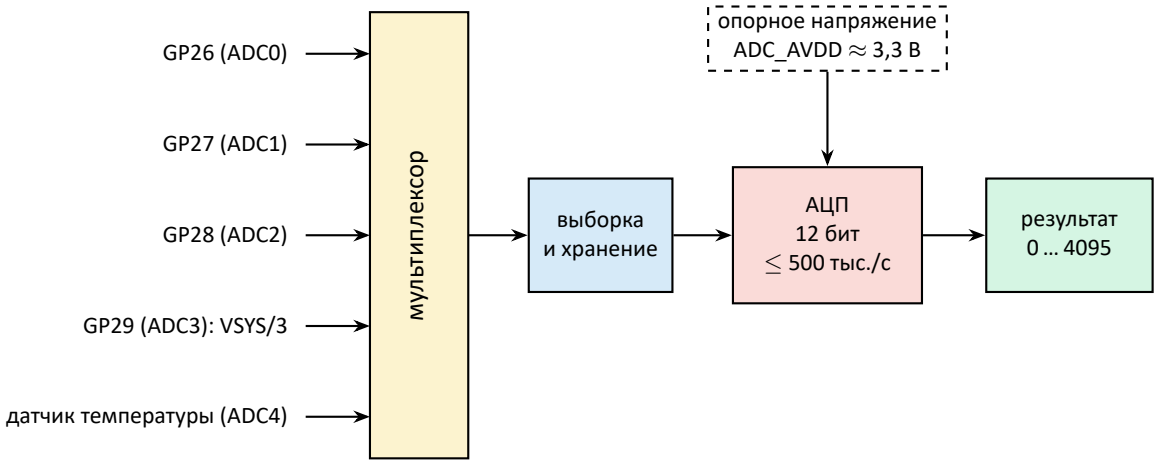


Рис. 6.2: Структура АЦП RP2040: пять входов, один преобразователь, общий опорный источник

Канал	Вывод	Контакт	MicroPython	CircuitPython
ADC0	GP26	31	`ADC(26)` или `ADC(0)`	`board.GP26` (= `board.A0`)
ADC1	GP27	32	`ADC(27)`	`board.GP27` (= `board.A1`)
ADC2	GP28	34	`ADC(28)`	`board.GP28` (= `board.A2`)
ADC3	GP29	—	`ADC(29)`, VSYS/3 (только Pico)	`board.VOLTAGE_MONITOR` (Pico)
ADC4	датчик	—	`ADC(4)`	`microcontroller.cpu.temperature`

Таблица 6.2: Каналы АЦП. Контакт 33 (AGND) — «чистая» земля для аналоговых цепей, контакт 35 (ADC_VREF) — опорное напряжение

Тонкости:

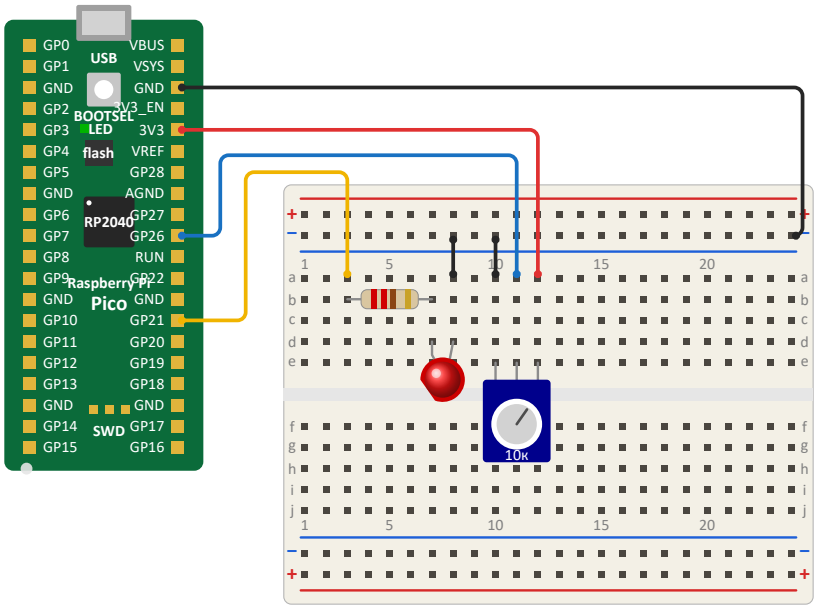
- Внешних аналоговых входов только **три**: GP26, GP27, GP28. Остальные выводы GPIO аналоговый сигнал читать не умеют.
- На Pico W вывод GP29 используется радиочипом, поэтому ADC3 (VSYS/3) там недоступен привычным способом.
- Pico 2 (RP2350) тоже имеет 12-битный АЦП; подробности — в уроке 24.

Опорное напряжение и «16 бит»

АЦП сравнивает входное напряжение с **опорным** U_{ref}. На Pico оно подаётся с питания 3V3 через небольшой фильтр (вывод ADC_VREF). Значит, если питание «плавает», плавают и результаты.

Напряжение на аналоговом входе не должно превышать U_{ref} (то есть $3{,}3\,\text{V}$), а при достижении этого значения код равен максимуму.
Чтобы программам на разных чипах было проще, и **MicroPython** (`read_u16()`), и **CircuitPython** (`AnalogIn.value`) приводят результат к **16-битной шкале** от 0 до 65535: 12-битное значение сдвигается на 4 бита влево. Реальная точность от этого не растёт — младшие 4 бита просто «заполняют» шкалу. Пересчёт в напряжение:

$$U = \text{код}_{16} \cdot \frac{U_{\text{ref}}}{65535}.$$



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 6.3: Потенциометр на GP26 (ADC0) и светодиод на GP21: плата Pico слева от макетной

Что здесь можно заметить: у потенциометра три вывода — крайние подключены к 3V3 и GND, а **средний (движок)** идёт на вход АЦП. Поворачивая ручку, вы двигаете движок между землёй и питанием, и напряжение на нём плавно меняется от 0 до $3{,}3\,\text{V}$. Вывод 3V3 на плате — это выход встроенного стабилизатора, а не вход питания, и потенциометр от него питать *можно* (ток измеряется долями миллиампера).

6.4 Читаем АЦП

Листинг 6.1: MicroPython: чтение кода и напряжения

```
from machine import ADC, Pin
import time

adc = ADC(Pin(26))              # GP26 = канал ADC0 (можно написать и просто ADC(26))
VREF = 3.3                      # опорное напряжение АЦП, В

while True:
    raw = adc.read_u16()        # 0..65535: 12-битный результат, сдвинутый на 4 бита
    volts = raw * VREF / 65535  # перевод кода в вольты
    print(raw, "%.3f В" % volts)
    time.sleep(0.5)
```

Листинг 6.2: CircuitPython: то же

```
import time
import board
import analogio

adc = analogio.AnalogIn(board.GP26)      # то же, что board.A0

while True:
    raw = adc.value                          # 0..65535
    volts = raw * adc.reference_voltage / 65535   # reference_voltage = 3,3 В
    print(raw, "%.3f В" % volts)
    time.sleep(0.5)
```

Откройте REPL и крутите ручку: числа плавно идут от 0 до 65535, а напряжение — от 0 до 3,3 В. Если вывод оставить «в воздухе», код будет произвольным, как у плавающего цифрового входа (урок 3): аналоговому входу тоже нужен источник сигнала.

В CircuitPython объект `AnalogIn` имеет свойство `value` (0–65535) и `reference_voltage` (для RP2040 это 3,3 В), поэтому формулу пересчёта можно записать без «магических» чисел. В MicroPython опорное напряжение задаёте вы сами константой `VREF`.

6.5 Делитель напряжения

Часто нужно измерить не то, что умещается в 0–3,3 В. Например, напряжение батареи 6 В вывод *убьёт*. Решение — **делитель напряжения**: два резистора последовательно, а АЦП подключён к их средней точке:

$$U_{\text{вых}} = U_{\text{вх}} \cdot \frac{R_2}{R_1 + R_2}.$$

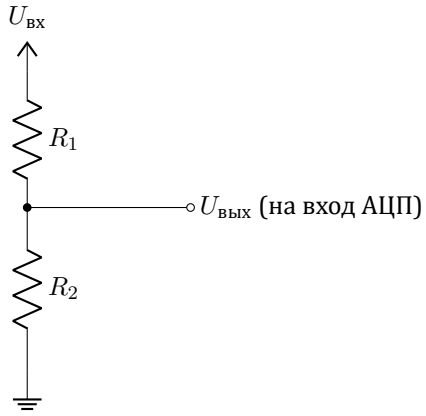


Рис. 6.4: Делитель напряжения: на АЦП попадает часть входного напряжения

Если $R_1 = R_2$, напряжение делится пополам. Для измерения до $6{,}6\,\text{V}$ подойдут два резистора по $10\,\text{k}\Omega$: при $U_{вх} = 6{,}4\,\text{V}$ (свежие четыре батарейки AA) на выводе будет $3{,}2$ В — с небольшим запасом.

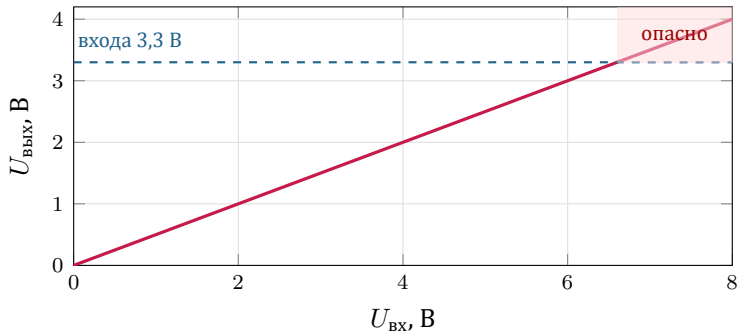


Рис. 6.5: Делитель $R_1 = R_2$: при входе выше 6,6 В напряжение на выводе превысит допустимое

Осторожно

Напряжение на аналоговом входе не должно превышать $3{,}3\,\text{V}$ ни при каких условиях, даже на миллисекунду. Считайте запас на максимальное напряжение источника и на допуск резисторов. Кроме того, АЦП лучше всего работает от источников с небольшим выходным сопротивлением (ориентир — не более 5 – $10\,\text{k}\Omega$): слишком большие резисторы делителя дадут ошибку из-за «заряда» входного конденсатора АЦП. У потенциометра $10\,\text{k}\Omega$ выходное сопротивление не превышает $R/4 = 2{,}5\,\text{k}\Omega$ — этого достаточно.

Листинг 6.3: MicroPython: монитор напряжения батареи через делитель

```
from machine import ADC, Pin
import time

adc = ADC(Pin(27))              # GP27 = ADC1; на него идёт середина делителя
low = Pin(21, Pin.OUT)          # светодиод «батарея разряжена»

DIVIDER = 2.0                   # R1 = R2 = 10 кОм: на вход приходит половина напряжения
LOW_LIMIT = 4.8                 # порог предупреждения для батареи 4 x AA, В

def battery_volts():
    total = 0
    for _ in range(16):         # усредняем 16 измерений, чтобы убрать шум
        total += adc.read_u16()
```

```
    raw = total / 16
    return raw * 3.3 / 65535 * DIVIDER

while True:
    v = battery_volts()
    low.value(1 if v < LOW_LIMIT else 0)
    print("батарея: %.2f В" % v)
    time.sleep(1)
```

Листинг 6.4: CircuitPython: то же

```
import time
import board
import analogio
import digitalio

adc = analogio.AnalogIn(board.GP27)      # GP27 = A1; на него идёт середина делителя
low = digitalio.DigitalInOut(board.GP21)
low.direction = digitalio.Direction.OUTPUT

DIVIDER = 2.0                    # R1 = R2 = 10 кОм: на вход приходит половина напряжения
LOW_LIMIT = 4.8                  # порог предупреждения для батареи 4 x AA, В

def battery_volts():
    total = 0
    for _ in range(16):          # усредняем 16 измерений, чтобы убрать шум
        total += adc.value
    raw = total / 16
    return raw * adc.reference_voltage / 65535 * DIVIDER

while True:
    v = battery_volts()
    low.value = v < LOW_LIMIT
    print("батарея: %.2f В" % v)
    time.sleep(1)
```

6.6 Практика: потенциометр

Сначала выведем показания ручки в консоль с «мёртвой зоной»: печатаем, только если значение изменилось заметно (иначе из-за шума в консоли будет мелькать каждое измерение).

Листинг 6.5: MicroPython: показания ручки в процентах и вольтах

```
from machine import ADC, Pin
import time

pot = ADC(Pin(26))

last = pot.read_u16()
while True:
    raw = pot.read_u16()
    # печатаем, только если значение изменилось заметно (около 1 % шкалы):
    # так младшие биты шума не засоряют вывод
    if abs(raw - last) > 650:
        last = raw
        percent = raw * 100 / 65535
        volts = raw * 3.3 / 65535
        print("%5.1f %%   %.2f В" % (percent, volts))
    time.sleep_ms(50)
```

Листинг 6.6: CircuitPython: то же

```
import time
import board
import analogio

pot = analogio.AnalogIn(board.GP26)

last = pot.value
while True:
    raw = pot.value
    # печатаем, только если значение изменилось заметно (около 1 % шкалы)
    if abs(raw - last) > 650:
        last = raw
        percent = raw * 100 / 65535
        volts = raw * pot.reference_voltage / 65535
        print("%5.1f %%   %.2f B" % (percent, volts))
    time.sleep(0.05)
```

А теперь соединим два урока: напряжение ручки задаёт яркость светодиода. Оба значения 16-битные, поэтому всё получается очень естественно. Для равномерной яркости применим гамма-коррекцию из урока 5.

Листинг 6.7: MicroPython: диммер — яркость светодиода регулирует потенциометр

```
from machine import ADC, Pin, PWM
import time

pot = ADC(Pin(26))
led = PWM(Pin(21))
led.freq(1000)

while True:
    x = pot.read_u16() / 65535              # положение ручки: 0.0 ... 1.0
    led.duty_u16(int(65535 * x ** 2.2))     # яркость с гамма-коррекцией (урок 5)
    time.sleep_ms(10)
```

Листинг 6.8: CircuitPython: то же

```
import time
import board
import analogio
import pwmio

pot = analogio.AnalogIn(board.GP26)
led = pwmio.PWMOut(board.GP21, frequency=1000, duty_cycle=0)

while True:
    x = pot.value / 65535                   # положение ручки: 0.0 ... 1.0
    led.duty_cycle = int(65535 * x ** 2.2)  # яркость с гамма-коррекцией (урок 5)
    time.sleep(0.01)
```

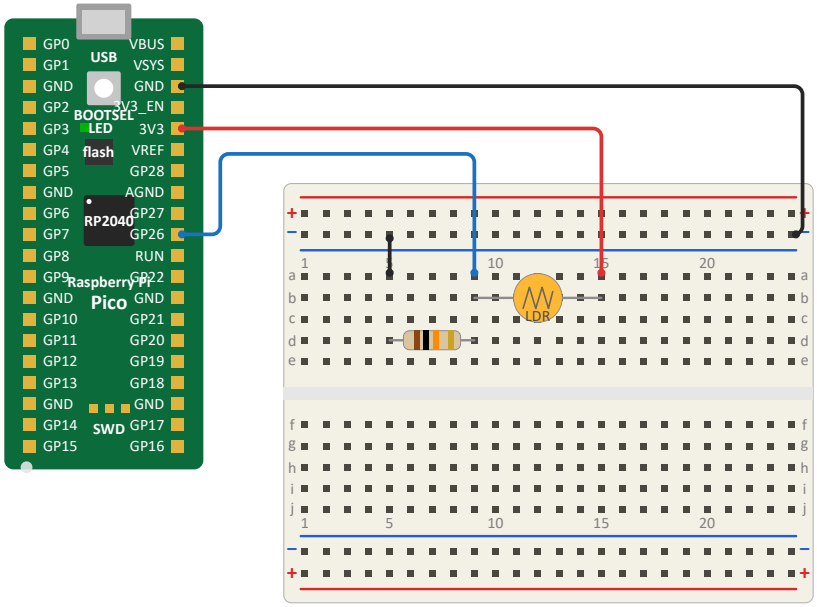
6.7 Датчик света: фоторезистор

Фоторезистор (LDR, Light Dependent Resistor) — резистор, сопротивление которого уменьшается с ростом освещённости: от сотен килоом в темноте до сотен ом–единиц килоом на ярком

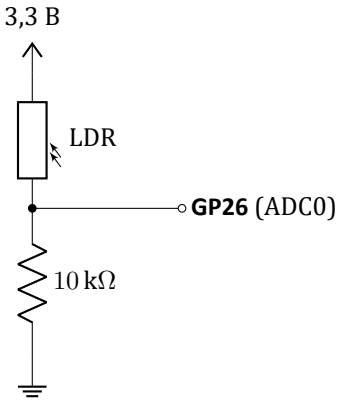
свету. Его включают в делитель напряжения вместе с обычным резистором. Если фоторезистор стоит «сверху» (между 3V3 и средней точкой), то на свету его сопротивление падает, и напряжение в средней точке растёт.



Рис. 6.6: Типичная зависимость сопротивления фоторезистора от освещённости (график условный, у реального экземпляра цифры другие)



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 6.7: Фоторезистор и резистор 10 kΩ в делителе, средняя точка на GP26

Резистор 10 kΩ в нижней части делителя выбирают примерно равным сопротивлению фоторе-

зистора при «средней» освещённости. Тогда напряжение изменяется в самом широком диапазоне.

Учимся программировать: гистерезис

Построим «ночник»: когда темно — лампа включается. Если взять одно пороговое значение, то при освещённости около порога свет будет «дрожать»: лампа сама освещает датчик, и всё начнёт мигать. Поэтому ставят **два порога**: включаем при $<$ DARK, выключаем только при $>$ BRIGHT. Область между ними — «зона нечувствительности». Это **гистерезис**: то же, что триггер Шмитта в цифровом входе (урок 3), только в программе.

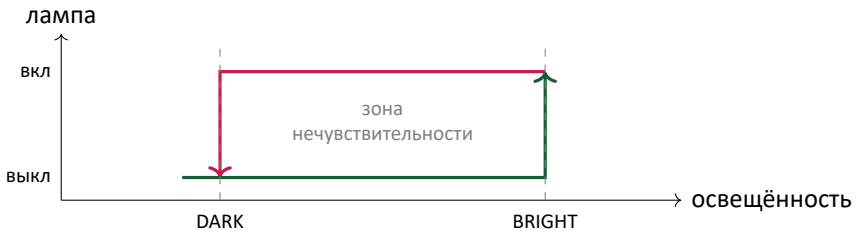


Рис. 6.8: Гистерезис: путь «выкл → вкл» и обратный проходят через разные пороги

Листинг 6.9: MicroPython: ночник с гистерезисом

```
from machine import ADC, Pin
import time

sensor = ADC(Pin(26))            # делитель: 3V3 -- фоторезистор -- GP26 -- 10 кОм -- GND
lamp = Pin(21, Pin.OUT)          # «ночник»

# Два порога = гистерезис: свет включается в темноте, а выключается,
# только когда станет заметно светлее. Подберите числа под свой датчик:
DARK = 20000                     # ниже — темно
BRIGHT = 26000                   # выше — светло

night = False
while True:
    raw = sensor.read_u16()      # на свету напряжение на GP26 растёт
    if not night and raw < DARK:
        night = True
        lamp.value(1)
    elif night and raw > BRIGHT:
        night = False
        lamp.value(0)
    print(raw, "ночь" if night else "день")
    time.sleep_ms(200)
```

Листинг 6.10: CircuitPython: то же

```
import time
import board
import analogio
import digitalio

sensor = analogio.AnalogIn(board.GP26)   # делитель: 3V3 -- фоторезистор -- GP26 -- 10 кОм --
↪ GND
lamp = digitalio.DigitalInOut(board.GP21)
```

```
lamp.direction = digitalio.Direction.OUTPUT

# Два порога = гистерезис. Подберите числа под свой датчик:
DARK = 20000                    # ниже — темно
BRIGHT = 26000                  # выше — светло

night = False
while True:
    raw = sensor.value          # на свету напряжение на GP26 растёт
    if not night and raw < DARK:
        night = True
        lamp.value = True
    elif night and raw > BRIGHT:
        night = False
        lamp.value = False
    print(raw, "ночь" if night else "день")
    time.sleep(0.2)
```

Чтобы подобрать пороги, сначала запустите программу и посмотрите числа при закрытом и освещённом датчике. Светодиод подключите как на рис. 6.3 (вывод **GP21** с резистором $220\,\Omega$).

6.8 Датчик температуры: NTC-термистор

Термистор — резистор, сопротивление которого сильно зависит от температуры. У **NTC**-термистора (negative temperature coefficient) с ростом температуры сопротивление *падает*. Его номинал обычно указывают при 25 °C: например, 10 кОм.

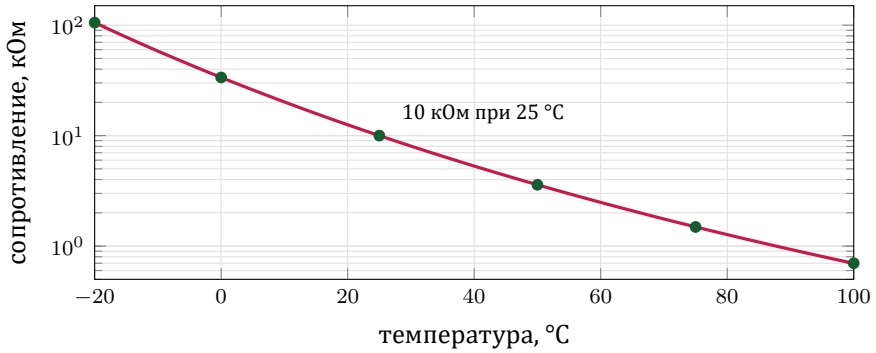


Рис. 6.9: Сопротивление NTC-термистора 10 кОм ($\beta = 3950$) в зависимости от температуры

Зависимость нелинейная, поэтому для перевода в градусы используют формулу. Простая **β-модель**:

$$R = R_0\, e^{\beta\left(\frac{1}{T}-\frac{1}{T_0}\right)} \quad\Longleftrightarrow\quad \frac{1}{T} = \frac{1}{T_0} + \frac{1}{\beta}\ln\frac{R}{R_0},$$

где T — температура в *кельвинах* ($T = t + 273{,}15$), R_0 — номинал при температуре $T_0 = 298{,}15$ K (25 °C), β — коэффициент термистора (для распространённых модулей 3435 или 3950 — смотрите описание). Для большей точности применяют **уравнение Стейнхарта—Харта**: $\frac{1}{T} = A + B\ln R + C(\ln R)^3$ с тремя коэффициентами из документации.

Термистор включаем в делитель с постоянным резистором $R_\text{ф}$: 3V3 → $R_\text{ф}$ → узел → NTC → GND. Вывод АЦП подключён к узлу. Напряжение в узле $U = 3{,}3\,\text{V}\cdot R_\text{NTC}/(R_\text{ф} + R_\text{NTC})$, откуда сопротивление термистора находится из кода АЦП:

$$R_\text{NTC} = R_\text{ф}\cdot\frac{\text{код}}{65535 - \text{код}}.$$

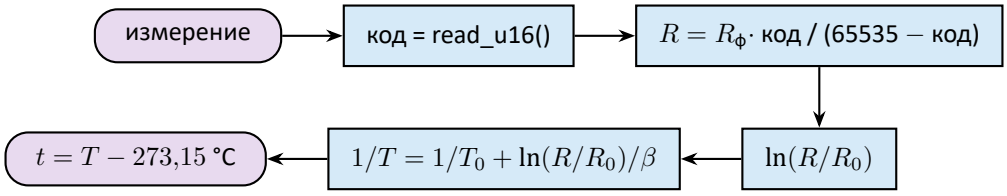


Рис. 6.10: Цепочка вычислений: код АЦП → сопротивление → температура

Листинг 6.11: MicroPython: температура по NTC-термистору (бета-модель)

```
from machine import ADC, Pin
import math
import time

adc = ADC(Pin(26))              # делитель: 3V3 -- R_FIXED -- GP26 -- NTC -- GND

R_FIXED = 10000.0               # постоянный резистор, Ом
R0 = 10000.0                    # сопротивление NTC при 25 °C, Ом
T0 = 298.15                     # 25 °C в кельвинах
BETA = 3950.0                   # коэффициент бета термистора (смотрите в документации)

def read_celsius():
    raw = adc.read_u16()
    raw = min(max(raw, 1), 65534)           # защита от деления на ноль и логарифма нуля
    r_ntc = R_FIXED * raw / (65535 - raw)   # из формулы делителя
    inv_t = 1 / T0 + math.log(r_ntc / R0) / BETA   # бета-модель
    return 1 / inv_t - 273.15               # из кельвинов в градусы Цельсия

while True:
    print("%.1f °C" % read_celsius())
    time.sleep(1)
```

Листинг 6.12: CircuitPython: то же

```
import math
import time
import board
import analogio

adc = analogio.AnalogIn(board.GP26)      # делитель: 3V3 -- R_FIXED -- GP26 -- NTC -- GND

R_FIXED = 10000.0               # постоянный резистор, Ом
R0 = 10000.0                    # сопротивление NTC при 25 °C, Ом
T0 = 298.15                     # 25 °C в кельвинах
BETA = 3950.0                   # коэффициент бета термистора

def read_celsius():
    raw = adc.value
    raw = min(max(raw, 1), 65534)           # защита от деления на ноль
    r_ntc = R_FIXED * raw / (65535 - raw)
    inv_t = 1 / T0 + math.log(r_ntc / R0) / BETA
    return 1 / inv_t - 273.15

while True:
    print("%.1f °C" % read_celsius())
    time.sleep(1)
```

Совет

Через термистор течёт ток и немного его греет (самонагрев). Для точных измерений не держите на нём напряжение постоянно и выбирайте резистор R_{ϕ} порядка номинала термистора. Строку `min(max(raw, 1), 65534)` не удаляйте: если термистор отпал, код станет 0 или 65535, и без защиты программа упадёт с ошибкой деления на ноль или логарифма нуля.

6.9 Встроенный датчик температуры

Внутри RP2040 есть датчик температуры кристалла, подключённый к пятому каналу АЦП. Это биполярный диод: напряжение на нём около $0{,}706\,\text{V}$ при 27 °C и уменьшается примерно на $1{,}721\,\text{mV}$ с каждым градусом. Из документации следует формула

$$t = 27 - \frac{U - 0{,}706}{0{,}001721}.$$

Листинг 6.13: MicroPython: температура кристалла по формуле

```
from machine import ADC
import time

sensor = ADC(4)                  # канал 4 — внутренний датчик температуры кристалла

while True:
    raw = sensor.read_u16()
    volts = raw * 3.3 / 65535
    # формула из документации RP2040: при 27 °C напряжение датчика 0,706 В,
    # с ростом температуры оно падает на 1,721 мВ на градус
    celsius = 27 - (volts - 0.706) / 0.001721
    print("%.1f °C" % celsius)
    time.sleep(1)
```

Листинг 6.14: CircuitPython: готовое свойство

```
import time
import microcontroller

while True:
    # CircuitPython сама читает датчик и пересчитывает код в градусы
    print("%.1f °C" % microcontroller.cpu.temperature)
    time.sleep(1)
```

CircuitPython делает всю работу сама: `microcontroller.cpu.temperature` возвращает градусы Цельсия. Помните, что датчик измеряет температуру *кристалла*, которая выше температуры воздуха (процессор греется сам), а точность абсолютных значений невелика — несколько градусов. Датчик хорошо показывает *изменение*: например, как нагревается плата под нагрузкой.

6.10 Шум и как с ним бороться

Реальный АЦП никогда не даёт абсолютно ровного числа: младшие разряды «прыгают» из-за теплового шума, помех от цифровых цепей и пульсаций питания. По документации RP2040 **эффективная разрядность** (ENOB) АЦП — около 8,7 бита, то есть из 12 бит 2–3 младших фактически шумят. Кроме того, в errata чипа описаны локальные скачки результата вблизи некоторых кодов (около 512, 1536, 2560 и 3584 в 12-битной шкале), из-за чего график измерений может иметь небольшие «ступеньки». Это не ошибка вашей программы.
Способы борьбы с шумом:

- **усреднение**: измерить n раз и взять среднее; случайный шум уменьшается примерно в $\sqrt{n}$ раз (4-кратное увеличение числа измерений даёт примерно +1 бит);
- **медианный фильтр**: взять n измерений (нечётное), отсортировать и выбрать *среднее по порядку*; отлично убирает одиночные «выбросы»;
- **скользящее среднее**: держать последние k значений и усреднять их при каждом новом отсчёте;
- аппаратно: короткие провода, керамический конденсатор 100 nF между входом и землёй, использование AGND (контакт 33) для аналоговой части, отдельное питание датчиков.

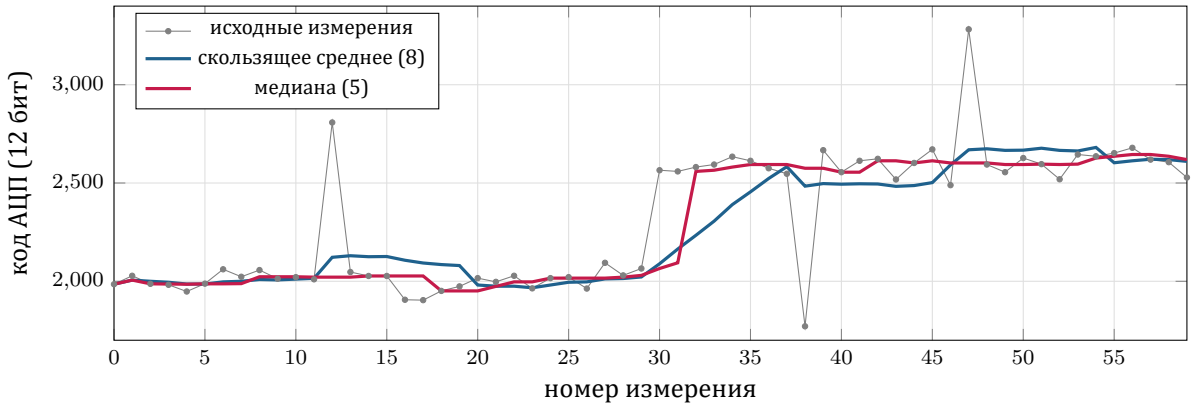


Рис. 6.11: Три выброса в данных: среднее «размазывает» выброс на несколько соседних точек, а медиана его просто отбрасывает. Зато среднее отлично сглаживает обычный шум

Выбор фильтра зависит от задачи. Для медленно меняющихся величин (температура, свет) применяют усреднение 16–64 измерений. Для сигналов с редкими выбросами (помехи от мотора, плохие контакты) — медиану. Скользящее среднее удобно, когда нужен непрерывный поток сглаженных значений без задержки на новую серию измерений. Плата за фильтрацию — запаздывание реакции: окно из k значений отстаёт примерно на $k/2$ шагов.

Листинг 6.15: MicroPython: усреднение, медиана и скользящее среднее

```
from machine import ADC, Pin
import time

adc = ADC(Pin(26))

def average(n):
    """Среднее арифметическое n измерений подряд."""
    total = 0
    for _ in range(n):
        total += adc.read_u16()
    return total // n

def median(n):
    """Медиана n измерений (n нечётное): «средний» элемент отсортированного списка."""
    values = sorted([adc.read_u16() for _ in range(n)])
    return values[n // 2]

class MovingAverage:
    """Скользящее среднее по последним size значениям."""
    def __init__(self, size, first):
        self.buf = [first] * size
        self.pos = 0
```

```
        self.total = first * size

    def add(self, x):
        self.total += x - self.buf[self.pos]     # вычли самое старое, прибавили новое
        self.buf[self.pos] = x
        self.pos = (self.pos + 1) % len(self.buf)
        return self.total // len(self.buf)

moving = MovingAverage(8, adc.read_u16())

while True:
    raw = adc.read_u16()
    # числа через пробел — их нарисует «Плоттер» в Thonny
    print(raw, average(16), median(9), moving.add(raw))
    time.sleep_ms(20)
```

Листинг 6.16: CircuitPython: то же

```
import time
import board
import analogio

adc = analogio.AnalogIn(board.GP26)

def average(n):
    """Среднее арифметическое n измерений подряд."""
    total = 0
    for _ in range(n):
        total += adc.value
    return total // n

def median(n):
    """Медиана n измерений (n нечётное)."""
    values = sorted([adc.value for _ in range(n)])
    return values[n // 2]

class MovingAverage:
    """Скользящее среднее по последним size значениям."""
    def __init__(self, size, first):
        self.buf = [first] * size
        self.pos = 0
        self.total = first * size

    def add(self, x):
        self.total += x - self.buf[self.pos]
        self.buf[self.pos] = x
        self.pos = (self.pos + 1) % len(self.buf)
        return self.total // len(self.buf)

moving = MovingAverage(8, adc.value)

while True:
    raw = adc.value
    # кортеж в скобках — формат, который понимает «Плоттер» редактора Mu и Thonny
    print((raw, average(16), median(9), moving.add(raw)))
    time.sleep(0.02)
```

Запустите программу и откройте «Плоттер» в Thonny (меню «Вид»): вы увидите четыре линии

— сырую, среднюю, медианную и скользящую. Постучите по столу рядом с проводами или коснитесь входа пальцем, чтобы появились выбросы.

Осторожно

Не забывайте, что фильтры стоят времени: усреднение 16 измерений занимает в 16 раз больше времени, чем одно. Если вы одновременно управляете сервоприводом или читаете кнопки, фильтр в основном цикле может нарушить расписание (урок 4). Тогда измеряйте по одному отсчёту за шаг цикла и используйте скользящее среднее.

Это интересно

На обычной Pico (не W) вывод GP23 управляет режимом импульсного источника питания: если установить его в 1 (`Pin(23, Pin.OUT).value(1)`), стабилизатор переходит в режим ШИМ и пульсации на 3V3 падают, а значит, шум АЦП тоже. На Pico W этот вывод относится к радиочипу — там так делать нельзя.

6.11 Самостоятельный опыт

Подключите потенциометр и посмотрите числа в REPL. Затем соберите диммер. Если есть фоторезистор — сделайте «ночник» и подберите пороги. Измерьте внутреннюю температуру, затем подержите палец на микросхеме RP2040 и посмотрите, как температура растёт. Запишите, насколько меняется показание АЦП, когда вы просто держите ручку неподвижно: это и есть шум вашей схемы.

Если не работает

- **Код всегда около 0 или 65535** — нет подключения к земле/питанию, перепутаны выводы потенциометра (средний — на АЦП) или вывод выбран не тот (аналоговые только GP26–GP28).
- **Значения «скачут»** — плохой контакт, длинные провода, нет усреднения; добавьте фильтр и конденсатор 100 nF на вход.
- **Ошибка `ValueError` при создании `ADC`** — вывод не аналоговый или номер канала неверен.
- **Температура NTC абсурдная** — перепутан порядок в делителе (формула рассчитана на NTC «снизу»), неверное β или R_0.
- **Максимум не доходит до 65535** — нормально: у 12-битного АЦП младшие биты «прыгают», а при питании от USB напряжение 3V3 чуть ниже ожидаемого.
- **Показания зависят от Wi-Fi или мотора** — пульсации питания; используйте усреднение, AGND и режим ШИМ источника (см. врезку выше).

Итоги урока

- АЦП RP2040 12-битный, но результат приводится к шкале 0–65535 (`read_u16()`, `AnalogIn.value`); напряжение $U = \text{код} \cdot 3{,}3/65535$.
- Аналоговые входы — GP26, GP27, GP28; напряжение на них не должно превышать $3{,}3$ V.
- Делитель напряжения $U_{\text{вых}} = U_{\text{вх}} R_2/(R_1 + R_2)$ позволяет измерять большие напряжения и сопротивления (фоторезистор, термистор).
- Для NTC работает β-модель: $1/T = 1/T_0 + \ln(R/R_0)/\beta$; встроенный датчик температуры

— `ADC(4)` или `microcontroller.cpu.temperature`.

- Реальная разрядность около 9 бит: измерения шумят; помогают усреднение, медианный и скользящий фильтры.
- Гистерезис (два порога) предотвращает «дребезг» решения вблизи порога.

Проверь себя

1. Что такое квантование и чему равна ширина ступеньки 12-битного АЦП при опорном напряжении $3{,}3$ V?
2. Какие выводы Pico могут читать аналоговый сигнал и каков предельно допустимый уровень напряжения на них?
3. Как рассчитать выходное напряжение делителя напряжения? Как измерить батарею 9 V?
4. Чем медианный фильтр отличается от усреднения? В каких случаях лучше медиана?
5. Зачем в программе «ночника» нужны два порога?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите потенциометр и запустите программу из урока. Запишите коды при крайних положениях ручки и сравните с 0 и 65535.
2. ★ Подключите светодиод и соберите диммер. Уберите гамма-коррекцию: замените `x ** 2.2` на `x`. В чём разница на глаз?
3. ★★ Рассчитайте делитель для измерения напряжения до 12 V (например, автомобильного аккумулятора): подберите R_1 и R_2 из ряда E12 и проверьте, что на вход попадёт не больше $3{,}3$ V.
4. ★★ Измените «ночник»: добавьте плавное включение лампы через ШИМ (урок 5) и подберите пороги гистерезиса так, чтобы лампа не «мигала» от собственного света.
5. ★★★ Сделайте термометр на NTC и сравните его показания с встроенным датчиком и с настоящим термометром. Подберите значение β так, чтобы ошибка в диапазоне 20–40 °C была минимальной.
6. ★★★★ Проведите эксперимент с шумом: снимите 1000 измерений при неподвижной ручке, посчитайте среднее и стандартное отклонение (`math.sqrt`), постройте таблицу зависимости шума от числа усредняемых измерений $n = 1, 4, 16, 64$ и проверьте закон $1/\sqrt{n}$.

Урок 7. UART и последовательный обмен

Цель	понять, как два устройства обмениваются байтами по двум проводам без общего тактового сигнала, и научиться принимать и разбирать текстовые команды
Опирается на	урок 2: цифровые выводы; урок 4: неблокирующее ожидание времени
Результат	плата отвечает на команды `LED ON`/`LED OFF`; программа разбирает предложение GPS-приёмника и проверяет его контрольную сумму; к плате подключён USB-UART адаптер или Bluetooth-модуль
Понадобится	USB-UART адаптер на 3,3 V (CP2102, CH340 или FT232), провода, по желанию — GPS-модуль NEO-6M и Bluetooth-модуль HC-05/HC-06

7.1 Идея урока коротко

UART — самый простой способ «поговорить» двум устройствам: нужны всего два сигнальных провода (один передаёт, другой принимает) и общая земля. Общего тактового сигнала нет: оба участника заранее договариваются о скорости, а каждый байт заворачивается в рамку из стартового и стоп-бита. Через UART плата общается с GPS-приёмником, Bluetooth-модулем, другой платой и компьютером. Рядом живёт ещё один канал — **USB-консоль**, по которой работают `print()` и `input()`.

7.2 Как говорят без общих часов

Представьте двух людей, которые диктуют друг другу цифры по телефону. Они заранее договорились: «говорим по одной цифре в секунду». Метронома, который стучал бы в обе трубки, нет, но если оба держат темп, всё получится. Это и есть **асинхронная** передача: у каждой стороны свои часы, а согласованность держится на одинаковой **скорости** — она называется *бодрейт* (baud rate) и измеряется в бодах: практически 1 бод = 1 бит в секунду.



Рис. 7.1: Передатчик одного устройства соединяется с приёмником другого: провода «перекрещиваются»

Главное правило подключения: **TX одного — к RX другого**. Подпись TX означает «здесь я передаю», поэтому два передатчика, соединённые вместе, будут «кричать» друг на друга. Земли тоже обязательно соединяются: напряжение всегда измеряется относительно общего нуля.

7.3 Кадр 8N1: как выглядит один байт

Пока ничего не передаётся, линия находится в состоянии покоя — на ней логическая `1` (высокий уровень). Передача байта начинается с **стартового бита** — на один бит линия опускается в `0`. Это сигнал «внимание, пошли данные»: приёмник запускает свои часы. Дальше идут 8 бит данных, *младшим битом вперёд*, а в конце — **стоп-бит**: линия возвращается в `1`. Запись «8N1» расшифровывается так: 8 бит данных, чётность (**N**one) не используется, 1 стоп-бит.

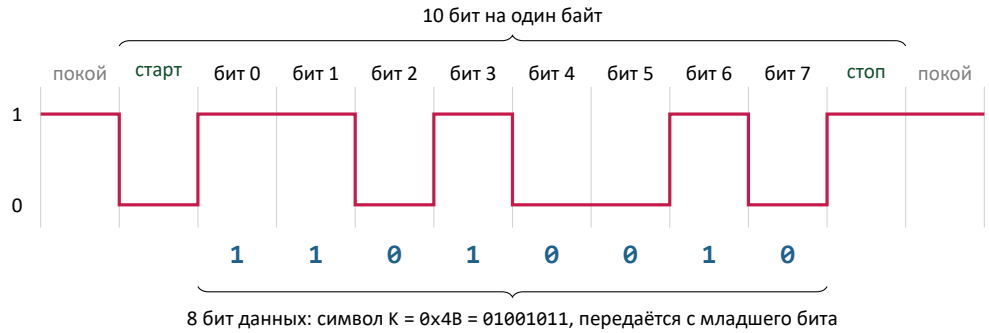


Рис. 7.2: Кадр 8N1 для символа «K». Длительность каждой ячейки равна $1/\text{бодрейт}$

Скорость, бод	Длительность бита	Длительность байта (10 бит)	Байт в секунду
9 600	104 µs	1,04 ms	960
115 200	8,7 µs	87 µs	11 520
921 600	1,1 µs	11 µs	92 160

Таблица 7.1: Скорость UART и время передачи одного байта. Из каждых 10 бит полезными являются только 8

Расчёт простой: время одного бита $T_{\text{бит}} = 1/\text{бодрейт}$, а байт занимает 10 таких бит. Строка `"hello\n"` из шести символов на скорости 9600 бод будет передаваться $6 \cdot 1{,}04 \approx 6{,}2$ мс, а на скорости 115 200 — меньше полумиллисекунды.

7.4 Почему скорости должны совпадать

Приёмник, заметив стартовый бит, ждёт полтора бита и проверяет линию *в середине* каждой ячейки — там сигнал самый устойчивый. Если часы приёмника и передатчика чуть расходятся, точка проверки с каждым битом уезжает в сторону. К последнему (десятому) биту накопленная ошибка не должна превысить половины бита: $9{,}5 \cdot \varepsilon < 0{,}5$, откуда $\varepsilon < 5\%$. На практике допуск ещё жёстче, а несовпадение настроенной скорости даёт на экране «кракозябры».

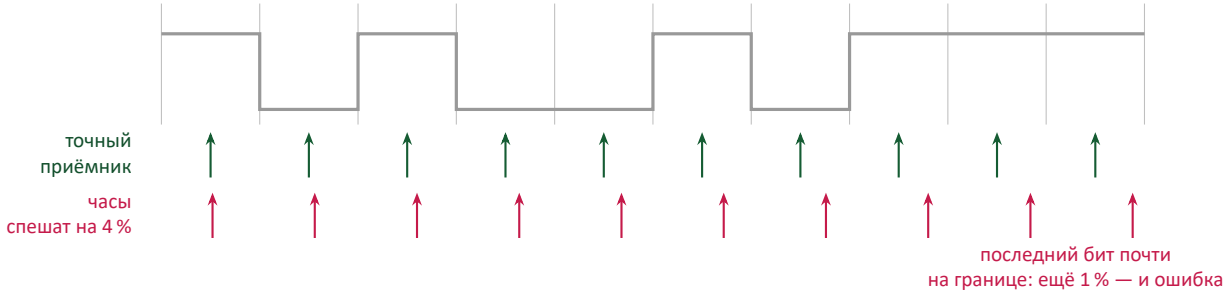


Рис. 7.3: Моменты проверки линии: у точного приёмника — середина бита, при расхождении скоростей они постепенно сдвигаются к границе

7.5 Уровни напряжения: UART, 5 В и RS-232

Слово «UART» описывает только *кадр*, а не электрические уровни. На проводах между платами используют логические уровни микроконтроллеров (TTL/CMOS), а в старых компьютерных разъёмах COM — стандарт **RS-232**, в котором уровни другие и *инвертированные*.

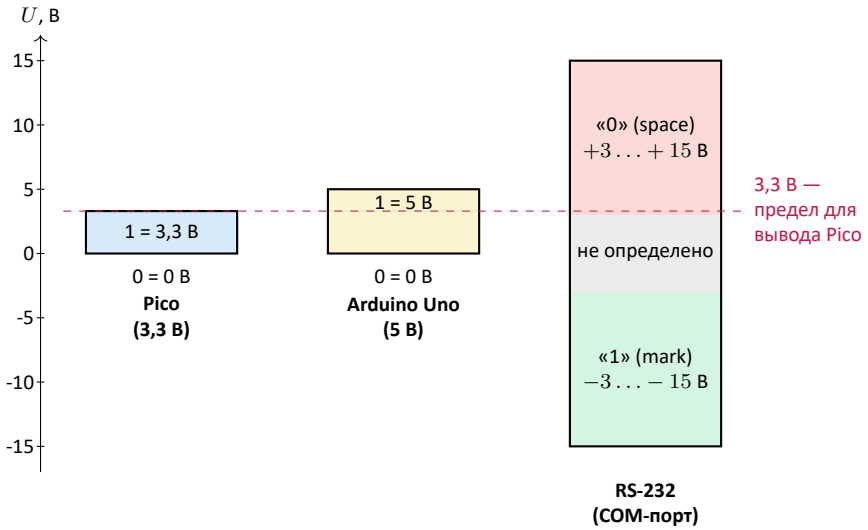


Рис. 7.4: Уровни сигнала: логика Pico, логика 5 В и RS-232 (инвертированная, с отрицательными напряжениями)

Осторожно

- Выводы Pico рассчитаны на 3,3 V. Подключать RX платы к передатчику на 5 V (многие модули, Arduino Uno) без делителя или преобразователя уровней нельзя.
- К **RS-232** (разъём COM, уровни до ±12 V) вывод Pico подключается только через микросхему-преобразователь MAX3232. Прямое соединение выведет плату из строя.
- Дешёвые USB-UART адаптеры часто имеют перемычку выбора 3,3 V/5 V. Поставьте её на 3,3 V.

7.6 UART в Pico: два блока и USB-консоль

В RP2040 и RP2350 два независимых блока UART: **UART0** и **UART1**. Каждый блок можно «развести» на несколько пар выводов; выбор определяется таблицей (каждая функция доступна только на своих GP):

Блок	TX (передача)	RX (приём)
UART0	GP0, GP12, GP16, GP28	GP1, GP13, GP17
UART1	GP4, GP8, GP20	GP5, GP9, GP21

Таблица 7.2: Выводы UART на Raspberry Pi Pico. TX и RX одного блока можно брать из разных пар

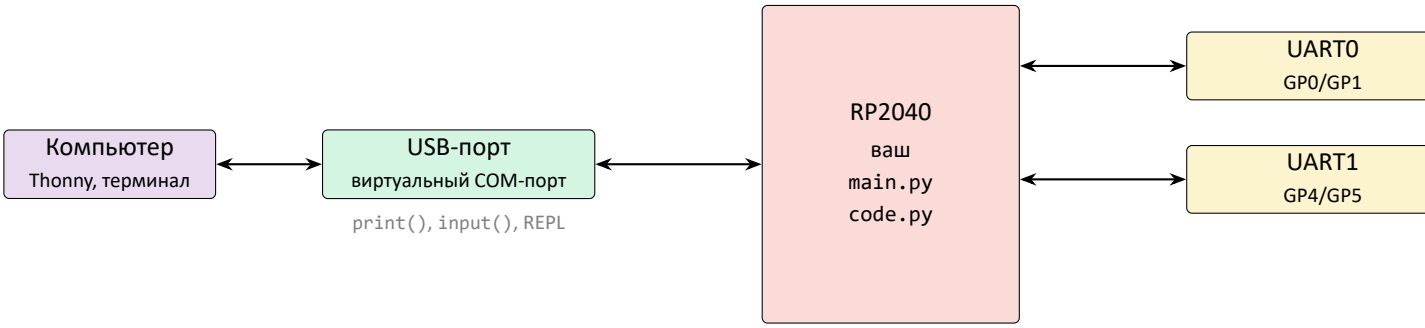


Рис. 7.5: Три канала связи: USB-консоль (всегда доступна) и два аппаратных UART

У каждого блока есть небольшая аппаратная очередь (FIFO) на 32 байта, а **MicroPython** добавляет к ней программный буфер (по умолчанию 256 байт на приём). Зачем это нужно, видно на графике: на скорости 115 200 бод в приёмник прилетает 11,5 байта каждую миллисекунду, поэтому если программа занята чем-то другим, очередь быстро переполняется и новые байты теряются.

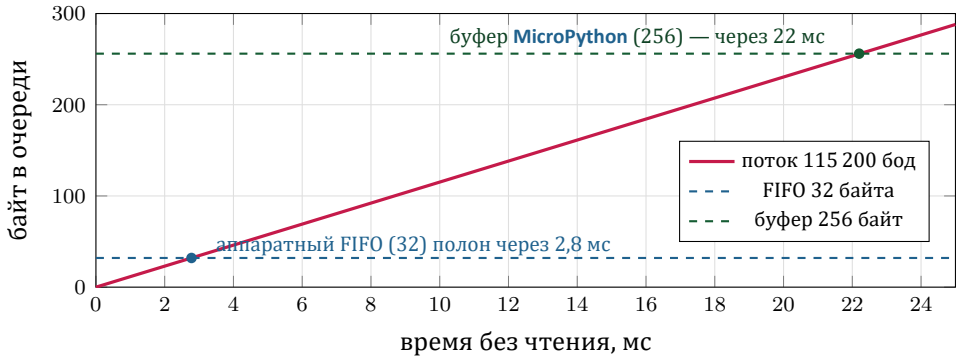
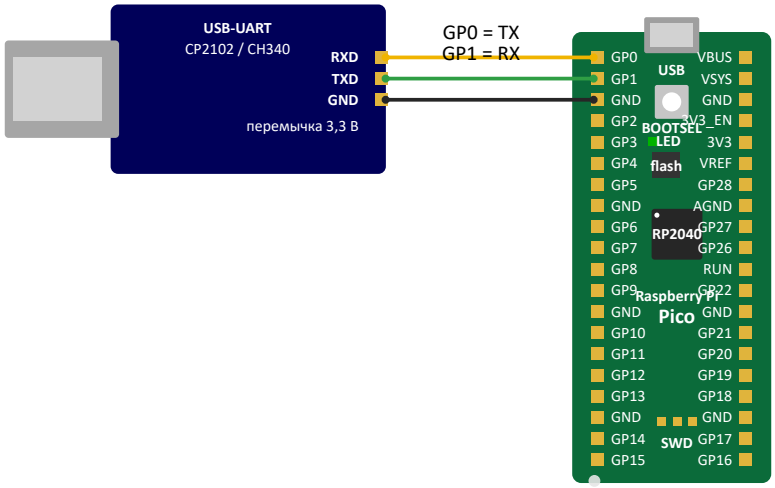


Рис. 7.6: Как быстро переполняется приёмная очередь на скорости 115 200 бод, если программа ничего не читает

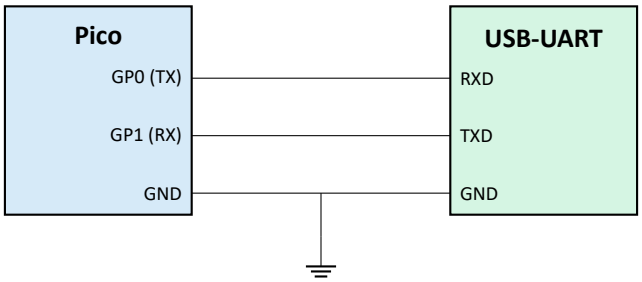
Отсюда практический вывод: не засыпайте надолго с `time.sleep()`, когда ждёте данные по UART; опрашивайте порт часто и забирайте всё, что накопилось (`uart.any()`). И не путайте UART с USB-консолью: `print()` в обычной конфигурации никак не использует UART0, а работает через USB.

7.7 Подключаем USB-UART адаптер

Чтобы увидеть работу UART своими глазами, подключим к Pico USB-UART адаптер: он создаёт на компьютере второй COM-порт, на котором можно открыть терминал (PuTTY, Thonny-терминал, `screen`, Arduino Serial Monitor). Нам понадобятся три провода: передатчик платы (GP0) к приёмнику адаптера (RXD), приёмник платы (GP1) к передатчику адаптера (TXD) и общая земля. *Питание адаптера (*5 V *или* 3,3 V*) к плате не подключаем*: плата питается от своего USB-кабеля.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 7.7: Pico и USB-UART адаптер: UART0 на выводах GP0 (TX) и GP1 (RX)

Если адаптер определился в системе, откройте в терминале его порт на скорости 115 200 бод (8N1). Теперь протестируем сам UART *без* всякого адаптера: достаточно соединить GP0 и GP1 одним проводом. Всё, что плата передаёт, сразу вернётся ей в приёмник. Это называется «шлейф» или loopback.

Листинг 7.1: MicroPython: проверка UART перемычкой GP0–GP1

```
from machine import UART, Pin
import time

# UART0: TX = GP0, RX = GP1. Соедините GP0 и GP1 перемычкой (шлейф-петля)
uart = UART(0, baudrate=115200, tx=Pin(0), rx=Pin(1), timeout=100)

for i in range(5):
    message = "hello {}\n".format(i)
    uart.write(message.encode())      # UART передаёт байты, а не строки
    time.sleep_ms(50)
    data = uart.readline()            # читаем строку до \n (или до таймаута)
    if data:
        print("принято:", data.decode().strip())
    else:
        print("ничего не принято: проверьте перемычку GP0-GP1")
```

Листинг 7.2: CircuitPython: то же

```
import time
import board
import busio

# TX = GP0, RX = GP1. Соедините GP0 и GP1 перемычкой.
# timeout здесь в СЕКУНДАХ (в MicroPython --- в миллисекундах)
uart = busio.UART(board.GP0, board.GP1, baudrate=115200, timeout=0.1)

for i in range(5):
    message = "hello {}\n".format(i)
    uart.write(message.encode())      # только байты
    time.sleep(0.05)
    data = uart.readline()            # bytes или None, если ничего не пришло
    if data:
        print("принято:", data.decode().strip())
    else:
        print("ничего не принято: проверьте перемычку GP0-GP1")
```

Действие	MicroPython	CircuitPython
Создание	`UART(0, baudrate=115200, tx=Pin(0), rx=Pin(1))`	`busio.UART(board.GP0, board.GP1, baudr`
Таймаут	`timeout=100` (мс)	`timeout=0.1` (с)
Передача	`uart.write(b"...")`	`uart.write(b"...")`
Сколько байт ждёт	`uart.any()`	`uart.in_waiting`
Чтение	`read(n)`, `readline()`, `readinto(buf)`	`read(n)`, `readline()`, `readinto(buf)`
Если данных нет	`None`	`None`
Формат кадра	`bits=8, parity=None, stop=1`	`bits=8, parity=None, stop=1`

Таблица 7.3: Основные операции UART в двух языках

Совет

UART передаёт *байты*, а не строки. Строку перед отправкой превращают в байты методом `.encode()`, принятые байты превращают в текст методом `.decode()`. Именно поэтому в примерах стоит `uart.write(message.encode())`.

7.8 Учимся программировать: байты и строки

Текст на экране и текст на проводе — разные вещи. Строка (`str`) хранит символы, а по проводу ходят **байты** (`bytes`). Одна латинская буква — один байт, одна русская — два байта в кодировке UTF-8. Поэтому `len("привет")` равно 6, а `len("привет".encode())` равно 12. Литерал байтовой строки пишется с буквой b: `b"LED ON"`.

Два метода строк понадобятся постоянно. `split()` режет строку по пробелам на список слов: `"LED ON".split()` даёт `["LED", "ON"]`. `partition(",")` режет строку по первому разделителю на три части (до него, он сам, после него). С их помощью мы разберём и команды, и GPS-предложение.

7.9 Протокол из строк: команды и ответы

Чтобы двум программам было легко договориться, придумывают **протокол** — правила, как формируются сообщения. Самый удобный для начинающих протокол: одна команда — одна

строка текста, заканчивающаяся символом перевода строки \n. Приёмнику не нужно угадывать, где конец сообщения: он копит байты в буфере, пока не встретит \n. Плата отвечает тоже строкой, например OK или ERR unknown.

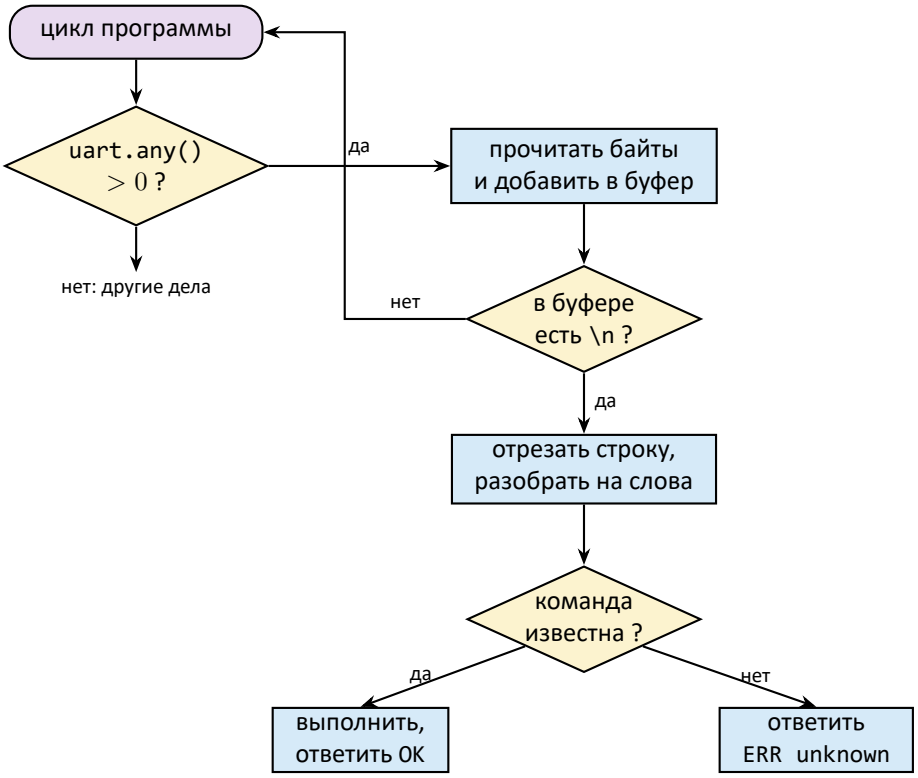


Рис. 7.8: Приём строк из UART: копим байты в буфере, пока не придёт символ конца строки

Листинг 7.3: MicroPython: управление светодиодом текстовыми командами

```
from machine import UART, Pin

led = Pin("LED", Pin.OUT)
uart = UART(0, baudrate=115200, tx=Pin(0), rx=Pin(1))

def handle(line):
    """Разбирает одну строку-команду и возвращает текст ответа."""
    parts = line.strip().upper().split()
    if not parts:
        return "ERR empty"
    cmd = parts[0]
    if cmd == "LED" and len(parts) == 2:
        if parts[1] == "ON":
            led.value(1)
            return "OK"
        if parts[1] == "OFF":
            led.value(0)
            return "OK"
        return "ERR arg"
    if cmd == "STATUS":
        return "LED=" + str(led.value())
    if cmd == "ADD" and len(parts) == 3:
        try:
            return str(int(parts[1]) + int(parts[2]))
        except ValueError:
```

```
            return "ERR number"
    return "ERR unknown"

buf = b""                           # сюда копим принятые байты
while True:
    n = uart.any()                  # сколько байт уже ждёт в буфере
    if n:
        buf += uart.read(n)
        while b"\n" in buf:         # пока в буфере есть целая строка
            raw, buf = buf.split(b"\n", 1)
            try:
                answer = handle(raw.decode())
            except UnicodeError:
                answer = "ERR decode"
            uart.write((answer + "\r\n").encode())
```

Листинг 7.4: CircuitPython: то же

```
import board
import busio
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT
uart = busio.UART(board.GP0, board.GP1, baudrate=115200, timeout=0)

def handle(line):
    """Разбирает одну строку-команду и возвращает текст ответа."""
    parts = line.strip().upper().split()
    if not parts:
        return "ERR empty"
    cmd = parts[0]
    if cmd == "LED" and len(parts) == 2:
        if parts[1] == "ON":
            led.value = True
            return "OK"
        if parts[1] == "OFF":
            led.value = False
            return "OK"
        return "ERR arg"
    if cmd == "STATUS":
        return "LED=" + str(int(led.value))
    if cmd == "ADD" and len(parts) == 3:
        try:
            return str(int(parts[1]) + int(parts[2]))
        except ValueError:
            return "ERR number"
    return "ERR unknown"

buf = b""
while True:
    n = uart.in_waiting             # сколько байт ждёт в приёмном буфере
    if n:
        buf += uart.read(n)
        while b"\n" in buf:
            raw, buf = buf.split(b"\n", 1)
            try:
                answer = handle(raw.decode())
```

```
            except UnicodeError:
                answer = "ERR decode"
            uart.write((answer + "\r\n").encode())
```

Подключите адаптер, откройте терминал на скорости 115 200 и введите `LED ON`, потом `STATUS` и `ADD 2 3`. В терминале нужно включить отправку символа конца строки (Enter с `LF` или `CRLF`); лишний символ `\r` не мешает: метод `strip()` его убирает.

Осторожно

Если программа ждёт строку, а терминал не отправляет `\n`, ответа не будет: программа правильно дожидается конца сообщения. Проверьте настройку «Line ending» в терминале.

7.10 USB-консоль: `print()` и `input()`

Самый привычный «последовательный порт» — тот, по которому работает Thonny. В обоих языках `print()` печатает в USB-консоль, а `input()` читает из неё строку. Но `input()` *блокирует* программу: пока пользователь не нажмёт Enter, дальше ничего не происходит. Чтобы совмещать диалог с другими делами, сначала проверяют, есть ли вообще данные.

Листинг 7.5: MicroPython: ввод с консоли, в том числе неблокирующий

```
import sys
import select
import time
from machine import Pin

led = Pin("LED", Pin.OUT)

# 1. Простой диалог: input() ждёт, пока пользователь нажмёт Enter
name = input("Как тебя зовут? ")
print("Привет,", name)

# 2. Неблокирующий приём: проверяем, есть ли символ, и спокойно мигаем
poll = select.poll()
poll.register(sys.stdin, select.POLLIN)

last = time.ticks_ms()
print("Печатай строки, Enter отправляет. Светодиод мигает сам.")
while True:
    if poll.poll(0):                        # 0 --- не ждать
        line = sys.stdin.readline().strip()
        print("получено:", line)
    if time.ticks_diff(time.ticks_ms(), last) >= 500:
        last = time.ticks_ms()
        led.toggle()
```

Листинг 7.6: CircuitPython: то же

```
import time
import board
import digitalio
import supervisor

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

```

```
# 1. Простой диалог: input() ждёт, пока пользователь нажмёт Enter
name = input("Как тебя зовут? ")
print("Привет,", name)

# 2. Неблокирующий приём: input() вызываем, только когда уже есть данные
print("Печатай строки, Enter отправляет. Светодиод мигает сам.")
last = time.monotonic()
while True:
    if supervisor.runtime.serial_bytes_available:
        line = input().strip()
        print("получено:", line)
    if time.monotonic() - last >= 0.5:
        last = time.monotonic()
        led.value = not led.value
```

В **MicroPython** для этого служат `select.poll()` и `sys.stdin`, в **CircuitPython** — свойство `supervisor.runtime.serial_bytes_available`. А в **CircuitPython** есть ещё одна возможность: можно включить **второй** USB-порт для данных и оставить консоль свободной для отладочных сообщений. Включение делается в файле `boot.py`, который выполняется до подключения USB.

Листинг 7.7: **CircuitPython**: `boot.py` — включаем второй USB-порт

```
# boot.py --- выполняется один раз при включении, до подключения USB к компьютеру.
# После изменения нужно отключить и снова подключить плату (или нажать Reset).
import usb_cdc

# console=True  --- обычная консоль REPL (первый COM-порт)
# data=True     --- второй COM-порт, свободный для ваших данных
usb_cdc.enable(console=True, data=True)
```

Листинг 7.8: **CircuitPython**: обмен данными через `usb_cdc.data`

```
# code.py --- обмен данными через второй USB-порт, консоль остаётся свободной
import time
import usb_cdc

ser = usb_cdc.data                  # None, если в boot.py data=True не включили
if ser is None:
    raise RuntimeError("Включите usb_cdc.data в boot.py")
ser.timeout = 0                     # не блокировать чтение

counter = 0
while True:
    ser.write("count={}\n".format(counter).encode())   # данные в канал данных
    counter += 1
    n = ser.in_waiting
    if n:
        command = ser.read(n)
        print("с компьютера пришло:", command)         # а это --- в консоль REPL
    time.sleep(1)
```

После перезапуска платы компьютер увидит два COM-порта: первый — консоль, второй — ваши данные. В стандартной сборке **MicroPython** для Pico второго USB-порта нет: данные и консоль делят один канал, поэтому для отдельного канала используют UART.

7.11 Практика: разбор предложения GPS

GPS-модули (например, NEO-6M) каждую секунду шлют по UART текстовые предложения в формате **NMEA 0183**. Скорость по умолчанию — обычно 9600 бод. Строка начинается с $, поля разделены запятыми, а после звёздочки стоит контрольная сумма — XOR (исключающее ИЛИ) кодов всех символов между $ и *.

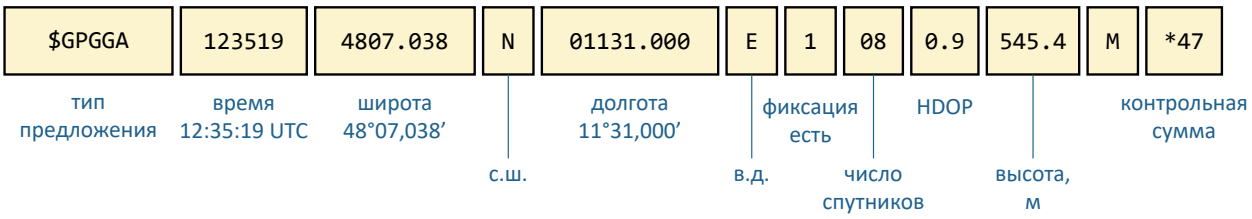


Рис. 7.9: Строение предложения GGA: поля разделены запятыми, в конце — контрольная сумма

Координаты записаны в необычном виде `ddmm.mmmm`: сначала градусы, потом минуты. Чтобы получить градусы с дробной частью, минуты делят на 60: $48 + 7{,}038/60 = 48{,}1173°$. Южная широта и западная долгота получают знак минус. Программа ниже проверяет сумму, разбирает предложение и умеет работать на встроенном образце, поэтому проверять логику можно без GPS-модуля.

Листинг 7.9: MicroPython: разбор предложения GGA

```
from machine import UART, Pin

# GPS-модуль на UART1: TX модуля -> GP5 (RX платы), RX модуля -> GP4 (TX платы)
gps = UART(1, baudrate=9600, tx=Pin(4), rx=Pin(5), timeout=1000)

def checksum_ok(sentence):
    """Контрольная сумма NMEA: XOR всех символов между '$' и '*'."""
    if not sentence.startswith("$") or "*" not in sentence:
        return False
    body, _, tail = sentence[1:].partition("*")
    calc = 0
    for ch in body:
        calc ^= ord(ch)
    try:
        return calc == int(tail[:2], 16)
    except ValueError:
        return False

def to_degrees(value, hemisphere):
    """Переводит ddmm.mmmm (градусы и минуты) в градусы с дробной частью."""
    dot = value.index(".")
    degrees = int(value[:dot - 2])
    minutes = float(value[dot - 2:])
    result = degrees + minutes / 60
    if hemisphere in ("S", "W"):     # юг и запад --- отрицательные числа
        result = -result
    return result

def parse_gga(sentence):
    """Из предложения GGA достаёт время, координаты, число спутников, высоту."""
    f = sentence.split("*")[0].split(",")
    if f[0][3:] != "GGA" or f[6] in ("", "0"):   # f[6] --- качество фиксации
        return None                              # положение ещё не определено
    return {
```

```
        "time": f[1],
        "lat": to_degrees(f[2], f[3]),
        "lon": to_degrees(f[4], f[5]),
        "sats": int(f[7]),
        "alt": float(f[9]),
    }

# Проверка без GPS: известный пример из документации NMEA
TEST = "$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47"
print("контрольная сумма:", checksum_ok(TEST))
print(parse_gga(TEST))

while True:
    raw = gps.readline()
    if not raw:
        continue
    try:
        text = raw.decode().strip()
    except UnicodeError:
        continue                        # мусор в линии --- пропускаем
    if text[3:6] == "GGA" and checksum_ok(text):
        fix = parse_gga(text)
        if fix:
            print("%.5f, %.5f, спутников: %d" % (fix["lat"], fix["lon"], fix["sats"]))
```

Листинг 7.10: CircuitPython: тот же разбор

```
import board
import busio

# GPS-модуль на GP4 (TX платы) и GP5 (RX платы); timeout в секундах
gps = busio.UART(board.GP4, board.GP5, baudrate=9600, timeout=1)

def checksum_ok(sentence):
    """Контрольная сумма NMEA: XOR всех символов между '$' и '*'."""
    if not sentence.startswith("$") or "*" not in sentence:
        return False
    body, _, tail = sentence[1:].partition("*")
    calc = 0
    for ch in body:
        calc ^= ord(ch)
    try:
        return calc == int(tail[:2], 16)
    except ValueError:
        return False

def to_degrees(value, hemisphere):
    """Переводит ddmm.mmmm (градусы и минуты) в градусы с дробной частью."""
    dot = value.index(".")
    degrees = int(value[:dot - 2])
    minutes = float(value[dot - 2:])
    result = degrees + minutes / 60
    if hemisphere in ("S", "W"):
        result = -result
    return result

def parse_gga(sentence):
    f = sentence.split("*")[0].split(",")
```

```
    if f[0][3:] != "GGA" or f[6] in ("", "0"):
        return None
    return {
        "time": f[1],
        "lat": to_degrees(f[2], f[3]),
        "lon": to_degrees(f[4], f[5]),
        "sats": int(f[7]),
        "alt": float(f[9]),
    }

TEST = "$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47"
print("контрольная сумма:", checksum_ok(TEST))
print(parse_gga(TEST))

while True:
    raw = gps.readline()              # bytes или None по таймауту
    if not raw:
        continue
    try:
        text = raw.decode().strip()
    except UnicodeError:
        continue
    if text[3:6] == "GGA" and checksum_ok(text):
        fix = parse_gga(text)
        if fix:
            print("%.5f, %.5f, спутников: %d" % (fix["lat"], fix["lon"], fix["sats"]))
```

В **CircuitPython** есть готовая библиотека `adafruit_gps`, которая делает всё это за вас; но разобрать предложение руками полезно один раз, чтобы понимать, как устроен «настоящий» протокол. Для **MicroPython** популярна библиотека `micropyGPS`. Антенну GPS нужно выносить к окну: в помещении сигнала часто нет, и поле «качество фиксации» остаётся равным нулю.

7.12 Bluetooth-модули HC-05 и HC-06

Классический Bluetooth-модуль HC-05 или HC-06 — это по сути «беспроводной UART»: всё, что плата отправляет в UART, модуль передаёт по радио на телефон или компьютер, и наоборот. В телефоне подойдёт любое приложение «Bluetooth Serial Terminal». Модуль питается от $5\,\text{V}$ (вывод VBUS платы), а логические выводы TXD и RXD у него $3{,}3\,\text{V}$ — с Pico он дружит без дополнительных деталей. По умолчанию модули работают на 9600 бод, 8N1.

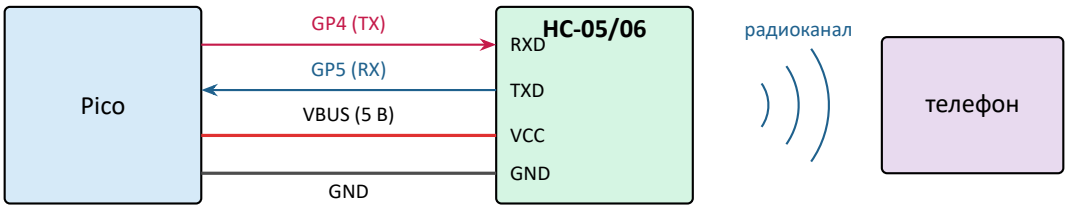


Рис. 7.10: Bluetooth-модуль как беспроводное продолжение UART

Листинг 7.11: MicroPython: управление светодиодом с телефона по символам `1` и `0`

```
from machine import UART, Pin

led = Pin("LED", Pin.OUT)
# HC-05/HC-06 по умолчанию работают на 9600 бод (HC-06 --- 9600, 8N1)
bt = UART(1, 9600, tx=Pin(4), rx=Pin(5))

while True:
    if bt.any():
        c = bt.read(1)                # один байт: управляющий символ
        if c == b"1":
            led.value(1)
            bt.write(b"LED ON\r\n")
        elif c == b"0":
            led.value(0)
            bt.write(b"LED OFF\r\n")
```

Листинг 7.12: CircuitPython: то же

```
import board
import busio
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT
# timeout=0: read() сразу возвращает None, если байтов нет
bt = busio.UART(board.GP4, board.GP5, baudrate=9600, timeout=0)

while True:
    c = bt.read(1)                    # один байт или None
    if c == b"1":
        led.value = True
        bt.write(b"LED ON\r\n")
    elif c == b"0":
        led.value = False
        bt.write(b"LED OFF\r\n")
```

Pico W и Pico 2 W имеют собственный Bluetooth (урок 19), поэтому внешний HC-05 нужен прежде всего для обычной Pico или для работы со старыми приложениями «Bluetooth Serial».

Это интересно

Первый UART-терминал был электромеханическим: телетайп Teletype 33 передавал буквы 10 символов в секунду. Кадр со стартовым и стоп-битом придумали ещё для него: стартовый бит запускал мотор приёмника, а стоп-бит давал ему время остановиться. Современная электроника унаследовала формат без изменений.

Если не работает

- **На экране «кракозябры»** — скорости передатчика и приёмника не совпадают (или не совпадает формат кадра: биты, чётность, стоп-биты). Проверьте `baudrate` с обеих сторон.
- **Ничего не приходит** — TX с TX и RX с RX. Нужно перекрестить: TX одного — к RX другого. Проверьте и общий GND.

- **Приходят лишние или потерянные байты** — программа слишком долго занята и не успевает читать: уберите длинные `sleep()`, читайте всё, что накопилось.
- **Ошибка при создании UART** — выбрана пара выводов не из таблицы или один блок использован дважды.
- **`decode()` выдаёт ошибку** — в линию попали посторонние байты (не та скорость, шум); ловите `UnicodeError`.
- **Плата нагревается или адаптер «горит»** — к RX подан 5 V или уровни RS-232. Отключите питание и проверьте уровни.

Итоги урока

- UART — асинхронный обмен по двум проводам: TX одного к RX другого, плюс общий GND; общих часов нет, скорость (бодрейт) договаривается заранее.
- Кадр 8N1: стартовый бит (0), 8 бит данных младшим вперёд, стоп-бит (1); на байт уходит 10 бит.
- Уровни Pico — 3,3 V; RS-232 и 5 V без преобразователя подключать нельзя.
- В MicroPython: `UART(id, baudrate, tx, rx)`, `any()`, `read()`, `readline()`, `write()`; в CircuitPython: `busio.UART`, `in_waiting`.
- Строки хорошо передавать как текстовый протокол: накопить байты до `\n`, потом разобрать `split()`.
- Контрольная сумма NMEA — XOR символов между `$` и `*`; USB-консоль и UART — разные каналы.

Проверь себя

1. Почему UART называют асинхронным и что в нём заменяет общий тактовый сигнал?
2. Нарисуйте кадр для байта `0x41` (латинская «A») в формате 8N1. Какой бит идёт первым?
3. Сколько времени займёт передача 100 байт на скорости 9600 бод?
4. Почему нельзя соединять TX с TX и подключать RX платы к выходу на 5 V?
5. Зачем при приёме строки накапливают байты в буфере, а не используют сразу `uart.read()`?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соедините GP0 и GP1 и запустите пример-эхо. Измените скорость на 9600 и убедитесь, что обмен работает.
2. ★ Разорвите перемычку и посмотрите, что напечатает программа. Какое сообщение вы увидели и почему?
3. ★★ Добавьте в протокол команду `BLINK n`, мигающую светодиодом `n` раз, и ответ `DONE`.
4. ★★ Подключите USB-UART адаптер и управляйте светодиодом из терминала на компьютере. Добавьте команду `HELP`, которая перечисляет все команды.

5. ★★★ Напишите функцию, проверяющую контрольную сумму произвольного NMEA-предложения, и протестируйте её на пяти строках — трёх верных и двух с одной испорченной буквой.
6. ★★★★ Соедините две платы (TX↔RX, GND) и сделайте «чат»: сообщения, набранные на одной плате в консоли, выводятся на другой. Придумайте формат, который отличает приказ от обычного сообщения.

Урок 8. Шина I2C: OLED, датчики, часы

Цель	понять, как на двух проводах работает шина I2C, и подключить к плате дисплей, датчик и часы реального времени
Опирается на	урок 7: обмен байтами; урок 3: цифровые входы и подтяжка; шестнадцатеричные числа
Результат	на OLED-дисплее выводятся температура, влажность и давление с BME280; плата умеет искать устройства на шине и читать время из DS3231
Понадобится	OLED SSD1306 128×64 (I2C), датчик BME280 (или BMP280), модуль часов DS3231, макетная плата, провода

8.1 Идея урока коротко

I2C (читается «ай-ту-си») — шина, на которой *много* устройств общаются по *двум* проводам. Линия **SDA** несёт данные, линия **SCL** — тактовые импульсы. Каждое устройство имеет свой адрес, и ведущий (наша плата) обращается к нужному, как учитель, вызывающий ученика по фамилии. Так к одной паре выводов можно подключить экран, десяток датчиков и часы.

8.2 Как устроена шина

В отличие от UART, здесь есть общий такт: ведущий (*master*) сам формирует импульсы на SCL, и все ведомые (*slave*) читают биты по этим импульсам. Подключаться можно в любой точке шины: линии SDA и SCL просто обегают все устройства.

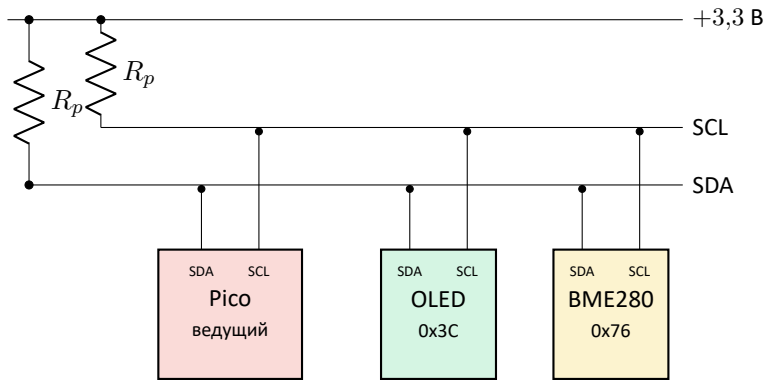
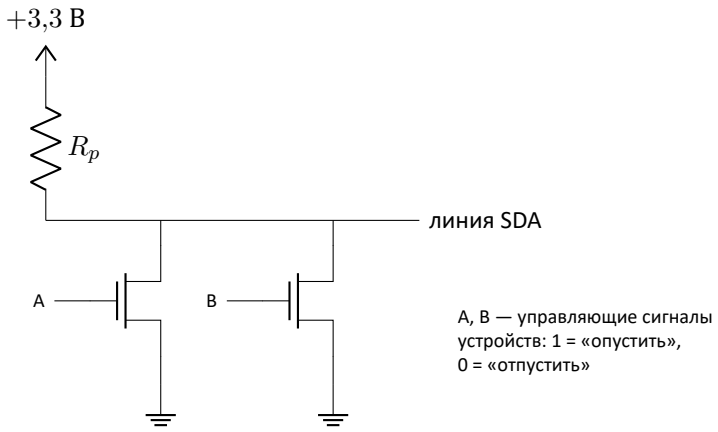


Рис. 8.1: Шина I2C: две общие линии и по одному подтягивающему резистору R_p на каждую линию. Земля общая и здесь не нарисована

Открытый сток и подтяжка

Если бы два устройства одновременно «толкали» линию — одно к 3,3 В, другое к земле, — возникло бы короткое замыкание. Поэтому на I2C устройства умеют только *опустить* линию к нулю или *отпустить* её. Такой выход называется **открытым стоком** (open-drain). Вверх линию поднимает резистор подтяжки R_p к питанию. Результат — «монтажное И»: линия равна единице, только если *все* отпустили её.



A	B	уровень на линии
отпустил	отпустил	1 (подтяжка)
опустил	отпустил	0
отпустил	опустил	0
опустил	опустил	0

Рис. 8.2: Выход с открытым стоком: каждый может только «опустить» линию, поднимает её резистор

Какой резистор выбрать

Линия имеет небольшую ёмкость C_b (провода, входы микросхем, обычно 50–200 пФ). Резистор и ёмкость образуют RC-цепь, и фронт нарастает не мгновенно: $U(t) = U_{\text{пит}}\,(1 - e^{-t/R_pC_b})$. Время подъёма от 30 % до 70 % примерно равно $t_r \approx 0{,}85\,R_pC_b$. Для частоты 100 кГц оно не должно превышать 1000 нс, для 400 кГц — 300 нс.

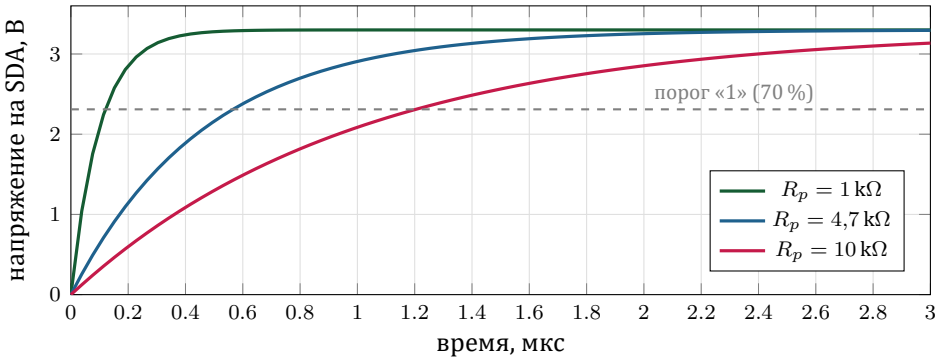


Рис. 8.3: Нарастание напряжения на линии при $C_b = 100\,\text{pF}$: чем больше сопротивление подтяжки, тем медленнее фронт

Но слишком маленькое сопротивление тоже вредно: устройство, опуская линию, должно «перетянуть» резистор. По стандарту I2C ведомый обязан выдерживать ток $I_{\text{OL}} = 3\,\text{mA}$ при напря-

жении не выше 0,4 В, поэтому

$$R_p \geq \frac{U_{\text{пит}} - 0{,}4}{3\ \text{мА}} = \frac{3{,}3 - 0{,}4}{0{,}003} \approx 970\,\Omega.$$

Итого для 3,3 В и обычных проводов подходят $2{,}2$ – $4{,}7\,\text{k}\Omega$. На большинстве готовых модулей (OLED, BME280, DS3231) подтяжки **уже распаяны**, обычно по $10\,\text{k}\Omega$ (на DS3231 — $4{,}7\,\text{k}\Omega$). Когда на шину подключают несколько модулей, их подтяжки включаются параллельно: три модуля по $10\,\text{k}\Omega$ дадут $10/3 \approx 3{,}3\,\text{k}\Omega$. Это нормально, но десять модулей уже опустят сопротивление ниже допустимого.

Это интересно

Шину I2C придумала фирма Philips в 1982 году для телевизоров: нужно было соединить микросхемы внутри одного корпуса с минимумом проводов. Название расшифровывается как Inter-Integrated Circuit. Фирма Atmel позже назвала то же самое TWI (Two Wire Interface), чтобы не платить за торговую марку, поэтому в документации к AVR вы встретите именно это название.

8.3 Один обмен: старт, адрес, подтверждение, стоп

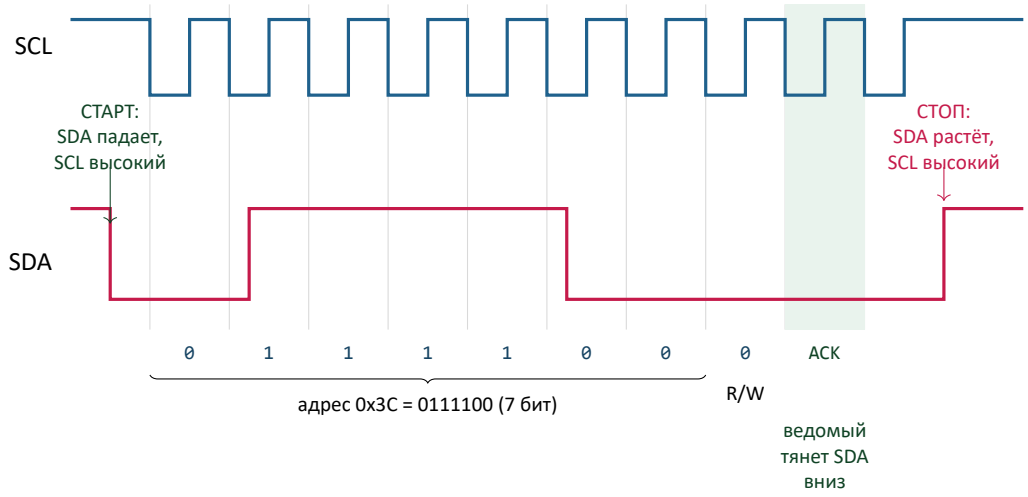


Рис. 8.4: Начало обмена: старт, 7-битный адрес и бит записи, подтверждение ACK, стоп

Правила на рисунке легко запомнить:

- Данные на SDA *меняются, пока SCL низкий*, и *держатся неизменными, пока SCL высокий*: именно в этот момент приёмник считывает бит.
- **Старт** — единственное место, где SDA падает при высоком SCL. **Стоп** — единственное, где SDA растёт при высоком SCL. Так передатчик отличает служебные сигналы от данных.
- Сначала идут 7 бит **адреса** (старшим вперёд), затем бит направления: `0` — «я запишу», `1` — «я прочитаю».
- После каждого байта наступает девятый такт: приёмник **подтверждает** приём, опуская SDA (ACK). Если никто не опустил линию (NACK), значит, устройства с таким адресом на шине нет, либо оно занято.

Частота SCL: 100 кГц (стандартный режим), 400 кГц (быстрый режим), до 1 МГц (Fast-mode Plus). Один байт вместе с подтверждением занимает 9 тактов, поэтому на 400 кГц передаётся около 44 000 байт/с. Устройство, которое не успевает, может удержать SCL низким (*растяжка такта*, clock stretching): ведущий обязан подождать.

8.4 Выводы I2C в Pico

В RP2040 и RP2350 два аппаратных блока I2C. Каждому доступно несколько пар выводов. SDA и SCL *одного* блока образуют пару: SDA на выводе с чётным «номером в паре», SCL — с нечётным.

Блок	SDA	SCL
I2C0	GP0, GP4, GP8, GP12, GP16, GP20	GP1, GP5, GP9, GP13, GP17, GP21
I2C1	GP2, GP6, GP10, GP14, GP18, GP22, GP26	GP3, GP7, GP11, GP15, GP19, GP27

Таблица 8.1: Выводы I2C на Pico. Пара SDA/SCL должна принадлежать одному блоку

В этом уроке используем **I2C0**: SDA на GP4, SCL на GP5. Всегда указывайте выводы явно: умолчания у MicroPython и CircuitPython различаются, а на разных платах семейства бывают разными.

8.5 Сначала ищем устройства: scan()

Первое, что делают с новым модулем, — проверяют, виден ли он на шине. Ведущий по очереди выставляет на шину адреса от `0x08` до `0x77` (адреса `0x00–0x07` и `0x78–0x7F` зарезервированы) и смотрит, ответил ли кто-нибудь подтверждением ACK.

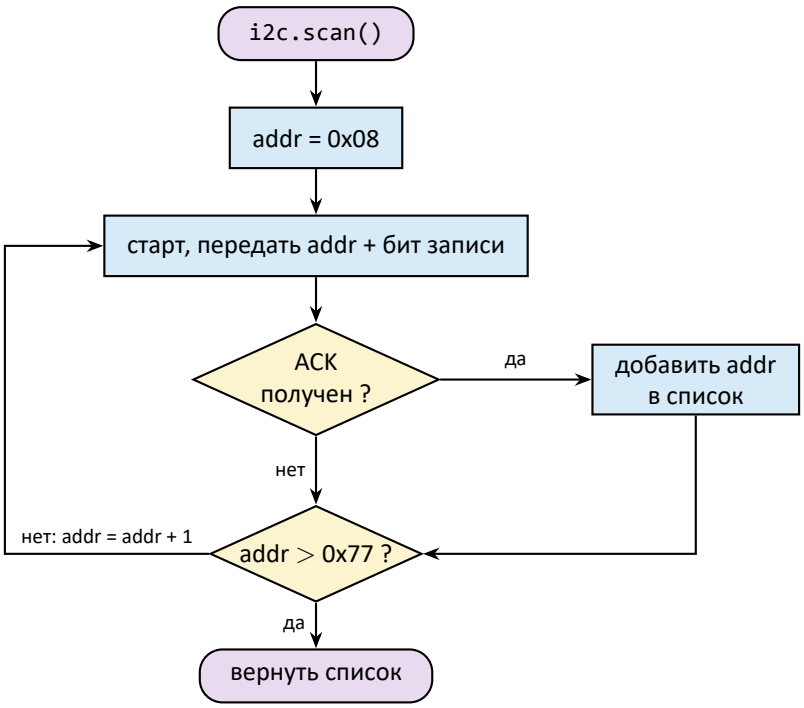


Рис. 8.5: Как работает `scan()`: перебор всех допустимых адресов

Листинг 8.1: MicroPython: сканирование шины

```
from machine import I2C, Pin

# I2C0: SDA = GP4, SCL = GP5, частота 400 кГц
i2c = I2C(0, sda=Pin(4), scl=Pin(5), freq=400000)

found = i2c.scan()                  # список 7-битных адресов, ответивших ACK
print("Найдено устройств:", len(found))
for addr in found:
    print("  адрес 0x%02X (%d)" % (addr, addr))
```

Листинг 8.2: CircuitPython: то же

```
import board
import busio

# ВНИМАНИЕ: порядок аргументов --- сначала SCL, потом SDA
i2c = busio.I2C(board.GP5, board.GP4, frequency=400000)

while not i2c.try_lock():            # шину нужно «захватить» перед работой
    pass
try:
    found = i2c.scan()
finally:
    i2c.unlock()                     # и обязательно отпустить

print("Найдено устройств:", len(found))
for addr in found:
    print("  адрес 0x%02X (%d)" % (addr, addr))
```

Различие в CircuitPython: шину нужно *захватить* (`try_lock()`) перед использованием и *освободить* (`unlock()`), потому что шиной могут пользоваться несколько библиотек сразу. Готовые драйверы (`adafruit_ssd1306` и др.) делают это сами, а при работе напрямую захват нужно писать вручную. Ещё одна ловушка: конструктор `busio.I2C` принимает **сначала SCL, потом SDA**, в MicroPython — по именам.

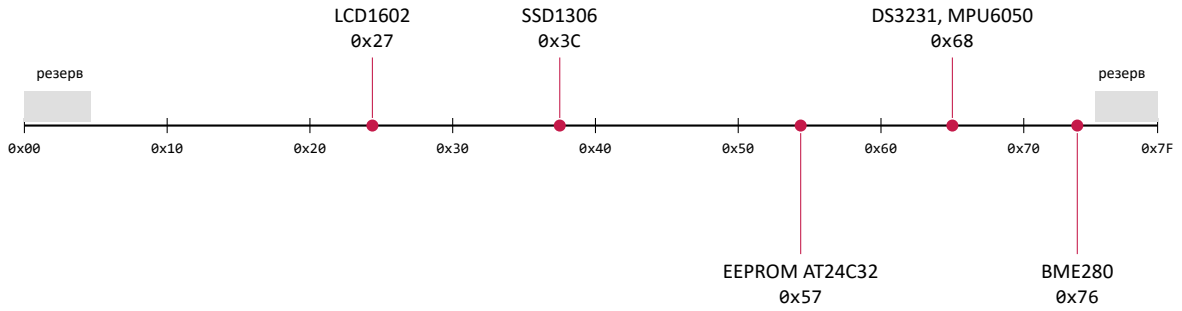


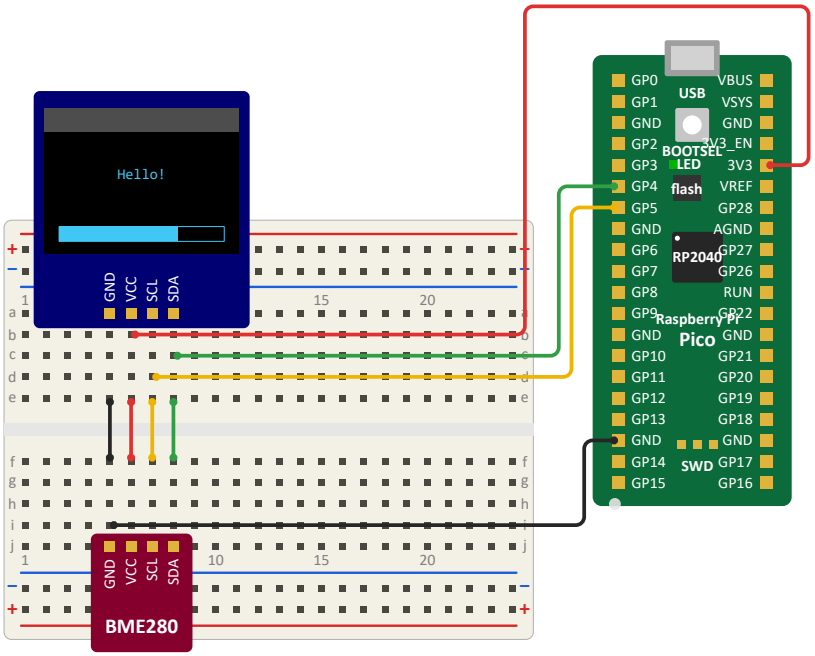
Рис. 8.6: Типичные адреса популярных I2C-устройств. Точное значение смотрите в описании модуля и проверяйте `scan()`

Совет

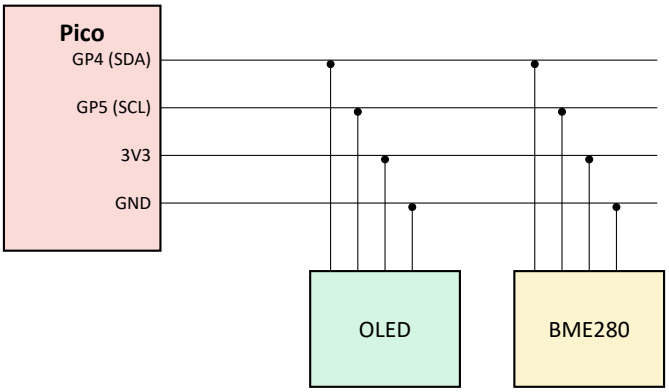
Если два устройства с одинаковыми адресами подключить на одну шину, они будут отвечать одновременно, и обмен сломается. У многих модулей адрес меняется перемычкой (OLED: `0x3C`/`0x3D`, BME280: `0x76`/`0x77`, MPU6050: `0x68`/`0x69`). Иначе подключайте вторую копию ко второму блоку I2C1.

8.6 Монтажная схема: OLED и BME280 на общей шине

Подключим сразу два модуля параллельно: экран выводит, датчик измеряет. Выводы у них одинаковые (питание, земля, SDA, SCL), поэтому через макетную плату мы просто соединяем одноимённые выводы. В схеме OLED вставлен в верхнюю половину платы, датчик — в нижнюю, а короткие перемычки через канавку объединяют их линии. Порядок выводов на вашем модуле может отличаться — всегда сверяйте с подписью на плате.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 8.7: OLED-дисплей и датчик BME280 на общей шине I2C0 (SDA — GP4, SCL — GP5)

Если после включения экран пустой, а `scan()` находит оба адреса, то схема собрана верно, и дело в программе.

8.7 Дисплей SSD1306

Популярный дисплей на 0,96 дюйма с разрешением 128×64 (или 128×32) пикселей — монохромный OLED: каждый пиксель либо светится, либо нет. Микросхема SSD1306 принимает данные по I2C. Изображение удобно рисовать не прямо на экране, а в **буфере** в оперативной памяти: 128×64 пикселей по одному биту на пиксель дают $128 \cdot 64/8 = 1024$ байта. Рисуем в буфере сколько угодно, а команда `show()` отправляет весь буфер на экран сразу. Так изображение не мигает.

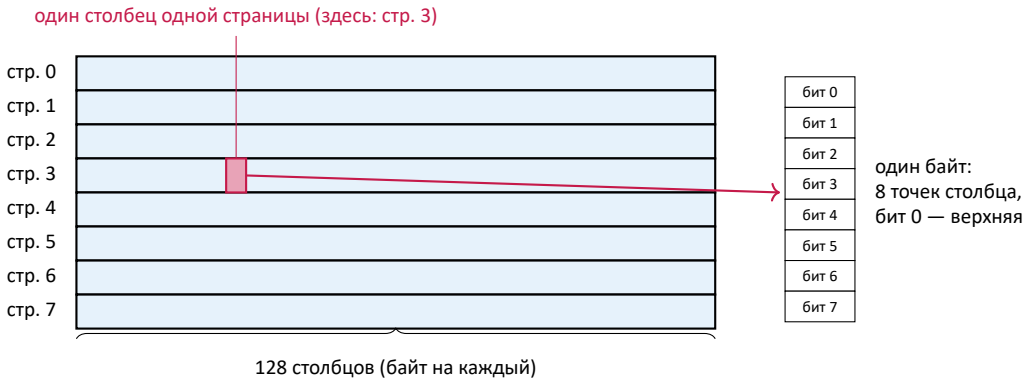


Рис. 8.8: Память SSD1306: 8 «страниц» высотой по 8 пикселей; один байт управляет вертикальной полоской из 8 точек

MicroPython включает в себя модуль `framebuf`, поэтому драйвер `ssd1306` (из репозитория micropython-lib) очень короткий. Установить его можно прямо из Thonny (пункт «Управление пакетами») или командами `import mip; mip.install("ssd1306")` — для Pico W/2 W, у которых есть Wi-Fi. На обычной Pico скопируйте файл `ssd1306.py` на плату вручную. В **CircuitPython** ставится библиотека `adafruit_ssd1306` (проще всего утилитой `circup`); для неё на диске `CIRCUITPY` должен лежать ещё файл шрифта `font5x8.bin`.

Листинг 8.3: **MicroPython**: текст, рамка, линии и пиксели

```
from machine import I2C, Pin
import ssd1306                          # установка: import mip; mip.install("ssd1306")

i2c = I2C(0, sda=Pin(4), scl=Pin(5), freq=400000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c)     # адрес по умолчанию 0x3C

oled.fill(0)                            # очистить буфер (0 --- чёрный)
oled.text("Hello, Pico!", 0, 0)         # текст 8x8 пикселей: x, y
oled.text("I2C OLED", 0, 12)
oled.rect(0, 26, 128, 38, 1)            # рамка: x, y, ширина, высота, цвет
oled.fill_rect(4, 30, 56, 12, 1)        # закрашенный прямоугольник
oled.text("OK", 8, 32, 0)               # чёрный текст на белом фоне
oled.line(0, 63, 127, 26, 1)            # линия по двум точкам
oled.pixel(100, 50, 1)                  # одна точка
oled.show()                             # только теперь буфер уходит на экран
```

Листинг 8.4: **CircuitPython**: то же

```
import board
import busio
import adafruit_ssd1306                 # circup install adafruit_ssd1306
# Нужен ещё файл font5x8.bin в корне диска CIRCUITPY (есть в архиве примеров библиотеки)

i2c = busio.I2C(board.GP5, board.GP4, frequency=400000)
oled = adafruit_ssd1306.SSD1306_I2C(128, 64, i2c, addr=0x3C)

oled.fill(0)
oled.text("Hello, Pico!", 0, 0, 1)   # здесь цвет (1) указывать обязательно
oled.text("I2C OLED", 0, 12, 1)
oled.rect(0, 26, 128, 38, 1)
oled.fill_rect(4, 30, 56, 12, 1)
oled.text("OK", 8, 32, 0)
oled.line(0, 63, 127, 26, 1)
```

```
oled.pixel(100, 50, 1)
oled.show()
```

Координаты экрана: начало в левом верхнем углу, x растёт вправо, y — *вниз*. Встроенный шрифт имеет размер 8×8, поэтому на экране помещается 16 символов в строке и 8 строк. Кириллицы и знака градуса в нём нет: пишите латиницей (`C`, `mmHg`) или используйте сторонние шрифты. Некоторые дисплеи на 1,3 дюйма построены на другой микросхеме (SH1106) и с драйвером SSD1306 показывают «сдвинутую» картинку — нужен свой драйвер.

8.8 Датчик BME280: температура, влажность, давление

BME280 измеряет сразу три величины и работает по I2C на адресах `0x76` или `0x77`. Хорошая новость: он хранит в себе калибровочные коэффициенты, и драйвер переводит «сырые» значения в градусы, проценты и паскали. Похожий датчик BMP280 **не измеряет влажность**: на плате обе микросхемы выглядят почти одинаково. Различить их можно, прочитав регистр идентификатора `0xD0`: у BME280 в нём `0x60`, у BMP280 — `0x58`.
Давление зависит от высоты: чем выше мы поднимаемся, тем тоньше столб воздуха над нами. На уровне моря давление около 1013 гПа (760 мм рт. ст.), и каждые 8 м подъёма уменьшают его примерно на 1 гПа. Из этого получается **барометрическая формула** для оценки высоты:

$$h \approx 44\,330 \cdot \left[1 - \left(\frac{p}{p_0}\right)^{1/5{,}255}\right] \text{ м},$$

где p_0 — давление на уровне моря (1013,25 гПа в стандартной атмосфере), а p — измеренное.

Листинг 8.5: MicroPython: показания BME280 на экране (драйвер `bme280_float.py`)

```
import time
from machine import I2C, Pin
import ssd1306
import bme280_float as bme280        # файл bme280_float.py из проекта robert-hh/BME280

i2c = I2C(0, sda=Pin(4), scl=Pin(5), freq=400000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c)
sensor = bme280.BME280(i2c=i2c, address=0x76)    # у некоторых модулей адрес 0x77

while True:
    t, p, h = sensor.read_compensated_data()     # °C, паскали, проценты
    mm = p / 133.322                              # давление в мм рт. ст.
    oled.fill(0)
    oled.text("T: %.1f C" % t, 0, 4)              # символ градуса в шрифте 8x8 нет
    oled.text("H: %.0f %%" % h, 0, 20)
    oled.text("P: %.0f mmHg" % mm, 0, 36)
    oled.show()
    print("%.1f C, %.0f %%, %.0f hPa" % (t, h, p / 100))
    time.sleep(2)
```

Листинг 8.6: CircuitPython: то же через `adafruit_bme280`

```
import time
import board
import busio
import adafruit_ssd1306
from adafruit_bme280 import basic as adafruit_bme280   # circup install adafruit_bme280

i2c = busio.I2C(board.GP5, board.GP4, frequency=400000)    # одна шина на оба прибора
oled = adafruit_ssd1306.SSD1306_I2C(128, 64, i2c, addr=0x3C)
sensor = adafruit_bme280.Adafruit_BME280_I2C(i2c, address=0x76)

while True:
    t = sensor.temperature                # °C
    h = sensor.relative_humidity          # %
    p = sensor.pressure                   # гектопаскали (гПа)
    oled.fill(0)
    oled.text("T: %.1f C" % t, 0, 4, 1)
    oled.text("H: %.0f %%" % h, 0, 20, 1)
    oled.text("P: %.0f mmHg" % (p * 0.75006), 0, 36, 1)
    oled.show()
    print("%.1f C, %.0f %%, %.0f hPa" % (t, h, p))
    time.sleep(2)
```

В примерах *одна* шина обслуживает и экран, и датчик: мы создаём один объект `I2C` и передаём его в оба драйвера. Создавать второй объект на тех же выводах не нужно.

8.9 Учимся программировать: шестнадцатеричные числа и биты

Адреса I2C принято писать в шестнадцатеричной системе: `0x3C`. В ней 16 цифр (0–9 и A–F), и каждая соответствует ровно четырём битам: `0x3C` $=$ `0011 1100`$_2$ $=$ 60. Поэтому программисты любят «hex»: байт записывается двумя цифрами. Несколько операций с битами понадобятся в этом уроке:

- `x >> 4` — сдвиг вправо на 4 бита: `0x59 >> 4` даёт `5`;
- `x & 0x0F` — побитовое И с маской: оставляет младшие 4 бита; `0x59 & 0x0F` даёт `9`;
- `(a << 4) | b` — собрать байт из двух половинок: `(5 << 4) | 9` даёт `0x59`.

Для вывода пригодится формат `%02X`: он печатает число в шестнадцатеричном виде, всегда двумя цифрами.

8.10 Часы реального времени DS3231

Плата Pico знает время только с момента включения. Чтобы время «помнилось» между включениями, используют модуль часов реального времени DS3231: точный кварцевый генератор (погрешность около ± 1 минуты в год), литиевая батарейка CR2032 для питания при отключённой плате и встроенный термометр. Адрес — `0x68`. Время хранится в регистрах `0x00`–`0x06`: секунды, минуты, часы, день недели, число, месяц, год.
Особенность: числа лежат в регистрах в **двоично-десятичном коде** (BCD). В каждой половине байта — одна *десятичная* цифра, поэтому время 59 секунд хранится как `0x59`, а не как `0x3B`.

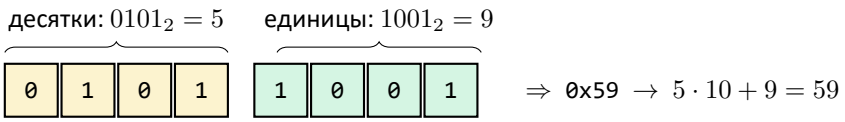


Рис. 8.9: Двоично-десятичный код (BCD): число 59 хранится в байте как две цифры, по четыре бита на каждую

Перевод в обе стороны — две короткие формулы: $d = (b \gg 4) \cdot 10 + (b \,\&\, \texttt{0x0F})$ и обратно $b = ((d \,//\, 10) \ll 4) \,|\, (d \bmod 10)$. Нужно также маскировать служебные биты: у регистра часов 24-часовой формат — биты 0–5, у месяца старший бит — признак века.

Листинг 8.7: MicroPython: чтение и запись регистров DS3231 вручную

```
import time
from machine import I2C, Pin

DS3231 = 0x68
i2c = I2C(0, sda=Pin(4), scl=Pin(5), freq=400000)

def bcd2dec(b):
    """Две десятичные цифры в одном байте: 0x59 -> 59."""
    return (b >> 4) * 10 + (b & 0x0F)

def dec2bcd(d):
    """59 -> 0x59."""
    return ((d // 10) << 4) | (d % 10)

def set_time(year, month, day, weekday, hour, minute, second):
    # регистры 0x00..0x06: секунды, минуты, часы, день недели, число, месяц, год
    data = bytes([dec2bcd(second), dec2bcd(minute), dec2bcd(hour),
                  dec2bcd(weekday), dec2bcd(day), dec2bcd(month),
                  dec2bcd(year - 2000)])
    i2c.writeto_mem(DS3231, 0x00, data)

def get_time():
    d = i2c.readfrom_mem(DS3231, 0x00, 7)      # читаем все семь регистров за раз
    second = bcd2dec(d[0] & 0x7F)
    minute = bcd2dec(d[1])
    hour = bcd2dec(d[2] & 0x3F)                # 24-часовой формат
    day = bcd2dec(d[4])
    month = bcd2dec(d[5] & 0x1F)               # старший бит --- признак века
    year = 2000 + bcd2dec(d[6])
    return year, month, day, hour, minute, second

def get_temperature():
    t = i2c.readfrom_mem(DS3231, 0x11, 2)      # температура в регистрах 0x11, 0x12
    raw = (t[0] << 8) | t[1]
    if raw & 0x8000:                           # отрицательное число (дополнительный код)
        raw -= 65536
    return (raw >> 6) * 0.25

# set_time(2026, 11, 5, 4, 12, 30, 0)        # раскомментируйте ОДИН раз, чтобы поставить часы

while True:
    y, mo, d, h, mi, s = get_time()
    print("%04d-%02d-%02d %02d:%02d:%02d  %.2f C" % (y, mo, d, h, mi, s, get_temperature()))
    time.sleep(1)
```

Листинг 8.8: CircuitPython: библиотека `adafruit_ds3231`

```
import time
import board
import busio
import adafruit_ds3231                # circup install adafruit_ds3231

i2c = busio.I2C(board.GP5, board.GP4)
rtc = adafruit_ds3231.DS3231(i2c)     # адрес 0x68; BCD библиотека переводит сама

# Однократно поставить время: (год, месяц, число, час, мин, сек, день недели, -1, -1)
# rtc.datetime = time.struct_time((2026, 11, 5, 12, 30, 0, 3, -1, -1))

while True:
    t = rtc.datetime
    print("%04d-%02d-%02d %02d:%02d:%02d  %.2f C" % (
        t.tm_year, t.tm_mon, t.tm_mday, t.tm_hour, t.tm_min, t.tm_sec, rtc.temperature))
    time.sleep(1)
```

Осторожно

Дешёвые модули DS3231 часто имеют на плате цепь подзарядки батарейки, рассчитанную на аккумулятор LIR2032. Если вставить обычную **неперезаряжаемую** CR2032, она может нагреться и повредиться. Используйте LIR2032 или отключите цепь зарядки (удалив резистор около батарейки).

8.11 Рисуем крупно: свой «шрифт» из framebuf

Встроенный шрифт мелкий. Увеличить его можно простым приёмом: нарисовать текст во временном буфере и перенести каждый его пиксель на экран квадратиком $k \times k$. Так получается крупный текст без сторонних шрифтов.

Листинг 8.9: MicroPython: крупные цифры

```
import time
import framebuf
from machine import I2C, Pin
import ssd1306

i2c = I2C(0, sda=Pin(4), scl=Pin(5), freq=400000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c)

def big_text(display, s, x, y, k=2):
    """Рисует строку s с увеличением в k раз (встроенный шрифт 8x8)."""
    w = 8 * len(s)
    buf = bytearray(w)                        # картинка w x 8, один бит на пиксель
    fb = framebuf.FrameBuffer(buf, w, 8, framebuf.MONO_VLSB)
    fb.text(s, 0, 0, 1)                       # рисуем мелкий текст во временном буфере
    for i in range(w):
        for j in range(8):
            if fb.pixel(i, j):                # каждую точку превращаем в квадрат k x k
                display.fill_rect(x + i * k, y + j * k, k, k, 1)

n = 0
while True:
    oled.fill(0)
```

```
    big_text(oled, "COUNT", 0, 0, 2)
    big_text(oled, "%05d" % n, 0, 24, 3)     # цифры 24x24 пикселя
    oled.show()
    n += 1
    time.sleep(0.5)
```

Листинг 8.10: CircuitPython: то же

```
import time
import board
import busio
import framebuf                          # встроен в CircuitPython
import adafruit_ssd1306

i2c = busio.I2C(board.GP5, board.GP4, frequency=400000)
oled = adafruit_ssd1306.SSD1306_I2C(128, 64, i2c)

def big_text(display, s, x, y, k=2):
    """Рисует строку s с увеличением в k раз (шрифт 8x8 берём из framebuf)."""
    w = 8 * len(s)
    buf = bytearray(w)
    fb = framebuf.FrameBuffer(buf, w, 8, framebuf.MVLSB)   # так называется формат в CP
    fb.text(s, 0, 0, 1)
    for i in range(w):
        for j in range(8):
            if fb.pixel(i, j):
                display.fill_rect(x + i * k, y + j * k, k, k, 1)

n = 0
while True:
    oled.fill(0)
    big_text(oled, "COUNT", 0, 0, 2)
    big_text(oled, "%05d" % n, 0, 24, 3)
    oled.show()
    n += 1
    time.sleep(0.5)
```

Название формата буфера различается: в MicroPython — `framebuf.MONO_VLSB`, в CircuitPython — `framebuf.MVLSB`. Для вывода шрифтов с кириллицей в CircuitPython применяют `displayio` вместе с `adafruit_display_text`, а в MicroPython — сторонние библиотеки вроде `writer` или `micropython-font-to-py`.

Если не работает

- **`scan()` возвращает пустой список** — не подключены питание или земля, перепутаны SDA и SCL, дисплей сидит на другой паре выводов, плохой контакт в макетной плате.
- **`OSError: [Errno 5] EIO`** (MicroPython) или **`ValueError: No I2C device at address`** (CircuitPython) — устройство не отвечает по указанному адресу. Запустите `scan()` и сверьте адрес.
- **Экран пустой, хотя адрес найден** — забыт `show()`, или в буфер рисуют цвет `0` (чёрный).
- **Шина работает, потом «зависает»** — слишком много модулей с подтяжками, провода слишком длинные для 400 кГц; уменьшите частоту до 100 кГц.
- **BME280 показывает влажность 0** — это BMP280: влажности у него нет.
- **Время сбрасывается после выключения** — в модуле нет или разряжена батарейка.

- **Одна шина — две «одинаковые» библиотеки** — создавайте объект `I2C` один раз и передавайте его всем драйверам.

Итоги урока

- I2C: две линии (SDA, SCL), много устройств, у каждого свой 7-битный адрес; такт формирует ведущий.
- Выход с открытым стоком: линии подтягивают резисторы R_p ($2{,}2 - 4{,}7\ \text{k}\Omega$); ведомые подтверждают приём битом ACK.
- Старт, адрес, R/W, ACK, данные, стоп; частота 100 или 400 кГц.
- **MicroPython**: `I2C(0, sda=Pin(4), scl=Pin(5), freq=400000)`, `scan()`; **CircuitPython**: `busio.I2C(scl, sda)`, `try_lock()`.
- SSD1306 рисуется в буфере и выводится командой `show()`; BME280 даёт температуру, влажность, давление; DS3231 хранит время в BCD.

Проверь себя

1. Зачем на линиях SDA и SCL нужны подтягивающие резисторы и почему выходы делают с открытым стоком?
2. Какие значения принимают SDA и SCL при старте, при передаче бита и при стопе?
3. Что произойдёт, если подключить к одной шине два устройства с одинаковым адресом?
4. Как записывается число 47 в BCD? Чему равно число `0x23` в обычной записи, а чему — в BCD?
5. Почему при рисовании на OLED нужно вызывать `show()`?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Запустите `scan()` с подключённым экраном и выпишите найденный адрес в десятичной и шестнадцатеричной записи.
2. ★ Выведите на OLED своё имя в рамке и прямоугольник, закрашенный наполовину.
3. ★★ Покажите на экране температуру и влажность, а при превышении 30 °C выведите надпись `HOT` инвертированным цветом (чёрным по белому).
4. ★★ Напишите функцию, которая переводит давление в гПа в высоту над уровнем моря по барометрической формуле, и выведите высоту на экран.
5. ★★★ Сделайте цифровые часы на DS3231 с крупными цифрами на экране; кнопкой переключайте показ времени и температуры.
6. ★★★★ Добавьте на шину третье устройство (например, MPU6050) и выведите на экран температуру, влажность и угол наклона. Проверьте, что `scan()` находит три адреса.

Урок 9. Шина SPI: SD-карта и дисплеи

Цель	разобраться, как устроена шина SPI с отдельными линиями данных и выбора устройства, и подключить к плате SD-карту, дисплей и сдвиговый регистр
Опирается на	урок 8: шина и адреса; урок 7: обмен байтами и скорость
Результат	на SD-карту записывается журнал измерений; восемь светодиодов управляются через 74HC595 по трём проводам; дисплей ST7789 заливается цветом
Понадобится	microSD-модуль с картой, микросхема 74HC595 и 8 светодиодов с резисторами, дисплей ST7789 (по желанию), провода

9.1 Идея урока коротко

SPI — шина для *быстрых* устройств: SD-карт, дисплеев, АЦП, радиомодулей. Она проще и быстрее I2C: данные идут в обе стороны по отдельным линиям, а выбор нужного устройства делает не адрес, а отдельный провод **CS** (Chip Select). Цена за скорость — лишние провода: на каждое новое устройство нужен ещё один провод CS.

9.2 Четыре провода и ведущий

Сигнал	Расшифровка	Что делает
SCK (SCLK)	Serial Clock	такт, его создаёт ведущий
MOSI (SDO/COPI)	Master Out, Slave In	данные от ведущего к ведомому
MISO (SDI/CIPO)	Master In, Slave Out	данные от ведомого к ведущему
CS (SS)	Chip Select	выбор устройства, активный уровень — **низкий**

Таблица 9.1: Линии SPI. В новой документации вместо слов master/slave всё чаще пишут controller/peripheral

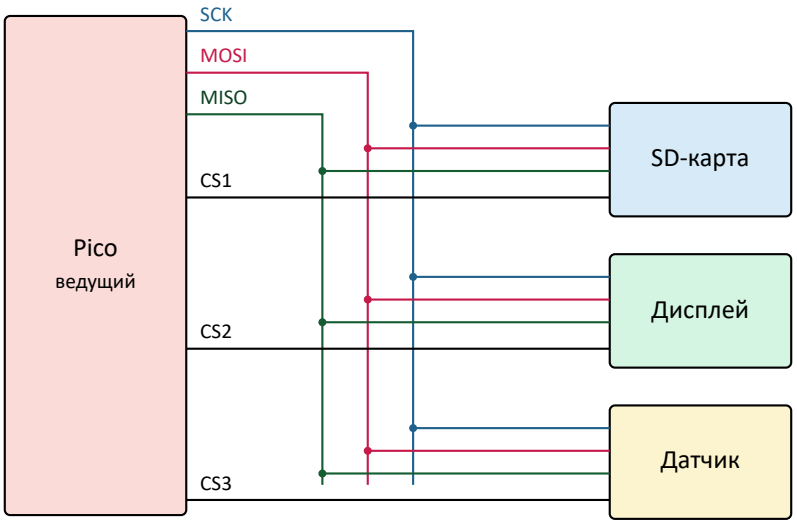


Рис. 9.1: Три общие линии (SCK, MOSI, MISO) и отдельная линия CS для каждого устройства; пересечения без точки — не соединения

Осторожно

Запомните главное: у ведомых *общие* SCK, MOSI и MISO, но *свои* CS. Пока на CS высокий уровень, устройство «спит» и отпускает линию MISO, поэтому несколько устройств не мешают друг другу.

9.3 Обмен как сдвиг

Внутри ведущего и ведомого находится по **сдвиговому регистру** на 8 бит. Выходы регистров соединены кольцом: старший бит ведущего поступает на MOSI в младший разряд ведомого, а старший бит ведомого — на MISO в младший разряд ведущего. С каждым тактом SCK оба регистра сдвигаются на один разряд. Через 8 тактов ведущий и ведомый *обменялись* байтами. Отсюда главное свойство SPI: **каждый переданный бит одновременно означает принятый бит**. Чтобы просто прочитать данные, ведущий отправляет «пустые» байты (обычно `0x00` или `0xFF`).

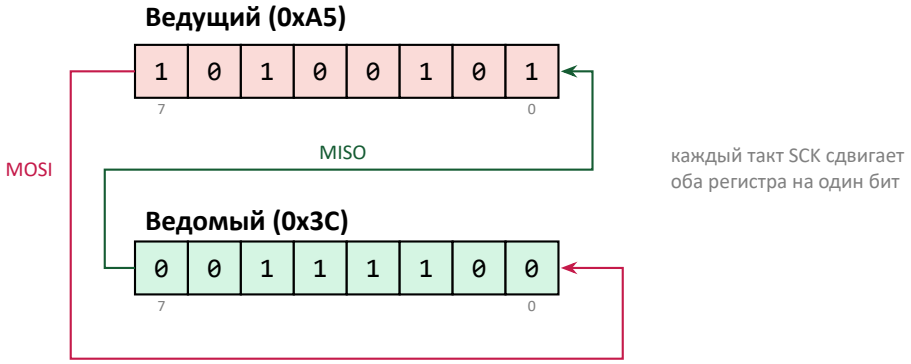


Рис. 9.2: Сдвиговые регистры ведущего и ведомого образуют кольцо: за 8 тактов байты меняются местами

9.4 Режимы SPI: полярность и фаза

Бит нужно *считывать* в момент, когда он уже установился на линии. Но есть два способа договориться о моменте считывания. **CPOL** (полярность) определяет уровень такта в покое: 0 — низкий, 1 — высокий. **CPHA** (фаза) определяет, по какому фронту считывают данные: 0 — по первому («ведущему»), 1 — по второму («хвостовому»). Получается четыре режима:

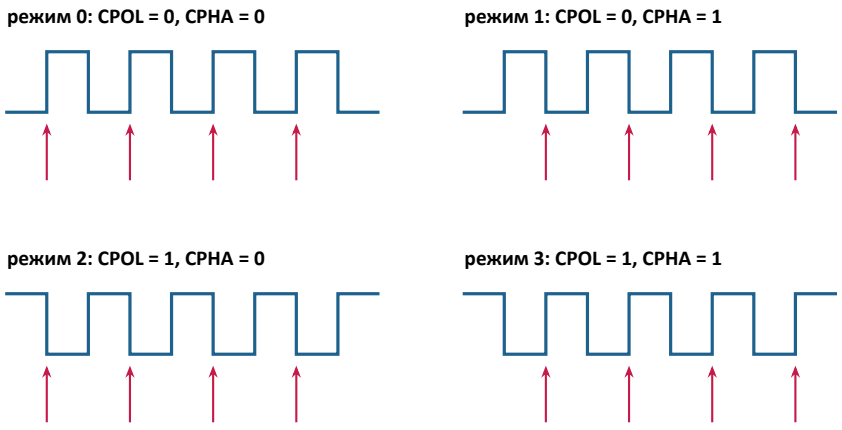


Рис. 9.3: Четыре режима SPI. Красные стрелки — моменты, когда принимающий читает бит данных. Режимы 0 и 3 встречаются чаще всего

Режим задаётся в документации устройства. У SD-карты, ST7789, 74HC595 и большинства датчиков он нулевой (или 3, что для них равнозначно). В MicroPython это параметры `polarity` и

`phase`, в **CircuitPython** — те же имена в `configure()`.

Как выглядит один байт

Ведущий опускает CS, затем подаёт 8 тактов SCK. В режиме 0 данные меняются по спадающему фронту такта, а считываются по нарастающему. Первым идёт *старший* бит (MSB first).

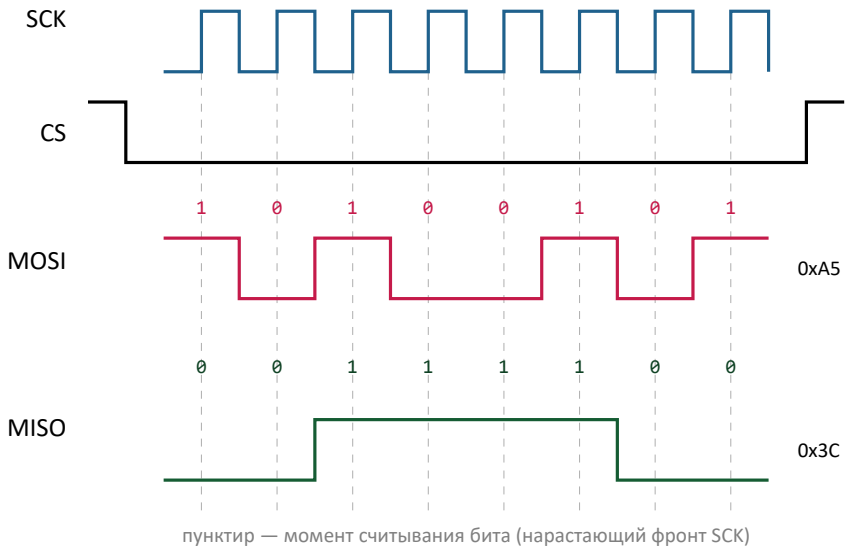


Рис. 9.4: Обмен одним байтом в режиме 0: ведущий отправляет `0xA5`, ведомый одновременно отвечает `0x3C`

9.5 Скорость и выводы в Pico

В SPI нет жёсткой скорости: частота SCK задаётся программно и ограничена возможностями устройства и качеством проводов. SD-карта при запуске работает на малой частоте (около 400 кГц), потом разгоняется до нескольких мегагерц; дисплеи и АЦП допускают 20–60 МГц. Блок SPI в RP2040 теоретически выдаёт до 62,5 МГц, но на макетной плате с длинными проводами уверенно работают единицы мегагерц.

Блок	MISO (RX)	CS (CSn)	SCK	MOSI (TX)
SPI0	GP0, GP4, GP16, GP20	GP1, GP5, GP17, GP21	GP2, GP6, GP18, GP22	GP3, GP7, GP19
SPI1	GP8, GP12, GP28	GP9, GP13	GP10, GP14, GP26	GP11, GP15, GP27

Таблица 9.2: Выводы аппаратных SPI на Raspberry Pi Pico. Вывод CS можно использовать и как обычный GPIO

Выводы CS в таблице нужны только для «режима CS от периферии»; на практике CS всегда делают обычным выходом GPIO, которым управляет программа: так проще подключать несколько устройств. **MicroPython** *не* управляет CS сам: `cs(0)` перед обменом и `cs(1)` после — обязанность программиста.

Действие	MicroPython	CircuitPython
Создание	`SPI(0, baudrate=..., sck=..., mosi=..., miso=...)`	`busio.SPI(clock, MOSI, MISO)`
Режим и скорость	параметры конструктора	`spi.configure(baudrate, polarity, p`
Захват шины	не нужен	`try_lock()` / `unlock()`
Отправить	`spi.write(buf)`	`spi.write(buf)`
Принять	`spi.read(n)`	`spi.readinto(buf)`
Обмен	`spi.write_readinto(out, in)`	`spi.write_readinto(out, in)`
Программная шина	`SoftSPI(...)`	`bitbangio.SPI(...)`

Таблица 9.3: SPI в двух языках

Для проверки блока SPI без внешней детали соедините GP3 (MOSI) и GP4 (MISO) проводом: всё отправленное сразу вернётся — «шлейф», как в уроке 7.

Листинг 9.1: MicroPython: SPI-«шлейф» между MOSI и MISO

```
from machine import SPI, Pin

# SPI0: SCK = GP2, MOSI = GP3, MISO = GP4. Соедините GP3 и GP4 перемычкой.
spi = SPI(0, baudrate=1000000, polarity=0, phase=0, bits=8,
          firstbit=SPI.MSB, sck=Pin(2), mosi=Pin(3), miso=Pin(4))
cs = Pin(5, Pin.OUT, value=1)        # CS неактивен = 1

out = bytearray(b"Hello")
back = bytearray(len(out))           # сюда придёт то, что «вернётся» по MISO

cs(0)                                # выбрали устройство
spi.write_readinto(out, back)        # SPI всегда передаёт и принимает одновременно
cs(1)                                # отпустили

print("отправлено:", bytes(out))
print("принято:   ", bytes(back))    # при перемычке MOSI-MISO совпадёт
```

Листинг 9.2: CircuitPython: то же

```
import board
import busio
import digitalio

# SCK = GP2, MOSI = GP3, MISO = GP4. Соедините GP3 и GP4 перемычкой.
spi = busio.SPI(board.GP2, MOSI=board.GP3, MISO=board.GP4)
cs = digitalio.DigitalInOut(board.GP5)
cs.direction = digitalio.Direction.OUTPUT
cs.value = True                      # CS неактивен = 1

while not spi.try_lock():            # как и I2C, шину нужно захватить
    pass
try:
    spi.configure(baudrate=1000000, polarity=0, phase=0, bits=8)
    out = bytearray(b"Hello")
    back = bytearray(len(out))
    cs.value = False                 # выбрали устройство
    spi.write_readinto(out, back)
    cs.value = True                  # отпустили
finally:
    spi.unlock()

```

```
print("отправлено:", bytes(out))
print("принято:   ", bytes(back))
```

9.6 Учимся программировать: биты, маски и bytearray

Для управления микросхемами SPI нужно уверенно обращаться с отдельными битами. Основные приёмы (число x, номер бита n):

- **установить** бит: `x |= 1 << n;`
- **сбросить** бит: `x &= ~(1 << n);`
- **переключить**: `x ^= 1 << n;`
- **проверить**: `(x >> n) & 1`.

Все они используют *маску* `1 << n` — число, в котором единица стоит только на нужном месте. Состояние восьми светодиодов или восьми кнопок удобно хранить в одном байте, а последовательность байтов — в `bytearray`: это изменяемая «строка из байтов», с которой работают `spi.write()` и `readinto()`.

9.7 Сдвиговый регистр 74HC595

Выводов у Pico не хватает всегда. Микросхема **74HC595** превращает три вывода в восемь: байт передаётся в неё по SPI, а затем импульсом «защёлки» RCLK выставляется на выходы Q0–Q7. Такие микросхемы можно соединять цепочкой: выход Q7’ первой идёт на вход SER второй, и три провода управляют уже 16, 24 и больше светодиодами.

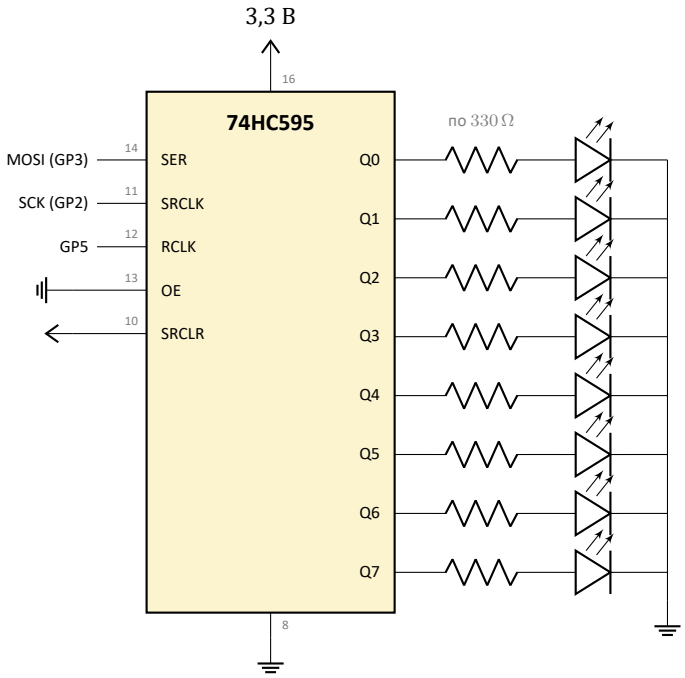


Рис. 9.5: Подключение 74HC595: байт приходит на SER по SPI, защёлка RCLK выставляет его на выходы Q0–Q7. OE подтянут к земле, SRCLR — к питанию

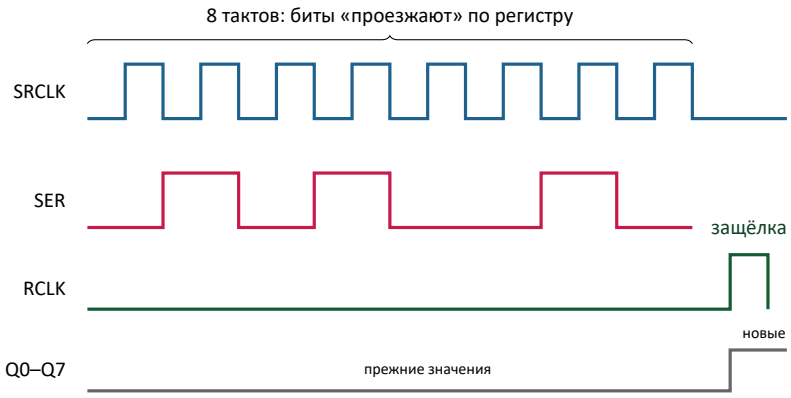


Рис. 9.6: Порядок работы: 8 тактов загружают байт в регистр, и только импульс RCLK переносит его на выходы

Схема собирается проще, чем кажется: SER соединяем с MOSI платы (GP3), SRCLK — с SCK (GP2), RCLK — с GP5 (это обычный вывод в роли CS). Вывод OE (разрешение выходов) подключаем к земле, SRCLR — к 3,3 В. Каждый светодиод подключаем через резистор $330\ \Omega$, как в уроке 2. Ток восьми светодиодов при выходах 74HC595 (до 6 mA на выход) безопасен.

Листинг 9.3: MicroPython: бегущий огонь через 74HC595 (аппаратный SPI)

```
import time
from machine import SPI, Pin

# 74HC595: SER(DS) <- GP3 (MOSI), SRCLK(SHCP) <- GP2 (SCK), RCLK(STCP) <- GP5
spi = SPI(0, baudrate=1000000, sck=Pin(2), mosi=Pin(3), miso=Pin(4))
latch = Pin(5, Pin.OUT, value=0)

def shift_out(value):
    spi.write(bytes([value]))        # 8 тактов SCK: байт уезжает в сдвиговый регистр
    latch(1)                         # импульс защёлки: регистр -> выходы Q0..Q7
    latch(0)

while True:
    for i in range(8):               # «бегущий огонь» вперёд
        shift_out(1 << i)
        time.sleep_ms(100)
    for i in range(6, 0, -1):        # и назад, без повтора крайних
        shift_out(1 << i)
        time.sleep_ms(100)
```

Листинг 9.4: CircuitPython: то же

```
import time
import board
import busio
import digitalio

# 74HC595: SER(DS) <- GP3 (MOSI), SRCLK(SHCP) <- GP2 (SCK), RCLK(STCP) <- GP5
spi = busio.SPI(board.GP2, MOSI=board.GP3)
latch = digitalio.DigitalInOut(board.GP5)
latch.direction = digitalio.Direction.OUTPUT

while not spi.try_lock():
    pass
spi.configure(baudrate=1000000)      # шину держим захваченной всю программу
```

```

def shift_out(value):
    spi.write(bytes([value]))
    latch.value = True              # импульс защёлки
    latch.value = False

while True:
    for i in range(8):
        shift_out(1 << i)
        time.sleep(0.1)
    for i in range(6, 0, -1):
        shift_out(1 << i)
        time.sleep(0.1)
```

Чтобы понять работу микросхемы «на пальцах», подключите те же выводы к *любым* GPIO и передавайте биты вручную: выставили бит на SER, подали импульс на SRCLK — и так восемь раз. Такой способ называют **bit-bang** (программная реализация шины). Он медленнее аппаратного, зато не привязан к определённым выводам.

Листинг 9.5: MicroPython: тот же регистр, но «вручную» (bit-bang)

```
import time
from machine import Pin

# Те же выводы, но «вручную» (bit-bang): никакого SPI-блока, работает на любых GPIO
data = Pin(3, Pin.OUT, value=0)      # SER
clock = Pin(2, Pin.OUT, value=0)     # SRCLK
latch = Pin(5, Pin.OUT, value=0)     # RCLK

def shift_out(value):
    for bit in range(7, -1, -1):     # старший бит первым (MSB first)
        data.value((value >> bit) & 1)   # выставили бит на линии данных
        clock.value(1)               # фронт такта: микросхема запомнила бит
        clock.value(0)
    latch.value(1)                   # защёлка: байт появляется на выходах
    latch.value(0)

n = 0
while True:
    shift_out(n)                     # светодиоды показывают двоичный счётчик
    n = (n + 1) & 0xFF
    time.sleep_ms(200)
```

Листинг 9.6: CircuitPython: то же

```
import time
import board
import digitalio

def make_output(pin):
    p = digitalio.DigitalInOut(pin)
    p.direction = digitalio.Direction.OUTPUT
    return p

data = make_output(board.GP3)        # SER
clock = make_output(board.GP2)       # SRCLK
latch = make_output(board.GP5)       # RCLK

```

```
def shift_out(value):
    for bit in range(7, -1, -1):      # старший бит первым
        data.value = bool((value >> bit) & 1)
        clock.value = True            # фронт такта
        clock.value = False
    latch.value = True                # защёлка
    latch.value = False

n = 0
while True:
    shift_out(n)                      # двоичный счётчик на светодиодах
    n = (n + 1) & 0xFF
    time.sleep(0.2)
```

Совет

Для светодиодных матриц, управляемых микросхемой MAX7219, действуют те же правила: три провода (DIN, CLK, CS) идут к MOSI, SCK и обычному GPIO, а дальше нужна только библиотека (`max7219.py` для **MicroPython**, `adafruit_max7219` для **CircuitPython**). Протокол — два байта (адрес регистра и значение), которые вы уже умеете отправлять.

9.8 SD-карта

SD-карту можно подключать по SPI — у неё есть специальный «SPI-режим» с минимальным числом выводов: SCK, MOSI, MISO и CS. Для этого продаются модули-переходники microSD на шесть выводов: GND, VCC, MISO, MOSI, SCK, CS (порядок бывает разным). Карта хранит данные в виде файловой системы FAT, которую понимают и компьютер, и плата — поэтому файл, записанный Pico, можно прочитать на ПК, вынув карту.

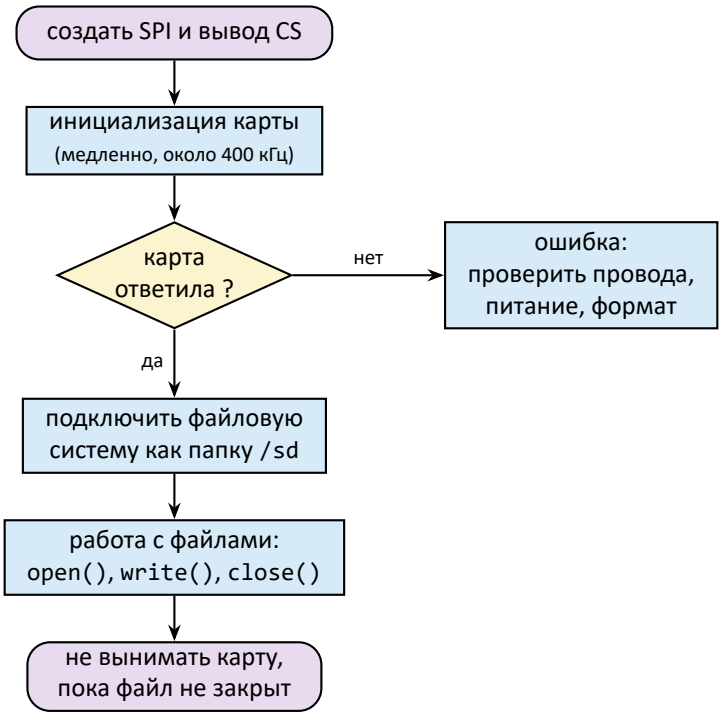
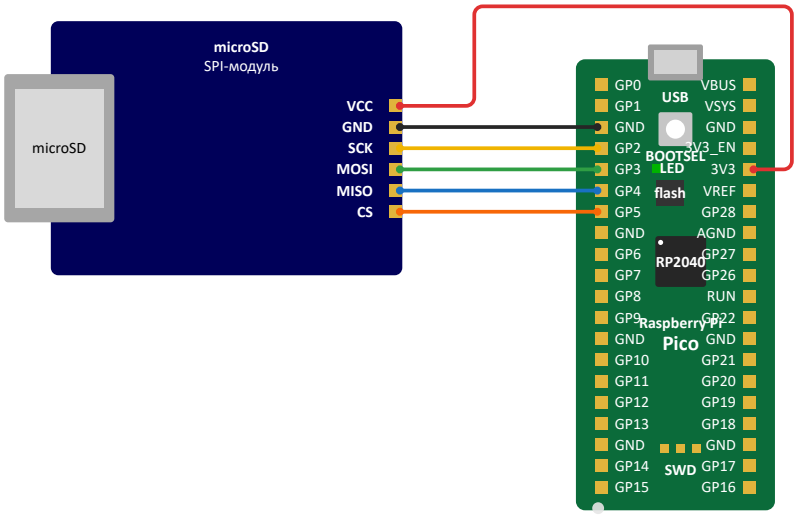
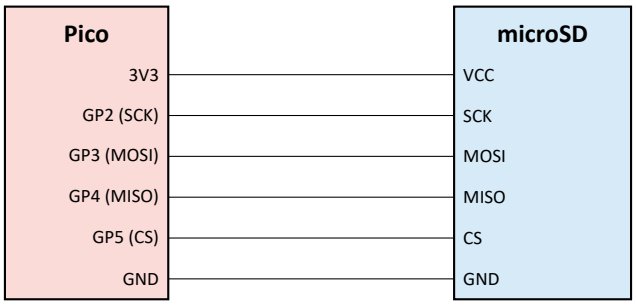


Рис. 9.7: Порядок работы с SD-картой: инициализация, подключение файловой системы, файлы



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 9.8: microSD-модуль, подключённый к SPI0: SCK — GP2, MOSI — GP3, MISO — GP4, CS — GP5

Осторожно

- Карта работает на **3,3 В**. Многие дешёвые модули имеют на плате стабилизатор и преобразователи уровней и рассчитаны на питание 5 V: у них питание подаётся на VCC от VBUS, а логика остаётся трёхвольтовой. Прочитайте описание вашего модуля. Если на плате нет микросхем, подавайте 3,3 V.
- Карты ёмкостью свыше 32 ГБ (SDXC, exFAT) библиотеки не понимают. Берите SD/SDHC до 32 ГБ и форматируйте в FAT32.
- Перед тем как вынуть карту, закройте файл (`close()` или блок `with`); иначе данные могут пропасть или файловая система повредится.

В **MicroPython** драйвер карты `sdcard.py` лежит в репозитории micropython-lib (для установки `mip.install("sdcard")` либо копирование файла). Он реализует интерфейс «блочного устройства», и его подключают командой `vfs.mount()` (в версиях до 1.21 — `os.mount()`). В **CircuitPython** модуль `sdcardio` встроен в прошивку: нужно создать `sdcardio.SDCard`, обернуть его в `storage.VfsFat` и подключить как папку.

Листинг 9.7: MicroPython: запись журнала на SD-карту

```
import os
import time
from machine import SPI, Pin
import sdcard                          # установка: import mip; mip.install("sdcard")

try:
    import vfs                         # MicroPython 1.21 и новее
except ImportError:
    import os as vfs                   # в старых версиях mount() лежит в os

# SPI0: SCK = GP2, MOSI = GP3, MISO = GP4, CS = GP5
spi = SPI(0, sck=Pin(2), mosi=Pin(3), miso=Pin(4))
sd = sdcard.SDCard(spi, Pin(5))        # драйвер сам запускает карту на малой скорости
vfs.mount(sd, "/sd")                   # теперь карта --- папка /sd

with open("/sd/log.csv", "a") as f:    # "a" --- дописывать в конец файла
    f.write("%d,%d\n" % (time.ticks_ms(), 42))

print("файлы на карте:", os.listdir("/sd"))
with open("/sd/log.csv") as f:
    print(f.read())

st = os.statvfs("/sd")                 # размер блока и число блоков
print("свободно: %.1f МБ" % (st[0] * st[3] / 1e6))
```

Листинг 9.8: CircuitPython: то же

```
import os
import time
import board
import busio
import sdcardio
import storage

# SCK = GP2, MOSI = GP3, MISO = GP4, CS = GP5
spi = busio.SPI(board.GP2, board.GP3, board.GP4)
sd = sdcardio.SDCard(spi, board.GP5)     # встроенный модуль, библиотеку ставить не надо
vfs = storage.VfsFat(sd)                 # файловая система FAT на карте
storage.mount(vfs, "/sd")                # теперь карта --- папка /sd

with open("/sd/log.csv", "a") as f:      # "a" --- дописывать в конец файла
    f.write("%f,%d\n" % (time.monotonic(), 42))

print("файлы на карте:", os.listdir("/sd"))
with open("/sd/log.csv") as f:
    print(f.read())

st = os.statvfs("/sd")
print("свободно: %.1f МБ" % (st[0] * st[3] / 1e6))
```

Формат строки «время;значение» называется CSV: его без дополнительных программ откроет Excel или Google Таблицы. Для журнала датчика откройте файл в режиме добавления (`"a"`) и записывайте по строке в секунду. Не пишите в файл слишком часто: каждая запись изнашивает карту.

9.9 Цветные дисплеи ST7789 и другие

Малогабаритные цветные TFT-дисплеи (ST7789, ILI9341, ST7735) тоже подключаются по SPI. К четырём «обычным» линиям добавляется ещё одна: **DC** (Data/Command). Она сообщает дисплею, чем является очередной байт: командой (DC = 0) или данными пикселей (DC = 1). Ещё есть сигнал сброса RES и подсветка BLK. Заметим, что MISO у дисплея нет: он только принимает данные, а на его выводе «SDA» подаётся MOSI.

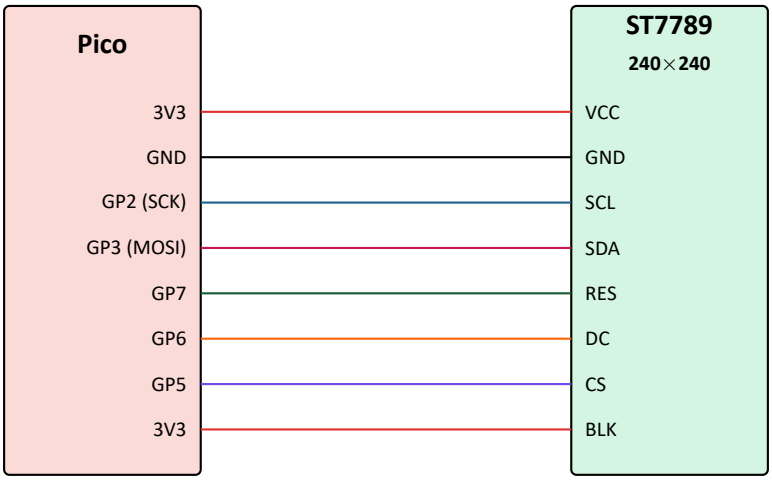


Рис. 9.9: Подключение дисплея ST7789 (названия выводов бывают разными: SCL = SCK, SDA = MOSI)

Для общего понимания в **MicroPython**-примере ниже дисплей инициализируется вручную — командами с кодами из документации микросхемы. Так видно, как работает вывод DC: он переключается между «команда» и «данные». Пиксель в формате **RGB565** занимает 2 байта: 5 бит красного, 6 — зелёного, 5 — синего. Например, красный — `0xF800`, зелёный — `0x07E0`, синий — `0x001F`. Полный экран 240×240 — это 115 200 байт, которые на частоте 20 МГц проходят за 46 мс.

Листинг 9.9: **MicroPython**: низкоуровневая заливка ST7789 (принцип работы)

```
import time
from machine import SPI, Pin

# Дисплей ST7789 240x240: SCK = GP2, MOSI(SDA) = GP3, CS = GP5, DC = GP6, RST = GP7
spi = SPI(0, baudrate=20000000, sck=Pin(2), mosi=Pin(3), miso=Pin(4))
cs = Pin(5, Pin.OUT, value=1)
dc = Pin(6, Pin.OUT, value=0)        # DC: 0 --- байт является командой, 1 --- данными
rst = Pin(7, Pin.OUT, value=1)

def write_cmd(cmd, data=None):
    cs(0)
    dc(0)
    spi.write(bytes([cmd]))          # команда
    if data:
        dc(1)
        spi.write(data)              # её параметры
    cs(1)

# аппаратный сброс и начальная настройка
rst(0); time.sleep_ms(20); rst(1); time.sleep_ms(120)
write_cmd(0x01); time.sleep_ms(150)  # SWRESET: программный сброс
write_cmd(0x11); time.sleep_ms(120)  # SLPOUT: выйти из режима сна
write_cmd(0x3A, b"\x55")             # COLMOD: 16 бит на пиксель (RGB565)
write_cmd(0x36, b"\x00")             # MADCTL: порядок строк и цветов
```

```
write_cmd(0x21)                      # INVON: у большинства IPS-модулей нужна инверсия
write_cmd(0x29)                      # DISPON: включить изображение

def fill(color):
    """Заливает весь экран цветом RGB565."""
    write_cmd(0x2A, bytes([0, 0, 0, 239]))   # CASET: столбцы 0..239
    write_cmd(0x2B, bytes([0, 0, 0, 239]))   # RASET: строки 0..239
    cs(0)
    dc(0)
    spi.write(b"\x2C")               # RAMWR: дальше идут пиксели
    dc(1)
    row = bytes([color >> 8, color & 0xFF]) * 240    # одна строка из 240 пикселей
    for _ in range(240):
        spi.write(row)
    cs(1)

while True:
    for c in (0xF800, 0x07E0, 0x001F):   # красный, зелёный, синий в RGB565
        fill(c)
        time.sleep(1)
```

Листинг 9.10: **CircuitPython**: тот же экран через `displayio`

```
import board
import busio
import displayio
import terminalio
from adafruit_display_text import label      # circup install adafruit_display_text
from adafruit_st7789 import ST7789           # circup install adafruit_st7789
try:
    from fourwire import FourWire            # CircuitPython 9.1 и новее
except ImportError:
    from displayio import FourWire           # более старые версии

displayio.release_displays()                 # освободить выводы, занятые прежним экраном

# SCK = GP2, MOSI = GP3, CS = GP5, DC = GP6, RST = GP7
spi = busio.SPI(board.GP2, board.GP3)
bus = FourWire(spi, command=board.GP6, chip_select=board.GP5, reset=board.GP7)
# Для некоторых модулей нужны ещё rowstart/colstart/rotation: смотрите документацию модуля
display = ST7789(bus, width=240, height=240)

screen = displayio.Group()
display.root_group = screen                  # в старых версиях: display.show(screen)

palette = displayio.Palette(1)
palette[0] = 0x0000FF                        # синий фон (цвет в формате 0xRRGGBB)
background = displayio.TileGrid(displayio.Bitmap(240, 240, 1), pixel_shader=palette)
screen.append(background)

text = label.Label(terminalio.FONT, text="Hello, Pico!", color=0xFFFF00, scale=3,
                   x=10, y=120)
screen.append(text)

while True:
    pass                                     # displayio обновляет экран сам
```

В реальных проектах низкоуровневый код обычно не пишут. Для **MicroPython** популярны драйверы `st7789py/st7789_mpy` (Russ Hughes), а для **CircuitPython** дисплеи работают через

`displayio` и библиотеки `adafruit_st7789`, `adafruit_ili9341`, `adafruit_display_text`. Название и API библиотеки зависят от версии, поэтому сверяйтесь с её описанием. Если на экране видны цвета наоборот, добавьте инверсию (`INVON`); если картинка сдвинута, нужны смещения `rowstart`/`colstart` (у модулей 135×240 и 240×280 они ненулевые).

9.10 I2C, SPI или UART: что выбрать

	UART	I2C	SPI
Провода	2 (TX, RX) + GND	2 (SDA, SCL) + GND	3 + по 1 CS на устройство + GND
Устройств на линии	2 (точка—точка)	десятки (адреса)	сколько хватит CS
Такт	нет, скорость договорная	есть, от ведущего	есть, от ведущего
Типичная скорость	9600–921 600 бод	100–400 кбит/с	1–60 Мбит/с
Направление	дуплекс (одновременно)	полудуплекс	дуплекс
Где встречается	GPS, Bluetooth, консоль	датчики, OLED, часы	SD, дисплеи, АЦП, радио

Таблица 9.4: Сравнение трёх последовательных интерфейсов

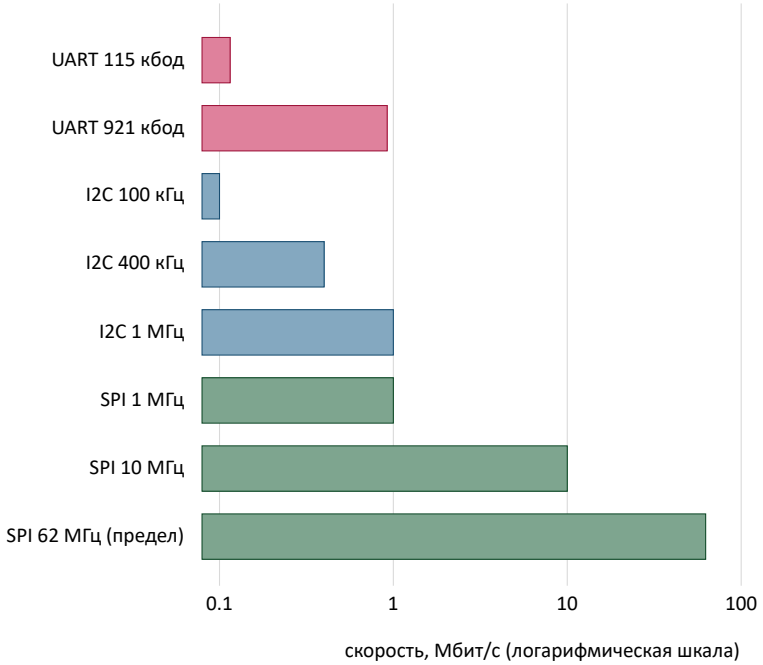


Рис. 9.10: Типичные скорости последовательных интерфейсов (у UART на каждый байт приходится 10 бит, у I2C — 9 тактов, поэтому полезная скорость ниже)

Это интересно

Шину SPI придумала фирма Motorola в середине 1980-х для микроконтроллеров 68HC11. Единого стандарта нет: документ описывает лишь идею, а режимы и порядок битов каждый производитель выбирает сам. Поэтому с новым устройством всегда читают таблицу «SPI timing» в его описании.

Если не работает

- **SD-карта: `OSError: no SD card` или `timeout waiting for v2 card`** — не те выводы или перепутаны MISO/MOSI, карта не отформатирована (FAT32), модулю нужно 5 V, плохой контакт в проводах (SD очень чувствительна к проводам: делайте их короче).
- **Данные приходят сдвинутыми или все байты равны `0xFF`/`0x00`** — неверный режим (`polarity`/`phase`) или CS не опущен; ведомый не отвечает.
- **Дисплей показывает «снег», цвета инвертированы или картинка смещена** — проверьте режим SPI, порядок цветов (RGB/BGR), инверсию и смещения у вашего размера матрицы.
- **Ничего не светится на 74HC595** — не подана защёлка RCLK или OE не подключён к GND.
- **Два устройства на одной шине мешают друг другу** — у второго устройства нет собственного CS или он не отпускает линию MISO; убедитесь, что CS неактивных устройств равен 1.
- **CircuitPython: `RuntimeError: Function requires lock`** — забыли `try_lock()` перед `configure()`/`write()`.

Итоги урока

- SPI: общие SCK, MOSI, MISO и отдельный CS (активный уровень — низкий) для каждого устройства; каждый обмен — сдвиг 8 бит в обе стороны.
- Режим определяется CPOL (уровень такта в покое) и CPHA (фронт считывания); у большинства устройств это режим 0.
- **MicroPython**: `SPI(0, baudrate, sck, mosi, miso)`, `write()`, `write_readinto()`; CS управляется вручную. **CircuitPython**: `busio.SPI`, захват и `configure()`.
- 74HC595 превращает 3 вывода в 8 выходов; bit-bang — это программная реализация SPI на обычных GPIO.
- SD-карта подключается как папка (`vfs.mount` / `storage.mount`); дисплей использует дополнительный вывод DC.

Проверь себя

1. Чем SPI отличается от I2C по числу проводов и по способу выбора устройства?
2. Что такое CPOL и CPHA и как узнать нужный режим для нового устройства?
3. Почему в SPI приём и передача происходят одновременно? Как «просто прочитать» данные?
4. Зачем в 74HC595 нужна защёлка RCLK, если данные уже загружены в регистр?
5. Почему SD-карту нельзя вынимать, пока файл открыт, и какую файловую систему она должна содержать?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соедините MOSI и MISO и запустите программу «шлейф»; отправьте свою строку из пяти букв и убедитесь, что она вернулась.

2. ★ Запустите бегущий огонь на 74HC595 и измените его скорость и направление.
3. ★★ Покажите на светодиодах 74HC595 двоичный счётчик, который увеличивается при нажатии кнопки (используйте маски битов).
4. ★★ Запишите на SD-карту 100 значений температуры (или значений АЦП) в формате CSV, потом прочитайте файл и найдите максимальное значение.
5. ★★★ Соедините два 74HC595 цепочкой (Q7' первого на SER второго), отправляйте два байта подряд и сделайте 16-разрядный бегущий огонь.
6. ★★★★ Сделайте журнал «чёрного ящика»: каждые 10 секунд пишите в файл время и значение датчика, перед каждой записью проверяйте свободное место, а при заполнении переходите на новый файл.

Урок 10. Датчики: температура, расстояние, движение

Цель	научиться подключать и читать популярные датчики, превращать «сырые» показания в физические величины и сглаживать шум
Опирается на	урок 6: аналоговые входы; урок 8: шина I2C; урок 4: измерение времени и импульсов
Результат	плата измеряет температуру и влажность (DHT22, DS18B20), расстояние (HC-SR04), движение (PIR), наклон (MPU6050), освещённость и громкость; показания сглажены
Понадобится	DHT22, DS18B20, HC-SR04, HC-SR501, MPU6050, фоторезистор, резисторы 1 kΩ, 2 kΩ, 4,7 kΩ, 10 kΩ

10.1 Идея урока коротко

Датчик превращает физическую величину в электрический сигнал. Одни датчики просто меняют сопротивление (фоторезистор), другие выдают напряжение (аналоговые модули), третьи — уже готовые числа по цифровой шине (BME280, MPU6050) или по собственному протоколу (DHT22, DS18B20, HC-SR04). Мы разберём самые популярные и научимся главному: *подключить, проверить, пересчитать в нормальные единицы и убрать шум*.

Датчик	Что измеряет	Связь с платой	Типичная точность
DHT22	температура, влажность	один вывод, свой протокол	±0,5 °C; ±2–5 %
DS18B20	температура	1-Wire (один вывод)	±0,5 °C
HC-SR04	расстояние до преграды	два вывода (TRIG, ECHO)	около 3 мм
HC-SR501 (PIR)	движение тёплых тел	один цифровой вход	–
MPU6050	ускорение, вращение	I2C	16 бит
Фоторезистор	освещённость	аналоговый вход	качественная
Модуль микрофона	громкость	аналоговый вход	качественная

Таблица 10.1: Датчики урока

10.2 Температура и влажность: DHT22

DHT22 (он же AM2302) — бюджетный датчик температуры и влажности. Внутри емкостной датчик влажности, терморезистор и крошечный процессор, который сам измеряет, переводит

в числа и передаёт результат *по одному проводу* по собственному протоколу. Диапазон: от -40 до $+80\,^{\circ}\mathrm{C}$, влажность 0–100 %. Чаще 1 раза в 2 секунды его опрашивать нельзя — датчик не успевает. Младший брат DHT11 проще и хуже (диапазон 0–50 °C, точность $\pm 2\,^{\circ}\mathrm{C}$), но подключается и читается так же.

Протокол на пальцах

Линия данных подтянута резистором к питанию, поэтому в покое там единица. Чтобы получить измерение, плата *опускает* линию на 1–10 мс (стартовый сигнал), затем отпускает. Датчик отвечает сигналом готовности (80 мкс низкий, 80 мкс высокий) и передаёт 40 бит: 16 бит влажности, 16 бит температуры и 8 бит контрольной суммы. Каждый бит начинается с низкого уровня 50 мкс, а его значение кодируется *длиной высокого уровня*: короткий (26–28 мкс) — `0`, длинный (около 70 мкс) — `1`.

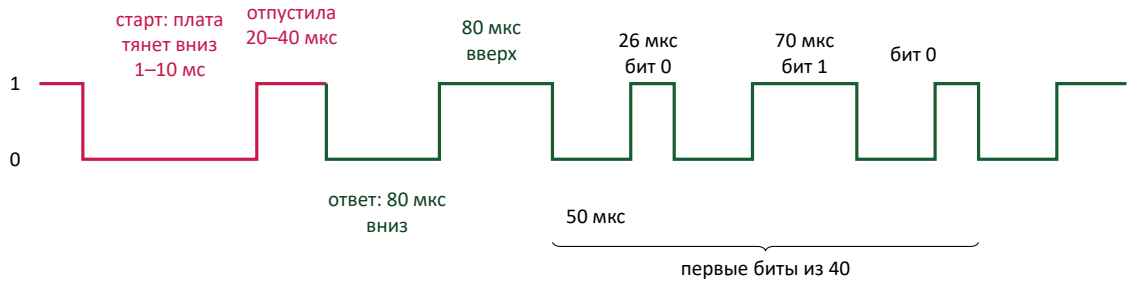
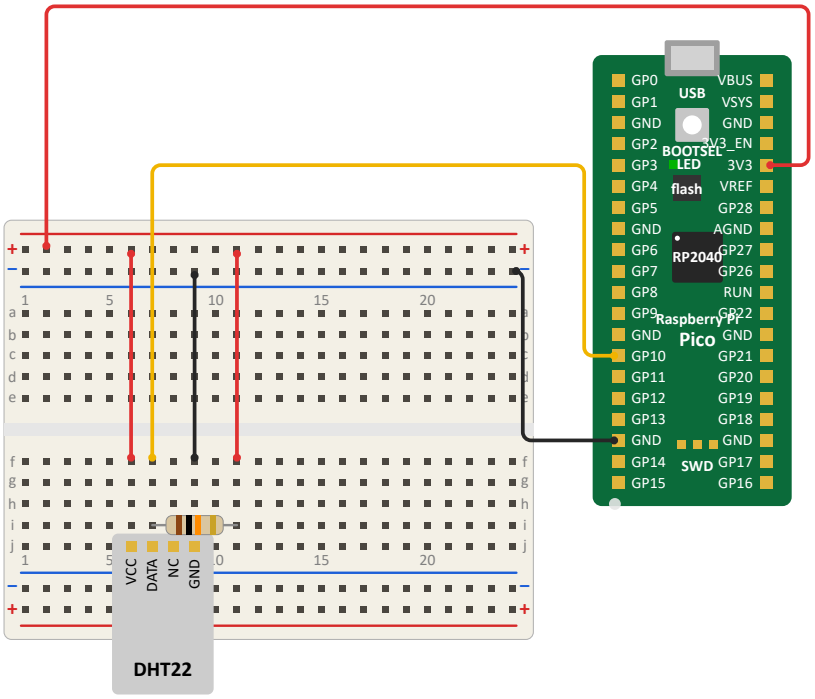


Рис. 10.1: Обмен с DHT22: запрос от платы (красный), ответ датчика (зелёный) и начало потока битов. Масштаб времени условный

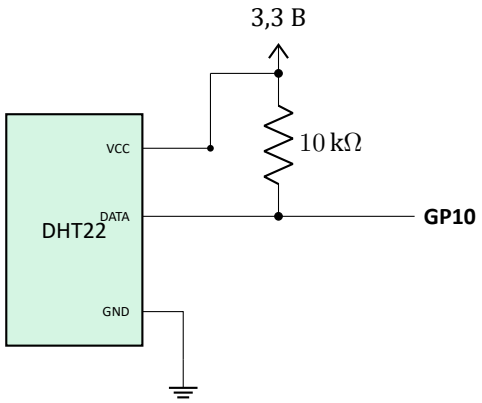
MicroPython и **CircuitPython** прячут весь этот протокол в библиотеке. Но принцип полезен: он показывает, что измерение чувствительно к *времени*, поэтому ошибки чтения (`OSError` в **MicroPython**, `RuntimeError` в **CircuitPython**) для DHT — нормальное, случайное явление. Лучшее лекарство — повторить попытку и не опрашивать чаще раза в 2 секунды.

Монтажная схема

У DHT22 четыре вывода (шаг 2,54 мм): питание, данные, пустой (не подключается), земля. Между питанием и линией данных ставится резистор подтяжки $10\,\mathrm{k}\Omega$. На готовых модулях с тремя выводами он уже есть.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 10.2: DHT22: линия данных на GP10 с подтяжкой 10 kΩ к 3,3 В

Листинг 10.1: MicroPython: DHT22 (встроенный модуль `dht`)

```
import time
import dht                          # встроен в MicroPython
from machine import Pin

sensor = dht.DHT22(Pin(10))         # для DHT11 --- dht.DHT11(Pin(10))

while True:
    try:
        sensor.measure()            # запрос к датчику (занимает около 20 мс)
        t = sensor.temperature()    # °C
        h = sensor.humidity()       # %
        print("T = %.1f C, влажность = %.1f %%" % (t, h))
    except OSError as e:            # не пришёл ответ или не сошлась контрольная сумма
        print("ошибка чтения:", e)
    time.sleep(2)                   # DHT22 нельзя опрашивать чаще раза в 2 секунды
```

Листинг 10.2: CircuitPython: DHT22 (библиотека `adafruit_dht`)

```
import time
import board
import adafruit_dht                  # circup install adafruit_dht

sensor = adafruit_dht.DHT22(board.GP10)      # для DHT11 --- adafruit_dht.DHT11

while True:
    try:
        t = sensor.temperature       # чтение происходит в момент обращения
        h = sensor.humidity
        print("T = %.1f C, влажность = %.1f %%" % (t, h))
    except RuntimeError as e:        # временные сбои чтения для этой библиотеки обычны
        print("ошибка чтения:", e.args[0])
    time.sleep(2)
```

Осторожно

Если датчик питается от 5 V, сигнал данных тоже может быть пятивольтовым. Для Pico питайте DHT22 от 3,3 V: тогда линия данных безопасна для вывода. Не подключайте провод данных к 5 V и не питайте датчик от VBUS, если не уверены в уровнях.

10.3 Много термометров на одном проводе: DS18B20

DS18B20 — цифровой термометр на шине **1-Wire**: данные ходят *по одному проводу*, причём к нему можно подключить много датчиков параллельно. У каждого датчика есть уникальный 64-битный номер, прошитый на заводе (8 бит код семейства `0x28`, 48 бит серийный номер, 8 бит контрольной суммы), по которому плата обращается к нужному. Диапазон от -55 до $+125$ °C, точность $\pm 0{,}5$ °C, а разрядность выбирается от 9 до 12 бит (шаг 0,5 °C … 0,0625 °C). Чем выше разрядность, тем дольше преобразование: от 94 мс до 750 мс. Герметичные версии в металлической трубке удобны для измерения воды, почвы и улицы.

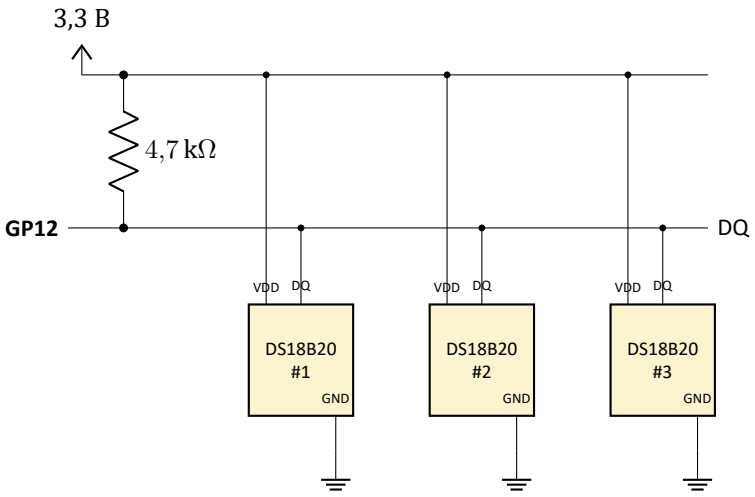


Рис. 10.3: Шина 1-Wire: один резистор $4{,}7\,\text{k}\Omega$ на всю линию данных, датчики подключены параллельно

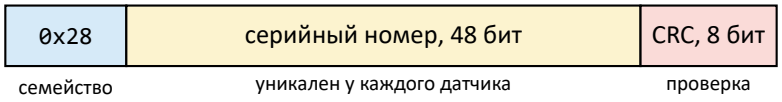


Рис. 10.4: 64-битный адрес DS18B20 (ROM): по нему плата различает датчики на одной шине

Работа с несколькими датчиками выглядит так: плата командой `scan()` получает список адресов всех найденных датчиков, запускает измерение сразу у всех (`convert_temp()`), ждёт 750 мс и читает каждого по его адресу. Сопротивление подтяжки 4,7 кОм — обязательное условие; без него датчик не будет найден.

Листинг 10.3: MicroPython: DS18B20 (модули `onewire` и `ds18x20`)

```
import time
import onewire                          # встроены в MicroPython
import ds18x20
from machine import Pin

ow = onewire.OneWire(Pin(12))           # линия 1-Wire с подтяжкой 4,7 кОм к 3,3 В
ds = ds18x20.DS18X20(ow)

roms = ds.scan()                        # у каждого датчика свой 64-битный адрес
print("найдено датчиков:", len(roms))
for rom in roms:
    print("  адрес:", "".join("%02X" % b for b in rom))

while True:
    ds.convert_temp()                   # запустить измерение сразу на ВСЕХ датчиках шины
    time.sleep_ms(750)                  # 12 бит: преобразование занимает до 750 мс
    for rom in roms:
        print("%.2f C" % ds.read_temp(rom), end="  ")
    print()
```

Листинг 10.4: CircuitPython: DS18B20 (`adafruit_onewire`, `adafruit_ds18x20`)

```
import time
import board
from adafruit_onewire.bus import OneWireBus      # circup install adafruit_onewire
from adafruit_ds18x20 import DS18X20             # circup install adafruit_ds18x20

bus = OneWireBus(board.GP12)            # линия 1-Wire с подтяжкой 4,7 кОм к 3,3 В
devices = bus.scan()                    # список найденных датчиков
print("найдено датчиков:", len(devices))

sensors = [DS18X20(bus, d) for d in devices]
while True:
    for s in sensors:
        print("%.2f C" % s.temperature, end="  ")   # внутри ждёт конец преобразования
    print()
    time.sleep(1)
```

Совет

Если нужен «паспорт» датчика (в какой точке он стоит), один раз выпишите адрес каждого, а потом подписывайте датчики наклейками. Адреса не меняются.

10.4 Расстояние: ультразвуковой HC-SR04

HC-SR04 — «эхолот»: излучатель посылает короткую серию ультразвуковых импульсов (40 кГц, их не слышно), они отражаются от преграды, и приёмник ловит эхо. Модуль сам измеряет, сколько времени прошло между отправкой и приёмом, и выдаёт на выход ECHO импульс *такой же длительности*. Нам остаётся измерить длину этого импульса.

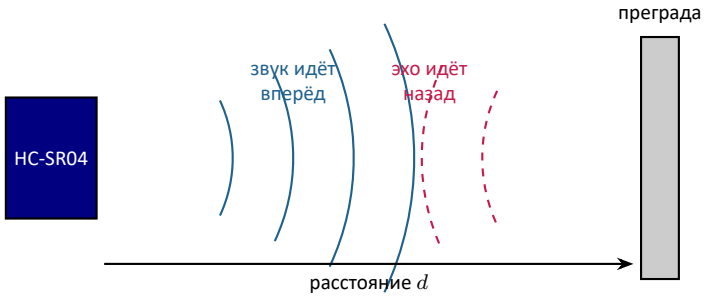


Рис. 10.5: Принцип эхолокации: звук проходит путь $2d$ за время t

Звук идёт к преграде и обратно, поэтому путь равен $2d$:

$$d = \frac{v \cdot t}{2}, \qquad v \approx 331{,}3 + 0{,}606 \cdot T \text{ м/с} \quad (T \text{ — температура воздуха, °C}).$$

При 20°C скорость звука $v = 343$ м/с $= 0{,}0343$ см/мкс, поэтому $d\,[\text{см}] \approx t\,[\text{мкс}]/58{,}3$. Эхо от преграды на расстоянии 1 м приходит через 5,8 мс. Если выйти за пределы измерения (диапазон 2–400 см) или направить датчик в мягкую ткань, эха не будет: импульс не придёт, и программа должна уметь это обработать.

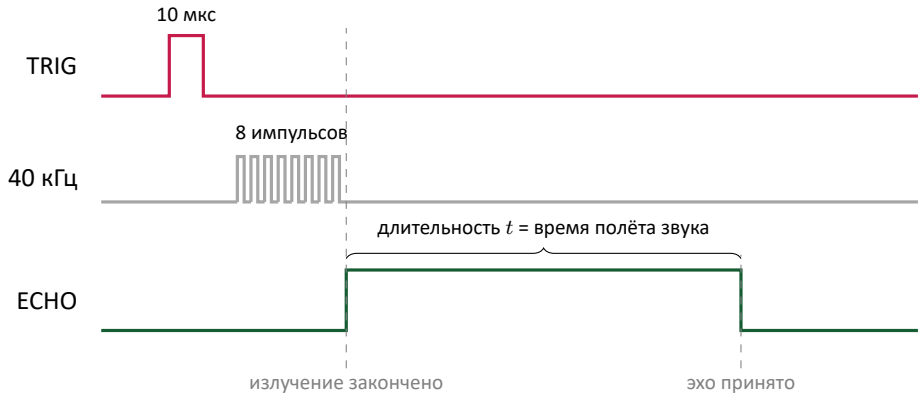


Рис. 10.6: Измерение расстояния: короткий запускающий импульс TRIG, затем импульс на ECHO, длина которого пропорциональна расстоянию

Осторожно

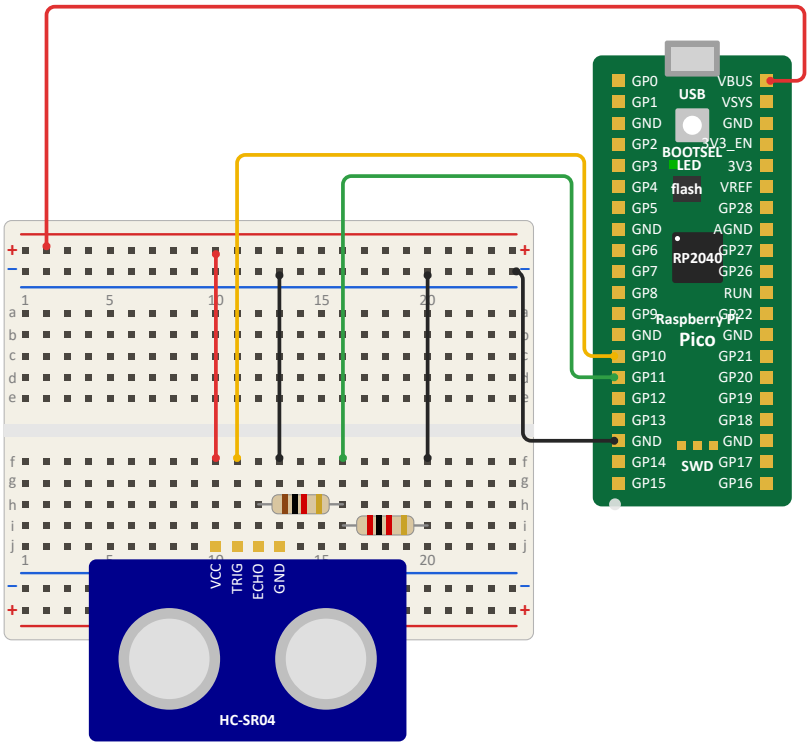
Модуль HC-SR04 питается от 5 V и выдаёт на выходе ECHO сигнал с амплитудой 5 V. Вход Pico выдерживает только $3{,}3$ V! Подключать ECHO прямо к GPIO **нельзя**: плата может выйти из строя. Нужен делитель напряжения из двух резисторов (или модуль HC-SR04P/RCWL-1601 с питанием 3,3 В).

Делитель напряжения

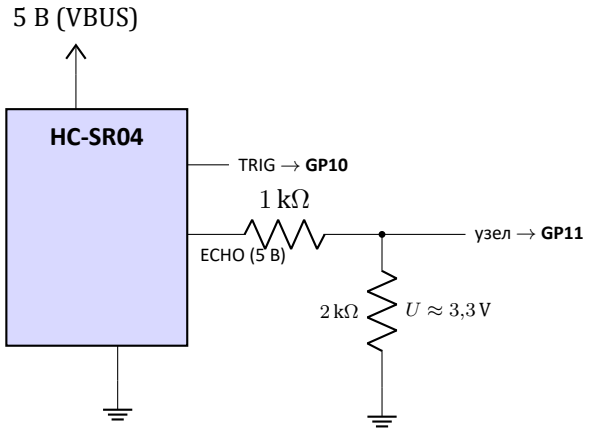
Два резистора, включённые последовательно между входным напряжением и землёй, делят напряжение в соотношении своих сопротивлений. Напряжение на нижнем резисторе R_2:

$$U_{\text{вых}} = U_{\text{вх}} \cdot \frac{R_2}{R_1 + R_2} = 5\,\text{V} \cdot \frac{2\,\text{k}\Omega}{1\,\text{k}\Omega + 2\,\text{k}\Omega} \approx 3{,}3\,\text{V}.$$

Подойдёт любая пара с отношением $R_2/(R_1 + R_2) \approx 0{,}66$: 1 кОм и 2 кОм, 2,2 кОм и 4,7 кОм, 10 кОм и 20 кОм (но слишком большие сопротивления замедляют фронт). Если нужного 2 кОм нет, можно взять два резистора по 1 кОм подряд. Делитель нужен только на линии *от датчика к плате*: сигнал TRIG идёт в обратную сторону, а вход HC-SR04 воспринимает и 3,3 В, поэтому TRIG подключается напрямую.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 10.7: HC-SR04: ECHO подключён через делитель 1 kΩ + 2 kΩ. Питание модуля — 5 В (VBUS)

Листинг 10.5: MicroPython: измерение расстояния (функция `time_pulse_us`)

```
import time
from machine import Pin, time_pulse_us

trig = Pin(10, Pin.OUT, value=0)
echo = Pin(11, Pin.IN)              # ECHO подключён ЧЕРЕЗ делитель напряжения!

def distance_cm():
    trig.value(0)
    time.sleep_us(2)
    trig.value(1)                   # импульс запуска длиной 10 мкс
    time.sleep_us(10)
    trig.value(0)
    # измеряем длительность высокого уровня на ECHO; ждём не дольше 30 мс (около 5 м)
    t = time_pulse_us(echo, 1, 30000)
    if t < 0:                       # -1 или -2: эхо не пришло
        return None
    return t * 0.0343 / 2           # скорость звука 0,0343 см/мкс, путь туда и обратно

while True:
    d = distance_cm()
    if d is None:
        print("нет эха")
    else:
        print("расстояние: %.1f см" % d)
    time.sleep_ms(100)
```

Листинг 10.6: CircuitPython: то же через `pulseio.PulseIn`

```
import time
import board
import digitalio
import pulseio

trig = digitalio.DigitalInOut(board.GP10)
trig.direction = digitalio.Direction.OUTPUT
# PulseIn записывает длительности импульсов; idle_state=False --- линия в покое «0»
echo = pulseio.PulseIn(board.GP11, maxlen=2, idle_state=False)

def distance_cm():
    echo.clear()                    # забыть старые измерения
    trig.value = True               # импульс запуска; 10 мкс обеспечивает сама
    trig.value = False              # скорость исполнения (медленнее не будет)
    t0 = time.monotonic()
    while len(echo) == 0:           # ждём, пока ECHO запишет импульс
        if time.monotonic() - t0 > 0.03:
            return None
    return echo[0] * 0.0343 / 2     # длительность в микросекундах -> сантиметры

while True:
    d = distance_cm()
    print("нет эха" if d is None else "расстояние: %.1f см" % d)
    time.sleep(0.1)

# Готовая библиотека делает то же самое:
#   import adafruit_hcsr04
```

```
#   sonar = adafruit_hcsr04.HCSR04(trigger_pin=board.GP10, echo_pin=board.GP11)
#   print(sonar.distance)
```

MicroPython предлагает готовую функцию `machine.time_pulse_us(pin, 1, timeout)`: она возвращает длительность импульса в микросекундах или отрицательное число при таймауте. В **CircuitPython** импульсы записывает `pulseio.PulseIn`, а готовая библиотека `adafruit_hcsr04` прячет всё это в свойство `sonar.distance`. Если плата и датчик находятся в воздухе с известной температурой, расстояние можно уточнить по формуле скорости звука.

10.5 Движение: PIR-датчик HC-SR501

PIR (пассивный инфракрасный датчик) не излучает ничего: он ловит изменения *теплового излучения* перед собой. Когда человек (или кошка) входит в поле зрения, линза Френеля разбивает картинку на зоны, и датчик видит скачок. Выход HC-SR501 — цифровой: 1, пока датчик «считает», что есть движение. Модуль питается от 5 V (или до 20 V), а выход у него 3,3 V, так что к плате его можно подключать напрямую. Два подстроечных резистора регулируют чувствительность (дальность) и время удержания выхода, а перемычка выбирает режим: **L** — одиночный запуск, **H** — повторный (время удержания продлевается, пока движение продолжается).

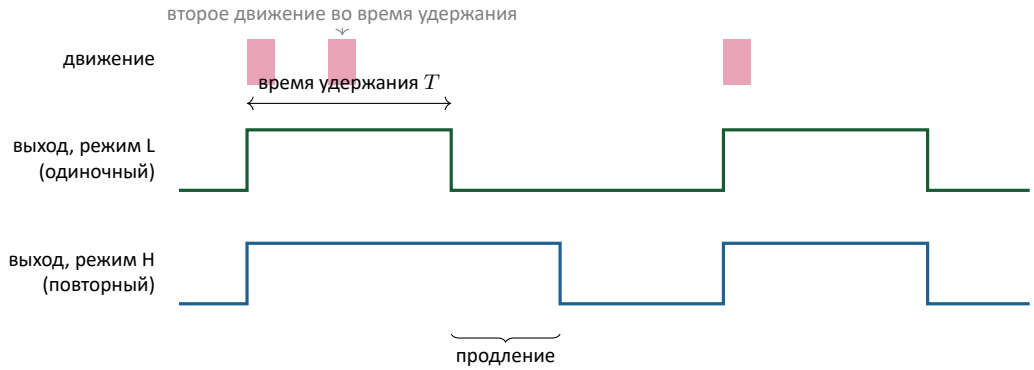


Рис. 10.8: Выход PIR-датчика: в режиме L второе движение игнорируется, в режиме H время удержания отсчитывается заново

Совет

При включении PIR «прогревается» и учится: первые 30–60 секунд выход может срабатывать случайно. Ставьте датчик подальше от батарей отопления и окон, на которые попадает солнце: они дают ложные срабатывания.

Листинг 10.7: **MicroPython**: PIR с прерыванием

```
import time
from machine import Pin

pir = Pin(14, Pin.IN)                # выход HC-SR501 --- логический 3,3 В
led = Pin("LED", Pin.OUT)
moved = False

def on_motion(pin):
    global moved
    moved = True                     # в обработчике только ставим флаг, остальное --- в цикле

pir.irq(trigger=Pin.IRQ_RISING, handler=on_motion)

```

```
print("Датчик греется около минуты. Ждём движение...")
while True:
    if moved:
        moved = False
        led.value(1)
        print("движение!", time.ticks_ms())
    if led.value() and not pir.value():   # выход датчика снова 0 --- движения нет
        led.value(0)
    time.sleep_ms(50)
```

Листинг 10.8: CircuitPython: PIR, опрос входа

```
import time
import board
import digitalio

pir = digitalio.DigitalInOut(board.GP14)      # выход HC-SR501 --- логический 3,3 В
pir.direction = digitalio.Direction.INPUT
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

print("Датчик греется около минуты. Ждём движение...")
previous = False
while True:
    now = pir.value
    if now and not previous:         # переход 0 -> 1: движение началось
        print("движение!", time.monotonic())
    led.value = now                  # светодиод повторяет выход датчика
    previous = now
    time.sleep(0.05)
```

В MicroPython обработчик прерывания `on_motion()` только поднимает флаг: делать в обработчике долгие операции (например, печать) нельзя. Основной цикл замечает флаг и делает остальное. В CircuitPython прерываний на выводах нет, поэтому вход просто опрашивается; чтобы заметить именно *начало* движения, программа запоминает предыдущее значение и ловит переход $0 \to 1$.

10.6 Аналоговые датчики: свет и звук

Фоторезистор (LDR) меняет сопротивление в зависимости от освещённости: на свету несколько килоом, в темноте сотни килоом и больше. Чтобы превратить сопротивление в напряжение, его включают в делитель с обычным резистором. Если фоторезистор подключить к 3,3 В, а резистор $R = 10\,\text{k}\Omega$ — к земле, то

$$U = 3{,}3\,\text{V} \cdot \frac{R}{R + R_{\text{LDR}}},$$

и чем ярче свет, тем больше напряжение. Это напряжение читает АЦП (урок 6).

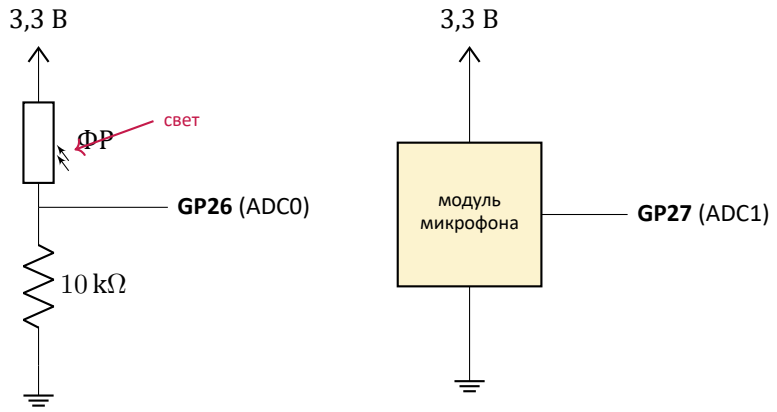


Рис. 10.9: Фоторезистор в делителе напряжения и аналоговый модуль микрофона, подключённые к входам АЦП

Микрофонные модули (на основе усилителя MAX4466 или LM393) подают на выход сигнал около половины питания, на который накладывается колебание от звука. Громкость оценивают не по самому значению, а по **размаху** — разности максимума и минимума за короткое окно (50 мс). Цифровой вывод DO таких модулей — компаратор с порогом, который настраивают подстроечным резистором: удобно для «хлопка».

Листинг 10.9: MicroPython: освещённость и громкость

```
import time
from machine import ADC, Pin

ldr = ADC(Pin(26))                    # делитель: фоторезистор + резистор 10 кОм
mic = ADC(Pin(27))                    # модуль микрофона с усилителем, выход аналоговый

def average(adc, n=16):
    """Среднее из n отсчётов: убирает мелкий шум АЦП."""
    total = 0
    for _ in range(n):
        total += adc.read_u16()
    return total // n

def sound_level(window_ms=50):
    """Размах сигнала (максимум минус минимум) за окно: громкость звука."""
    lo, hi = 65535, 0
    start = time.ticks_ms()
    while time.ticks_diff(time.ticks_ms(), start) < window_ms:
        v = mic.read_u16()
        if v < lo:
            lo = v
        if v > hi:
            hi = v
    return hi - lo

while True:
    light = average(ldr) * 100 // 65535      # освещённость в процентах (условных)
    print("свет: %3d %%   звук: %5d" % (light, sound_level()))
    time.sleep_ms(200)
```

Листинг 10.10: CircuitPython: то же

```
import time
import board
import analogio

ldr = analogio.AnalogIn(board.GP26)  # делитель: фоторезистор + резистор 10 кОм
mic = analogio.AnalogIn(board.GP27)  # модуль микрофона с аналоговым выходом

def average(adc, n=16):
    total = 0
    for _ in range(n):
        total += adc.value           # 0...65535
    return total // n

def sound_level(window=0.05):
    """Размах сигнала (максимум минус минимум) за окно в секундах."""
    lo, hi = 65535, 0
    start = time.monotonic()
    while time.monotonic() - start < window:
        v = mic.value
        if v < lo:
            lo = v
        if v > hi:
            hi = v
    return hi - lo

while True:
    light = average(ldr) * 100 // 65535
    print("свет: %3d %%   звук: %5d" % (light, sound_level()))
    time.sleep(0.2)
```

10.7 Наклон и движение: акселерометр MPU6050

MPU6050 — микросхема с трёхосевым акселерометром и трёхосевым гироскопом на шине I2C (адрес `0x68`, при подаче $3{,}3\,\text{V}$ на вывод AD0 — `0x69`). Подключение то же, что у других I2C-модулей: питание 3,3 В, земля, SDA на GP4, SCL на GP5. Датчик выдаёт 16-битные знаковые числа: при диапазоне $\pm 2g$ — 16 384 единицы на $1g$, для гироскопа в диапазоне $\pm 250\,^\circ/\text{с}$ — 131 единица на $^\circ/\text{с}$. После включения его нужно «разбудить»: записать `0` в регистр `0x6B`.

Покоящийся акселерометр измеряет ускорение свободного падения g, направленное вниз. Если наклонить плату на угол θ, то по оси x будет $a_x = g\sin\theta$, по оси z — $a_z = g\cos\theta$, откуда

$$\theta_{\text{наклон}} = \operatorname{atan2}\!\left(a_x,\ \sqrt{a_y^2 + a_z^2}\right), \qquad \theta_{\text{крен}} = \operatorname{atan2}(a_y,\ a_z).$$

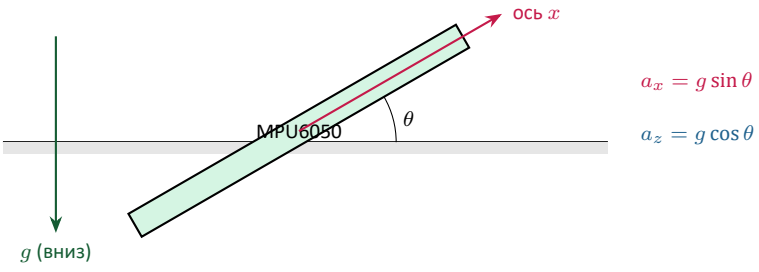


Рис. 10.10: Угол наклона по проекции ускорения свободного падения на оси датчика

Листинг 10.11: MicroPython: MPU6050 — чтение регистров и расчёт углов

```
import time
import math
import struct
from machine import I2C, Pin

MPU = 0x68                              # если вывод AD0 подтянут к 3,3 В --- адрес 0x69
i2c = I2C(0, sda=Pin(4), scl=Pin(5), freq=400000)

i2c.writeto_mem(MPU, 0x6B, b"\x00")  # регистр PWR_MGMT_1: «разбудить» датчик

def read_mpu():
    # 14 байт подряд с адреса 0x3B: ускорение XYZ, температура, гироскоп XYZ
    raw = i2c.readfrom_mem(MPU, 0x3B, 14)
    ax, ay, az, t, gx, gy, gz = struct.unpack(">hhhhhhh", raw)   # знаковые 16 бит
    return ax / 16384, ay / 16384, az / 16384, t / 340 + 36.53, gx / 131, gy / 131, gz / 131

while True:
    ax, ay, az, temp, gx, gy, gz = read_mpu()      # ускорения в g, угловая скорость в °/с
    pitch = math.degrees(math.atan2(ax, math.sqrt(ay * ay + az * az)))
    roll = math.degrees(math.atan2(ay, az))
    print("наклон: %6.1f  крен: %6.1f  (|a| = %.2f g)" % (
        pitch, roll, math.sqrt(ax * ax + ay * ay + az * az)))
    time.sleep_ms(200)
```

Листинг 10.12: CircuitPython: MPU6050 через библиотеку `adafruit_mpu6050`

```
import time
import math
import board
import busio
import adafruit_mpu6050              # circup install adafruit_mpu6050

i2c = busio.I2C(board.GP5, board.GP4, frequency=400000)
mpu = adafruit_mpu6050.MPU6050(i2c)  # адрес 0x68; библиотека сама будит датчик

while True:
    ax, ay, az = mpu.acceleration    # в м/с^2 (покоящийся датчик: примерно 9,8 по оси вниз)
    pitch = math.degrees(math.atan2(ax, math.sqrt(ay * ay + az * az)))
    roll = math.degrees(math.atan2(ay, az))
    print("наклон: %6.1f  крен: %6.1f  (T = %.1f C)" % (pitch, roll, mpu.temperature))
    time.sleep(0.2)
```

Акселерометр хорошо чувствует направление вниз, но при тряске показывает «лишние» ускорения. Гироскоп измеряет скорость вращения: его интегрирование даёт угол, но с накапливающимся дрейфом. Лучшие результаты даёт их комбинация — **комплементарный фильтр**: $\theta = 0{,}98\,(\theta + \omega\,\Delta t) + 0{,}02\,\theta_{\text{акс}}$.

10.8 Сбор и обработка данных

Любое измерение содержит шум. Показания АЦП «дрожат», расстояние скачет на сантиметр, а иногда попадает случайный выброс. Три приёма, которыми пользуются постоянно:

- **скользящее среднее** — среднее арифметическое последних N отсчётов; шум уменьшается примерно в $\sqrt{N}$ раз, но отклик замедляется;

- **экспоненциальное сглаживание (EMA)** — $y_n = y_{n-1} + \alpha\,(x_n - y_{n-1})$, где $0 < \alpha \le 1$: хранить нужно только одно число;
- **медианный фильтр** — берут медиану (среднее по порядку) трёх–пяти последних отсчётов: он выбрасывает одиночные «иглы».

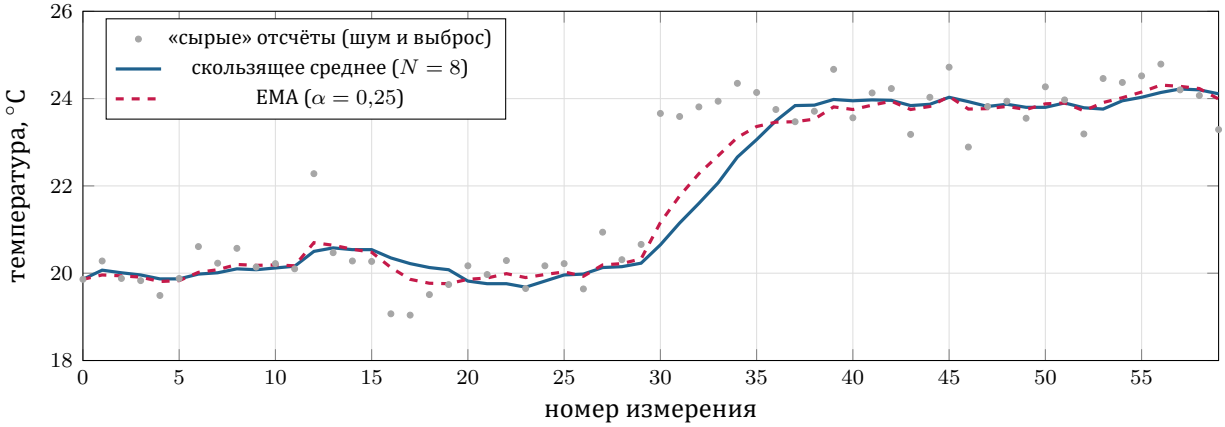


Рис. 10.11: Сглаживание зашумлённых данных: температура скачком меняется с 20 до 24 °C, на 12-м отсчёте — одиночный выброс

Цена сглаживания — запаздывание: чем сильнее фильтр, тем дольше он «догоняет» реальное изменение. Это видно на графике реакции фильтров на скачок.

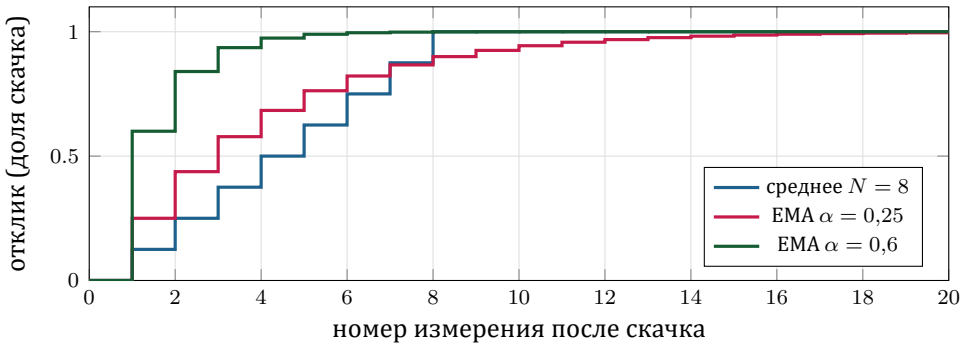


Рис. 10.12: Реакция фильтров на скачок: чем сильнее сглаживание, тем медленнее отклик

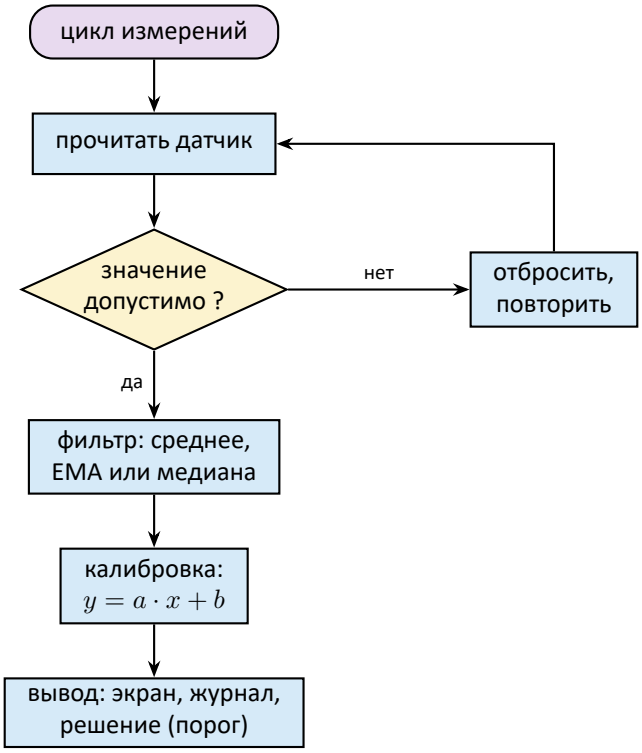


Рис. 10.13: Обработка одного измерения: проверка, фильтрация, калибровка, вывод

Калибровка — пересчёт «сырого» числа в настоящую величину по формуле прямой $y = a\,x + b$. Для двух точек (например, лёд и кипящая вода, показания x_1, x_2 и эталонные значения y_1, y_2) коэффициенты находят так: $a = \dfrac{y_2 - y_1}{x_2 - x_1}$, $b = y_1 - a\,x_1$.

10.9 Учимся программировать: классы

В примерах для сглаживания используются классы. **Класс** — «чертёж» объекта, который хранит свои данные и умеет что-то делать с ними. `MovingAverage(8)` создаёт отдельный фильтр с собственным буфером; можно держать несколько независимых фильтров (для расстояния, температуры и освещённости) и не путать их данные. Слово `self` означает «этот самый объект», а метод `__init__` вызывается автоматически при создании. Буфер работает по кругу: номер следующей ячейки `(i + 1) % n` после последней ячейки возвращается к нулю — так «кольцевой буфер» хранит ровно n последних значений.

Листинг 10.13: MicroPython: скользящее среднее и EMA (классы)

```
import time
from machine import ADC, Pin

adc = ADC(Pin(26))

class MovingAverage:
    """Скользящее среднее по последним n отсчётам."""
    def __init__(self, n):
        self.n = n
        self.buf = [0] * n          # кольцевой буфер
        self.i = 0                  # куда писать следующий отсчёт
        self.count = 0              # сколько отсчётов уже накоплено

    def update(self, x):
        self.buf[self.i] = x
```

```
        self.i = (self.i + 1) % self.n
        if self.count < self.n:
            self.count += 1
        return sum(self.buf) / self.count

class Ema:
    """Экспоненциальное сглаживание: новое = старое + alpha * (отсчёт - старое)."""
    def __init__(self, alpha):
        self.alpha = alpha
        self.value = None

    def update(self, x):
        if self.value is None:
            self.value = x           # первый отсчёт берём как есть
        else:
            self.value += self.alpha * (x - self.value)
        return self.value

def median3(a, b, c):
    """Медиана трёх чисел: отбрасывает одиночные выбросы."""
    return sorted((a, b, c))[1]

ma = MovingAverage(8)
ema = Ema(0.2)
while True:
    x = adc.read_u16()
    print(x, int(ma.update(x)), int(ema.update(x)))   # исходное, среднее, EMA
    time.sleep_ms(50)
```

Листинг 10.14: CircuitPython: то же

```
import time
import board
import analogio

adc = analogio.AnalogIn(board.GP26)

class MovingAverage:
    """Скользящее среднее по последним n отсчётам."""
    def __init__(self, n):
        self.n = n
        self.buf = [0] * n
        self.i = 0
        self.count = 0

    def update(self, x):
        self.buf[self.i] = x
        self.i = (self.i + 1) % self.n
        if self.count < self.n:
            self.count += 1
        return sum(self.buf) / self.count

class Ema:
    """Экспоненциальное сглаживание: новое = старое + alpha * (отсчёт - старое)."""
    def __init__(self, alpha):
        self.alpha = alpha
        self.value = None

```

```
    def update(self, x):
        if self.value is None:
            self.value = x
        else:
            self.value += self.alpha * (x - self.value)
        return self.value

def median3(a, b, c):
    return sorted((a, b, c))[1]

ma = MovingAverage(8)
ema = Ema(0.2)
while True:
    x = adc.value
    print((x, int(ma.update(x)), int(ema.update(x))))   # кортеж: график в Mu/Thonny
    time.sleep(0.05)
```

Это интересно

Скорость звука в воздухе зависит от температуры, но почти не зависит от давления и частоты. Летучие мыши и дельфины измеряют расстояние тем же способом, что и HC-SR04, а точность их «сонара» — миллиметры. Дальномеры автомобильных парктроников устроены так же: излучатель, приёмник и подсчёт времени задержки эха.

Если не работает

- **DHT22: `OSError: [Errno 110] ETIMEDOUT`** (или `RuntimeError`) — нет подтяжки $10\,\text{k}\Omega$, перепутаны выводы, провод слишком длинный, или чтение чаще раза в 2 с.
- **DS18B20 не найден** (`scan()` возвращает пустой список или `-127/85` °C) — нет резистора $4{,}7\,\text{k}\Omega$ между DQ и питанием, перепутаны питание и земля (датчик сильно греется!). Значение $85\,^\circ$C означает «преобразование не выполнено».
- **HC-SR04 всё время «нет эха» или показывает ерунду** — нет питания $5\,\text{V}$, перепутаны TRIG и ECHO, делитель собран неверно. Проверьте напряжение на выводе GPIO мультиметром: оно не должно превышать $3{,}3\,\text{V}$.
- **Расстояние «прыгает»** — мягкая или наклонная преграда, нет сглаживания: добавьте медианный фильтр и среднее по 5 измерениям.
- **PIR срабатывает сам** — прогрев, источник тепла рядом, слишком высокая чувствительность; уменьшите её подстроечным резистором.
- **MPU6050: все значения нулевые** — датчик не «разбужен» (регистр `0x6B`); адрес `0x68` или `0x69`.

Итоги урока

- DHT22 — температура и влажность по одному проводу, опрос не чаще раза в 2 с, подтяжка $10\,\text{k}\Omega$; DS18B20 — 1-Wire, подтяжка $4{,}7\,\text{k}\Omega$, много датчиков на одном проводе.
- HC-SR04: $d = vt/2$, $d\,[\text{см}] \approx t\,[\text{мкс}]/58{,}3$; ECHO на 5 В подключается к Pico *только через делитель* 1 кОм + 2 кОм.
- PIR даёт цифровой сигнал «есть движение»; фоторезистор и микрофон читаются АЦП (в делителе или по размаху).
- MPU6050: углы наклона из ускорений, $\theta = \text{atan2}(a_x, \sqrt{a_y^2 + a_z^2})$; перед работой датчик

нужно разбудить.

- Показания фильтруют: скользящее среднее, EMA, медиана; калибровка — линейный пересчёт $y = ax + b$.

Проверь себя

1. Как DHT22 передаёт нулевой и единичный биты по одному проводу?
2. Почему на линию данных DS18B20 нужен резистор и как плата различает несколько датчиков на одном проводе?
3. Вычислите расстояние, если эхо HC-SR04 длилось 4,66 мс при температуре воздуха 20 °C.
4. Почему ECHO нельзя подключать напрямую к Pico? Сколько вольт будет на выходе делителя 1 кОм + 2 кОм при входе 5 В?
5. Чем скользящее среднее отличается от медианного фильтра и в каком случае медиана лучше?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Подключите DHT22 и выведите температуру и влажность на экран консоли раз в 2 секунды.
2. ★ Измерьте расстояние HC-SR04 до стены и сравните с рулеткой; запишите погрешность.
3. ★★ Сделайте «парктроник»: пищалка (или светодиод) мигает чаще по мере приближения к преграде; ближе 10 см — звучит непрерывно.
4. ★★ Подключите два DS18B20 и выводите разность температур; подпишите адреса датчиков.
5. ★★★ Добавьте в измерение расстояния температурную поправку: возьмите температуру из DS18B20 и скорректируйте скорость звука.
6. ★★★★ Сделайте «охранную сигнализацию»: PIR + HC-SR04 + MPU6050; тревога срабатывает при движении или при наклоне платы больше $15°$, события записываются в журнал с отметкой времени (урок 12).

Урок 11. Адресные светодиоды WS2812

Цель	научиться управлять адресными светодиодами WS2812 (NeoPixel): понять протокол, рассчитать питание, смешивать цвета в RGB и HSV и писать красивые эффекты
Опирается на	урок 2: цифровой выход; урок 4: время без `sleep()`; понятия цикла и списка
Результат	кольцо из 12 светодиодов показывает радугу, «комету», «дыхание» и «огонь»; ты умеешь считать ток и выбирать блок питания
Понадобится	кольцо или лента WS2812B, резистор $330\,\Omega$, конденсатор $1000\,\mu F$ ($6{,}3\,V$ и выше), провода

11.1 Идея урока коротко

Обычному светодиоду нужны свой вывод платы и свой резистор. Гирлянду из сотни ламп так не подключить: не хватит ни выводов, ни проводов. В светодиоде **WS2812** (его же называют **NeoPixel**) внутри красного, зелёного и синего кристаллов спрятан ещё и крошечный микроконтроллер. Все светодиоды соединяют цепочкой: от Pico по одному проводу данных идёт поток цветов, каждый пиксель забирает свою порцию и передаёт остальное дальше. Поэтому на любой длины ленту хватает *одного* вывода GPIO. В **MicroPython** лентой управляет класс `neopixel.NeoPixel`, в **CircuitPython** — одноимённая библиотека `neopixel`.

11.2 Как устроен WS2812

В корпусе размером 5×5 мм находятся три кристалла (R, G, B) и управляющая микросхема. Выводов у неё четыре: питание **VDD** ($5\,V$), земля **GND**, вход данных **DIN** и выход данных **DOUT**. Схема принимает 24 бита, запоминает их, а остальные биты усиливает и пересылает на DOUT следующего пикселя. Яркость каждого цвета регулируется ШИМ (урок 5) прямо в светодиоде, поэтому Pico не нужно постоянно «держать» нужный цвет: достаточно один раз отправить кадр.

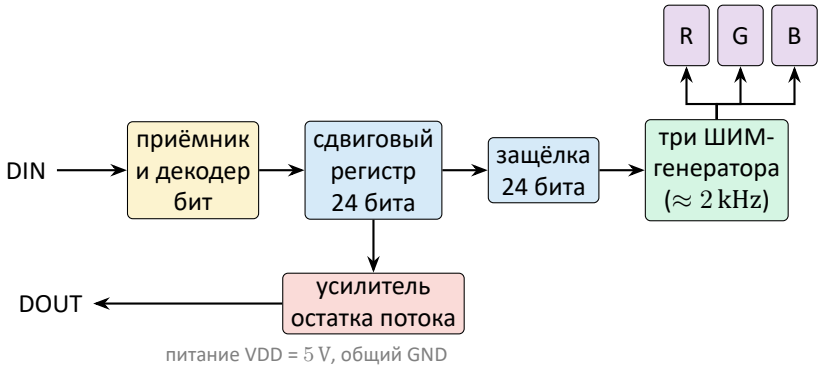


Рис. 11.1: Упрощённая структура одного пикселя WS2812: первые 24 бита остаются у него, остальные уходят дальше

11.3 Протокол: один провод, 800 кГц

Тактового провода нет: и данные, и «такт» передаются по одному проводу DIN. Каждый бит — это импульс фиксированного периода $T = 1{,}25\,\mu s$ (частота $800\,kHz$), а значение бита задаёт *ширина* импульса: короткий ($\approx 0{,}4\,\mu s$ единицы) означает «0», длинный ($\approx 0{,}8\,\mu s$) — «1». Это похоже на азбуку Морзе из коротких и длинных сигналов. Когда линия остаётся на низком уровне

дольше 50 µs (у новых версий — до 280 µs), все пиксели считают, что кадр закончился, и выводят принятые цвета на кристаллы. Это называется **сброс** (reset, latch).

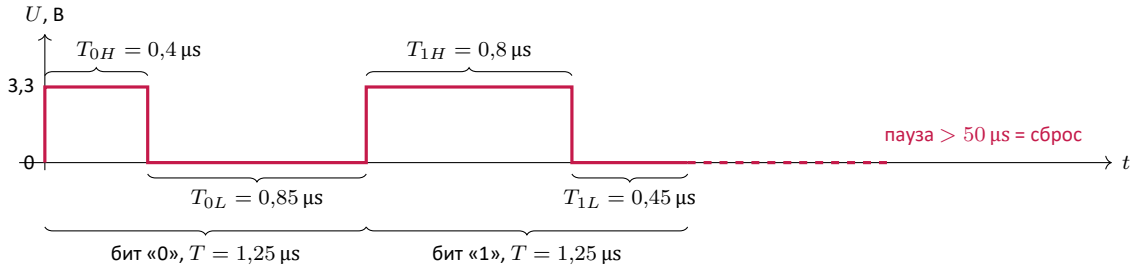


Рис. 11.2: Временная диаграмма: кодирование нуля и единицы шириной импульса (допуск по данным производителя ± 150 ns)

Задержки измеряются сотнями наносекунд, поэтому обычный цикл Python такой сигнал сформировать не может: интерпретатор слишком медленный и неровный. **MicroPython** на RP2040 использует функцию `machine.bitstream()`, которая отправляет биты точно по времени в готовом машинном коде. Есть и третий путь — аппаратный блок PIO, который сам выдаёт точные импульсы (урок 14); именно на нём строится реализация в **CircuitPython**.

Формат кадра: 24 бита на пиксель

Каждый пиксель получает 24 бита: по 8 на каждый цвет. Порядок в WS2812 — не RGB, а **GRB** (сначала зелёный!). Старший бит идёт первым. Значения каналов лежат в диапазоне 0–255, всего $256^3 = 16{,}7$ млн цветов.

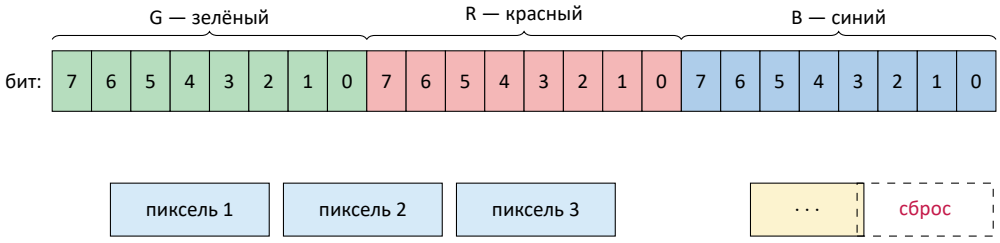


Рис. 11.3: Кадр: 24 бита на пиксель в порядке G–R–B и цепочка пикселей, завершаемая паузой-сбросом

Это интересно

Скорость обновления ограничена протоколом. На один пиксель уходит $24 \cdot 1{,}25$ µs $= 30$ µs, на ленту из N пикселей — $N \cdot 30$ µs плюс сброс. Для $N = 60$ получается около 2 мс (теоретически ≈ 470 кадров в секунду), а для $N = 1000$ — уже около 30 мс, т. е. не больше 33 кадров в секунду. Прерывать передачу нельзя: если линия «провиснет» ниже уровня на 50 µs и дольше, лента решит, что кадр закончился.

11.4 Питание: самая важная часть

Каждый из трёх кристаллов потребляет до 20 mA, значит, один пиксель с белым цветом на полной яркости — до 60 mA. Пока светодиодов немного, это заметно только в расчётах, но для 30 пикселей получается уже 1,8 A, а Pico от USB может отдать не больше 0,5 A (и сама потребляет 25 – 30 mA). Ток примерно пропорционален сумме яркостей каналов:

$$I \approx N \cdot 20\,\text{mA} \cdot \frac{R + G + B}{255}.$$

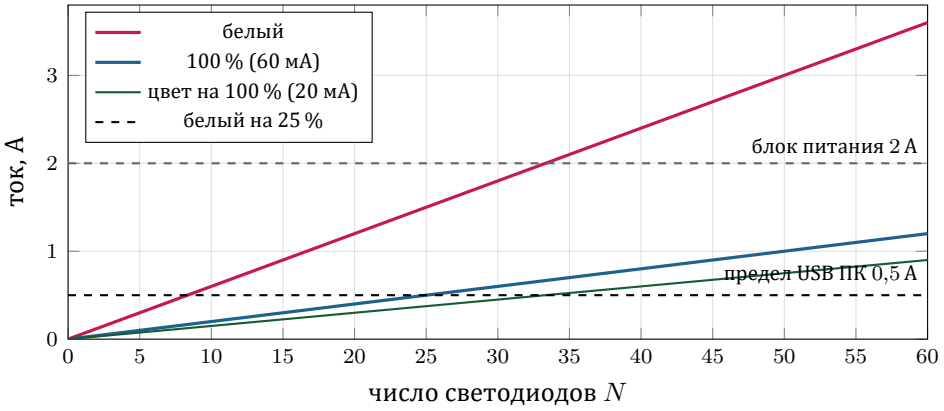


Рис. 11.4: Ток ленты в зависимости от числа светодиодов: при яркости 25 % ток падает в четыре раза

Как подобрать блок питания. Определи самый тяжёлый режим (обычно — полный белый), умножь на N, добавь запас 20 %:

$$I_{\text{БП}} = 1{,}2 \cdot N \cdot 60\,\text{mA} \cdot k_{\text{ярк}}.$$

Для 60 светодиодов на полной яркости: $1{,}2 \cdot 60 \cdot 0{,}06 = 4{,}3$ А, берём блок 5 V/5 А. Если же ограничить яркость программно до 30 %, достаточно $1{,}5$ А. Ограничивая яркость, ты экономишь деньги и нервы (и меньше слепишь глаза!).

Осторожно

- Светодиоды питают от 5 V, а не от 3V3: вывод 3V3 платы слабый и просто «просядет».
- От USB компьютера безопасно питать **не больше 8–12 пикселей** и с невысокой яркостью. Длинную ленту подключай к отдельному блоку питания.
- **Земля общая**: GND блока питания, ленты и Pico обязательно соединяют вместе, иначе сигнал данных не с чем сравнивать.
- Провода питания для длинной ленты должны быть толстыми, иначе на них «упадёт» напряжение, и дальние светодиоды станут желтоватыми.
- Сначала соединяй GND, потом подавай питание; не подключай ленту к включённому питанию «на горячую».

Защитные детали: конденсатор и резистор

- **Конденсатор** 1000 µF ($6{,}3$ – 16 V) ставят между +5V и GND рядом с лентой. В момент включения лента «хватает» большой ток, а конденсатор подаёт его как маленький аккумулятор и сглаживает броски. Электролитический конденсатор *полярный*: вывод с полосой — это «−», его включают в GND. Перепутанный конденсатор может **взорваться**.
- **Резистор** 300 – $500\,\Omega$ в линии данных (рядом с первым пикселем) гасит выбросы и звон на проводе и защищает вход DIN.

Уровень сигнала: 3,3 В или 5 В?

Для питания 5 V вход DIN считает единицей напряжение выше $0{,}7 \cdot V_{\text{DD}} = 3{,}5$ V. Pico выдаёт лишь $3{,}3$ V, т. е. чуть-чуть меньше. На практике кольца и короткие ленты с первого пикселя обычно работают, но иногда начинают «шалить» (случайные вспышки, неверные цвета). Есть три решения:

1. поставить **преобразователь уровней** (микросхемы 74AHCT125, 74HCT245 или готовый модуль 3,3 В→5 В) между Pico и DIN;
2. питать **первый** пиксель чуть меньшим напряжением (один диод в цепи +5V снижает его до $\approx 4{,}3$ V и порог падает до $\approx 3{,}0$ V);
3. держать провод данных очень коротким.

Обратное (подать 5 V с выхода ленты *в* Pico) — категорически нельзя: GPIO не терпит 5 В.

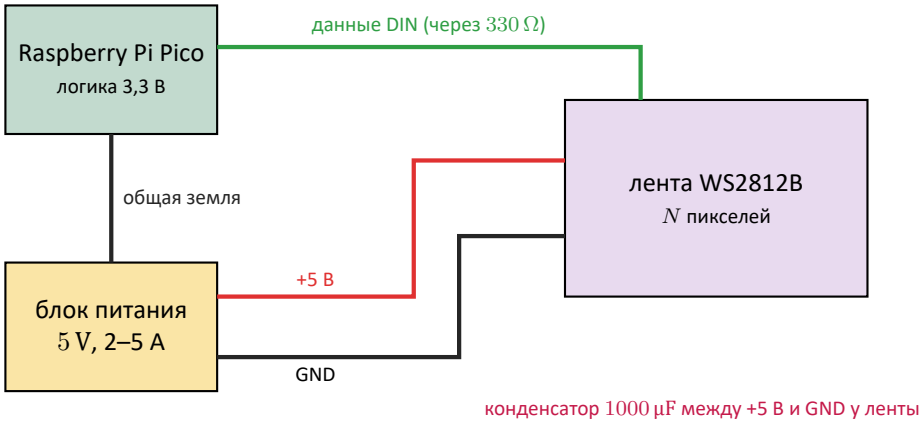
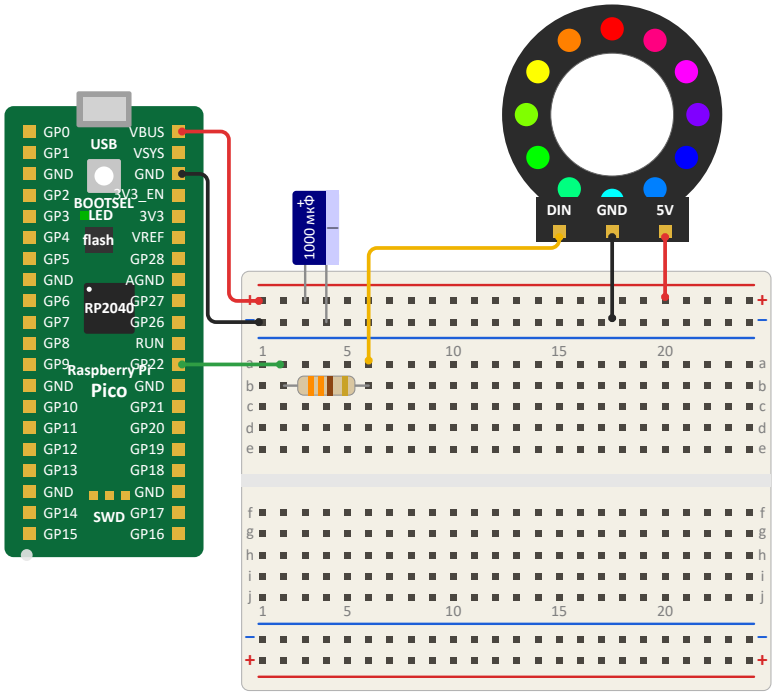


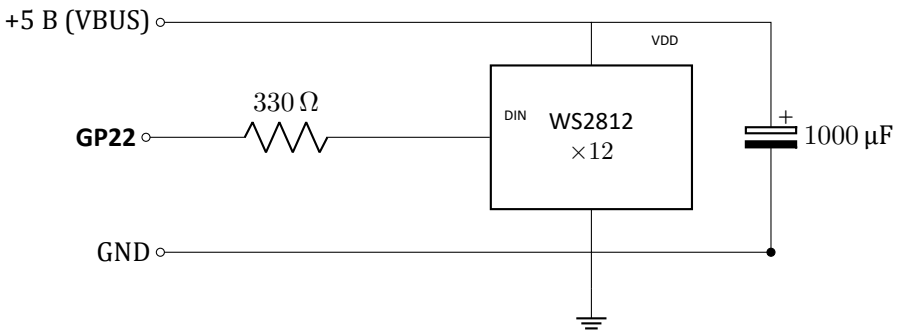
Рис. 11.5: Питание длинной ленты от отдельного блока: логика Pico и лента делят общий провод GND

11.5 Подключаем кольцо

Соберём схему с кольцом из 12 светодиодов. Данные идут с вывода **GP22** через резистор $330\,\Omega$ на DIN. Питание кольца берём с вывода VBUS (это 5 V от USB); пока светодиодов 12 и яркость умеренная, это допустимо. Конденсатор 1000 µF включён между шинами «+» и «−» макетной платы.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 11.6: Кольцо WS2812 на 12 светодиодов: данные — GP22, питание — VBUS

Проследи путь тока: красный провод несёт $+5$ V от VBUS на верхнюю красную шину, а с неё — на контакт 5V кольца; чёрные провода собирают землю от Pico и от кольца на синюю шину; жёлтый провод несёт сигнал от резистора на DIN. Конденсатор стоит между шинами, у самого места, где питание уходит на светодиоды.

11.6 Первая программа

Листинг 11.1: MicroPython: первые пиксели и заливка

```
from machine import Pin
import neopixel
import time

N = 12                                  # число светодиодов в кольце
pixels = neopixel.NeoPixel(Pin(22), N)  # линия данных DIN подключена к GP22

pixels.fill((0, 0, 0))                  # погасить все пиксели
pixels[0] = (40, 0, 0)                  # цвет задаётся тройкой (R, G, B), 0..255
pixels[1] = (0, 40, 0)
pixels[2] = (0, 0, 40)
```

```
pixels.write()                          # только теперь данные уходят на ленту
time.sleep(2)

pixels.fill((10, 10, 10))               # слабый белый на все пиксели
pixels.write()
```

Листинг 11.2: CircuitPython: то же самое

```
import time
import board
import neopixel

N = 12                                  # число светодиодов в кольце
# brightness --- общий множитель яркости (0.0 ... 1.0)
# auto_write=False: данные отправляются только по команде show()
pixels = neopixel.NeoPixel(board.GP22, N, brightness=0.2, auto_write=False)

pixels.fill((0, 0, 0))                  # погасить все пиксели
pixels[0] = (255, 0, 0)                 # (R, G, B); реальная яркость = значение x 0.2
pixels[1] = (0, 255, 0)
pixels[2] = (0, 0, 255)
pixels.show()
time.sleep(2)

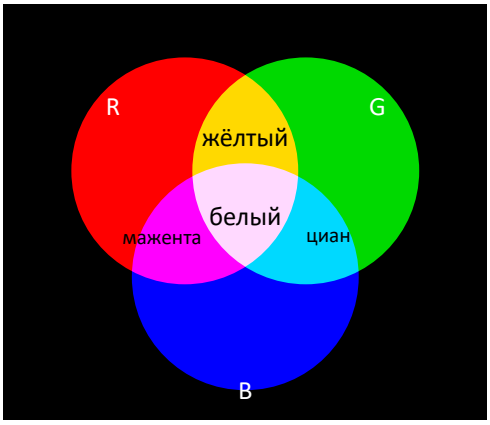
pixels.fill((255, 255, 255))            # белый на 20 % яркости
pixels.show()
```

Разберём отличия:

- MicroPython хранит цвета в буфере, а метод `write()` отправляет его на ленту. В CircuitPython это `show()`; при `auto_write=True` (значение по умолчанию) `show()` вызывается после *каждого* присваивания, что для 12 пикселей незаметно, а для длинной ленты резко замедляет программу. В анимациях всегда ставь `auto_write=False`.
- В CircuitPython есть параметр `brightness` — общий множитель яркости. Один пиксель задаётся как (R, G, B) с привычным диапазоном 0–255, а реальная яркость умножается на 0,2. В MicroPython такого параметра нет: яркость приходится уменьшать самостоятельно (в примерах мы берём малые значения каналов).
- Порядок GRB обе библиотеки скрывают: ты всегда пишешь (R, G, B). Для лент с белым кристаллом (SK6812 RGBW, 4 байта на пиксель) в MicroPython нужно указать `bpp=4`, в CircuitPython — `pixel_order=neopixel.GRBW`.
- Если цвета перепутаны (красный горит зелёным), у ленты другой порядок: в MicroPython его нужно менять в данных, в CircuitPython — параметром `pixel_order`, например `neopixel.RGB`.

11.7 Цвет: смешиваем свет

Глаз человека различает цвета тремя видами колбочек: «красными», «зелёными» и «синими». Поэтому любой цвет на экране и в светодиоде получают смешиванием трёх источников. Это **аддитивная** смесь (свет складывается): красный + зелёный даёт жёлтый, все три — белый. (В рисовании красками всё наоборот: краски поглощают свет.)



$(255, 0, 0)$ красный
$(0, 255, 0)$ зелёный
$(0, 0, 255)$ синий
$(255, 255, 0)$ жёлтый
$(255, 0, 255)$ мажента
$(0, 255, 255)$ циан
$(255, 255, 255)$ белый
$(0, 0, 0)$ чёрный (выкл.)

Рис. 11.7: Аддитивное смешивание цветов RGB и значения каналов

Модель HSV

Подбирать цвета тройками (R, G, B) неудобно: как сделать «чуть более оранжевый» или «чуть бледнее»? Для этого придумана модель **HSV**:

- **H (hue, оттенок)** — угол на цветовом круге 0–360° (в программе удобно число от 0 до 1);
- **S (saturation, насыщенность)** — от 0 (серый/белый) до 1 (чистый цвет);
- **V (value, яркость)** — от 0 (чёрный) до 1 (максимум).

Чтобы получить радугу, достаточно менять только H от 0 до 1, оставив S и V постоянными. А чтобы приглушить свет — уменьшать V.

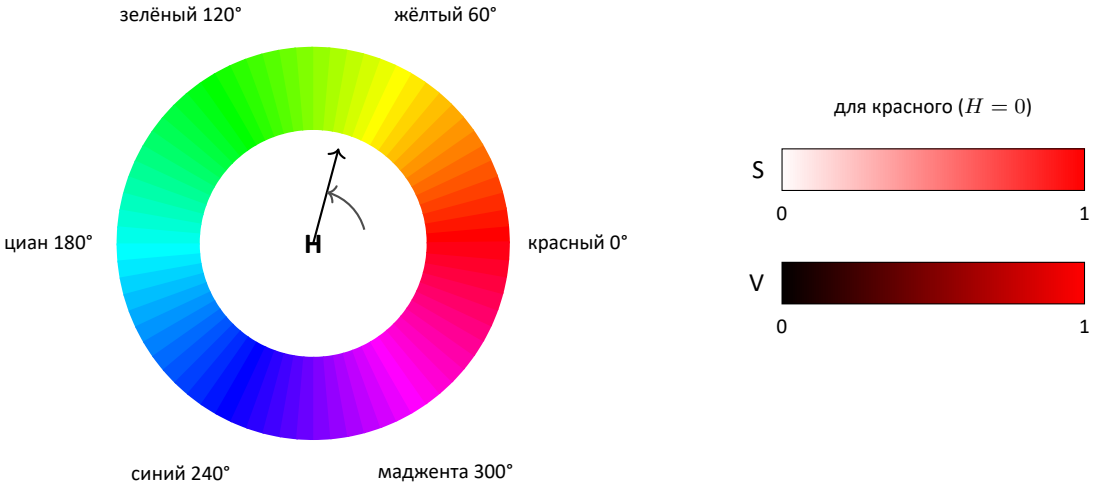


Рис. 11.8: Цветовой круг HSV: оттенок — угол, насыщенность и яркость — две шкалы

Светодиоду нужны R, G, B, поэтому надо пересчитать. Идея: круг делят на 6 секторов по 60°; в каждом секторе один канал максимальный, один минимальный, а третий плавно растёт или падает (функция `hsv_to_rgb` в примере ниже). Результат: красный → жёлтый → зелёный → циан → синий → мажента → красный.

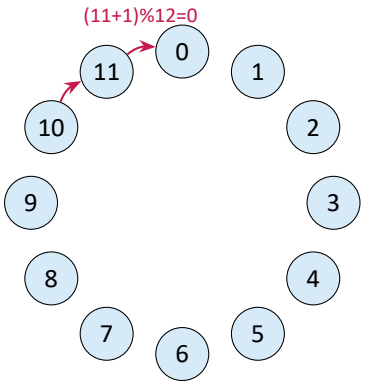
11.8 Учимся программировать: функции и остаток от деления

В примере с радугой две новых идеи.

1. **Функция** `hsv_to_rgb(h, s, v)` принимает числа и возвращает кортеж из трёх. Благодаря ей главный цикл остаётся коротким и читается как предложение: «цвет пикселя i —

оттенок h с насыщенностью 1».

2. **Остаток от деления** `%` превращает прямую ленту в *кольцо*: после пикселя с номером $N-1$ идёт пиксель 0, ведь $(N-1+1) \bmod N = 0$. Тот же приём возвращает оттенок $h > 1$ обратно в диапазон 0–1.

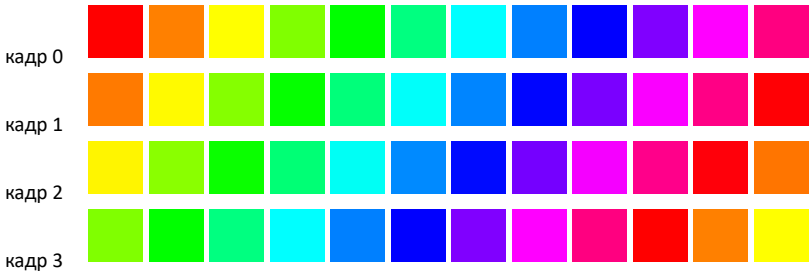


`pos = (pos + 1) % N`
индекс всегда в пределах $0 \ldots N-1$,
счёт замыкается в кольцо

Рис. 11.9: Остаток от деления замыкает номера пикселей в кольцо

11.9 Радуга

Каждому пикселю достаётся свой оттенок: $h_i = i/N$, и чтобы радуга двигалась, ко всем оттенкам прибавляют общий сдвиг, который с каждым кадром растёт.



$h_i = (i/N + \text{сдвиг}) \bmod 1$; сдвиг растёт с каждым кадром

Рис. 11.10: Движение радуги: те же оттенки смещаются вдоль ленты от кадра к кадру

Листинг 11.3: MicroPython: бегущая радуга через HSV

```
from machine import Pin
import neopixel
import time

N = 12
pixels = neopixel.NeoPixel(Pin(22), N)
BRIGHT = 0.2                        # яркость задаём через V (value) в HSV


def hsv_to_rgb(h, s, v):
    """h, s, v в диапазоне 0..1 -> кортеж (R, G, B) в диапазоне 0..255"""
    sector = int(h * 6)             # номер сектора цветового круга 0..5
    f = h * 6 - sector              # положение внутри сектора 0..1
    p = v * (1 - s)
    q = v * (1 - f * s)
```

```
    t = v * (1 - (1 - f) * s)
    r, g, b = ((v, t, p), (q, v, p), (p, v, t),
               (p, q, v), (t, p, v), (v, p, q))[sector % 6]
    return int(r * 255), int(g * 255), int(b * 255)


offset = 0.0                        # сдвиг оттенка: он «крутит» радугу
while True:
    for i in range(N):
        hue = (i / N + offset) % 1.0
        pixels[i] = hsv_to_rgb(hue, 1.0, BRIGHT)
    pixels.write()
    offset = (offset + 0.01) % 1.0
    time.sleep_ms(20)
```

Листинг 11.4: CircuitPython: то же; рядом — готовая функция `colorwheel`

```
import time
import board
import neopixel
from rainbowio import colorwheel      # встроенный модуль CircuitPython

N = 12
pixels = neopixel.NeoPixel(board.GP22, N, brightness=0.2, auto_write=False)


def hsv_to_rgb(h, s, v):
    """h, s, v в диапазоне 0..1 -> кортеж (R, G, B) в диапазоне 0..255"""
    sector = int(h * 6)
    f = h * 6 - sector
    p = v * (1 - s)
    q = v * (1 - f * s)
    t = v * (1 - (1 - f) * s)
    r, g, b = ((v, t, p), (q, v, p), (p, v, t),
               (p, q, v), (t, p, v), (v, p, q))[sector % 6]
    return int(r * 255), int(g * 255), int(b * 255)


offset = 0.0
while True:
    for i in range(N):
        hue = (i / N + offset) % 1.0
        pixels[i] = hsv_to_rgb(hue, 1.0, 1.0)   # яркость задаёт brightness
        # то же самое готовой функцией: pixels[i] = colorwheel(int(hue * 255))
    pixels.show()
    offset = (offset + 0.01) % 1.0
    time.sleep(0.02)
```

Обрати внимание на разницу в способе приглушить свет. В MicroPython мы подставляем `BRIGHT` в канал V, в CircuitPython ставим V= 1, а яркость уменьшает `brightness=0.2` у самой ленты. `rainbowio.colorwheel(pos)` — встроенная в CircuitPython функция: принимает число 0–255 и возвращает цвет с радужной окраской (в MicroPython её нет).

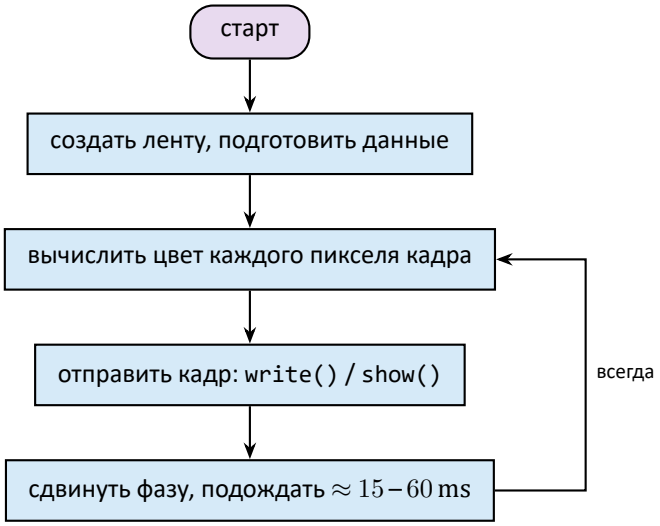


Рис. 11.11: Общая блок-схема любой анимации: вычислить кадр, отправить, подождать

11.10 Бегущая «комета»

Эффект «комета»: голова ярко горит, а за ней тянется хвост, затухающий с каждым кадром. Для этого храним в списке `level` яркость каждой ячейки и на каждом кадре умножаем её на число меньше единицы (например, 0,75): через несколько кадров яркость падает экспоненциально, как у остывающего металла.

Листинг 11.5: MicroPython: комета

```
from machine import Pin
import neopixel
import time

N = 12
pixels = neopixel.NeoPixel(Pin(22), N)
COLOR = (0, 60, 60)                 # голубая «комета»
FADE = 0.75                         # во сколько раз тускнеет хвост за кадр

level = [0.0] * N                   # яркость каждого пикселя, 0.0 ... 1.0
pos = 0                             # где сейчас голова кометы

while True:
    for i in range(N):
        level[i] *= FADE            # хвост затухает
    level[pos] = 1.0                # голова горит ярко
    for i in range(N):
        k = level[i]
        pixels[i] = (int(COLOR[0] * k), int(COLOR[1] * k), int(COLOR[2] * k))
    pixels.write()
    pos = (pos + 1) % N             # после последнего пикселя снова нулевой
    time.sleep_ms(60)
```

Листинг 11.6: CircuitPython: комета

```
import time
import board
import neopixel

N = 12
pixels = neopixel.NeoPixel(board.GP22, N, brightness=0.25, auto_write=False)
COLOR = (0, 255, 255)               # голубая «комета»
FADE = 0.75                         # во сколько раз тускнеет хвост за кадр

level = [0.0] * N                   # яркость каждого пикселя, 0.0 ... 1.0
pos = 0                             # где сейчас голова кометы

while True:
    for i in range(N):
        level[i] *= FADE            # хвост затухает
    level[pos] = 1.0                # голова горит ярко
    for i in range(N):
        k = level[i]
        pixels[i] = (int(COLOR[0] * k), int(COLOR[1] * k), int(COLOR[2] * k))
    pixels.show()
    pos = (pos + 1) % N             # после последнего пикселя снова нулевой
    time.sleep(0.06)
```

11.11 Яркость и гамма-коррекция

Глаз воспринимает яркость нелинейно: разница между значениями 10 и 20 кажется очень заметной, а между 200 и 210 — почти невидимой. Поэтому, если сделать «дыхание» просто линейным $(0, 1, 2, \ldots, 255)$, светодиод будет *долго* казаться яркой, а затем резко гаснуть. Исправляют это **гамма-коррекцией**: значение возводят в степень $\gamma \approx 2{,}2$–$2{,}8$:

$$v_{\text{вых}} = 255 \cdot \left(\frac{v_{\text{вх}}}{255}\right)^{\gamma}.$$

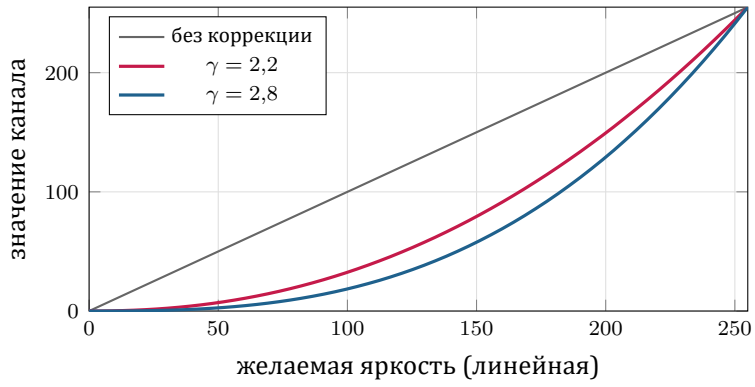


Рис. 11.12: Гамма-кривая: чтобы ощущаемая яркость росла равномерно, значение канала растёт медленнее вначале и быстрее в конце

Считать степень на каждом кадре дорого, поэтому программисты заранее строят **таблицу** из 256 значений (см. список `GAMMA` в примере ниже) и просто берут из неё элемент по индексу. Есть и обратная сторона: при 8 битах на канал в самом начале шкалы остаётся мало ступеней (значения 0, 1 и 2 после коррекции дают 0, 0 и 0), и очень тусклый свет «ступенчатый». Это известное ограничение 8-битных светодиодов.

11.12 Эффект «дыхание»

Яркость меняется по формуле $\frac{1-\cos(2\pi t/T)}{2}$: это плавный «вдох-выдох» с периодом T. Затем значение проходит гамма-таблицу и умножается на потолок яркости.

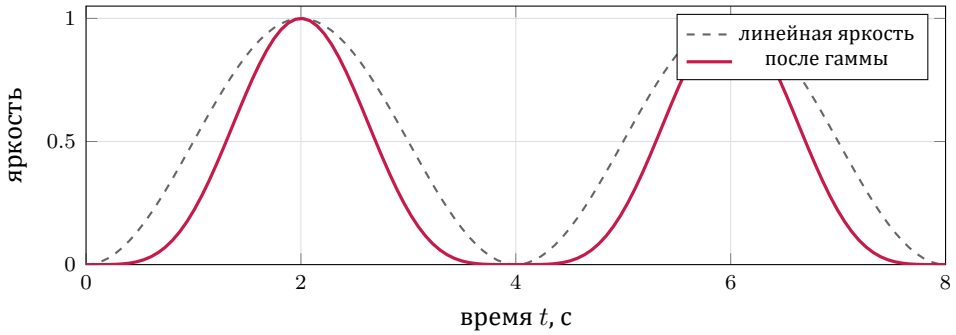


Рис. 11.13: Закон дыхания с периодом 4 с до и после гамма-коррекции

Листинг 11.7: MicroPython: дыхание с гамма-таблицей

```
from machine import Pin
import neopixel
import math
import time

N = 12
pixels = neopixel.NeoPixel(Pin(22), N)
PERIOD = 4.0                        # длительность одного «вдоха-выдоха», с
BRIGHT = 80                         # максимальное значение канала (из 255)

# таблица гамма-коррекции: 256 значений, показатель 2,2
GAMMA = [int((i / 255) ** 2.2 * 255 + 0.5) for i in range(256)]

t0 = time.ticks_ms()
while True:
    t = time.ticks_diff(time.ticks_ms(), t0) / 1000
    level = (1 - math.cos(2 * math.pi * t / PERIOD)) / 2   # плавно 0 -> 1 -> 0
    v = GAMMA[int(level * 255)] * BRIGHT // 255            # гамма, затем потолок
    pixels.fill((v, v // 3, 0))                            # тёплый оранжевый
    pixels.write()
    time.sleep_ms(15)
```

Листинг 11.8: CircuitPython: то же

```
import math
import time
import board
import neopixel

N = 12
pixels = neopixel.NeoPixel(board.GP22, N, brightness=1.0, auto_write=False)
PERIOD = 4.0                        # длительность одного «вдоха-выдоха», с
BRIGHT = 80                         # максимальное значение канала (из 255)

# таблица гамма-коррекции: 256 значений, показатель 2,2
GAMMA = [int((i / 255) ** 2.2 * 255 + 0.5) for i in range(256)]

```

```
t0 = time.monotonic()
while True:
    t = time.monotonic() - t0
    level = (1 - math.cos(2 * math.pi * t / PERIOD)) / 2   # плавно 0 -> 1 -> 0
    v = GAMMA[int(level * 255)] * BRIGHT // 255            # гамма, затем потолок
    pixels.fill((v, v // 3, 0))                            # тёплый оранжевый
    pixels.show()
    time.sleep(0.015)
```

Время мы берём из часов (`ticks_ms()` / `monotonic()`), а не считаем кадры: так дыхание остаётся ровным, даже если кадр рассчитывается то быстрее, то медленнее.

11.13 Эффект «огонь»

Алгоритм «Fire2012» — классика ленточной анимации. Лента — это вертикальный «столб» ячеек, у каждой есть температура. Четыре шага на каждом кадре:

1. **остывание**: каждая ячейка теряет случайный кусочек тепла;
2. **подъём**: тепло течёт вверх, усредняясь с нижними соседями;
3. **искры**: у основания (первые три ячейки) случайно добавляется тепло;
4. **цвет**: температура переводится в цвет по «чёрное → красное → жёлтое → белое».

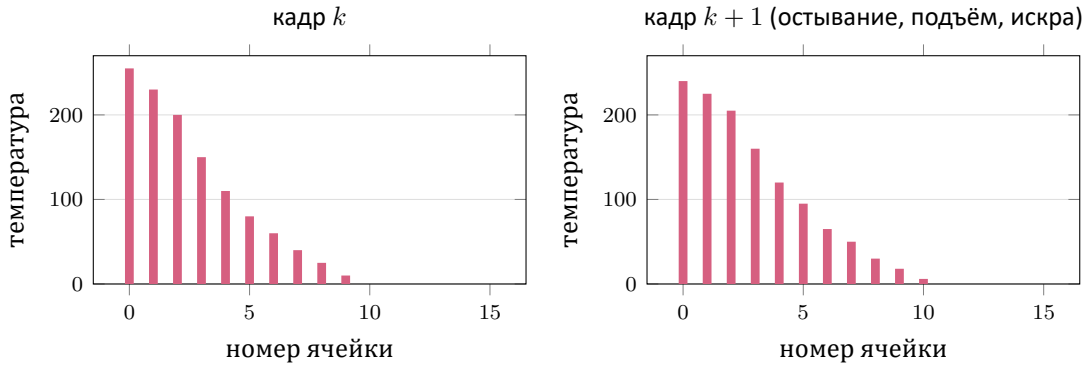


Рис. 11.14: «Температурный профиль» ленты в соседних кадрах: тепло поднимается и остывает



Рис. 11.15: Шкала «температура → цвет», используемая функцией `heat_color`

Листинг 11.9: MicroPython: огонь

```
from machine import Pin
import neopixel
import random
import time

N = 16                              # лента из 16 светодиодов, основание огня --- пиксель 0
pixels = neopixel.NeoPixel(Pin(22), N)
heat = [0] * N                      # «температура» каждой ячейки, 0..255
COOLING = 25                        # насколько ячейка остывает за кадр
```

```
SPARK = 120                           # вероятность искры (из 255)
BR = 60                               # потолок яркости (из 255)


def heat_color(h):
    """температура 0..255 -> цвет: чёрный, красный, жёлтый, белый"""
    if h < 85:
        r, g, b = h * 3, 0, 0
    elif h < 170:
        r, g, b = 255, (h - 85) * 3, 0
    else:
        r, g, b = 255, 255, (h - 170) * 3
    return r * BR // 255, g * BR // 255, b * BR // 255


while True:
    for i in range(N):                                  # 1. остывание
        heat[i] = max(0, heat[i] - random.randint(0, COOLING))
    for i in range(N - 1, 1, -1):                       # 2. тепло поднимается вверх
        heat[i] = (heat[i - 1] + heat[i - 2] + heat[i - 2]) // 3
    if random.randint(0, 255) < SPARK:                  # 3. новая искра у основания
        y = random.randint(0, 2)
        heat[y] = min(255, heat[y] + random.randint(160, 255))
    for i in range(N):                                  # 4. температура -> цвет
        pixels[i] = heat_color(heat[i])
    pixels.write()
    time.sleep_ms(30)
```

Листинг 11.10: CircuitPython: огонь

```
import random
import time
import board
import neopixel

N = 16                                # лента из 16 светодиодов, основание огня --- пиксель 0
pixels = neopixel.NeoPixel(board.GP22, N, brightness=0.25, auto_write=False)
heat = [0] * N                        # «температура» каждой ячейки, 0..255
COOLING = 25                          # насколько ячейка остывает за кадр
SPARK = 120                           # вероятность искры (из 255)


def heat_color(h):
    """температура 0..255 -> цвет: чёрный, красный, жёлтый, белый"""
    if h < 85:
        return (h * 3, 0, 0)
    if h < 170:
        return (255, (h - 85) * 3, 0)
    return (255, 255, (h - 170) * 3)


while True:
    for i in range(N):                                  # 1. остывание
        heat[i] = max(0, heat[i] - random.randint(0, COOLING))
    for i in range(N - 1, 1, -1):                       # 2. тепло поднимается вверх
        heat[i] = (heat[i - 1] + heat[i - 2] + heat[i - 2]) // 3
    if random.randint(0, 255) < SPARK:                  # 3. новая искра у основания
        y = random.randint(0, 2)
```

```
        heat[y] = min(255, heat[y] + random.randint(160, 255))
    for i in range(N):                                  # 4. температура -> цвет
        pixels[i] = heat_color(heat[i])
    pixels.show()
    time.sleep(0.03)
```

11.14 Готовые анимации в CircuitPython

Для **CircuitPython** есть библиотека `adafruit_led_animation` (ставится через `circup install adafruit_led_animation`). Она содержит десятки готовых эффектов (`Rainbow`, `Pulse`, `Comet`, `Chase`, `Sparkle`, `Blink`…) и умеет переключать их по таймеру. Важное преимущество: вызов `animate()` *не блокирует* программу, поэтому между его вызовами можно опрашивать кнопки и датчики.

Листинг 11.11: CircuitPython: последовательность готовых эффектов

```
import board
import neopixel
from adafruit_led_animation.animation.rainbow import Rainbow
from adafruit_led_animation.animation.pulse import Pulse
from adafruit_led_animation.animation.comet import Comet
from adafruit_led_animation.sequence import AnimationSequence
from adafruit_led_animation.color import RED, PURPLE

pixels = neopixel.NeoPixel(board.GP22, 12, brightness=0.2, auto_write=False)

rainbow = Rainbow(pixels, speed=0.02, period=4)                  # радуга
pulse = Pulse(pixels, speed=0.02, color=RED, period=3)           # дыхание
comet = Comet(pixels, speed=0.05, color=PURPLE, tail_length=6,   # комета с хвостом
              bounce=True)

# каждые 6 секунд эффекты переключаются по кругу
animations = AnimationSequence(rainbow, pulse, comet,
                               advance_interval=6, auto_clear=True)

while True:
    animations.animate()             # один шаг; сам решает, пора ли обновлять кадр
```

В **MicroPython** стандартной библиотеки анимаций нет; зато есть сообщество и простота: примеры выше — это ровно то, что делают внутри такие библиотеки. Для длинных лент и очень быстрых кадров в **MicroPython** используют реализацию на PIO: она не занимает процессор на время передачи (см. урок 14).

11.15 Самостоятельный опыт

Собери кольцо по схеме (рис. 11.6) и запусти по очереди все примеры. Для каждого эффекта запиши, как меняются параметры: что произойдёт, если изменить `FADE` в комете, `PERIOD` в дыхании, `COOLING` в огне? Попробуй менять яркость от 10 % до 100 % и измерь ток (если есть мультиметр) — сравни с рис. 11.4.

Если не работает

- **Ничего не светится** — проверь питание ленты (5V и GND), направление: данные подаются на **DIN**, а не DOUT; плата и лента должны иметь общую землю.
- **Светится только первый пиксель или всё белое/случайные цвета** — слабый или шумный сигнал. Добавь резистор $300 – 500\,\Omega$, укороти провод данных, поставь преобразова-

тель уровней 3,3 В→5 В.

- **Цвета перепутаны (красный вместо зелёного)** — у ленты другой порядок байтов (GRB/RGB/BRG); в CircuitPython задай `pixel_order`.
- **Плата перезагружается при включении яркого белого** — не хватает мощности блока питания или USB; уменьши яркость, добавь конденсатор, подключи отдельное питание.
- **Дальние пиксели жёлтые или тусклые** — падение напряжения на тонких проводах; подведи питание в нескольких точках ленты.
- **`OSError` или нет вывода в MicroPython** — проверь номер GP и версию прошивки: функция `machine.bitstream` входит в современные прошивки MicroPython для RP2.
- **Эффект тормозит** — слишком длинная лента или `auto_write=True` в CircuitPython: вычисляй кадр целиком и вызывай `show()` один раз.

Итоги урока

- WS2812 — цепочка пикселей с одним проводом данных; кадр — 24 бита GRB на пиксель, бит кодируется шириной импульса при 800 kHz, пауза > 50 µs закрывает кадр.
- Пиксель потребляет до 60 mA; питание — 5 V, общий GND, конденсатор 1000 µF, резистор 330 Ω в линии данных.
- Цвет удобно задавать в HSV: оттенок (угол), насыщенность, яркость; для светодиода пересчитывают в RGB.
- Яркость нелинейна, поэтому для плавных эффектов нужна гамма-коррекция (таблица из 256 значений).
- MicroPython: `neopixel.NeoPixel`, `write()`; CircuitPython: `neopixel.NeoPixel`, `brightness`, `show()`, библиотека `adafruit_led_animation`.

Проверь себя

1. Как WS2812 отличает бит «0» от бита «1», если у него нет тактового провода?
2. Сколько тока может потреблять 30 пикселей и какой блок питания ты выберешь?
3. Зачем нужны конденсатор и резистор в схеме подключения ленты?
4. Чем модель HSV удобнее RGB? Как получить радугу?
5. Зачем нужна гамма-коррекция и как её реализуют программно?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Собери кольцо и запусти пример с радугой. Измени число пикселей `N`, скорость сдвига и направление движения.
2. ★ Зажги каждый третий пиксель красным, а остальные оставь выключенными. Какой индекс нужно проверять?
3. ★★ Рассчитай ток для 16 пикселей, если светится белый на 25 % яркости, а затем — на 100 %. Хватит ли USB?
4. ★★ Измени «комету» так, чтобы две кометы бежали навстречу друг другу, а хвост одной

был красным, другой — синим.

5. ★★★ Сделай «светофор-часы»: пиксели кольца показывают секунды (один пиксель — одна пятая минуты), цвет зависит от значения. Используй `time.ticks_ms()` / `time.monotonic()`.

6. ★★★★ Сделай «индикатор громкости» (VU-метр): значение потенциометра на **GP26** (урок 6) определяет, сколько пикселей горит, а цвет меняется от зелёного к красному; добавь «пик» — один пиксель, медленно опускающийся вниз.

Урок 12. Память и файлы

Цель	понять, какие бывают виды памяти в Pico, научиться безопасно хранить настройки и журналы во flash и на SD-карте и следить за оперативной памятью
Опирается на	урок 2–4: основы программы; словари и списки Python; урок 6: чтение датчика
Результат	программа читает и пишет файлы, хранит настройки в JSON, ведёт журнал температуры в CSV, а в CircuitPython ты разрешаешь запись кнопкой
Понадобится	кнопка и провода для опыта с `boot.py`; по желанию — модуль SD-карты (урок 9)

12.1 Идея урока коротко

Переменные в программе живут, пока плата включена: стоит отключить питание — все значения исчезнут. Чтобы устройство «помнило» настройки, счётчики и измерения, данные записывают в **файл**. Файлы лежат во flash-памяти платы (как на флешке) или на SD-карте. Работать с файлами в Python очень просто (`open`, `read`, `write`), но у flash есть особенности: у неё ограничено число перезаписей, а в CircuitPython диск ещё и принадлежит компьютеру, поэтому программе записывать файлы нельзя, пока это явно не разрешено.

12.2 Какая память есть в Pico

Разные виды памяти различаются скоростью, ёмкостью и тем, сохраняются ли данные без питания.

Память	Объём (Pico)	После выключения	Для чего служит
SRAM	264 КБ (6 банков)	исчезает	переменные, стек, «куча» Python, буферы
Boot ROM	16 КБ	сохраняется	начальный загрузчик, режим USB-диска BOOTSEL; не стирается
QSPI flash	2 МБ (Pico, Pico W)	сохраняется	прошивка и файлы пользователя
Регистры	—	—	управление периферией

Таблица 12.1: Основные виды памяти RP2040

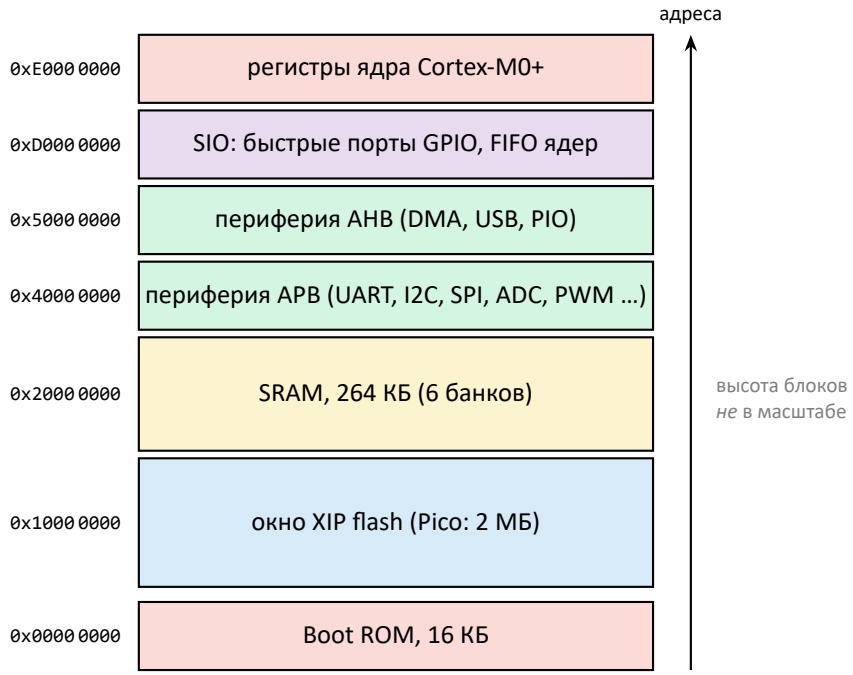


Рис. 12.1: Карта адресов RP2040: все виды памяти и периферия лежат в одном адресном пространстве

Из карты видно главное: процессор обращается к flash *как к обычной памяти* по адресам с `0x1000 0000`. Эта технология называется **XIP** (eXecute In Place, «выполнение на месте»): команды читаются прямо из микросхемы flash через небольшой кэш, а в SRAM копировать программу не нужно. Поэтому для прошивки хватает 264 КБ оперативной памяти, а остальное занимает код во flash.

12.3 Как разделена flash-память

На плате Pico стоит микросхема QSPI flash на 2 МБ (в Pico 2 — 4 МБ). В ней рядом лежат прошивка (интерпретатор **MicroPython** или **CircuitPython**) и **файловая система** с твоими программами и данными. Точные размеры зависят от прошивки и платы, поэтому ориентируйся на то, что показывает сама плата (`os.statvfs`, пример ниже).

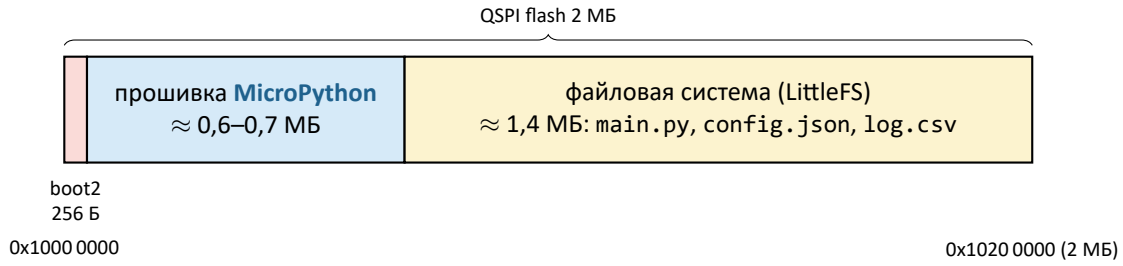


Рис. 12.2: Раздел flash в **MicroPython** на Pico: первые 256 байт — второй загрузчик, затем прошивка и файловая система

Это интересно

В **CircuitPython** диск `CIRCUITPY` занимает примерно 0,8–1 МБ и использует файловую систему FAT (её понимает любой компьютер). В **MicroPython** — LittleFS: она предназначена для микроконтроллеров, устойчива к внезапному отключению питания и сама выравнивает износ ячеек. Размеры зависят от версии прошивки, поэтому лучше спросить у платы.

12.4 Почему flash «изнашивается»

Ячейки flash нельзя записать по одной. Запись происходит в два шага:

- **стирание** — сразу целый *сектор* 4 КБ ставится в состояние «все единицы»; это медленно (десятки миллисекунд);
- **запись (программирование)** — можно превратить единицы в нули, по *странице* до 256 байт.

Вернуть нуль в единицу можно только стиранием всего сектора. Поэтому даже «изменить один байт» — это стереть и переписать сектор. Каждая ячейка выдерживает около 10^5 (сто тысяч) циклов стирания, после чего перестаёт надёжно хранить заряд.



Рис. 12.3: Изменение данных во flash: стирание сектора, затем запись страниц

Простой расчёт. Пусть программа раз в τ секунд записывает файл в *один и тот же* сектор. За год это $3{,}15 \cdot 10^7/\tau$ записей, а ресурс — 10^5 циклов, значит срок жизни сектора

$$T \approx \frac{10^5 \cdot \tau}{3{,}15 \cdot 10^7} \text{ лет} \approx 0{,}003\,\tau \text{ лет}.$$

При $\tau = 1$ с получается примерно *28 часов*! При $\tau = 10$ мин — около 5 лет.

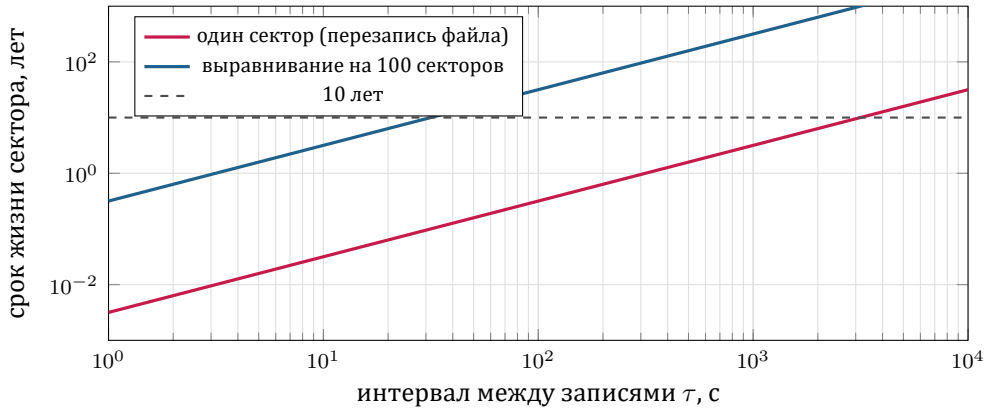


Рис. 12.4: Срок жизни flash при циклической перезаписи: чем реже пишем и чем больше секторов задействовано, тем дольше служит память

Стратегии бережной записи

1. **Пиши реже.** Накапливай данные в оперативной памяти и записывай пачкой (пример журнала ниже).
2. **Пиши только изменения.** Настройки сохраняй, когда пользователь их *изменил*, а не каждую секунду.

3. **Добавляй, а не переписывай.** Режим `'a'` (append) для журнала дописывает в конец; LittleFS распределяет записи по разным блокам.
4. **Ограничивай размер.** Журнал, который растёт вечно, рано или поздно заполнит диск: ограничь размер и «вращай» файлы (`log.csv` → `log.csv.old`).
5. **Для больших и частых данных — SD-карта.** Её контроллер сам выравнивает износ, а замена карты стоит дёшево.
6. **Не бойся читать.** Чтение flash износа не вызывает.

Осторожно

Во время стирания и записи flash чип недоступен для чтения, поэтому программа (и второе ядро, и прерывания) на несколько миллисекунд *замирает*. Не записывай файлы в моменты, когда нужна точная реакция: например, при управлении двигателем или при считывании быстрого сигнала.

12.5 Файловая система LittleFS

Между программой и микросхемой работает несколько слоёв. Программа вызывает `open()` — слой VFS находит нужную файловую систему — LittleFS управляет блоками — драйвер flash стирает и записывает сектора.

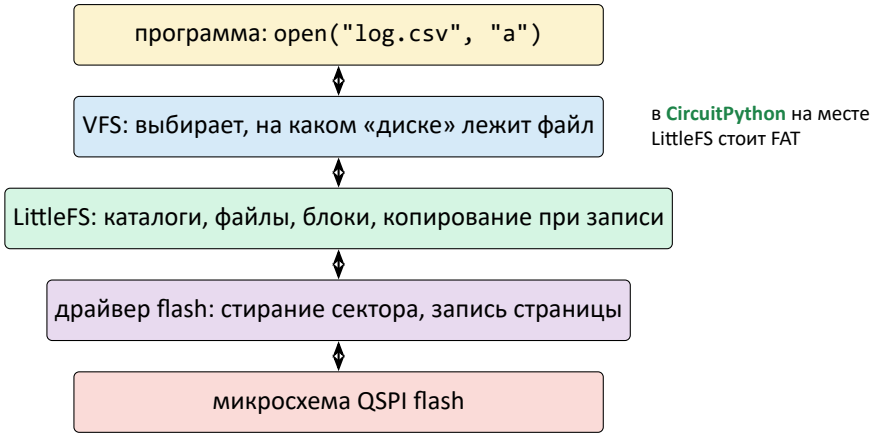


Рис. 12.5: Слои хранения данных: от функции `open()` до микросхемы

Главная идея LittleFS — **копирование при записи** (copy-on-write): новая версия блока пишется в *новое* место, и только когда запись завершена, метаданные «переключаются» на неё. Если питание пропало на середине операции, остаётся предыдущая целая версия, а не «полуфайл». Именно поэтому её выбрали для микроконтроллеров. Тем не менее файл, который ты *не закрыл* (`close()`), может потерять последние данные, поэтому используй `with`.

12.6 Работа с файлами в Python

Основные операции описаны в таблице; они одинаковы в обеих системах.

Операция	Что делает
`open(name, "w")`	создать или очистить файл и записывать
`open(name, "a")`	дописывать в конец (append)
`open(name)` или `"r"`	читать (по умолчанию текст)
`f.read()`, `f.readline()`	прочитать всё / одну строку
`f.write(s)`	записать строку (нужно самому добавлять `\n`)
`with open(...) as f:`	автоматически закрывает файл
`os.listdir()`, `os.remove()`, `os.rename()`	список файлов, удаление, переименование
`os.stat(name)[6]`	размер файла в байтах
`os.statvfs("/")`	размер блока и число свободных блоков

Таблица 12.2: Основные операции с файлами

Листинг 12.1: MicroPython: чтение, запись, дозапись, размер, удаление

```
import os

# --- запись: 'w' создаёт файл заново (старое содержимое пропадает)
with open("hello.txt", "w") as f:
    f.write("Привет, Pico!\n")
    f.write("Вторая строка\n")

# --- дозапись в конец: 'a' (append)
with open("hello.txt", "a") as f:
    f.write("Третья строка\n")

# --- чтение целиком и построчно
with open("hello.txt") as f:
    text = f.read()
print(text)

with open("hello.txt") as f:
    for n, line in enumerate(f, 1):
        print(n, line.strip())

# --- каталог, размер, удаление
print(os.listdir())                     # список файлов в корне
print(os.stat("hello.txt")[6])          # размер файла в байтах
st = os.statvfs("/")                    # сведения о файловой системе
print("свободно, КБ:", st[0] * st[3] // 1024)   # размер блока х число свободных блоков
os.rename("hello.txt", "old.txt")
os.remove("old.txt")
```

Листинг 12.2: CircuitPython: те же операции (но запись заблокирована)

```
import os

# По умолчанию диск CIRCUITPY открыт для записи ТОЛЬКО компьютеру.
# Из программы запись вызовет OSError: [Errno 30] Read-only filesystem
try:
    with open("hello.txt", "w") as f:
        f.write("Привет, Pico!\n")
    print("запись удалась")
```

```
except OSError as e:
    print("запись запрещена:", e)

# Чтение работает всегда
print(os.listdir("/"))                  # список файлов в корне
with open("code.py") as f:
    print(f.readline())                 # первая строка самой программы

st = os.statvfs("/")
print("свободно, КБ:", st[0] * st[3] // 1024)   # размер блока х число свободных блоков
# Чтобы разрешить запись из программы, нужен boot.py (см. ниже в уроке)
```

Если запустить **CircuitPython**-вариант на свежей плате, строка «запись запрещена» — не ошибка плат, а защита: почему так, объясним ниже.

12.7 Учимся программировать: контекстный менеджер `with`

Файл обязательно нужно закрыть: только при закрытии буфер сбрасывается на диск. Забыть `f.close()` легко, особенно если между `open` и `close` случится ошибка. Конструкция `with open(...) as f:` решает проблему: когда блок кончается (в том числе из-за ошибки), файл закрывается автоматически. Это пример **контекстного менеджера**. Тот же приём пригодится для блокировок (урок 13), соединений и других ресурсов, которые надо вернуть.

12.8 Форматы данных: текст, JSON и CSV

Файл — это просто последовательность байтов, поэтому данные нужно записать в понятном виде. Два самых полезных формата:

- **JSON** — вложенные структуры «словарь/список/число/строка/логическое»; идеально для *настроек*.
- **CSV** — таблица, строки через перевод строки, значения через запятую; идеально для *журналов измерений* (открывается в Excel).

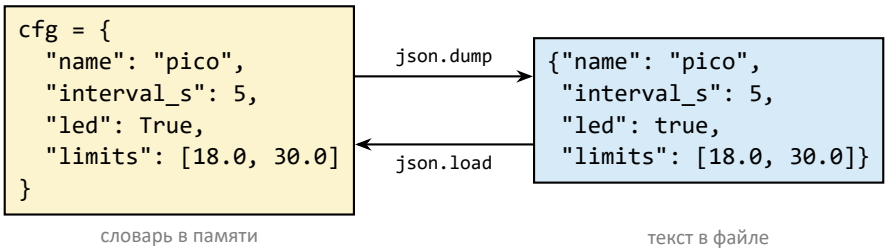


Рис. 12.6: JSON: словарь Python превращается в текст и обратно; `True` в JSON пишется как `true`

Нужно знать три правила: JSON не хранит кортежи (они станут списками), ключи словаря обязаны быть строками, а повреждённый файл вызывает ошибку `ValueError` — её надо ловить и брать значения по умолчанию.

12.9 Хранение настроек

Настройки читают при старте и записывают только при изменении. Чтобы устройство не ломалось из-за испорченного файла, пишут **атомарно**: сначала во временный файл, затем переименовывают. Переименование в LittleFS выполняется целиком, либо не происходит вовсе.

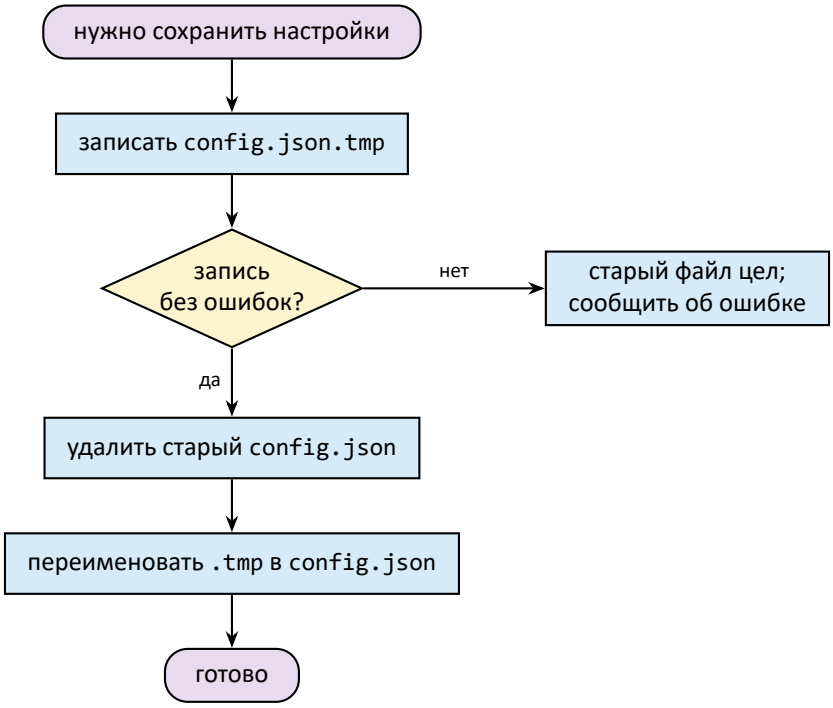


Рис. 12.7: Атомарное сохранение: старая версия остаётся целой, пока новая не записана полностью

Листинг 12.3: MicroPython: загрузка и сохранение настроек в JSON

```
import json
import os

CONFIG = "config.json"
DEFAULTS = {"name": "pico", "interval_s": 5, "led": True, "limits": [18.0, 30.0]}


def load_config():
    """Прочитать настройки; если файла нет или он испорчен --- взять значения по умолчанию"""
    try:
        with open(CONFIG) as f:
            data = json.load(f)
    except (OSError, ValueError):       # нет файла / ошибка в JSON
        data = {}
    cfg = dict(DEFAULTS)                # копия значений по умолчанию
    cfg.update(data)                    # поверх них --- то, что сохранено
    return cfg


def save_config(cfg):
    """Безопасная запись: сначала во временный файл, затем переименование"""
    tmp = CONFIG + ".tmp"
    with open(tmp, "w") as f:
        json.dump(cfg, f)
    try:
        os.remove(CONFIG)
    except OSError:
        pass
    os.rename(tmp, CONFIG)


cfg = load_config()
```

```
print("интервал:", cfg["interval_s"], "с, пределы:", cfg["limits"])
cfg["interval_s"] += 1                  # меняем настройку ...
save_config(cfg)                        # ... и сохраняем
print(json.dumps(cfg))                  # как выглядит текст JSON
```

Листинг 12.4: CircuitPython: то же; запись требует разрешения

```
import json
import os

# Запись работает только если boot.py сделал storage.remount("/", readonly=False)
CONFIG = "/config.json"
DEFAULTS = {"name": "pico", "interval_s": 5, "led": True, "limits": [18.0, 30.0]}


def load_config():
    """Прочитать настройки; если файла нет или он испорчен --- значения по умолчанию"""
    try:
        with open(CONFIG) as f:
            data = json.load(f)
    except (OSError, ValueError):       # нет файла / ошибка в JSON
        data = {}
    cfg = dict(DEFAULTS)
    cfg.update(data)
    return cfg


def save_config(cfg):
    """Безопасная запись: сначала во временный файл, затем переименование"""
    tmp = CONFIG + ".tmp"
    with open(tmp, "w") as f:
        json.dump(cfg, f)
        f.flush()
        os.sync()                       # дождаться записи во flash
    try:
        os.remove(CONFIG)
    except OSError:
        pass
    os.rename(tmp, CONFIG)


cfg = load_config()
print("интервал:", cfg["interval_s"], "с, пределы:", cfg["limits"])
cfg["interval_s"] += 1
try:
    save_config(cfg)
except OSError as e:
    print("диск только для чтения, нужен boot.py:", e)
print(json.dumps(cfg))
```

12.10 Журнал измерений с временем

В Pico нет батарейки часов: после включения часы начинаются с условной даты. Поэтому в журнале удобно хранить *время от запуска* в секундах (а точное время можно получить по сети, см. урок 16). Мы записываем температуру самого чипа (встроенный датчик, урок 6). Строки копим в списке и записываем пачкой; размер файла ограничен.

Листинг 12.5: MicroPython: журнал температуры в CSV с ограничением размера

```
from machine import ADC
import os
import time

adc_t = ADC(4)                          # канал 4 --- встроенный датчик температуры чипа
LOG = "log.csv"
MAX_SIZE = 20_000                       # байт; больше --- начинаем новый файл
PERIOD_S = 10                           # как часто писать строку
FLUSH_EVERY = 6                         # сколько строк копить в памяти перед записью


def cpu_temp():
    volts = adc_t.read_u16() * 3.3 / 65535
    return 27 - (volts - 0.706) / 0.001721      # формула из документации RP2040


def rotate_if_big():
    """Если журнал разросся, переименовываем его и начинаем заново"""
    try:
        if os.stat(LOG)[6] > MAX_SIZE:
            try:
                os.remove(LOG + ".old")
            except OSError:
                pass
            os.rename(LOG, LOG + ".old")
    except OSError:
        pass                            # файла ещё нет


if LOG not in os.listdir():
    with open(LOG, "w") as f:
        f.write("t_s,temp_c\n")         # заголовок CSV

buffer = []
t0 = time.ticks_ms()
while True:
    t = time.ticks_diff(time.ticks_ms(), t0) // 1000    # секунд с запуска
    buffer.append("{},{:.1f}\n".format(t, cpu_temp()))
    if len(buffer) >= FLUSH_EVERY:      # пишем пачкой: меньше циклов стирания
        rotate_if_big()
        with open(LOG, "a") as f:
            f.write("".join(buffer))
        buffer = []
    time.sleep(PERIOD_S)
```

Листинг 12.6: CircuitPython: то же

```
import os
import time
import microcontroller            # встроенный датчик температуры чипа

LOG = "/log.csv"
MAX_SIZE = 20000                  # байт; больше --- начинаем новый файл
PERIOD_S = 10                     # как часто писать строку
FLUSH_EVERY = 6                   # сколько строк копить в памяти перед записью
```

```
def rotate_if_big():
    """Если журнал разросся, переименовываем его и начинаем заново"""
    try:
        if os.stat(LOG)[6] > MAX_SIZE:
            try:
                os.remove(LOG + ".old")
            except OSError:
                pass
            os.rename(LOG, LOG + ".old")
    except OSError:
        pass                        # файла ещё нет


try:
    names = os.listdir("/")
    if "log.csv" not in names:
        with open(LOG, "w") as f:
            f.write("t_s,temp_c\n")
except OSError:
    print("диск только для чтения: нужен boot.py с storage.remount")

buffer = []
t0 = time.monotonic()
while True:
    t = int(time.monotonic() - t0)
    buffer.append("{},{:.1f}\n".format(t, microcontroller.cpu.temperature))
    if len(buffer) >= FLUSH_EVERY:
        try:
            rotate_if_big()
            with open(LOG, "a") as f:
                f.write("".join(buffer))
            buffer = []
        except OSError:
            print("запись не удалась, держу данные в памяти")
            buffer = buffer[-60:]     # не даём буферу расти бесконечно
    time.sleep(PERIOD_S)
```

Формула датчика **MicroPython**: $T = 27 - \dfrac{U - 0{,}706}{0{,}001721}$, где U — напряжение с канала 4 АЦП в вольтах. В **CircuitPython** готовое значение лежит в `microcontroller.cpu.temperature`. Чип нагревается сам, поэтому цифры будут выше температуры воздуха на несколько градусов.

12.11 CircuitPython: диск принадлежит компьютеру

Это главное отличие двух систем. Когда плата с **CircuitPython** подключена по USB, компьютер видит диск `CIRCUITPY` и записывает на него `code.py`. Если бы программа в то же время писала на тот же диск, файловая система была бы повреждена: два владельца ничего не знают друг о друге. Поэтому **CircuitPython** по умолчанию отдаёт запись *компьютеру*, а программе запрещает (`OSError: Read-only filesystem`). **MicroPython** работает иначе: у него нет автоматического USB-диска, файлы передают программой (Thonny, `mpremote`), поэтому писать из программы можно всегда.

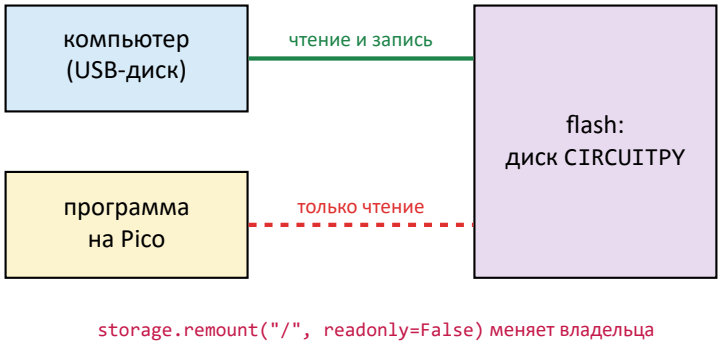


Рис. 12.8: По умолчанию в **CircuitPython** записывать на диск может только компьютер; программа его только читает

boot.py и перемычка выбора режима

Файл `boot.py` выполняется один раз при включении, *раньше*, чем USB отдаст диск компьютеру. В нём можно выбрать владельца диска вызовом `storage.remount("/", readonly=False)`: тогда программа сможет писать, а компьютер увидит диск только для чтения (редактировать `code.py` с компьютера станет нельзя!). Чтобы не остаться без возможности исправить программу, режим выбирают **физическим переключателем**: кнопкой или перемычкой.

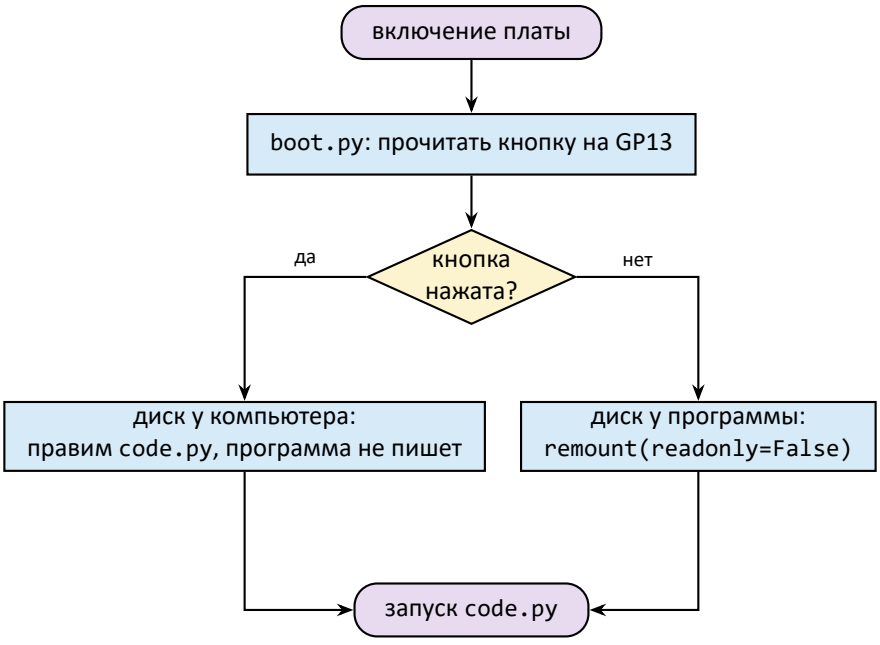
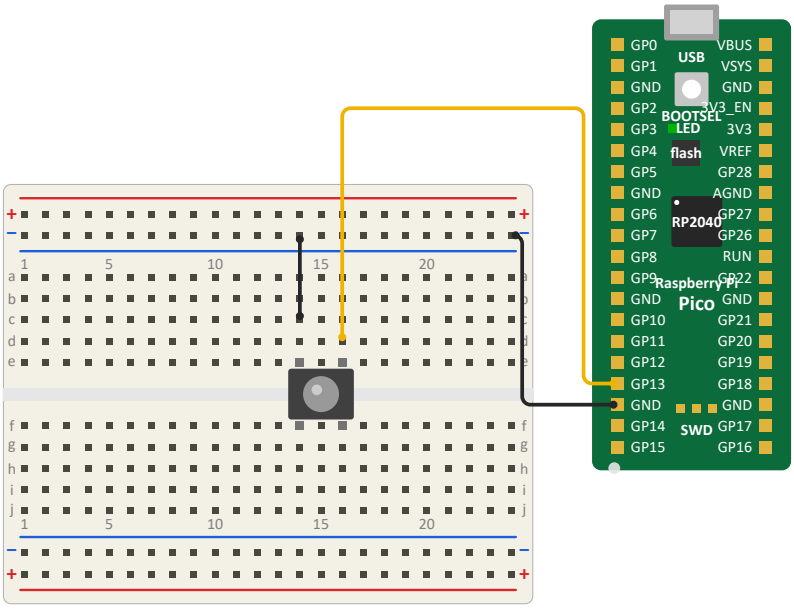
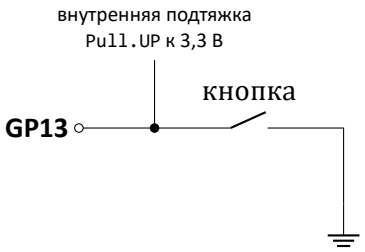


Рис. 12.9: Выбор режима в `boot.py`: нажатая кнопка оставляет диск компьютеру

Подключим кнопку между **GP13** и GND. Внутренняя подтяжка к питанию (`Pull.UP`) даёт на входе единицу, нажатие замыкает вывод на землю.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 12.10: Кнопка выбора режима диска: GP13 — кнопка — GND

Листинг 12.7: CircuitPython: файл `boot.py`: кнопка выбирает владельца диска

```
# boot.py --- выполняется один раз при включении, ДО того как USB отдаёт диск компьютеру
import board
import digitalio
import storage

# Кнопка на GP13 замыкает вывод на GND; без нажатия включена подтяжка к 3,3 В
switch = digitalio.DigitalInOut(board.GP13)
switch.switch_to_input(pull=digitalio.Pull.UP)

if switch.value:
    # кнопка НЕ нажата: запись разрешена программе, компьютер видит диск только для чтения
    storage.remount("/", readonly=False)
# кнопка нажата при включении: всё как обычно --- диск открыт компьютеру,
# можно редактировать code.py, а программа писать не сможет
```

Совет

Если случайно оставил диск в режиме «только чтение» для компьютера, нажми кнопку и перезагрузи плату: диск вернётся к компьютеру. Крайний случай — удалить `boot.py` через REPL (`import os; os.remove("boot.py")`) или в режиме BOOTSEL залить прошивку заново.

Если нужен простой способ *скрыть* диск от компьютера совсем, в `boot.py` можно вызвать `storage.disable_usb_drive()`; а для безопасности файловой системы после неожиданного от-

ключения питания у **CircuitPython** есть функция `storage.erase_filesystem()` — она стирает *все* файлы.

12.12 SD-карта

Когда flash мало или данные пишутся часто, используют SD-карту: она подключается к шине SPI (подробно в уроке 9), использует FAT и в **MicroPython**, и в **CircuitPython**. На шине четыре сигнала: SCK, MOSI, MISO и выбор устройства CS. Питание большинства модулей — 3,3 V (модули на 5 В имеют встроенный стабилизатор и преобразователи уровней; проверь маркировку).

Листинг 12.8: **MicroPython**: SD-карта через драйвер `sdcard`

```
from machine import Pin, SPI
import os
import sdcard                           # драйвер sdcard.py из micropython-lib (mip install
↪  sdcard)

# SPI0: SCK = GP18, MOSI = GP19, MISO = GP16, CS = GP17
spi = SPI(0, baudrate=1_000_000, sck=Pin(18), mosi=Pin(19), miso=Pin(16))
sd = sdcard.SDCard(spi, Pin(17))
os.mount(sd, "/sd")                     # карта FAT появляется как каталог /sd
print(os.listdir("/sd"))

with open("/sd/log.txt", "a") as f:     # дальше --- обычная работа с файлами
    f.write("запись на SD-карту\n")

os.umount("/sd")                        # перед извлечением карты отмонтируем
```

Листинг 12.9: **CircuitPython**: SD-карта через `adafruit_sdcard`

```
import os
import board
import busio
import digitalio
import storage
import adafruit_sdcard                # библиотека из бандла (circup install adafruit_sdcard)

# SPI0: SCK = GP18, MOSI = GP19, MISO = GP16, CS = GP17
spi = busio.SPI(board.GP18, MOSI=board.GP19, MISO=board.GP16)
cs = digitalio.DigitalInOut(board.GP17)
sdcard = adafruit_sdcard.SDCard(spi, cs)
vfs = storage.VfsFat(sdcard)
storage.mount(vfs, "/sd")               # карта появляется как каталог /sd
print(os.listdir("/sd"))

with open("/sd/log.txt", "a") as f:     # на SD запись разрешена всегда
    f.write("запись на SD-карту\n")

storage.umount(vfs)                     # перед извлечением карты отмонтируем
```

Драйвер `sdcard.py` в **MicroPython** ставится командой `mpremote mip install sdcard` (в старых версиях файл копируют из репозитория micropython-lib вручную). В **CircuitPython** нужна библиотека `adafruit_sdcard`. Перед извлечением карты файл надо закрыть, а карту — отмонтировать, иначе можно потерять последние записи.

12.13 Оперативная память и сборка мусора

В отличие от flash, оперативная память быстрая, но её мало. Python сам выделяет память для списков и строк и сам освобождает её: этим занимается **сборщик мусора** (garbage collector, GC). Он периодически находит объекты, на которые никто не ссылается, и возвращает их память. В этот момент программа ненадолго задерживается, поэтому в чувствительных ко времени местах вызывают `gc.collect()` заранее, в удобную паузу.

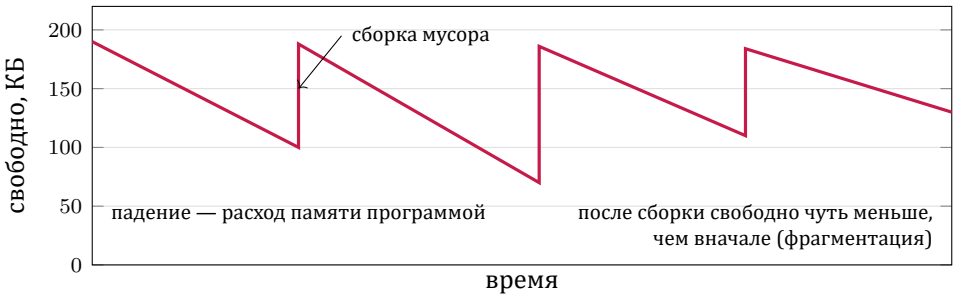


Рис. 12.11: Свободная память «пилой»: расходуется, пока сборщик не очистит неиспользуемые объекты (схема, цифры условные)

Листинг 12.10: MicroPython: сколько памяти свободно и что делает сборка мусора

```
import gc
import micropython

gc.collect()
print("свободно:", gc.mem_free(), "байт; занято:", gc.mem_alloc(), "байт")

data = [i * i for i in range(2000)]     # список из 2000 чисел занимает память
print("после списка, свободно:", gc.mem_free())

del data                                # ссылок нет, но память пока не возвращена
print("после del, свободно:", gc.mem_free())
gc.collect()                            # сборка мусора: освобождаем неиспользуемое
print("после collect, свободно:", gc.mem_free())

big = bytearray(50_000)                 # один большой кусок: нужна непрерывная область
print("bytearray(50000) создан, свободно:", gc.mem_free())
micropython.mem_info()                  # подробная таблица: сколько кусков, какие они
```

Листинг 12.11: CircuitPython: то же

```
import gc

gc.collect()
print("свободно:", gc.mem_free(), "байт; занято:", gc.mem_alloc(), "байт")

data = [i * i for i in range(2000)]     # список из 2000 чисел занимает память
print("после списка, свободно:", gc.mem_free())

del data                                # ссылок нет, но память пока не возвращена
print("после del, свободно:", gc.mem_free())
gc.collect()                            # сборка мусора: освобождаем неиспользуемое
print("после collect, свободно:", gc.mem_free())

big = bytearray(50000)                  # один большой кусок: нужна непрерывная область
```

```
print("bytearray(50000) создан, свободно:", gc.mem_free())
# модуля micropython.mem_info() в CircuitPython нет; хватает gc.mem_free() / gc.mem_alloc()
```

Если памяти не хватает, возникает `MemoryError`. Помогают такие приёмы: читать файл построчно, а не целиком (`for line in f`); использовать `bytearray` и `memoryview` вместо создания множества копий; не склеивать строки в цикле через `+`; и выделять большие буферы *один раз* в начале, а не в цикле. Из-за фрагментации можно получить `MemoryError` даже при «свободных» 40 КБ: нужен кусок *подряд*, а память раздроблена.

12.14 Копирование, удаление и резервные копии

- **MicroPython**: утилита `mpremote` с компьютера. `mpremote fs ls` — список файлов; `mpremote fs cp main.py :` — копировать на плату; `mpremote fs cp :config.json .` — с платы на компьютер; `mpremote fs rm :old.txt` — удалить; `mpremote fs cat :log.csv` — показать файл; `mpremote fs mkdir :data` — создать каталог. Двоеточие означает «на плате».
- **MicroPython**: Thonny показывает файлы платы в левой панели и позволяет перетаскивать их.
- **CircuitPython**: плата — обычный диск `CIRCUITPY`; копируй, удаляй и переименовывай файлы проводником. После записи дождись окончания копирования, прежде чем извлекать устройство.
- **Резервная копия**: перед экспериментами и обновлением прошивки скопируй на компьютер все свои файлы. Обновление **MicroPython** через BOOTSEL *не* сотрёт файлы, а вот смена типа прошивки (с **MicroPython** на **CircuitPython**) создаст новую файловую систему.
- Полное форматирование: в **MicroPython** — `os.umount("/")`, затем `os.VfsLfs2.mkfs(rp2.Flash())` и `os.mount(...)` (описано в документации порта rp2); в **CircuitPython** — `storage.erase_filesystem()`.

Осторожно

Не вынимай USB-кабель, пока идёт запись: даже у LittleFS незакрытый файл может потерять данные, а у FAT в **CircuitPython** можно получить «битый» диск. Если диск `CIRCUITPY` повредился, его можно переформатировать (`storage.erase_filesystem()`), но все файлы пропадут.

12.15 Самостоятельный опыт

Запусти пример журнала на 2–3 минуты, затем останови программу и открой файл: в **MicroPython** — `mpremote fs cat :log.csv`, в **CircuitPython** — открой `log.csv` на диске `CIRCUITPY` (если запись разрешена кнопкой). Построй график температуры в таблице. Зажми пальцами чип RP2040 и посмотри, как растёт кривая.

Если не работает

- **`OSError: [Errno 30] Read-only filesystem` в CircuitPython** — нет `boot.py` с `remount(readonly=False)`, или в нём вызвана кнопка в режиме «диск компьютеру». Измени `boot.py` (диск доступен, когда кнопка нажата) и перезагрузи плату.
- **`OSError: [Errno 28] No space left on device`** — диск полон: удали старые файлы или переходи на SD.
- **`OSError: [Errno 2] ENOENT`** — файла или каталога нет; проверь путь (в **CircuitPython** лучше писать абсолютные: `"/config.json"`).
- **`ValueError` при `json.load`** — файл пустой или повреждён (после сбоя питания); поймай

исключение и пересоздай файл.

- **Файл на плате не меняется после правки на компьютере** — для CircuitPython дождись записи; для MicroPython убедись, что редактируешь файл *на плате*, а не локальную копию.
- **`MemoryError`** — читай файлы построчно, вызывай `gc.collect()`, не создавай огромных списков.
- **SD-карта не определяется** — проверь питание модуля, номера пинов SPI, скорость (начни с 1 МГц) и форматирование в FAT32.

Итоги урока

- SRAM (264 КБ) — рабочая память; flash (2 МБ) хранит прошивку и файлы; программа выполняется из flash по технологии XIP.
- Flash стирается секторами по 4 КБ и выдерживает порядка 10^5 циклов: пиши редко, пачками, добавляй вместо перезаписи.
- LittleFS защищена от потери питания; открывай файлы через `with`, настройки сохраняй атомарно (временный файл + переименование).
- JSON подходит для настроек, CSV — для журналов; время в журнале — от запуска.
- В CircuitPython диск принадлежит компьютеру, запись из программы включается в `boot.py` (`storage.remount`), режим удобно выбирать кнопкой.
- Память контролируется через `gc.mem_free()` и `gc.collect()`; для файлов и копий есть `mpremote fs`, Thonny и SD-карта.

Проверь себя

1. Чем отличаются SRAM и flash? Какая из них сохраняет данные после выключения?
2. Почему нельзя бесконечно часто перезаписывать один файл во flash? Оцени срок жизни при записи раз в минуту.
3. Что такое атомарная запись и как её сделать в Python?
4. Почему в CircuitPython программа по умолчанию не может писать на диск и как это изменить?
5. Зачем нужна сборка мусора и что такое фрагментация памяти?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Создай файл `notes.txt`, запиши в него три строки, потом прочитай и выведи их с номерами строк.
2. ★ Определи объём свободного места на диске (`os.statvfs`) и размер каждого файла в корне платы.
3. ★★ Сделай счётчик запусков: при каждом включении читает число из файла, увеличивает на единицу и сохраняет. Как избежать износа, если плата включается тысячи раз?
4. ★★ Измени журнал температуры так, чтобы он писал строку только если температура изменилась больше чем на $0{,}5$ °C, и посчитай, насколько уменьшилось число записей.

5. ★★★ Хранение «рекордов»: JSON-файл с максимумом и минимумом температуры, обновляется только при новом рекорде; при повреждённом файле создаётся заново.
6. ★★★★ Подключи SD-карту и сделай регистратор: каждые 5 с строка в CSV (время, температура, яркость с фоторезистора) с ротацией файла по размеру; кнопка закрывает файл и безопасно отмонтирует карту перед извлечением.

Урок 13. Многозадачность: два ядра, потоки, asyncio

Цель	научиться выполнять несколько дел «одновременно»: запускать код на втором ядре через `_thread`, писать кооперативные задачи на `asyncio` и защищать общие данные
Опирается на	урок 2: цифровой выход; урок 3: кнопка; урок 4: время без `sleep()`; функции Python
Результат	красный светодиод мигает, пока кнопка переключает скорость и зелёный индикатор; «датчик, экран и светодиод» работают как три независимые задачи
Понадобится	светодиоды (2 шт.) с резисторами 220 Ω, кнопка, провода

13.1 Идея урока коротко

Настоящее устройство делает много дел сразу: мигает, слушает кнопки, измеряет, рисует на экране, связывается по сети. Если все дела писать в одном цикле с `sleep()`, то пока одно «спит», остальные ждут. Есть три способа разделить работу: **(1)** запустить дело на *втором ядре* процессора (в **MicroPython** модуль `_thread`); **(2)** разбить программу на *кооперативные задачи*, которые по очереди уступают друг другу управление (`asyncio`, работает в обеих системах); **(3)** вручную сравнивать время (урок 4). В этом уроке мы изучим первые два способа и научимся выбирать между ними.

13.2 Зачем нужна параллельность

Представь повара, который варит суп и жарит котлету. Он не стоит у кастрюли, пока та закипает: он ставит суп, переключается на котлету, снова смотрит на суп. Это **кооперативная многозадачность**: один исполнитель, но он умеет переключаться. А если на кухне двое поваров — это **настоящая параллельность**. В RP2040 два ядра — два повара.

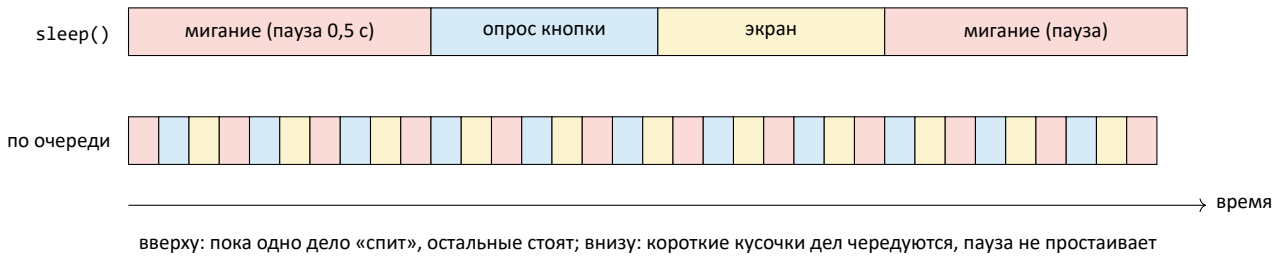


Рис. 13.1: Последовательная программа с `sleep()` и программа, в которой дела чередуются

13.3 Два ядра RP2040

В RP2040 два одинаковых ядра Arm Cortex-M0+ на частоте до 133 MHz. Они видят одну и ту же память и одну и ту же периферию, а общаются через специальный блок **SIO**: два FIFO по 8 слов (передача сообщений между ядрами) и *аппаратные спин-блокировки* (spinlocks) для защиты общих ресурсов. Обычно MicroPython работает на ядре 0, а ядро 1 спит, пока его не запросят.

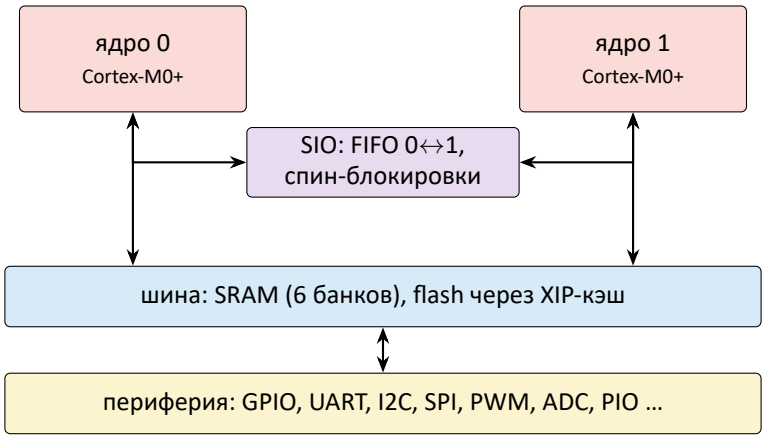


Рис. 13.2: Два ядра используют общую память и периферию; SIO помогает согласовать их работу

Это интересно

В CircuitPython нет модуля `_thread`: пользовательская программа выполняется на одном ядре. Параллельность там строится на `asyncio` и на аппаратных блоках, которые работают сами по себе: PWM, PIO, DMA (урок 14).

13.4 Модуль _thread в MicroPython

Поток (thread) — независимая «нить» выполнения программы. На RP2 в MicroPython программа сначала работает в одном (главном) потоке на ядре 0. Вызов `_thread.start_new_thread(функция, (аргументы,))` запускает функцию на **ядре 1**.



Рис. 13.3: Жизнь потока: запуск, работа на втором ядре, завершение по флагу

Листинг 13.1: MicroPython: мигание на втором ядре, пока основная программа печатает

```
import _thread
import time
from machine import Pin

led = Pin(13, Pin.OUT)
run = True                              # флаг «продолжать работу», общий для обоих ядер


def blinker():
```

```
    """Эта функция работает на втором ядре (ядро 1)"""
    while run:
        led.toggle()
        time.sleep_ms(250)
    led.value(0)


_thread.start_new_thread(blinker, ())   # запускаем поток; (): аргументов нет

for i in range(10):                     # основная программа (ядро 0) занята своим делом
    print("основная программа, шаг", i)
    time.sleep(1)

run = False                             # просим поток завершиться
time.sleep_ms(300)
print("готово")
```

Что важно знать (проверяй по документации своей версии прошивки):

- На RP2 можно запустить **только один** дополнительный поток (ядро 1); попытка запустить второй вызовет `OSError`. Когда поток закончится, ядро снова свободно.
- Память у потоков *общая*: глобальные переменные видны обоим. Поэтому передавать данные можно просто через переменные — но тут и подстерегают гонки (см. ниже).
- Поток нельзя «убить» снаружи: надо завершать его самому, проверяя флаг (как `run` в примере) или вызывая `_thread.exit()`.
- Исключение в потоке просто завершает поток, а сообщение об ошибке может выглядеть не там, где ты ждёшь. Оборачивай тело потока в `try/except` и печатай ошибку.
- В **MicroPython** для RP2 может действовать *глобальная блокировка интерпретатора* (GIL; зависит от сборки и версии): тогда два потока Python не исполняют байт-код буквально одновременно, и вычислительная работа ускорится не вдвое. Не рассчитывай на ускорение Python-расчётов вдвое. Выигрыш чаще получается там, где один поток *ждёт* (датчик, сеть), пока другой работает.
- Во время записи во flash (сохранение файла, урок 12) второе ядро приостанавливается на несколько миллисекунд. Мигание на ядре 1 может на мгновение «споткнуться».
- Избегай одновременной работы двух потоков с одной и той же периферией (один UART, одна шина I2C) без блокировки.

13.5 Гонка данных и замок

Допустим, оба ядра увеличивают счётчик `counter += 1`. Для процессора это не одна операция, а три: прочитать, прибавить, записать. Если второй поток вмешается между шагами первого, одно из прибавлений *потеряется*.

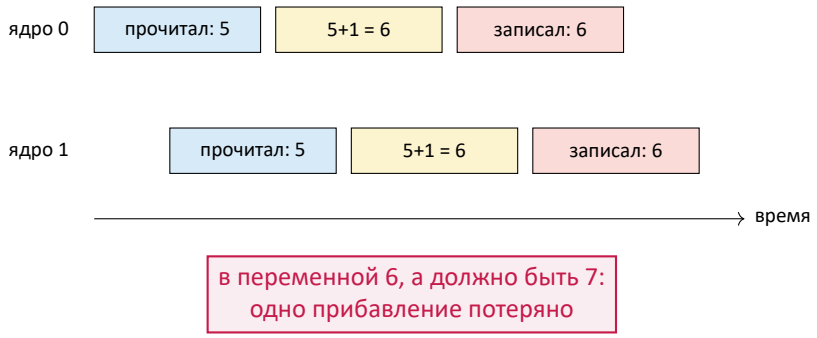


Рис. 13.4: Гонка данных: оба ядра прочитали 5 до того, как кто-либо записал результат

Лекарство — **замок** (lock, mutex): только тот поток, который «держит» замок, может работать с общей переменной. Остальные ждут. Код под замком называют *критической секцией*; держи её как можно короче.

Листинг 13.2: MicroPython: счётчик без замка и с замком

```
import _thread
import time

N = 100_000                             # сколько раз каждый поток увеличивает счётчик
counter = 0                             # общая переменная
finished = False
lock = _thread.allocate_lock()          # замок для защиты общей переменной


def count(n, use_lock):
    global counter
    for _ in range(n):
        if use_lock:
            with lock:                  # внутри блока только один поток
                counter += 1
        else:
            counter += 1                # чтение, сложение и запись --- три шага


def experiment(use_lock):
    global counter, finished
    counter = 0
    finished = False

    def second():
        global finished
        count(N, use_lock)
        finished = True

    _thread.start_new_thread(second, ())    # второе ядро
    count(N, use_lock)                      # и основное ядро делают одно и то же
    while not finished:
        time.sleep_ms(10)
    name = "с замком   " if use_lock else "без замка  "
    print(name, "итог:", counter, "ожидали:", 2 * N)


experiment(False)                       # итог может оказаться меньше ожидаемого
experiment(True)                        # итог всегда равен 2 * N
```

Осторожно

Отсутствие потерь в твоём запуске *не доказывает* безопасность: гонки зависят от тайминга и появляются редко, зато в самый неподходящий момент. Любое чтение-изменение-запись общей переменной защищай замком. Не используй замок внутри прерывания и не захватывай два замка в разном порядке — получится взаимная блокировка (deadlock): оба потока ждут друг друга вечно.

13.6 Кооперативная многозадачность: asyncio

Второй подход вообще не нужен второму ядру. В `asyncio` программа состоит из **корутин**: функций `async def`, которые в определённых местах (`await`) добровольно уступают управление. Планировщик — **цикл событий** (event loop) — выбирает, какая задача будет работать дальше. Пока задача спит в `await asyncio.sleep(...)`, процессор выполняет остальные.

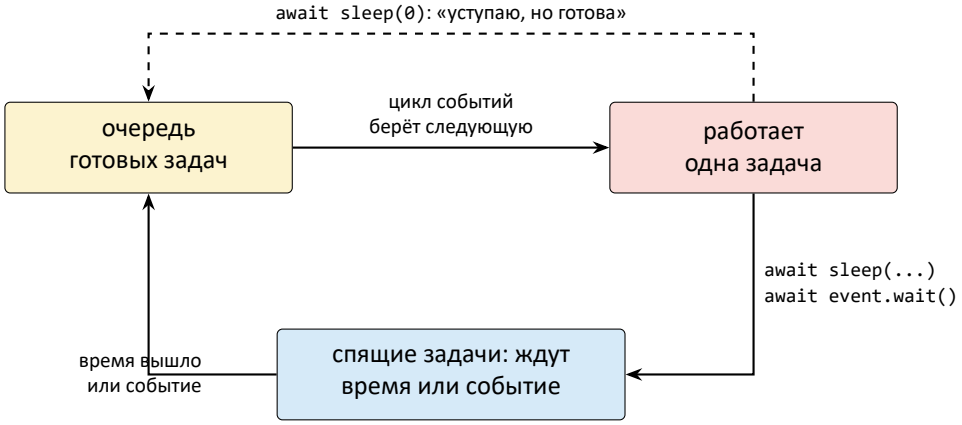


Рис. 13.5: Цикл событий asyncio: задача работает до ближайшего `await`, затем планировщик выбирает следующую

Три ключевых слова:

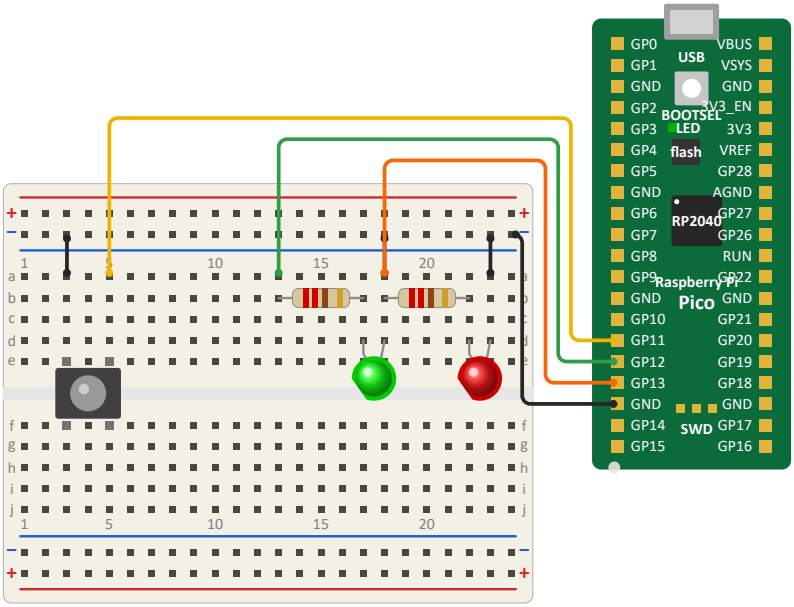
- `async def f():` — объявление корутины. Вызов `f()` *не* выполняет её, а создаёт объект.
- `await x` — «подожду результата и пока отдам управление другим». Использовать можно только внутри `async def`.
- `asyncio.run(main())` — запускает цикл событий и главную корутину.

Вызов	Смысл
`asyncio.run(main())`	запустить цикл событий и дождаться завершения `main`
`asyncio.create_task(coro)`	запустить корутину как задачу в фоне и сразу идти дальше
`await asyncio.gather(a(), b())`	запустить несколько корутин и дождаться, пока все закончатся
`await asyncio.sleep(s)` / `sleep_ms(ms)`	уснуть, не блокируя остальные (`sleep_ms` есть только в MicroPython)
`asyncio.Event()`: `set`, `clear`, `await wait()`	сигнал «что-то произошло» для одной или многих задач
`asyncio.Lock()`: `async with lock:`	критическая секция между задачами
`await asyncio.wait_for(coro, t)`	ограничить ожидание по времени
`task.cancel()`	отменить задачу

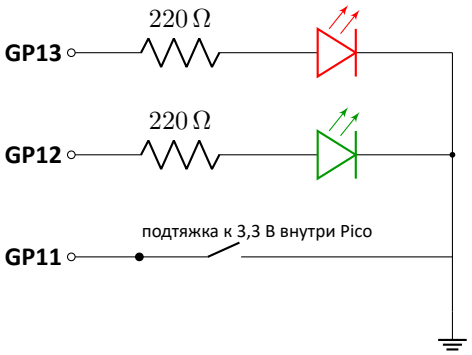
Таблица 13.1: Основной набор `asyncio`

13.7 Практика: мигаем и опрашиваем кнопку

Красный светодиод мигает, а кнопка переключает его скорость и зелёный индикатор. Раньше с `sleep()` кнопка срабатывала бы только в паузы между миганиями. Теперь у каждого дела своя задача.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 13.6: Красный светодиод на GP13, зелёный на GP12, кнопка на GP11

Листинг 13.3: MicroPython: две задачи asyncio — мигание и опрос кнопки

```
import asyncio                          # в версиях до 1.21: import uasyncio as asyncio
from machine import Pin

led = Pin(13, Pin.OUT)                  # красный: мигает
status = Pin(12, Pin.OUT)               # зелёный: переключается нажатием
button = Pin(11, Pin.IN, Pin.PULL_UP)   # кнопка на GND, нажатие = 0
blink_ms = 500                          # текущая половина периода мигания


async def blink():
    while True:
        led.toggle()
        await asyncio.sleep_ms(blink_ms)    # отдаём управление другим задачам

```

```
15
16 async def watch_button():
17     global blink_ms
18     was = 1
19     while True:
20         now = button.value()
21         if was == 1 and now == 0:       # край: был 1, стал 0 --- нажали
22             blink_ms = 100 if blink_ms == 500 else 500
23             status.toggle()
24             await asyncio.sleep_ms(30)  # пережидаем дребезг контактов
25         was = now
26         await asyncio.sleep_ms(5)       # опрашиваем 200 раз в секунду
27
28
29 async def main():
30     await asyncio.gather(blink(), watch_button())   # обе задачи работают «вместе»
31
32
33 asyncio.run(main())
```

Листинг 13.4: CircuitPython: то же самое

```
1  import asyncio                          # библиотека asyncio: circup install asyncio
2  import board
3  import digitalio
4
5  led = digitalio.DigitalInOut(board.GP13)        # красный: мигает
6  led.direction = digitalio.Direction.OUTPUT
7  status = digitalio.DigitalInOut(board.GP12)     # зелёный: переключается нажатием
8  status.direction = digitalio.Direction.OUTPUT
9  button = digitalio.DigitalInOut(board.GP11)     # кнопка на GND, нажатие = False
10 button.switch_to_input(pull=digitalio.Pull.UP)
11 blink_s = 0.5                                   # текущая половина периода мигания
12
13
14 async def blink():
15     while True:
16         led.value = not led.value
17         await asyncio.sleep(blink_s)            # отдаём управление другим задачам
18
19
20 async def watch_button():
21     global blink_s
22     was = True
23     while True:
24         now = button.value
25         if was and not now:                     # был HIGH, стал LOW --- нажали
26             blink_s = 0.1 if blink_s == 0.5 else 0.5
27             status.value = not status.value
28             await asyncio.sleep(0.03)           # пережидаем дребезг контактов
29         was = now
30         await asyncio.sleep(0.005)              # опрашиваем 200 раз в секунду
31
32
33 async def main():
34     await asyncio.gather(blink(), watch_button())
35
36
37 asyncio.run(main())
```

Почему кнопка теперь реагирует мгновенно? Задача `watch_button` просыпается каждые 5 мс, и всё остальное время занята только тем, что быстро проверяет вход. Все задачи вместе укладываются в доли миллисекунды, а остальное время процессор отдаёт другим задачам или спит. Таблица отличий систем: в **MicroPython** `asyncio` встроен и имеет `sleep_ms`; в **CircuitPython** это библиотека (ставится `circup install asyncio`, зависимость `adafruit_ticks`), время задают в секундах (`asyncio.sleep(0.1)`).

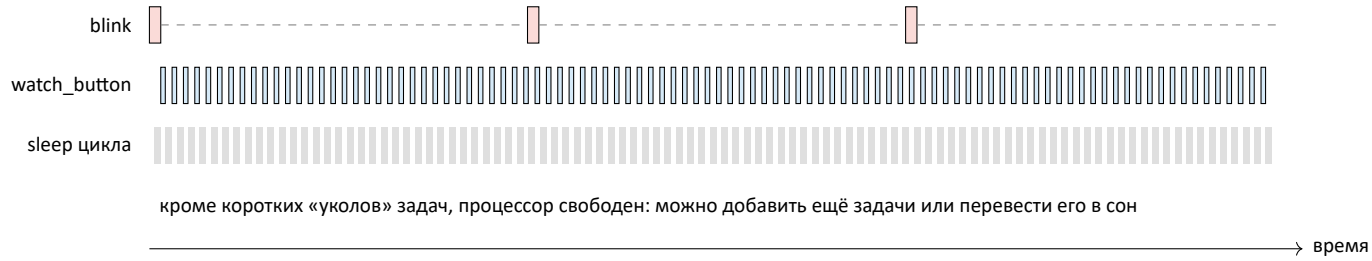


Рис. 13.7: Три «задачи» во времени: короткие включения и длинные паузы; на диаграмме не в масштабе

13.8 Учимся программировать: корутины и await

Идея «функция, которую можно приостановить» — новая и самая трудная в уроке. Запомни четыре правила.

1. Любая функция, которая внутри себя использует `await`, должна быть объявлена как `async def`.
2. Вызов `async`-функции без `await` или `create_task` *ничего не делает* (частая ошибка: забыть `await`).
3. Внутри задачи **нельзя** вызывать долгие блокирующие функции (`time.sleep(2)`, ожидание в цикле без `await`): остановится вся программа, ведь планировщик один.
4. Всегда пиши в долгом цикле хотя бы один `await asyncio.sleep(...)`, иначе другие задачи никогда не получат управление.

13.9 Обмен данными между задачами: «датчик, экран, светодиод»

Три задачи: `sensor` измеряет, `display` показывает, `alarm` мигает при перегреве. Датчик кладёт измерение в список-«ящик» и будит экран сигналом `Event`. Так задачи не знают друг о друге и взаимодействуют только через общие данные и сигналы.

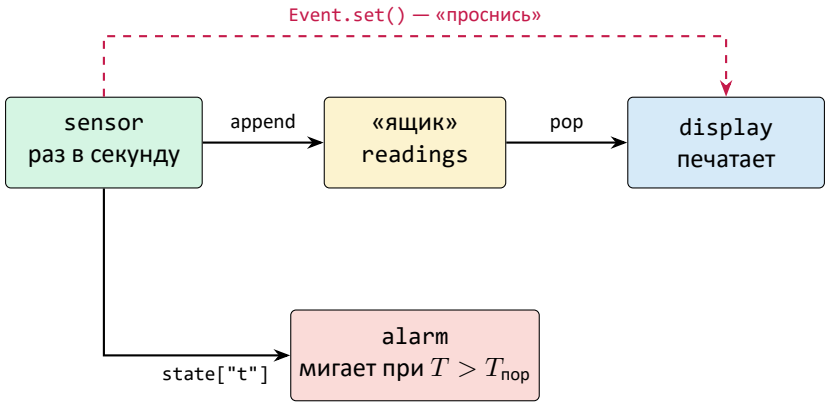


Рис. 13.8: Три задачи общаются через общие данные (ящик и последнее значение) и сигнал-событие

Листинг 13.5: MicroPython: датчик, экран и светодиод как три задачи

```
import asyncio
from machine import ADC, Pin

adc = ADC(4)                            # встроенный датчик температуры чипа
led = Pin(13, Pin.OUT)                  # сигнал перегрева
LIMIT = 30.0                            # порог тревоги, °C

readings = []                           # «почтовый ящик» между задачами
new_data = asyncio.Event()              # сигнал «есть новые данные»
state = {"t": 0.0}                      # последняя температура


async def sensor():
    """Датчик: раз в секунду кладёт измерение в ящик и будит экран"""
    while True:
        volts = adc.read_u16() * 3.3 / 65535
        t = 27 - (volts - 0.706) / 0.001721
        state["t"] = t
        readings.append(t)
        new_data.set()
        await asyncio.sleep(1)


async def display():
    """«Экран»: ждёт сигнала и печатает все накопленные значения.
    Вместо print сюда можно вставить вывод на OLED (урок про I2C)"""
    while True:
        await new_data.wait()           # спим, пока датчик не позовёт
        new_data.clear()
        while readings:
            print("T = {:.1f} C".format(readings.pop(0)))


async def alarm():
    """Светодиод: при перегреве быстро мигает, иначе погашен"""
    while True:
        if state["t"] > LIMIT:
            led.toggle()
            await asyncio.sleep_ms(100)
        else:
            led.value(0)
            await asyncio.sleep_ms(200)


async def main():
    await asyncio.gather(sensor(), display(), alarm())


asyncio.run(main())
```

Листинг 13.6: CircuitPython: то же

```
import asyncio
import board
import digitalio
import microcontroller

led = digitalio.DigitalInOut(board.GP13)        # сигнал перегрева
led.direction = digitalio.Direction.OUTPUT
LIMIT = 30.0                                    # порог тревоги, °C

readings = []                                   # «почтовый ящик» между задачами
new_data = asyncio.Event()                      # сигнал «есть новые данные»
state = {"t": 0.0}                              # последняя температура


async def sensor():
    """Датчик: раз в секунду кладёт измерение в ящик и будит экран"""
    while True:
        t = microcontroller.cpu.temperature
        state["t"] = t
        readings.append(t)
        new_data.set()
        await asyncio.sleep(1)


async def display():
    """«Экран»: ждёт сигнала и печатает все накопленные значения.
    Вместо print сюда можно вставить вывод на OLED (урок про I2C)"""
    while True:
        await new_data.wait()                   # спим, пока датчик не позовёт
        new_data.clear()
        while readings:
            print("T = {:.1f} C".format(readings.pop(0)))


async def alarm():
    """Светодиод: при перегреве быстро мигает, иначе погашен"""
    while True:
        if state["t"] > LIMIT:
            led.value = not led.value
            await asyncio.sleep(0.1)
        else:
            led.value = False
            await asyncio.sleep(0.2)


async def main():
    await asyncio.gather(sensor(), display(), alarm())


asyncio.run(main())
```

Совет

Стандартной готовой очереди `asyncio.Queue` в ядре MicroPython и в библиотеке `asyncio` для CircuitPython может не быть (зависит от версии и набора расширений), поэтому здесь

«ящик» — обычный список плюс `Event`. Это безопасно, потому что в `asyncio` задачи переключаются только на `await` и не мешают друг другу посреди операции. Если у тебя есть `asyncio.Queue`, используй её.

13.10 Замок в asyncio и когда он нужен

В `asyncio` гонки редки: задача никто не прерывает до ближайшего `await`. Но если между чтением и записью общей переменной стоит `await`, другая задача успеет вмешаться, и результат окажется неверным. Тогда нужен `asyncio.Lock`.

Листинг 13.7: MicroPython: замок для задач asyncio

```
import asyncio

lock = asyncio.Lock()                   # замок для задач asyncio
balance = 100


async def spend(name, amount):
    global balance
    async with lock:                    # внутри блока --- только одна задача
        current = balance               # читаем
        await asyncio.sleep_ms(10)      # здесь другая задача могла бы вмешаться без замка
        balance = current - amount      # пишем
        print(name, "потратил", amount, "-> остаток", balance)


async def main():
    await asyncio.gather(spend("A", 30), spend("B", 50))
    print("итого:", balance)            # всегда 20


asyncio.run(main())
```

Листинг 13.8: CircuitPython: то же

```
import asyncio

lock = asyncio.Lock()                   # замок для задач asyncio
balance = 100


async def spend(name, amount):
    global balance
    async with lock:                    # внутри блока --- только одна задача
        current = balance               # читаем
        await asyncio.sleep(0.01)       # здесь другая задача могла бы вмешаться без замка
        balance = current - amount      # пишем
        print(name, "потратил", amount, "-> остаток", balance)


async def main():
    await asyncio.gather(spend("A", 30), spend("B", 50))
    print("итого:", balance)            # всегда 20


asyncio.run(main())
```

Без `async with lock` обе задачи прочитали бы 100 и записали 70 и 50 — одна трата «потерялась» бы. С замком результат всегда 20.

13.11 Сравнение подходов и как выбрать

	Опрос времени (урок 4)	`asyncio`	`_thread`
MicroPython	да	да (встроен)	да, один поток (ядро 1)
CircuitPython	да	да (библиотека)	нет
Настоящая параллельность	нет	нет	да (ядра), с оговорками
Гонки данных	нет	редки (только между `await`)	да, нужны замки
Сложность	низкая, но код «растёт»	средняя	высокая, трудно отлаживать
Лучше всего для	2–3 простых дела	много ожиданий (кнопки, датчики, сеть)	тяжёлые долгие расчёты в фоне

Таблица 13.2: Подходы к многозадачности

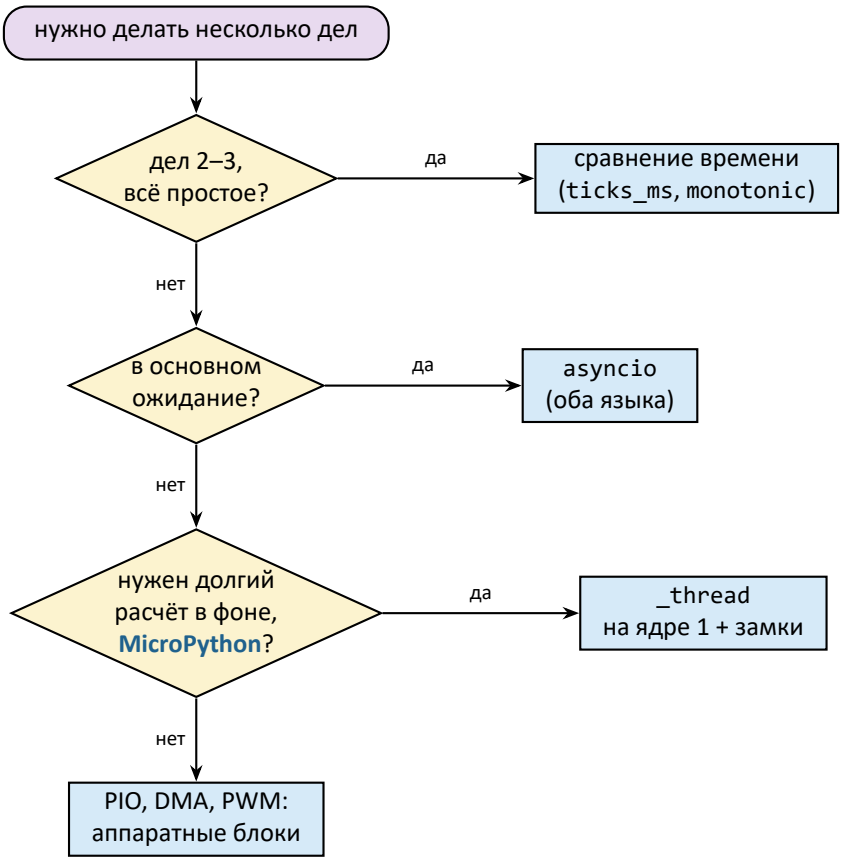


Рис. 13.9: Как выбрать способ распараллеливания

13.12 Самостоятельный опыт

Собери схему (рис. 13.6) и запусти пример с кнопкой. Затем вставь в `blink()` вызов `time.sleep(1)` вместо `await asyncio.sleep_ms(...)` (в CircuitPython — `time.sleep(1)`): кнопка

перестанет реагировать, и ты своими глазами увидишь, что «блокирующий» вызов ломает кооперативную многозадачность.

Если не работает

- **`RuntimeWarning: coroutine ... was never awaited`** (или «ничего не происходит») — забыт `await` или `asyncio.create_task` перед вызовом корутины.
- **Задачи не переключаются, светодиод «замерзает»** — где-то стоит `time.sleep()` или долгий цикл без `await`.
- **`OSError: can't create thread`** (MicroPython) — на RP2 уже запущен второй поток; дождись его завершения.
- **`ImportError: no module named 'asyncio'`** — в старых MicroPython модуль называется `uasyncio`; в CircuitPython установи библиотеку `asyncio` и `adafruit_ticks` через `circup`.
- **Счётчик в потоках иногда меньше нужного** — не защищена общая переменная замком.
- **Ctrl+C не останавливает программу с потоком** — сначала поставь флаг `run = False` либо перезагрузи плату (Ctrl+D в REPL MicroPython).
- **Кнопка срабатывает несколько раз за нажатие** — не выдержана пауза на дребезг; увеличь её до 30–50 мс.

Итоги урока

- В RP2040 два ядра; в MicroPython модуль `_thread` запускает функцию на втором ядре (один дополнительный поток), общая память и гонки требуют замков.
- `asyncio` — кооперативная многозадачность: задача работает до `await`, цикл событий переключает задачи; работает и в MicroPython, и в CircuitPython.
- Для общения задач используют общие данные, `Event`, `Lock`; блокирующие вызовы внутри задач запрещены.
- В CircuitPython потоков нет; параллельность дают `asyncio` и аппаратные блоки (PWM, PIO, DMA).
- Для 2–3 простых дел хватает сравнения времени; для многих ожиданий лучше `asyncio`; для тяжёлых расчётов — поток на ядре 1.

Проверь себя

1. Чем кооперативная многозадачность отличается от настоящей параллельности?
2. Что такое гонка данных? Приведи пример на счётчике.
3. Почему в `asyncio` нельзя вызывать `time.sleep()` внутри задачи?
4. Сколько дополнительных потоков можно запустить на RP2 в MicroPython и на каком ядре они работают?
5. Как задачи `sensor` и `display` передают друг другу данные в примере урока?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Запусти пример с кнопкой и измени скорости мигания (500 и 100 мс) на 800 и 200 мс.

2. ★ Запусти пример `_thread` (MicroPython) и измени период мигания на втором ядре. Что печатает основная программа?
3. ★★ Добавь в пример с кнопкой третью задачу: зелёный светодиод мигает в противофазе с красным.
4. ★★ В примере «датчик, экран, светодиод» поменяй порог перегрева на температуру чуть выше комнатной и подогрей чип пальцем, чтобы сработала тревога.
5. ★★★ Сделай «светофор» из трёх светодиодов на `asyncio`, в котором кнопка пешехода запрашивает переключение, не нарушая длительности фаз.
6. ★★★★ Реализуй на двух ядрах (MicroPython) «регистратор»: ядро 1 каждые 100 мс измеряет АЦП и кладёт значения в общий буфер под замком, ядро 0 раз в секунду забирает данные, считает среднее и печатает. Проверь, нет ли потерянных или повторённых значений.

Урок 14. PIO: программируемый ввод-вывод

Цель	понять, как устроен PIO — самая необычная часть RP2040, — и написать свои маленькие «аппаратные» программы: мигалку, передатчик адресных светодиодов, измеритель импульса и UART-передатчик
Опирается на	урок 7: UART и скорость обмена; урок 11: протокол WS2812 и его тайминги; двоичные числа и сдвиги
Результат	светодиод мигает без участия процессора; измеритель считает ширину импульса с точностью $0{,}2\,\mu s$; ты читаешь простые программы на ассемблере PIO
Понадобится	светодиод с резистором $220\,\Omega$, провод для перемычки GP14–GP15; по желанию — кольцо WS2812 из урока 11

14.1 Идея урока коротко

Бывают сигналы, которые нужно формировать с точностью до долей микросекунды: протокол WS2812, нестандартный последовательный интерфейс, измерение ширины короткого импульса. Интерпретатор Python для этого слишком медленный и неровный, а делать это «в лоб» процессором значит занять его целиком. В RP2040 есть для этого специальные блоки **PIO** (Programmable I/O). Это маленькие программируемые процессоры, которые умеют только одно: с точностью до такта читать и писать выводы GPIO. Программа для них состоит из девяти команд и пишется на особом ассемблере. Запущенная программа работает сама, пока основной процессор занят другим делом.

14.2 Зачем нужен PIO

Сравним способы «пошевелить» выводом:

- **Программно (bit-banging)**: цикл Python включает и выключает вывод. Точность — десятки микросекунд, процессор занят полностью, любое прерывание портит сигнал.
- **Готовая периферия** (UART, SPI, I2C, PWM): точно и без нагрузки, но только тем протоколам, которые аппаратно заложены.

- **PIO**: точность в один такт (8 ns на частоте 125 MHz), нагрузка на процессор почти нулевая, а протокол программируется самим пользователем.

Поэтому PIO часто называют «конструктором интерфейсов»: программы на PIO уже реализуют WS2812, SD-карты по четырём линиям, DVI-видео, VGA, камеры, I2S-звук, квадратурные энкодеры, 1-Wire.

14.3 Устройство PIO

В RP2040 два блока PIO (PIO0 и PIO1). В каждом блоке — четыре независимых **автомата** (state machine, SM) и общая для них память программ на **32 команды**. Все автоматы блока работают от тех же команд, но могут запускать разные программы из этой памяти.

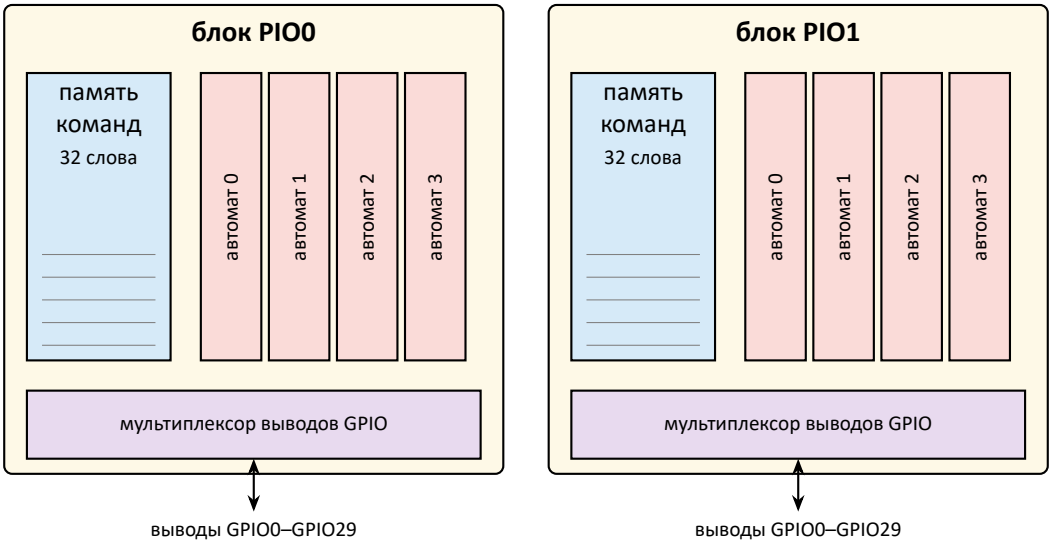


Рис. 14.1: Два блока PIO по четыре автомата; память команд общая у четырёх автоматов своего блока

Что есть внутри одного автомата (рис. 14.2):

- **TX FIFO** и **RX FIFO** — очереди по 4 слова (по 32 бита) для обмена с процессором или DMA: процессор кладёт данные для вывода в TX, а забирает принятое из RX. Очереди можно объединить в одну на 8 слов (`fifo_join`).
- **OSR** (Output Shift Register) — выходной сдвиговый регистр. Из TX FIFO слово переписывается в OSR, а из OSR команда `out` выдвигает биты.
- **ISR** (Input Shift Register) — входной сдвиговый регистр: команда `in` вдвигает в него биты, а `push` отправляет в RX FIFO.
- **X** и **Y** — два 32-битных «блокнота» (scratch): счётчики, временные значения.
- **PC** — счётчик команд; **делитель частоты** — задаёт скорость выполнения.
- **Выводы**: группы `OUT`, `SET`, `IN`, `side-set` и один вывод для условия `JMP PIN` — каждая настраивается своим начальным выводом и количеством.

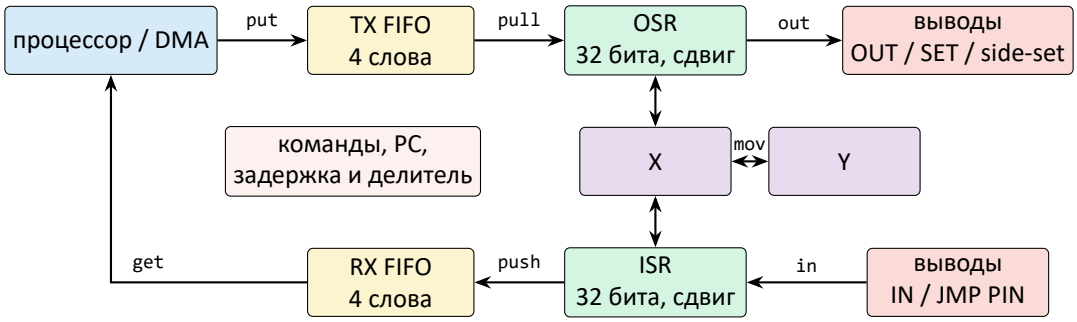


Рис. 14.2: Устройство автомата PIO: два сдвиговых регистра, две очереди, два «блокнота» X и Y

14.4 Тактирование: делитель частоты

Автомат выполняет **одну команду за один свой такт**. Частота автомата получается делением системной частоты (по умолчанию 125 MHz у Pico и 150 MHz у Pico 2):

$$f_{\text{SM}} = \frac{f_{\text{sys}}}{\text{div}}, \qquad 1 \le \text{div} \le 65536 \ \text{(с дробной частью)}.$$

Максимальная частота равна системной, а минимальная — $125\,\text{МГц}/65536 \approx 1{,}9\,\text{kHz}$. В MicroPython задаём её параметром `freq=`, в CircuitPython — `frequency=`. Правильный выбор частоты — половина искусства PIO: например, для протокола с периодом $1{,}25\,\mu\text{s}$ берут частоту 8 МГц, чтобы на бит приходилось ровно 10 тактов.

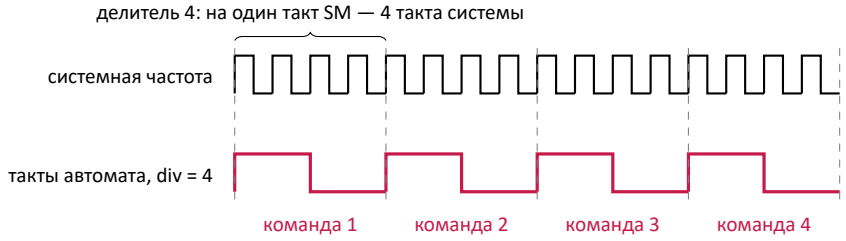


Рис. 14.3: Делитель частоты: одна команда PIO выполняется за один такт автомата

14.5 Девять команд PIO

У PIO всего девять команд. Каждая занимает **16 бит**: три бита — код команды, пять бит — поле задержки (и side-set), восемь бит — операнд.



Рис. 14.4: Формат одной команды PIO

Команда	Запись в MicroPython	Что делает
`jmp`	`jmp(x_dec, "метка")`	условный переход: всегда, `not_x`, `x_dec` (уменьшить и перейти, если не 0), `not_y`, `y_dec`, `x_not_y`, `pin`, `not_osre`
`wait`	`wait(1, pin, 0)`	ждать, пока вывод (`gpio`, `pin`) или флаг `irq` примет значение 0 или 1
`in`	`in_(pins, 1)`	вдвинуть биты из источника (`pins`, `x`, `y`, `null`, `isr`, `osr`) в ISR
`out`	`out(pins, 1)`	выдвинуть биты из OSR в приёмник (`pins`, `x`, `y`, `null`, `pindirs`, `pc`, `isr`, `exec`)
`push`	`push(noblock)`	отправить ISR в RX FIFO (при `block` — ждать место)
`pull`	`pull()`	взять слово из TX FIFO в OSR (при пустой очереди автомат ждёт; `noblock` — берёт из X)
`mov`	`mov(x, invert(null))`	скопировать значение (можно инвертировать `invert` или развернуть биты `reverse`)
`irq`	`irq(rel(0))`	установить, сбросить или ждать флаг прерывания (сообщает процессору или другому автомату)
`set`	`set(pins, 1)`	записать константу 0–31 в выводы, `x`, `y` или направления выводов

Таблица 14.1: Девять команд PIO (запись MicroPython; в CircuitPython то же в текстовом виде: `jmp x--, метка`, `wait 1 pin 0`)

К этому добавляют вспомогательные пометки: `label("имя")` — метка; `wrap_target()` и `wrap()` — границы цикла (по достижении `wrap` автомат прыгает на `wrap_target` *бесплатно*, без тратящей такт команды `jmp`); `nop()` — «пустая» команда (на самом деле `mov y, y`). Из таблицы видно, чего нет: ни сложения, ни умножения. Считать в PIO можно только вычитанием единицы (`x_dec`) и сравнением, и этого хватает для задержек и счётчиков.

Задержки и side-set

К любой команде можно добавить **задержку** `[n]`: после выполнения автомат подождёт ещё n тактов, и вся команда займёт $1 + n$ тактов. Максимум — 31 такт, но если часть пяти бит занята под **side-set**, максимум уменьшается.

Side-set — возможность выставить уровень на выводах *одновременно* с выполнением любой команды, не тратя на это отдельных тактов. Выводы для side-set задаются параметрами `sideset_base` и `sideset_init`, а значение — пометкой `.side(1)` после команды. Благодаря этому формирование импульса WS2812 укладывается в 10 тактов строго.

14.6 Первая программа: мигалка на PIO

Сначала самое простое: светодиод мигает с периодом 1 с, и *процессор для этого не нужен*. Автомат считает такты вложенными циклами: `set(x, 31)`, затем `nop() [29]` и `jmp(x_dec, ...)`. Внутренний цикл выполняется 32 раза, по $30 + 1 = 31$ такту, то есть около 1000 тактов; при частоте 2000 Hz это 0,5 s.

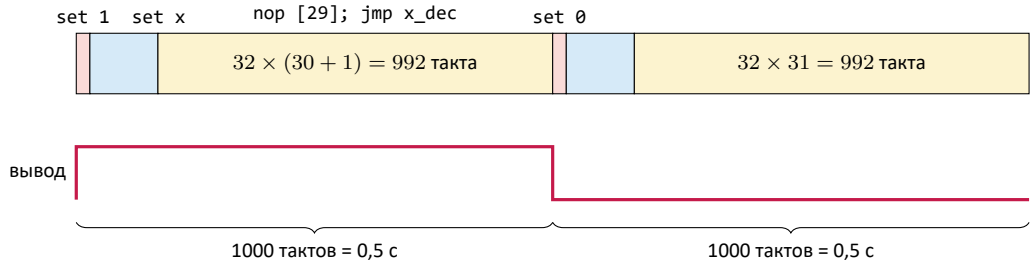


Рис. 14.5: Мигалка на PIO: на каждую половину периода уходит ровно по 1000 тактов

Листинг 14.1: MicroPython: мигалка на PIO (`rp2.asm_pio`)

```
import rp2
from machine import Pin


@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)      # вывод для команды set: выход, начальный уровень 0
def blink_1hz():
    # Такты: 1 + 7 + 32 * (30 + 1) = 1000 на «высокую» часть
    set(pins, 1)                    # включить светодиод (1 такт)
    set(x, 31)                  [6] # счётчик x = 31, затем ждать ещё 6 тактов (итого 7)
    label("delay_high")
    nop()                       [29]    # 1 такт + 29 тактов задержки = 30
    jmp(x_dec, "delay_high")        # x -= 1; если x был не 0 --- на метку (1 такт)
    # Такты: 1 + 7 + 32 * (30 + 1) = 1000 на «низкую» часть
    set(pins, 0)                    # выключить
    set(x, 31)                  [6]
    label("delay_low")
    nop()                       [29]
    jmp(x_dec, "delay_low")


# Автомат 0 блока PIO 0; частота автомата 2000 Гц --- один такт = 0,5 мс,
# 1000 тактов = 0,5 с: светодиод мигает с периодом 1 с
sm = rp2.StateMachine(0, blink_1hz, freq=2000, set_base=Pin(13))
sm.active(1)                        # пуск: дальше процессор свободен, мигает автомат

# sm.active(0)                      # остановка
```

Листинг 14.2: CircuitPython: то же на `rp2pio` и `adafruit_pioasm`

```
import time
import board
import rp2pio
import adafruit_pioasm      # библиотека из бандла: circup install adafruit_pioasm

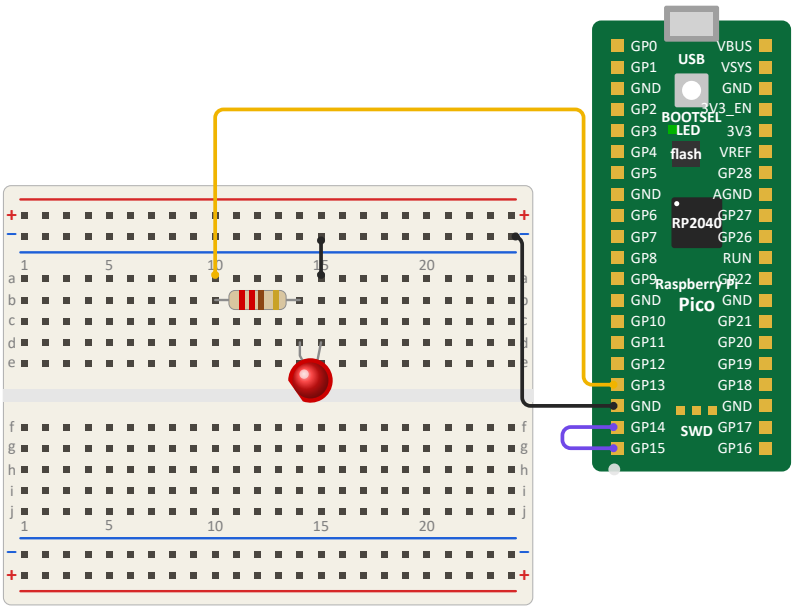
# Ассемблер PIO в виде обычной строки (синтаксис как в документации Raspberry Pi)
program = adafruit_pioasm.assemble("""
    set pins, 1                 ; включить светодиод
    set x, 31           [6]     ; счётчик x = 31, пауза ещё 6 тактов
delay_high:
    nop                 [29]    ; 1 такт + 29 тактов задержки
    jmp x-- delay_high          ; x -= 1, если x был не 0 --- на метку
    set pins, 0                 ; выключить светодиод
    set x, 31           [6]
delay_low:
```

```
    nop                [29]
    jmp x-- delay_low
""")

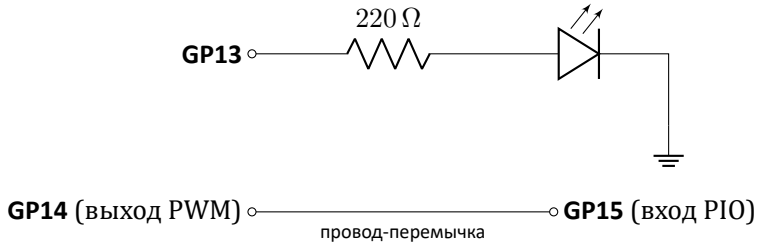
# frequency=2000: один такт = 0,5 мс, период мигания 2 x 1000 тактов = 1 с.
# Программа без .wrap_target / .wrap по умолчанию зацикливается сама
sm = rp2pio.StateMachine(program, frequency=2000, first_set_pin=board.GP13)

while True:
    time.sleep(1)          # процессор свободен; светодиодом занят автомат PIO
```

Разница в подходах. В **MicroPython** ассемблер PIO встроен в Python: программа — это обычная функция с декоратором `@rp2.asm_pio`, а команды записываются как вызовы функций (`set(pins, 1)`, `jmp(x_dec, "метка")`); задержка пишется в квадратных скобках `[29]`, а side-set — через `.side()`. В **CircuitPython** программа — *строка* на настоящем ассемблере PIO (`set pins, 1`, `jmp x-- метка`, `[29]`), которую библиотека `adafruit_pioasm` собирает в машинные слова, а `rp2pio.StateMachine` запускает. Названия параметров тоже разные: `set_base` и `first_set_pin`, `freq` и `frequency`.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 14.6: Светодиод на GP13 (мигалка PIO) и перемычка GP14–GP15 для опытов с измерением

Фиолетовая перемычка между **GP14** и **GP15** понадобится ниже: один вывод будет выдавать испытательный импульс, а автомат PIO его измерять. Красный светодиод на **GP13** мигает под управлением PIO.

14.7 Учимся программировать: биты и сдвиги

PIO работает с отдельными битами, поэтому нужно привыкнуть к **сдвигам**. Число в регистре — ряд из 32 бит. Команда `out(pins, 1)` *выдвигает* один бит из OSR и подаёт его на вывод, а оставшиеся биты сдвигаются. Направление выбирается параметром `out_shiftdir`: `SHIFT_LEFT` — первым уходит старший бит (так хочет WS2812), `SHIFT_RIGHT` — младший (так передаёт UART).



Рис. 14.7: Одно и то же число выдвигается в разном порядке

В Python те же операции записывают так: `x << n` и `x >> n` (сдвиг), `x & 0xFF` (маска: оставить младшие 8 бит), `x | y` (объединение), `(x >> 16) & 0xFF` (достать нужный байт). Именно так в программе WS2812 цвета упаковываются в одно 24-битное слово: `(g << 16) | (r << 8) | b`.

14.8 WS2812 на PIO

Теперь протокол из урока 11. Для каждого бита линия должна быть: бит «0» — коротко в 1, долго в 0; бит «1» — долго в 1, коротко в 0. Период — $1{,}25$ µs, на частоте 8 MHz это ровно 10 тактов (по 125 нс). Разобьём период на три части: $T_3 = 3$ такта (начало бита, линия 0), $T_1 = 2$ такта (линия 1 в начале любого бита), $T_2 = 5$ тактов (для «1» линия остаётся 1, для «0» опускается).

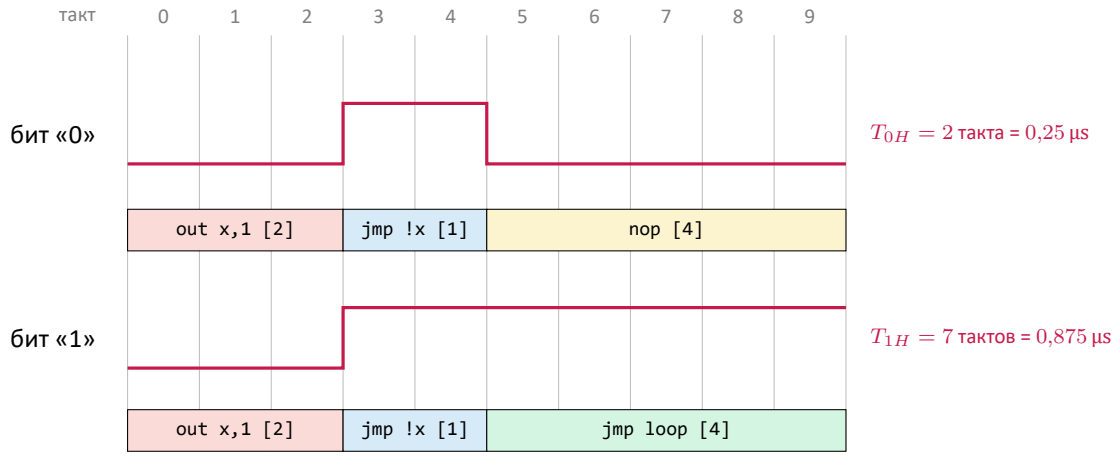


Рис. 14.8: Разложение периода WS2812 на такты: три команды для «0» и три для «1»; красная линия — уровень на выводе (side-set 0, 1, 1 или 0)

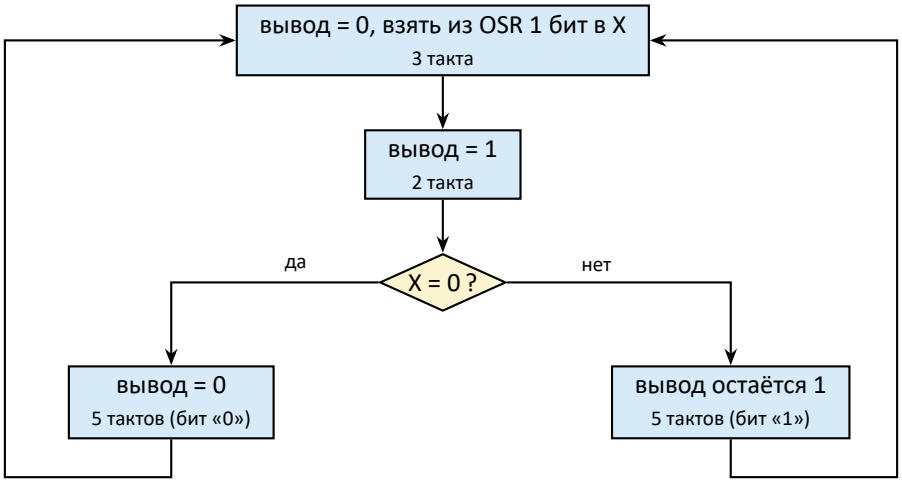


Рис. 14.9: Блок-схема программы WS2812: цикл на один бит; `autopull` сам подкладывает следующие 24 бита

Важная деталь — **autopull**: автомат сам берёт новое слово из TX FIFO, когда в OSR остаётся меньше заданного порога (`pull_thresh=24`). Поэтому процессору достаточно положить в очередь слова-цвета, а остальное автомат сделает сам, и на передачу кадра процессор тратит минимум времени. Слова при этом надо сдвинуть на 8 бит влево (`sm.put(массив, 8)`), чтобы нужные 24 бита оказались «слева», ведь выдвигается старший бит.

Листинг 14.3: MicroPython: WS2812 на PIO, бегущая радуга

```
import array
import time
import rp2
from machine import Pin

NUM_LEDS = 12


@rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT,
             autopull=True, pull_thresh=24)     # каждые 24 бита автомат сам берёт новое слово
def ws2812():
    T1 = 2                          # такты: T1 + T2 + T3 = 10 тактов = 1,25 мкс при 8 МГц
    T2 = 5
    T3 = 3
    wrap_target()
    label("bitloop")
    out(x, 1)               .side(0)    [T3 - 1]    # взять очередной бит; линия 0
    jmp(not_x, "do_zero")   .side(1)    [T1 - 1]    # начало импульса: линия 1
    jmp("bitloop")          .side(1)    [T2 - 1]    # бит «1»: ещё держим 1
    label("do_zero")
    nop()                   .side(0)    [T2 - 1]    # бит «0»: опускаем линию
    wrap()


sm = rp2.StateMachine(0, ws2812, freq=8_000_000, sideset_base=Pin(22))
sm.active(1)

pixels = array.array("I", [0] * NUM_LEDS)       # слово пикселя: 0x00GGRRBB


def wheel(pos):
    """Радужный цвет по числу 0..255 -> (R, G, B)"""
```

```
    if pos < 85:
        return 255 - pos * 3, pos * 3, 0
    if pos < 170:
        pos -= 85
        return 0, 255 - pos * 3, pos * 3
    pos -= 170
    return pos * 3, 0, 255 - pos * 3


def show(brightness=0.15):
    """Упаковать цвета в формат GRB, ослабить яркость и отправить в FIFO"""
    frame = array.array("I", [0] * NUM_LEDS)
    for i, c in enumerate(pixels):
        g = int(((c >> 16) & 0xFF) * brightness)
        r = int(((c >> 8) & 0xFF) * brightness)
        b = int((c & 0xFF) * brightness)
        frame[i] = (g << 16) | (r << 8) | b
    sm.put(frame, 8)                # сдвиг на 8 бит: нужные 24 бита оказываются «слева»
    time.sleep_ms(1)                # пауза > 50 мкс: сброс, лента применяет цвета


offset = 0
while True:
    for i in range(NUM_LEDS):
        r, g, b = wheel((i * 256 // NUM_LEDS + offset) & 255)
        pixels[i] = (g << 16) | (r << 8) | b
    show()
    offset = (offset + 4) & 255
    time.sleep_ms(20)
```

В **CircuitPython** аналогичная программа не нужна: библиотека `neopixel` использует аппаратные возможности RP2040 сама. Если же тебе нужен собственный протокол (например, ленты с другим порядком битов), используй `rp2pio` по образцу следующих примеров. Сравни с урок 11: там мы вызывали готовую библиотеку, здесь показано, что происходит «под капотом».

14.9 Измеряем импульс

PIO умеет и принимать. Программа ниже измеряет ширину положительного импульса на выводе: ждёт фронт, затем в цикле вычитает единицу из X, пока вывод равен 1, а в конце отправляет X в RX FIFO. Один проход цикла занимает 2 такта (`jmp x_dec` и `jmp pin`). Начиная с $X = \texttt{0xFFFFFFFF}$ (получаем через `mov(x, invert(null))`), по окончании импульса число прошедших проходов равно $\texttt{0xFFFFFFFF} - X$, а ширина

$$t = \frac{2 \cdot (\texttt{0xFFFFFFFF} - X)}{f_{\text{SM}}}.$$

При $f_{\text{SM}} = 10\,\text{MHz}$ точность равна $0{,}2\,\mu\text{s}$. Именно так измеряют эхо-импульс ультразвукового датчика расстояния HC-SR04.

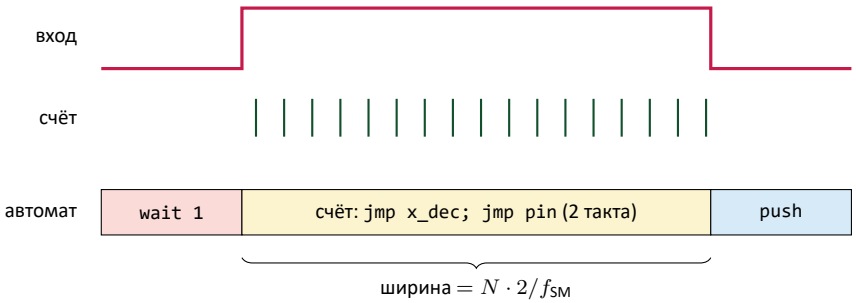


Рис. 14.10: Измерение импульса: пока вход равен 1, автомат уменьшает X; затем отправляет результат процессору

Листинг 14.4: MicroPython: измеритель ширины импульса на PIO

```
import time
import rp2
from machine import Pin, PWM

FREQ = 10_000_000                   # частота автомата, Гц


@rp2.asm_pio()
def pulse_len():
    wrap_target()
    wait(0, pin, 0)                 # дождаться низкого уровня (пропускаем начатый импульс)
    wait(1, pin, 0)                 # дождаться фронта: началась единица
    mov(x, invert(null))            # x = 0xFFFFFFFF
    label("count")
    jmp(x_dec, "check")             # x -= 1 (в любом случае идём дальше)
    label("check")
    jmp(pin, "count")               # пока на выводе 1 --- считаем
    mov(isr, x)                     # импульс кончился: остаток счётчика в ISR
    push(noblock)                   # и в приёмный FIFO
    wrap()


sm = rp2.StateMachine(0, pulse_len, freq=FREQ, in_base=Pin(15, Pin.IN), jmp_pin=Pin(15))
sm.active(1)

# Испытательный сигнал: GP14 выдаёт 1 кГц с заполнением 30 %. Соедини GP14 с GP15 проводом
pwm = PWM(Pin(14), freq=1000, duty_u16=int(0.3 * 65535))

while True:
    raw = sm.get() & 0xFFFFFFFF     # ждём результат от автомата
    count = 0xFFFFFFFF - raw        # сколько раз выполнился счётный цикл
    width_us = count * 2 * 1e6 / FREQ   # цикл занимает 2 такта
    print("ширина импульса: {:.1f} мкс".format(width_us))   # ожидаем около 300
    time.sleep(0.5)
```

Листинг 14.5: CircuitPython: то же

```
import array
import time
import board
import pwmio
import rp2pio
import adafruit_pioasm

FREQ = 10_000_000                   # частота автомата, Гц

program = adafruit_pioasm.assemble("""
.program pulse_len
    wait 0 pin 0                ; дождаться низкого уровня
    wait 1 pin 0                ; дождаться фронта
    mov x ~ null                ; x = 0xFFFFFFFF
count:
    jmp x-- check               ; x -= 1 (идём дальше в любом случае)
check:
    jmp pin count               ; пока на выводе 1 --- считаем
    mov isr x                   ; результат в ISR
    push noblock                ; и в приёмный FIFO
""")

sm = rp2pio.StateMachine(program, frequency=FREQ,
                         first_in_pin=board.GP15, jmp_pin=board.GP15)

# Испытательный сигнал: GP14 выдаёт 1 кГц с заполнением 30 %. Соедини GP14 с GP15 проводом
pwm = pwmio.PWMOut(board.GP14, frequency=1000, duty_cycle=int(0.3 * 65535))

buf = array.array("I", [0])
while True:
    sm.readinto(buf)                # ждём одно 32-битное слово
    count = 0xFFFFFFFF - buf[0]
    width_us = count * 2 * 1e6 / FREQ
    print("ширина импульса: {:.1f} мкс".format(width_us))   # ожидаем около 300
    time.sleep(0.5)
```

При соединённых **GP14** и **GP15** программа должна показывать значения около 300 мкс (заполнение 30 % при 1 kHz). Обрати внимание, как использованы параметры: `in_base` (**MicroPython**) и `first_in_pin` (**CircuitPython**) выбирают вывод для `wait(pin)`, а `jmp_pin` — вывод, который проверяет `jmp(pin, ...)`.

14.10 UART-передатчик на PIO

Скорости обмена и формат кадра UART вы уже знаете (урок 7): линия в покое 1, стартовый бит 0, затем 8 бит данных (младший первым), стоп-бит 1. Аппаратных UART у Pico два, а нужно больше? PIO позволяет сделать ещё один передатчик на любом выводе. Трюк: делаем частоту автомата равной $8\times$ скорость обмена; тогда на каждый бит приходится ровно 8 тактов.

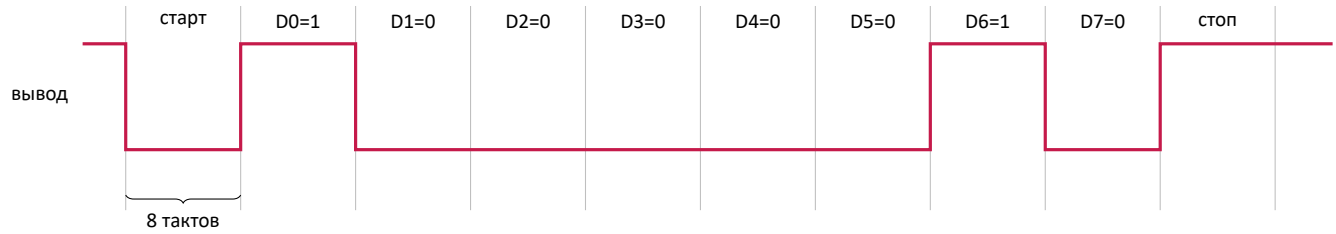


Рис. 14.11: Кадр UART для байта `0x41` (буква A): линия, 8 тактов автомата на каждый бит

Листинг 14.6: MicroPython: UART TX на PIO (по примеру из набора примеров MicroPython)

```
import time
import rp2
from machine import Pin


@rp2.asm_pio(sideset_init=rp2.PIO.OUT_HIGH, out_init=rp2.PIO.OUT_HIGH,
             out_shiftdir=rp2.PIO.SHIFT_RIGHT)      # UART отправляет младший бит первым
def uart_tx():
    # Пока в FIFO ничего нет, автомат стоит на pull(), а линия в состоянии покоя (1)
    pull()                                          # взять байт из FIFO в OSR (1 такт)
    set(x, 7)       .side(0)    [7]                 # старт-бит: линия 0 на 8 тактов; x = 7
    label("bitloop")
    out(pins, 1)                [6]                 # выдать один бит из OSR на вывод (1 + 6 = 7 тактов)
    jmp(x_dec, "bitloop")                           # ещё один такт: всего 8 тактов на бит
    nop()           .side(1)    [6]                 # стоп-бит: линия 1 (7 тактов) + pull() = 8


BAUD = 115200
# 8 тактов на бит: частота автомата = 8 x скорость обмена
sm = rp2.StateMachine(0, uart_tx, freq=8 * BAUD, sideset_base=Pin(0), out_base=Pin(0))
sm.active(1)

for ch in "Hello, PIO!\r\n":
    sm.put(ord(ch))                 # каждый байт --- одно слово FIFO
time.sleep_ms(20)
```

Листинг 14.7: CircuitPython: то же

```
import time
import board
import rp2pio
import adafruit_pioasm

# Передатчик UART 8N1. Необязательный side-set (.side_set 1 opt) задаёт старт- и стоп-биты
program = adafruit_pioasm.assemble("""
.program uart_tx
.side_set 1 opt
    pull                side 1 [7]   ; стоп-бит или ожидание данных (линия 1)
    set x, 7            side 0 [7]   ; старт-бит на 8 тактов
bitloop:
    out pins, 1                      ; один бит из OSR на вывод
    jmp x-- bitloop           [6]    ; 8 тактов на бит
""")

BAUD = 115200
```

```
sm = rp2pio.StateMachine(
    program,
    frequency=8 * BAUD,                    # 8 тактов на бит
    first_out_pin=board.GP0, first_sideset_pin=board.GP0,
    initial_out_pin_state=1, initial_out_pin_direction=1,
    initial_sideset_pin_state=1, initial_sideset_pin_direction=1,
    sideset_enable=True,                   # side-set «opt»
    out_shift_right=True,                  # младший бит первым
)

sm.write(b"Hello, PIO!\r\n")               # каждый байт --- одно слово в FIFO
time.sleep(0.1)
```

Что происходит: команда `pull()` ждёт байт в очереди, пока линия остаётся в покое (последний side-set оставил 1). Затем `set(x, 7).side(0)[7]` опускает линию на 8 тактов (старт-бит) и записывает в X число 7, а цикл `out(pins, 1)[6]; jmp(x_dec, ...)` выдаёт по одному биту (8 тактов на бит, 8 раз). В конце `nop().side(1)[6]` — стоп-бит. Пара `sideset_base` и `out_base` указывает один и тот же вывод.

14.11 PIO и CircuitPython: `rp2pio` и `adafruit_pioasm`

В **CircuitPython** за PIO отвечает модуль `rp2pio` (встроен в прошивки для RP2040/RP2350), а текст ассемблера превращает в слова библиотека `adafruit_pioasm`. Основные соответствия:

	MicroPython	CircuitPython
Программа	функция с `@rp2.asm_pio`	строка, `adafruit_pioasm.assemble(...)`
Создание автомата	`rp2.StateMachine(id, prog, freq=...)`	`rp2pio.StateMachine(prog, frequency=...)`
Вывод для `set`	`set_base=Pin(n)`	`first_set_pin=board.GPn`
Вывод для `out`	`out_base=Pin(n)`	`first_out_pin=board.GPn`
Вывод для `in` / `jmp pin`	`in_base=`, `jmp_pin=`	`first_in_pin=`, `jmp_pin=`
Side-set	`sideset_base=`, `sideset_init=`	`first_sideset_pin=`, `sideset_enable=`
Передача данных	`sm.put(x)` (слово)	`sm.write(buf)` (буфер)
Приём	`sm.get()`	`sm.readinto(buf)`
Запуск и остановка	`sm.active(1/0)`	создание и `sm.deinit()`
Номер автомата/блока	выбирает программист (0–7)	выбирает система

Таблица 14.2: Сопоставление интерфейсов **MicroPython** и **CircuitPython** для PIO

Ограничения у **CircuitPython**: автомат стартует сразу при создании, а имена параметров могут отличаться в разных версиях (сверяйся с документацией `rp2pio` для своей версии), а сборщик `adafruit_pioasm` понимает почти весь синтаксис официального ассемблера PIO, но не всё (например, часть директив препроцессора). Примеры этого урока проверяй в той версии, которую ставишь.

14.12 Ограничения PIO и RP2350

- **32 команды** на блок: программа должна быть короткой. Хитрость — пользоваться `x_dec`, задержками `[n]` и `wrap` вместо лишних команд.
- **Нет арифметики**: только сравнение, уменьшение на единицу, сдвиги и логика. Сложные расчёты — в программе процессора.

- **Нет прерываний «изнутри»**: PIO может поднять флаг (`irq`), по которому процессор вызовет обработчик (`sm.irq(handler)` в MicroPython).
- **Отладка трудная**: ошибка обычно проявляется как «ничего не происходит». Помогают логический анализатор и осциллограф, а в MicroPython — пошаговое выполнение командой `sm.exec(...)`.
- Автомат тратит на работу время процессора только при обмене данными; для длинных передач данные кладут через DMA, но в этом уроке мы обходимся `put/get`.

Это интересно

В микроконтроллере RP2350 (Pico 2) блоков PIO уже **три**, то есть **12 автоматов**; память команд всё так же по 32 слова на блок. Добавлены новые возможности (например, больше вариантов команды `mov`, косвенные флаги прерываний и выбор окна из 32 выводов для платформ с 48 выводами). В MicroPython для Pico 2 номера автоматов продолжаются до 11, а в программах для RP2040 ничего менять не нужно. Подробнее — в уроке 24.

Если не работает

- **Ничего не происходит** — забыт `sm.active(1)` (в MicroPython), не указаны выводы (`set_base`, `sideset_base`) или не выставлен нужный `*_init`; проверь, что вывод не занят другой периферией.
- **Мигает не с той частотой** — неверно посчитаны такты. Каждая команда занимает $1 + n$ тактов; проверь делитель: `freq` не может быть ниже $\approx 1{,}9$ kHz при 125 MHz.
- **`ValueError: ... too many instructions` / программа не помещается** — в блоке всего 32 команды на все автоматы; упрости программу или освободи память (`rp2.PIO(0).remove_program()`).
- **Лента WS2812 показывает не те цвета** — перепутан порядок G–R–B или не выполнен сдвиг на 8 бит в `sm.put(..., 8)`.
- **Бессмысленные результаты измерения** — перемычка не на тех выводах, измеряемый вывод не настроен как вход, либо частота автомата слишком велика (счётчик успевает переполниться).
- **UART выдаёт «мусор»** — частота автомата не равна $8\times$ скорость, либо приёмник настроен на другую скорость.
- **В CircuitPython ошибка в тексте программы** — сборщик сообщает номер строки; обрати внимание на запятые, регистр и скобки задержек.

Итоги урока

- PIO — это 2 блока по 4 автомата, у каждого FIFO TX/RX, регистры OSR/ISR, «блокноты» X и Y и общая на блок память на 32 команды.
- Всего девять команд (`jmp`, `wait`, `in`, `out`, `push`, `pull`, `mov`, `irq`, `set`); каждая — один такт плюс необязательная задержка `[n]` и значение side-set.
- Скорость выбирается делителем: $f_{\text{SM}} = f_{\text{sys}}/\text{div}$; точность — один такт.
- MicroPython: `rp2.asm_pio`, `rp2.StateMachine`, `sm.put`, `sm.get`, `sm.active`; CircuitPython: строка ассемблера, `adafruit_pioasm`, `rp2pio.StateMachine`, `write`, `readinto`.
- PIO формирует точные протоколы (WS2812, UART) и измеряет импульсы, оставляя процессор свободным.

- В RP2350 — 3 блока и 12 автоматов.

Проверь себя

1. Сколько автоматов в одном блоке PIO RP2040 и сколько команд помещается в память блока?
2. Как рассчитать длительность одной команды PIO, если системная частота 125 MHz, а делитель равен 15,625? Сколько тактов занимает команда `nop() [3]`?
3. Что такое side-set и чем он выгоднее отдельной команды `set`?
4. Зачем в программе WS2812 включён `autopull` и почему слова сдвигают на 8 бит влево?
5. Как программа измеряет ширину импульса и от чего зависит точность измерения?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Запусти мигалку PIO и измени частоту автомата (`freq`) так, чтобы светодиод мигал вдвое быстрее. Какова минимально допустимая `freq`?
2. ★ Рассчитай по формуле, сколько тактов нужно на половину периода мигания 5 Гц при частоте автомата 2000 Hz, и подбери задержки и счётчик.
3. ★★ Измени ширину испытательного импульса (заполнение 10 – 90 %) и проверь, как меняются показания измерителя. Насколько точно они совпадают с расчётом?
4. ★★ Передай по PIO-UART строку и прочитай её через USB-UART-адаптер или другой UART Pico (урок 7); подбери одинаковую скорость 9600 и 115200.
5. ★★★ Напиши программу PIO, которая выдаёт на вывод меандр 1 кГц (командами `set`, `nop` и задержками) и сравни её с аппаратным PWM по стабильности частоты при высокой нагрузке процессора.
6. ★★★★ Реализуй на PIO передатчик протокола с *двумя* выводами (данные и такт, как у простого SPI с режимом 0) со скоростью 1 MHz; используй side-set для сигнала такта и `out` для данных.

Урок 15. Питание, сон и сторожевой таймер

Цель	понять, откуда плата берёт энергию, как сделать устройство автономным, заставить его спать, измерять заряд батареи и самостоятельно «лечиться» после зависания
Опирается на	урок 6 (АЦП), урок 4 (таймеры и время), урок 2 (светодиод)
Результат	плата работает от батарей; программа спит между измерениями и просыпается по таймеру или кнопке; напряжение батареи измеряется через делитель; сторожевой таймер перезапускает «зависшую» программу
Понадобится	Pico или Pico W, держатель 2–3 батареек AA, диод Шоттки (например, 1N5819), два резистора 100 kΩ, кнопка, мультиметр, макетная плата

15.1 Идея урока коротко

До сих пор плата питалась от USB-кабеля. Настоящие устройства — метеостанция в огороде, датчик в теплице, умный брелок — должны жить на батарейках. А у батарейки ограничен запас энергии, поэтому главный вопрос звучит так: *как тратить меньше, чем тратит процессор по умолчанию?* Ответов три: питать плату правильно, **больше спать** и не делать лишнюю работу. Четвёртая тема — надёжность: устройство, которое никто не видит, должно само выходить из зависаний. Для этого есть **сторожевой таймер** (watchdog).

15.2 Откуда плата берёт энергию

На плате Pico три важных «питающих» вывода. Их полезно запомнить как маленькую систему водопровода:

Вывод	Напряжение	Что это
VBUS	5 V	Напряжение прямо с USB-разъёма. Есть только когда подключён кабель.
VSYS	1,8 – 5,5 V	«Системная» шина питания. Сюда можно подать питание от батарей. От VSYS работает регулятор 3,3 В.
3V3	3,3 V	*Выход* регулятора. Питает процессор, память и все ваши датчики. Подавать сюда напряжение извне нельзя (кроме случаев, описанных в документации).
3V3_EN	—	Вход включения регулятора. Если замкнуть его на землю, регулятор выключится, и плата погаснет.
GND	0 V	Земля. Общий «ноль» всей схемы.

Таблица 15.1: Выводы питания Raspberry Pi Pico

Устройство питания можно нарисовать цепочкой блоков. Источник (USB или батареи) попадает на шину VSYS, регулятор превращает её в стабильные 3,3 В, а уже от них питается всё остальное:

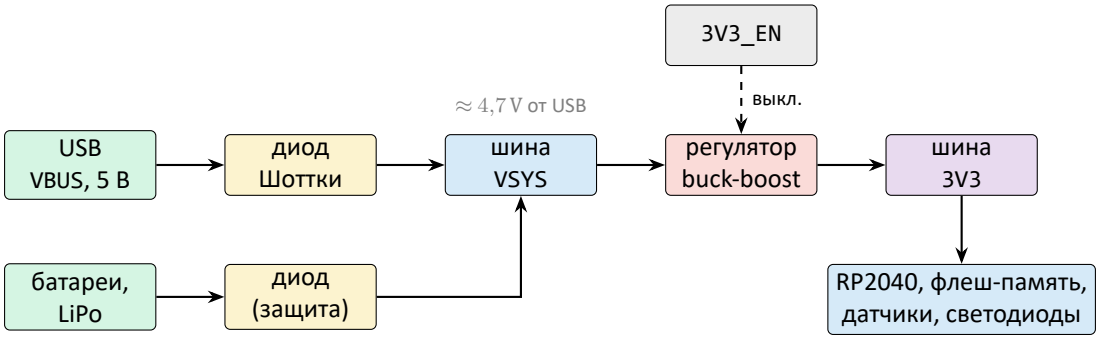


Рис. 15.1: Путь энергии на плате Pico: от источника к процессору

Диод-ИЛИ: два источника без драки

Что будет, если подключить и USB, и батареи сразу? Без защиты источники начнут «качать» ток друг в друга: USB попытается зарядить обычные батарейки, а они — подать напряжение на разъём компьютера. Решение — **диод**: он пропускает ток только в одну сторону. Если на каждый источник поставить по диоду, а выходы диодов соединить, получится схема «ИЛИ»: на общей шине окажется напряжение того источника, у которого оно выше.

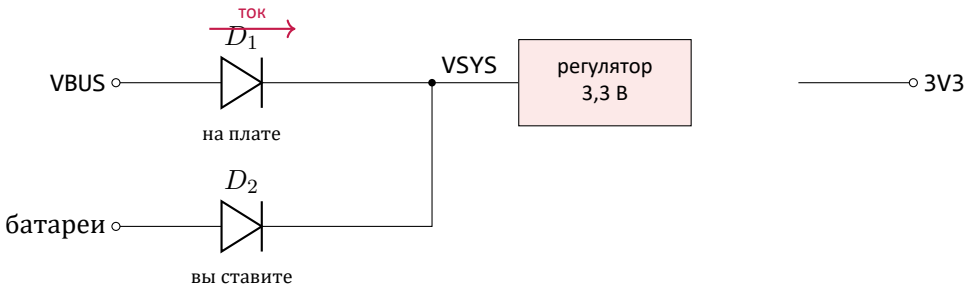


Рис. 15.2: Принципиальная схема «диод-ИЛИ»: на VSYS попадает напряжение более «сильного» источника

Диод не бесплатный: на нём теряется $0{,}2$ – $0{,}5$ V. На Pico для этого стоит диод Шоттки (у него потери меньше). Поэтому от USB на VSYS приходит около $4{,}7$ V, а не 5 V.

Осторожно

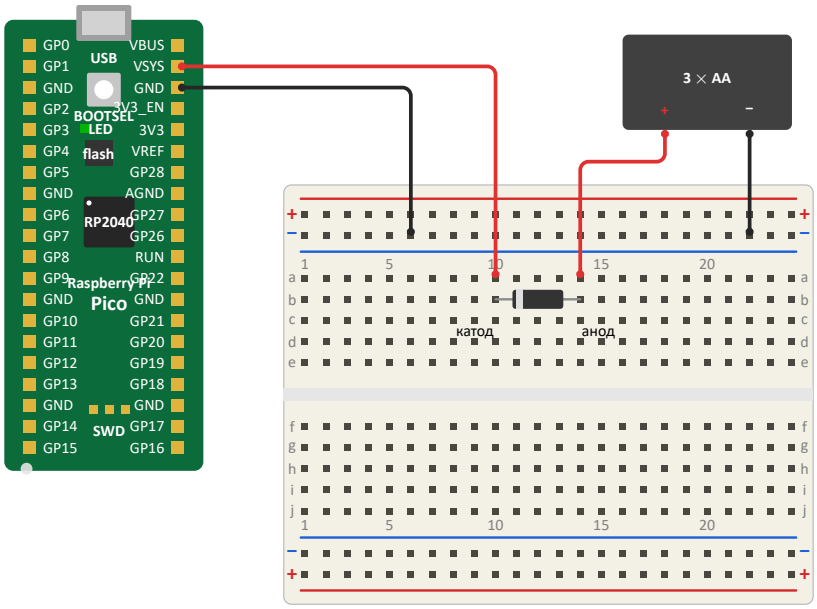
- **Никогда не подавайте 5 V на вывод 3V3 или на GPIO**, а на VSYS — не больше $5{,}5$ V.
- Если запитываете плату *через VSYS* и одновременно втыкаете USB, между батареями и VBUS должен стоять диод (в схеме выше это D_2): иначе энергия потечёт «обратно». Тот же диод защищает и от неправильной полярности батарей.
- Выводом 3V3 (OUT) можно питать датчики, но не моторы, реле и мощные светодиодные ленты: регулятор рассчитан примерно на 300 mA с запасом на нужды самой платы.
- Всегда проверяйте полярность перед включением: **красный — плюс, чёрный — минус**.

15.3 Питание от батареек

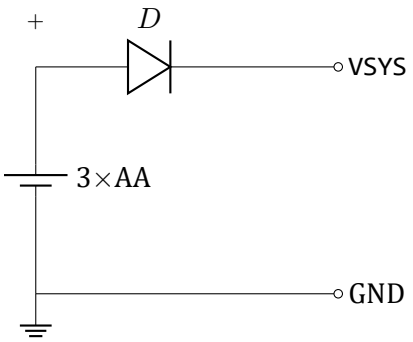
Батареи надо подбирать так, чтобы их напряжение укладывалось в окно $1{,}8$ – $5{,}5$ V — это окно шины VSYS. Нас выручает то, что регулятор Pico *повышающе-понижающий*: он умеет и снижать напряжение (с 5 V до 3,3), и повышать (с $2{,}2$ V до 3,3). Поэтому батарея может сильно «просесть», а плата продолжит работать.

Источник	Напряжение, В	Ёмкость	Замечания
2 щелочные AA	3,2 → 1,8	≈ 2500 мА·ч	Работает почти до конца, ограниченный ток.
3 щелочные AA	4,8 → 2,7	≈ 2500 мА·ч	Лучший вариант с диодом: $4{,}8 - 0{,}3 = 4{,}5$ В.
4 щелочные AA	6,4	—	**Слишком много!** Выше $5{,}5$ V: плата может выйти из строя.
Li-Ion/LiPo 1S	4,2 → 3,0	500–3000 мА·ч	Нужны плата заряда и защиты. Можно подключать к VSYS.
Блок питания 5 V	5,0	—	К VBUS или к VSYS (через диод).

Таблица 15.2: Источники питания для Pico



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 15.3: Питание Pico от трёх батареек AA через диод к выводу VSYS

Красный провод с батареи идёт на анод диода, катод (серая полоса на корпусе) соединён с VSYS. Чёрный провод и GND платы соединены общей шиной. При подключённом USB плата берёт больший из двух уровней, а диод не даёт энергии течь в батареи.

Литий-полимерный аккумулятор

Аккумулятор Li-Ion/LiPo ёмче и лёгче щелочных батареек, и его можно заряжать — но он гораздо опаснее. Перегрев, прокол, короткое замыкание или перезаряд могут привести к пожару. Поэтому действуют жёсткие правила:

Осторожно

- Заряжать аккумулятор можно **только через специальный модуль заряда с защитой** (например, на микросхемах TP4056 + DW01 или готовые платы «Pico LiPo»). Подключать его к зарядке «напрямую» — нельзя.
- Берите аккумуляторы с **встроенной платой защиты** (от перезаряда, глубокого разряда и короткого замыкания).
- Полярность разъёмов JST у разных производителей **может быть разной**. Проверяйте мультиметром, а не по цвету проводов.
- Не оставляйте заряжающийся аккумулятор без присмотра; не заряжайте на ковре, диване, бумаге.
- Вздувшийся, повреждённый или горячий аккумулятор — не использовать, сдать на утилизацию.
- Работы с LiPo школьникам до 14 лет — только вместе со взрослыми.

Идея схемы автономного устройства показана на рисунке: аккумулятор через плату заряда питает VSYS, а при подключении USB-питания плата заряда заряжает аккумулятор.

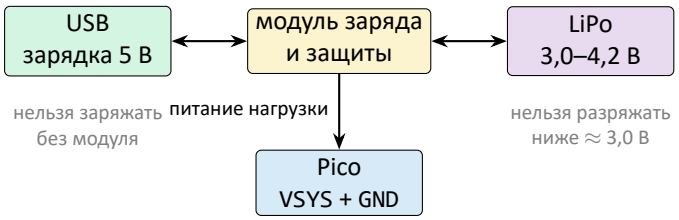


Рис. 15.4: Схема питания от аккумулятора: заряд и нагрузка проходят через защитный модуль

15.4 Сколько тратит плата

Закон энергии простой. Заряд батареи считают в **миллиампер-часах** (мА·ч): батарея на 2000 mAh в принципе может отдавать 1 mA в течение 2000 часов или 100 mA в течение 20 часов. Значит, время работы:

$$T = \frac{Q}{I}.$$

Плата Pico в обычном режиме потребляет порядка 20 – 25 mA (это ориентировочная цифра, реальный ток зависит от версии платы, программы и подключённых деталей). Тогда от батареи 2500 mAh она проработает $\frac{2500}{22} \approx 110$ часов — меньше пяти суток. Для метеостанции это беда.

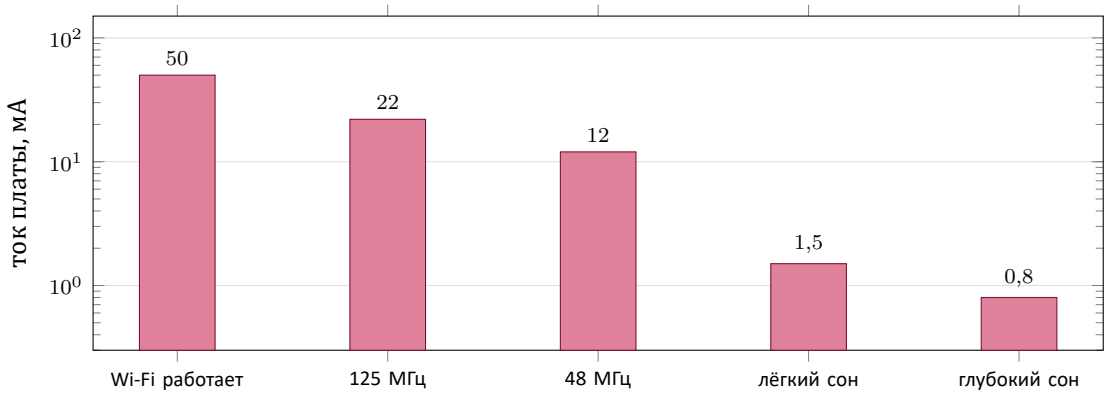


Рис. 15.5: Порядок величин тока платы Pico в разных режимах (логарифмическая шкала; значения ориентировочные, у Pico W больше)

Из графика видно главное: сон уменьшает ток не в два раза, а в *десятки* раз. Но даже в самом глубоком сне плата не бывает «нулевой»: часть энергии съедает сам регулятор, светодиод питания (если он есть) и радиочип на Pico W. Реальные цифры измеряйте мультиметром.

Совет

Чтобы измерить ток, мультиметр включают *последовательно* в разрыв провода питания, на пределе mA (сначала на самом грубом, 200 mA). Если включить амперметр параллельно, как вольтметр, он станет коротким замыканием.

15.5 Снижение тактовой частоты

Процессор потребляет тем больше, чем чаще он переключает транзисторы. Приближённо ток растёт с частотой как $I \approx I_0 + kf$: небольшая постоянная часть плюс слагаемое, пропорциональное частоте.

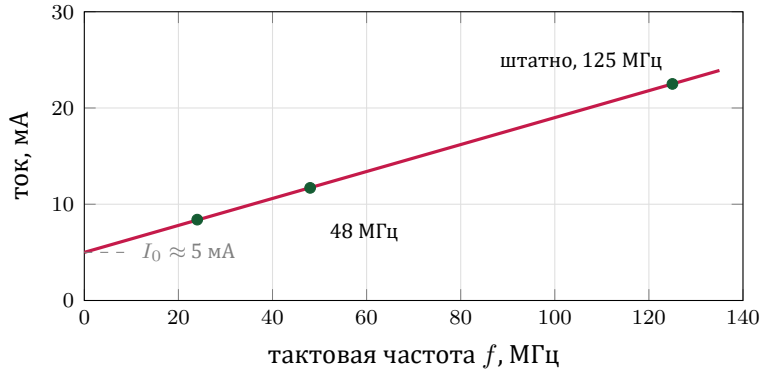


Рис. 15.6: Модель зависимости тока от частоты ($I \approx 5 + 0{,}14\,f$ мА; приближённо)

Понизить частоту можно одной командой. Но за экономию приходится платить скоростью: код выполняется медленнее, а те устройства, чьи тактовые сигналы выводятся из системной частоты (UART, I2C, SPI, PWM, PIO), изменят свои скорости. USB, как правило, работает от отдельного генератора, но при частотах ниже примерно 20 MHz связь с компьютером может стать нестабильной.

Листинг 15.1: MicroPython: меняем частоту и смотрим, как долго выполняется работа

```
import machine
import time

print("Частота по умолчанию:", machine.freq() // 1_000_000, "МГц")


def work(n):
    """Тестовая нагрузка: считаем сумму квадратов."""
    s = 0
    for i in range(n):
        s += i * i
    return s


for f in (125_000_000, 48_000_000, 24_000_000):
    machine.freq(f)                          # меняем тактовую частоту ядер
    t0 = time.ticks_us()
    work(20000)
    dt = time.ticks_diff(time.ticks_us(), t0)
    print(machine.freq() // 1_000_000, "МГц:", dt / 1000, "мс")

machine.freq(125_000_000)                    # вернём штатную частоту
```

Листинг 15.2: CircuitPython: то же

```
import microcontroller
import time

print("Частота по умолчанию:", microcontroller.cpu.frequency // 1_000_000, "МГц")


def work(n):
    """Тестовая нагрузка: считаем сумму квадратов."""
    s = 0
    for i in range(n):
        s += i * i
    return s


for f in (125_000_000, 48_000_000, 24_000_000):
    microcontroller.cpu.frequency = f        # на RP2040 частоту можно менять
    t0 = time.monotonic()
    work(20000)
    dt = time.monotonic() - t0
    print(microcontroller.cpu.frequency // 1_000_000, "МГц:", round(dt * 1000), "мс")

microcontroller.cpu.frequency = 125_000_000  # вернём штатную частоту
```

Это интересно

Частота RP2040 легко разгоняется выше штатных 125 MHz — до 200 – 250 MHz. Поэтому иногда экономят энергию наоборот: быстро делают всю работу на высокой частоте и сразу засыпают. Что выгоднее, зависит от задачи — проверяйте измерением.

15.6 Сон: идея и режимы

Лучший способ сэкономить — ничего не делать. Большую часть времени типичный датчик ждёт: температура меняется медленно, и измерять её достаточно раз в минуту. В остальное время процессор может спать.
У микроконтроллеров обычно несколько «глубин» сна:

- **Лёгкий сон (light sleep).** Процессор останавливается, но всё содержимое памяти (переменные) сохраняется. После пробуждения программа *продолжается со следующей строки*. Ток снижается сильно, пробуждение быстрое.
- **Глубокий сон (deep sleep).** Отключается ещё больше блоков. Ток минимален, но память может быть потеряна, и программа начинается заново (как после включения). Чтобы что-то помнить, данные нужно сохранять в файл или в специальную «память сна».

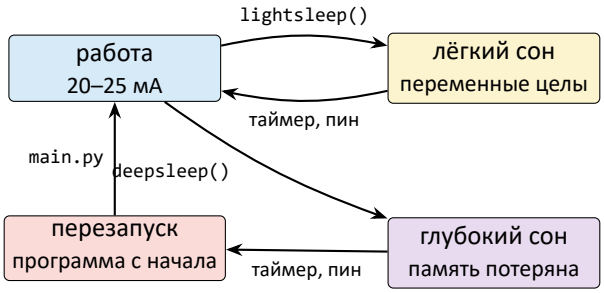


Рис. 15.7: Режимы работы и переходы между ними

15.7 Лёгкий сон

MicroPython предлагает функцию `machine.lightsleep(время_мс)`. Если время не указано, плата спит до внешнего события. **CircuitPython** использует модуль `alarm`: сначала создаётся «будильник» (по времени или по выводу), потом вызывается `alarm.light_sleep_until_alarms(будильник)`.

Листинг 15.3: **MicroPython**: пять циклов «вспышка — сон на 5 секунд»

```
from machine import Pin, lightsleep
import time

led = Pin("LED", Pin.OUT)

for i in range(5):
    led.on()                  # короткая вспышка: «я бодрствую»
    time.sleep_ms(200)
    led.off()
    print("цикл", i, "- засыпаю на 5 секунд")
    lightsleep(5000)          # лёгкий сон; через 5000 мс программа продолжится
    print("проснулся")
```

Листинг 15.4: CircuitPython: то же с будильником по времени

```
import time
import board
import digitalio
import alarm

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

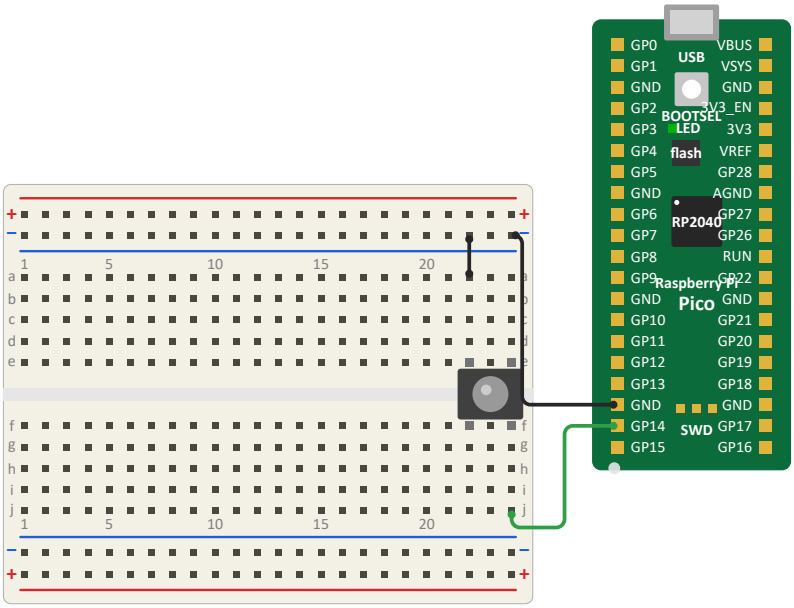
for i in range(5):
    led.value = True                          # короткая вспышка: «я бодрствую»
    time.sleep(0.2)
    led.value = False
    print("цикл", i, "- засыпаю на 5 секунд")
    # будильник по времени: monotonic_time --- момент пробуждения
    wake = alarm.time.TimeAlarm(monotonic_time=time.monotonic() + 5)
    alarm.light_sleep_until_alarms(wake)      # лёгкий сон; программа продолжится дальше
    print("проснулся")
```

Осторожно

Во время сна плата может *пропадать из USB*: в Thonny связь обрывается, консоль замолкает, а после пробуждения возвращается. Это нормально: USB тоже «засыпает». Для отладки используйте светодиод или сначала проверяйте программу с короткими интервалами.

Как работает пробуждение

Любой сон можно прервать **событием-будильником**. Их источников два: время (таймер) и внешний вывод (кнопка, датчик). В CircuitPython оба оформлены классами: `alarm.time.TimeAlarm` и `alarm.pin.PinAlarm`. В MicroPython на RP2 надёжно работает пробуждение по таймеру; пробуждение по выводу зависит от версии прошивки и режима, поэтому надёжный универсальный приём — «просыпаться раз в 100 ms и смотреть, нажата ли кнопка». Короткие пробуждения почти не увеличивают средний ток, а реакция на кнопку остаётся быстрой.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 15.8: Кнопка для пробуждения: GP14 и общий провод

Кнопка соединяет **GP14** с землёй. Внутренний подтягивающий резистор держит вход на «1», а нажатие даёт «0». Как раз такая схема и нужна для пробуждения: «будильник сработает, когда на выводе будет низкий уровень».

Листинг 15.5: MicroPython: сон короткими отрезками с проверкой кнопки

```
from machine import Pin, lightsleep
import time

button = Pin(14, Pin.IN, Pin.PULL_UP)      # кнопка на землю: нажата = 0
led = Pin("LED", Pin.OUT)

last = time.ticks_ms()
while True:
    lightsleep(100)                        # спим короткими отрезками по 100 мс
    if button.value() == 0:                # проснулись и увидели нажатие
        led.on()
        print("кнопка нажата --- работаем!")
        time.sleep_ms(500)
        led.off()
    if time.ticks_diff(time.ticks_ms(), last) >= 10_000:
        last = time.ticks_ms()             # раз в 10 с --- плановый замер
        print("плановый замер")
```

Листинг 15.6: CircuitPython: пробуждение по таймеру или по кнопке на GP14

```
import time
import board
import alarm

while True:
    # два будильника: по времени (10 с) и по кнопке на GP14 (низкий уровень)
    t_alarm = alarm.time.TimeAlarm(monotonic_time=time.monotonic() + 10)
    p_alarm = alarm.pin.PinAlarm(pin=board.GP14, value=False, pull=True)
    woke = alarm.light_sleep_until_alarms(t_alarm, p_alarm)
    if isinstance(woke, alarm.pin.PinAlarm):
        print("разбудила кнопка")
    else:
        print("разбудил таймер")
    time.sleep(0.3)           # подавление дребезга и «разгон»
```

15.8 Глубокий сон

Глубокий сон вызывают `machine.deepsleep(мс)` в **MicroPython** и `alarm.exit_and_deep_sleep_until_alarms(...)` в **CircuitPython**. Главное отличие от лёгкого: после пробуждения в **CircuitPython** выполнение `code.py` всегда начинается *с самого начала*. Поэтому нужно знать, *из-за чего* мы проснулись. В **CircuitPython** это `alarm.wake_alarm`: он равен `None`, если мы только что включили плату.

В **MicroPython** на RP2 поведение зависит от версии прошивки: в одних сборках после сна программа продолжается с места вызова, в других — перезапускается. Пример ниже написан так, что верен в обоих случаях: измерение стоит в бесконечном цикле, и состояние не нужно хранить между пробуждениями.

Листинг 15.7: MicroPython: измерение раз в 10 секунд с глубоким сном

```
import machine
from machine import ADC, Pin
import time

sensor = ADC(4)                      # встроенный датчик температуры кристалла


def temp_c():
    v = sensor.read_u16() * 3.3 / 65535
    return 27 - (v - 0.706) / 0.001721


# Причина запуска: после включения, после сторожевого таймера или после сна
print("причина сброса:", machine.reset_cause())

while True:
    print("температура кристалла: %.1f °C" % temp_c())
    time.sleep_ms(100)               # дать тексту уйти в USB
    # Глубокий сон на 10 с. В одних версиях прошивки программа затем продолжится
    # с этого места, в других --- main.py начнётся с начала. Код верен в обоих случаях.
    machine.deepsleep(10_000)
```

Листинг 15.8: CircuitPython: тот же узел, счётчик пробуждений в `sleep_memory`

```
import time
import alarm
import microcontroller

# Что нас разбудило? None --- это первое включение питания или перезагрузка.
if alarm.wake_alarm is None:
    print("холодный старт")
    alarm.sleep_memory[0] = 0
else:
    print("проснулись от", alarm.wake_alarm)

# Память, переживающая глубокий сон (очень маленькая): считаем пробуждения
alarm.sleep_memory[0] = (alarm.sleep_memory[0] + 1) % 256
print("замер №", alarm.sleep_memory[0],
      "температура: %.1f °C" % microcontroller.cpu.temperature)

time.sleep(0.1)                                 # дать тексту уйти в USB
wake = alarm.time.TimeAlarm(monotonic_time=time.monotonic() + 10)
# После глубокого сна code.py всегда начинается с самого начала
alarm.exit_and_deep_sleep_until_alarms(wake)
```

Осторожно

Не пробуйте проверять глубокий сон с подключённой отладкой и открытой консолью Thonny без запаса: после первого же сна связь может пропасть, и плату придётся «будить» (перезапуском, кнопкой RUN или повторным включением). Держите под рукой простую программу-«спасатель» (например, мигалку) и умейте остановить программу Ctrl+C до первого сна. Если плата не отзывается — см. раздел «Если не работает».

Работаем циклами: «проснулся — измерил — заснул»

Типичное автономное устройство большую часть времени спит, а проснувшись, быстро делает дело. Тогда средний ток вычисляется как *взвешенное по времени среднее*:

$$I_{\text{ср}} = \frac{I_{\text{раб}}\, t_{\text{раб}} + I_{\text{сон}}\, t_{\text{сон}}}{t_{\text{раб}} + t_{\text{сон}}}.$$

Например, если плата работает $0{,}1$ s с током 22 mA, а потом спит $9{,}9$ s с током $1{,}5$ mA:

$$I_{\text{ср}} = \frac{22 \cdot 0{,}1 + 1{,}5 \cdot 9{,}9}{10} \approx 1{,}7 \text{ мА}, \qquad T = \frac{2500}{1{,}7} \approx 1470 \text{ ч} \approx 61 \text{ сутки}.$$

Вместо 4,5 суток — два месяца. Цена такого выигрыша — пара строк программы.

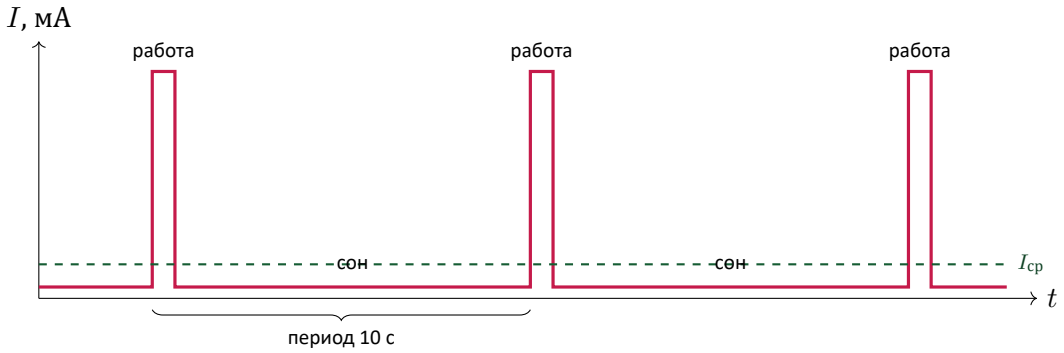


Рис. 15.9: Ток при работе по циклу «проснулся — сделал дело — заснул»

Режим программы	$I_{ср}$, мА	Время от 2500 мА·ч	Комментарий
Без сна, 125 МГц	22	≈ 4,7 суток	«Эталон» плохой экономии
Без сна, 48 МГц	12	≈ 8,7 суток	Лучше, но всё равно мало
Цикл: раз в 10 с, лёгкий сон	1,7	≈ 2 месяца	Хорошо
Цикл: раз в 1 мин, лёгкий сон	1,6	≈ 2,1 месяца	Упирается в ток сна

Таблица 15.3: Ориентировочное время работы от батареи (расчёт на примере; для настоящего прибора измеряйте)

Видно, что при частом пробуждении главный вклад вносит «рабочая» доля, а при редком — сам ток сна. Дальше экономят уже схемой: убирают светодиод питания, регулятор, ставят специальные платы с малым током покоя.

Это интересно

Плата Pico, собранная для универсальности, тратит в «глубоком сне» заметно больше, чем сам чип RP2040 (в режиме «dormant» он потребляет доли миллиампера): регулятор и обвязка платы потребляют свой ток всегда. Поэтому у серьёзных малопотребляющих проектов ставят специальные платы с мизерным током покоя.

Листинг 15.9: MicroPython: типичный «датчик на батарейке» с лёгким сном между замерами

```
from machine import ADC, Pin, lightsleep
import time

sensor = ADC(4)                  # датчик температуры внутри RP2040
led = Pin("LED", Pin.OUT)
INTERVAL_MS = 10_000             # период измерений


def temp_c():
    v = sensor.read_u16() * 3.3 / 65535
    return 27 - (v - 0.706) / 0.001721


def measure():
    """Среднее по 8 отсчётам: так шум меньше."""
    return sum(temp_c() for _ in range(8)) / 8


while True:
```

```
    t0 = time.ticks_ms()
    led.on()                      # индикатор активной работы (вспышка)
    t = measure()
    print("T = %.1f °C" % t)
    led.off()
    spent = time.ticks_diff(time.ticks_ms(), t0)
    lightsleep(max(1, INTERVAL_MS - spent))   # спим остаток периода
```

Листинг 15.10: CircuitPython: то же

```
import time
import board
import digitalio
import microcontroller
import alarm

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT
INTERVAL = 10                     # период измерений, секунды


def measure():
    """Среднее по 8 отсчётам: так шум меньше."""
    return sum(microcontroller.cpu.temperature for _ in range(8)) / 8

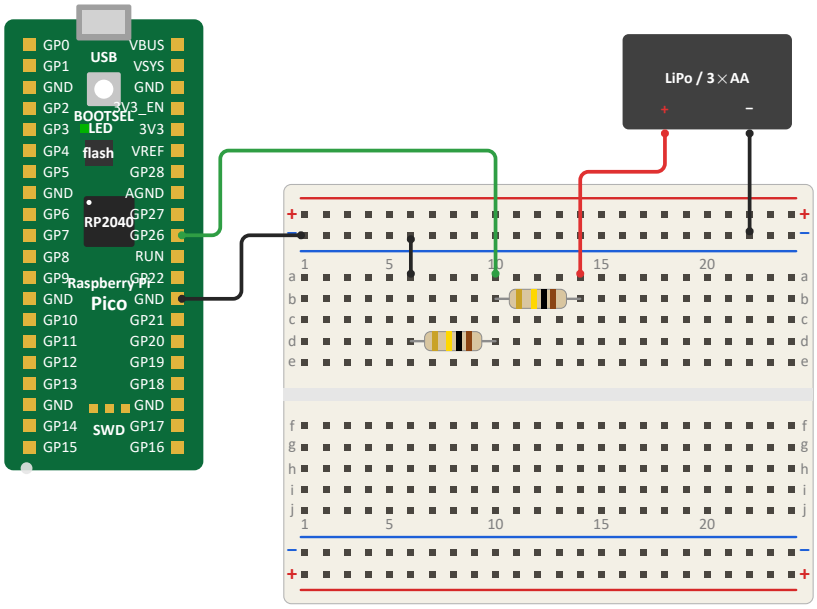

while True:
    t0 = time.monotonic()
    led.value = True              # индикатор активной работы (вспышка)
    t = measure()
    print("T = %.1f °C" % t)
    led.value = False
    # спим остаток периода (лёгкий сон: переменные сохраняются)
    left = max(0.1, INTERVAL - (time.monotonic() - t0))
    wake = alarm.time.TimeAlarm(monotonic_time=time.monotonic() + left)
    alarm.light_sleep_until_alarms(wake)
```

15.9 Измеряем заряд батареи

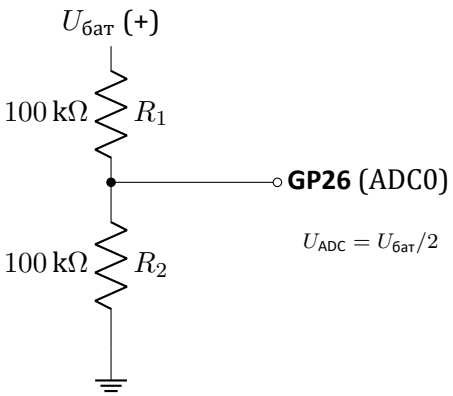
Узнать, когда пора менять батарейки, можно, измерив её напряжение. Но у Pico АЦП измеряет только 0 – $3{,}3$ V. Поэтому напряжение батареи (4,2 В у LiPo) приходится сначала *уменьшить* с помощью **делителя напряжения** из двух резисторов:

$$U_{\text{вых}} = U_{\text{вх}} \frac{R_2}{R_1 + R_2}.$$

При $R_1 = R_2$ напряжение уменьшается вдвое: $4{,}2$ V превращается в $2{,}1$ V, что безопасно для АЦП. Резисторы берём большие (100 kΩ), чтобы делитель не расходовал энергию: ток через него $I = U/(R_1 + R_2) = 4{,}2/200000 \approx 21$ μA.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 15.10: Делитель напряжения для измерения батареи на выводе GP26 (ADC0)

Общая земля батареи, делителя и платы — обязательна: без неё АЦП измерит неизвестно что. Следите, чтобы напряжение на ножке никогда не превышало $3{,}3\,\text{V}$: если у вас батарея до $5{,}5\,\text{V}$ (или 4 АА), поставьте делитель с большим коэффициентом (например, $100\,\text{k}\Omega$ и $47\,\text{k}\Omega$).

Листинг 15.11: MicroPython: измерение напряжения батареи и оценка заряда

```
from machine import ADC, Pin
import time

adc = ADC(Pin(26))          # GP26 = ADC0; делитель 100 кОм + 100 кОм
VREF = 3.3                  # опорное напряжение АЦП (питание 3V3)
K = 2.0                     # коэффициент делителя: (R1 + R2) / R2


def vbat():
    """Напряжение батареи в вольтах, усреднение по 32 отсчётам."""
    s = 0
    for _ in range(32):
        s += adc.read_u16()
    return s / 32 / 65535 * VREF * K

```

```

def percent(v, vmin=3.3, vmax=4.2):
    """Грубый заряд в процентах (линейно; настоящая кривая нелинейна)."""
    p = (v - vmin) / (vmax - vmin) * 100
    return max(0, min(100, p))


while True:
    v = vbat()
    print("U = %.2f В, заряд ~ %d %%" % (v, percent(v)))
    if v < 3.3:
        print("Батарея разряжена! Пора спасать данные и выключаться.")
    time.sleep(2)
```

Листинг 15.12: CircuitPython: то же

```
import time
import board
import analogio

adc = analogio.AnalogIn(board.GP26)   # GP26 = ADC0; делитель 100 кОм + 100 кОм
K = 2.0                               # коэффициент делителя: (R1 + R2) / R2


def vbat():
    """Напряжение батареи в вольтах, усреднение по 32 отсчётам."""
    s = 0
    for _ in range(32):
        s += adc.value
    return s / 32 / 65535 * adc.reference_voltage * K


def percent(v, vmin=3.3, vmax=4.2):
    """Грубый заряд в процентах (линейно; настоящая кривая нелинейна)."""
    p = (v - vmin) / (vmax - vmin) * 100
    return max(0, min(100, p))


while True:
    v = vbat()
    print("U = %.2f В, заряд ~ %d %%" % (v, percent(v)))
    if v < 3.3:
        print("Батарея разряжена! Пора спасать данные и выключаться.")
    time.sleep(2)
```

А что с «встроенным» измерением VSYS?

На обычной Pico вывод **GP29** через внутренний делитель 1:3 подключён к VSYS: `ADC(29).read_u16() * 3 * 3.3 / 65535` даёт напряжение шины. Но у **Pico W и Pico 2 W** вывод **GP29** занят радиочипом (на нём линия тактирования SPI), поэтому читать его неудобно и ненадёжно. Мы пользуемся внешним делителем: он работает на любой плате. Кстати, номер АЦП-канала `ADC(3)` — тот же вывод **GP29**.

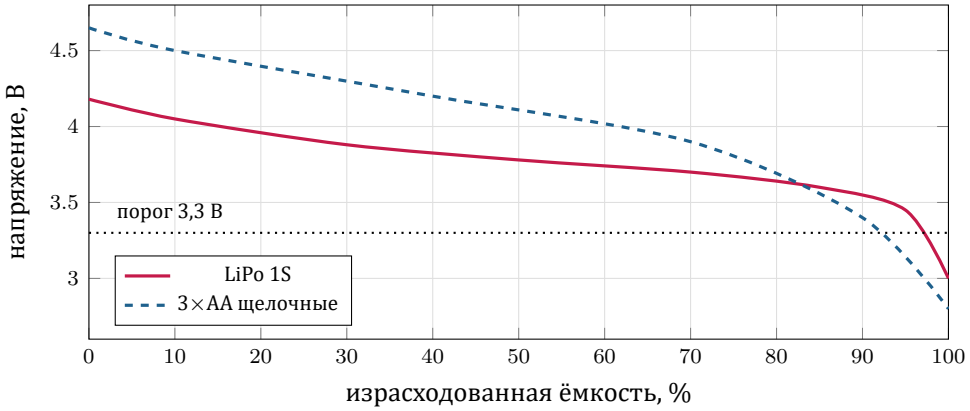


Рис. 15.11: Примерные разрядные кривые: напряжение падает нелинейно, поэтому «проценты» — лишь приблизительная оценка

Видно, что напряжение LiPo почти не меняется на большей части разряда, а потом резко падает. Поэтому линейная формула в программе годится для грубой «батарейки на экране», но не для точного расчёта.

15.10 Учимся программировать: усреднение и фильтр

АЦП «шумит»: два измерения подряд дают чуть разные числа. Самый простой способ уменьшить шум — взять несколько отсчётов и усреднить (мы так и делали в `vbat()`). Есть и экономичный способ: **экспоненциальный фильтр**, которому нужно помнить лишь одно число — прошлый результат:

$$y_{\text{нов}} = \alpha\, x + (1 - \alpha)\, y_{\text{стар}}, \qquad 0 < \alpha \le 1.$$

Чем меньше α, тем сильнее сглаживание, но тем медленнее фильтр реагирует на настоящие изменения.

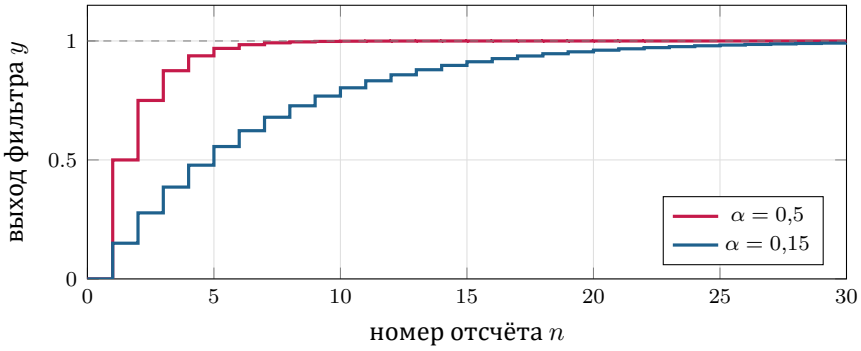


Рис. 15.12: Реакция фильтра на скачок входа: меньший α сглаживает сильнее, но «тормозит»

```
filtered = None
def smooth(x, alpha=0.2):
    global filtered
    filtered = x if filtered is None else alpha * x + (1 - alpha) * filtered
    return filtered
```

Здесь новая идея Python: **глобальная переменная внутри функции**. Слово `global` позволяет функции изменять переменную, объявленную снаружи, и помнить её значение между вызовами.

15.11 Сторожевой таймер

Любая программа может зависнуть: ошибка в коде, сбой датчика, ожидание ответа, которого не будет. Устройство в огороде некому перезагрузить. Поэтому в микроконтроллере есть **сторожевой таймер** (watchdog) — обратный счётчик, который нужно регулярно «кормить». Если программа живёт, она вовремя кричит: «Я работаю!» (`feed()`), и счётчик начинается заново. Если программа зависла, никто счётчик не сбрасывает, он доходит до нуля и перезапускает плату.

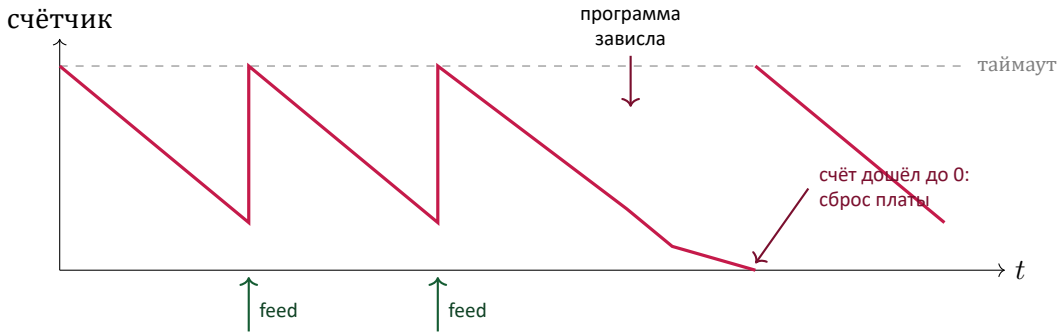


Рис. 15.13: Сторожевой таймер: «кормление» сбрасывает счёт, а зависание приводит к перезапуску

В **MicroPython** это класс `machine.WDT(timeout=мс)`, а метод `feed()` сбрасывает счёт. В **CircuitPython** — объект `microcontroller.watchdog` с полями `timeout` (в секундах) и `mode`. У RP2040 предельное время около $8{,}3\,\text{s}$ ($8388\,\text{ms}$), дольше не получится. Остановить уже запущенный `WDT` в **MicroPython** нельзя.

Листинг 15.13: **MicroPython**: восемь «правильных» шагов, затем имитация зависания и перезапуск

```
import machine
from machine import WDT, Pin
import time

led = Pin("LED", Pin.OUT)

# Если прошлый запуск закончился сбросом по сторожевому таймеру --- моргнём быстро
if machine.reset_cause() == machine.WDT_RESET:
    print("Был сброс по сторожевому таймеру!")
    for _ in range(10):
        led.toggle()
        time.sleep_ms(80)
else:
    print("Обычный запуск")

wdt = WDT(timeout=3000)      # 3 секунды; максимум на RP2040 около 8 с
n = 0
while True:
    n += 1
    print("шаг", n)
    if n == 8:
        print("имитируем зависание...")
        while True:          # бесконечный цикл без wdt.feed()
            pass
    wdt.feed()               # «я жив» --- таймер отсчитывает заново
    time.sleep(0.5)
```

Листинг 15.14: CircuitPython: то же

```
import time
import board
import digitalio
import microcontroller
from watchdog import WatchDogMode

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

# Почему плата перезапустилась в прошлый раз?
reason = microcontroller.cpu.reset_reason
print("причина сброса:", reason)
if reason == microcontroller.ResetReason.WATCHDOG:
    print("Был сброс по сторожевому таймеру!")
    for _ in range(10):
        led.value = not led.value
        time.sleep(0.08)

wdt = microcontroller.watchdog
wdt.timeout = 3                         # секунды; максимум на RP2040 около 8 с
wdt.mode = WatchDogMode.RESET           # при срабатывании --- перезагрузка

n = 0
while True:
    n += 1
    print("шаг", n)
    if n == 8:
        print("имитируем зависание...")
        while True:                     # бесконечный цикл без wdt.feed()
            pass
    wdt.feed()                          # «я жив»
    time.sleep(0.5)
```

Осторожно

Запущенный сторожевой таймер нельзя остановить (в MicroPython) или он будет срабатывать, пока вы правите программу. Не включайте его, пока программа не отлажена. Если плата превратилась в «кирпич» из бесконечных перезагрузок, загрузите её в режиме BOOTSEL (урок 1) и сотрите ошибочную программу.

Где «кормить» сторожа

Главное правило: `feed()` должен стоять в *главном* цикле программы, а не в отдельном потоке или прерывании: тогда его «кормит» только живая программа. Если «кормление» вынести в таймер, оно продолжится даже при зависании основного кода, и смысл защиты теряется.

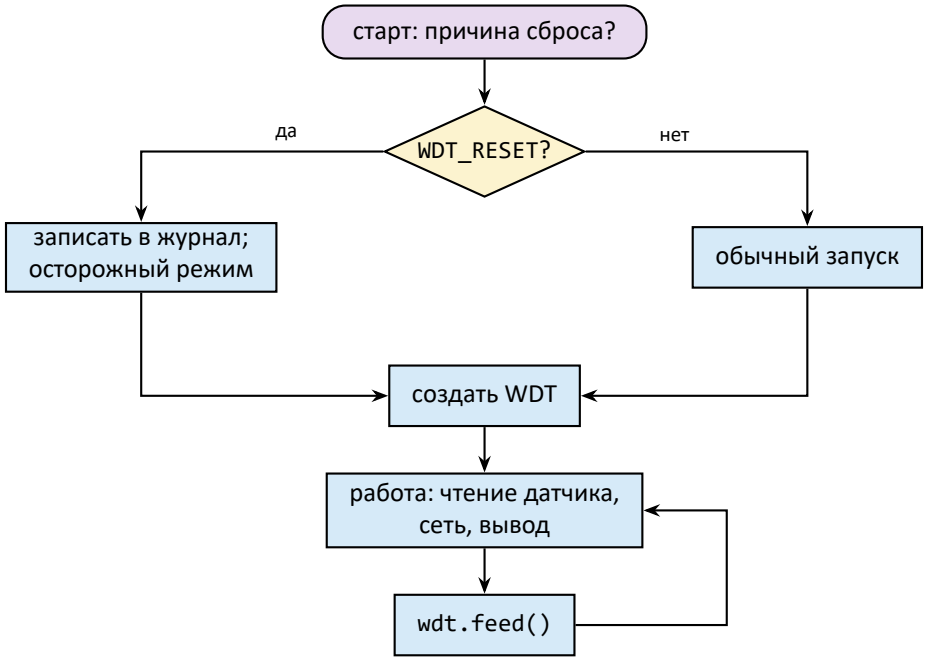


Рис. 15.14: Каркас надёжной программы: анализ причины сброса, затем рабочий цикл с «кормлением»

15.12 Причины сброса

После запуска полезно выяснить, *почему* плата перезапустилась: включили питание, сработал watchdog, нажали RESET или программа вернулась из глубокого сна. От причины зависит поведение: после обычного включения нужно выставить время, после сбоя — записать событие в журнал.

Причина	MicroPython (`machine.reset_cause()`)	CircuitPython (`microcontroller.cpu.reset_reason`)
Включение питания	`machine.PWRON_RESET`	`ResetReason.POWER_ON`
Сторожевой таймер	`machine.WDT_RESET`	`ResetReason.WATCHDOG`
Программный сброс	`machine.SOFT_RESET` (если определена)	`ResetReason.SOFTWARE`
Вывод RUN	— (как включение питания)	`ResetReason.RESET_PIN`
Пробуждение из глубокого сна	зависит от версии	`ResetReason.DEEP_SLEEP_ALARM`
Просадка питания	—	`ResetReason.BROWNOUT`

Таблица 15.4: Причины сброса в двух языках (набор констант зависит от платы и версии прошивки)

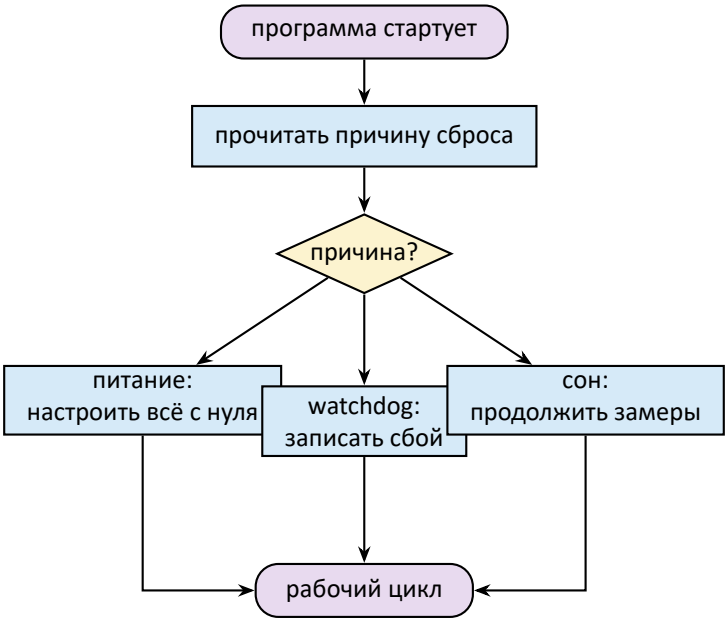


Рис. 15.15: Выбор поведения по причине сброса

15.13 Практика: автономный датчик

Соберём всё вместе: плата с питанием от батарей спит между измерениями, измеряет температуру, следит за напряжением батареи и «лечится» watchdog'ом после сбоя. Порядок работы:

1. Соберите схему питания (рис. 15.3) и делитель (рис. 15.10).
2. Запустите примеры по отдельности: сон, измерение батареи, watchdog.
3. Объедините: цикл «проснулся → измерил температуру и напряжение → вывел → заснул».
4. Измерьте ток мультиметром до и после включения сна и сравните с расчётом.

Если не работает

- **Плата не включается от батарей** — проверьте полярность и диод: катод (полоса) должен смотреть на VSYS. Измерьте напряжение на VSYS: должно быть не меньше $1{,}8\,\text{V}$.
- **После `lightsleep()` пропала связь с Thonny** — это обычное поведение; нажмите «Стоп» или перезапустите плату, а при отладке используйте короткие интервалы и светодиод.
- **После глубокого сна программа начинается заново** — так и должно быть: храните состояние в `alarm.sleep_memory` (в CircuitPython) или во флеш-файле.
- **Сторожевой таймер перезапускает плату всё время** — интервал между `feed()` длиннее таймаута (например, долгий `sleep` или медленный запрос по сети). Увеличьте таймаут или «кормите» чаще.
- **Вольтметр на батарее показывает одно, а программа — другое** — проверьте коэффициент делителя и опорное напряжение; на ножке АЦП должно быть *меньше* $3{,}3\,\text{V}$.
- **Плата нагревается или пахнет** — немедленно отключите питание: вероятно, короткое замыкание или неверная полярность.

Итоги урока

- Pico питается от USB (VBUS) или от VSYS ($1{,}8$ – $5{,}5$ V); регулятор делает из этого 3,3 В; источники объединяют диодом.
- Время работы $T = Q/I$; сон снижает средний ток в десятки раз, понижение частоты — примерно вдвое.
- `lightsleep` сохраняет память, `deepsleep` — нет (программа стартует с начала); в **CircuitPython** будильники: `alarm.time`, `alarm.pin`.
- Напряжение батареи измеряют АЦП через делитель; на ножке не должно быть больше 3,3 В.
- Сторожевой таймер (WDT, `microcontroller.watchdog`) перезапускает зависшую программу; причину сброса можно узнать при старте.

Проверь себя

1. Чем отличаются VBUS, VSYS и 3V3? Какой вывод можно использовать для питания от батарей?
2. Зачем нужен диод между батареями и VSYS, когда плата подключена к USB?
3. Как рассчитать время работы от батареи и почему сон так сильно его увеличивает?
4. Чем лёгкий сон отличается от глубокого? Что делать, чтобы программа «помнила» состояние после глубокого сна?
5. Зачем «кормить» сторожевой таймер и почему `feed()` нельзя выносить в таймер?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Запустите пример с изменением частоты, запишите время работы на трёх частотах и постройте график «частота — время».
2. ★ Рассчитайте время работы от батареи 1000 mAh для платы, которая 1 с работает с током 20 mA и 59 с спит с током $1{,}5$ mA.
3. ★★ Измерьте мультиметром ток платы в обычном режиме и в сне; сравните с расчётом и объясните расхождения.
4. ★★ Добавьте в пример измерения батареи экспоненциальный фильтр и сравните «шум» показаний со сглаживанием и без него.
5. ★★★ Сделайте «датчик на батарейке»: раз в минуту измеряет температуру, записывает результат в файл и засыпает; при напряжении ниже порога записывает предупреждение.
6. ★★★★ Добавьте watchdog и журнал сбоев в файл: после перезапуска программа должна определить причину и дописать строку «сбой / включение / сон», а кнопка на GP14 должна будить плату и принудительно записывать замер.

Урок 16. Wi-Fi: Pico W и Pico 2 W

Цель	подключить Pico W к домашней сети, понять, как устроены адреса и запросы, и получать из интернета данные: погоду и точное время
Опирается на	урок 2 (светодиод), урок 4 (время и таймеры), урок 15 (питание и надёжность)
Результат	плата сама подключается к Wi-Fi и восстанавливает связь после обрыва; сканирует эфир; делает HTTP-запросы с разбором JSON; узнаёт время по NTP
Понадобится	Pico W или Pico 2 W (**именно с буквой W**), домашняя сеть Wi-Fi на $2{,}4$ GHz, внешний светодиод и резистор $220\,\Omega$

16.1 Идея урока коротко

Раньше плата общалась только с вами через USB. С Wi-Fi у неё появляется целый мир: она может узнать погоду, время, отправить данные на сервер, получить команду с телефона. Эти возможности нужны любому «умному» устройству. Наша задача — научиться делать это *надёжно*: сеть не бывает идеальной, роутер перезагружается, пароль вводится неверно, ответ сервера не приходит. Программа, которая не умеет справляться с такими сбоями, работает ровно до первого неудачного дня.

Осторожно

Всё в этом уроке работает **только на платах с буквой W**: Pico W и Pico 2 W (а также Pico 2 WH и Pico WH: это те же платы с впаянными штырьками). У обычных Pico и Pico 2 радиочипа нет, и `network.WLAN` либо не существует, либо ничего не делает.

16.2 Радиочип CYW43439

Ни RP2040, ни RP2350 сами не умеют работать с радиоэфиром: на плате W установлен отдельный чип **CYW43439** (Infineon/Cypress). Он поддерживает Wi-Fi 4 (802.11n) на частоте $2{,}4$ GHz и Bluetooth 5.2. Радиочип подключён к микроконтроллеру по особой последовательной шине, похожей на SPI. Эту шину программно создаёт блок **PIO** (урок 14), а не аппаратный SPI. Поэтому на плате заняты четыре вывода, которые вы не можете использовать как обычные GPIO.

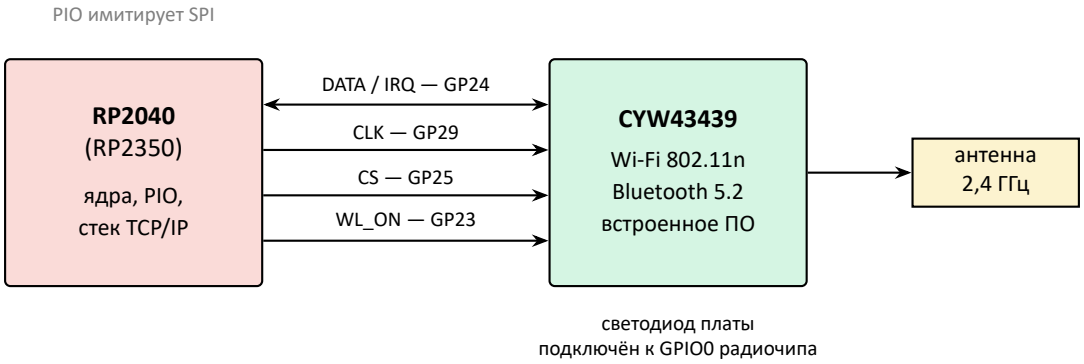


Рис. 16.1: Архитектура Pico W: микроконтроллер управляет радиочипом по шине на основе PIO

Из этого следует несколько практических вещей:

- `Pin("LED")` (а не `Pin(25)`) управляет светодиодом через радиочип, поэтому он работает только после инициализации радиочипа и чуть медленнее обычного вывода (урок 2).
- Выводы **GP23**, **GP24**, **GP25**, **GP29** на плате W заняты радиочипом и не выведены на штырьки: своими деталями их не занять (поэтому, например, встроенное измерение VSYS через **GP29** на плате W неудобно; см. урок efles:power).
- Прошивка радиочипа (*firmware blob*) хранится во флеш-памяти платы и загружается при включении Wi-Fi.
- Только 2,4 ГГц: 5-гигагерцовые сети платой не видны. Если роутер раздаёт две сети с одним именем, проверьте, что включён диапазон 2,4 ГГц.

Диапазон 2,4 ГГц и каналы

Весь диапазон 2,4 ГГц разделён на каналы шириной по 5 MHz (в Европе — 13 каналов), но сигнал одной сети занимает около 20 MHz. Поэтому соседние каналы перекрываются, и одновременно *не мешают друг другу* только каналы 1, 6 и 11. Если вокруг много сетей на одном канале, скорость падает — как в комнате, где все говорят хором.

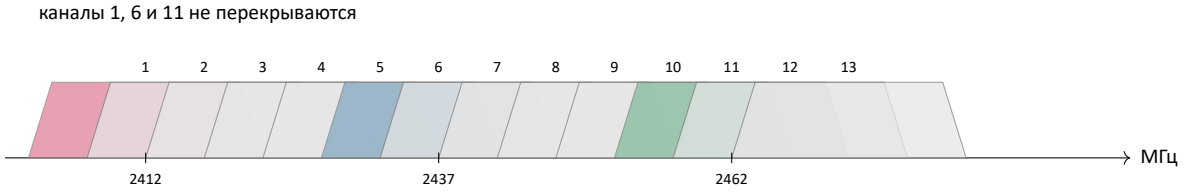


Рис. 16.2: Каналы Wi-Fi в диапазоне 2,4 ГГц: ширина сети — около 20 МГц, шаг каналов — 5 МГц

16.3 Два режима: станция и точка доступа

Радиочип умеет работать в двух основных ролях. Роль выбирают, создавая объект `network.WLAN` с константой `STA_IF` или `AP_IF` (в **CircuitPython** — функции `wifi.radio.connect()` и `wifi.radio.start_ap()`).

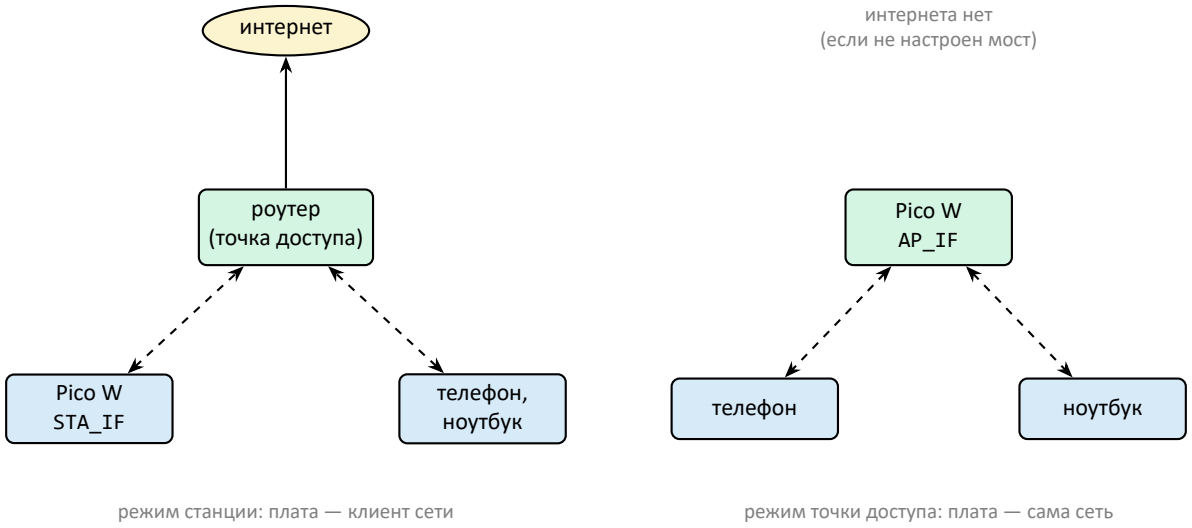


Рис. 16.3: Два режима работы Wi-Fi: клиент и собственная сеть

В режиме **станции** (STA) плата подключается к существующей сети (как телефон или ноутбук). Это основной режим для устройств интернета вещей: роутер дома есть, интернет у него есть. В режиме **точки доступа** (AP) плата сама создаёт сеть: удобно для первой настройки («введите

пароль вашей сети») и для работы без роутера (урок 17). В режиме AP одновременно подключаются лишь несколько клиентов, а интернет-доступа нет.

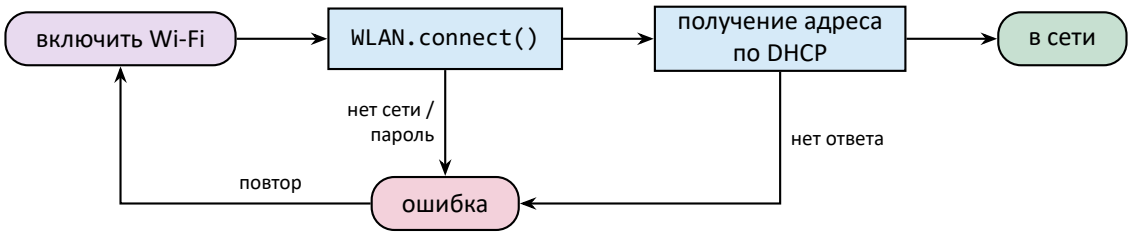


Рис. 16.4: Этапы подключения к сети: на каждом шаге возможен отказ

16.4 Первое подключение

Для подключения нужны три вещи: название сети (SSID), пароль и код страны. Код страны важен: он определяет разрешённые каналы и мощность передатчика. Для России — `"RU"`. В **MicroPython** его задают вызовом `rp2.country("RU")` *до* включения Wi-Fi. В **CircuitPython** для домашних сетей отдельная настройка страны обычно не нужна: радиочип работает со стандартным набором каналов. Если сеть на нестандартном канале (12–13) не видна, смотрите документацию своей версии.

Где хранить пароль

Пароль от Wi-Fi нельзя писать прямо в программе: если выложить её на GitHub или показать друзьям на экране, пароль станет известен всем. Секреты кладут в отдельный файл: в **MicroPython** — `secrets.py`, в **CircuitPython** — `settings.toml`. Основная программа читает значения из него.

Листинг 16.1: MicroPython: файл `secrets.py` на плате

```
# secrets.py --- лежит на плате рядом с main.py и НЕ попадает в Git!
# (добавьте строку secrets.py в файл .gitignore)
SSID = "MyHomeWiFi"
PASSWORD = "my-secret-password"
COUNTRY = "RU"          # код страны для радиочипа
```

Листинг 16.2: CircuitPython: файл `settings.toml` на диске CIRCUITPY

```
# settings.toml --- на диске CIRCUITPY рядом с code.py
# Этот файл тоже нельзя публиковать в Git!
CIRCUITPY_WIFI_SSID = "MyHomeWiFi"
CIRCUITPY_WIFI_PASSWORD = "my-secret-password"
# свои настройки программы можно добавлять сюда же:
TIMEZONE_HOURS = 3
```

Осторожно

Файлы `secrets.py` и `settings.toml` **не должны попадать в Git**. Добавьте их имена в файл `.gitignore` в корне проекта. Если пароль всё же «утёк», считайте его скомпрометированным: смените его на роутере. Для общих устройств заведите на роутере *гостевую* сеть.

В **CircuitPython** есть приятная особенность: если в `settings.toml` заданы `CIRCUITPY_WIFI_SSID` и `CIRCUITPY_WIFI_PASSWORD`, плата подключается к сети *сама при старте*, ещё до выполнения `code.py`. Явный вызов `connect()` нужен, чтобы видеть ошибки и повторять попытки.

Листинг 16.3: MicroPython: подключение с проверкой состояния и вывод адресов

```
import network
import rp2
import time
import secrets                      # файл secrets.py с SSID и PASSWORD

rp2.country(secrets.COUNTRY)        # правила радиочастот страны (до включения Wi-Fi)

wlan = network.WLAN(network.STA_IF)   # режим станции (клиент точки доступа)
wlan.active(True)                     # включаем радиомодуль
wlan.connect(secrets.SSID, secrets.PASSWORD)

t0 = time.ticks_ms()
while True:
    st = wlan.status()
    print("статус:", st)
    if st == network.STAT_GOT_IP:     # 3: адрес получен --- мы в сети
        break
    if st < 0:                        # отрицательные коды --- ошибки
        raise RuntimeError("не удалось подключиться, код %d" % st)
    if time.ticks_diff(time.ticks_ms(), t0) > 15_000:
        raise RuntimeError("таймаут подключения")
    time.sleep(1)

ip, mask, gw, dns = wlan.ifconfig()
print("IP-адрес:", ip)
print("маска:   ", mask)
print("шлюз:    ", gw)
print("DNS:     ", dns)
print("уровень сигнала:", wlan.status("rssi"), "дБм")
```

Листинг 16.4: CircuitPython: то же

```
import os
import time
import wifi

# Если в settings.toml есть CIRCUITPY_WIFI_SSID, плата подключается сама при старте.
# Здесь подключаемся явно, чтобы видеть ошибки.
ssid = os.getenv("CIRCUITPY_WIFI_SSID")
password = os.getenv("CIRCUITPY_WIFI_PASSWORD")

print("подключаюсь к", ssid)
try:
    wifi.radio.connect(ssid, password, timeout=15)   # блокирует до результата
except ConnectionError as e:
    raise RuntimeError("не удалось подключиться: %s" % e)

print("IP-адрес:", wifi.radio.ipv4_address)
print("маска:   ", wifi.radio.ipv4_subnet)
print("шлюз:    ", wifi.radio.ipv4_gateway)
print("DNS:     ", wifi.radio.ipv4_dns)
print("MAC:     ", [hex(b) for b in wifi.radio.mac_address])
print("уровень сигнала:", wifi.radio.ap_info.rssi, "дБм")
```

Разбираем статус

Метод `wlan.status()` в **MicroPython** возвращает число — этап подключения. Положительные значения означают процесс, отрицательные — ошибку.

Код	Константа	Что произошло
0	`STAT_IDLE`	Подключения нет, ничего не происходит.
1	`STAT_CONNECTING`	Идёт подключение к точке доступа.
2	(внутренний)	Подключились к точке доступа, ждём IP-адрес.
3	`STAT_GOT_IP`	Всё в порядке: адрес получен.
−1	`STAT_CONNECT_FAIL`	Общий сбой подключения.
−2	`STAT_NO_AP_FOUND`	Сеть с таким именем не найдена.
−3	`STAT_WRONG_PASSWORD`	Неверный пароль (или не тот тип защиты).

Таблица 16.1: Коды `wlan.status()` для радиочипа CYW43 в **MicroPython**

В **CircuitPython** подход другой: `wifi.radio.connect()` сам ждёт результата и при ошибке *выбрасывает исключение* `ConnectionError` с поясняющим текстом (например, «No network with that ssid» или «Authentication failure»). Поэтому подключение оборачивают в `try/except`.

16.5 Надёжная функция подключения

Домашний роутер иногда перезагружается, а плата рядом с микроволновкой теряет сигнал. Устройство, которое ничего не делает после первого обрыва, бесполезно. Нужна функция, которая: (1) пытается подключиться несколько раз; (2) делает паузы между попытками, и паузы растут (чтобы не «бомбить» роутер запросами); (3) не повторяет бессмысленные попытки (при неверном пароле); (4) вызывается заново, когда связь пропала.

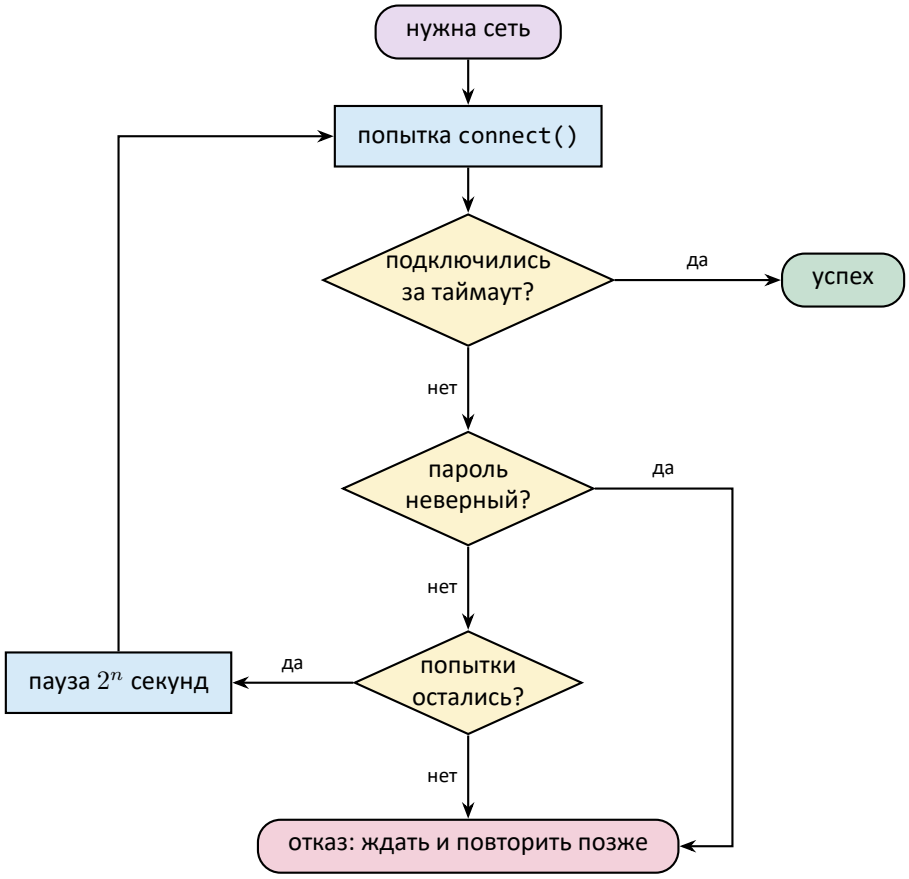
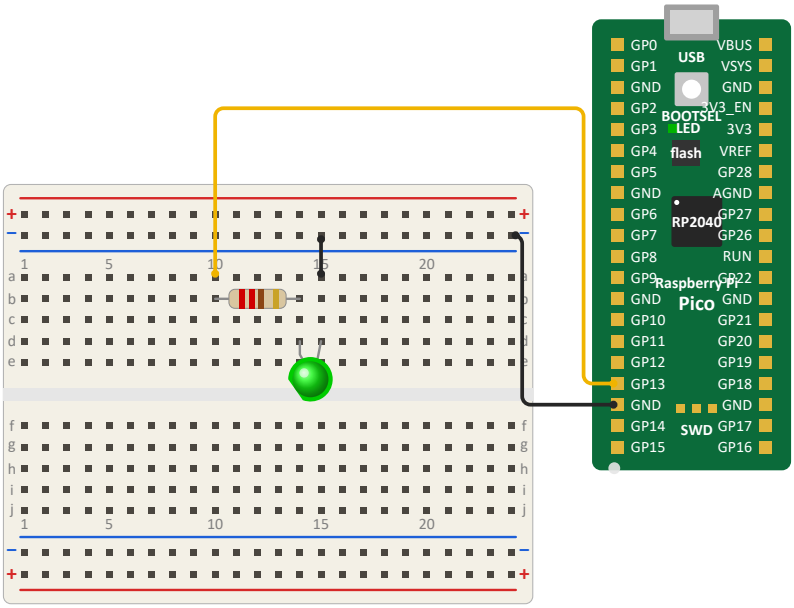
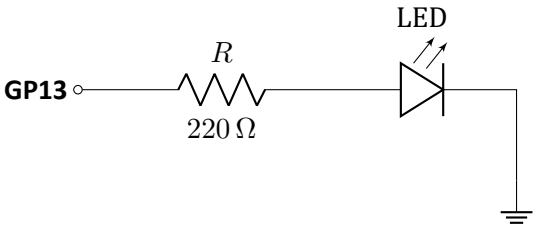


Рис. 16.5: Блок-схема надёжного подключения: ограниченное число попыток и растущие паузы

Для наглядности добавим внешний светодиод: он мигает, пока идёт подключение, и горит постоянно, когда сеть есть. Если связь пропала — гаснет. Это самый простой способ «увидеть» состояние сети без компьютера.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 16.6: Индикатор состояния сети: светодиод на GP13 через резистор

Листинг 16.5: MicroPython: подключение с повторами, паузами и переподключением

```
import network
import rp2
import time
from machine import Pin
import secrets

rp2.country(secrets.COUNTRY)
wlan = network.WLAN(network.STA_IF)
led = Pin(13, Pin.OUT)             # внешний светодиод: статус сети


def connect(timeout_s=20, attempts=5):
    """Подключается к Wi-Fi с повторами. Возвращает True при успехе."""
    wlan.active(True)
    for attempt in range(1, attempts + 1):
        print("попытка", attempt, "из", attempts)
        wlan.disconnect()                      # сбрасываем прошлое состояние
        wlan.connect(secrets.SSID, secrets.PASSWORD)
        t0 = time.ticks_ms()
        while time.ticks_diff(time.ticks_ms(), t0) < timeout_s * 1000:
            st = wlan.status()
            if st == network.STAT_GOT_IP:
                led.on()                       # горит постоянно: сеть есть
                return True
```

```
            if st == network.STAT_WRONG_PASSWORD:
                print("неверный пароль --- повторять бессмысленно")
                return False
            led.toggle()                          # мигаем: идёт подключение
            time.sleep_ms(250)
        time.sleep(2 ** attempt)                  # пауза растёт: 2, 4, 8, 16... с
    led.off()
    return False


while True:
    if not wlan.isconnected():                    # сеть пропала --- восстанавливаем
        led.off()
        print("связи нет, подключаюсь...")
        if not connect():
            print("не вышло, ждём минуту")
            time.sleep(60)
            continue
        print("в сети:", wlan.ifconfig()[0])
    # ... здесь полезная работа устройства ...
    time.sleep(5)
```

Листинг 16.6: CircuitPython: то же

```
import os
import time
import board
import digitalio
import wifi

led = digitalio.DigitalInOut(board.GP13)       # внешний светодиод: статус сети
led.direction = digitalio.Direction.OUTPUT


def connect(attempts=5):
    """Подключается к Wi-Fi с повторами. Возвращает True при успехе."""
    for attempt in range(1, attempts + 1):
        print("попытка", attempt, "из", attempts)
        led.value = not led.value                 # мигаем при каждой попытке
        try:
            wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                               os.getenv("CIRCUITPY_WIFI_PASSWORD"),
                               timeout=15)
            led.value = True                      # горит постоянно: сеть есть
            return True
        except ConnectionError as e:
            print("ошибка:", e)
            time.sleep(2 ** attempt)              # пауза растёт: 2, 4, 8, 16... с
    led.value = False
    return False


while True:
    if not wifi.radio.connected:                  # сеть пропала --- восстанавливаем
        led.value = False
        print("связи нет, подключаюсь...")
        if not connect():
            print("не вышло, ждём минуту")
```

```
            time.sleep(60)
            continue
        print("в сети:", wifi.radio.ipv4_address)
    # ... здесь полезная работа устройства ...
    time.sleep(5)
```

Это интересно

Паузы вида $2, 4, 8, 16, \ldots$ секунд называются **экспоненциальной задержкой** (exponential backoff). Этим приёмом пользуются все большие системы: если тысяча устройств одновременно потеряла сеть, они не должны одновременно «накинуться» на сервер сразу после восстановления.

16.6 Сила сигнала и сканирование

Уровень сигнала измеряют в **дБм** (dBm) — логарифмических единицах относительно 1 mW. Значения отрицательные: чем ближе к нулю, тем сигнал сильнее. Каждые -3 дБ — это *вдвое* меньше мощности, поэтому цифры нелинейны.

RSSI, дБм	Качество связи
> -50	Отлично, рядом с роутером
$-50 \ldots -67$	Хорошо: подходит для видео и любых задач
$-67 \ldots -80$	Удовлетворительно: страницы грузятся, возможны разрывы
$-80 \ldots -90$	Слабо: частые потери
< -90	Соединение практически невозможно

Таблица 16.2: Как читать значения RSSI

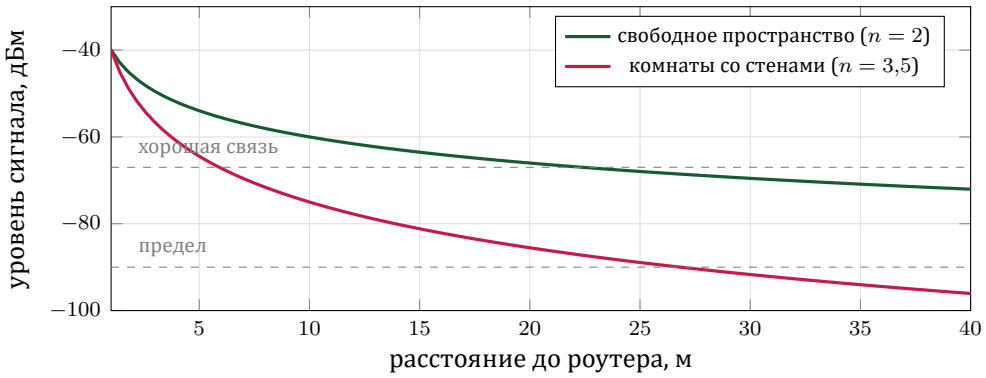


Рис. 16.7: Модель затухания: $\mathrm{RSSI}(d) = -40 - 10\,n\,\lg d$. Стены и перекрытия резко сокращают дальность

Узнать RSSI можно после подключения (`wlan.status("rssi")` в **MicroPython**, `wifi.radio.ap_info.rssi` в **CircuitPython**) или просканировав эфир: тогда вы увидите все сети вокруг, их каналы и уровни. Это помогает выбрать свободный канал для роутера.

Листинг 16.7: MicroPython: список сетей вокруг с уровнем и каналом

```
import network

wlan = network.WLAN(network.STA_IF)
wlan.active(True)

# scan() возвращает список кортежей:
# (ssid, bssid, канал, RSSI, тип защиты, скрытая)
SECURITY = {0: "открытая", 1: "WEP", 2: "WPA", 3: "WPA2", 4: "WPA/WPA2"}

nets = wlan.scan()
nets.sort(key=lambda n: n[3], reverse=True)    # сначала самые сильные

print("найдено сетей:", len(nets))
for ssid, bssid, channel, rssi, security, hidden in nets:
    name = ssid.decode() if ssid else "<скрытая>"
    bars = "#" * max(0, (rssi + 100) // 5)     # простая «шкала» сигнала
    print("%-24s канал %2d  %4d дБм  %-9s %s"
          % (name, channel, rssi, SECURITY.get(security, "?"), bars))
```

Листинг 16.8: CircuitPython: то же

```
import wifi

found = []
# Сканирование идёт «по одной сети»: это генератор
for net in wifi.radio.start_scanning_networks():
    found.append(net)
wifi.radio.stop_scanning_networks()            # обязательно останавливаем

found.sort(key=lambda n: n.rssi, reverse=True)  # сначала самые сильные

print("найдено сетей:", len(found))
for net in found:
    name = net.ssid if net.ssid else "<скрытая>"
    bars = "#" * max(0, (net.rssi + 100) // 5)  # простая «шкала» сигнала
    print("%-24s канал %2d  %4d дБм  %s" % (name, net.channel, net.rssi, bars))
```

В MicroPython `scan()` возвращает список кортежей. Обратите внимание: SSID приходит как *байты* (`b'MyNet'`), поэтому его декодируют методом `decode()`. В CircuitPython каждая сеть — объект с полями `ssid`, `rssi`, `channel`, `authmode`.

16.7 Адреса в сети

Чтобы данные нашли нужное устройство, у него должен быть адрес. В локальной сети это **IP-адрес** из четырёх чисел от 0 до 255, например `192.168.1.37`. Вместе с адресом плата получает ещё три значения:

- **маска подсети** (например, `255.255.255.0`) — показывает, какая часть адреса означает «номер сети», а какая — «номер устройства»;
- **шлюз** (gateway) — адрес роутера: туда плата отправляет всё, что не из её сети;
- **DNS-сервер** — «телефонная книга», переводящая имена сайтов в IP-адреса.

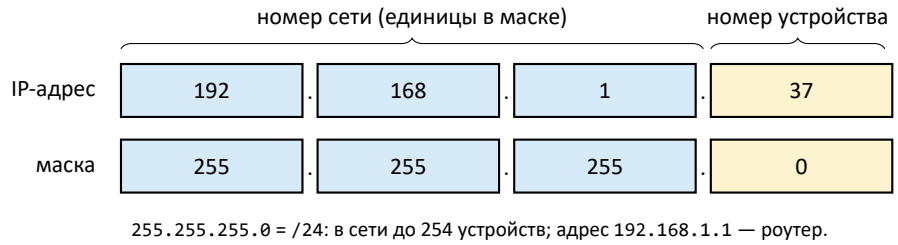


Рис. 16.8: IP-адрес и маска: что относится к сети, а что — к устройству

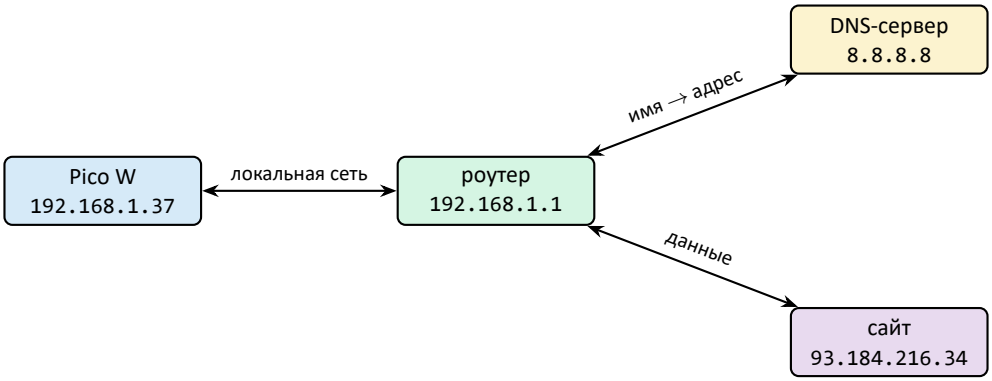


Рис. 16.9: Путь запроса: плата → роутер (шлюз) → DNS и сервер в интернете

Адрес плата получает автоматически по протоколу **DHCP**: она «спрашивает» у роутера адрес, роутер выдаёт свободный на определённое время (аренда). Обмен состоит из четырёх сообщений, их называют DORA:

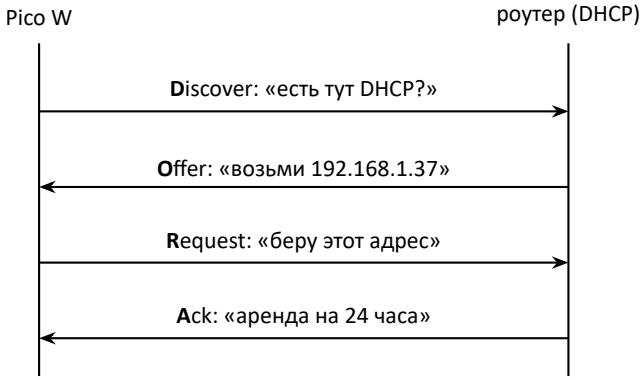


Рис. 16.10: Получение адреса по DHCP: четыре сообщения

Совет

Адрес, выданный по DHCP, может измениться при перезагрузке роутера. Чтобы плата всегда имела один адрес, зарезервируйте его для платы в настройках роутера по MAC-адресу. Можно также задать статический адрес: `wlan.ifconfig(("192.168.1.50", "255.255.255.0", "192.168.1.1", "8.8.8.8"))`, но тогда следите, чтобы он не пересекался с адресами DHCP.

16.8 Стек протоколов: от радиоволн до JSON

Сетевое взаимодействие устроено «слоями»: каждый уровень делает свою работу и опирается на нижележащий. Благодаря этому мы пишем `requests.get(url)`, не думая об антенне.

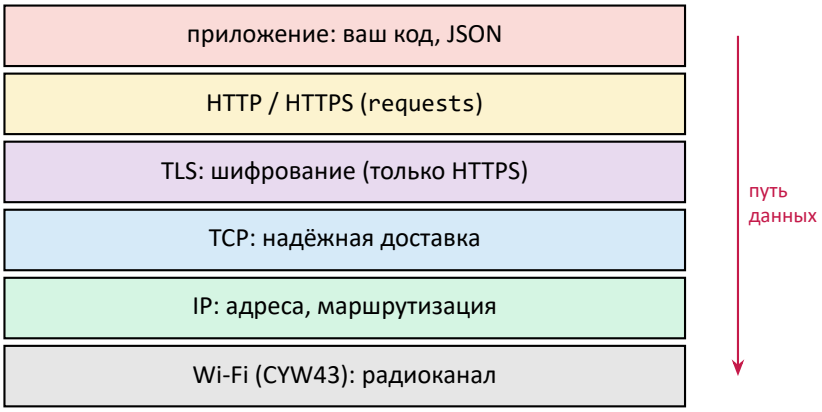


Рис. 16.11: Слои сетевых протоколов: каждый нижний слой обслуживает верхний

16.9 HTTP-клиент

HTTP — язык, на котором общаются браузеры и серверы. Клиент отправляет *запрос*: метод (`GET` — «дай», `POST` — «прими»), путь и заголовки; сервер возвращает *ответ*: код состояния (200 — успех, 404 — не найдено, 500 — ошибка сервера) и тело. Подробно протокол рассмотрен в уроке 17, а здесь мы пользуемся им как клиент.

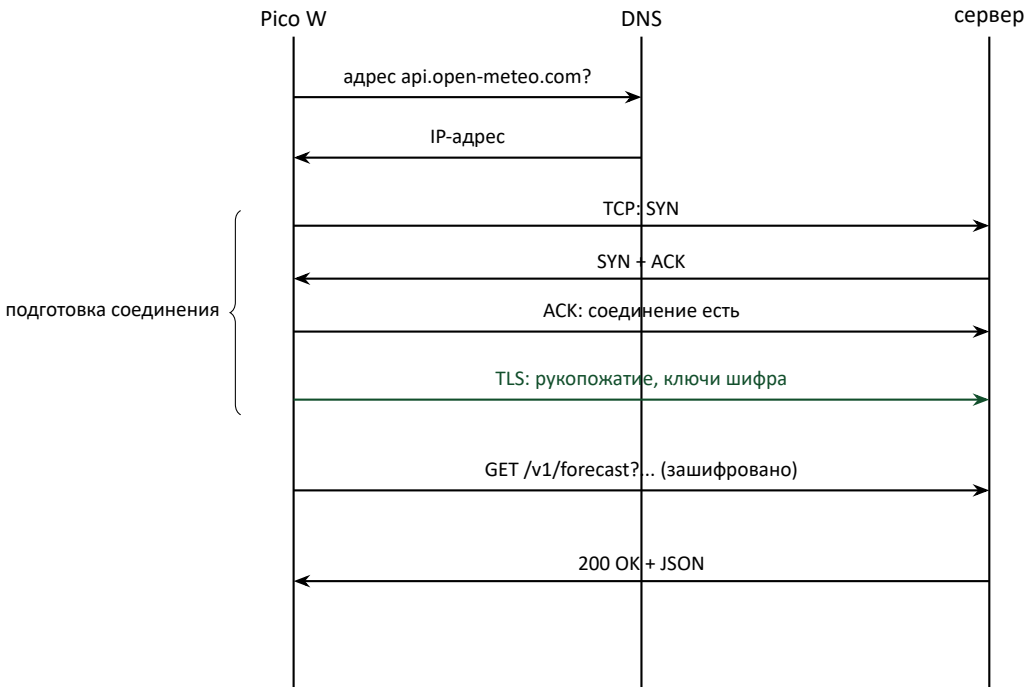


Рис. 16.12: Последовательность одного HTTPS-запроса: DNS, TCP, TLS и сам запрос

Погода без регистрации

Сервис **Open-Meteo** (`open-meteo.com`) отдаёт прогноз по координатам и не требует ключа для некоммерческого использования. Запрос по адресу

```
https://api.open-meteo.com/v1/forecast?latitude=55.75&longitude=37.62
    &current=temperature_2m,wind_speed_10m
```

возвращает данные в формате **JSON**: текст со словарями и списками, очень похожий на словари Python:

```
{"latitude":55.75, "longitude":37.62,
 "current": {"time":"2026-11-05T12:00", "temperature_2m":4.1, "wind_speed_10m":9.7}}
```

Метод `r.json()` превращает такой текст в обычный словарь Python, и дальше мы берём значения по ключам.

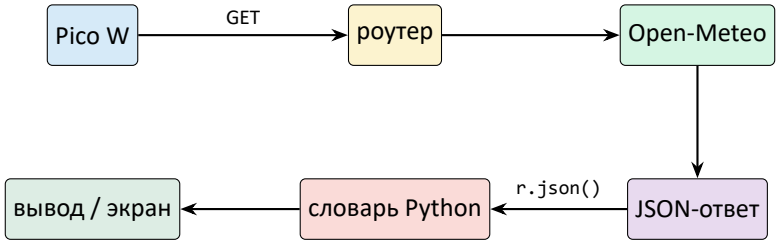


Рис. 16.13: Путь данных от сервера до экрана: запрос, JSON и словарь

Листинг 16.9: MicroPython: запрос погоды раз в 10 минут

```
import network
import time
import secrets

try:
    import requests                 # новые версии: mip.install("requests")
except ImportError:
    import urequests as requests    # старое имя того же модуля

URL = ("https://api.open-meteo.com/v1/forecast"
       "?latitude=55.75&longitude=37.62"                # Москва
       "&current=temperature_2m,wind_speed_10m")

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)

while True:
    try:
        r = requests.get(URL)
        try:
            if r.status_code == 200:
                data = r.json()                         # словарь Python
                cur = data["current"]
                print("Температура: %.1f °C, ветер: %.1f км/ч"
                      % (cur["temperature_2m"], cur["wind_speed_10m"]))
            else:
                print("ошибка сервера:", r.status_code)
        finally:
            r.close()               # ВАЖНО: освободить сокет и память
    except Exception as e:          # нет сети, DNS, обрыв и т. п.
        print("запрос не удался:", e)
    time.sleep(600)                 # погода меняется медленно: раз в 10 минут
```

Листинг 16.10: CircuitPython: то же

```
import os
import ssl
import time
import socketpool
import wifi
import adafruit_requests

URL = ("https://api.open-meteo.com/v1/forecast"
       "?latitude=55.75&longitude=37.62"                # Москва
       "&current=temperature_2m,wind_speed_10m")

if not wifi.radio.connected:         # обычно плата уже подключилась по settings.toml
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
session = adafruit_requests.Session(pool, ssl.create_default_context())

while True:
    try:
        with session.get(URL) as r:              # with сам закроет ответ
            if r.status_code == 200:
                cur = r.json()["current"]
                print("Температура: %.1f °C, ветер: %.1f км/ч"
                      % (cur["temperature_2m"], cur["wind_speed_10m"]))
            else:
                print("ошибка сервера:", r.status_code)
    except Exception as e:                       # нет сети, DNS, обрыв и т. п.
        print("запрос не удался:", e)
    time.sleep(600)                              # погода меняется медленно
```

Разберём главное:

- В MicroPython модуль называется `requests` (в старых прошивках — `urequests`). В актуальных версиях его ставят командой `mip.install("requests")` из Thonny или в REPL; в некоторых прошивках он уже включён. В CircuitPython нужна библиотека `adafruit_requests` (устанавливается через `circup install adafruit_requests` или копированием в папку `lib`).
- **Ответ нужно закрывать** (`r.close()`, в CircuitPython — `with`). Иначе сокет остаётся открытым, и после десятка запросов память кончится или сервер откажет в соединении.
- Запрос — единственное место, где всё может пойти не так, поэтому он целиком в `try/except`. Если нет связи, мы просто печатаем ошибку и пробуем через 10 минут.
- **Не спрашивайте сервер слишком часто.** Погода меняется медленно, а у бесплатных сервисов есть лимиты.

Осторожно

HTTPS и память. Для шифрования (TLS) нужны десятки килобайт оперативной памяти, а у RP2040 всего 264 kB. Если память раздроблена, запрос может завершиться ошибкой `MemoryError` или `OSError: -17040`. Помогает вызов `gc.collect()` перед запросом и небольшие ответы. Кроме того, по умолчанию MicroPython *не проверяет сертификат сервера*: соединение зашифровано, но подлинность сервера не подтверждена. Для учебных опытов это

допустимо, для систем с паролями и платежами — нет. **CircuitPython** проверяет сертификаты по встроенному списку корневых центров.

Отправляем данные: POST и JSON

Чтобы *отправить* данные, используют POST: данные кладут в тело запроса. Самый распространённый формат — снова JSON. Параметр `json=...` сам превращает словарь в текст и добавляет заголовок `Content-Type: application/json`. Для проверки удобен сервис `httpbin.org`: он возвращает то, что получил.

Листинг 16.11: **MicroPython**: отправка словаря в формате JSON

```
import machine
import network
import time
import secrets

try:
    import requests
except ImportError:
    import urequests as requests

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)

sensor = machine.ADC(4)                          # температура кристалла

# httpbin.org/post «отзеркаливает» то, что ему отправили --- удобно для проверки
URL = "https://httpbin.org/post"

v = sensor.read_u16() * 3.3 / 65535
payload = {
    "device": "pico-1",
    "temp_c": round(27 - (v - 0.706) / 0.001721, 1),
    "uptime_s": time.ticks_ms() // 1000,
}

r = requests.post(URL, json=payload)             # словарь превращается в JSON
try:
    print("код ответа:", r.status_code)
    answer = r.json()
    print("сервер получил:", answer["json"])     # httpbin кладёт наши данные в «json»
finally:
    r.close()
```

Листинг 16.12: CircuitPython: то же

```
import os
import ssl
import time
import microcontroller
import socketpool
import wifi
import adafruit_requests

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
session = adafruit_requests.Session(pool, ssl.create_default_context())

# httpbin.org/post «отзеркаливает» то, что ему отправили --- удобно для проверки
URL = "https://httpbin.org/post"

payload = {
    "device": "pico-1",
    "temp_c": round(microcontroller.cpu.temperature, 1),
    "uptime_s": int(time.monotonic()),
}

with session.post(URL, json=payload) as r:       # словарь превращается в JSON
    print("код ответа:", r.status_code)
    answer = r.json()
    print("сервер получил:", answer["json"])     # httpbin кладёт наши данные в «json»
```

Совет

Проверяйте `status_code`: 200–299 — успех; 400–499 — ошибка клиента (неверный адрес, нет прав); 500–599 — ошибка на стороне сервера. Не пытайтесь разбирать JSON, если код не 200.

16.10 Время по сети: NTP

В плате нет часов, которые идут при выключенном питании: после включения время сбрасывается на 1 января 2021 (в CircuitPython — в районе 2000 года). Чтобы узнать точное время, спрашивают у серверов **NTP** (*Network Time Protocol*). Протокол учитывает задержку сети: клиент записывает время отправки запроса T_1 и получения ответа T_4, а сервер сообщает время приёма T_2 и отправки T_3.

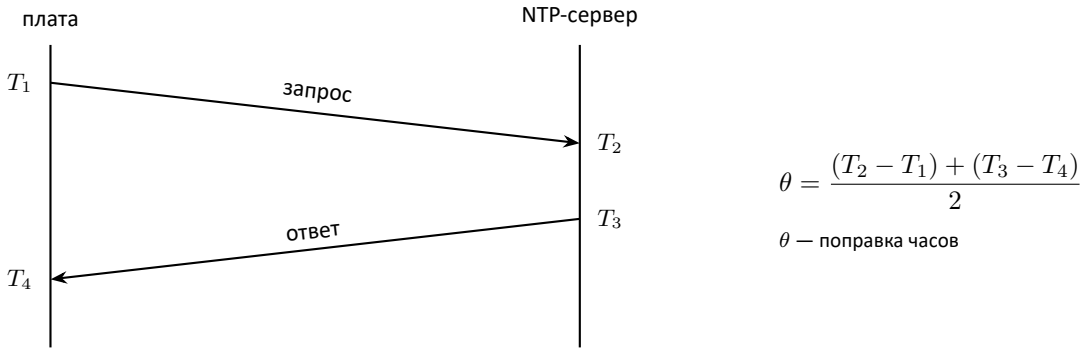


Рис. 16.14: Обмен NTP: четыре отметки времени позволяют вычесть задержку сети

Пример: если плата отстаёт на 5 с, поправка $\theta \approx +5$ с — именно столько надо прибавить к её часам. Ответ NTP всегда в **UTC** (мировое время). Часовой пояс прибавляем сами: для Москвы UTC + 3, т. е. 3×3600 секунд. Переход на «летнее/зимнее» время в России отменён, поэтому достаточно константы.

Листинг 16.13: MicroPython: установка часов по NTP и вывод местного времени

```
import network
import ntptime
import time
import secrets

TZ_HOURS = 3                          # часовой пояс (Москва = UTC+3)

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)

ntptime.host = "pool.ntp.org"         # можно указать ближайший сервер
for attempt in range(5):
    try:
        ntptime.settime()             # записывает UTC во внутренние часы платы
        break
    except OSError as e:              # сервер не ответил --- пробуем ещё
        print("NTP не ответил:", e)
        time.sleep(2)

while True:
    t = time.localtime(time.time() + TZ_HOURS * 3600)   # прибавляем часовой пояс
    print("%04d-%02d-%02d %02d:%02d:%02d" % t[:6])
    time.sleep(1)
```

Листинг 16.14: CircuitPython: то же с `adafruit_ntp`

```
import os
import time
import socketpool
import wifi
import adafruit_ntp

TZ_HOURS = int(os.getenv("TIMEZONE_HOURS", 3))   # часовой пояс из settings.toml

```

```
if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
ntp = adafruit_ntp.NTP(pool, server="pool.ntp.org", tz_offset=TZ_HOURS)

while True:
    t = ntp.datetime                              # struct_time с учётом пояса
    print("%04d-%02d-%02d %02d:%02d:%02d"
          % (t.tm_year, t.tm_mon, t.tm_mday, t.tm_hour, t.tm_min, t.tm_sec))
    time.sleep(1)
```

Это интересно

Обновлять время по NTP достаточно раз в несколько часов: кварц платы уходит всего на секунды в сутки. Слишком частые запросы к публичному серверу считаются невежливыми, а иногда сервер блокирует такого клиента.

16.11 Учимся программировать: обработка ошибок и словари

Работа с сетью — лучшая школа обработки ошибок. Конструкция `try/except` позволяет сказать: «попробуй это; если случится сбой, сделай вот так». Блок `finally` выполняется *всегда* — и при успехе, и при ошибке: идеальное место для `r.close()`. Ещё одна новая идея — **словари**: коллекция пар «ключ — значение», записанная в фигурных скобках. JSON после `r.json()` превращается именно в вложенные словари: `data["current"]["temperature_2m"]`.

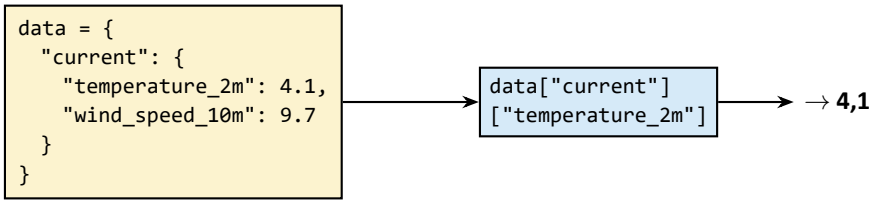


Рис. 16.15: Доступ к значению во вложенном словаре по цепочке ключей

16.12 Практика: «сетевая метеостанция»

Объедините всё в проект:

1. Подключитесь надёжной функцией; светодиод показывает состояние сети.
2. Установите время по NTP.
3. Раз в 10 минут запрашивайте температуру на улице по Open-Meteo, а температуру внутри платы измеряйте сами.
4. Печатайте строку «`время; улица; плата`» — уже почти журнал данных (урок 12).

Если не работает

- **`AttributeError: 'module' object has no attribute 'WLAN'`** — у вас плата без радио: нужна именно Pico W или Pico 2 W.
- **Статус -2, сеть не найдена** — это сеть 5 ГГц или имя написано с ошибкой (регистр букв важен!). Проверьте сканированием, что сеть вообще видна.
- **Статус -3 или «Authentication failure»** — неверный пароль; проверьте раскладку, лишние пробелы, тип защиты (WPA2 надёжнее WPA3-only).

- **Подключение зависает, статус 1** — плохой сигнал, неверный код страны, или сеть скрыта. Приблизьтесь к роутеру, проверьте `rp2.country()`.
- **`OSError: [Errno 110] ETIMEDOUT` или `-2` при запросе** — нет интернета, DNS не отвечает, сервер недоступен. Проверьте `ping` с ноутбука.
- **`MemoryError` при HTTPS** — вызовите `gc.collect()`, закройте предыдущие ответы, уменьшите ответ сервера (параметрами запроса).
- **Время неверное на час и более** — забыт часовой пояс или он прибавлен дважды.
- **Ошибка импорта `requests`, `adafruit_requests`** — библиотека не установлена; установите через `mip` (MicroPython) или `circup` (CircuitPython).

Итоги урока

- Радиочип CYW43439 подключён к RP2040 через PIO; Wi-Fi только 2,4 ГГц; нужна плата W.
- Режимы: станция (STA) — клиент сети, точка доступа (AP) — плата создаёт сеть.
- Подключение: код страны, SSID и пароль (в `secrets.py` или `settings.toml`, не в Git), проверка статуса, повторы с паузами.
- IP-адрес, маска, шлюз и DNS выдаются по DHCP; сила сигнала измеряется в дБм.
- HTTP-клиент: `GET`/`POST`, проверка `status_code`, `r.json()`, обязательно закрывать ответ; HTTPS требует памяти.
- Время по NTP приходит в UTC; часовой пояс прибавляем сами.

Проверь себя

1. Чем отличаются режимы STA и AP? Когда нужен каждый из них?
2. Зачем нужен код страны и почему пароль хранят отдельно от программы?
3. Что означают IP-адрес, маска, шлюз и DNS и как плата их получает?
4. Почему ответ HTTP-запроса нужно закрывать и зачем запрос оборачивают в `try/except`?
5. Почему время по NTP приходит в UTC и как получить местное время?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Подключите Pico W к домашней сети и выведите свой IP-адрес, маску, шлюз и силу сигнала. Объясните, что означает каждое значение.
2. ★ Просканируйте эфир и составьте таблицу сетей: имя, канал, RSSI. Какой канал в вашем доме самый загруженный?
3. ★★ Измените пример погоды: возьмите координаты вашего города и добавьте вывод влажности (`relative_humidity_2m`).
4. ★★ Добавьте в программу подключения светодиод: мигает при подключении, горит при успехе, три коротких вспышки при ошибке пароля.
5. ★★★ Сделайте «будильник по NTP»: плата узнаёт время, а в заданный час включает светодиод. Время обновляйте раз в 6 часов.

6. ★★★★ Создайте «журнал связи»: раз в минуту записывайте в файл время и RSSI, при обрыве — фиксируйте время обрыва и восстановления. Постройте по данным график на компьютере.

Урок 17. Веб-сервер на Pico W

Цель	превратить Pico W в маленький веб-сервер: управлять светодиодом и смотреть показания датчика с телефона через браузер
Опирается на	урок 16 (подключение к Wi-Fi, IP-адреса, HTTP-клиент), урок 2 (светодиод), урок 6 (АЦП)
Результат	страница с кнопками и температурой; JSON-интерфейс с обновлением без перезагрузки страницы; сервер на `asyncio`; собственная сеть Wi-Fi; простая защита ключом
Понадобится	Pico W или Pico 2 W, домашний Wi-Fi, телефон или ноутбук, два светодиода и два резистора 220 Ω

17.1 Идея урока коротко

В прошлом уроке плата была *клиентом*: она сама ходила на чужие серверы. Теперь поменяем роли. Плата станет *сервером*: будет ждать, когда кто-нибудь в сети откроет её адрес в браузере, и отвечать веб-страницей. Тогда пульт управления вашим устройством — это любой телефон или компьютер без установки программ. Для этого нужны три вещи: понимать протокол HTTP, уметь писать немного HTML и уметь работать с сокетами.

17.2 Клиент и сервер

В сети всегда два участника: тот, кто спрашивает (**клиент**), и тот, кто отвечает (**сервер**). Сервер ничего не делает сам: он ждёт запросов. Клиент знает адрес сервера и *инициирует* обмен.

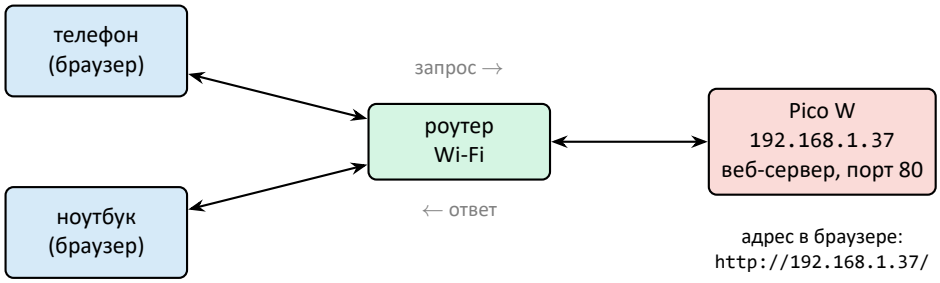


Рис. 17.1: Pico W как сервер: любые устройства домашней сети могут её «спросить»

Адрес сервера состоит из IP-адреса и **порта** — «номера двери» программы на устройстве. Веб использует порт 80 (для HTTP) и 443 (для HTTPS); если порт не указан, браузер подставляет 80.

17.3 HTTP построчно

HTTP — текстовый протокол. Запрос и ответ — обычный текст, который можно прочитать глазами. Запрос состоит из **стартовой строки**, **заголовков** и (необязательно) **тела**, а между заголовками и телом ставится пустая строка.

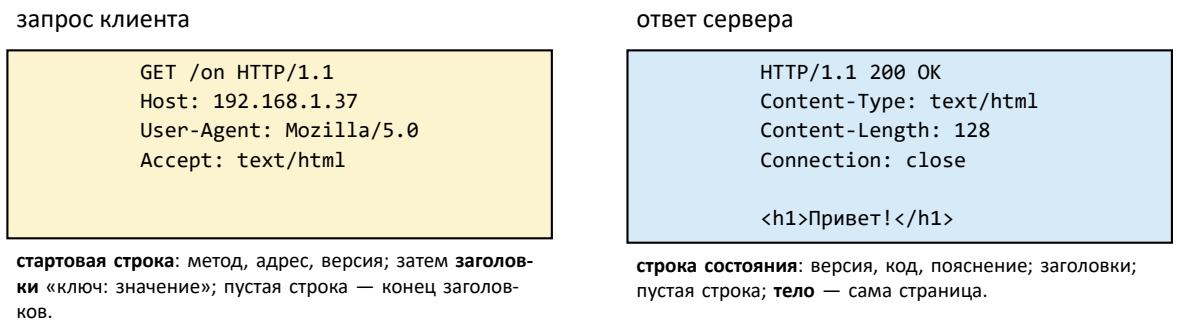


Рис. 17.2: Устройство HTTP-запроса и HTTP-ответа

Коды ответа, которые нужно знать:

Код	Смысл	Когда сервер так отвечает
200	OK	запрос выполнен, в теле — результат
302	Found (перенаправление)	«иди по другому адресу» (заголовок `Location`)
400	Bad Request	запрос непонятен
403	Forbidden	доступ запрещён (неверный ключ)
404	Not Found	такой страницы нет
500	Internal Server Error	ошибка в программе сервера

Таблица 17.1: Основные коды состояния HTTP

Это интересно

Если нажать в браузере «Показать исходный код страницы», вы увидите тот самый HTML, который отправил сервер. А на ноутбуке команда `curl -i http://192.168.1.37/` покажет и заголовки ответа. Это лучший способ отладки: можно посмотреть запрос, не дожидаясь браузера.

17.4 Минимум HTML, CSS и JavaScript

Веб-страница — это три слоя. **HTML** описывает содержимое (заголовки, абзацы, кнопки), **CSS** — внешний вид (цвета, размеры), **JavaScript** — поведение (реакция на нажатия, запросы к серверу без перезагрузки). Нам хватит нескольких тегов:

Тег	Что делает
`<h1>...</h1>`	заголовок (бывают h1–h6)
`<p>...</p>`	абзац
`<b>...</b>`	жирный текст
`<a href="/on">текст</a>`	ссылка: по нажатию браузер запросит адрес /on
`<button>...</button>`	кнопка
`<style>...</style>`	правила CSS
`<script>...</script>`	программа на JavaScript

Таблица 17.2: Теги, которых хватит для панели управления

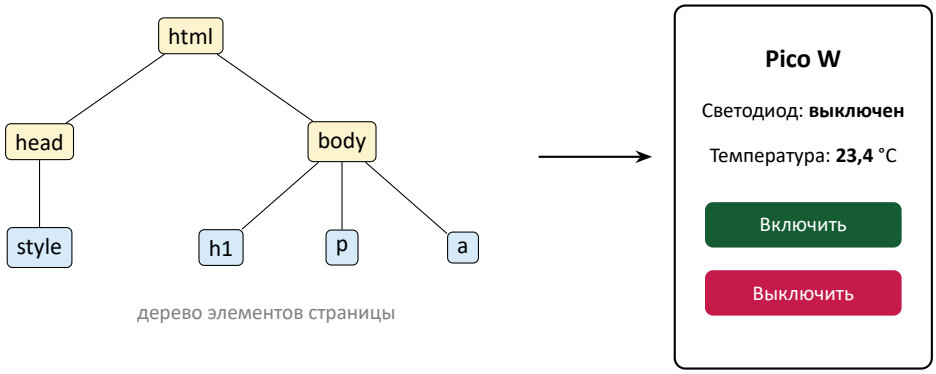


Рис. 17.3: Браузер превращает дерево HTML-элементов в картинку на экране

Вот минимальная страница с кнопками. Обратите внимание на `<meta charset="utf-8">`: без неё русские буквы превратятся в «кракозябры», а `<meta name="viewport"...>` делает страницу удобной на телефоне.

```
<!DOCTYPE html>
<html><head><meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<style> .btn{padding:14px 28px; color:#fff; background:#2e8b57} </style>
</head><body>
  <h1>Pico W</h1>
  <a class="btn" href="/on">Включить</a>
</body></html>
```

17.5 Сокеты: как сервер принимает клиентов

Программа общается по сети через **сокет** (socket) — «розетку», в которую можно писать и из которой можно читать. Сервер на сокетах работает по неизменному сценарию:

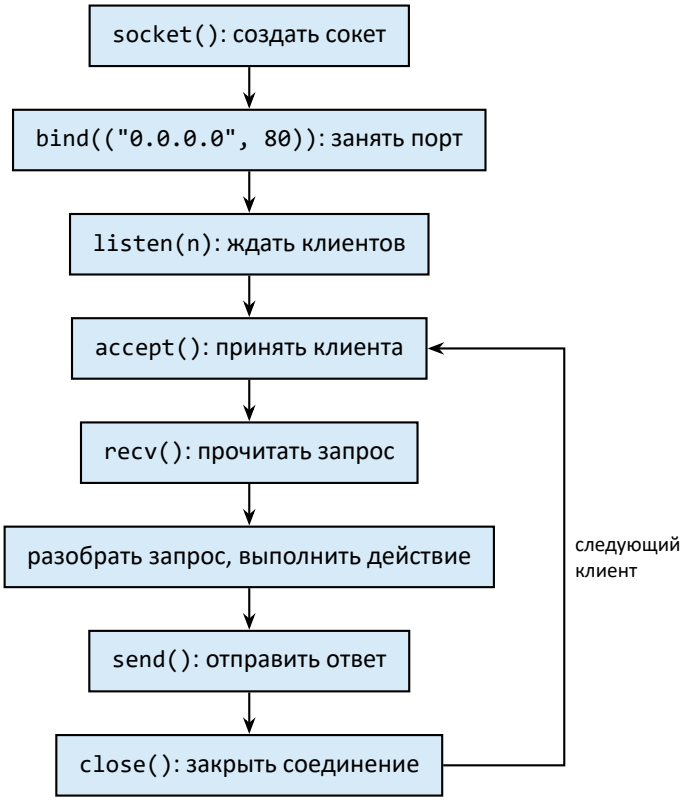


Рис. 17.4: Жизненный цикл простого сокетного сервера

Адрес `"0.0.0.0"` означает «слушать на всех сетевых интерфейсах платы». Метод `accept()` *блокирует* программу: она замирает, пока кто-то не подключится. Метод `recv(1024)` возвращает *байты*, поэтому их надо превратить в текст методом `decode()`, а текст перед отправкой — обратно в байты методом `encode()`.

Листинг 17.1: MicroPython: самый короткий веб-сервер

```
import network
import socket
import time
import secrets

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)
print("Откройте в браузере: http://%s/" % wlan.ifconfig()[0])

s = socket.socket()                                  # TCP-сокет
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 80))                              # порт 80 --- стандартный для HTTP
s.listen(2)                                          # очередь из двух ожидающих клиентов

while True:
    client, addr = s.accept()                        # ждём, пока кто-то подключится
    print("клиент:", addr)
    request = client.recv(1024)                      # читаем запрос (байты)
    print(request.decode().split("\r\n")[0])         # первая строка: GET / HTTP/1.1
    body = "<h1>Привет от Pico W!</h1>"
    client.send("HTTP/1.1 200 OK\r\n"
```

```
                "Content-Type: text/html; charset=utf-8\r\n"
                "Connection: close\r\n\r\n")          # пустая строка завершает заголовки
    client.send(body.encode("utf-8"))
    client.close()                                    # закрываем: ответ готов
```

Листинг 17.2: CircuitPython: то же с библиотекой `adafruit_httpserver`

```
import os
import socketpool
import wifi
from adafruit_httpserver import Server, Request, Response

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
server = Server(pool, debug=True)             # debug=True печатает запросы


@server.route("/")                            # обработчик для адреса «/»
def home(request: Request):
    return Response(request, "<h1>Привет от Pico W!</h1>",
                    content_type="text/html; charset=utf-8")

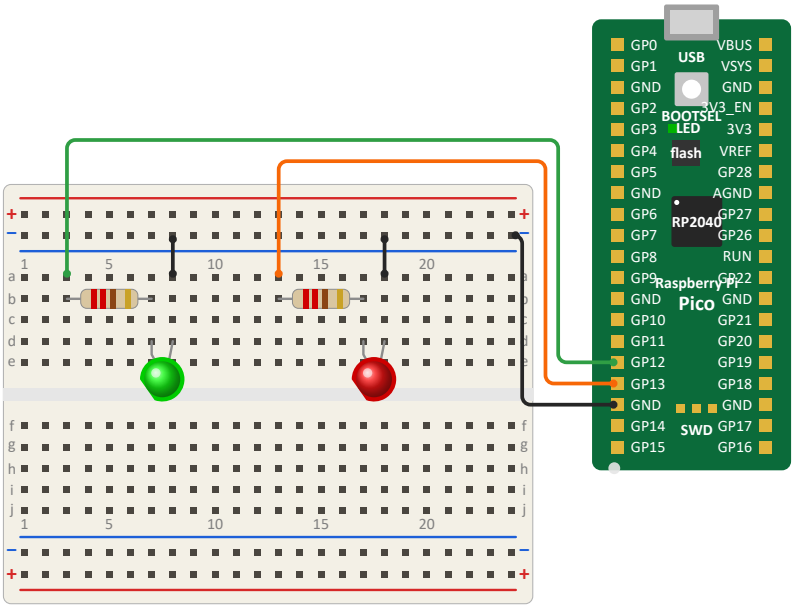

server.start(str(wifi.radio.ipv4_address), 80)
print("Откройте в браузере: http://%s/" % wifi.radio.ipv4_address)

while True:
    server.poll()                             # обработать входящие запросы (если есть)
```

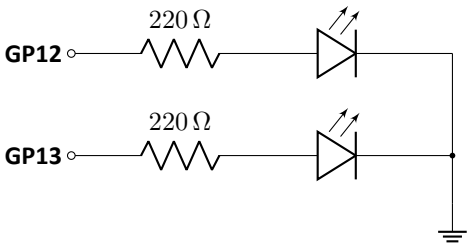
В CircuitPython сокеты скрыты внутри библиотеки: вы описываете, *какая функция отвечает на какой адрес* (декоратор `@server.route`), а в главном цикле просто вызываете `server.poll()`. Библиотеку ставят командой `circup install adafruit_httpserver`.

17.6 Практика: панель управления светодиодом

Подключим два светодиода: зелёный на **GP12** и красный на **GP13**; сервер будет управлять красным, а зелёный пригодится для заданий. Резисторы нужны обязательно (урок 2).



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 17.5: Два светодиода: зелёный на GP12, красный на GP13

Логика сервера — «маршрутизатор»: смотрим на адрес в запросе и решаем, что делать. Для адреса `/on` включаем светодиод, для `/off` выключаем, для остальных просто показываем страницу с текущим состоянием.

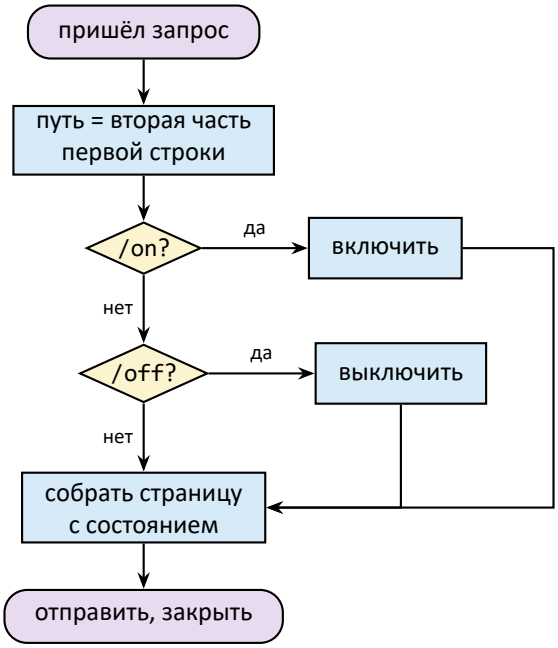


Рис. 17.6: Маршрутизация запроса: адрес определяет действие, ответ собирается в конце

Листинг 17.3: MicroPython: страница с кнопками и температурой

```
import network
import socket
import time
from machine import Pin, ADC
import secrets

led = Pin(13, Pin.OUT)                  # внешний светодиод на GP13
sensor = ADC(4)                         # температура кристалла

PAGE = """<!DOCTYPE html>
<html><head><meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Pico W</title>
<style>
body{font-family:sans-serif;text-align:center;background:#f4f4f4}
.btn{display:inline-block;padding:14px 28px;margin:8px;border-radius:8px;
     color:#fff;text-decoration:none;font-size:20px}
.on{background:#2e8b57} .off{background:#c0392b}
</style></head><body>
<h1>Pico W</h1>
<p>Светодиод: <b>{{LED}}</b></p>
<p>Температура: <b>{{T}}</b> °C</p>
<a class="btn on" href="/on">Включить</a>
<a class="btn off" href="/off">Выключить</a>
</body></html>"""


def temp_c():
    v = sensor.read_u16() * 3.3 / 65535
    return 27 - (v - 0.706) / 0.001721


def make_page():
    html = PAGE.replace("{{LED}}", "включён" if led.value() else "выключен")
```

```
    return html.replace("{{T}}", "%.1f" % temp_c())


wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)
print("Откройте в браузере: http://%s/" % wlan.ifconfig()[0])

s = socket.socket()
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 80))
s.listen(3)

while True:
    client, addr = s.accept()
    try:
        client.settimeout(3)                          # не ждать клиента вечно
        request = client.recv(1024).decode()
        parts = request.split("\r\n")[0].split(" ")  # ["GET", "/on", "HTTP/1.1"]
        path = parts[1] if len(parts) > 1 else "/"
        if path == "/on":
            led.value(1)
        elif path == "/off":
            led.value(0)
        body = make_page().encode("utf-8")
        client.send("HTTP/1.1 200 OK\r\n"
                    "Content-Type: text/html; charset=utf-8\r\n"
                    "Content-Length: %d\r\n"
                    "Connection: close\r\n\r\n" % len(body))
        client.send(body)
    except OSError as e:                               # клиент отвалился --- не страшно
        print("ошибка:", e)
    finally:
        client.close()
```

Листинг 17.4: CircuitPython: то же

```
import os
import board
import digitalio
import microcontroller
import socketpool
import wifi
from adafruit_httpserver import Server, Request, Response, Redirect

led = digitalio.DigitalInOut(board.GP13)       # внешний светодиод на GP13
led.direction = digitalio.Direction.OUTPUT

PAGE = """<!DOCTYPE html>
<html><head><meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Pico W</title>
<style>
body{font-family:sans-serif;text-align:center;background:#f4f4f4}
.btn{display:inline-block;padding:14px 28px;margin:8px;border-radius:8px;
     color:#fff;text-decoration:none;font-size:20px}
```

```
.on{background:#2e8b57} .off{background:#c0392b}
</style></head><body>
<h1>Pico W</h1>
<p>Светодиод: <b>{{LED}}</b></p>
<p>Температура: <b>{{T}}</b> °C</p>
<a class="btn on" href="/on">Включить</a>
<a class="btn off" href="/off">Выключить</a>
</body></html>"""

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
server = Server(pool, debug=True)


@server.route("/")
def home(request: Request):
    html = PAGE.replace("{{LED}}", "включён" if led.value else "выключен")
    html = html.replace("{{T}}", "%.1f" % microcontroller.cpu.temperature)
    return Response(request, html, content_type="text/html; charset=utf-8")


@server.route("/on")
def led_on(request: Request):
    led.value = True
    return Redirect(request, "/")              # вернуть пользователя на главную


@server.route("/off")
def led_off(request: Request):
    led.value = False
    return Redirect(request, "/")


server.start(str(wifi.radio.ipv4_address), 80)
print("Откройте в браузере: http://%s/" % wifi.radio.ipv4_address)
while True:
    server.poll()
```

Запустите программу, посмотрите в консоли адрес и откройте его на телефоне (в той же сети!). Нажмите «Включить» — браузер запросит `/on`, плата включит светодиод и вернёт страницу. Адрес в строке браузера при этом изменится на `.../on`. В **CircuitPython** лучше: функция возвращает `Redirect` (HTTP-код 302), и браузер сам возвращается на главную.

Осторожно

- Запросы по адресу меняют состояние (включают светодиод) — значит, их нельзя выдавать «на чтение»: браузеры иногда сами запрашивают адрес заранее (предзагрузка, иконка `/favicon.ico`). Для настоящих систем изменения делают запросами `POST`, а не `GET`. В учебных примерах мы упрощаем.
- Один пользователь — один запрос за раз: пока сервер обслуживает клиента, остальные ждут. Не ставьте внутрь обработчика долгие операции и `sleep()`.

17.7 Учимся программировать: строки и словари

Разбор запроса — это работа со строками. Пригодятся методы:

- `s.split(" ")` режет строку по пробелам: `"GET /on HTTP/1.1".split(" ")` → `["GET", "/on", "HTTP/1.1"]`;
- `s.partition("?")` режет строку *один раз* по первому разделителю и возвращает тройку (до, разделитель, после);
- `s.replace("T", "23.4")` заменяет вхождения — так мы вставляем значения в «шаблон» страницы;
- `"%.1f" % x` форматирует число с одним знаком после запятой.

А для параметров запроса (`?value=1&key=abc`) удобен **словарь**: `params["value"]` вернёт `"1"`. Метод `params.get("key")` безопаснее: если ключа нет, он вернёт `None`, а не вызовет ошибку.

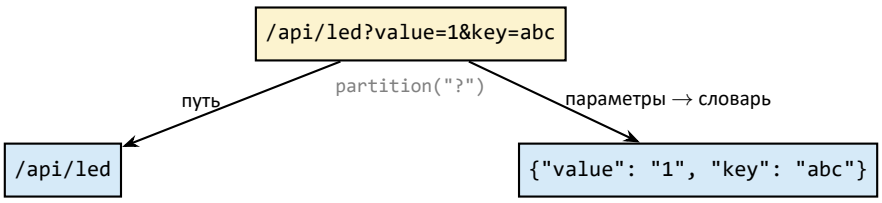


Рис. 17.7: Адрес запроса разбирается на путь и словарь параметров

17.8 JSON-интерфейс и `fetch()`

Страница из предыдущего примера перезагружается целиком при каждом нажатии. Современные интерфейсы устроены иначе: страница загружается один раз, а затем JavaScript незаметно запрашивает у сервера *только данные* (в формате JSON) функцией `fetch()` и обновляет нужные куски страницы. Такой «программный» интерфейс сервера называют **API**.

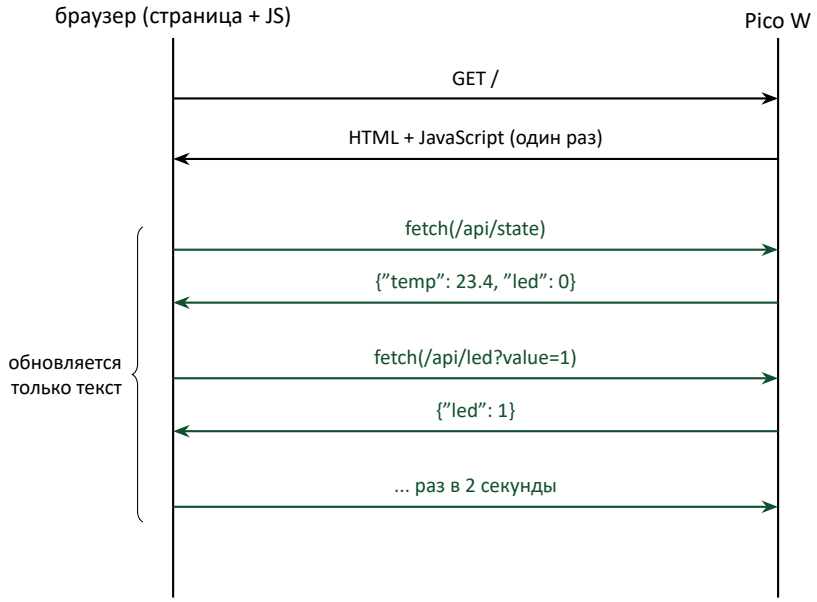


Рис. 17.8: После первой загрузки страница обменивается с платой только маленькими JSON-сообщениями

Листинг 17.5: MicroPython: сервер с адресами `/api/state` и `/api/led`

```
import network
import socket
import json
import time
from machine import Pin, ADC
import secrets

led = Pin(13, Pin.OUT)
sensor = ADC(4)

# Страница не перезагружается: JavaScript раз в 2 с спрашивает /api/state
PAGE = """<!DOCTYPE html><html><head><meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Pico API</title></head>
<body style="font-family:sans-serif;text-align:center">
<h1>Pico W</h1>
<p>Температура: <b id="t">--</b> °C</p>
<p>Светодиод: <b id="led">--</b></p>
<button onclick="setLed(1)">Включить</button>
<button onclick="setLed(0)">Выключить</button>
<script>
async function upd() {
  try {
    const r = await fetch('/api/state');
    const d = await r.json();
    document.getElementById('t').textContent = d.temp.toFixed(1);
    document.getElementById('led').textContent = d.led ? 'включён' : 'выключен';
  } catch (e) { document.getElementById('t').textContent = 'нет связи'; }
}
async function setLed(v) { await fetch('/api/led?value=' + v); upd(); }
setInterval(upd, 2000); upd();
</script></body></html>"""


def temp_c():
    v = sensor.read_u16() * 3.3 / 65535
    return 27 - (v - 0.706) / 0.001721


def parse(target):
    """'/api/led?value=1' -> ('/api/led', {'value': '1'})"""
    path, _, query = target.partition("?")
    params = {}
    if query:
        for pair in query.split("&"):
            k, _, v = pair.partition("=")
            params[k] = v
    return path, params


def respond(client, body, ctype="application/json", code="200 OK"):
    if isinstance(body, str):
        body = body.encode("utf-8")
    client.send("HTTP/1.1 %s\r\nContent-Type: %s; charset=utf-8\r\n"
                "Content-Length: %d\r\nConnection: close\r\n\r\n"
                % (code, ctype, len(body)))
```

```
    client.send(body)


wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)
print("http://%s/" % wlan.ifconfig()[0])

s = socket.socket()
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 80))
s.listen(3)

while True:
    client, addr = s.accept()
    try:
        client.settimeout(3)
        parts = client.recv(1024).decode().split("\r\n")[0].split(" ")
        path, params = parse(parts[1] if len(parts) > 1 else "/")
        if path == "/":
            respond(client, PAGE, "text/html")
        elif path == "/api/state":
            respond(client, json.dumps({"temp": temp_c(), "led": led.value()}))
        elif path == "/api/led":
            led.value(1 if params.get("value") == "1" else 0)
            respond(client, json.dumps({"led": led.value()}))
        else:
            respond(client, "Not found", "text/plain", "404 Not Found")
    except OSError as e:
        print("ошибка:", e)
    finally:
        client.close()
```

Листинг 17.6: CircuitPython: то же с `JSONResponse`

```
import os
import board
import digitalio
import microcontroller
import socketpool
import wifi
from adafruit_httpserver import Server, Request, Response, JSONResponse

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

# Страница не перезагружается: JavaScript раз в 2 с спрашивает /api/state
PAGE = """<!DOCTYPE html><html><head><meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Pico API</title></head>
<body style="font-family:sans-serif;text-align:center">
<h1>Pico W</h1>
<p>Температура: <b id="t">--</b> °C</p>
<p>Светодиод: <b id="led">--</b></p>
<button onclick="setLed(1)">Включить</button>
<button onclick="setLed(0)">Выключить</button>
```

```
<script>
async function upd() {
  try {
    const r = await fetch('/api/state');
    const d = await r.json();
    document.getElementById('t').textContent = d.temp.toFixed(1);
    document.getElementById('led').textContent = d.led ? 'включён' : 'выключен';
  } catch (e) { document.getElementById('t').textContent = 'нет связи'; }
}
async function setLed(v) { await fetch('/api/led?value=' + v); upd(); }
setInterval(upd, 2000); upd();
</script></body></html>"""

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
server = Server(pool)


@server.route("/")
def home(request: Request):
    return Response(request, PAGE, content_type="text/html; charset=utf-8")


@server.route("/api/state")
def state(request: Request):
    return JSONResponse(request, {"temp": microcontroller.cpu.temperature,
                                  "led": led.value})


@server.route("/api/led")
def set_led(request: Request):
    led.value = request.query_params.get("value") == "1"   # ?value=1
    return JSONResponse(request, {"led": led.value})


server.start(str(wifi.radio.ipv4_address), 80)
print("http://%s/" % wifi.radio.ipv4_address)
while True:
    server.poll()
```

Здесь JSON работает в обе стороны: плата формирует его функцией `json.dumps()` (в **CircuitPython** это делает `JSONResponse`), а JavaScript разбирает методом `r.json()`. Попробуйте открыть в браузере адрес `/api/state`: вы увидите JSON как есть. Такой интерфейс можно использовать не только из страницы, но и из других программ — например, из второй платы Pico W (урок 16).

17.9 Асинхронный сервер

Простой сервер с `accept()` обслуживает клиентов по одному: пока он ждёт данных от медленного клиента, больше ничего не происходит. А нам нужно ещё и измерять датчик, мигать светодиодом и т. п. Выход — **асинхронное** программирование (урок 13): задачи добровольно уступают друг другу управление в точках `await`, и процессор переключается между ними, пока одна ждёт.

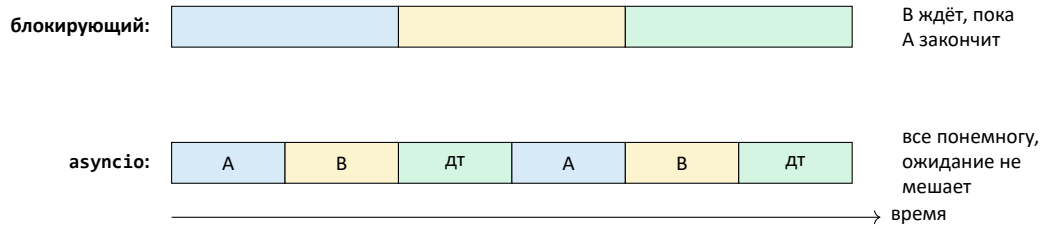


Рис. 17.9: Блокирующий сервер делает дела строго по очереди, асинхронный чередует задачи в моменты ожидания

В **MicroPython** модуль `asyncio` встроен, и у него есть готовый TCP-сервер `asyncio.start_server()`: для каждого клиента запускается своя функция-обработчик, которая получает `reader` и `writer`. В **CircuitPython** есть библиотека `asyncio` (устанавливается через `circup`), но `adafruit_httpserver` не асинхронна по своей сути: её `poll()` просто вызывают в цикле одной из задач и после него делают `await asyncio.sleep(0)`.

Листинг 17.7: MicroPython: асинхронный сервер и фоновая задача измерения

```
import asyncio
import json
import network
import time
from machine import ADC
import secrets

sensor = ADC(4)
avg = 0.0                                   # сглаженная температура (общая для задач)


def temp_c():
    v = sensor.read_u16() * 3.3 / 65535
    return 27 - (v - 0.706) / 0.001721


async def sampler():
    """Фоновая задача: раз в секунду обновляет температуру."""
    global avg
    avg = temp_c()
    while True:
        avg = 0.8 * avg + 0.2 * temp_c()    # экспоненциальный фильтр
        await asyncio.sleep(1)              # уступаем управление другим задачам


async def handle(reader, writer):
    """Обслуживает одного клиента; сервер при этом не блокируется."""
    try:
        request_line = await reader.readline()          # b"GET /api HTTP/1.1\r\n"
        while True:                                     # пропускаем заголовки
            h = await reader.readline()
            if h in (b"\r\n", b""):
                break
        path = request_line.decode().split(" ")[1]
        if path == "/api":
            body = json.dumps({"temp": round(avg, 1), "uptime": time.ticks_ms() // 1000})
            ctype, code = "application/json", "200 OK"
        else:
            body, ctype, code = "Not found", "text/plain", "404 Not Found"
```

```
        writer.write(("HTTP/1.0 %s\r\nContent-Type: %s\r\n\r\n%s" % (code, ctype,
        ↪  body)).encode())
        await writer.drain()                            # дождаться отправки
    except Exception as e:
        print("ошибка клиента:", e)
    finally:
        writer.close()
        await writer.wait_closed()


async def main():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(secrets.SSID, secrets.PASSWORD)
    while not wlan.isconnected():
        await asyncio.sleep(0.5)
    print("http://%s/api" % wlan.ifconfig()[0])
    asyncio.create_task(sampler())                      # запускаем фоновую задачу
    await asyncio.start_server(handle, "0.0.0.0", 80)   # запускаем сервер
    while True:
        await asyncio.sleep(3600)                       # главная задача просто живёт


asyncio.run(main())
```

Листинг 17.8: CircuitPython: опрос `poll()` в задаче `asyncio`

```
import asyncio
import os
import microcontroller
import socketpool
import wifi
from adafruit_httpserver import Server, Request, JSONResponse

avg = microcontroller.cpu.temperature       # сглаженная температура (общая для задач)

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
server = Server(pool)


@server.route("/api")
def api(request: Request):
    return JSONResponse(request, {"temp": round(avg, 1)})


async def sampler():
    """Фоновая задача: раз в секунду обновляет температуру."""
    global avg
    while True:
        avg = 0.8 * avg + 0.2 * microcontroller.cpu.temperature
        await asyncio.sleep(1)


async def serve():
```

```
    """Веб-сервер adafruit_httpserver опрашивается внутри цикла asyncio."""
    server.start(str(wifi.radio.ipv4_address), 80)
    print("http://%s/api" % wifi.radio.ipv4_address)
    while True:
        server.poll()
        await asyncio.sleep(0)              # дать поработать другим задачам


async def main():
    await asyncio.gather(sampler(), serve())


asyncio.run(main())
```

Совет

Правило асинхронного кода: в обработчике нет ни `time.sleep()`, ни длинных циклов без `await`. Любая такая остановка замораживает *всех*: и сервер, и измерения. Ждать нужно только через `await asyncio.sleep(...)`.

Готовые библиотеки

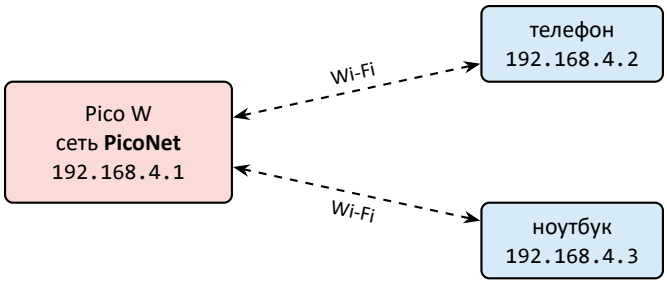
Писать разбор запросов вручную полезно для понимания, но в больших проектах берут готовые решения:

- **`microdot`** для **MicroPython** — крошечный фреймворк в стиле Flask: маршруты-декораторы, параметры, JSON, работает поверх `asyncio`. Устанавливается `mip.install("github:miguelgrinberg/microdot/src/microdot/microdot.py")` (точную команду смотрите в документации проекта).
- **`adafruit_httpserver`** для **CircuitPython** — то, чем мы пользовались: маршруты, формы, JSON, статические файлы, WebSocket-подобные возможности зависят от версии.

Принципы остаются теми же: маршрут, обработчик, ответ.

17.10 Собственная сеть: режим точки доступа

Что делать, если роутера нет? Плата сама создаёт Wi-Fi сеть (режим AP, урок 16), к ней подключается телефон, и дальше всё работает так же. IP-адрес платы в такой сети по умолчанию — `192.168.4.1`. Пароль для WPA2 должен быть не короче 8 символов; без пароля сеть будет открытой, и любой в радиусе действия сможет управлять вашей платой.



плата выдаёт адреса сама (DHCP); выхода в интернет нет

Рис. 17.10: Pico W в режиме точки доступа: собственная «мини-сеть»

Листинг 17.9: MicroPython: точка доступа с переключателем светодиода

```
import network
import socket
import time
from machine import Pin

led = Pin("LED", Pin.OUT)

ap = network.WLAN(network.AP_IF)              # режим точки доступа
ap.config(essid="PicoNet", password="pico12345")   # пароль не короче 8 символов
ap.active(True)
while not ap.active():
    time.sleep(0.1)
print("Сеть создана. Адрес платы:", ap.ifconfig()[0])   # обычно 192.168.4.1

s = socket.socket()
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 80))
s.listen(3)

while True:
    client, addr = s.accept()
    try:
        client.settimeout(3)
        parts = client.recv(1024).decode().split("\r\n")[0].split(" ")
        path = parts[1] if len(parts) > 1 else "/"
        if path == "/toggle":
            led.toggle()
        body = ("<meta charset='utf-8'><h1>Я --- Pico W</h1>"
                "<p>Светодиод: %s</p><a href='/toggle'>Переключить</a>"
                % ("включён" if led.value() else "выключен")).encode("utf-8")
        client.send("HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=utf-8\r\n"
                    "Connection: close\r\n\r\n")
        client.send(body)
    except OSError as e:
        print("ошибка:", e)
    finally:
        client.close()
```

Листинг 17.10: CircuitPython: то же

```
import board
import digitalio
import socketpool
import wifi
from adafruit_httpserver import Server, Request, Response, Redirect

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

# Создаём собственную сеть; пароль не короче 8 символов
wifi.radio.start_ap(ssid="PicoNet", password="pico12345")
print("Сеть создана. Адрес платы:", wifi.radio.ipv4_address_ap)   # обычно 192.168.4.1

pool = socketpool.SocketPool(wifi.radio)
server = Server(pool)
```

```
16
17
18 @server.route("/")
19 def home(request: Request):
20     html = ("<meta charset='utf-8'><h1>Я --- Pico W</h1>"
21             "<p>Светодиод: %s</p><a href='/toggle'>Переключить</a>"
22             % ("включён" if led.value else "выключен"))
23     return Response(request, html, content_type="text/html; charset=utf-8")
24
25
26 @server.route("/toggle")
27 def toggle(request: Request):
28     led.value = not led.value
29     return Redirect(request, "/")
30
31
32 server.start(str(wifi.radio.ipv4_address_ap), 80)
33 while True:
34     server.poll()
```

Подключитесь телефоном к сети `PicoNet`, откройте в браузере `http://192.168.4.1/`. Телефон может ворчать, что «в сети нет интернета» — это нормально, не отключайтесь.

Идея «captive portal»

Когда вы подключаетесь к Wi-Fi в кафе, телефон сам открывает страницу входа. Это **captive portal** («портал-ловушка»): точка доступа отвечает на *любой* запрос одной и той же страницей. Для этого плата в режиме AP запускает ещё и мини-DNS-сервер, который на любое имя отвечает своим адресом, а веб-сервер на любой путь отдаёт страницу настройки. Так делают «мастера первоначальной настройки» устройств: пользователь вводит название и пароль домашней сети, плата сохраняет их в файл и перезагружается уже в режиме станции. Полностью реализовывать портал мы не будем, но идею запомните: она пригодится в итоговом проекте (урок 25).

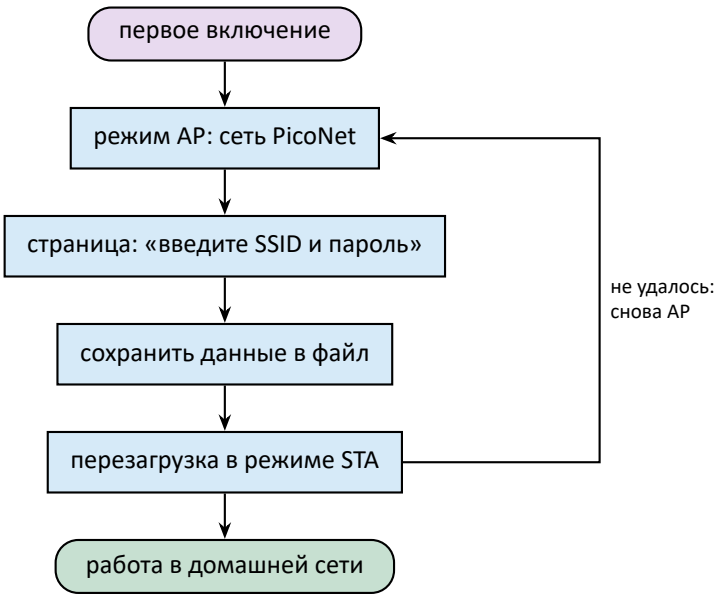


Рис. 17.11: Типичный сценарий первоначальной настройки через точку доступа

17.11 Безопасность

Наш сервер — как дверь без замка: любой, кто знает адрес и находится в той же сети, может нажать кнопку. Причём:

- **HTTP не шифрует данные.** Всё, что вы отправляете (в том числе пароли и ключи), видно любому, кто «слушает» сеть.
- **Нет проверки, кто вы.** Если не добавить авторизацию, управлять сможет любой.
- Если открыть плату наружу (проброс порта на роутере), на неё очень быстро начнут приходить запросы от автоматических сканеров из интернета.

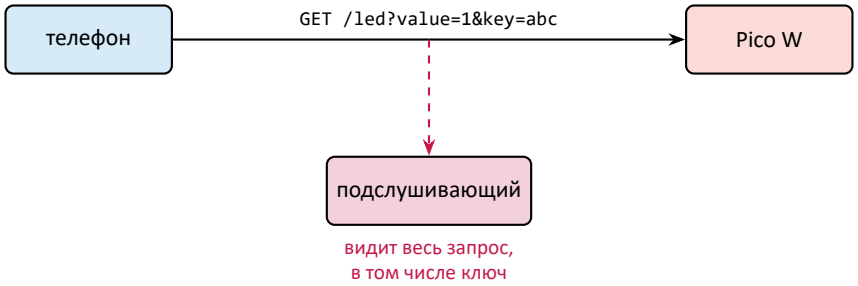


Рис. 17.12: В открытой сети HTTP-запрос читается как обычный текст: ключ в адресе виден всем

Что можно сделать в учебном проекте. Использовать *ключ* (токен): команда выполняется, только если в запросе правильное значение `key`. Ключ хранится в `secrets.py` (`settings.toml`), а не в программе. Добавьте счётчик неудачных попыток и задержку — так перебор ключей станет медленным. Не публикуйте адрес платы и не пробрасывайте порт в интернет. Настоящую безопасность даёт HTTPS и авторизация на отдельном сервере.

Листинг 17.11: MicroPython: команда выполняется только с правильным ключом

```
import network
import socket
import time
from machine import Pin
import secrets                                  # в secrets.py добавлена строка TOKEN = "..."

led = Pin(13, Pin.OUT)


def parse(target):
    """'/led?value=1&key=abc' -> ('/led', {'value': '1', 'key': 'abc'})"""
    path, _, query = target.partition("?")
    params = {}
    for pair in query.split("&"):
        if pair:
            k, _, v = pair.partition("=")
            params[k] = v
    return path, params


def reply(client, code, text):
    body = text.encode("utf-8")
    client.send("HTTP/1.1 %s\r\nContent-Type: text/plain; charset=utf-8\r\n"
                "Content-Length: %d\r\nConnection: close\r\n\r\n" % (code, len(body)))
    client.send(body)

```

```

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)
print("http://%s/led?value=1&key=ВАШ_TOKEN" % wlan.ifconfig()[0])

s = socket.socket()
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 80))
s.listen(3)

fails = 0                                   # счётчик неудачных попыток
while True:
    client, addr = s.accept()
    try:
        client.settimeout(3)
        parts = client.recv(1024).decode().split("\r\n")[0].split(" ")
        path, params = parse(parts[1] if len(parts) > 1 else "/")
        if params.get("key") != secrets.TOKEN:       # ключ неверный или не указан
            fails += 1
            print("отказ от", addr[0], "попыток:", fails)
            time.sleep(min(fails, 5))                # замедляем перебор
            reply(client, "403 Forbidden", "Доступ запрещён")
        elif path == "/led":
            fails = 0
            led.value(1 if params.get("value") == "1" else 0)
            reply(client, "200 OK", "Светодиод: %d" % led.value())
        else:
            reply(client, "404 Not Found", "Нет такой страницы")
    except OSError as e:
        print("ошибка:", e)
    finally:
        client.close()
```

Листинг 17.12: CircuitPython: то же

```
import os
import time
import board
import digitalio
import socketpool
import wifi
from adafruit_httpserver import Server, Request, Response

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

TOKEN = os.getenv("API_TOKEN")              # строка API_TOKEN = "..." в settings.toml

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
server = Server(pool)
fails = 0                                   # счётчик неудачных попыток
```

```
21
22
23 @server.route("/led")
24 def set_led(request: Request):
25     global fails
26     if request.query_params.get("key") != TOKEN:     # ключ неверный или не указан
27         fails += 1
28         print("отказ, попыток:", fails)
29         time.sleep(min(fails, 5))                    # замедляем перебор
30         return Response(request, "Доступ запрещён", status=(403, "Forbidden"))
31     fails = 0
32     led.value = request.query_params.get("value") == "1"
33     return Response(request, "Светодиод: %d" % led.value)
34
35
36 server.start(str(wifi.radio.ipv4_address), 80)
37 print("http://%s/led?value=1&key=ВАШ_ТОКЕН" % wifi.radio.ipv4_address)
38 while True:
39     server.poll()
```

Осторожно

Ключ в адресе — защита лишь от случайного вмешательства, не от злоумышленника: в открытой сети его можно подсмотреть. Не используйте плату с таким сервером для замка двери, обогревателя без термозащиты и других опасных устройств. Для `key` придумайте длинное случайное значение, а не «1234».

17.12 Практика: «умная лампа»

Соберите схему на рис. 17.5. Запустите сервер с JSON-интерфейсом и откройте страницу на телефоне. Добавьте к странице кнопку для зелёного светодиода и показ состояния обоих. Проверьте, что страница работает на двух устройствах одновременно (и что показания обновляются у обоих).

Если не работает

- **Страница не открывается** — телефон и плата в разных сетях (гостевая сеть, мобильный интернет), неверный IP, забыт `http://`. Проверьте IP в консоли.
- **Вместо русских букв «кракозябры»** — нет `<meta charset="utf-8">` или заголовка `charset=utf-8` в ответе.
- **`OSError: [Errno 98] EADDRINUSE`** — порт занят: предыдущий запуск сервера не закрыл сокет. Перезагрузите плату или используйте `SO_REUSEADDR`.
- **Страница открывается очень медленно** — в обработчике есть долгие операции; браузер ещё и запрашивает `/favicon.ico`. Добавьте ответ 404 для неизвестных адресов.
- **После нескольких нажатий сервер замолкает** — не закрываются сокеты: проверьте `client.close()` в блоке `finally`, смотрите на `MemoryError`.
- **CircuitPython: `ImportError: no module named adafruit_httpserver`** — библиотека не скопирована в `CIRCUITPY/lib`; установите `circup install adafruit_httpserver`.
- **AP: телефон отключается от сети** — телефон считает, что интернета нет, и переключается на мобильную передачу данных; отключите автоматическое переключение для этой сети.

Итоги урока

- Сервер ждёт запросов, клиент их посылает; HTTP — текстовый протокол: стартовая строка, заголовки, пустая строка, тело.
- Сокетный сервер: `socket`, `bind`, `listen`, `accept`, `recv`, `send`, `close`; ответ нужно кодировать в UTF-8.
- HTML описывает страницу, CSS — вид, JavaScript с `fetch()` обновляет данные без перезагрузки; обмен идёт в формате JSON.
- `asyncio` позволяет обслуживать клиентов и делать другие дела; внутри задач нельзя блокировать выполнение.
- Режим точки доступа даёт собственную сеть; адрес платы обычно `192.168.4.1`.
- HTTP без шифрования: ключ в адресе — слабая защита; не открывайте плату в интернет и не управляйте опасными устройствами.

Проверь себя

1. Из каких частей состоят HTTP-запрос и ответ? Что означают коды 200, 404 и 403?
2. Зачем серверу нужны `bind`, `listen` и `accept`? Что делает `accept()`?
3. Чем `fetch()` с JSON лучше перезагрузки страницы целиком?
4. Почему в асинхронном обработчике нельзя использовать `time.sleep()`?
5. Почему ключ в адресе не защищает от подслушивания и как можно повысить безопасность?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Запустите сервер из урока, откройте страницу с телефона и включите светодиод. Найдите в консоли платы строку запроса и объясните её части.
2. ★ Измените заголовок и цвета кнопок на странице. Добавьте к странице абзац с вашим именем.
3. ★★ Добавьте второй светодиод (GP12) и вторую пару кнопок; страница должна показывать состояние обоих.
4. ★★ Добавьте в API адрес `/api/uptime`, возвращающий время работы платы в секундах, и покажите его на странице.
5. ★★★ Сделайте страницу с графиком: JavaScript хранит последние 30 значений температуры и рисует линию в элементе `<canvas>`.
6. ★★★★ Реализуйте мини-«captive portal»: плата в режиме AP показывает форму для ввода SSID и пароля, сохраняет их в файл и перезапускается в режиме станции (используйте подсказки урока 12).

Урок 18. MQTT и интернет вещей

Цель	понять модель «публикация — подписка», настроить свой брокер MQTT и построить надёжный узел, который отправляет данные датчика и принимает команды
Опирается на	урок 16 (Wi-Fi и надёжное подключение), урок 17 (клиент и сервер, JSON), урок 3 (кнопка)
Результат	плата публикует температуру в формате JSON; светодиод включается сообщением с телефона; узел сам переподключается и сообщает брокеру о своём состоянии; датчик виден в Home Assistant
Понадобится	Pico W, брокер Mosquitto (на ПК или Raspberry Pi), светодиод и резистор 220 Ω, кнопка, приложение MQTT на телефоне

18.1 Идея урока коротко

В прошлом уроке мы управляли платой через веб-страницу: клиент спрашивает адрес платы и получает ответ. Но представьте сто датчиков и десять экранов: каждый экран должен знать адрес каждого датчика и опрашивать их по очереди. Это неудобно. Протокол **MQTT** решает задачу иначе: между устройствами стоит посредник — **брокер**. Датчики *публикуют* сообщения в «тему» (topic), экраны *подписываются* на нужные темы, а брокер раздаёт сообщения. Устройства не знают друг друга и могут появляться и исчезать. Поэтому MQTT — главный язык интернета вещей.

18.2 Модель «публикация — подписка»

Сравните с газетным киоском: редакция (издатель) не знает читателей, а читатель не знает редакцию. Читатель подписывается на газету, киоск (брокер) выдаёт номера. Издатель отправляет сообщение только брокеру, брокер сам доставляет его всем подписчикам.

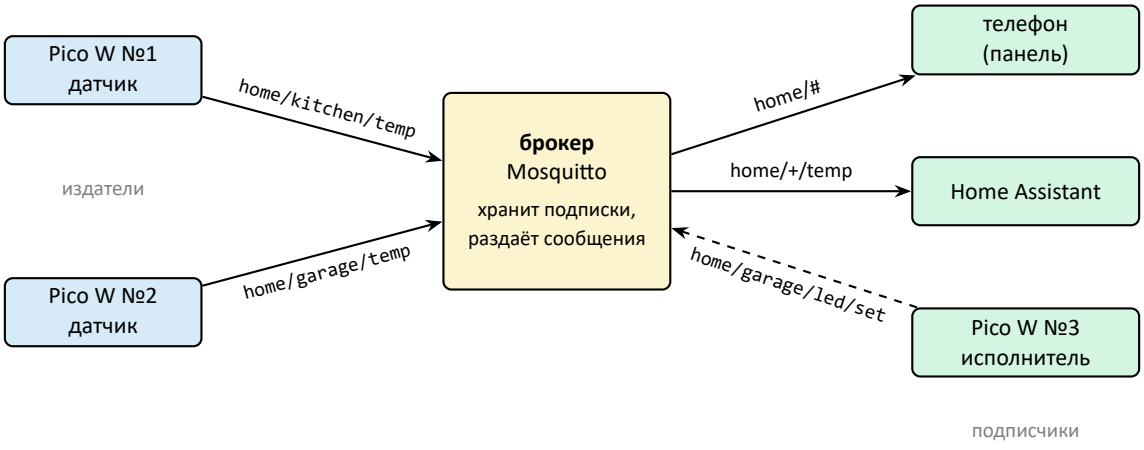


Рис. 18.1: Издатели и подписчики не знают друг друга: все сообщения идут через брокер

Роли не закреплены: плата может публиковать температуру и одновременно подписываться на команды (пунктирная стрелка на рисунке — плата №3 *получает* команды, а сообщения публикует кто-то другой).

Темы и шаблоны

Тема — это строка, разделённая косой чертой, как путь к файлу: `home/kitchen/temp`. Темы образуют дерево. Подписываться можно не только на точное имя, но и на шаблон: `+` заменяет *один* уровень, `#` — любое число уровней до конца.

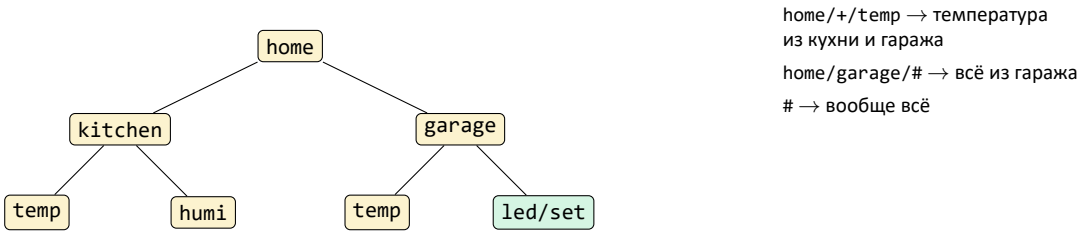


Рис. 18.2: Дерево тем и шаблоны подписки: + — один уровень, # — «всё остальное»

Хорошая структура тем — половина успеха проекта. Правила:

- начинайте от общего к частному: `дом/комната/устройство/параметр` (латиницей, например `home/room1/temp`);
- один топик — одно значение; для связанных значений используйте JSON-сообщение (см. ниже);
- отделяйте *состояние* (`.../led/state`) от *команды* (`.../led/set`): команда — просьба, состояние — факт;
- не начинайте тему с `/` и не используйте пробелы и кириллицу.

18.3 Надёжность доставки: QoS, retained и «завещание»

Сеть ненадёжна, поэтому MQTT предлагает три **уровня качества обслуживания** (QoS), которые выбирает отправитель:

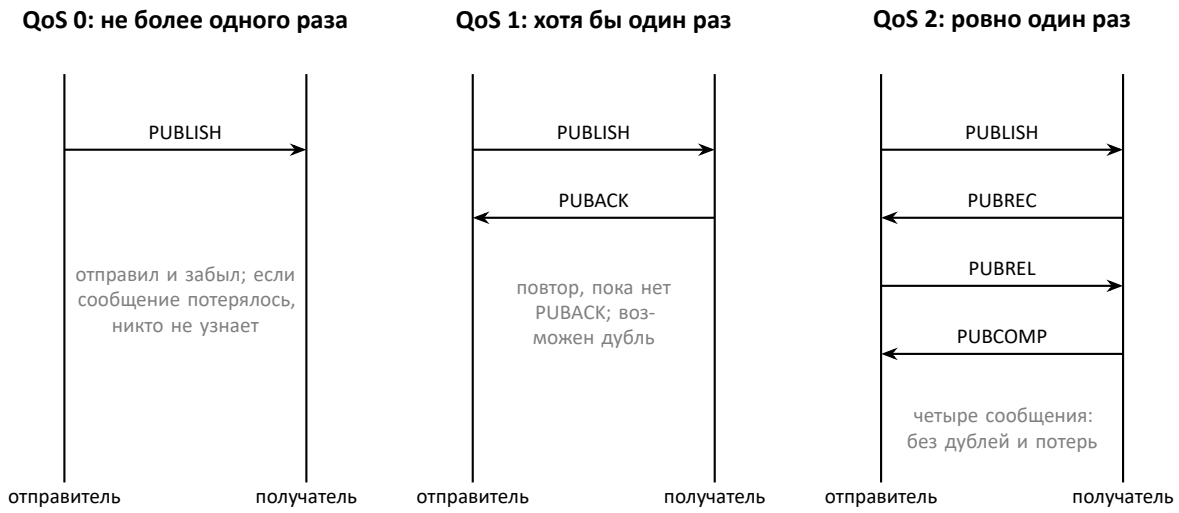


Рис. 18.3: Три уровня качества обслуживания MQTT

QoS	Гарантия	Когда использовать
0	не больше одного раза	частые показания датчика: потеря одного значения не страшна (по умолчанию)
1	хотя бы один раз	команды и важные события (дубликат безвреден: «включить» дважды = включить)
2	ровно один раз	счётчики, платежи; в микроконтроллерах используется редко

Таблица 18.1: Выбор QoS

Это интересно

Библиотека `umqtt.simple` поддерживает QoS 0 и 1, но не QoS 2. `adafruit_minimqtt` — QoS 0 и 1. Для учебных проектов этого достаточно. QoS описывает путь «плата ↔ брокер», а не «издатель → подписчик»: на каждом участке уровень выбирается отдельно.

Retained-сообщения

Подписчик получает только то, что опубликовано *после* подписки. Если датчик публикует раз в час, новый экран будет пустым до следующей публикации. Выход — флаг **retain**: брокер запоминает *последнее* сообщение темы и отдаёт его каждому новому подписчику сразу. Так хранят состояния («свет включён») и конфигурации.

«Завещание» (Last Will)

Как узнать, что плата отключилась, если она уже ничего не может сказать? При подключении плата вручает брокеру «завещание»: сообщение, которое брокер сам опубликует, если связь с платой оборвётся без нормального прощания (`DISCONNECT`). Типичная схема: `status = online` (retain) при подключении и завещание `status = offline` (retain).

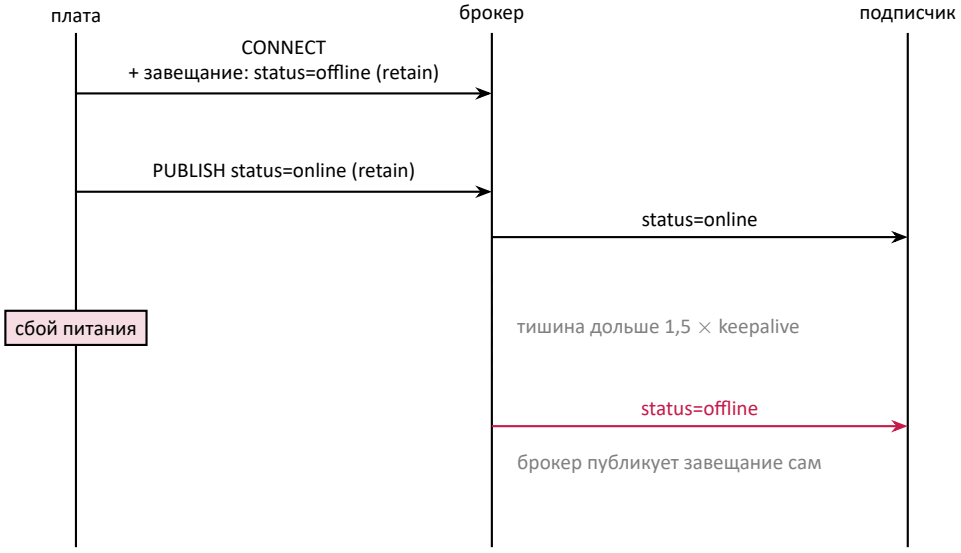


Рис. 18.4: Подключение с завещанием: брокер сообщает подписчикам об «исчезновении» платы

Время «молчания», после которого брокер считает клиента потерянным, определяется **keepalive**. Если клиент долго ничего не публикует, он обязан слать короткий пакет `PINGREQ` (в `umqtt.simple` это метод `ping()`); иначе брокер через $1{,}5 \times$ keepalive решит, что клиент умер.

18.4 Брокер Mosquitto

Eclipse Mosquitto — простой и легкий брокер с открытым кодом. Его можно запустить на ноутбуке, мини-ПК или Raspberry Pi (круглосуточно).

- **Linux, Raspberry Pi OS:** `sudo apt install mosquitto mosquitto-clients`.
- **Windows:** установщик с сайта `mosquitto.org`; запуск как службы.
- **Docker:** образ `eclipse-mosquitto`.

Начиная с версии 2.0, Mosquitto по умолчанию слушает только `localhost` и без настройки не пускает чужих. Минимальная конфигурация (`/etc/mosquitto/conf.d/pico.conf`) для домашней сети с паролем:

```
listener 1883
allow_anonymous false
password_file /etc/mosquitto/passwd
```

Создать пользователя: `sudo mosquitto_passwd -c /etc/mosquitto/passwd pico1` (спросит пароль), затем перезапустить службу. Для проверки есть клиенты командной строки:

```
mosquitto_sub -h 192.168.1.10 -u pico1 -P ПАРОЛЬ -t "home/#" -v
mosquitto_pub -h 192.168.1.10 -u pico1 -P ПАРОЛЬ -t home/pico1/led/set -m ON
```

Первая команда «слушает» все темы `home/` и печатает их вместе с именем темы (`-v`), вторая публикует команду. На телефоне те же действия выполняют приложения «MQTT Dash», «IoT MQTT Panel», «MQTT Explorer» (на компьютере) и подобные.

Осторожно

Публичные брокеры (`test.mosquitto.org`, `broker.hivemq.com` и др.) удобны для первого опыта без установки, но: темы *общие для всего мира* (любой подпишется на `#` и увидит ваши данные), любой может отправить команду на вашу плату, и вы не знаете, кто смотрит. Используйте длинный случайный префикс темы (`pico-x7f3a9c/...`), не публикуйте личные данные (адреса, расписание, фото) и никогда не управляйте с их помощью чем-то серьёзным. Для постоянных устройств — только свой брокер.

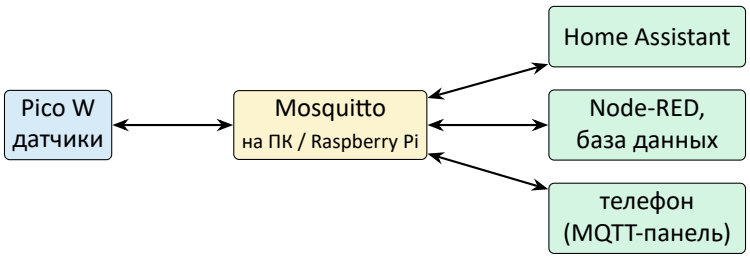


Рис. 18.5: Типичная система «умного дома»: брокер в центре, вокруг — устройства и программы

18.5 Формат пакета и почему MQTT экономен

MQTT придуман для медленных и слабых каналов (первые применения — нефтепроводы по спутниковой связи). Пакет `PUBLISH` состоит из очень короткого заголовка, имени темы и самих данных.

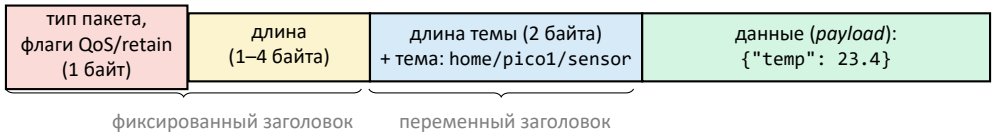


Рис. 18.6: Структура пакета `PUBLISH`: на служебные данные уходит всего несколько байт

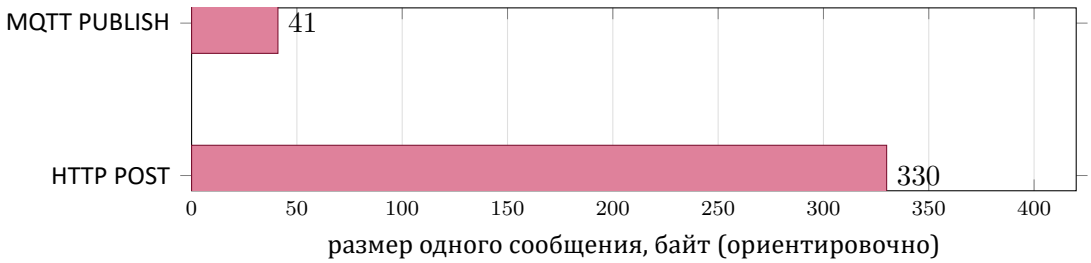
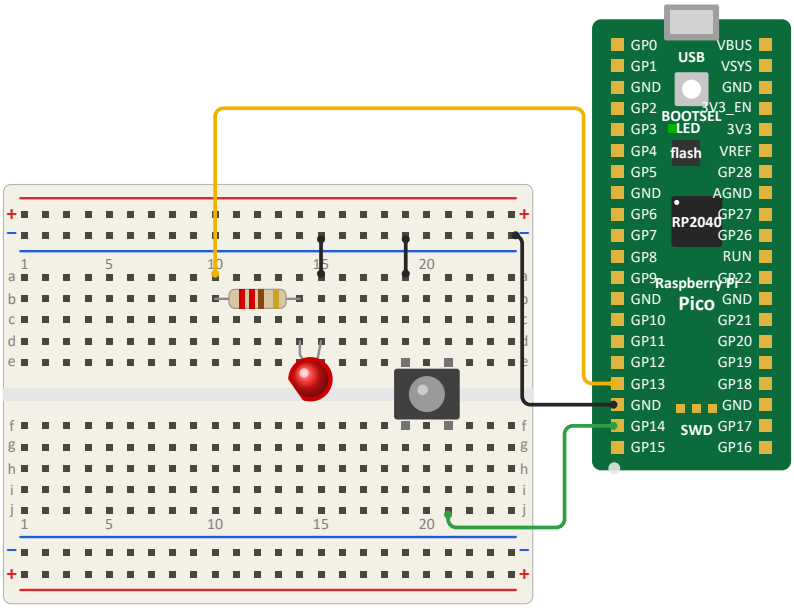


Рис. 18.7: Для одного и того же показания датчика MQTT тратит на порядок меньше байт, чем HTTP (оценка)

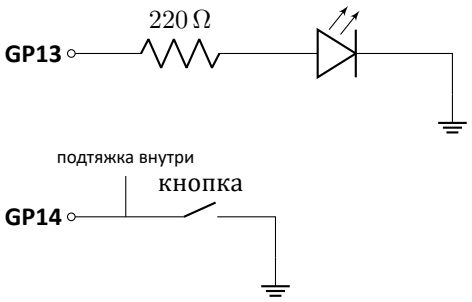
Кроме экономии трафика, соединение с брокером *постоянное*: не нужно каждый раз заново устанавливать TCP и TLS. Именно поэтому узел может отвечать на команду за доли секунды.

18.6 Практика: узел с датчиком и исполнителем

Подключим светодиод на **GP13** (получает команды) и кнопку на **GP14** (переключает светодиод вручную и сообщает об этом брокеру).



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 18.8: Светодиод (GP13) и кнопка (GP14) для MQTT-узла

Публикация данных

Начнём с самого простого: плата подключается к брокеру и раз в 10 секунд отправляет температуру. Данные кладём в формате JSON: так можно передать несколько значений в одном сообщении (`{"temp": 23.4, "rssi": -58}`), а подписчик разберёт их по ключам.
В **MicroPython** нужна библиотека `umqtt.simple` (ставится из Thonny: «Инструменты → Управление пакетами», или кодом `import mip; mip.install("umqtt.simple")`); в **CircuitPython** — `adafruit_minimqtt` (`circup install adafruit_minimqtt`). Параметры брокера (адрес, логин, пароль) хранятся в файле секретов:

Листинг 18.1: MicroPython: `secrets.py` с параметрами брокера

```
# Дополнение к secrets.py (рядом с main.py; в Git не попадает!)
SSID = "MyHomeWiFi"
PASSWORD = "my-secret-password"
COUNTRY = "RU"

MQTT_HOST = "192.168.1.10"      # адрес брокера Mosquitto в вашей сети
MQTT_PORT = 1883
MQTT_USER = "pico1"             # логин для брокера (или None, если без пароля)
MQTT_PASS = "long-random-password"
```

Листинг 18.2: CircuitPython: `settings.toml`

```
# settings.toml для CircuitPython (в Git не публиковать!)
CIRCUITPY_WIFI_SSID = "MyHomeWiFi"
CIRCUITPY_WIFI_PASSWORD = "my-secret-password"

MQTT_BROKER = "192.168.1.10"
MQTT_PORT = 1883
MQTT_USER = "pico1"
MQTT_PASS = "long-random-password"
```

Листинг 18.3: MicroPython: публикация показаний датчика

```
import json
import time
import ubinascii
import machine
import network
from umqtt.simple import MQTTClient       # mip.install("umqtt.simple")
import secrets

sensor = machine.ADC(4)                   # температура кристалла
# Уникальный идентификатор клиента: у двух клиентов с одним id брокер «выкидывает» первого
CLIENT_ID = b"pico-" + ubinascii.hexlify(machine.unique_id())
TOPIC = b"home/pico1/sensor"

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)


def temp_c():
    v = sensor.read_u16() * 3.3 / 65535
    return 27 - (v - 0.706) / 0.001721


client = MQTTClient(CLIENT_ID, secrets.MQTT_HOST, port=secrets.MQTT_PORT,
                    user=secrets.MQTT_USER, password=secrets.MQTT_PASS,
                    keepalive=60)
client.connect()
print("соединение с брокером установлено")

while True:
    payload = json.dumps({"temp": round(temp_c(), 1),
                          "rssi": wlan.status("rssi")})   # JSON-строка
    client.publish(TOPIC, payload)                        # QoS 0, без retain
    print("опубликовано:", payload)
    time.sleep(10)
```

Листинг 18.4: CircuitPython: то же

```
import json
import os
import time
import microcontroller
import socketpool
import wifi
import adafruit_minimqtt.adafruit_minimqtt as MQTT   # circup install adafruit_minimqtt

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
mqtt = MQTT.MQTT(
    broker=os.getenv("MQTT_BROKER"),
    port=os.getenv("MQTT_PORT"),
    username=os.getenv("MQTT_USER"),
    password=os.getenv("MQTT_PASS"),
    socket_pool=pool,
    keep_alive=60,
)
TOPIC = "home/pico1/sensor"

mqtt.connect()
print("соединение с брокером установлено")

while True:
    payload = json.dumps({"temp": round(microcontroller.cpu.temperature, 1),
                          "rssi": wifi.radio.ap_info.rssi})   # JSON-строка
    mqtt.publish(TOPIC, payload)                              # QoS 0, без retain
    print("опубликовано:", payload)
    time.sleep(10)
```

Откройте на компьютере `mosquitto_sub -t "home/#" -v`: раз в 10 секунд будет появляться строка вида `home/pico1/sensor {"temp": 23.4, "rssi": -58}`. Обратите внимание на идентификатор клиента: у каждого клиента на брокере он должен быть *уникальным*, иначе второй клиент «выбьет» первого, и они будут бесконечно выбивать друг друга. Мы строим идентификатор из уникального номера чипа `machine.unique_id()`.

Подписка на команды

Теперь научим плату *слушать*. Подписываемся на тему команд и обрабатываем сообщения в функции-обработчике (**колбэке**). Колбэк вызывает библиотека, когда пришло сообщение. В MicroPython аргументы колбэка — *байты* (`b"ON"`), в CircuitPython — строки (`"ON"`). Чтобы колбэк вообще вызывался, нужно регулярно проверять входящие: `check_msg()` в MicroPython (не блокирует, в отличие от `wait_msg()`) и `mqtt.loop()` в CircuitPython.

Листинг 18.5: MicroPython: светодиод по команде ON/OFF, кнопка — вручную

```
import time
import ubinascii
import machine
import network
from machine import Pin
from umqtt.simple import MQTTClient
import secrets

led = Pin(13, Pin.OUT)
button = Pin(14, Pin.IN, Pin.PULL_UP)
CLIENT_ID = b"pico-" + ubinascii.hexlify(machine.unique_id())
T_SET = b"home/pico1/led/set"             # сюда приходят команды: ON / OFF
T_STATE = b"home/pico1/led/state"         # сюда плата сообщает фактическое состояние

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
while not wlan.isconnected():
    time.sleep(0.5)


def on_message(topic, msg):
    """Вызывается библиотекой при каждом входящем сообщении (оба аргумента --- bytes)."""
    print("получено:", topic, msg)
    if topic == T_SET:
        if msg == b"ON":
            led.value(1)
        elif msg == b"OFF":
            led.value(0)
        client.publish(T_STATE, b"ON" if led.value() else b"OFF", retain=True)


client = MQTTClient(CLIENT_ID, secrets.MQTT_HOST, port=secrets.MQTT_PORT,
                    user=secrets.MQTT_USER, password=secrets.MQTT_PASS, keepalive=60)
client.set_callback(on_message)
client.connect()
client.subscribe(T_SET)                   # подписка на команды
client.publish(T_STATE, b"OFF", retain=True)
print("жду команды в", T_SET)

was_pressed = False
while True:
    client.check_msg()                    # неблокирующая проверка входящих (в отличие от
    ↪  wait_msg)
    pressed = button.value() == 0
    if pressed and not was_pressed:       # кнопка: переключаем светодиод локально
        led.toggle()
        client.publish(T_STATE, b"ON" if led.value() else b"OFF", retain=True)
    was_pressed = pressed
    time.sleep_ms(50)
```

Листинг 18.6: CircuitPython: то же

```python
import os
import time
import board
import digitalio
import socketpool
import wifi
import adafruit_minimqtt.adafruit_minimqtt as MQTT

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
button = digitalio.DigitalInOut(board.GP14)
button.switch_to_input(pull=digitalio.Pull.UP)

T_SET = "home/pico1/led/set"              # сюда приходят команды: ON / OFF
T_STATE = "home/pico1/led/state"          # сюда плата сообщает фактическое состояние

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))


def on_connect(client, userdata, flags, rc):
    print("подключились к брокеру")
    client.subscribe(T_SET)               # подписку делаем после каждого (пере)подключения


def on_message(client, topic, message):
    """Вызывается при входящем сообщении; message --- уже строка."""
    print("получено:", topic, message)
    if topic == T_SET:
        if message == "ON":
            led.value = True
        elif message == "OFF":
            led.value = False
        client.publish(T_STATE, "ON" if led.value else "OFF", retain=True)


pool = socketpool.SocketPool(wifi.radio)
mqtt = MQTT.MQTT(broker=os.getenv("MQTT_BROKER"), port=os.getenv("MQTT_PORT"),
                 username=os.getenv("MQTT_USER"), password=os.getenv("MQTT_PASS"),
                 socket_pool=pool, keep_alive=60)
mqtt.on_connect = on_connect
mqtt.on_message = on_message
mqtt.connect()
mqtt.publish(T_STATE, "OFF", retain=True)

was_pressed = False
while True:
    mqtt.loop(timeout=0.05)               # принять входящие и поддержать связь (пинг)
    pressed = not button.value
    if pressed and not was_pressed:       # кнопка: переключаем светодиод локально
        led.value = not led.value
        mqtt.publish(T_STATE, "ON" if led.value else "OFF", retain=True)
    was_pressed = pressed
```

Проверьте: `mosquitto_pub -t home/pico1/led/set -m ON` включит светодиод, а в теме `.../state`

появится актуальное состояние (оно `retain`, так что новый подписчик сразу узнает его). Нажатие кнопки меняет светодиод и *тоже* отправляет состояние — интерфейс на телефоне останется синхронным.

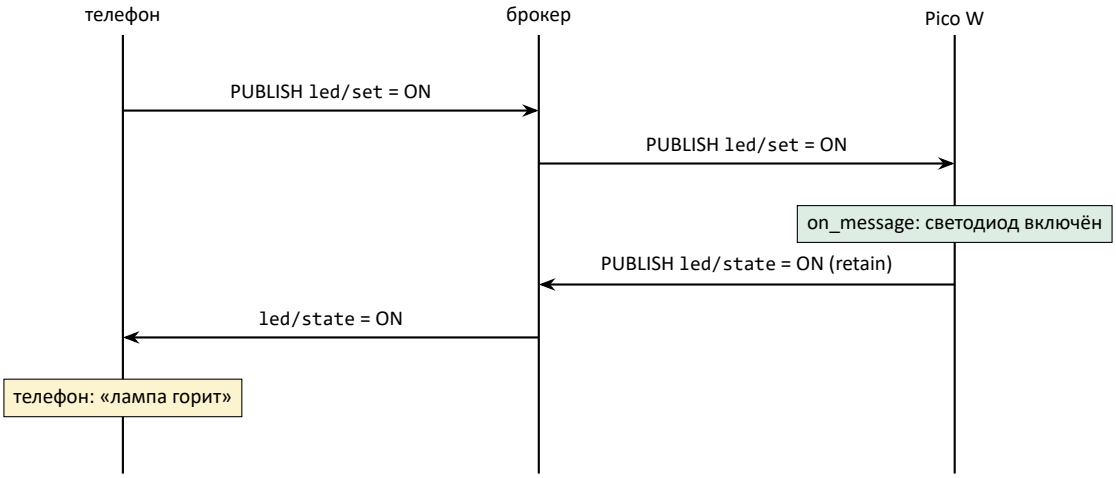


Рис. 18.9: Путь команды: телефон → брокер → плата и подтверждение обратно

18.7 Учимся программировать: функции обратного вызова

Раньше мы вызывали функции сами. Здесь впервые мы *передаём функцию библиотеке*, чтобы та вызвала её, когда произойдёт событие. Такая функция называется **колбэком** (callback, «перезвони мне»). Это очень общая идея: так работают кнопки в приложениях, прерывания (урок 4), таймеры, веб-серверы. Строка `client.set_callback(on_message)` не вызывает `on_message`: скобок нет! Мы отдаём функцию как значение, а библиотека вызовет её сама, когда придёт сообщение.

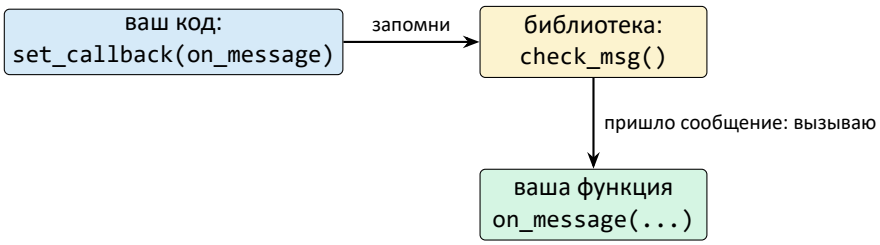


Рис. 18.10: Колбэк: мы регистрируем функцию один раз, а вызывает её библиотека

18.8 Надёжный узел

Узел, который должен работать месяцами, обязан переживать сбои: роутер перезагрузился, брокер перезапущен, пропал интернет. Для этого нужны: проверка Wi-Fi, переподключение к брокеру с повторной подпиской (подписки не сохраняются при новом подключении!), периодический `ping`, завещание и сообщение «online». Всю схему удобно оформить как цикл с «защитным» блоком `try/except`.

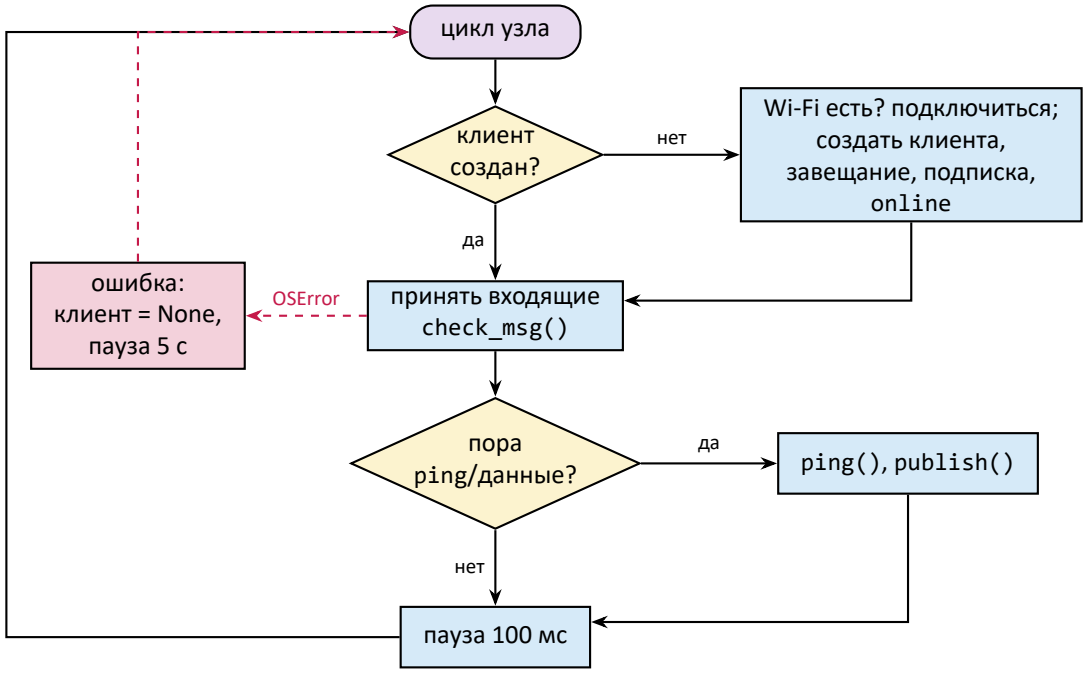


Рис. 18.11: Алгоритм надёжного MQTT-узла: любой сбой ведёт к пересозданию клиента

Листинг 18.7: MicroPython: узел с завещанием, пингом и переподключением

```
import json
import time
import ubinascii
import machine
import network
from machine import Pin
from umqtt.simple import MQTTClient
import secrets

led = Pin(13, Pin.OUT)
sensor = machine.ADC(4)
CLIENT_ID = b"pico-" + ubinascii.hexlify(machine.unique_id())
T_STATUS = b"home/pico1/status"          # online / offline
T_DATA = b"home/pico1/sensor"
T_SET = b"home/pico1/led/set"
KEEPALIVE = 60                           # секунд

wlan = network.WLAN(network.STA_IF)
client = None


def temp_c():
    v = sensor.read_u16() * 3.3 / 65535
    return 27 - (v - 0.706) / 0.001721


def on_message(topic, msg):
    if topic == T_SET:
        led.value(1 if msg == b"ON" else 0)


def wifi_up():
    if wlan.isconnected():
```

```
        return True
    wlan.active(True)
    wlan.connect(secrets.SSID, secrets.PASSWORD)
    for _ in range(40):                  # ждём до 20 секунд
        if wlan.isconnected():
            return True
        time.sleep_ms(500)
    return False


def mqtt_up():
    """Создаёт клиента: завещание, подписка, сообщение «online»."""
    c = MQTTClient(CLIENT_ID, secrets.MQTT_HOST, port=secrets.MQTT_PORT,
                   user=secrets.MQTT_USER, password=secrets.MQTT_PASS,
                   keepalive=KEEPALIVE)
    c.set_callback(on_message)
    c.set_last_will(T_STATUS, b"offline", retain=True)    # «завещание» для брокера
    c.connect()
    c.subscribe(T_SET)
    c.publish(T_STATUS, b"online", retain=True)
    return c


t_ping = t_pub = time.ticks_ms()
while True:
    try:
        if client is None:
            if not wifi_up():
                raise OSError("нет Wi-Fi")
            client = mqtt_up()
            print("MQTT: подключено")
        client.check_msg()                                # входящие команды
        now = time.ticks_ms()
        if time.ticks_diff(now, t_ping) >= KEEPALIVE * 500:   # вдвое чаще keepalive
            client.ping()                                 # «я жив»
            t_ping = now
        if time.ticks_diff(now, t_pub) >= 10_000:
            client.publish(T_DATA, json.dumps({"temp": round(temp_c(), 1)}))
            t_pub = now
        time.sleep_ms(100)
    except OSError as e:                                  # любой сетевой сбой
        print("сбой:", e, "--- переподключение через 5 с")
        client = None                                     # создадим клиента заново
        time.sleep(5)
```

Листинг 18.8: CircuitPython: то же

```
import json
import os
import time
import board
import digitalio
import microcontroller
import socketpool
import wifi
import adafruit_minimqtt.adafruit_minimqtt as MQTT

led = digitalio.DigitalInOut(board.GP13)
```

```
led.direction = digitalio.Direction.OUTPUT
T_STATUS = "home/pico1/status"            # online / offline
T_DATA = "home/pico1/sensor"
T_SET = "home/pico1/led/set"


def on_connect(client, userdata, flags, rc):
    client.subscribe(T_SET)               # подписка после каждого (пере)подключения
    client.publish(T_STATUS, "online", retain=True)


def on_message(client, topic, message):
    if topic == T_SET:
        led.value = message == "ON"


def build_client():
    pool = socketpool.SocketPool(wifi.radio)
    c = MQTT.MQTT(broker=os.getenv("MQTT_BROKER"), port=os.getenv("MQTT_PORT"),
                  username=os.getenv("MQTT_USER"), password=os.getenv("MQTT_PASS"),
                  socket_pool=pool, keep_alive=60)
    c.will_set(T_STATUS, "offline", retain=True)          # «завещание» для брокера
    c.on_connect = on_connect
    c.on_message = on_message
    return c


def wifi_up():
    if wifi.radio.connected:
        return True
    try:
        wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                           os.getenv("CIRCUITPY_WIFI_PASSWORD"), timeout=15)
        return True
    except ConnectionError:
        return False


mqtt = None
t_pub = time.monotonic()
while True:
    try:
        if mqtt is None:
            if not wifi_up():
                raise OSError("нет Wi-Fi")
            mqtt = build_client()
            mqtt.connect()
            print("MQTT: подключено")
        mqtt.loop(timeout=0.2)            # входящие команды и автоматический пинг
        if time.monotonic() - t_pub >= 10:
            mqtt.publish(T_DATA, json.dumps({"temp": round(microcontroller.cpu.temperature,
            ↪  1)}))
            t_pub = time.monotonic()
    except (OSError, MQTT.MMQTTException) as e:          # любой сетевой сбой
        print("сбой:", e, "--- переподключение через 5 с")
        mqtt = None                                      # создадим клиента заново
        time.sleep(5)
```

Чтобы проверить надёжность: отключите роутер на минуту и включите снова; перезапустите

службу брокера (`sudo systemctl restart mosquitto`). В консоли должно появиться сообщение о сбое, а потом о восстановлении, а в теме `home/pico1/status` — смена `offline` → `online`.

Совет

Проверяйте *не только* подключение, но и то, что ваши данные доходят: подпишитесь с компьютера на `home/#`. Если на экране пусто, ищите проблему по цепочке: Wi-Fi → адрес брокера → порт 1883 в брандмауэре → логин и пароль.

18.9 Home Assistant: автоматическое обнаружение

Home Assistant — популярная бесплатная система умного дома. Она умеет работать с MQTT и находить новые устройства сама (**MQTT discovery**). Для этого устройство один раз публикует в специальную тему `homeassistant/<тип>/<id>/config` сообщение (retain) с описанием: название, из какой темы брать значение, единицы измерения. Home Assistant автоматически создаёт датчик, которому достаточно показаний в обычной теме.

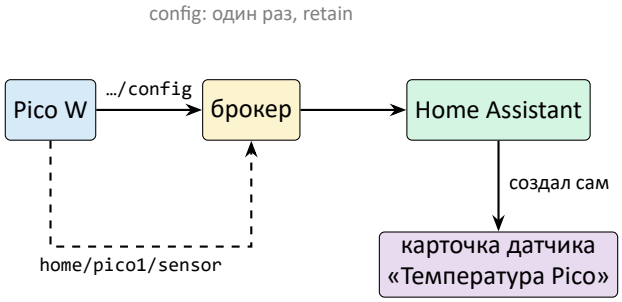


Рис. 18.12: Обнаружение устройства: конфигурация публикуется один раз, дальше идут обычные показания

Листинг 18.9: MicroPython: публикация конфигурации датчика температуры

```
import json
import time
import ubinascii
import machine
import network
from umqtt.simple import MQTTClient
import secrets

# Home Assistant ищет конфигурации в топиках homeassistant/<тип>/<id>/config
DISCOVERY = b"homeassistant/sensor/pico1_temp/config"
config = {
    "name": "Температура Pico",
    "unique_id": "pico1_temp",                      # уникальный, не менять!
    "state_topic": "home/pico1/sensor",
    "value_template": "{{ value_json.temp }}",      # достаём поле temp из JSON
    "unit_of_measurement": "°C",
    "device_class": "temperature",
    "availability_topic": "home/pico1/status",      # online/offline из «завещания»
    "device": {"identifiers": ["pico1"], "name": "Pico 1", "model": "Raspberry Pi Pico W"},
}

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(secrets.SSID, secrets.PASSWORD)
```

```
while not wlan.isconnected():
    time.sleep(0.5)

client = MQTTClient(b"pico-" + ubinascii.hexlify(machine.unique_id()), secrets.MQTT_HOST,
                    port=secrets.MQTT_PORT, user=secrets.MQTT_USER, password=secrets.MQTT_PASS)
client.connect()
# retain=True: Home Assistant увидит конфигурацию, даже если запустится позже платы
client.publish(DISCOVERY, json.dumps(config), retain=True)
print("конфигурация опубликована")
client.disconnect()
```

Листинг 18.10: CircuitPython: то же

```
import json
import os
import socketpool
import wifi
import adafruit_minimqtt.adafruit_minimqtt as MQTT

# Home Assistant ищет конфигурации в топиках homeassistant/<тип>/<id>/config
DISCOVERY = "homeassistant/sensor/pico1_temp/config"
config = {
    "name": "Температура Pico",
    "unique_id": "pico1_temp",                          # уникальный, не менять!
    "state_topic": "home/pico1/sensor",
    "value_template": "{{ value_json.temp }}",          # достаём поле temp из JSON
    "unit_of_measurement": "°C",
    "device_class": "temperature",
    "availability_topic": "home/pico1/status",          # online/offline из «завещания»
    "device": {"identifiers": ["pico1"], "name": "Pico 1", "model": "Raspberry Pi Pico W"},
}

if not wifi.radio.connected:
    wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
                       os.getenv("CIRCUITPY_WIFI_PASSWORD"))

pool = socketpool.SocketPool(wifi.radio)
mqtt = MQTT.MQTT(broker=os.getenv("MQTT_BROKER"), port=os.getenv("MQTT_PORT"),
                 username=os.getenv("MQTT_USER"), password=os.getenv("MQTT_PASS"),
                 socket_pool=pool)
mqtt.connect()
# retain=True: Home Assistant увидит конфигурацию, даже если запустится позже платы
mqtt.publish(DISCOVERY, json.dumps(config), retain=True)
print("конфигурация опубликована")
mqtt.disconnect()
```

В Home Assistant в настройках интеграции MQTT укажите адрес вашего брокера. После запуска примера в разделе «Устройства» появится «Pico 1» с датчиком температуры. Поле `availability_topic` связывает датчик с нашим «завещанием»: если плата пропала, Home Assistant покажет датчик как «недоступен». Конкретные поля могут немного отличаться в разных версиях Home Assistant — сверяйтесь с документацией по MQTT-интеграции.

18.10 Безопасность

- **Логин и пароль на брокере обязательны** (`allow_anonymous false`); у каждого устройства — свой пользователь, чтобы можно было отозвать доступ.

- **Порт 1883 не шифрует** данные и пароль, как и HTTP из урока 17. Внутри домашней сети это обычно допустимо. Через интернет — только TLS (порт **8883**): в **MicroPython** параметр `ssl` конструктора `MQTTClient`, в **CircuitPython** `is_ssl=True` и `ssl_context`. TLS требует памяти и сертификатов; сверьтесь с документацией вашей прошивки.
- **Права доступа (ACL)** в Mosquitto позволяют ограничить, кто в какие темы может писать.
- **Не публикуйте секреты**: пароли, ключи Wi-Fi, личные данные не должны попадать в темы и сообщения. Не публикуйте адрес и пароль брокера в Git.
- **Не открывайте брокер в интернет** без TLS, паролей и ACL: сканеры находят открытые брокеры за минуты.



Рис. 18.13: Защита строят слоями: каждый уровень закрывает свои угрозы

Если не работает

- **`OSError: [Errno 104] ECONNRESET` или `MQTTException: 5`** — брокер отверг соединение: неверный логин или пароль (код 5), либо брокер доступен только с `localhost` (см. конфигурацию Mosquitto 2.x).
- **Постоянно «выбрасывает» то одну плату, то другую** — одинаковый `client_id` у нескольких клиентов.
- **Плата публикует, но подписчик ничего не видит** — опечатка в теме (регистр важен!), подписка раньше, чем публикация без retain, брандмауэр на компьютере с брокером закрыл порт 1883.
- **После переподключения плата перестала принимать команды** — забыли подписаться заново после `connect()`: подписки хранятся только на время соединения.
- **Колбэк не вызывается** — не вызывается `check_msg()` / `loop()`, либо программа надолго засыпает в `time.sleep()`.
- **`ImportError: no module named umqtt.simple`** — пакет не установлен: `mip.install("umqtt.simple")`.
- **Узел «молчит» после нескольких часов работы** — не вызывается `ping()` при редкой публикации, брокер отключил клиента по keepalive.

Итоги урока

- MQTT: издатели публикуют в темы, подписчики получают сообщения от посредника — брокера; участникам не нужно знать друг друга.
- Темы образуют иерархию; шаблоны + и # позволяют подписываться на группы.
- QoS 0/1/2 — уровни гарантии доставки; `retain` хранит последнее значение; «завещание» сообщает о неожиданном отключении.

- В **MicroPython** — `umqtt.simple`: `publish`, `subscribe`, `check_msg`, `ping`; в **CircuitPython** — `adafruit_minimqtt`: `publish`, `subscribe`, `loop`.
- Надёжный узел сам переподключается, заново подписывается и сообщает свой статус; данные удобно отправлять JSON-строкой.
- Безопасность: пароли, TLS (8883) и ACL; публичные брокеры — только для опытов без личных данных.

Проверь себя

1. В чём разница между моделью «клиент–сервер» (HTTP) и «публикация–подписка» (MQTT)? Какую роль играет брокер?
2. Что означают шаблоны `+` и `#`? На какие темы подпишется `home/+/temp`?
3. Чем отличаются QoS 0 и QoS 1? Когда нужен флаг `retain`?
4. Для чего служит «завещание» и как оно связано с keepalive?
5. Почему после переподключения нужно заново подписываться и зачем нужен уникальный `client_id`?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Установите Mosquitto (или используйте брокер преподавателя) и проверьте обмен командами `mosquitto_pub` и `mosquitto_sub` между двумя окнами терминала.
2. ★ Запустите публикацию температуры и найдите свои сообщения в `mosquitto_sub`. Измените период публикации и тему.
3. ★★ Добавьте в JSON-сообщение уровень Wi-Fi и время работы (`uptime`); подпишитесь на тему с телефона и прочитайте значения.
4. ★★ Сделайте так, чтобы кнопка отправляла отдельное событие `home/pico1/button` со значением `pressed`, а подписчик (другой Pico W) включал по нему светодиод.
5. ★★★ Добавьте второй исполнитель (реле или второй светодиод) с отдельными темами `set/state`; проверьте, что состояние восстанавливается после перезагрузки платы (retain).
6. ★★★★ Добавьте в узел Home Assistant discovery для светодиода (`light` или `switch`) и датчика; проверьте, как выглядит устройство при отключении платы (завещание), и поведение при перезапуске брокера.

Урок 19. Bluetooth Low Energy

Цель	понять, как устроен Bluetooth Low Energy (BLE), и научить Pico W общаться со смартфоном: передавать показания датчика и принимать команды
Опирается на	урок 16 (радиочип CYW43439), урок 2 (светодиод), урок 18 (колбэки и события), урок 13 (асинхронные задачи)
Результат	плата рекламирует себя; термометр отправляет температуру в приложение; светодиод включается с телефона; плата работает как «Bluetooth-терминал»
Понадобится	Pico W или Pico 2 W, светодиод и резистор 220 Ω, смартфон с приложениями nRF Connect и Serial Bluetooth Terminal

19.1 Идея урока коротко

Wi-Fi требует роутера, пароля и заметного расхода энергии. А как подключить плату к телефону проще и экономнее: без сети, прямо «из рук в руки»? Для этого придуман **Bluetooth Low Energy** (BLE, «Bluetooth с низким энергопотреблением»). Именно по BLE общаются фитнес-браслеты, беспроводные датчики, метки поиска вещей и умные лампы. В этом уроке плата будет *рекламировать* себя, телефон найдёт её, подключится, прочитает температуру и включит светодиод.

Осторожно

Как и Wi-Fi, BLE работает только на **Pico W** и **Pico 2 W**: радиочип CYW43439 умеет и Wi-Fi, и Bluetooth. У обычных Pico и Pico 2 радио нет. Поддержка BLE в прошивках разных годов отличается: в MicroPython она давно есть, в CircuitPython на Pico W появилась позже, чем Wi-Fi (нужна свежая версия 9.x; проверьте, что команда `import _bleio` выполняется без ошибки).

19.2 BLE и «классический» Bluetooth

Название «Bluetooth» объединяет два *разных* протокола, которые не понимают друг друга: **Classic** (наушники, колонки, передача аудио) и **Low Energy** (датчики и управление). Радиочип платы поддерживает оба, но MicroPython и CircuitPython на Pico W дают доступ только к BLE.

	Bluetooth Classic	Bluetooth Low Energy
Назначение	потоки: звук, файлы	редкие короткие сообщения, датчики
Скорость	до ~2 Мбит/с	до ~1 Мбит/с (на практике 5–100 кБ/с)
Потребление	заметное	в десятки раз меньше
Подключение	долгое (секунды)	быстрое (доли секунды)
Топология	«точка—точка»	реклама и соединения; mesh в расширениях
Типичные устройства	наушники, колонки	браслеты, метки, датчики, лампы

Таблица 19.1: Сравнение двух «Bluetooth»

Радиоканалы BLE

BLE использует тот же диапазон 2,4 GHz, что и Wi-Fi, но делит его на 40 каналов по 2 MHz. Три из них (37, 38, 39) — **рекламные**: они расставлены так, чтобы не пересекаться с самыми популярными каналами Wi-Fi (1, 6, 11). Остальные 37 — каналы данных: во время соединения устройства быстро «прыгают» по ним (скачкообразная перестройка частоты), что защищает от помех.

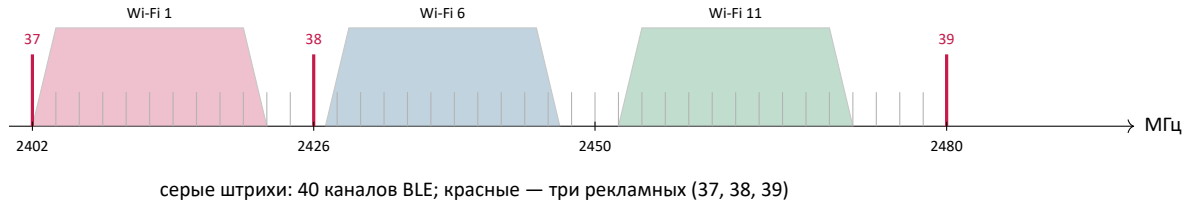


Рис. 19.1: Рекламные каналы BLE расположены в «окнах» между самыми популярными каналами Wi-Fi

19.3 Стек BLE: GAP и GATT

Протокол BLE сложный, но программисту нужны только две верхние ступени:



Рис. 19.2: Слои BLE: GAP отвечает за «знакомство», GATT — за «разговор»

- **GAP** (Generic Access Profile) — «знакомство»: как устройство объявляет о себе (реклама), как другое его находит (сканирование) и как они устанавливают соединение.
- **GATT** (Generic Attribute Profile) — «разговор»: как устроены данные устройства и как их читать и записывать.

Роли

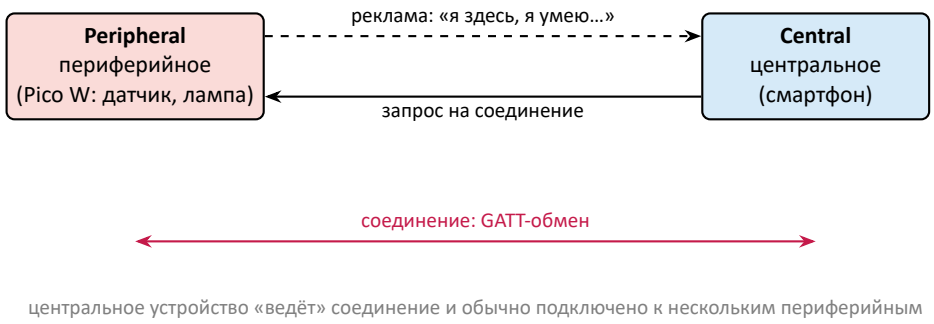


Рис. 19.3: Роли в BLE: периферийное устройство рекламируется, центральное — находит и подключается

Плата чаще всего — **периферийное** устройство (peripheral, «сервер данных»): рекламирует себя и ждёт подключения. Телефон чаще **центральное** (central, «клиент»): сканирует эфир, выбирает устройство, подключается и читает данные. Но плата может быть и центральным (например, читать данные с другого датчика) — пример ниже. Есть ещё две «односторонние» роли: *broadcaster* (только рассылает рекламу, например метка-маяк) и *observer* (только слушает рекламу).

19.4 Реклама: как устройство «кричит» о себе

Периферийное устройство периодически (раз в десятки–сотни миллисекунд) отправляет короткий рекламный пакет по каналам 37, 38, 39. Вот что внутри:



Рис. 19.4: Рекламный пакет: последовательность структур «длина — тип — данные»

В пакете может быть имя, список сервисов, производитель, уровень мощности. Если нужных данных много, их делят на две части: *рекламный пакет* и *ответ на сканирование* (scan response); центральное устройство запрашивает второй при желании (*active scan*).

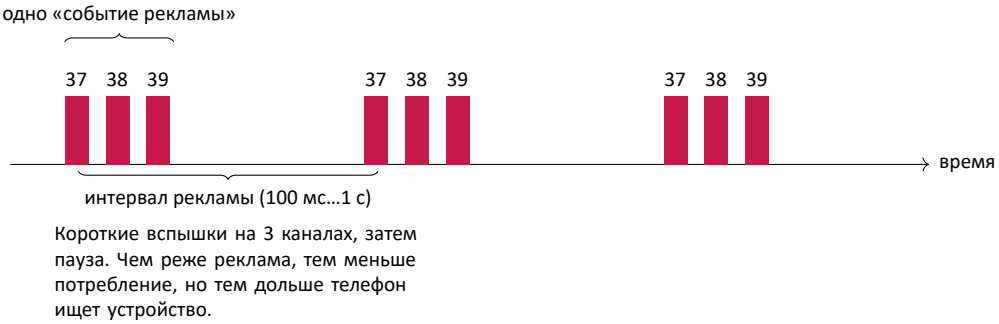


Рис. 19.5: Рекламные события: три коротких сигнала, затем пауза

Листинг 19.1: MicroPython: реклама имени и обработка подключения (низкоуровневый API)

```
import bluetooth
from micropython import const
from machine import Pin

_IRQ_CENTRAL_CONNECT = const(1)        # к нам подключился телефон
_IRQ_CENTRAL_DISCONNECT = const(2)     # телефон отключился

led = Pin("LED", Pin.OUT)              # горит, пока есть подключение


def adv_payload(name):
    """Рекламный пакет из двух AD-структур: флаги и полное имя."""
    flags = b"\x02\x01\x06"            # длина 2, тип 0x01 (флаги), 0x06: только BLE,
    ↪  обнаруживаемое
    nm = name.encode()
    return flags + bytes((len(nm) + 1, 0x09)) + nm    # длина, тип 0x09 (имя), имя


def irq(event, data):
    """Обработчик событий BLE; вызывается системой, поэтому должен быть быстрым."""
    if event == _IRQ_CENTRAL_CONNECT:
        conn_handle, addr_type, addr = data
        print("подключился", bytes(addr).hex())
        led.on()
```

```
    elif event == _IRQ_CENTRAL_DISCONNECT:
        print("отключился, снова рекламируем")
        led.off()
        advertise()


def advertise():
    ble.gap_advertise(100_000, adv_data=adv_payload("PicoW"))   # интервал 100 мс


ble = bluetooth.BLE()
ble.active(True)
ble.irq(irq)
advertise()
print("Реклама запущена: ищите устройство PicoW в nRF Connect")
```

Запустите программу и откройте на телефоне приложение **nRF Connect for Mobile** (бесплатное, от Nordic Semiconductor, есть для Android и iOS). Нажмите «Scan»: в списке появится устройство `PicoW` с уровнем сигнала. Нажмите «Connect» — светодиод на плате загорится, в консоли появится адрес телефона. Телефон, как правило, показывает не настоящий адрес, а случайный: это защита приватности.

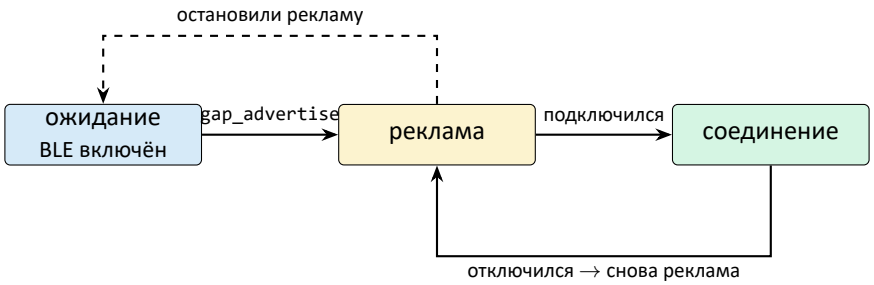


Рис. 19.6: Состояния периферийного устройства

19.5 GATT: как устроены данные

После соединения данные устройства устроены иерархически, как папки и файлы:

- **Профиль** — набор сервисов, описывающий тип устройства (термометр, пульсометр).
- **Сервис** (service) — группа связанных данных (например, «окружающая среда»).
- **Характеристика** (characteristic) — одно значение (например, температура) вместе с *свойствами*: читать (read), писать (write), получать уведомления (notify).
- **Дескрипторы** — дополнительные сведения о характеристике; самый важный — CCCD, «переключатель уведомлений»: клиент записывает в него 1, чтобы подписаться.

У каждого элемента есть **UUID** — уникальный идентификатор. Стандартные сервисы и характеристики имеют короткие 16-битные номера (0x181A — «окружающая среда», 0x2A6E — «температура»), а для своих данных придумывают случайный 128-битный UUID.

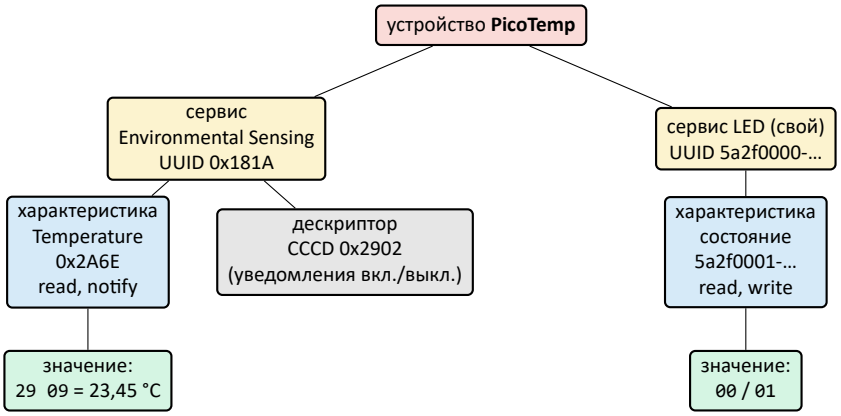


Рис. 19.7: Дерево данных GATT: устройство → сервис → характеристика → значение

Как закодирована температура

По стандарту характеристика «Temperature» (0x2A6E) хранит температуру как **знаковое 16-битное целое в сотых долях градуса**, в порядке байтов *little-endian* (младший байт первым). Для 23,45 °C число равно 2345; в шестнадцатеричной записи $2345 = \texttt{0x0929}$, значит, в пакет уйдут байты `29 09`. Любой стандартный клиент (в том числе nRF Connect) распознаёт такую характеристику и сам делит число на 100.

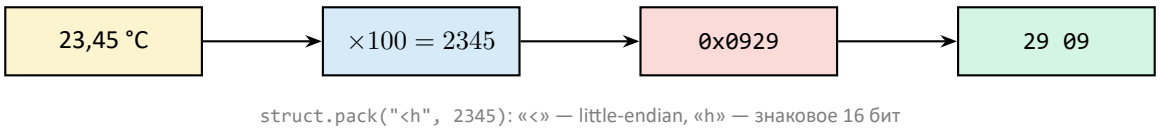


Рис. 19.8: Путь значения температуры от градусов до байтов характеристики

19.6 Чтение, запись и уведомления

С характеристиками можно делать три операции:

- **read** — центральное устройство *спрашивает* текущее значение (по запросу).
- **write** — центральное *записывает* значение (команда на устройство).
- **notify** — периферия *сама сообщает* об изменении, когда клиент заранее подписался. Это экономнее, чем постоянно опрашивать.

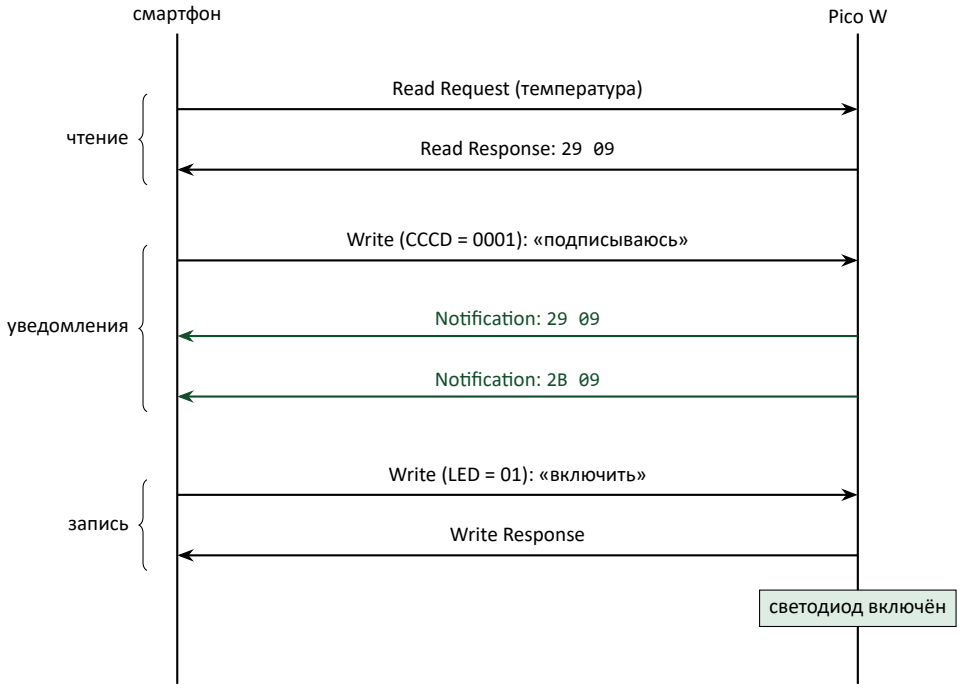


Рис. 19.9: Три операции GATT на примере: чтение температуры, подписка на уведомления и запись команды

19.7 MicroPython: низкоуровневый API и aioble

В MicroPython Bluetooth работает через модуль `bluetooth`. Это «низкоуровневый» интерфейс: вы сами регистрируете сервисы (`gatts_register_services`), запускаете рекламу (`gap_advertise`) и обрабатываете события в одной функции `irq(event, data)`. События — числа (1 — подключение, 2 — отключение, 3 — запись в характеристику и т. д.). Такой код надёжен и экономит память, но быстро становится громоздким.

Поэтому на практике пользуются библиотекой **aioble**: она построена поверх `bluetooth` и `asyncio` (урок 13). Вместо таблицы событий вы пишете обычные последовательные функции с `await`: «жди подключения», «жди записи». Установка: `mip.install("aioble")` (в пакетах micropython-lib; в некоторых прошивках есть в комплекте).

	`bluetooth` (IRQ)	`aioble`
Стиль	события и обработчик `irq()`	`async/await`, как обычный код
Память и скорость	минимальные	немного больше
Сложность	высокая: десятки констант	ниже: понятные классы
Рекомендация	простые примеры, тонкая настройка	большинство проектов

Таблица 19.2: Два способа работать с BLE в MicroPython

19.8 Практика: термометр

Реализуем стандартный сервис «Environmental Sensing» с характеристикой «Temperature»: любое приложение BLE увидит его как термометр. Температуру берём из встроенного датчика чипа; можно подставить любой датчик из урока 10.

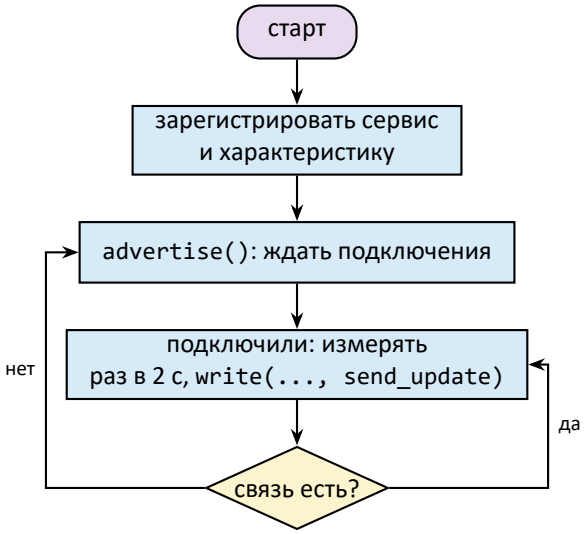


Рис. 19.10: Алгоритм BLE-термометра

Листинг 19.2: MicroPython: термометр на `aioble` (две асинхронные задачи)

```
import asyncio
import struct
import machine
import aioble                          # mip.install("aioble")
import bluetooth

sensor = machine.ADC(4)

_ENV_SENSING = bluetooth.UUID(0x181A)      # стандартный сервис «Environmental Sensing»
_TEMPERATURE = bluetooth.UUID(0x2A6E)      # стандартная характеристика «Temperature»
_APPEARANCE_THERMOMETER = 0x0300           # значок «термометр» в телефоне

service = aioble.Service(_ENV_SENSING)
temp_char = aioble.Characteristic(service, _TEMPERATURE, read=True, notify=True)
aioble.register_services(service)


def temp_c():
    v = sensor.read_u16() * 3.3 / 65535
    return 27 - (v - 0.706) / 0.001721


def encode(t):
    """Температура как знаковое 16-битное целое в сотых долях градуса, little-endian."""
    return struct.pack("<h", round(t * 100))


async def sensor_task():
    """Раз в 2 с обновляет значение; подписанные клиенты получают уведомление."""
    while True:
        temp_char.write(encode(temp_c()), send_update=True)
        await asyncio.sleep(2)


async def peripheral_task():
    """Рекламируемся, пока никто не подключился; после отключения --- снова."""
    while True:
```

```
        async with await aioble.advertise(
                250_000, name="PicoTemp", services=[_ENV_SENSING],
                appearance=_APPEARANCE_THERMOMETER) as connection:
            print("подключился", connection.device)
            await connection.disconnected(timeout_ms=None)
            print("отключился")


async def main():
    await asyncio.gather(sensor_task(), peripheral_task())


asyncio.run(main())
```

Листинг 19.3: CircuitPython: термометр на `adafruit_ble`

```
import time
import microcontroller
from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services import Service
from adafruit_ble.uuid import StandardUUID
from adafruit_ble.characteristics import Characteristic
from adafruit_ble.characteristics.int import Int16Characteristic


class EnvService(Service):
    """Стандартный сервис Environmental Sensing (0x181A) с температурой (0x2A6E)."""
    uuid = StandardUUID(0x181A)
    temperature = Int16Characteristic(
        uuid=StandardUUID(0x2A6E),
        properties=Characteristic.READ | Characteristic.NOTIFY,
    )


ble = BLERadio()
ble.name = "PicoTemp"
env = EnvService()
adv = ProvideServicesAdvertisement(env)

while True:
    print("реклама...")
    ble.start_advertising(adv)
    while not ble.connected:
        pass                                    # ждём подключения
    print("подключился")
    while ble.connected:
        # запись нового значения сама отправляет уведомление подписанным клиентам
        env.temperature = round(microcontroller.cpu.temperature * 100)   # сотые доли °C
        time.sleep(2)
    print("отключился")
```

В MicroPython две задачи работают одновременно: одна обновляет значение, другая рекламирует и ждёт подключений. Вызов `write(..., send_update=True)` не только меняет значение, но и отправляет уведомления подписанным клиентам (в старых версиях `aioble` уведомления отправляются отдельным вызовом `notify(connection, data)`; сверьтесь с версией). В CircuitPython присваивание `env.temperature = ...` делает то же самое.

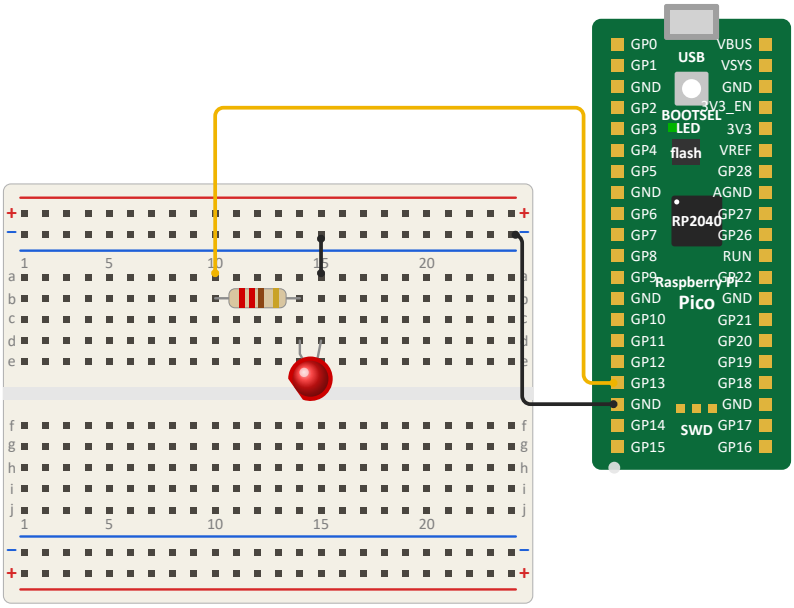
В **CircuitPython** вместо `asyncio` — привычный цикл: библиотека `adafruit_ble` *синхронная*, а сервисы описываются как Python-классы: `Service` с атрибутами-характеристиками. Её ставят командой `circup install adafruit_ble`.

Совет

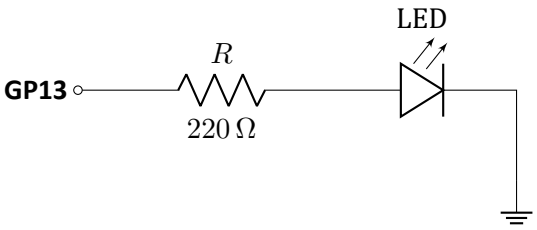
В nRF Connect выберите «PicoTemp» → «Connect», раскройте сервис «Environmental Sensing», нажмите значок «Read» (одно чтение) или значок «троекратная стрелка» (подписка на уведомления): значение обновляется каждые две секунды. Если характеристика не расшифровывается автоматически, нажмите на её значение и выберите формат «Signed Int, 16 bit».

Свои сервисы: светодиод с телефона

Для управления нужна *записываемая* характеристика. Придумаем собственный 128-битный UUID (его можно сгенерировать на сайте или командой `python -c "import uuid; print(uuid.uuid4())"`). Все такие идентификаторы в проекте должны быть разными.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 19.11: Светодиод на GP13 — исполнительное устройство BLE

Листинг 19.4: MicroPython: характеристика «состояние светодиода» (чтение и запись)

```
import asyncio
import aioble
import bluetooth
from machine import Pin

led = Pin(13, Pin.OUT)                # внешний светодиод на GP13

# Собственный (128-битный) идентификатор нашего сервиса и характеристики
_LED_SERVICE = bluetooth.UUID("5a2f0000-6b1c-4f6a-9d2e-1c3b7e8a9f00")
_LED_CHAR = bluetooth.UUID("5a2f0001-6b1c-4f6a-9d2e-1c3b7e8a9f00")

service = aioble.Service(_LED_SERVICE)
# read: телефон может прочитать состояние; write: может его изменить
led_char = aioble.Characteristic(service, _LED_CHAR, read=True, write=True, capture=True)
aioble.register_services(service)
led_char.write(b"\x00")               # начальное значение: выключен


async def write_task():
    """Ждём записи от телефона: capture=True возвращает и подключение, и данные."""
    while True:
        connection, data = await led_char.written()
        value = data[0] if data else 0
        led.value(1 if value else 0)
        print("записали:", value)


async def peripheral_task():
    while True:
        async with await aioble.advertise(
                250_000, name="PicoLED", services=[_LED_SERVICE]) as connection:
            print("подключился", connection.device)
            await connection.disconnected(timeout_ms=None)


async def main():
    await asyncio.gather(write_task(), peripheral_task())


asyncio.run(main())
```

Листинг 19.5: CircuitPython: то же

```
import board
import digitalio
from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services import Service
from adafruit_ble.uuid import VendorUUID
from adafruit_ble.characteristics import Characteristic
from adafruit_ble.characteristics.int import Uint8Characteristic

led = digitalio.DigitalInOut(board.GP13)       # внешний светодиод на GP13
led.direction = digitalio.Direction.OUTPUT

```

```
13
14 class LedService(Service):
15     """Собственный сервис: одна характеристика, которую можно читать и записывать."""
16     uuid = VendorUUID("5a2f0000-6b1c-4f6a-9d2e-1c3b7e8a9f00")
17     state = Uint8Characteristic(
18         uuid=VendorUUID("5a2f0001-6b1c-4f6a-9d2e-1c3b7e8a9f00"),
19         properties=Characteristic.READ | Characteristic.WRITE,
20         initial_value=0,
21     )
22
23
24 ble = BLERadio()
25 ble.name = "PicoLED"
26 svc = LedService()
27 adv = ProvideServicesAdvertisement(svc)
28
29 while True:
30     ble.start_advertising(adv)
31     while not ble.connected:
32         pass
33     print("подключился")
34     while ble.connected:
35         led.value = svc.state != 0              # читаем то, что записал телефон
36     print("отключился")
```

В nRF Connect выберите «PicoLED», найдите неизвестный сервис с вашим UUID, у характеристики нажмите значок записи («Write»), введите `01` (формат Byte array) и отправьте: светодиод загорится; запись `00` — погаснет. Флаг `capture=True` в **MicroPython** нужен, чтобы метод `written()` вернул ещё и записанные данные.

Учимся программировать: байты и `struct`

BLE передаёт не числа, а **байты**. Чтобы упаковать число в байты, нужна договорённость о формате: размер (1, 2, 4 байта), знак, порядок байтов. В Python для этого есть модуль `struct`: `struct.pack("<h", 2345)` даёт `b'\x29\x09'`, а `struct.unpack("<h", data)[0]` возвращает число обратно. Байты нумеруются, как элементы списка: `data[0]` — число от 0 до 255. Запомните три правила: размер, знак и порядок байтов должны одинаково пониматься обеими сторонами.

19.9 Nordic UART Service: «последовательный порт» по BLE

Иногда не нужен «настоящий» GATT-профиль, достаточно обмениваться текстовыми строками. Для этого придумали **Nordic UART Service** (NUS): сервис с двумя характеристиками — *RX* (телефон пишет, плата читает) и *TX* (плата пишет, телефон получает уведомлениями). Его понимают многие приложения: Serial Bluetooth Terminal (на Android), nRF Toolbox, Bluefruit Connect и др.

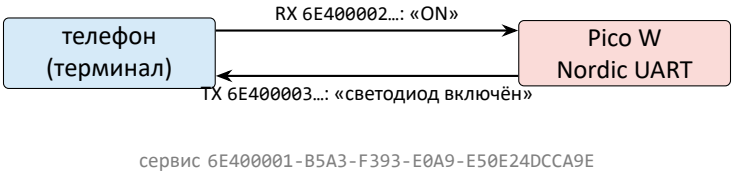


Рис. 19.12: Два канала Nordic UART Service: телефон пишет в RX, плата отвечает через TX

Листинг 19.6: MicroPython: Nordic UART на низкоуровневом API (команды ON, OFF, STATUS)

```
import bluetooth
from micropython import const
from machine import Pin

_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
_IRQ_GATTS_WRITE = const(3)

# Nordic UART Service: «последовательный порт» поверх BLE (его понимают многие приложения)
_UART_UUID = bluetooth.UUID("6E400001-B5A3-F393-E0A9-E50E24DCCA9E")
_UART_TX = (bluetooth.UUID("6E400003-B5A3-F393-E0A9-E50E24DCCA9E"), bluetooth.FLAG_NOTIFY)
_UART_RX = (bluetooth.UUID("6E400002-B5A3-F393-E0A9-E50E24DCCA9E"),
            bluetooth.FLAG_WRITE | bluetooth.FLAG_WRITE_NO_RESPONSE)
_UART_SERVICE = (_UART_UUID, (_UART_TX, _UART_RX))

led = Pin("LED", Pin.OUT)
conn = None                                  # номер текущего подключения (или None)


def irq(event, data):
    global conn
    if event == _IRQ_CENTRAL_CONNECT:
        conn = data[0]                       # (conn_handle, addr_type, addr)
    elif event == _IRQ_CENTRAL_DISCONNECT:
        conn = None
        advertise()
    elif event == _IRQ_GATTS_WRITE:
        conn_handle, attr_handle = data
        if attr_handle == rx_handle:
            text = ble.gatts_read(rx_handle).decode().strip().upper()
            handle_command(text)


def handle_command(text):
    """Команды: ON, OFF, STATUS --- ответ уходит уведомлением по каналу TX."""
    if text == "ON":
        led.on()
        reply("светодиод включён\n")
    elif text == "OFF":
        led.off()
        reply("светодиод выключен\n")
    elif text == "STATUS":
        reply("светодиод: %s\n" % ("ON" if led.value() else "OFF"))
    else:
        reply("неизвестная команда: " + text + "\n")


def reply(text):
    if conn is not None:
        data = text.encode()
        for i in range(0, len(data), 20):                    # пакеты по 20 байт (MTU 23)
            ble.gatts_notify(conn, tx_handle, data[i:i + 20])


def advertise():
    name = b"PicoUART"
```

```
    adv = b"\x02\x01\x06" + bytes((len(name) + 1, 0x09)) + name   # флаги + имя
    # 128-битный UUID (16 байт) не умещается рядом с именем в 31 байт, поэтому он в resp_data;
    # в пакете UUID записывается в обратном порядке байтов (little-endian)
    resp = bytes((17, 0x07)) +
    ↪  bytes(reversed(bytes.fromhex("6E400001B5A3F393E0A9E50E24DCCA9E")))
    ble.gap_advertise(100_000, adv_data=adv, resp_data=resp)


ble = bluetooth.BLE()
ble.active(True)
ble.irq(irq)
((tx_handle, rx_handle),) = ble.gatts_register_services((_UART_SERVICE,))
ble.gatts_set_buffer(rx_handle, 100)          # буфер для входящих строк
advertise()
print("Ищите PicoUART в приложении (Serial Bluetooth Terminal / nRF Connect)")
```

Листинг 19.7: CircuitPython: то же с готовым `UARTService`

```
import board
import digitalio
from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.nordic import UARTService

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

ble = BLERadio()
ble.name = "PicoUART"
uart = UARTService()                          # Nordic UART Service
adv = ProvideServicesAdvertisement(uart)

while True:
    ble.start_advertising(adv)
    print("жду подключения PicoUART...")
    while not ble.connected:
        pass
    ble.stop_advertising()
    print("подключился")
    while ble.connected:
        if uart.in_waiting:                   # пришли байты
            text = uart.readline()            # одна строка (bytes) или None
            if not text:
                continue
            cmd = text.decode().strip().upper()
            if cmd == "ON":
                led.value = True
                uart.write("светодиод включён\n".encode())
            elif cmd == "OFF":
                led.value = False
                uart.write("светодиод выключен\n".encode())
            elif cmd == "STATUS":
                uart.write(("светодиод: %s\n" % ("ON" if led.value else "OFF")).encode())
            else:
                uart.write(("неизвестная команда: " + cmd + "\n").encode())
    print("отключился")
```

Видно, как много скрывает `adafruit_ble`: в CircuitPython NUS — готовый класс, а в MicroPython сервис нужно описать, зарегистрировать и самим нарезать ответ на пакеты по 20 байт (размер

по умолчанию: MTU 23 байта минус 3 байта заголовка; при согласовании большего MTU можно отправлять больше). В приложении Serial Bluetooth Terminal выберите устройство `PicoUART`, подключитесь и отправьте `ON`: на плате загорится светодиод, а в терминале появится ответ. Поддержка BLE-терминалов зависит от приложения и версии Android/iOS: некоторые бесплатные приложения требуют включить геолокацию для сканирования (так устроены разрешения Android), на iPhone вместо Serial Bluetooth Terminal используйте nRF Connect или «Bluefruit Connect».

19.10 Центральное устройство: плата сканирует эфир

Плата может быть и *центральным*: сканировать эфир, находить другие BLE-устройства и подключаться к ним. Сканирование — хороший способ посмотреть, сколько BLE-устройств вокруг вас.

Листинг 19.8: MicroPython: сканирование BLE-устройств рядом (5 секунд)

```
import asyncio
import aioble


async def main():
    seen = {}
    # Сканируем 5 секунд; active=True просит у устройств ещё и «ответ на сканирование»
    async with aioble.scan(5000, interval_us=30000, window_us=30000, active=True) as scanner:
        async for result in scanner:
            addr = result.device.addr_hex()
            seen[addr] = (result.name() or "<без имени>", result.rssi)
    print("найдено устройств:", len(seen))
    for addr, (name, rssi) in sorted(seen.items(), key=lambda kv: -kv[1][1]):
        print("%s  %-16s %4d дБм" % (addr, name, rssi))


asyncio.run(main())
```

Листинг 19.9: CircuitPython: то же

```
from adafruit_ble import BLERadio

ble = BLERadio()
seen = {}
# Сканируем 5 секунд; в цикле приходят рекламные пакеты (один адрес --- много раз)
for adv in ble.start_scan(timeout=5, minimum_rssi=-95):
    name = adv.complete_name or adv.short_name or "<без имени>"
    seen[str(adv.address)] = (name, adv.rssi)
ble.stop_scan()

print("найдено устройств:", len(seen))
for addr, (name, rssi) in sorted(seen.items(), key=lambda kv: -kv[1][1]):
    print("%s  %-16s %4d дБм" % (addr, name, rssi))
```

Вы увидите десятки устройств: часы, наушники, лампочки, чьи-то телефоны. У многих нет имени, а адреса случайные (приватность). Чтобы *подключиться* к найденному устройству в **MicroPython**, используют `await device.connect()`, затем `await connection.service(UUID)`, `await service.characteristic(UUID)` и `await char.read()` или `await char.subscribe(notify=True)`; в **CircuitPython** — `ble.connect(advertisement)` и работу с сервисом по классам (см. примеры в документации библиотек).

19.11 Одновременная работа Wi-Fi и BLE

Радиочип CYW43439 один, антенна одна, и работают они на одной частоте. Поэтому Wi-Fi и BLE *делят эфирное время*: чип быстро переключается между ними. На практике это значит:

- одновременная работа поддерживается, но каждому достаётся меньше времени; при активной передаче данных по Wi-Fi BLE-соединения могут рваться, и наоборот;
- скорость Wi-Fi заметно падает во время сканирования BLE, поэтому сканируйте редко;
- реклама и соединение BLE потребляют память: на RP2040 всего 264 kB ОЗУ, а стеку Bluetooth, TLS и вашей программе нужна своя доля. Если появились `MemoryError` или зависания, отключите ненужное: например, не включайте Wi-Fi, пока не нужен;
- в **CircuitPython** одновременная работа Wi-Fi и BLE на Pico W может поддерживаться ограниченно и зависит от версии — смотрите примечания к выпуску.



Рис. 19.13: Разделение эфирного времени между Wi-Fi и BLE (схематично)

19.12 Безопасность и приватность

- **Реклама открыта для всех**: имя устройства и данные в рекламе видят все вокруг. Не кладите туда ничего личного.
- **Приватность**: телефоны используют случайные адреса, которые меняются, чтобы вас нельзя было отслеживать. Плата с постоянным адресом и именем — «метка» для слежки; в серьёзных проектах включают приватные адреса.
- **Сопряжение (pairing) и шифрование**: по умолчанию наши примеры позволяют подключиться и записывать команды *любому* телефону поблизости. Для защиты используют сопряжение с PIN-кодом (Passkey), создающее общий ключ и шифрование (*bonding* запоминает ключ). В **MicroPython** это настраивается флагами `gap_pair`/`_IRQ_PASSKEY_ACTION` и `ble.config(bond=True, mitm=True, le_secure=True)`, в **CircuitPython** сопряжение возможно через `connection.pair()`. Доступность этого зависит от версии прошивки.
- **Не управляйте опасными устройствами** (замок, нагреватель) без сопряжения и проверки: дальность BLE — десятки метров, и услышать команду может сосед.

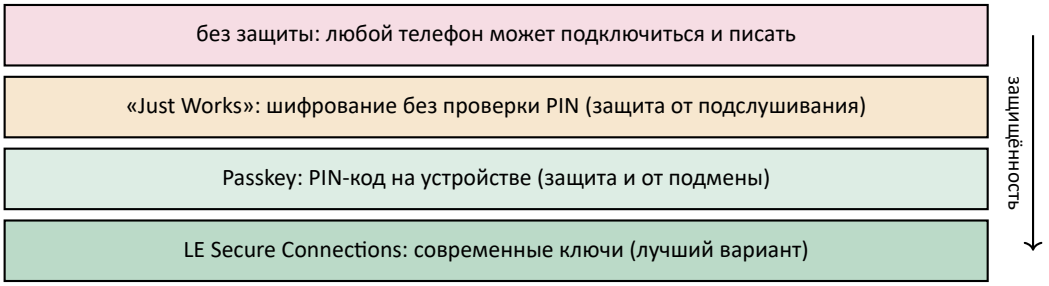


Рис. 19.14: Уровни защиты BLE-соединения: от открытого доступа до современных ключей

19.13 Практика: BLE-метеопульт

Соберите проект: термометр с уведомлениями (рис. 19.10) и светодиод, управляемый с телефона (рис. 19.11). Объедините два сервиса в одной программе: у одного устройства может быть несколько сервисов. Проверьте работу в nRF Connect.

Если не работает

- **Устройство не появляется в списке сканирования** — у платы без W нет радио; реклама не запущена; на Android выключены геолокация/Bluetooth; телефон слишком далеко.
- **`ImportError: no module named aioble`** — установите пакет: `mip.install("aioble")`.
- **Ошибка при `import bluetooth` или `ble.active(True)`** — прошивка собрана без Bluetooth или плата без W; нужна прошивка для Pico W с поддержкой BLE.
- **Телефон подключается, но данных нет** — не подписались на уведомления (CCCD) или записываете значение не в тот формат/UUID. Сверьте UUID и формат в nRF Connect.
- **После отключения устройство «пропало»** — не запущена реклама снова (в `irq` или в цикле программы).
- **Связь обрывается при активном Wi-Fi** — конфликт за эфир; уменьшите обмен по Wi-Fi или подождите, пока BLE-обмен закончится.
- **`import _bleio` в CircuitPython даёт ошибку** — в вашей версии нет BLE для Pico W: обновите прошивку CircuitPython или используйте MicroPython.
- **Приложение показывает «неизвестный сервис»** — так и должно быть для собственного UUID; значение смотрите как набор байтов.

Итоги урока

- BLE — радио с малым потреблением для коротких сообщений; на Pico W работает через радиочип CYW43439 (только на платах с W).
- GAP отвечает за рекламу и подключение (роли peripheral/central), GATT — за данные: сервис → характеристика → значение, у каждого свой UUID.
- С характеристикой можно читать (read), писать (write) и получать уведомления (notify); число упаковывается в байты (`struct.pack`).
- В MicroPython — модуль `bluetooth` (IRQ) или удобный `aioble`; в CircuitPython — `adafruit_ble` (нужна свежая версия 9.x).
- Nordic UART даёт «последовательный порт» по BLE; плата может быть и центральным (сканирование).
- Wi-Fi и BLE делят одно радио и память; реклама открыта, сопряжение и PIN нужны для защиты.

Проверь себя

1. Чем BLE отличается от классического Bluetooth? Для каких задач он подходит?
2. Что делают GAP и GATT? Кто такие central и peripheral?
3. Из каких уровней состоит дерево данных GATT? Для чего нужны UUID?
4. Чем `read`, `write` и `notify` отличаются друг от друга?

5. Почему Wi-Fi и BLE на одной плате могут мешать друг другу и как защитить BLE-соединение?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Запустите рекламу и найдите свою плату в nRF Connect. Запишите имя, уровень сигнала и адрес; измените имя на своё.
2. ★ Определите по рис. 19.8, как будет выглядеть температура 18,06 °C в байтах характеристики 0x2A6E.
3. ★★ Подключитесь к термометру в nRF Connect, подпишитесь на уведомления и посмотрите, как меняются значения, если подышать на плату.
4. ★★ Измените пример светодиода: значение `02` пусть означает «мигать», `00` — «выключить», `01` — «включить» (используйте `asyncio`).
5. ★★★ Добавьте в Nordic UART команду `TEMP`, возвращающую температуру текстом, и команду `BLINK n`, мигающую `n` раз.
6. ★★★★ Сделайте «ловушку для сигнала»: плата в роли центрального сканирует эфир каждые 10 секунд и записывает в файл число BLE-устройств рядом; через сутки постройте график и определите, в какие часы вокруг больше всего устройств.

Урок 20. Моторы, реле и драйверы

Цель	понять, почему моторы и реле нельзя подключать к выводам напрямую, и научиться управлять ими через транзисторный ключ, реле-модуль, H-мост и драйвер шагового двигателя
Опирается на	уроки 2–5: выход GPIO, ШИМ; закон Ома; урок 15: питание платы
Результат	мотор вращается с регулируемой скоростью и в двух направлениях; шаговый двигатель поворачивается на заданный угол; реле включает лампу; всё с общей землёй и защитным диодом
Понадобится	Pico, макетная плата, транзистор BC337 (или 2N2222), диод 1N4007, резисторы $680\,\Omega$ и $10\,\text{k}\Omega$, моторчик $3-6\,\text{V}$, модуль DRV8833 (или TB6612FNG), шаговый двигатель 28BYJ-48 с платой ULN2003, реле-модуль, батарейный отсек на 4 элемента AA, сервопривод SG90

20.1 Идея урока коротко

Вывод Pico умеет подавать *сигнал*, но не умеет давать *мощность*. Чтобы управлять мотором, мы ставим между платой и нагрузкой **усилитель-выключатель**: транзистор, реле или специальную микросхему-драйвер. Слабый ток вывода лишь командует выключателем, а основной ток течёт от отдельного источника питания. Три правила урока:

1. **нагрузку питаем отдельно**, а не от вывода и (почти всегда) не от $3{,}3\,\text{V}$;
2. **у катушки есть защитный диод**: мотор и реле — индуктивная нагрузка;
3. **земля общая**: минус внешнего питания соединяется с GND платы.

20.2 Почему мотор нельзя питать от вывода

В уроке 2 мы выяснили, что вывод GPIO даёт ток порядка 4 мА (максимум около 12 мА, а все выводы вместе — заметно меньше). Светодиоду этого хватает. Мотору — нет.

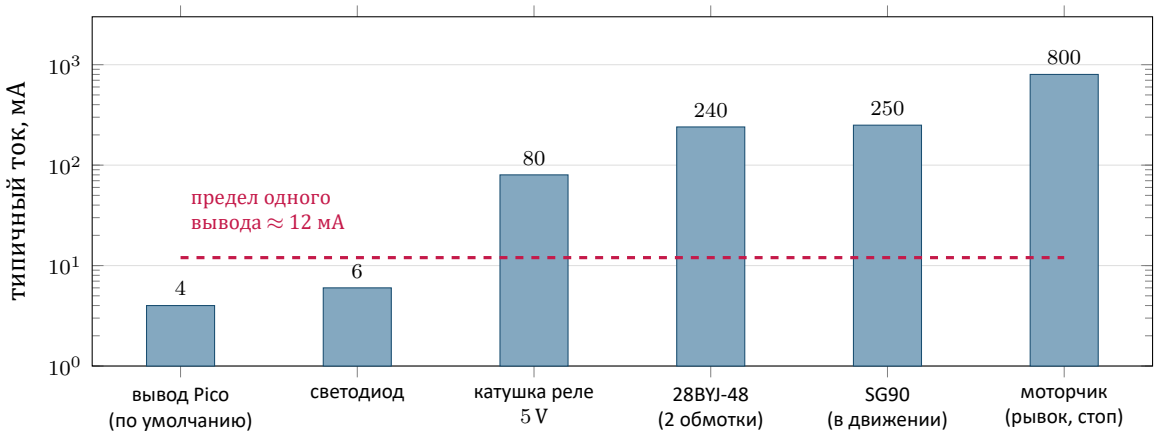


Рис. 20.1: Сколько тока нужно разным нагрузкам (порядок величин, шкала логарифмическая). Всё, что выше пунктира, питать от вывода нельзя

Мотор потребляет не постоянный ток: в момент пуска и при заклинивании ротора ток в несколько раз выше «рабочего», потому что неподвижный мотор — просто кусок провода. Если ограничить ток вывода нечем, вывод (а с ним и вся микросхема) перегреется и выйдет из строя. Моторчик на $3\,\text{V}$ с сопротивлением обмотки около $3\,\Omega$ в момент пуска потребует

$$I_{\text{пуска}} \approx \frac{U}{R} = \frac{3\,\text{В}}{3\,\Omega} = 1\,\text{А} \;=\; 1000\,\text{мА},$$

то есть почти в сто раз больше допустимого.

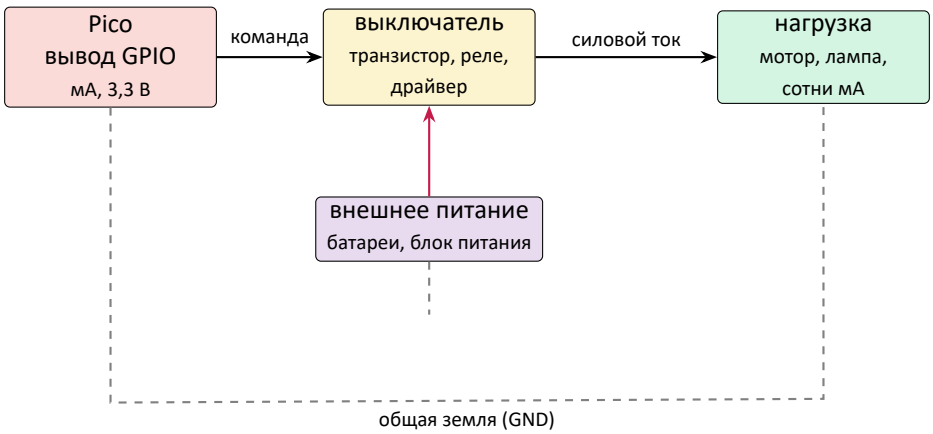


Рис. 20.2: Идея урока: слабый сигнал командует выключателем, а мощность приходит от отдельного источника

20.3 Индуктивная нагрузка и защитный диод

Обмотка мотора или реле — это катушка, у неё есть **индуктивность** L. Катушка не любит, когда ток через неё меняют быстро: она «сопротивляется» этому и создаёт напряжение самоиндукции

$$U_L = -L\,\frac{\Delta I}{\Delta t}.$$

Представьте тяжёлую тележку: разогнать её сложно, а остановить мгновенно — значит получить сильный удар. Если транзистор выключает ток $0{,}3$ А в катушке с $L = 10$ mH за одну микросекунду, то формально получится $0{,}01 \cdot 0{,}3/10^{-6} = 3000$ V. На практике скачок ограничивают паразитные ёмкости, но всё равно это десятки и сотни вольт: транзистор пробивается, а помехи перезагружают Pico.

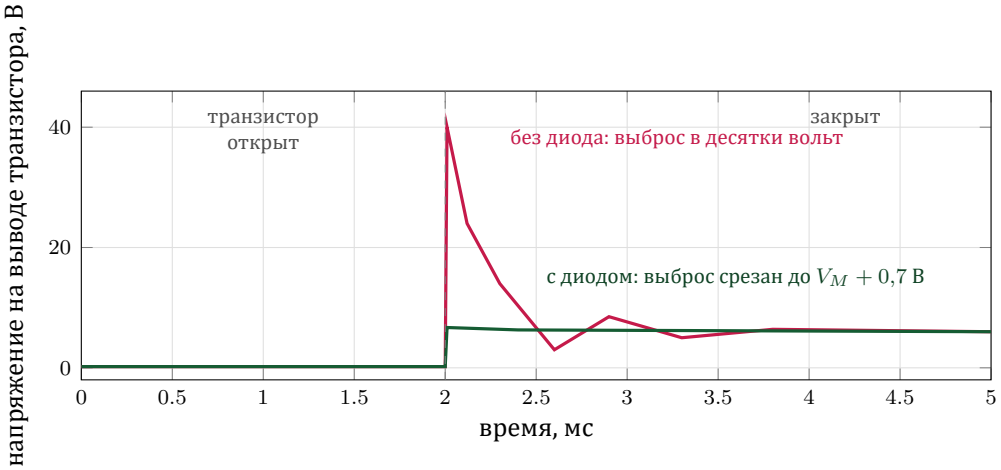
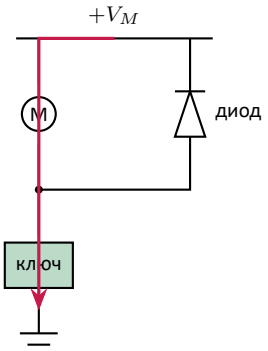
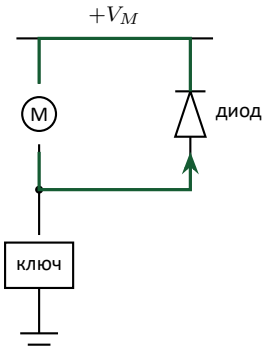


Рис. 20.3: Напряжение на транзисторе в момент выключения тока катушки (схематично, $V_M = 6$ В)

Спасает **защитный (обратный, «ловящий», flyback) диод**. Его включают параллельно катушке *в обратном направлении*: катодом (полоской) к плюсу питания. Пока ключ открыт, диод закрыт и не мешает. Когда ключ выключают, ток катушки не может исчезнуть мгновенно и находит себе новый путь — через диод, по кругу, где постепенно затухает.



а) ключ замкнут: ток идёт через мотор



б) ключ разомкнут: ток замыкается через диод

Рис. 20.4: Путь тока катушки с защитным диодом

Осторожно

Защитный диод ставят **всегда**: у мотора, реле, электромагнита, соленоида. Полярность легко перепутать: если диод включён «по току», он замкнёт питание накоротко и сгорит. Запомните: *полоска на диоде смотрит на плюс*. Подходят 1N4001–1N4007 для моторов и реле, 1N4148 — только для катушек с малым током. В готовых модулях реле и драйверов диоды уже стоят.

20.4 Транзисторный ключ

Транзистор — электронный выключатель, которым управляет маленький ток (или напряжение). Для микроконтроллеров чаще всего используют два типа.

- **Биполярный NPN** (BC337, 2N2222, 2N3904): открывается током *базы*. Нужен резистор в цепи базы. Выводы BC337 при взгляде на плоскую сторону слева направо: коллектор (К), база (Б), эмиттер (Э).
- **Полевой N-канальный MOSFET** (IRLZ44N, AO3400, 2N7000): открывается *напряжением* на затворе, ток затвора почти не течёт. Годятся только «логические» (logic-level) модели, которые полностью открываются от $3{,}3$ V.

Расчёт резистора базы

Пусть моторчик потребляет под нагрузкой $I_C = 150$ mA. Коэффициент усиления BC337 — h_{FE} не меньше 100 (в справочнике, для надёжности берём минимум). Чтобы транзистор *гарантированно насытился* (открылся полностью и почти не грелся), базовый ток берут с запасом в 2–3 раза:

$$I_B \geq \frac{I_C \cdot 2{,}5}{h_{FE\ \min}} = \frac{0{,}15\ \text{А} \cdot 2{,}5}{100} \approx 3{,}8\ \text{мА}.$$

Напряжение на переходе база–эмиттер около $0{,}7$ V, остальное падает на резисторе:

$$R_B = \frac{U_{GPIO} - U_{BE}}{I_B} = \frac{3{,}3 - 0{,}7}{0{,}0038} \approx 680\ \Omega.$$

Ток базы 3,8 мА укладывается в возможности вывода. Если нагрузка больше (скажем, 500 мА), ток базы получится слишком большим — в этом случае ставят MOSFET или готовый драйвер.

MOSFET: ключ, управляемый напряжением

У полевого транзистора ток затвора почти нулевой, поэтому «токового» расчёта нет, но остаются два правила: резистор $220\ \Omega$–$330\ \Omega$ *последовательно* с затвором (затвор — это конденсатор, в момент переключения он заряжается большим коротким током) и резистор $10\ \text{k}\Omega$ *между затвором и землёй*, чтобы мотор не запускался сам, пока Pico загружается и выводы «висят в воздухе».

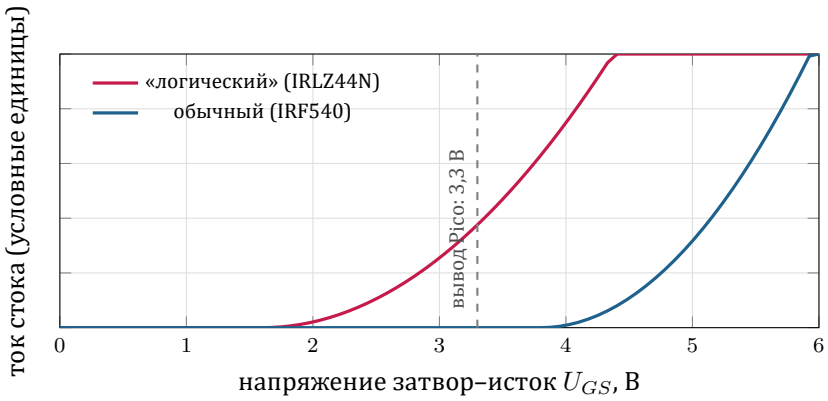


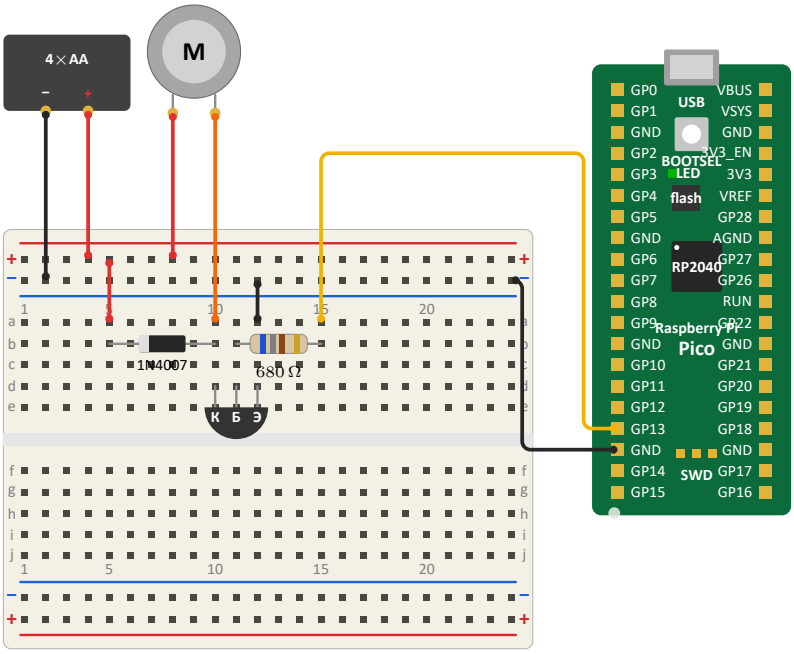
Рис. 20.5: Почему нужен именно «логический» MOSFET: при $3{,}3$ V обычный транзистор почти закрыт (схематичные кривые)

	Биполярный (BC337)	MOSFET (IRLZ44N, AO3400)
Управляется	током базы	напряжением на затворе
Нагрузка до	≈ 0,5 А	от 1 А до десятков ампер
Деталь в цепи управления	резистор базы $\approx 680\,\Omega$	резистор $330\,\Omega$ и подтяжка $10\,\mathrm{k}\Omega$
Падение при открытии	0,1–0,3 В	доли вольта (малое $R_{DS(on)}$)
Где удобен	светодиоды, реле, малые моторы	мощные моторы, ленты, быстрый ШИМ

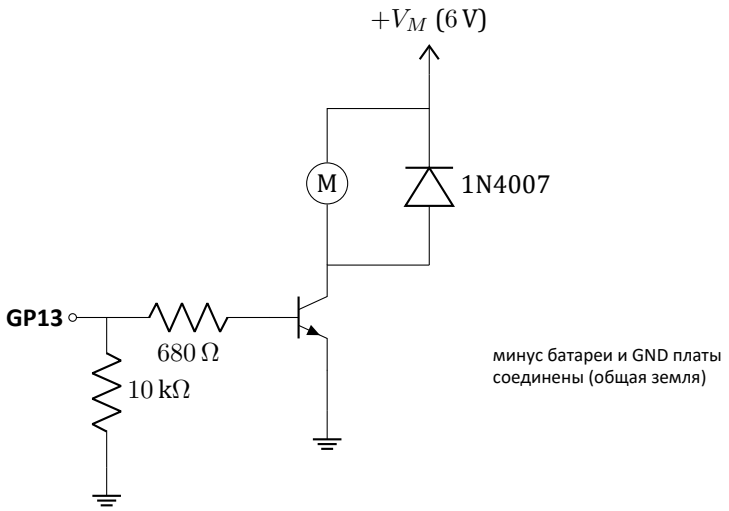
Таблица 20.1: Выбор транзисторного ключа

Монтажная схема: мотор через BC337

Моторчик получает питание от батарейного отсека (4 элемента AA = $6\,\mathrm{V}$, для маленького моторчика на $3\,\mathrm{V}$ достаточно двух элементов). Вывод **GP13** через резистор $680\,\Omega$ управляет базой. Диод стоит параллельно мотору: полоска (катод) на плюс. Минус батареи соединён с землёй Pico — это та самая общая земля.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 20.6: Мотор на транзисторе BC337: управление с GP13

Осторожно

Не подключайте мотор *без* батареи только к 3V3 или VSYS: бросок тока просадит питание и перезагрузит плату, а через USB может отключиться порт компьютера. Для опытов с мотором используйте отдельный источник или (для маленьких моторчиков) вывод VBUS, но только после измерения тока. Ничего не подключайте к плате при включённом питании моторов, пока не проверите полярность.

Управление скоростью

Скоростью мотора управляет ШИМ из урока 5: транзистор быстро включается и выключается, мотор из-за инерции и индуктивности «усредняет» импульсы. Чем больше скважность (доля времени «включено»), тем выше среднее напряжение и скорость.

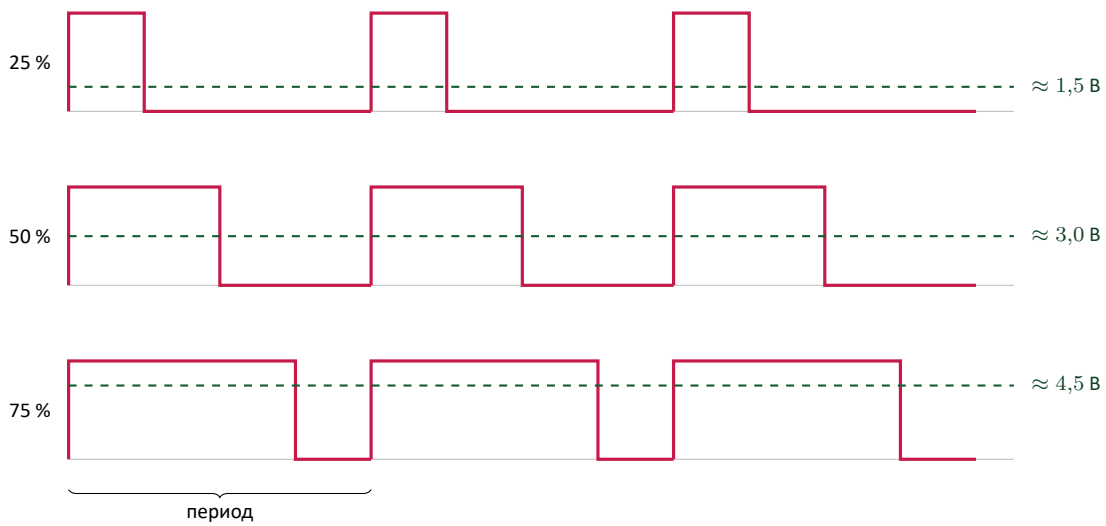


Рис. 20.7: ШИМ с разной скважностью и среднее напряжение на моторе при $V_M = 6$ В (пунктир)

Листинг 20.1: MicroPython: плавный разгон мотора с ШИМ на GP13

```
from machine import Pin, PWM
import time

# Коллекторный моторчик через NPN-транзистор BC337.
# База транзистора подключена к GP13 через резистор 680 Ом.
pwm = PWM(Pin(13))
pwm.freq(2000)              # 2 кГц: выше слышимого «писка» не уйти, но для опыта хватает
pwm.duty_u16(0)             # сначала мотор выключен

def speed(percent):
    """Скорость 0...100 %: доля времени, когда транзистор открыт."""
    percent = max(0, min(100, percent))      # защита от выхода за границы
    pwm.duty_u16(percent * 65535 // 100)

try:
    # «Мягкий старт»: разгоняем мотор плавно, чтобы не было броска тока
    for p in range(0, 101, 5):
        speed(p)
        time.sleep(0.2)
    time.sleep(2)
    for p in range(100, -1, -5):
        speed(p)
        time.sleep(0.2)
finally:
    pwm.duty_u16(0)          # при любой ошибке или Ctrl+C мотор должен остановиться
    pwm.deinit()
```

Листинг 20.2: CircuitPython: то же

```
import time
import board
import pwmio

# Коллекторный моторчик через NPN-транзистор BC337.
# База транзистора подключена к GP13 через резистор 680 Ом.
pwm = pwmio.PWMOut(board.GP13, frequency=2000, duty_cycle=0)
```

```
8
9  def speed(percent):
10     """Скорость 0...100 %: доля времени, когда транзистор открыт."""
11     percent = max(0, min(100, percent))      # защита от выхода за границы
12     pwm.duty_cycle = percent * 65535 // 100  # в CircuitPython диапазон тоже 0...65535
13
14 try:
15     # «Мягкий старт»: разгоняем мотор плавно
16     for p in range(0, 101, 5):
17         speed(p)
18         time.sleep(0.2)
19     time.sleep(2)
20     for p in range(100, -1, -5):
21         speed(p)
22         time.sleep(0.2)
23 finally:
24     pwm.duty_cycle = 0       # мотор обязан остановиться при любой ошибке
25     pwm.deinit()
```

Конструкция `try ... finally` гарантирует, что при ошибке или остановке программы клавишами Ctrl+C мотор будет выключен: без неё ШИМ мог бы продолжать вращать вал. Частоту ШИМ 2 kHz легко услышать: мотор слегка «поёт». Для тихой работы берут 15 – 20 kHz, но маленькие транзисторы на такой частоте греются сильнее — здесь нужен MOSFET.

Это интересно

Мотор можно вращать только в одну сторону, если ключ один. Чтобы менять направление, нужно менять полярность питания — для этого придумали H-мост (ниже в этом уроке).

20.5 Учимся программировать: классы и объекты

В примерах ниже появляется ключевое слово `class`. **Класс** — это чертёж, по которому создают одинаковые объекты. У мотора есть *данные* (на каких выводах он сидит) и *действия* (вращаться, тормозить). Класс объединяет их в одном месте:

- `__init__` — «конструктор», выполняется при создании объекта; здесь настраиваем выводы;
- `self` — ссылка на сам объект; через неё методы видят данные (`self.a`, `self.phase`);
- метод — функция внутри класса: `motor.drive(50)`, `motor.brake()`.

Выгода: два мотора — это просто два объекта `Motor(...)` с разными выводами, а не два экземпляра скопированного кода. Эта привычка пригодится в уроках 13 и 25.

20.6 Реле: выключатель с электромагнитом

Реле — электромагнит, который замыкает или размыкает пару контактов. Главное достоинство: *управляющая цепь и силовая цепь электрически не связаны*. У контактов три вывода: COM (общий), NO (normally open — нормально разомкнут) и NC (normally closed — нормально замкнут).

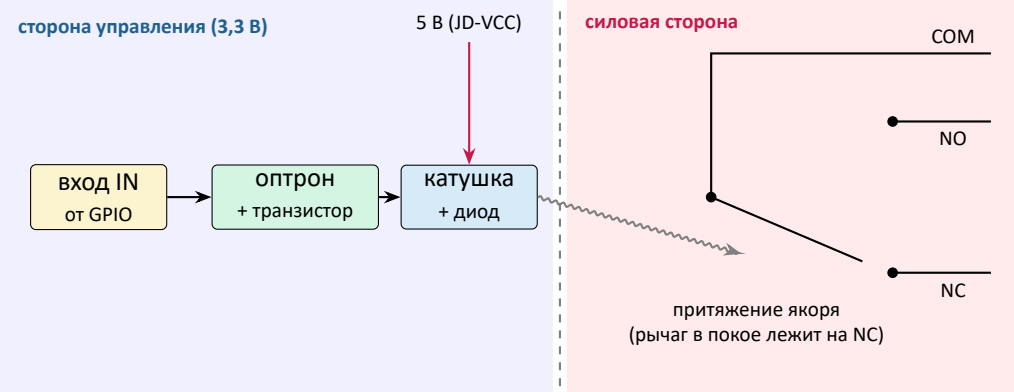


Рис. 20.8: Устройство реле-модуля: управление и контакты электрически разделены оптроном и электромагнитом

Готовые модули реле удобны: на плате уже есть транзистор, защитный диод и светодиод индикации. Обычно модуль включается *низким* уровнем на входе IN (active low) и имеет перемычку JD-VCC.

- Перемычка стоит: катушка и логика питаются от одного источника (5 V), развязки нет.
- Перемычку снимают: катушку питают от внешнего 5 V на JD-VCC, а VCC модуля (логика, оптрон) получает 3V3 от Pico. Так цепи полностью развязаны.

Осторожно

Сетевое напряжение 220 V **в этом курсе подключать нельзя.** Оно смертельно опасно: удар током, пожар. На контактах реле опыты проводят только с безопасными нагрузками: лампочка на батарейках, светодиодная лента 5 V, 12 V от блока питания. Подключение сети — дело взрослого специалиста с понимающим допуском и сертифицированными изделиями.

Листинг 20.3: MicroPython: реле-модуль на GP14, включение раз в 5 секунд

```
from machine import Pin
import time

# Реле-модуль: вход IN подключён к GP14.
# Большинство дешёвых модулей включаются НИЗКИМ уровнем (active low).
ACTIVE_LOW = True

# сразу задаём «выключено», чтобы реле не щёлкнуло при создании вывода
relay = Pin(14, Pin.OUT, value=1 if ACTIVE_LOW else 0)

def relay_set(on):
    """Включить или выключить реле с учётом полярности модуля."""
    relay.value(int(on != ACTIVE_LOW))   # active low: включить = 0, выключить = 1

relay_set(False)             # при старте реле должно быть выключено

state = False
last = time.ticks_ms()
while True:
    if time.ticks_diff(time.ticks_ms(), last) >= 5000:   # каждые 5 секунд
        state = not state
        relay_set(state)
        last = time.ticks_ms()
        print("реле:", "включено" if state else "выключено")
```

```
    time.sleep_ms(50)
```

Листинг 20.4: CircuitPython: то же

```
import time
import board
import digitalio

# Реле-модуль: вход IN подключён к GP14.
# Большинство дешёвых модулей включаются НИЗКИМ уровнем (active low).
ACTIVE_LOW = True

relay = digitalio.DigitalInOut(board.GP14)
# switch_to_output сразу задаёт «выключено», чтобы реле не щёлкнуло при старте
relay.switch_to_output(value=ACTIVE_LOW)

def relay_set(on):
    """Включить или выключить реле с учётом полярности модуля."""
    relay.value = (on != ACTIVE_LOW)     # active low: включить = False, выключить = True

relay_set(False)             # при старте реле должно быть выключено

state = False
last = time.monotonic()
while True:
    if time.monotonic() - last >= 5.0:     # каждые 5 секунд
        state = not state
        relay_set(state)
        last = time.monotonic()
        print("реле:", "включено" if state else "выключено")
    time.sleep(0.05)
```

Обратите внимание, что вывод создаётся уже в состоянии «выключено» (`value=1` для active low): если сначала создать выход, а потом записать значение, реле на мгновение щёлкнет. Реле — механическое устройство: ресурс контактов ограничен, а переключать чаще раза в секунду нельзя. ШИМ с реле *не используют* — для регулировки нужны транзистор или драйвер.

Совет

Если на вход подан высокий уровень 3,3 V, а реле всё равно остаётся включённым, значит, оптрон модуля питается от 5 V и «не дотягивает» до полного запирания. Снимите перемычку и подайте на VCC модуля 3,3 V (JD-VCC оставьте на 5 V).

20.7 H-мост: направление и скорость

Чтобы мотор вращался в обе стороны, нужно уметь менять полярность питания. Это делают четыре ключа, расположенные буквой H: **H-мост**. Сам мотор — перекладина буквы H.

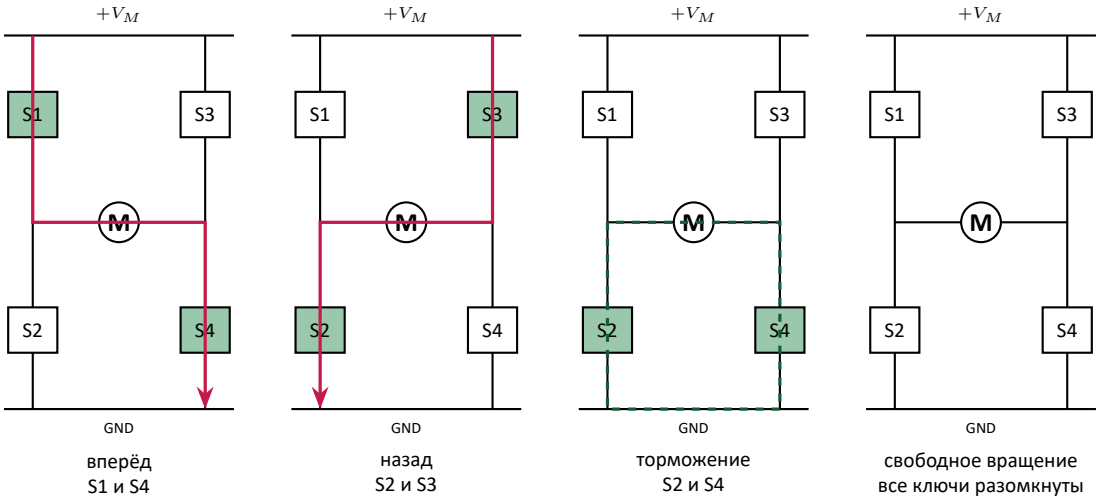


Рис. 20.9: Четыре режима H-моста. Зелёный — замкнутый ключ. Включать S1 и S2 (или S3 и S4) одновременно нельзя: это короткое замыкание питания

Если замкнуть S1 и S4, ток идёт слева направо — мотор вращается вперёд; S3 и S2 — в обратную сторону. Если замкнуть оба нижних ключа S2 и S4, обмотка мотора оказывается замкнутой накоротко: вращающийся мотор вырабатывает ток, который его же тормозит — это **динамическое торможение**. Если разомкнуть всё, мотор крутится по инерции (**свободное вращение**). В готовой микросхеме ключи и защитная логика уже внутри, а управляют ей двумя входами на мотор. Для DRV8833 (вход IN1, IN2) таблица такая:

IN1	IN2	Режим	Что делает мотор
0	0	свободное вращение	останавливается по инерции
ШИМ	0	вперёд	скорость пропорциональна скважности
0	ШИМ	назад	то же, в обратную сторону
1	1	торможение	быстро останавливается и держит вал

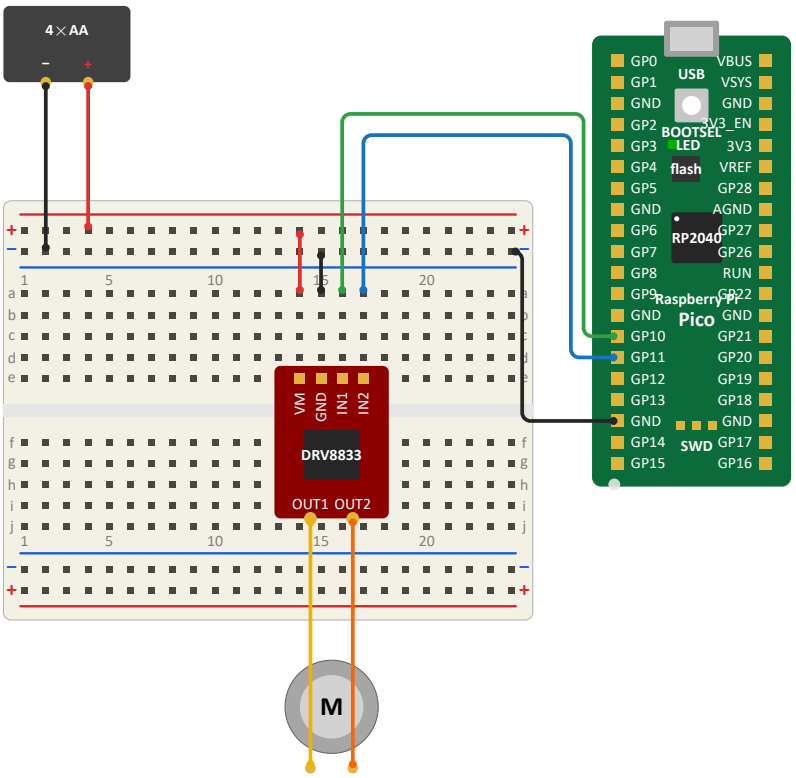
Таблица 20.2: Управление мотором через H-мост DRV8833

Драйвер	Питание мотора	Ток на канал	Особенности
DRV8833	2,7 – 10,8 V	около 1,2 A	малое падение, подходит для батареек; два канала
TB6612FNG	4,5 – 13,5 V	1,2 A (пик 3,2 A)	малое падение; ШИМ до 100 кГц; два канала
L9110S	2,5 – 12 V	около 0,8 A	самый простой и дешёвый модуль
L298N	5 – 35 V	до 2 A	старая биполярная схема: теряет около 2 В и греется; нужен радиатор

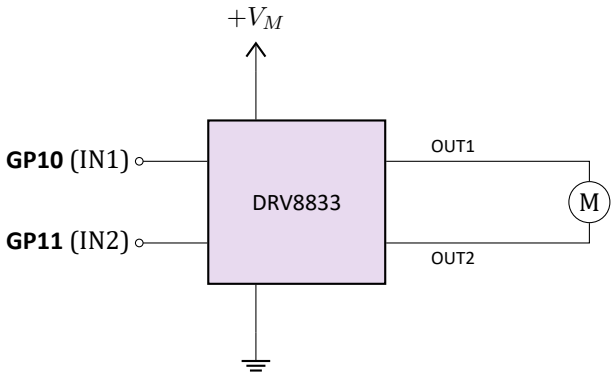
Таблица 20.3: Популярные драйверы коллекторных моторов (ориентировочные данные; сверяйтесь с документацией своего модуля)

Для питания от 3 – 6 V лучше всего подходят DRV8833 и TB6612FNG. L298N выпускается в виде красного модуля с радиатором, но из-за потерь на нём моторчик 3 V почти не вращается.

Монтажная схема: DRV8833 и мотор



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 20.10: Реверсивное управление мотором: модуль DRV8833, входы на GP10 и GP11

Если на вашем модуле есть вывод SLEEP (nSLEEP) или STBY, его нужно подтянуть к 3V3 (на многих модулях это уже сделано). Названия и порядок выводов у разных модулей отличаются — ориентируйтесь на надписи на плате, а не на рисунок.

Листинг 20.5: MicroPython: класс `Motor` и реверс через H-мост (DRV8833)

```
from machine import Pin, PWM
import time

class Motor:
    """Один мотор на двух входах H-моста (DRV8833: IN1 и IN2)."""
    def __init__(self, pin_a, pin_b, freq=2000):
        self.a = PWM(Pin(pin_a))
        self.b = PWM(Pin(pin_b))
```

```
        self.a.freq(freq)
        self.b.freq(freq)
        self.coast()

    def drive(self, speed):
        """speed от -100 (назад) до +100 (вперёд)."""
        speed = max(-100, min(100, speed))
        duty = abs(speed) * 65535 // 100
        if speed >= 0:
            self.a.duty_u16(duty)    # ШИМ на одном входе...
            self.b.duty_u16(0)       # ...на втором ноль
        else:
            self.a.duty_u16(0)
            self.b.duty_u16(duty)

    def brake(self):
        """Оба входа в «1»: обмотка мотора замкнута, он быстро тормозит."""
        self.a.duty_u16(65535)
        self.b.duty_u16(65535)

    def coast(self):
        """Оба входа в «0»: мотор свободно вращается по инерции."""
        self.a.duty_u16(0)
        self.b.duty_u16(0)

motor = Motor(10, 11)
try:
    for s in (40, 80, 100):
        motor.drive(s)
        time.sleep(1.5)
    motor.brake()
    time.sleep(1)                    # пауза перед реверсом: не меняем направление «на лету»
    for s in (-40, -80, -100):
        motor.drive(s)
        time.sleep(1.5)
    motor.coast()
finally:
    motor.coast()
```

Листинг 20.6: CircuitPython: то же

```
import time
import board
import pwmio

class Motor:
    """Один мотор на двух входах H-моста (DRV8833: IN1 и IN2)."""
    def __init__(self, pin_a, pin_b, freq=2000):
        self.a = pwmio.PWMOut(pin_a, frequency=freq, duty_cycle=0)
        self.b = pwmio.PWMOut(pin_b, frequency=freq, duty_cycle=0)

    def drive(self, speed):
        """speed от -100 (назад) до +100 (вперёд)."""
        speed = max(-100, min(100, speed))
        duty = abs(speed) * 65535 // 100
        if speed >= 0:
            self.a.duty_cycle = duty     # ШИМ на одном входе...
            self.b.duty_cycle = 0        # ...на втором ноль
```

```
        else:
            self.a.duty_cycle = 0
            self.b.duty_cycle = duty

    def brake(self):
        """Оба входа в «1»: обмотка мотора замкнута, он быстро тормозит."""
        self.a.duty_cycle = 65535
        self.b.duty_cycle = 65535

    def coast(self):
        """Оба входа в «0»: мотор свободно вращается по инерции."""
        self.a.duty_cycle = 0
        self.b.duty_cycle = 0

motor = Motor(board.GP10, board.GP11)
try:
    for s in (40, 80, 100):
        motor.drive(s)
        time.sleep(1.5)
    motor.brake()
    time.sleep(1)                          # пауза перед реверсом
    for s in (-40, -80, -100):
        motor.drive(s)
        time.sleep(1.5)
    motor.coast()
finally:
    motor.coast()
```

Выводы GP10 и GP11 принадлежат одному ШИМ-слайсу (канала A и B одного блока), поэтому частота у них общая — это нас устраивает, и в **CircuitPython** обе линии создаются с одинаковой `frequency`. Пауза между `brake()` и реверсом не даёт мотору получать «удар» в момент смены направления на полной скорости.

20.8 Шаговый двигатель

Шаговый двигатель поворачивается не плавно, а *шагами*: каждый шаг — строго определённый угол. Нет датчиков — а положение вала всё равно известно: нужно просто посчитать шаги. Поэтому шаговые двигатели стоят в принтерах, станках с ЧПУ, роботах-рисовальщиках.
Внутри 28BYJ-48 четыре обмотки, расположенные вокруг магнитного ротора (в реальности ротор многополюсный, а на рисунке он упрощён). Подавая ток на обмотки в определённом порядке, мы заставляем магнит ротора «гоняться» за магнитным полем. Если включать по одной обмотке (1, 2, 3, 4) — получаем полные шаги; если чередовать «одна» и «две вместе» — **полушаги**: вдвое больше положений, плавнее ход, но чуть меньше момент в некоторых положениях.

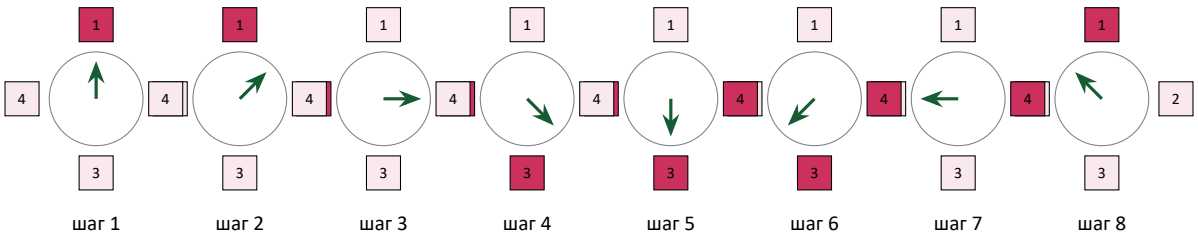


Рис. 20.11: Полушаговая последовательность: цифры 1–4 — обмотки (IN1–IN4), закрашена включённая обмотка, стрелка — ротор. Это упрощённая модель: у реального двигателя за 8 полушагов ротор поворачивается лишь на 45°

Полушаг	IN1	IN2	IN3	IN4	Включены обмотки
1	1	0	0	0	одна (1)
2	1	1	0	0	две (1+2)
3	0	1	0	0	одна (2)
4	0	1	1	0	две (2+3)
5	0	0	1	0	одна (3)
6	0	0	1	1	две (3+4)
7	0	0	0	1	одна (4)
8	1	0	0	1	две (4+1)

Таблица 20.4: Полушаговая последовательность 28BYJ-48 (после восьмого полушага цикл повторяется)

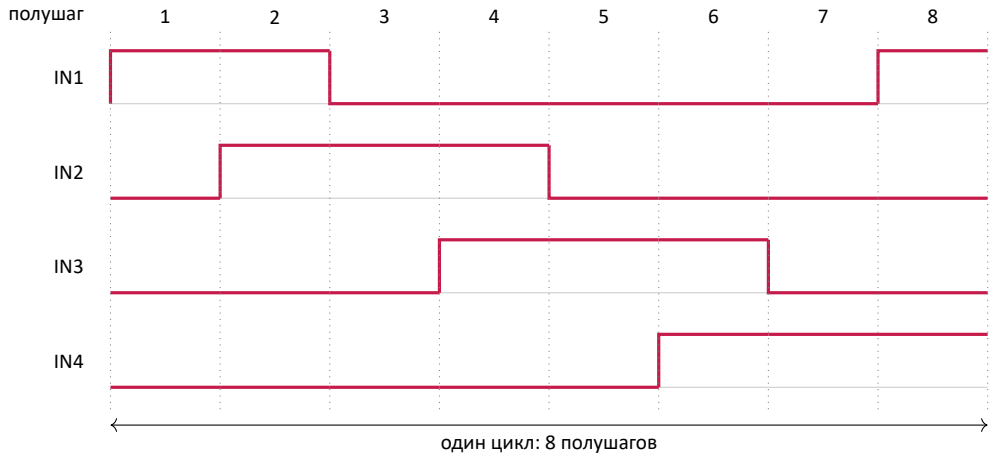
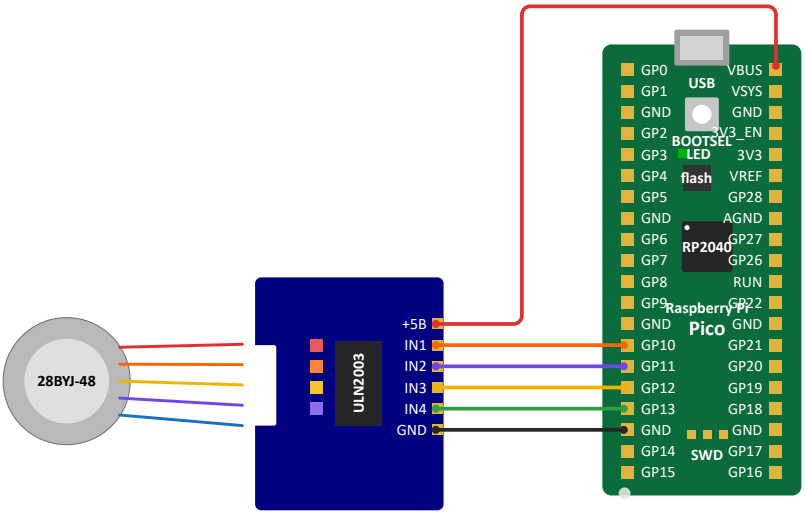


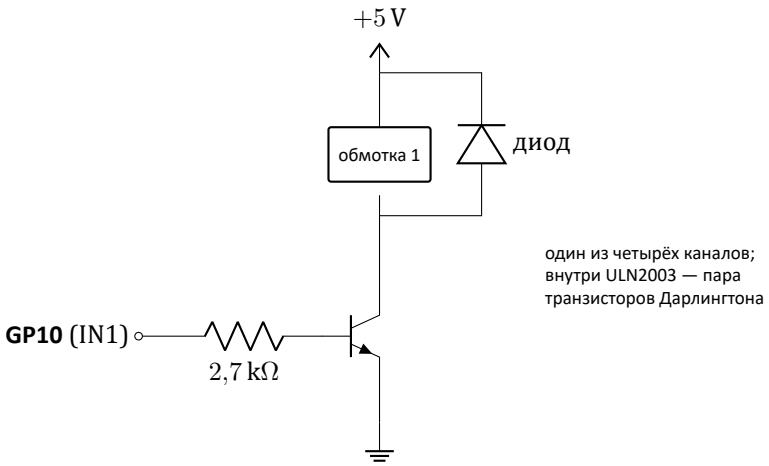
Рис. 20.12: Временная диаграмма сигналов на входах ULN2003 при вращении по часовой стрелке

Мотор 28BYJ-48 и плата ULN2003

Самый доступный шаговый двигатель для учебных проектов — **28BYJ-48** на $5\,\text{V}$. У него встроенный редуктор 1:64 (точнее, около 1:63,7), поэтому в полушаговом режиме вал делает **примерно 4096 полушагов на оборот** (в полношаговом — около 2048). Редуктор даёт большой момент, но скорость невелика. Сопротивление обмотки около $50\,\Omega$, ток одной обмотки $5/50 \approx 100\,\text{mA}$, двух вместе — около $200\,\text{mA}$: для вывода Pico это слишком много, поэтому между платой и мотором ставят плату-драйвер на микросхеме **ULN2003**. Внутри неё семь «ключей на транзисторах Дарлингтона» с защитными диодами; мы используем четыре.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 20.13: Шаговый двигатель 28BYJ-48 с платой ULN2003, выводы GP10–GP13

Осторожно

На плате ULN2003 есть перемычка питания и светодиоды индикации; порядок и подписи выводов («IN1–IN4», «+», «-») у разных плат могут отличаться — сверяйтесь с надписями. От порта USB компьютера двигатель питать можно (около 200 mA), но надёжнее взять отдельный источник 5 V; в этом случае **обязательно соедините его минус с GND платы Pico**. Двигатель 28BYJ-48 в версии на 12 V работает от другого питания — смотрите надпись на корпусе.

Листинг 20.7: MicroPython: класс `Stepper`, полушаги

```
from machine import Pin
import time

# Полушаговая последовательность для 28BYJ-48: 8 состояний IN1..IN4
HALF_STEP = (
    (1, 0, 0, 0),
    (1, 1, 0, 0),
    (0, 1, 0, 0),
    (0, 1, 1, 0),
    (0, 0, 1, 0),
```

```
    (0, 0, 1, 1),
    (0, 0, 0, 1),
    (1, 0, 0, 1),
)
STEPS_PER_REV = 4096         # полушагов на оборот выходного вала (приблизительно)

class Stepper:
    def __init__(self, pins):
        self.pins = [Pin(n, Pin.OUT, value=0) for n in pins]
        self.phase = 0

    def _apply(self):
        for pin, level in zip(self.pins, HALF_STEP[self.phase]):
            pin.value(level)

    def step(self, count, delay_ms=2):
        """count > 0 --- по часовой, count < 0 --- против."""
        direction = 1 if count > 0 else -1
        for _ in range(abs(count)):
            self.phase = (self.phase + direction) % 8   # номер фазы ходит по кругу 0...7
            self._apply()
            time.sleep_ms(delay_ms)

    def release(self):
        """Обесточить обмотки: мотор не греется, но и не держит вал."""
        for pin in self.pins:
            pin.value(0)

motor = Stepper((10, 11, 12, 13))
try:
    motor.step(STEPS_PER_REV)        # полный оборот в одну сторону
    time.sleep(0.5)
    motor.step(-STEPS_PER_REV // 4)  # четверть оборота назад
finally:
    motor.release()
```

Листинг 20.8: CircuitPython: то же

```
import time
import board
import digitalio

# Полушаговая последовательность для 28BYJ-48: 8 состояний IN1..IN4
HALF_STEP = (
    (1, 0, 0, 0),
    (1, 1, 0, 0),
    (0, 1, 0, 0),
    (0, 1, 1, 0),
    (0, 0, 1, 0),
    (0, 0, 1, 1),
    (0, 0, 0, 1),
    (1, 0, 0, 1),
)
STEPS_PER_REV = 4096         # полушагов на оборот выходного вала (приблизительно)

class Stepper:
    def __init__(self, pins):
        self.pins = []
```

```
        for p in pins:
            io = digitalio.DigitalInOut(p)
            io.direction = digitalio.Direction.OUTPUT
            io.value = False
            self.pins.append(io)
        self.phase = 0

    def _apply(self):
        for io, level in zip(self.pins, HALF_STEP[self.phase]):
            io.value = bool(level)

    def step(self, count, delay=0.002):
        """count > 0 --- по часовой, count < 0 --- против."""
        direction = 1 if count > 0 else -1
        for _ in range(abs(count)):
            self.phase = (self.phase + direction) % 8   # номер фазы ходит по кругу 0...7
            self._apply()
            time.sleep(delay)

    def release(self):
        """Обесточить обмотки: мотор не греется, но и не держит вал."""
        for io in self.pins:
            io.value = False

motor = Stepper((board.GP10, board.GP11, board.GP12, board.GP13))
try:
    motor.step(STEPS_PER_REV)        # полный оборот в одну сторону
    time.sleep(0.5)
    motor.step(-STEPS_PER_REV // 4)  # четверть оборота назад
finally:
    motor.release()
```

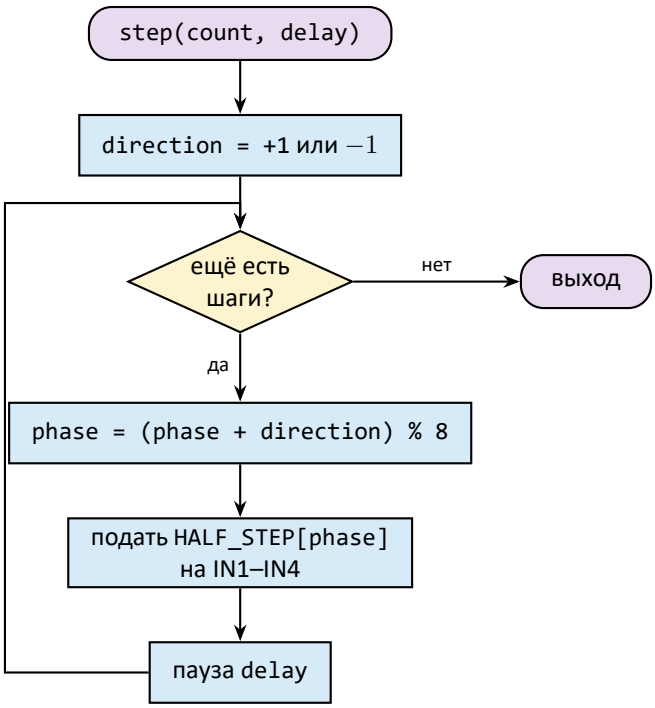


Рис. 20.14: Блок-схема метода `step()`: номер фазы ходит по кругу от 0 до 7

Главная хитрость — строка `phase = (phase + direction) % 8`. Операция `%` (остаток от деления)

замыкает числа 0–7 в кольцо: после 7 снова идёт 0, а при вращении назад после 0 получается 7 (в Python остаток от деления на положительное число никогда не бывает отрицательным). Скорость определяется паузой между шагами: при 2 ms полный оборот занимает около 8 секунд. Если паузу слишком сократить (меньше 1–1,5 мс), мотор начнёт пропускать шаги и дрожать на месте.

Совет

После остановки вызывайте `release()`: удерживаемые обмотки греют двигатель и впустую расходуют батарейки. Но учтите: без тока вал можно повернуть рукой, а положение «забудется». Для роботов, которые должны держать позицию, ток оставляют.

20.9 Сервопривод

Сервопривод (SG90, MG90S) — мотор с редуктором, датчиком положения и схемой управления внутри. Ему не нужен драйвер: угол поворота задаётся *шириной импульса* на сигнальном проводе, а ШИМ с частотой 50 Hz (период 20 ms). Подробности ШИМ вы разбирали в уроке 5; здесь важно подключение: у сервопривода три провода — коричневый (GND), красный (+5 В) и оранжевый (сигнал).

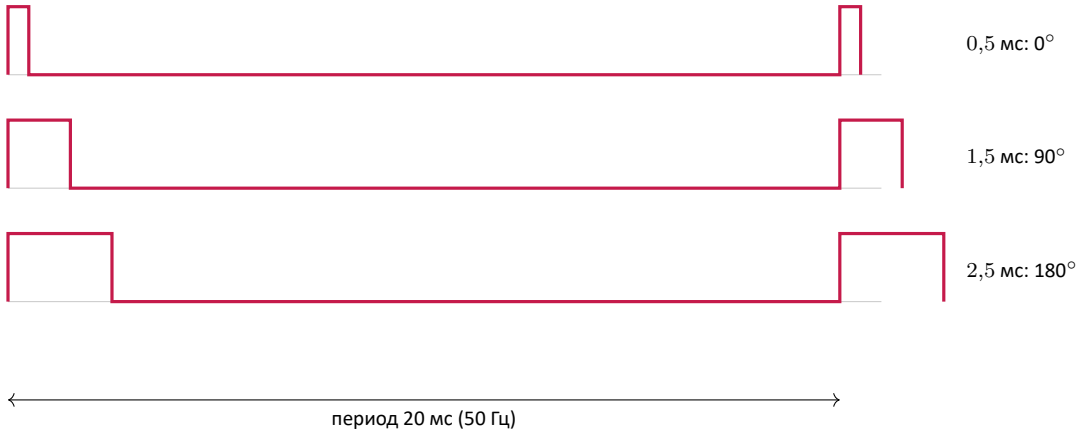


Рис. 20.15: Ширина импульса задаёт угол сервопривода (типичные значения; у разных моделей диапазон 0,5 – 2,5 ms может отличаться)

Листинг 20.9: MicroPython: сервопривод на GP15

```
from machine import Pin, PWM
import time

# Сервопривод SG90: сигнал на GP15, питание +5 В отдельно, общая земля с Pico
servo = PWM(Pin(15))
servo.freq(50)                      # период 20 мс

def angle(deg):
    """Угол 0...180 градусов -> импульс 0,5...2,5 мс -> код заполнения."""
    deg = max(0, min(180, deg))
    pulse_us = 500 + deg * 2000 / 180        # микросекунды
    servo.duty_u16(int(pulse_us / 20000 * 65535))

try:
    for a in (0, 90, 180, 90):
        angle(a)
        time.sleep(1)
```

```
finally:
    servo.duty_u16(0)               # нет импульсов --- сервопривод расслаблен
    servo.deinit()
```

Листинг 20.10: CircuitPython: то же

```
import time
import board
import pwmio

# Сервопривод SG90: сигнал на GP15, питание +5 В отдельно, общая земля с Pico
servo = pwmio.PWMOut(board.GP15, duty_cycle=0, frequency=50)   # период 20 мс

def angle(deg):
    """Угол 0...180 градусов -> импульс 0,5...2,5 мс -> код заполнения."""
    deg = max(0, min(180, deg))
    pulse_us = 500 + deg * 2000 / 180        # микросекунды
    servo.duty_cycle = int(pulse_us / 20000 * 65535)

# Готовый вариант: библиотека adafruit_motor (circup install adafruit_motor)
#   from adafruit_motor import servo as am_servo
#   s = am_servo.Servo(servo, min_pulse=500, max_pulse=2500)
#   s.angle = 90

try:
    for a in (0, 90, 180, 90):
        angle(a)
        time.sleep(1)
finally:
    servo.duty_cycle = 0            # нет импульсов --- сервопривод расслаблен
    servo.deinit()
```

Осторожно

Сервопривод тоже потребляет пик 500 – 700 mA при нагрузке. Питайте его от отдельных 5 V, а не от 3V3, и соедините землю с землёй Pico. Сигнальный вывод Pico ($3{,}3$ V) сервопривод с питанием 5 V обычно воспринимает нормально. При шуме и «дрожании» сервопривода добавьте конденсатор 470 µF по питанию.

20.10 Питание и общая земля

Типичная ошибка: «подключил мотор, всё перезагружается». Причины две: общий источник питания проседает из-за тока мотора, либо отсутствует общая земля.

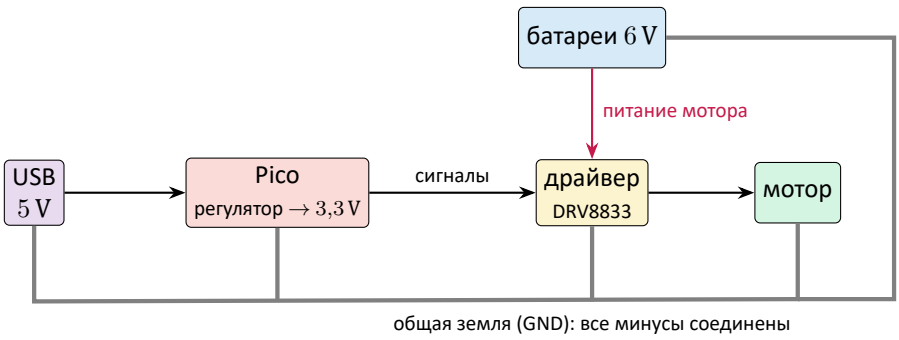


Рис. 20.16: Питание мотора отдельное, земля одна на всех

- **Раздельное питание.** Мотор и драйвер — от батарей или блока питания. Pico — от USB.
- **Общая земля.** Без неё сигнал с вывода Pico не с чем сравнивать: драйвер не поймёт, где логический ноль. Соедините минус внешнего питания с любым выводом GND платы.
- **Не подавайте напряжение больше $5\,\mathrm{V}$ на любой вывод Pico.** Питание мотора ($6\,\mathrm{V}$, $9\,\mathrm{V}$, $12\,\mathrm{V}$) должно идти только на вывод VM драйвера, но никогда на выводы GPIO и `3V3`.
- **Проводам — сечение.** Силовые провода короткие и толстые; тонкие перемычки для большого тока греются.

20.11 Помехи и конденсаторы

Коллекторный мотор — настоящий источник радиопомех: щётки искрят, ток скачет. Помехи по общему питанию попадают в Pico: «глючит» АЦП, сбиваются показания датчиков, иногда перезагружается плата. Борьба с ними простая:

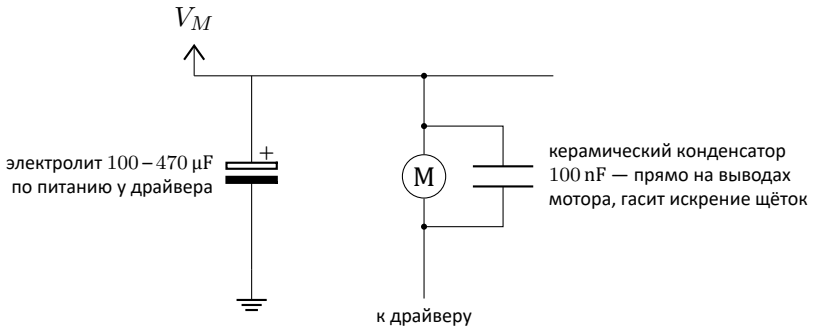


Рис. 20.17: Конденсаторы для борьбы с помехами моторов

- керамический конденсатор $100\,\mathrm{nF}$ припаивают *прямо к выводам мотора*;
- электролитический конденсатор $100 – 470\,\mu\mathrm{F}$ ставят рядом с драйвером по цепи питания мотора (длинная ножка — к плюсу!);
- питание мотора и питание Pico разделяют, а провода мотора держат подальше от проводов датчиков;
- в программе не переключайте направление «с полного хода»: сначала плавный спад скорости или торможение, пауза и только потом реверс.

20.12 Практика: машина-гусеничка

Соберите схему с двумя моторами: на два канала DRV8833 (GP10, GP11 — левый, GP12, GP13 — правый) и напишите функции `forward()`, `back()`, `left()`, `right()`, используя класс `Motor` из урока. Поворот на месте получается, если моторы вращаются в разные стороны. Это заготовка для проекта в уроке 25.

Если не работает

- **Мотор не вращается, а светодиод платы мигает** — нет общей земли или минус батареи не соединён с GND платы.
- **Pico перезагружается при включении мотора** — мотор питается от той же линии, что и плата, либо без конденсатора; разделите питание, добавьте $100\,\mu\mathrm{F}$ и $100\,\mathrm{nF}$.
- **Мотор вращается сам или не выключается** — нет резистора $10\,\mathrm{k}\Omega$ между базой (затвором) и землёй; перепутаны выводы транзистора (К–Б–Э).

- **Транзистор сильно греется** — слишком мал ток базы (транзистор не насыщается) или нагрузка больше предельной; проверьте R_B, замените BC337 на MOSFET.
- **Реле не отпускает** — оптрон питается от 5 V, а логика $3{,}3$ V; подайте на VCC модуля $3{,}3$ V.
- **Шаговый двигатель дрожит, но не вращается** — перепутан порядок проводов к IN1–IN4 или слишком короткая пауза между шагами.
- **Сервопривод дёргается** — не хватает тока питания; добавьте отдельный источник 5 V и общую землю.
- **Диод греется или питание «коротит»** — диод включён не той полярностью; полоска к плюсу.

Итоги урока

- Вывод GPIO отдаёт единицы миллиампер; мотору, реле и шаговому двигателю нужен ключ и отдельное питание.
- Индуктивная нагрузка даёт выброс напряжения: защитный диод параллельно катушке, полоска к плюсу.
- Для BC337: $R_B = (U_{GPIO} - 0{,}7)/I_B$, $I_B \approx 2{,}5\, I_C/h_{FE}$; для MOSFET — резистор затвора и подтяжка к земле, только logic-level.
- H-мост управляется двумя входами: ШИМ на одном даёт вращение вперёд, на другом — назад; оба в «1» — торможение.
- Шаговый двигатель 28BYJ-48 с ULN2003: 8 полушагов, около 4096 полушагов на оборот; `% 8` замыкает номер фазы в кольцо.
- Земля у всех источников общая; 220 V не подключаем.

Проверь себя

1. Почему мотор нельзя подключить прямо к выводу GPIO? Какой ток потребует заторможенный мотор с сопротивлением $3\,\Omega$ при 3 V?
2. Что такое защитный диод, как его включают и зачем он нужен?
3. Как рассчитать резистор базы транзистора, если нагрузка потребляет 200 mA, а $h_{FE\,\min} = 100$?
4. Как с помощью H-моста заставить мотор вращаться назад и как его быстро остановить?
5. Зачем нужна общая земля и что произойдёт, если её не будет?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите схему с BC337 и мотором. Запустите программу плавного разгона и измерьте напряжение на моторе мультиметром на разных значениях скорости.
2. ★ Рассчитайте резистор базы для нагрузки 250 mA при $h_{FE\,\min} = 100$ и запасе 2,5. Хватит ли этого тока от вывода Pico? Какой транзистор лучше выбрать?
3. ★★ Добавьте кнопку из урока 3: по нажатию мотор переключается между скоростями 0, 40, 70 и 100 %.

4. ★★ Измените программу шагового двигателя так, чтобы он поворачивал вал ровно на $90°$, затем на $-45°$. Сколько полушагов нужно?
5. ★★★ Сделайте «радар»: сервопривод медленно поворачивается от $0°$ до $180°$ и обратно (шагом по $2°$) без `sleep()` длиннее 20 мс.
6. ★★★★ Соберите двухмоторную тележку на DRV8833 с управлением через консоль (команды `w`, `a`, `s`, `d`); реализуйте мягкий старт и защиту: если мотор был включён более 5 секунд подряд, программа делает паузу на остывание.

Урок 21. Звук и музыка

Цель	понять, что такое звук и нота, научить Pico играть тоны и мелодии через ШИМ, регулировать громкость, собрать «пианино» из кнопок и узнать, как проигрывается настоящий цифровой звук (WAV, I2S)
Опирается на	урок 5: ШИМ, частота и скважность; урок 3: кнопки; урок 4: отсчёт времени без `sleep()`
Результат	пьезоизлучатель на **GP13** играет гамму и «Оду к радости»; четыре кнопки образуют мини-пианино; мелодия играет в фоне, пока программа делает другие дела; понятно, как подключить динамик и усилитель
Понадобится	пассивный пьезоизлучатель (зуммер без генератора), 4 тактовые кнопки, провода; по желанию: транзистор NPN (например, 2N2222 или BC337), динамик $8\,\Omega$, диод 1N4148, модуль усилителя PAM8403 или I2S-усилитель MAX98357A

21.1 Идея урока коротко

Звук — это колебания воздуха. Чем чаще они повторяются, тем *выше* тон; чем сильнее — тем *громче*. Микроконтроллер умеет очень быстро включать и выключать вывод. Если делать это 440 раз в секунду и подключить к выводу пьезоизлучатель, он будет колебаться с частотой $440\,\text{Hz}$ и мы услышим ноту «ля». Значит, **высота ноты — это частота ШИМ**, а **громкость — его скважность**. Всё остальное в уроке — способы красиво организовать такие тоны в мелодию и способы получить звук получше, чем «писк».

21.2 Что такое звук

Когда натягивают струну, бьют по барабану или двигают мембрану динамика, воздух рядом с ней то сжимается, то разрежается. Эти волны давления доходят до уха, и мы слышим звук. Простейшее колебание — синусоида. У неё две главные характеристики:

- **частота** f (герцы, Гц) — сколько полных колебаний происходит за секунду; период $T = 1/f$;
- **амплитуда** — размах колебания; от неё зависит громкость.

Человек слышит примерно от $20\,\text{Hz}$ до $20\,\text{kHz}$ (с возрастом верхняя граница опускается). Нота «ля» первой октавы, по которой настраивают оркестр, имеет частоту $440\,\text{Hz}$, период $T = 1/440 \approx 2{,}27$ мс. На рис. 21.1 показаны два тона: нота «ля» и нота на октаву выше.

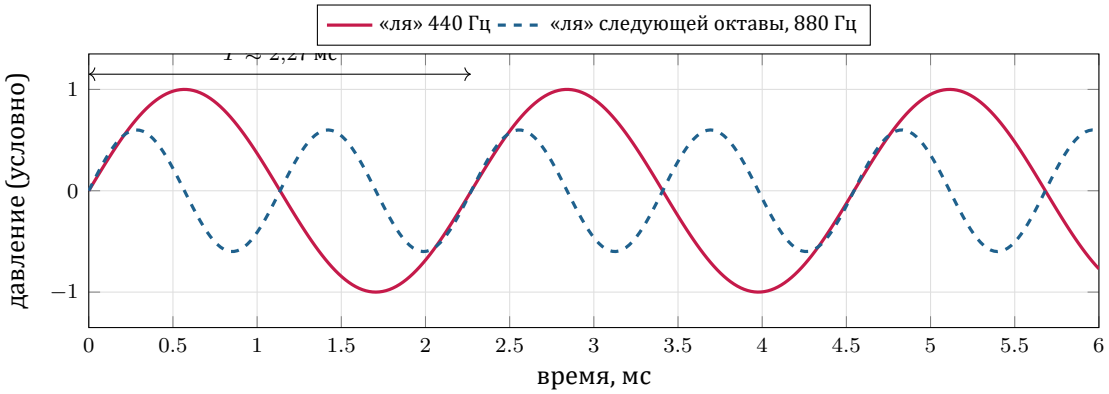


Рис. 21.1: Два тона: частота 880 Гц вдвое выше, и период вдвое короче. Меньшая амплитуда второго тона означает более тихий звук

21.3 Ноты и их частоты

Музыкальный строй устроен математически красиво. Октава — это удвоение частоты: «ля» первой октавы 440 Hz, «ля» второй — 880 Hz, «ля» малой октавы — 220 Hz. Каждая октава делится на **12 полутонов**, и каждый следующий полутон выше предыдущего в одно и то же число раз q. Через 12 шагов частота должна удвоиться, поэтому $q^{12} = 2$, то есть $q = 2^{1/12} \approx 1{,}0595$. Частота ноты, отстоящей от «ля» на n полутонов:

$$f = 440 \cdot 2^{n/12}.$$

В MIDI каждой ноте присвоен номер: «ля» первой октавы — 69, «до» первой октавы — 60. Тогда $f = 440 \cdot 2^{(m-69)/12}$, где m — номер ноты. Эту формулу использует программа `midi_freq()` ниже.

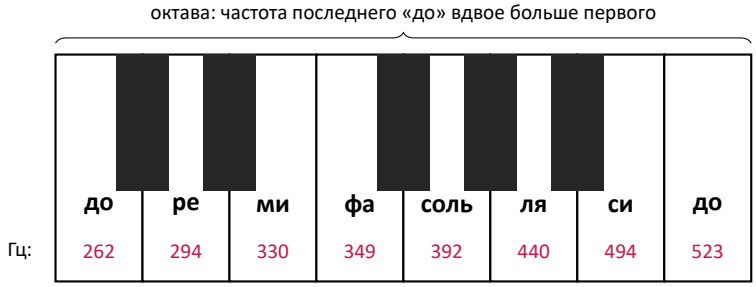


Рис. 21.2: Клавиатура одной октавы и частоты нот (округлены до целых герц)

Нота	Обозначение	Номер MIDI	n от «ля»	Частота, Гц	В программе
до	C4	60	-9	261,63	262
ре	D4	62	-7	293,66	294
ми	E4	64	-5	329,63	330
фа	F4	65	-4	349,23	349
соль	G4	67	-2	392,00	392
ля	A4	69	0	440,00	440
си	B4	71	$+2$	493,88	494
до (выше)	C5	72	$+3$	523,25	523

Таблица 21.1: Частоты нот первой октавы (равномерно темперированный строй)

Если построить частоту в зависимости от номера полутона, получится показательная кривая (рис. 21.3): на слух мы воспринимаем не разность частот, а их *отношение*. Поэтому гамма «звучит ровно», хотя шаги в герцах растут.

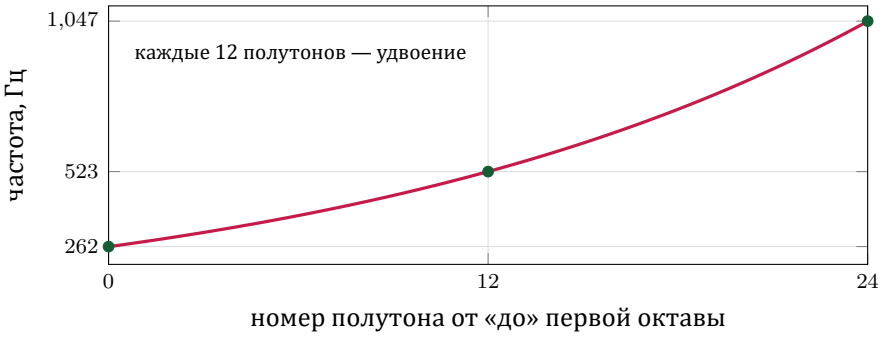


Рис. 21.3: Частота ноты растёт по показательному закону: $f = 261{,}63 \cdot 2^{m/12}$

21.4 Чем издать звук: активный и пассивный излучатель

Для опытов чаще всего берут **пьезоизлучатель** — керамический диск, приклеенный к металлической пластине. Если подать на него напряжение, диск слегка изгибается; если напряжение меняется, пластина колеблется и звучит. Бывают два вида.

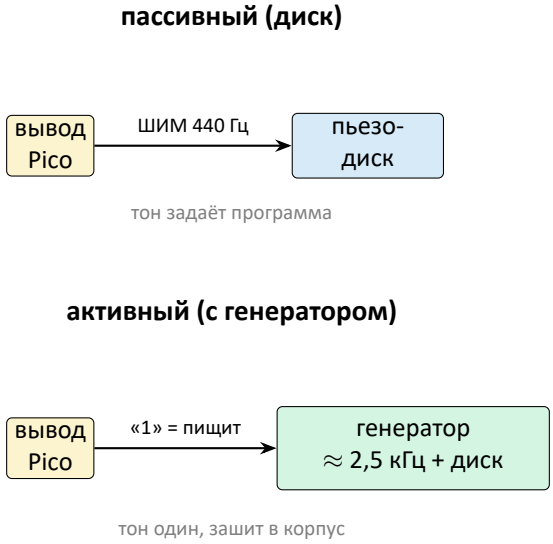


Рис. 21.4: Пассивный излучатель делает ровно то, что требует программа, активный — только «пищит»

	Пассивный (диск, «зуммер без генератора»)	Активный (со встроенным генератором)
Как звучит	любой тон, который подаёт программа	один фиксированный тон
Как включать	ШИМ нужной частоты	просто `1` или `0`
Ток	очень маленький (ёмкостная нагрузка)	обычно 10–30 мА: нужен транзистор
Использование	мелодии, сигналы разной высоты	«пищалка» тревоги, подтверждение нажатия

Таблица 21.2: Два вида пьезоизлучателей

Совет

Как отличить? У активного излучателя на корпусе часто есть знак «+» и он звучит, если просто подать питание; у пассивного (плоский диск) — нет. Для этого урока нужен именно

пассивный: активный сыграет вместо мелодии одну и ту же ноту с разными паузами. Пассивные диски почти не имеют полярности. Резонанс у них обычно в районе 2–4 кГц, там они самые громкие.

21.5 Тон средствами ШИМ

ШИМ (урок 5) выдаёт прямоугольные импульсы заданной частоты. Нам нужна не «яркость», а сама частота импульсов: она и становится высотой тона. Скважность (доля времени, когда сигнал равен единице) определяет громкость и тембр. Именно на скважности 50 % прямоугольная волна самая «сильная».

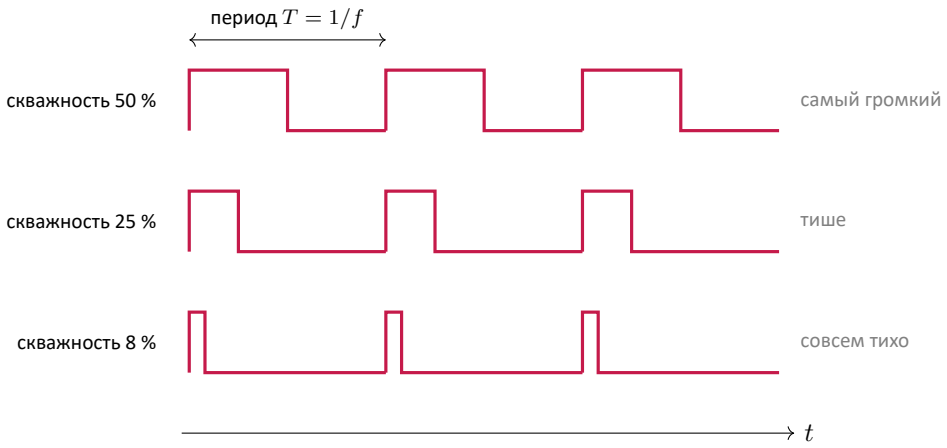


Рис. 21.5: Одна и та же частота, разная скважность: громкость меняется, высота тона нет

Первая программа играет четыре ноты. Функция `tone()` — «одна нота»: задать частоту, включить, подождать, выключить.

Листинг 21.1: MicroPython: функция tone() и четыре ноты

```
from machine import Pin, PWM
import time

buzzer = PWM(Pin(13))            # пьезоизлучатель на GP13

def tone(freq, ms, duty=0.5):
    """Играет ноту частотой freq (Гц) в течение ms миллисекунд."""
    buzzer.freq(freq)                    # частота ШИМ = частота звука
    buzzer.duty_u16(int(65535 * duty))   # скважность 50 % --- самый громкий звук
    time.sleep_ms(ms)
    buzzer.duty_u16(0)                   # тишина: на выходе постоянный ноль

for f in (262, 330, 392, 523):           # до, ми, соль, до следующей октавы
    tone(f, 300)
    time.sleep_ms(50)

buzzer.deinit()                          # освобождаем ШИМ-канал
```

Листинг 21.2: CircuitPython: то же

```
import time
import board
import pwmio

# variable_frequency=True разрешает менять частоту после создания
buzzer = pwmio.PWMOut(board.GP13, frequency=440, duty_cycle=0,
                      variable_frequency=True)

def tone(freq, seconds, duty=0.5):
    """Играет ноту частотой freq (Гц) в течение seconds секунд."""
    buzzer.frequency = freq                    # частота ШИМ = частота звука
    buzzer.duty_cycle = int(65535 * duty)      # 50 % --- самый громкий звук
    time.sleep(seconds)
    buzzer.duty_cycle = 0                      # тишина

for f in (262, 330, 392, 523):                 # до, ми, соль, до следующей октавы
    tone(f, 0.3)
    time.sleep(0.05)

buzzer.deinit()
```

Отличия: в MicroPython частоту меняет метод `freq()`, скважность — `duty_u16()`; в CircuitPython это свойства `frequency` и `duty_cycle`, причём для смены частоты при создании нужно указать `variable_frequency=True`. Диапазон скважности в обоих языках — от 0 до 65535.

Осторожно

- Выводы одного ШИМ-среза (например, GP12 и GP13) *делят одну частоту*: если на GP13 играет мелодия, соседний с ним в срезе GP12 нельзя использовать как ШИМ с другой частотой (урок 5). Для кнопок это не мешает, они обычные цифровые входы.
- Не подключайте к GPIO динамик 8 Ω напрямую: ток будет в десятки раз выше допустимого для вывода.
- Слишком громкий писк в наушниках и рядом с ухом вреден. Для опытов хватает звука на расстоянии вытянутой руки.

21.6 Гамма и номера нот

Вместо таблицы частот программа может вычислять частоту по формуле: нужны только номера нот и функция `midi_freq()`. Мажорная гамма строится по шаблону полутонов: 0, 2, 4, 5, 7, 9, 11, 12 (два целых тона, полутон, три тона, полутон). Если сдвигать начало гаммы на одну и ту же величину, получится гамма от любой ноты.

Листинг 21.3: MicroPython: гамма от «до» по формуле

```
from machine import Pin, PWM
import time

buzzer = PWM(Pin(13))

def midi_freq(n):
    """Частота ноты с номером MIDI n: ля первой октавы (A4) --- номер 69 --- 440 Гц."""
    return round(440 * 2 ** ((n - 69) / 12))

```

```
# мажорная гамма от до: интервалы в полутонах 0 2 4 5 7 9 11 12
STEPS = (0, 2, 4, 5, 7, 9, 11, 12)
C4 = 60                                  # номер MIDI ноты до первой октавы

for step in STEPS:
    f = midi_freq(C4 + step)
    print(C4 + step, f, "Гц")
    buzzer.freq(f)
    buzzer.duty_u16(32768)
    time.sleep_ms(350)
    buzzer.duty_u16(0)
    time.sleep_ms(50)

buzzer.deinit()
```

Листинг 21.4: CircuitPython: гамма от «до» по формуле

```
import time
import board
import pwmio

buzzer = pwmio.PWMOut(board.GP13, frequency=440, duty_cycle=0,
                      variable_frequency=True)

def midi_freq(n):
    """Частота ноты с номером MIDI n: ля первой октавы (A4) --- номер 69 --- 440 Гц."""
    return round(440 * 2 ** ((n - 69) / 12))

# мажорная гамма от до: интервалы в полутонах 0 2 4 5 7 9 11 12
STEPS = (0, 2, 4, 5, 7, 9, 11, 12)
C4 = 60                                  # номер MIDI ноты до первой октавы

for step in STEPS:
    f = midi_freq(C4 + step)
    print(C4 + step, f, "Гц")
    buzzer.frequency = f
    buzzer.duty_cycle = 32768
    time.sleep(0.35)
    buzzer.duty_cycle = 0
    time.sleep(0.05)

buzzer.deinit()
```

21.7 Мелодии: список нот и длительностей

Мелодию удобно хранить как **список пар** «нота — длительность». Длительность измеряют в долях четверти: 1 — четверть, 2 — половинная, 0,5 — восьмая, 4 — целая. А чтобы мелодию можно было ускорять и замедлять, вводят **темп** — число четвертей в минуту (BPM, beats per minute). Одна четверть длится $60\,000/\text{BPM}$ мс: при 120 BPM — 500 мс, при 60 BPM — 1 с.

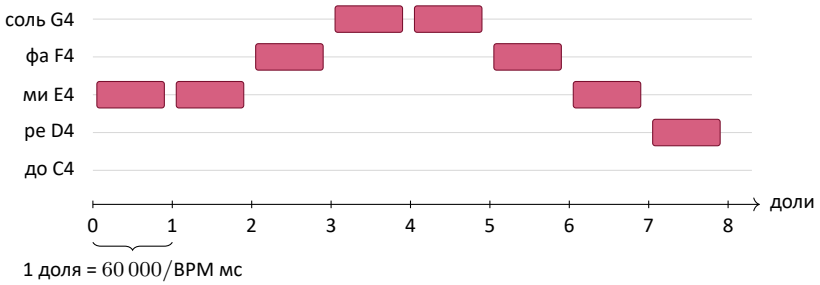


Рис. 21.6: Начало «Оды к радости» как «нотный стан»: высота по вертикали, длительность — ширина прямоугольника

Между соседними нотами нужен короткий зазор: иначе две одинаковые ноты подряд сольются в одну длинную. Поэтому нота звучит, например, 90 % своей длительности, а оставшиеся 10 % — тишина. Пауза в мелодии — это просто «нота» с частотой 0, во время которой скважность равна нулю.

Листинг 21.5: MicroPython: «Ода к радости» (темп 120)

```
from machine import Pin, PWM
import time

buzzer = PWM(Pin(13))

# частоты нот, Гц; "P" --- пауза
NOTES = {"C4": 262, "D4": 294, "E4": 330, "F4": 349,
         "G4": 392, "A4": 440, "B4": 494, "C5": 523, "P": 0}

# «Ода к радости»: (нота, длительность в долях четверти)
ODE = [("E4", 1), ("E4", 1), ("F4", 1), ("G4", 1),
       ("G4", 1), ("F4", 1), ("E4", 1), ("D4", 1),
       ("C4", 1), ("C4", 1), ("D4", 1), ("E4", 1),
       ("E4", 1.5), ("D4", 0.5), ("D4", 2)]

BPM = 120                                  # темп: четвертей в минуту
BEAT_MS = 60_000 // BPM                    # длина одной четверти, мс

def play(melody):
    for name, beats in melody:
        ms = int(BEAT_MS * beats)
        f = NOTES[name]
        if f:
            buzzer.freq(f)
            buzzer.duty_u16(32768)
            time.sleep_ms(int(ms * 0.9))   # звучит 90 % времени
            buzzer.duty_u16(0)
            time.sleep_ms(ms - int(ms * 0.9))   # короткий зазор между нотами
        else:
            buzzer.duty_u16(0)
            time.sleep_ms(ms)

play(ODE)
buzzer.deinit()
```

Листинг 21.6: CircuitPython: «Ода к радости» (темп 120)

```
import time
import board
import pwmio

buzzer = pwmio.PWMOut(board.GP13, frequency=440, duty_cycle=0,
                      variable_frequency=True)

# частоты нот, Гц; "P" --- пауза
NOTES = {"C4": 262, "D4": 294, "E4": 330, "F4": 349,
         "G4": 392, "A4": 440, "B4": 494, "C5": 523, "P": 0}

# «Ода к радости»: (нота, длительность в долях четверти)
ODE = [("E4", 1), ("E4", 1), ("F4", 1), ("G4", 1),
       ("G4", 1), ("F4", 1), ("E4", 1), ("D4", 1),
       ("C4", 1), ("C4", 1), ("D4", 1), ("E4", 1),
       ("E4", 1.5), ("D4", 0.5), ("D4", 2)]

BPM = 120                                   # темп: четвертей в минуту
BEAT = 60 / BPM                             # длина одной четверти, с

def play(melody):
    for name, beats in melody:
        length = BEAT * beats
        f = NOTES[name]
        if f:
            buzzer.frequency = f
            buzzer.duty_cycle = 32768
            time.sleep(length * 0.9)        # звучит 90 % времени
            buzzer.duty_cycle = 0
            time.sleep(length * 0.1)        # короткий зазор между нотами
        else:
            buzzer.duty_cycle = 0
            time.sleep(length)

play(ODE)
buzzer.deinit()
```

Это интересно

Музыкальные произведения, авторские права на которые закончились (Бетховен, Моцарт, народные песни), можно свободно записывать в виде таких списков. С современными песнями так не получится: нотная запись — тоже объект авторского права.

21.8 Громкость: скважность

Пассивный излучатель не имеет регулятора громкости, но громкость можно менять скважностью. Когда скважность близка к нулю, импульсы короткие и энергии в них мало; когда близка к 50 % — импульсы «полные». Дальше 50 % громкость не растёт, а снова падает (зеркально: скважность 90 % звучит как 10 %). Поэтому «регулятор громкости 0 … 10» превращается в скважность от 0 до 50 %. Реальная зависимость нелинейная, и у каждого излучателя свои особенности; в программе удобно собрать её в одну функцию.

Листинг 21.7: MicroPython: громкость от 0 до 10

```
from machine import Pin, PWM
import time

buzzer = PWM(Pin(13))
buzzer.freq(2700)                # у многих пьезоизлучателей резонанс около 2--4 кГц

def set_volume(level):
    """level от 0 (тишина) до 10 (максимум): скважность 0...50 %."""
    duty = 32768 * level // 10    # 50 % от 65535 --- это 32768
    buzzer.duty_u16(duty)

for level in (1, 2, 4, 6, 8, 10, 6, 3, 1, 0):
    print("громкость", level)
    set_volume(level)
    time.sleep_ms(600)

buzzer.deinit()
```

Листинг 21.8: CircuitPython: громкость от 0 до 10

```
import time
import board
import pwmio

buzzer = pwmio.PWMOut(board.GP13, frequency=2700, duty_cycle=0)
# у многих пьезоизлучателей резонанс около 2--4 кГц

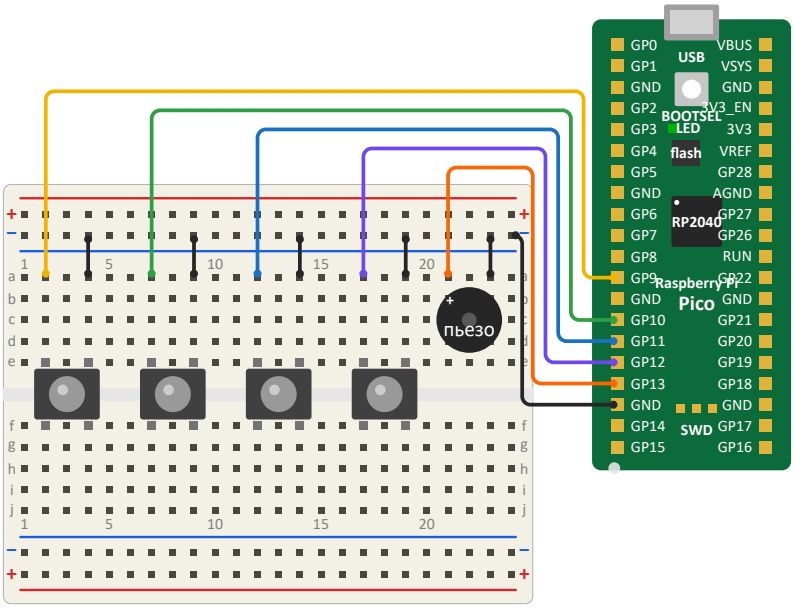
def set_volume(level):
    """level от 0 (тишина) до 10 (максимум): скважность 0...50 %."""
    buzzer.duty_cycle = 32768 * level // 10   # 50 % от 65535 --- это 32768

for level in (1, 2, 4, 6, 8, 10, 6, 3, 1, 0):
    print("громкость", level)
    set_volume(level)
    time.sleep(0.6)

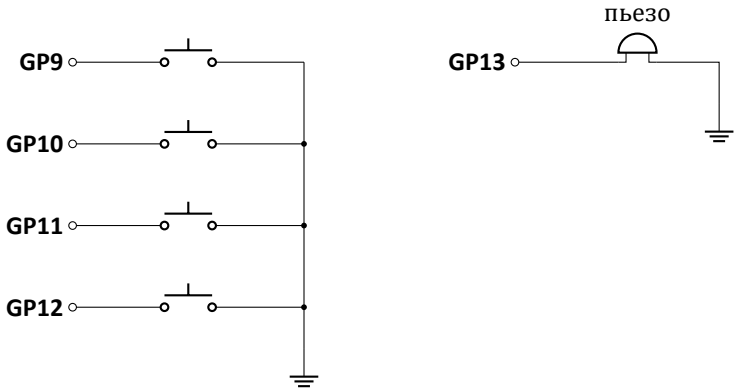
buzzer.deinit()
```

21.9 Монтажная схема: зуммер и кнопки-«пианино»

Подключим пассивный излучатель к **GP13**: «плюс» на GP13, «минус» на GND. Четыре тактовые кнопки соединим с GP9–GP12 с другой стороны на землю: когда кнопка нажата, вход соединяется с GND, и программа видит ноль (кнопка с подтяжкой, урок 3). Внешние резисторы не нужны: используется внутренняя подтяжка к 3,3 В.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 21.7: Мини-пианино: четыре кнопки (GP9–GP12) и пьезоизлучатель (GP13)

Обратите внимание на полярность и провода: каждый цвет — отдельный сигнал; чёрные провода — земля. Кнопки на макетной плате обязательно ставят «поперёк» центральной канавки, а подключают только одну её сторону (верхнюю), потому что вторая сторона нам не нужна.

Листинг 21.9: MicroPython: мини-пианино

```
from machine import Pin, PWM
import time

# четыре кнопки на GP9...GP12 (между выводом и GND), внутренняя подтяжка к 3,3 В
KEYS = {9: 262, 10: 330, 11: 392, 12: 523}      # GP: частота ноты (до, ми, соль, до)
buttons = {gp: Pin(gp, Pin.IN, Pin.PULL_UP) for gp in KEYS}
buzzer = PWM(Pin(13))
buzzer.duty_u16(0)

while True:
    current = 0
    for gp, btn in buttons.items():
        if btn.value() == 0:                    # 0 = кнопка нажата
            current = KEYS[gp]                  # при нескольких кнопках --- последняя
    if current:
```

```
        buzzer.freq(current)
        buzzer.duty_u16(32768)
    else:
        buzzer.duty_u16(0)
    time.sleep_ms(5)
```

Листинг 21.10: CircuitPython: мини-пианино

```
import time
import board
import digitalio
import pwmio

# четыре кнопки на GP9...GP12 (между выводом и GND), внутренняя подтяжка к 3,3 В
KEYS = ((board.GP9, 262), (board.GP10, 330), (board.GP11, 392), (board.GP12, 523))

buttons = []
for pin, freq in KEYS:
    b = digitalio.DigitalInOut(pin)
    b.direction = digitalio.Direction.INPUT
    b.pull = digitalio.Pull.UP
    buttons.append((b, freq))

buzzer = pwmio.PWMOut(board.GP13, frequency=440, duty_cycle=0,
                      variable_frequency=True)

while True:
    current = 0
    for b, freq in buttons:
        if not b.value:                         # False = кнопка нажата
            current = freq                      # при нескольких кнопках --- последняя
    if current:
        buzzer.frequency = current
        buzzer.duty_cycle = 32768
    else:
        buzzer.duty_cycle = 0
    time.sleep(0.005)
```

21.10 Учимся программировать: классы и неблокирующий проигрыватель

Функции `play()` из предыдущих примеров *блокируют* программу: пока играет мелодия, нельзя опросить кнопки или мигнуть светодиодом. Решение известно из урока 2: проверять время в главном цикле. Теперь у нас много связанных данных (вывод, мелодия, текущая нота, момент окончания ноты) и несколько действий над ними (начать, обновить). Для этого в Python есть **классы**: «чертёж» объекта, который хранит данные (*поля*) и умеет выполнять действия (*методы*). Можно создать несколько объектов-проигрывателей с разными мелодиями, и они не помешают друг другу.

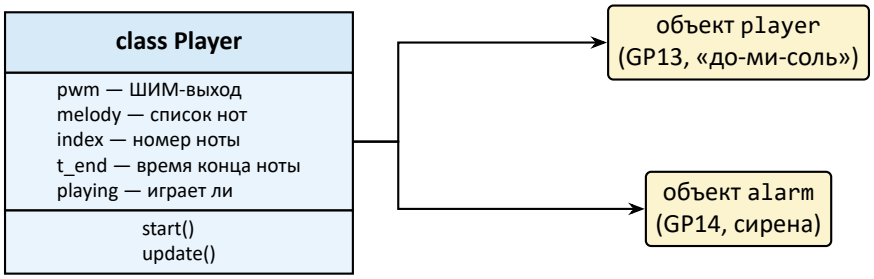


Рис. 21.8: Класс — чертёж, объекты — «изделия» по нему

Метод `update()` вызывают в каждом шаге главного цикла. Он смотрит на часы и, если пришло время, переходит к следующей ноте (рис. 21.9). Состояние хранится в полях объекта, поэтому при следующем вызове метод «помнит», на какой ноте остановился.

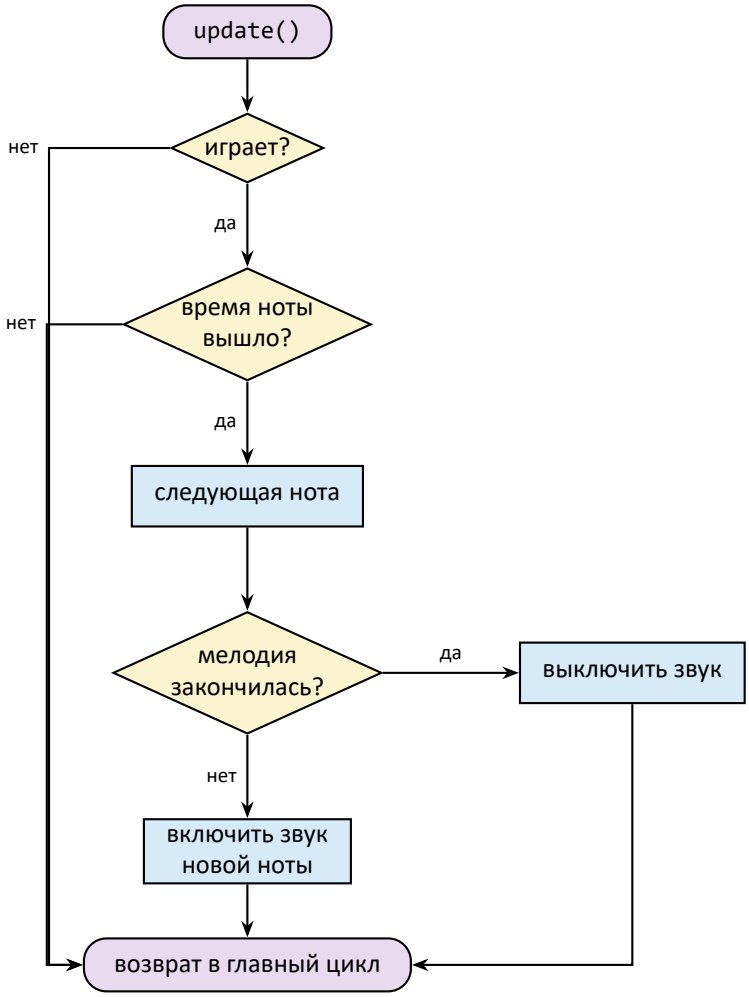


Рис. 21.9: Блок-схема метода `update()`: каждый вызов делает не больше одного небольшого шага

Листинг 21.11: MicroPython: класс Player и мигание светодиодом одновременно с мелодией

```
from machine import Pin, PWM
import time

NOTES = {"C4": 262, "D4": 294, "E4": 330, "F4": 349,
         "G4": 392, "A4": 440, "B4": 494, "C5": 523, "P": 0}

class Player:
    """Неблокирующий проигрыватель мелодии: вызывайте update() в главном цикле."""

    def __init__(self, pin, melody, bpm=120):
        self.pwm = PWM(Pin(pin))
        self.pwm.duty_u16(0)
        self.melody = melody
        self.beat_ms = 60_000 // bpm
        self.index = 0
        self.t_end = 0
        self.playing = False

    def start(self):
        self.index = 0
        self.playing = True
        self._begin_note()

    def _begin_note(self):
        name, beats = self.melody[self.index]
        f = NOTES[name]
        if f:
            self.pwm.freq(f)
            self.pwm.duty_u16(32768)
        else:
            self.pwm.duty_u16(0)
        self.t_end = time.ticks_add(time.ticks_ms(), int(self.beat_ms * beats))

    def update(self):
        if not self.playing:
            return
        if time.ticks_diff(time.ticks_ms(), self.t_end) >= 0:   # нота кончилась
            self.index += 1
            if self.index >= len(self.melody):
                self.playing = False
                self.pwm.duty_u16(0)
            else:
                self._begin_note()

song = [("C4", 1), ("E4", 1), ("G4", 1), ("C5", 2), ("P", 1), ("G4", 1), ("C5", 2)]
player = Player(13, song, bpm=140)
led = Pin("LED", Pin.OUT)          # во время музыки мигаем светодиодом
player.start()
t_led = time.ticks_ms()

while player.playing:
    player.update()
    if time.ticks_diff(time.ticks_ms(), t_led) >= 200:
        t_led = time.ticks_ms()
        led.toggle()
    # здесь могли бы опрашиваться кнопки, датчики, экран...
```

```

led.value(0)
```

Листинг 21.12: CircuitPython: то же

```
import time
import board
import digitalio
import pwmio

NOTES = {"C4": 262, "D4": 294, "E4": 330, "F4": 349,
         "G4": 392, "A4": 440, "B4": 494, "C5": 523, "P": 0}

class Player:
    """Неблокирующий проигрыватель мелодии: вызывайте update() в главном цикле."""

    def __init__(self, pin, melody, bpm=120):
        self.pwm = pwmio.PWMOut(pin, frequency=440, duty_cycle=0,
                                variable_frequency=True)
        self.melody = melody
        self.beat = 60 / bpm
        self.index = 0
        self.t_end = 0
        self.playing = False

    def start(self):
        self.index = 0
        self.playing = True
        self._begin_note()

    def _begin_note(self):
        name, beats = self.melody[self.index]
        f = NOTES[name]
        if f:
            self.pwm.frequency = f
            self.pwm.duty_cycle = 32768
        else:
            self.pwm.duty_cycle = 0
        self.t_end = time.monotonic() + self.beat * beats

    def update(self):
        if not self.playing:
            return
        if time.monotonic() >= self.t_end:          # нота кончилась
            self.index += 1
            if self.index >= len(self.melody):
                self.playing = False
                self.pwm.duty_cycle = 0
            else:
                self._begin_note()

song = [("C4", 1), ("E4", 1), ("G4", 1), ("C5", 2), ("P", 1), ("G4", 1), ("C5", 2)]
player = Player(board.GP13, song, bpm=140)
led = digitalio.DigitalInOut(board.LED)   # во время музыки мигаем светодиодом
led.direction = digitalio.Direction.OUTPUT
player.start()
t_led = time.monotonic()

```

```
while player.playing:
    player.update()
    if time.monotonic() - t_led >= 0.2:
        t_led = time.monotonic()
        led.value = not led.value
    # здесь могли бы опрашиваться кнопки, датчики, экран...

led.value = False
```

Теперь музыка играет «в фоне», а главный цикл свободен: в него легко добавить опрос кнопок или вывод на экран. Такой приём называют **конечным автоматом**: у объекта есть состояние, а метод по событиям (здесь — по времени) переключает его. Подобный подход мы развернём в уроке 13.

21.11 Усиление: динамик и транзистор

Пьезоизлучатель тихий и звучит «по-мультяшному». Для нормального звука нужен электродинамический динамик (громкоговоритель) сопротивлением $8\,\Omega$. Но вывод GPIO даёт единицы миллиампер, а динамику нужны десятки. Значит, нужен ключ — как для светодиодной ленты или мотора (урок 20). Схема простейшая: NPN-транзистор, резистор в базе и защитный диод параллельно катушке динамика.

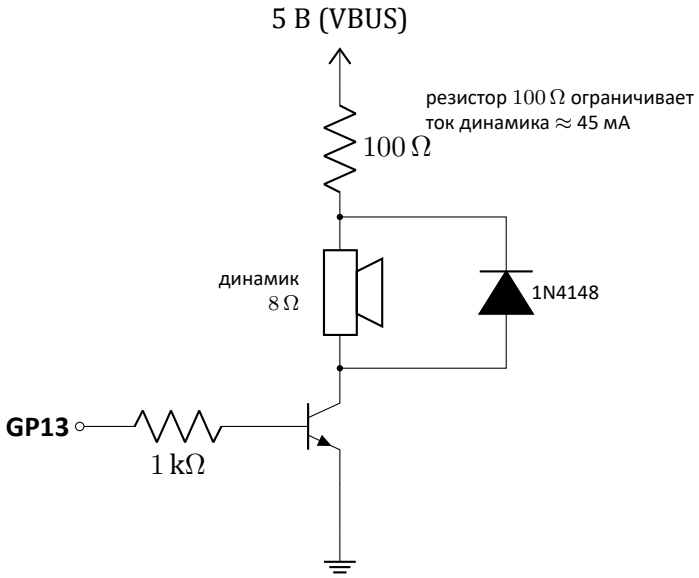


Рис. 21.10: Динамик через NPN-ключ: диод гасит выбросы напряжения катушки; земля Pico и 5 В — общая

Расчёт. Динамик $8\,\Omega$ с последовательным резистором $100\,\Omega$ от $5\,\text{V}$ пропускает $I_K \approx 5/108 \approx 46$ мА. Коэффициент усиления транзистора $\beta \geq 100$, значит, базе нужен ток $I_B \geq 0{,}46$ мА; для надёжного насыщения берут в 5 раз больше — около 2,3 мА. Напряжение на резисторе базы $3{,}3 - 0{,}7 = 2{,}6$ В, поэтому $R_B = 2{,}6/0{,}0023 \approx 1{,}1$ кОм, берём $1\,\text{k}\Omega$. Ток базы $2{,}6$ мА меньше предела вывода GPIO.

Осторожно

Динамик $8\,\Omega$ нельзя подключать к GPIO напрямую, а к $5\,\text{V}$ через транзистор — без токоограничивающего резистора: ток почти $0{,}6$ А перегрузит транзистор и порт USB. Даже с резистором через ключ получается тусклый звук с постоянной составляющей; для громкого и чистого звука применяют усилитель (следующие разделы). Диод обязателен: катушка динамика создаёт выбросы напряжения, которые могут убить транзистор.

21.12 ШИМ как цифро-аналоговый преобразователь

ШИМ умеет не только «пищать». Если повышать частоту ШИМ далеко за слышимый диапазон (десятки килогерц и выше) и менять скважность в такт звуку, а затем пропустить сигнал через **RC-фильтр** (интегрирующую цепочку), получится плавное напряжение, повторяющее форму звука. Так работает «цифро-аналоговый преобразователь на ШИМ» (ЦАП). Среднее напряжение на выходе фильтра пропорционально скважности:

$$U_{\text{ср}} = D \cdot 3{,}3\ \text{В}, \qquad f_{\text{ср}} = \frac{1}{2\pi RC}.$$

Для $R = 1\,\text{k}\Omega$ и $C = 10\,\text{nF}$ получаем $f_{\text{ср}} \approx 16$ кГц. Несущая частота ШИМ должна быть значительно выше (тогда фильтр её подавит), а слышимый звук — ниже. Конкретную частоту несущей выбирает библиотека **CircuitPython** (модуль `audiopwmio`); она зависит от версии, так что при плохом звуке экспериментируйте с ёмкостью конденсатора.

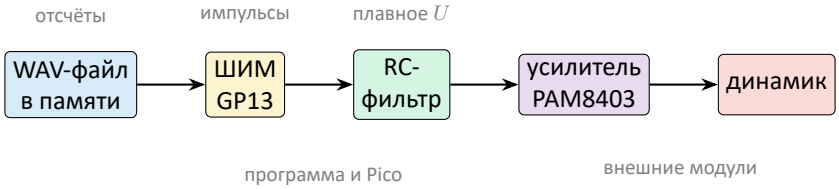


Рис. 21.11: Цепочка цифрового звука через ШИМ: данные → ШИМ → фильтр → усилитель → динамик

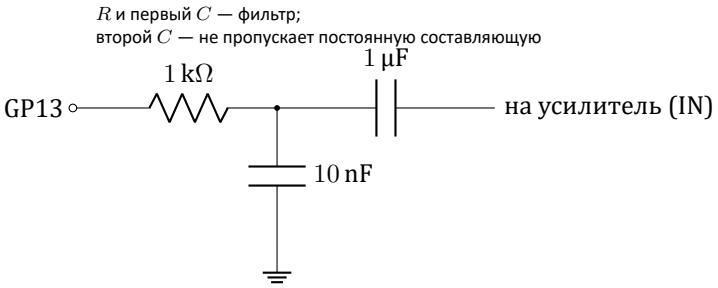


Рис. 21.12: RC-фильтр после ШИМ-звука; подбирайте C по слуху

21.13 Цифровой звук: отсчёты, WAV и I2S

Файл WAV хранит звук в виде **отсчётов**: много раз в секунду измеряется мгновенное значение волны, и числа записываются подряд. Сколько отсчётов в секунду — это **частота дискретизации** (CD — 44 100 Гц, телефон — 8 000 Гц), а число бит на отсчёт — **разрядность** (8 или 16). Чтобы воспроизвести, надо с той же скоростью отдавать отсчёты цифро-аналоговому преобразователю. Размер файла легко подсчитать: 16 бит, моно, 22 050 Hz дают $22\,050 \cdot 2 = 44\,100$ байт в секунду, то есть минута звука займёт около $2{,}6$ МВ. На флеш-памяти Pico (2 МБ у Pico и Pico W, 4 МБ у Pico 2 и Pico 2 W) помещаются лишь короткие эффекты.

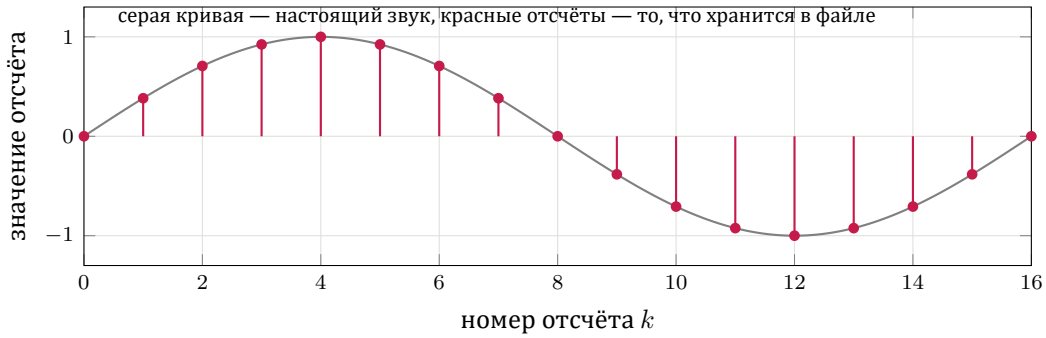


Рис. 21.13: Дискретизация: звук превращается в последовательность чисел

Ни у RP2040, ни у RP2350 нет встроенного ЦАП, поэтому есть два пути: ШИМ + фильтр (качество телефона) или цифровая шина **I2S**. В I2S отсчёты передаются по трём линиям: тактовая BCLK, выбор канала LRC (word select) и данные DIN. Преобразованием в звук занимается микросхема-усилитель (например, **MAX98357A**: усилитель до 3 Вт с входом I2S на плате). Для Pico I2S реализован на блоке PIO (урок 14), процессор почти не участвует в передаче.

Подключение	MicroPython	CircuitPython
ШИМ + RC	нет готовой библиотеки; только собственный код с `PWM/Timer` (нагрузка на CPU, качество среднее)	`audiopwmio.PWMAudioOut`: проигрывает WAV и `RawSample`
I2S	`machine.I2S` (порт rp2; конструктор `I2S(id, sck=, ws=, sd=, mode=, bits=, format=, rate=, ibuf=))`	`audiobusio.I2SOut(bit_clock, word_select, data)`
WAV и микширование	читаем файл и `write()` вручную, заголовок пропускаем сами	`audiocore.WaveFile`, `audiomixer.Mixer`

Таблица 21.3: Что где доступно. Поддержка может отличаться в зависимости от версии прошивки: сверяйтесь с документацией

Осторожно

Для обеих реализаций I2S на RP2 вывод word select (LRC) должен стоять *сразу за* выводом тактового сигнала (BCLK): BCLK = GP10, LRC = GP11. Это ограничение связано с тем, как устроена программа PIO. В **CircuitPython** убедитесь, что в вашей сборке есть модуль `audiobusio` — в некоторых урезанных сборках для малых плат его нет.

Синусоида через I2S

Листинг 21.13: **MicroPython**: чистый синус 440 Hz через I2S (MAX98357A)

```
from machine import Pin, I2S
import math
import struct

# Цифровой усилитель MAX98357A: BCLK -> GP10, LRC (word select) -> GP11, DIN -> GP9.
# На RP2 вывод ws обязан стоять сразу за sck (GP11 = GP10 + 1).
RATE = 22000                    # частота дискретизации, отсчётов в секунду
N = 50                          # отсчётов на один период синусоиды
FREQ = RATE // N                # 22000 / 50 = 440 Гц
AMPL = 6000                     # амплитуда из максимума 32767 (приглушённо!)

audio = I2S(0, sck=Pin(10), ws=Pin(11), sd=Pin(9),
```

```
            mode=I2S.TX, bits=16, format=I2S.MONO, rate=RATE, ibuf=4000)

# один период синусоиды в 16-битных знаковых отсчётах
buf = bytearray(N * 2)
for i in range(N):
    struct.pack_into("<h", buf, 2 * i, int(AMPL * math.sin(2 * math.pi * i / N)))

print("играем", FREQ, "Гц; Ctrl+C --- стоп")
try:
    while True:
        audio.write(buf)          # блокируется, пока в буфере нет места
finally:
    audio.deinit()
```

Листинг 21.14: **CircuitPython**: тот же синус через `audiobusio`

```
import array
import math
import time
import board
import audiobusio
import audiocore

# Цифровой усилитель MAX98357A: BCLK -> GP10, LRC (word select) -> GP11, DIN -> GP9.
# На RP2 word_select обязан стоять сразу за bit_clock (GP11 = GP10 + 1).
i2s = audiobusio.I2SOut(bit_clock=board.GP10, word_select=board.GP11,
                        data=board.GP9)

RATE = 22000
N = 50                                  # отсчётов на период: 22000 / 50 = 440 Гц
# тип "H" --- беззнаковые 16 бит, ноль звука лежит посередине (32768)
wave = array.array("H", [int(math.sin(2 * math.pi * i / N) * 6000 + 32768)
                         for i in range(N)])
sample = audiocore.RawSample(wave, sample_rate=RATE)

i2s.play(sample, loop=True)             # играет в фоне, без участия цикла
while True:
    time.sleep(1)                       # здесь могла бы быть другая работа
```

Заметьте разницу в подходах. В **MicroPython** нужно *самому* запихивать данные в буфер вызовом `write()` в цикле; пока вызов блокирует программу, больше ничего сделать нельзя (для фонового звука придётся использовать `asyncio` с неблокирующей записью, у `I2S` есть режим с обратным вызовом и режим `asyncio`). В **CircuitPython** вызов `play(..., loop=True)` запускает звук и немедленно возвращается: данные подаются через DMA в фоне.

Воспроизведение WAV-файла

Подготовьте файл в бесплатной программе Audacity: «Проект → Частота проекта 22050», дорожка моно, «Экспорт → WAV, 16-bit PCM». Скопируйте его на плату: `beep.wav` в **CircuitPython** (на диск `CIRCUITPY`) или в файловую систему Pico в **MicroPython** (через Thonny).

Листинг 21.15: MicroPython: проигрывание WAV через I2S

```
from machine import Pin, I2S

# Файл beep.wav на плате: PCM, 16 бит, моно, 22050 Гц (подготовьте в Audacity).
# Заголовок считаем стандартным --- ровно 44 байта; у файлов с доп. блоками это не так.
audio = I2S(0, sck=Pin(10), ws=Pin(11), sd=Pin(9),
            mode=I2S.TX, bits=16, format=I2S.MONO, rate=22050, ibuf=8000)

buf = bytearray(2048)
view = memoryview(buf)

try:
    with open("/beep.wav", "rb") as f:
        f.seek(44)                      # пропускаем заголовок
        while True:
            n = f.readinto(buf)         # читаем кусок данных
            if n == 0:
                break                   # файл закончился
            audio.write(view[:n])       # отдаём в I2S (блокирующая запись)
finally:
    audio.deinit()
```

Листинг 21.16: CircuitPython: проигрывание WAV через ШИМ (`audiopwmio`)

```
import board
import audiocore
import audiopwmio

# Аудио через ШИМ: на GP13 получается «цифровой» звук, который нужно
# пропустить через RC-фильтр и усилитель (см. урок). Для I2S замените
# на audiobusio.I2SOut(board.GP10, board.GP11, board.GP9).
audio = audiopwmio.PWMAudioOut(board.GP13)

with open("beep.wav", "rb") as f:       # WAV: PCM, 16 бит, моно, до 44100 Гц
    wav = audiocore.WaveFile(f)
    audio.play(wav)
    while audio.playing:                # звук идёт в фоне
        pass                            # здесь можно заниматься другими делами

audio.deinit()
```

Совет

Если звука нет, проверьте: файл в формате PCM (не MP3, не 24 бита), моно, частота не выше 44 100 Гц. Файл должен лежать там, где его ищет программа. В CircuitPython на малой памяти длинные WAV читаются кусками (буфер `bytearray(1024)`), в MicroPython — как в примере, `readinto()`.

Если нужно проигрывать музыку и накладывать на неё эффекты, в CircuitPython применяют **микшер**: он складывает несколько источников. У каждого голоса своя громкость (`level`), все файлы должны иметь параметры, указанные у микшера.

Листинг 21.17: CircuitPython: микшер — фоновая музыка и щелчки поверх

```
import time
import board
import audiocore
import audiomixer
import audiopwmio

audio = audiopwmio.PWMAudioOut(board.GP13)

# параметры микшера должны совпасть с параметрами всех WAV-файлов
mixer = audiomixer.Mixer(voice_count=2, sample_rate=22050, channel_count=1,
                         bits_per_sample=16, samples_signed=True)
audio.play(mixer)                          # микшер играет постоянно

music = audiocore.WaveFile(open("music.wav", "rb"), bytearray(1024))
click = audiocore.WaveFile(open("click.wav", "rb"), bytearray(1024))

mixer.voice[0].play(music, loop=True)      # голос 0: фоновая музыка по кругу
mixer.voice[0].level = 0.4                 # громкость 0.0 ... 1.0

while True:
    time.sleep(2)
    mixer.voice[1].play(click)             # голос 1: короткий звук поверх музыки
```

В MicroPython стандартного микшера нет: смешивание отсчётов (сумму с ограничением) пришлось бы писать самому.

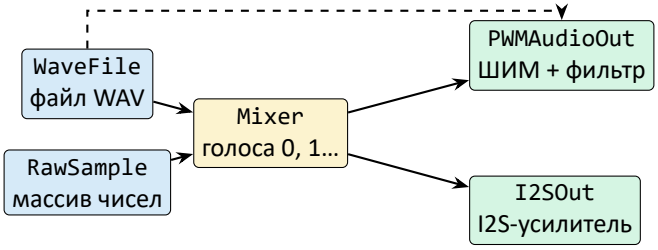


Рис. 21.14: Устройство звукового тракта в CircuitPython: источники → микшер → выход. Микшер необязателен

21.14 Усилитель PAM8403

Дешёвая плата с микросхемой PAM8403 — стереоусилитель класса D (до 3 Вт на канал при 5 В) с регулятором громкости. На вход подают звук (L/R и GND), питание 5 V (например, от VBUS), на выходе — динамики 4 Ω или 8 Ω. Не соединяйте выходные контакты динамика с землёй: у усилителя мостовой выход, и «минус» динамика не является землёй.

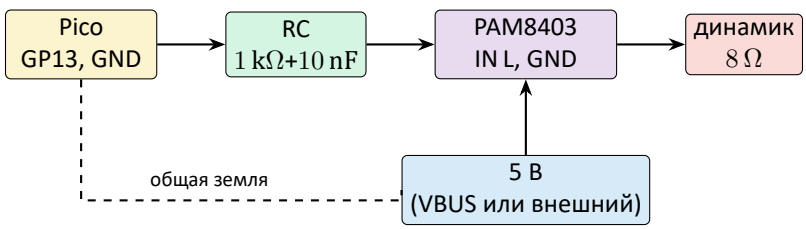


Рис. 21.15: Подключение PAM8403: земли Pico и усилителя обязательно общие

Осторожно

Динамик на ватт при полной громкости потребляет сильно больше, чем даёт USB-порт компьютера (500 mA): питайте усилитель от отдельного источника 5 В (но с общей землёй), а громкость не выкручивайте на максимум. Плохое питание проявляется как «жужжание» и перезагрузки Pico.

21.15 Практика: четыре шага к музыкальной шкатулке

1. Соберите монтажную схему рис. 21.7 и проверьте пьезоизлучатель программой `tone()`.
2. Запустите гамму и убедитесь, что ноты идут вверх. Если звучит «фальшиво», проверьте таблицу частот.
3. Запустите «Оду к радости». Измените `BPM`: слышно ли, что мелодия ускоряется, но ноты не меняются?
4. Запустите пианино и сыграйте одним пальцем «до — ми — соль — до».

Если не работает

- **Нет звука вообще** — перепутан пин, неподходящий излучатель (активный — нужна другая программа) или нулевая скважность: проверьте `duty_u16(32768)` / `duty_cycle = 32768`.
- **Тон «щёлкает» вместо ноты** — слишком низкая частота ШИМ: для частот ниже 20 Гц слышны отдельные щелчки.
- **В CircuitPython ошибка при смене частоты** — не указан `variable_frequency=True` при создании `PWMOut`.
- **«Пианино» играет само** — кнопка не подключена к GND, а подтяжка не включена: нужен `Pin.PULL_UP` / `Pull.UP`.
- **Звук тихий** — пассивный диск громче всего на резонансной частоте 2–4 кГц; для низких нот нужен резонатор (приклеить диск на коробочку) или динамик с усилителем.
- **Нет модуля `audiopwmio` / `audiobusio`** — в вашей сборке CircuitPython его нет; обновите прошивку. Список модулей: `help("modules")` в REPL.
- **WAV играет искажённо или быстро** — частота или разрядность файла не совпадают с настройкой (`rate`, микшер).
- **Гул и треск через усилитель** — общая земля, отдельное питание, провода короче, добавьте конденсатор 100 µF по питанию.

Итоги урока

- Высота тона — частота ШИМ, громкость — скважность (до 50 %); нота $f = 440 \cdot 2^{n/12}$.
- Мелодия — список пар «нота, длительность» и темп в BPM; между нотами нужны зазоры.
- Класс с методом `update()` позволяет играть музыку, не блокируя программу.
- Динамик подключают через транзистор с диодом или через усилитель, а не напрямую к GPIO.
- Цифровой звук — это отсчёты; на Pico его воспроизводят через ШИМ + RC или I2S (CircuitPython — `audiopwmio`, `audiobusio`, `audiomixer`; MicroPython — `machine.I2S`).

Проверь себя

1. Как связаны частота ШИМ, высота ноты и скважность с громкостью?
2. По какой формуле вычисляют частоту ноты на n полутонов выше «ля»? Чему равна частота «до» первой октавы?
3. Чем пассивный пьезоизлучатель отличается от активного и почему активный не сыграет мелодию?
4. Зачем между нотами нужен зазор, и как сделать паузу в мелодии?
5. Почему динамик не подключают к GPIO напрямую и какую роль в схеме с транзистором играют резистор базы и диод?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите схему с пьезоизлучателем и сыграйте гамму от «до» вверх и вниз.
2. ★ Измените темп «Оды к радости» вдвое (60 и 240 BPM) и подсчитайте длительность одной четверти.
3. ★★ Запишите другую мелодию (например, «В лесу родилась ёлочка») списком пар и сыграйте её.
4. ★★ Добавьте к пианино пятую кнопку и сделайте так, чтобы одна кнопка поднимала весь строй на октаву.
5. ★★★ Реализуйте «запись и воспроизведение»: нажатые ноты и их длительности запоминаются в список, а по длинному нажатию сохранённая мелодия проигрывается классом `Player`.
6. ★★★★ Соберите «музыкальный будильник»: класс `Player` играет мелодию с нарастающей громкостью, кнопка выключает её, а светодиод мигает в такт мелодии; всё это без `sleep()`.

Урок 22. USB: клавиатура, мышь, диск, консоль

Цель	понять, как устроено USB-устройство, какие «лица» может показывать компьютеру Pico (консоль, диск, клавиатура, мышь, MIDI), и собрать «макропад» из четырёх кнопок
Опирается на	урок 3: кнопки и дребезг; урок 2: GPIO; урок 1: REPL и загрузчик BOOTSEL
Результат	плата определяется компьютером как клавиатура и мышь, управляет громкостью и играет роль MIDI-пианино; вы умеете включать и отключать USB-устройства и диск в `boot.py`
Понадобится	плата Pico с USB-кабелем для данных, 4 кнопки, провода, макетная плата, компьютер с Windows, macOS или Linux

22.1 Идея урока коротко

Кабель USB, которым вы всё время прошиваете Pico, умеет гораздо больше, чем «передавать файлы». Компьютер спрашивает у платы: «Кто ты?», а плата *отвечает*: «Я последовательный порт», «Я флешка», «Я клавиатура» — или всё сразу. Меняя ответ, мы меняем поведение: Pico может нажимать клавиши, двигать курсор, регулировать громкость и играть ноты в музыкальной программе. В **CircuitPython** всё это есть «из коробки» и включается двумя строками. В **MicroPython** стандартная прошивка показывает компьютеру только последовательную консоль (и режим загрузчика), а клавиатуру придётся делать иначе — об этом честно поговорим ниже.

22.2 Как устроено USB

USB (Universal Serial Bus — «универсальная последовательная шина») построена по принципу «один главный и много подчинённых». **Хост** (компьютер) сам решает, когда и с кем говорить; **устройство** (наша плата) только отвечает. Устройство *никогда* не может заговорить первым — хост периодически спрашивает («опрашивает») его.

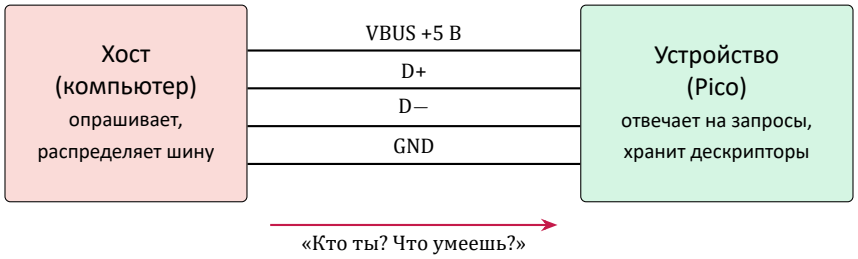


Рис. 22.1: Хост и устройство: четыре провода кабеля USB (питание, данные $D+$ и $D-$, земля)

Данные идут по двум проводам $D+$ и $D-$ **дифференциально**: когда на одном напряжение высокое, на другом низкое (рис. 22.2). Приёмник смотрит не на абсолютный уровень, а на *разность* сигналов, поэтому помехи, которые одинаково сидят на обоих проводах, сами сокращаются. В Pico USB работает в режиме Full-Speed: $12\,\text{Mbit/s}$.

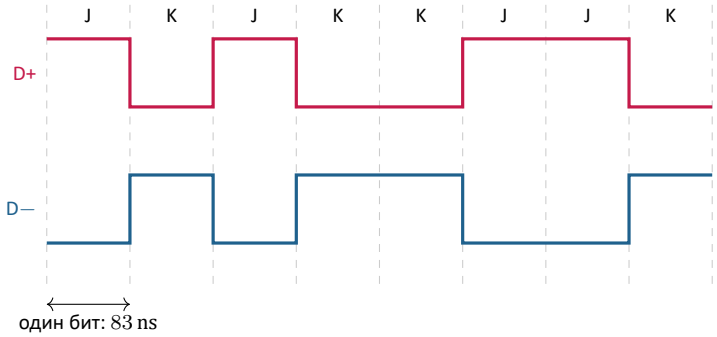


Рис. 22.2: Дифференциальная передача: линии $D+$ и $D-$ всегда в противофазе, состояния J и K несут биты

Дескрипторы: паспорт устройства

Сразу после подключения происходит **перечисление** (enumeration): хост присваивает устройству адрес и читает его **дескрипторы** — маленькие таблицы с описанием. Именно по ним операционная система выбирает драйвер. Дескрипторы образуют дерево (рис. 22.3).

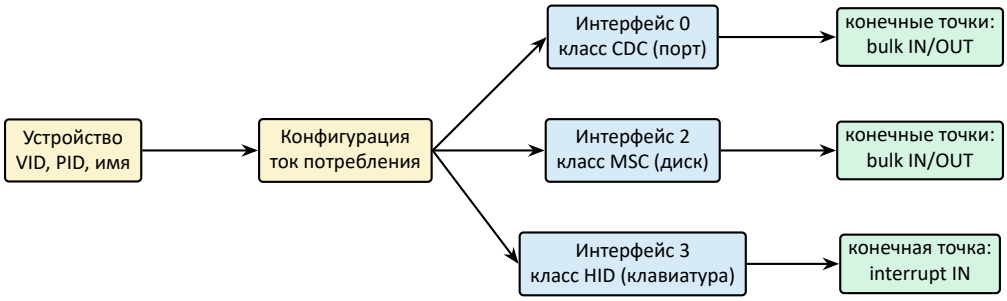


Рис. 22.3: Дерево дескрипторов составного (composite) устройства

- **VID и PID** — идентификатор производителя и изделия. У Raspberry Pi VID равен `0x2E8A`. Для своих устройств нельзя использовать чужие пары VID/PID в продаже, но для учёбы это допустимо.
- **Интерфейс** — одна «функция» устройства. Один кабель может нести несколько интерфейсов: это и есть *составное* устройство.
- **Конечная точка** (endpoint) — «почтовый ящик» на устройстве, куда хост кладёт данные или откуда забирает. У каждой точки свой номер и направление (IN — к хосту, OUT — от хоста). Число точек в чипе ограничено, поэтому нельзя включить *все* классы одновременно.

Тип передачи	Для чего	Гарантии	Пример
Control	настройка, дескрипторы	надёжно	перечисление
Interrupt	редкие короткие данные	опрос по расписанию, малая задержка	клавиатура, мышь
Bulk	большие блоки	надёжно, без гарантий по времени	флешка, консоль
Isochronous	потоки реального времени	по времени, без повторов	звук, камера

Таблица 22.1: Четыре типа передач USB

22.3 Классы: что умеет Pico

Класс устройства — договорённость о том, как с ним разговаривать, поэтому драйверы не нужно ставить вручную. Нам понадобятся три класса и один вспомогательный:

Класс	Что это	MicroPython	CircuitPython
CDC	последовательный порт	REPL и `print()`; есть всегда	консоль и дополнительный порт `usb_cdc.data`
MSC	флешка	только в BOOTSEL (диск RPI-RP2)	диск CIRCUITPY, можно скрыть
HID	клавиатура, мышь, кнопки громкости	нет в стандартной прошивке; сторонние библиотеки	`usb_hid` + `adafruit_hid`
MIDI	музыкальный контроллер	нет в стандартной прошивке	`usb_midi` + `adafruit_midi`

Таблица 22.2: Поддержка USB-классов на Pico (стандартные прошивки; версии могут меняться)

Это интересно

Загрузчик BOOTSEL зашит в ПЗУ чипа и не зависит от того, что за прошивка записана во flash. Он появляется как диск RPI-RP2 (у Pico 2 — RP2350) и принимает файлы `.uf2`. Поэтому «убить» плату программой практически невозможно: удерживая BOOTSEL при подключении, вы всегда получите диск.

22.4 USB-консоль (CDC)

Консоль, в которую выводит `print()`, — это последовательный порт CDC. На компьютере он виден как `COMx` (Windows), `/dev/ttyACM0` (Linux) или `/dev/cu.usbmodem...` (macOS). С его помощью плата может не только писать, но и *слушать* команды. Научим Pico включать светодиод по слову `on` и выключать по слову `off`.

Листинг 22.1: MicroPython: команды с компьютера через консоль

```
from machine import Pin
import sys
import select

led = Pin("LED", Pin.OUT)

# poll позволяет проверить, пришли ли байты по USB-консоли, не блокируя программу
poll = select.poll()
poll.register(sys.stdin, select.POLLIN)

print("Команды: on, off, status (по строке, Enter в конце)")

while True:
    if poll.poll(0):                        # 0 мс: не ждать, а только проверить
        cmd = sys.stdin.readline().strip()
        if cmd == "on":
            led.value(1)
            print("OK: светодиод включён")
        elif cmd == "off":
            led.value(0)
            print("OK: светодиод выключен")
        elif cmd == "status":
            print("LED =", led.value())
        elif cmd:
            print("Неизвестная команда:", cmd)
    # здесь может выполняться любая другая работа
```

В CircuitPython консоль занята REPL, поэтому для обмена данными заводят *второй* порт `usb_cdc.data`. Он выключен по умолчанию и включается в `boot.py` (подробнее — ниже).

Листинг 22.2: CircuitPython: обмен данными через usb_cdc.data

```
import board
import digitalio
import usb_cdc

# Второй последовательный порт (data) включается в boot.py: usb_cdc.enable(console=True, data=True)
serial = usb_cdc.data
serial.timeout = 0                          # чтение не блокирует программу

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

buf = b""
print("Порт data готов. Команды: on, off, status")

while True:
    if serial.in_waiting:
        buf += serial.read(serial.in_waiting)
        while b"\n" in buf:                 # собираем строку целиком
            line, buf = buf.split(b"\n", 1)
            cmd = line.strip().decode()
            if cmd == "on":
                led.value = True
                serial.write(b"OK: on\n")
            elif cmd == "off":
                led.value = False
                serial.write(b"OK: off\n")
            elif cmd == "status":
                serial.write(b"LED=%d\n" % led.value)
            elif cmd:
                serial.write(b"? " + cmd.encode() + b"\n")
```

Чтобы поговорить с платой с компьютера, подойдёт Thonny (для консоли MicroPython) или любая терминальная программа (PuTTY, `screen`, `pyserial`); для `usb_cdc.data` — именно терминальная программа или скрипт на `pyserial`, потому что Thonny подключается к консоли. Порты у CircuitPython получат два соседних номера: чаще всего первый — консоль, второй — data (сверьте в диспетчере устройств).

Осторожно

Если в MicroPython программа на плате читает `sys.stdin`, то Ctrl+C из Thonny остаётся служебной командой прерывания, а вот остальные символы попадут в вашу программу. Не удивляйтесь, если REPL «не отвечает», пока идёт чтение. Для надёжного обмена данными используют отдельные порты (в CircuitPython — `usb_cdc.data`).

Перезагрузка в BOOTSEL из программы

Чтобы не вынимать кабель и не держать кнопку, плату можно отправить в загрузчик командой из кода:

Листинг 22.3: MicroPython: перейти в режим загрузчика

```
import machine

# Перезагрузить плату в режим загрузчика: появится диск RPI-RP2 (RP2350 у Pico 2)
machine.bootloader()
```

Листинг 22.4: CircuitPython: перезагрузка в режим UF2

```
import microcontroller

# Следующая перезагрузка пойдёт в загрузчик UF2 (диск RPI-RP2)
microcontroller.on_next_reset(microcontroller.RunMode.UF2)
microcontroller.reset()
```

22.5 Режимы USB в CircuitPython: файл boot.py

У CircuitPython есть два файла запуска: `boot.py` выполняется **один раз при включении, до того как USB начнёт работать**, а `code.py` — потом, как обычно. Важно: именно в `boot.py` решается, какие устройства появятся на шине (рис. 22.4).

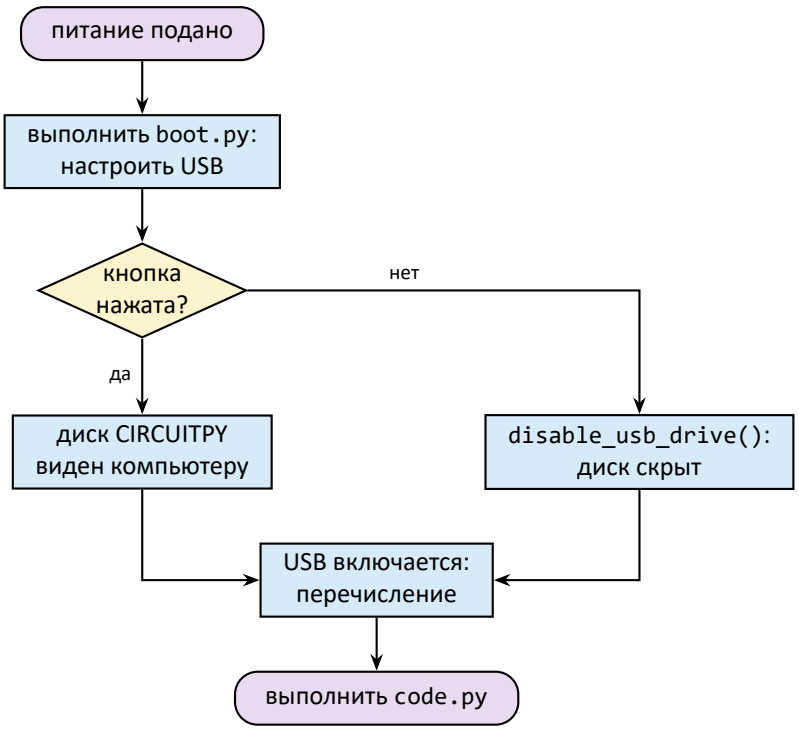


Рис. 22.4: Порядок запуска CircuitPython: `boot.py` решает судьбу USB-устройств и диска

Функции модулей настраивают устройства:

- `usb_hid.enable(devices)` — какие HID-устройства создать (клавиатура, мышь, `CONSUMER_CONTROL`); `usb_hid.disable()` — не создавать ни одного;
- `usb_cdc.enable(console=True, data=False)` — включить консоль и/или второй порт;
- `usb_midi.enable()` / `usb_midi.disable()` — MIDI-порты;
- `storage.disable_usb_drive()` — скрыть диск CIRCUITPY от компьютера;

- `storage.remount("/", readonly=False)` — разрешить программе писать на диск, когда компьютер его тоже видит (тогда компьютер будет видеть файлы «только для чтения»).

Листинг 22.5: CircuitPython: boot.py — набор устройств и скрытый диск

```
# boot.py: выполняется ОДИН раз при включении, ДО того как USB начнёт работать
import board
import digitalio
import storage
import usb_cdc
import usb_hid
import usb_midi

# Кнопка на GP10 между выводом и GND; нажата = False
button = digitalio.DigitalInOut(board.GP10)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP

# Какие устройства видит компьютер
usb_hid.enable((usb_hid.Device.KEYBOARD,
                usb_hid.Device.MOUSE,
                usb_hid.Device.CONSUMER_CONTROL))
usb_cdc.enable(console=True, data=True)    # REPL + второй порт для данных
usb_midi.enable()                          # MIDI (включён по умолчанию)

# Диск CIRCUITPY скрыт, если при включении кнопка НЕ нажата.
# Хотите изменить программу --- держите кнопку, пока втыкаете USB.
if button.value:
    storage.disable_usb_drive()

button.deinit()
```

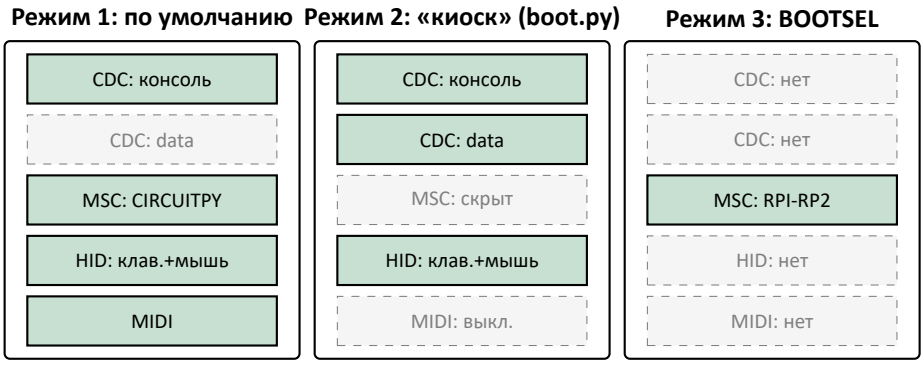


Рис. 22.5: Три «лица» платы: зелёным отмечено то, что видит компьютер (в режиме 1 data по умолчанию выключен)

Осторожно

- `boot.py` **не** перезапускается при сохранении файла и при мягкой перезагрузке (Ctrl+D). Чтобы он сработал, нужно *отключить и снова подключить* USB или нажать RUN.
- Если скрыть диск без «чёрного хода» (как кнопка в примере), вы потеряете удобный доступ к файлам. Выход всегда есть: консоль остаётся, а команда `storage.erase_filesystem()` в REPL стирает всё (подробнее о спасении платы — урок 23).
- Ошибки в `boot.py` вы не увидите на консоли: сообщения пишутся в файл `boot_out.txt` на

диске CIRCUITPY.

22.6 HID: клавиатура, мышь, кнопки громкости

HID (Human Interface Device) — «устройство взаимодействия с человеком». Все клавиатуры, мыши и геймпады в мире разговаривают с компьютером на этом языке. Устройство периодически (обычно раз в 1 – 10 ms) отдаёт хосту короткий **отчёт** (report) — набор байтов со снимком состояния.

Отчёт клавиатуры — 8 байт (рис. 22.6): первый — «флажки» модификаторов (Ctrl, Shift, Alt, Win), второй зарезервирован, остальные шесть — коды до шести одновременно нажатых клавиш. Код клавиши — не буква, а *положение* на клавиатуре: например, `0x04` — клавиша A (на русской раскладке получится «ф»).

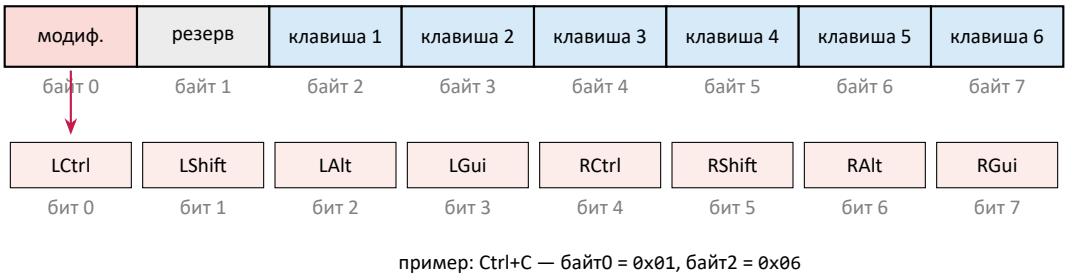


Рис. 22.6: Отчёт HID-клавиатуры: 8 байт и биты модификаторов

Клавиша	Код	Клавиша	Код	Примечание
A–Z	`0x04–0x1D`	Enter	`0x28`	в `adafruit_hid`
1–9, 0	`0x1E–0x27`	Esc	`0x29`	используются имена:
Пробел	`0x2C`	Tab	`0x2B`	`Keycode.ENTER`,
F1–F12	`0x3A–0x45`	Стрелка вправо	`0x4F`	`Keycode.F5`

Таблица 22.3: Часть кодов клавиш USB HID (Usage Table, раздел Keyboard)

Клавиатура в CircuitPython

Нужны две вещи: модули `usb_hid` (встроен) и библиотека `adafruit_hid` (положите папку `adafruit_hid` в каталог `lib` диска CIRCUITPY или установите командой `circup install adafruit_hid`). Класс `Keyboard` умеет `press()` — нажать и *держать*, `release_all()` — отпустить всё, `send()` — нажать и сразу отпустить. Для печати текста есть `KeyboardLayoutUS.write()`; он знает только раскладку США, то есть латиницу и знаки.

Листинг 22.6: CircuitPython: кнопка печатает текст и нажимает Ctrl+A

```
import time
import board
import digitalio
import usb_hid
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keyboard_layout_us import KeyboardLayoutUS
from adafruit_hid.keycode import Keycode

kbd = Keyboard(usb_hid.devices)
layout = KeyboardLayoutUS(kbd)              # раскладка США: печать текста

```

```
button = digitalio.DigitalInOut(board.GP10)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP

was_pressed = False
while True:
    pressed = not button.value              # нажата = 0
    if pressed and not was_pressed:         # только в момент нажатия
        layout.write("Hello, USB!")         # напечатать строку (латиница)
        kbd.press(Keycode.CONTROL, Keycode.A)   # Ctrl+A: выделить всё
        kbd.release_all()                   # ОБЯЗАТЕЛЬНО отпустить клавиши
    was_pressed = pressed
    time.sleep(0.02)
```

Осторожно

Если забыть `release_all()`, компьютер будет считать, что Ctrl «залип»: любые клавиши превратятся в сочетания. Отключите плату или нажмите физические клавиши Ctrl, чтобы распутать.

Совет

Начинайте с проверки в текстовом редакторе: откройте пустой документ и убедитесь, что Pico печатает именно туда. Не отлаживайте клавиатурные программы в окне, где символ может сделать что-то нужное (почта, браузер с формой оплаты, терминал).

Мышь

Мышь передаёт не координаты, а *смещения* за последний интервал (в условных «точках») и состояние кнопок. Поэтому `mouse.move(x=10, y=0)` сдвигает курсор вправо на 10 точек относительно *текущего* положения, откуда бы он ни стартовал. Планшет, который задаёт абсолютные координаты, — это уже другое устройство со своим дескриптором.

Листинг 22.7: CircuitPython: кнопка рисует квадрат курсором

```
import time
import board
import digitalio
import usb_hid
from adafruit_hid.mouse import Mouse

mouse = Mouse(usb_hid.devices)

button = digitalio.DigitalInOut(board.GP10)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP

while True:
    if not button.value:                    # пока кнопка нажата --- рисуем квадрат
        for dx, dy in ((10, 0), (0, 10), (-10, 0), (0, -10)):
            for _ in range(5):
                mouse.move(x=dx, y=dy)      # смещение относительно текущей позиции
                time.sleep(0.02)
        mouse.click(Mouse.LEFT_BUTTON)      # щелчок левой кнопкой
        mouse.move(wheel=-1)                # прокрутка колеса на один шаг
        time.sleep(0.3)
    time.sleep(0.02)
```

Кнопки громкости и мультимедиа

Клавиши громкости, паузы и «следующий трек» не входят в обычный отчёт клавиатуры: у них отдельное HID-устройство Consumer Control. В `adafruit_hid` это класс `ConsumerControl` и коды из `ConsumerControlCode`: `VOLUME_INCREMENT`, `VOLUME_DECREMENT`, `MUTE`, `PLAY_PAUSE`, `SCAN_NEXT_TRACK`. Пример встроен в макропад ниже.

22.7 Учимся программировать: таблица «кнопка → действие»

Макропад мог бы состоять из цепочки `if button1: ... elif button2: ...`. Но чем больше кнопок, тем длиннее цепочка, и добавить кнопку трудно. Программисты выносят различия в **данные**: заводят таблицу, где под номером кнопки лежит описание действия, а программа одна и та же для всех. Такую таблицу в Python удобно хранить в кортеже или словаре (`dict`).

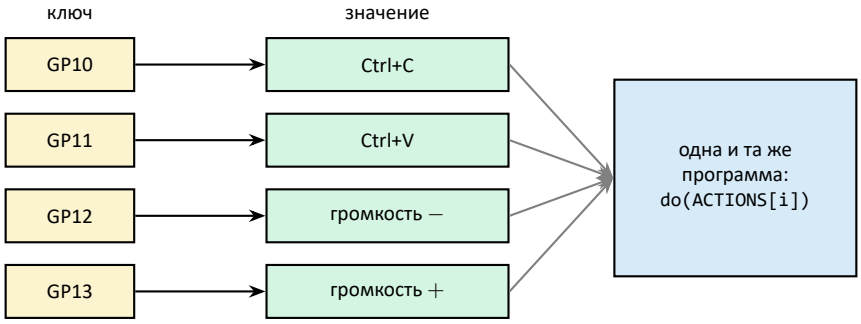
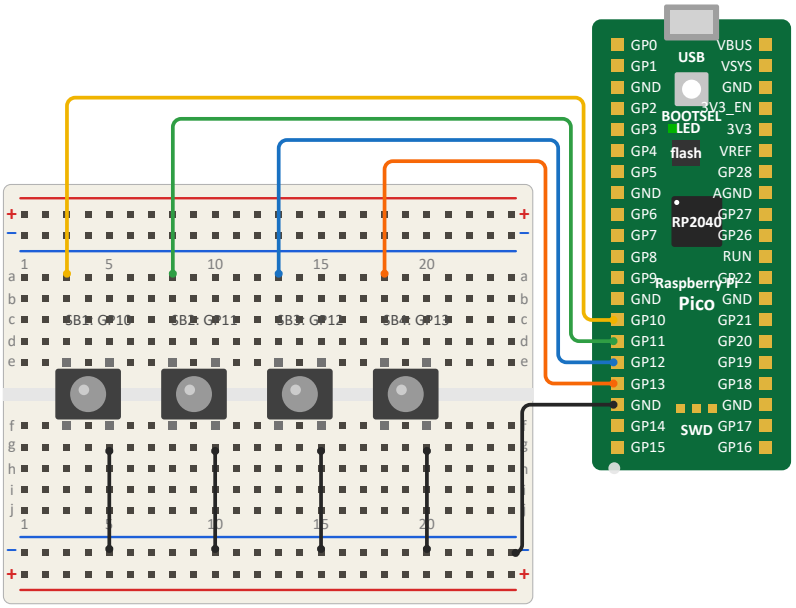


Рис. 22.7: Данные в таблице, поведение — в одной общей функции

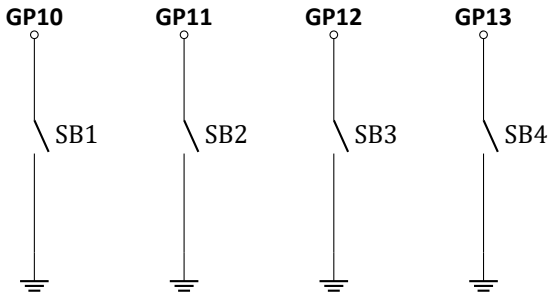
Это называется *управление данными* (table-driven design). Новую кнопку добавляют одной строкой в таблице. Заметьте и другой приём: нас интересует не то, *нажата* ли кнопка, а момент *нажатия* — переход из состояния «отпущена» в состояние «нажата». Такой переход называют **событием**; программа запоминает прошлое состояние и сравнивает с текущим.

22.8 Практика: макропад из четырёх кнопок

Макропад — мини-клавиатура с несколькими программируемыми кнопками. Подключим четыре кнопки к **GP10–GP13**; вторым выводом каждая кнопка идёт на землю. Подтягивающие резисторы включаются в программе (урок 3), внешние не нужны.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 22.8: Макропад: четыре кнопки на GP10–GP13

Кнопка, которую мы ставим поперёк канавки платы, замыкает пару столбцов x и $x+2$. Сигнальный провод подключён к верхней половине (столбец x), а земля — к нижней (столбец $x+2$). Пока кнопка не нажата, цепь разомкнута, внутренний резистор подтягивает вход к 3,3 В, и программа читает 1; при нажатии вход соединяется с землёй, и значение становится 0.

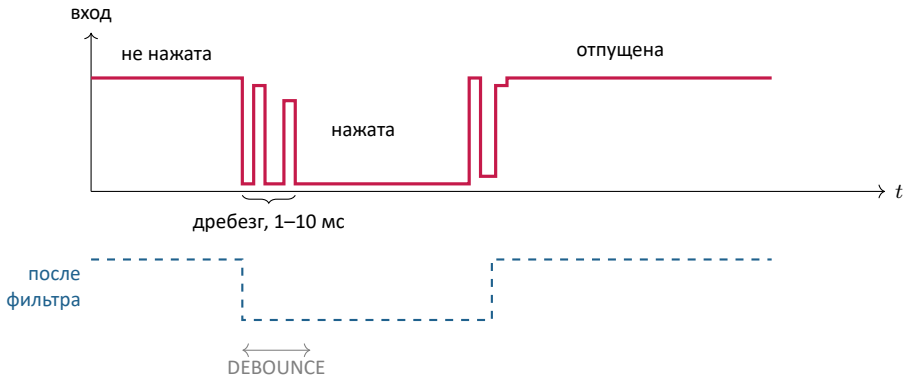


Рис. 22.9: Дребезг контактов и программный фильтр: решение принимается, когда сигнал устоялся

Механические контакты при замыкании «звенят» — несколько раз подряд размыкаются и замыкаются за миллисекунды. Для компьютера, который читает отчёты каждую миллисекунду, это выглядит как десяток нажатий. Поэтому в обеих программах используется **антидребезг**:

изменение признаётся настоящим, только если сигнал не меняется заданное время (30 мс).

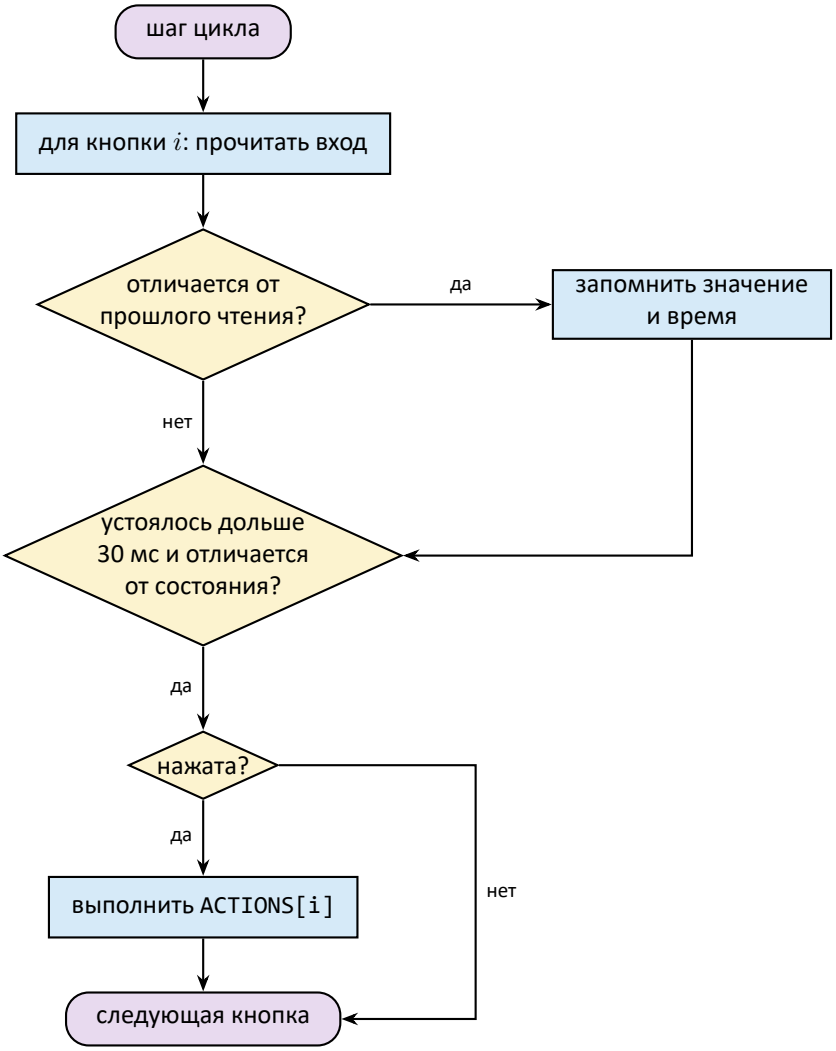


Рис. 22.10: Блок-схема опроса кнопок с антидребезгом

Листинг 22.8: CircuitPython: макропад с клавиатурой и кнопками громкости

```
import time
import board
import digitalio
import usb_hid
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keycode import Keycode
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.consumer_control_code import ConsumerControlCode

kbd = Keyboard(usb_hid.devices)
cc = ConsumerControl(usb_hid.devices)

# Таблица «кнопка -> действие»: (вид, что отправить)
PINS = (board.GP10, board.GP11, board.GP12, board.GP13)
ACTIONS = (
    ("key", (Keycode.CONTROL, Keycode.C)),            # копировать
    ("key", (Keycode.CONTROL, Keycode.V)),            # вставить
    ("cc", ConsumerControlCode.VOLUME_DECREMENT),     # тише
    ("cc", ConsumerControlCode.VOLUME_INCREMENT),     # громче
```

```
)

buttons = []
for pin in PINS:
    b = digitalio.DigitalInOut(pin)
    b.direction = digitalio.Direction.INPUT
    b.pull = digitalio.Pull.UP
    buttons.append(b)

DEBOUNCE = 0.03                                  # 30 мс
state = [False] * 4                              # подтверждённое состояние
changed = [0.0] * 4                              # когда пин последний раз менялся
raw_prev = [False] * 4


def do(action):
    kind, data = action
    if kind == "key":
        kbd.press(*data)
        kbd.release_all()
    else:
        cc.send(data)


while True:
    now = time.monotonic()
    for i, b in enumerate(buttons):
        raw = not b.value                        # нажата = True
        if raw != raw_prev[i]:                   # пин изменился --- ждём
            raw_prev[i] = raw
            changed[i] = now
        elif raw != state[i] and now - changed[i] > DEBOUNCE:
            state[i] = raw                       # устоялось
            if raw:                              # событие «нажали»
                do(ACTIONS[i])
    time.sleep(0.002)
```

Макропад в MicroPython: честно о возможностях

В стандартной прошивке MicroPython для Pico **нет готового HID**: она показывается компьютером как один последовательный порт. Есть три пути:

1. **Ваш скрипт на компьютере.** Плата печатает в консоль строки `KEY 0 DOWN`, а программа на компьютере (на Python с `pyserial`) читает их и нажимает клавиши, запускает программы. Плюс: работает на любой прошивке, видно всё происходящее. Минус: нужен запущенный скрипт.

2. **Сторонние библиотеки `usb-device-*` из репозитория micropython-lib.** В новых версиях MicroPython (с 1.23) в порте RP2 есть класс `machine.USBDevice`, на базе которого написаны пакеты для клавиатуры, мыши и MIDI (`usb-device-keyboard`, `usb-device-mouse`, `usb-device-midi`). Их ставят командой `mpremote mip install usb-device-keyboard`. Состав пакетов и интерфейс меняются от версии к версии, поэтому сверяйтесь с документацией вашей версии.

3. **Другая прошивка:** сборки MicroPython с TinyUSB-HID от сообщества или CircuitPython, в которой HID уже есть.

Листинг 22.9: MicroPython: вариант 1 — плата отправляет события по консоли

```
from machine import Pin
import time

# В стандартной прошивке MicroPython нет HID: плата не умеет «нажимать клавиши».
# Зато она может послать компьютеру строки "KEY 0".."KEY 3" по USB-консоли;
# на компьютере их читает скрипт (pyserial) и выполняет действия.
PINS = (10, 11, 12, 13)
buttons = [Pin(n, Pin.IN, Pin.PULL_UP) for n in PINS]

DEBOUNCE_MS = 30
state = [False] * 4
raw_prev = [False] * 4
changed = [0] * 4

while True:
    now = time.ticks_ms()
    for i, b in enumerate(buttons):
        raw = not b.value()                          # нажата = True
        if raw != raw_prev[i]:
            raw_prev[i] = raw
            changed[i] = now
        elif raw != state[i] and time.ticks_diff(now, changed[i]) > DEBOUNCE_MS:
            state[i] = raw
            print("KEY", i, "DOWN" if raw else "UP")  # событие уходит в консоль
    time.sleep_ms(2)
```

Листинг 22.10: MicroPython: вариант 2 — клавиатура через usb-device-keyboard (по версии библиотеки)

```
# Только для MicroPython >= 1.23 и библиотеки из micropython-lib:
#   mpremote mip install usb-device-keyboard
# API сторонней библиотеки менялся; сверяйтесь с README вашей версии.
import time
import usb.device
from usb.device.keyboard import KeyboardInterface, KeyCode
from machine import Pin

button = Pin(10, Pin.IN, Pin.PULL_UP)

kbd = KeyboardInterface()
# builtin_driver=True: оставляем встроенную консоль (CDC), иначе REPL по USB пропадёт
usb.device.get().init(kbd, builtin_driver=True)

while not kbd.is_open():                    # ждём, пока компьютер настроит устройство
    time.sleep_ms(100)

was = False
while True:
    now = not button.value()
    if now and not was:
        kbd.send_keys([KeyCode.H])          # нажать H
        time.sleep_ms(30)
        kbd.send_keys([])                   # отпустить все клавиши
    was = now
    time.sleep_ms(20)
```

Осторожно

Пример варианта 2 нужно воспринимать как схему: названия классов и методов в `usb-device-keyboard` зависят от версии, и на всех выпусках мы его не проверяли. Если импорт не удался, откройте каталог `usb-device-keyboard` в micropython-lib и посмотрите примеры рядом с исходным кодом пакета.

22.9 Этика и безопасность: BadUSB

Устройство, которое само себя выдаёт за клавиатуру и печатает быстрее человека, — мощный инструмент. Именно так устроены «BadUSB»-атаки: воткнули флешку — и «клавиатура» за секунды открыла командную строку и скачала вирус. Операционная система доверяет клавиатуре безусловно, потому что клавиатуры не атакуют.

Осторожно

- Эксперименты проводите только на **своём** компьютере и в текстовом редакторе, а не в чужих системах. Вмешательство в чужой компьютер без разрешения — нарушение закона.
- Не оставляйте Pico с «автопечатью» без присмотра: добавьте в программу «аварийный выключатель» (например, не стартовать, если зажата кнопка) и пауза-задержку перед печатью.
- Не публикуйте и не отдавайте другим платы с программами, рассчитанными на скрытое выполнение команд.
- Если компьютер внезапно «сам печатает», отключите USB-кабель.

22.10 MIDI-контроллер

MIDI (Musical Instrument Digital Interface) — стандарт общения музыкальных инструментов, придуманный в 1983 году. Это не звук, а *команды*: «нажата нота», «отпущена нота». Звучит синтезатор на компьютере. Основное сообщение состоит из трёх байтов (рис. 22.11).

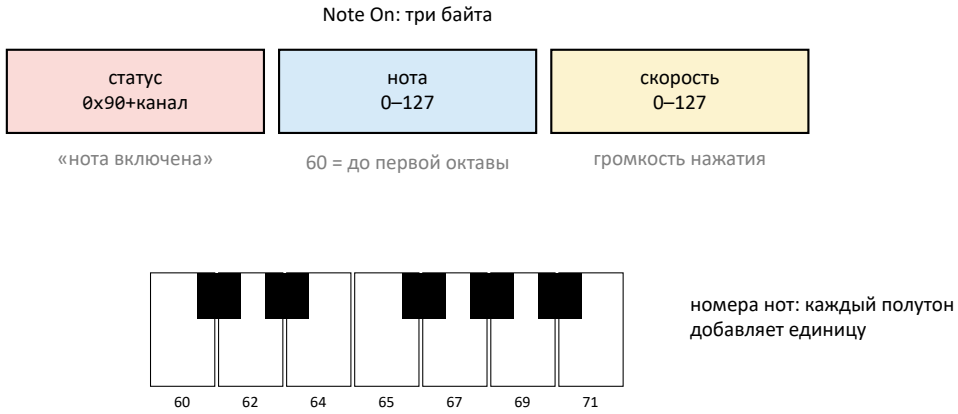


Рис. 22.11: Сообщение MIDI Note On и номера нот на клавишах

Сообщение	Статус-байт	Данные
Note Off	`0x80`+канал	нота, скорость
Note On	`0x90`+канал	нота, скорость (0 = выключить)
Control Change	`0xB0`+канал	номер регулятора, значение 0–127
Program Change	`0xC0`+канал	номер тембра

Таблица 22.4: Основные сообщения MIDI (канал 0–15)

Листинг 22.11: CircuitPython: четыре кнопки играют ноты по USB-MIDI

```
import time
import board
import digitalio
import usb_midi
import adafruit_midi
from adafruit_midi.note_on import NoteOn
from adafruit_midi.note_off import NoteOff

# usb_midi.ports[0] --- вход, ports[1] --- выход (с точки зрения платы)
midi = adafruit_midi.MIDI(midi_out=usb_midi.ports[1], out_channel=0)

NOTES = (60, 62, 64, 67)                    # до, ре, ми, соль (60 = до первой октавы)
PINS = (board.GP10, board.GP11, board.GP12, board.GP13)

buttons = []
for pin in PINS:
    b = digitalio.DigitalInOut(pin)
    b.direction = digitalio.Direction.INPUT
    b.pull = digitalio.Pull.UP
    buttons.append(b)

was = [False] * 4
while True:
    for i, b in enumerate(buttons):
        now = not b.value
        if now and not was[i]:
            midi.send(NoteOn(NOTES[i], 110))   # нота, громкость 0--127
        elif was[i] and not now:
            midi.send(NoteOff(NOTES[i], 0))
        was[i] = now
    time.sleep(0.01)
```

Чтобы услышать ноты, нужна программа-синтезатор: на компьютере подойдут бесплатные приложения вроде VMPK, любой DAW или онлайн-синтезаторы с поддержкой WebMIDI. Плата в списке MIDI-устройств появится под именем прошивки. Библиотеку `adafruit_midi` поставьте так же, как `adafruit_hid`.

В стандартном **MicroPython** USB-MIDI нет. Зато классический MIDI с разъёмом DIN-5 — это обычный UART на скорости 31250 бод, и с ним справится любая прошивка. В схеме на 3,3 В токовую петлю ограничивают резисторы по $33\,\Omega$ (рис. 22.12).

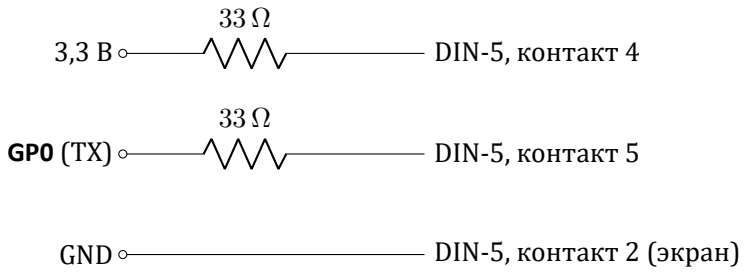


Рис. 22.12: Выход классического MIDI для логики 3,3 В (упрощённо, для самостоятельной сверки со спецификацией MIDI)

Листинг 22.12: MicroPython: MIDI по UART (DIN-5), 31250 бод

```
from machine import Pin, UART
import time

# Классический MIDI через разъём DIN-5: UART 31250 бод, 8N1.
# Вывод GP0 (TX) -> резистор 33 Ом -> контакт 5 разъёма MIDI OUT,
# +3,3 В -> резистор 33 Ом -> контакт 4 (токовая петля 5 мА по спецификации MIDI
# для логики 3,3 В); контакт 2 --- на землю. Сверьте со схемой в уроке.
# USB-MIDI в стандартной прошивке MicroPython нет.
uart = UART(0, baudrate=31250, tx=Pin(0), rx=Pin(1))

NOTES = (60, 62, 64, 67)
buttons = [Pin(n, Pin.IN, Pin.PULL_UP) for n in (10, 11, 12, 13)]
was = [False] * 4

def note_on(note, vel=110, ch=0):
    uart.write(bytes((0x90 | ch, note, vel)))   # статус, нота, скорость

def note_off(note, ch=0):
    uart.write(bytes((0x80 | ch, note, 0)))

while True:
    for i, b in enumerate(buttons):
        now = not b.value()
        if now and not was[i]:
            note_on(NOTES[i])
        elif was[i] and not now:
            note_off(NOTES[i])
        was[i] = now
    time.sleep_ms(10)
```

Совет

Подключайте гнездо DIN только к выходу Pico (MIDI OUT). Вход MIDI IN требует оптрона для гальванической развязки; сигнал напрямую на вывод Pico подавать не стоит.

22.11 Питание и безопасность USB

Контакт VBUS выдаёт 5 В от компьютера (до 500 mA по умолчанию для USB 2.0), и Pico получает отсюда питание. Помните три правила: **(а)** не соединяйте VBUS с выводами GPIO — там допустимо только 3,3 В; **(б)** кабель должен быть именно «для данных», а не «только зарядный»; **(в)** не пытайтесь питать от платы мощную нагрузку (моторы, длинные светодиодные ленты) — ток ограничен портом.

Если не работает

- **Компьютер не видит плату** — кабель «только для зарядки» или не тот порт; попробуйте другой кабель.
- **После `boot.py` пропал диск CIRCUITPY** — вы скрыли его; зайдите в REPL и выполните `import storage; storage.erase_filesystem()` или подключите плату с зажатой кнопкой из `boot.py`.
- **`ImportError: no module named adafruit_hid`** — нет папки `adafruit_hid` в `lib`.
- **Клавиши «залипли»** — забыт `release_all()`.
- **Плата печатает, а на русской раскладке получается «абракадабра»** — коды клавиш привязаны к положению, а не к букве; переключите раскладку на английскую.
- **Изменения в `boot.py` не действуют** — нужен полный сброс питания, а не Ctrl+D.
- **Порт `usb_cdc.data` не появился** — включите его в `boot.py` и переподключите USB.
- **MIDI-ноты не слышно** — компьютер сам звук не воспроизводит; запустите программный синтезатор и выберите плату как вход.

Итоги урока

- USB-устройство отвечает на запросы хоста; по дескрипторам компьютер определяет классы CDC, MSC, HID, MIDI.
- MicroPython показывает консоль CDC; CircuitPython — консоль, диск CIRCUITPY, HID и MIDI, а `boot.py` решает, что включать.
- HID-отчёт — несколько байт состояния; клавиатура передаёт код *положения* клавиши, мышь — смещение.
- Антидребезг и таблица «кнопка → действие» делают макропад надёжным и расширяемым.
- Устройства, «притворяющиеся клавиатурой», опасны: используйте их только на своём компьютере.

Проверь себя

1. Чем хост отличается от устройства в USB и почему устройство не может заговорить первым?
2. Что такое дескриптор и какую роль играет класс устройства?
3. Почему `boot.py` не срабатывает при сохранении файла и когда он выполняется?
4. Почему в программе нужен `release_all()` и что будет, если его забыть?
5. Какими способами можно сделать «клавиатуру» на MicroPython и в чём их ограничения?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите макропад и убедитесь в текстовом редакторе, что кнопки Ctrl+C и Ctrl+V работают.

2. ★ Замените действия кнопок: Esc, Enter, Win+D (`Keycode.GUI`), `PLAY_PAUSE`.
3. ★★ Измените время антидребезга с 30 мс на 5 и на 100 мс. Что изменилось при быстрых нажатиях?
4. ★★ Включите в `boot.py` режим «диск скрыт, пока кнопка не зажата» и проверьте, что вы умеете им пользоваться и восстанавливать доступ.
5. ★★★ Сделайте «слой»: при зажатой пятой кнопке-модификаторе остальные четыре кнопки выполняют другие действия (таблица из двух списков).
6. ★★★★ Реализуйте обмен через `usb_cdc.data` (или консоль MicroPython): компьютер присылает название слоя, плата отвечает числом нажатий кнопок, а скрипт на `pyserial` рисует график.

Урок 23. Отладка и надёжность

Цель	научиться находить и исправлять ошибки системно: читать сообщения об ошибках, вести журнал, следить за памятью, измерять сигналы приборами и делать программу, которая сама восстанавливается после сбоя
Опирается на	уроки 2–12 (ввод-вывод, шины, файлы) и урок 13 (несколько дел одновременно)
Результат	вы читаете traceback без паники, пишете журнал с уровнями и ротацией, ловите исключения, замеряете память, подключаете логический анализатор и Debug Probe, ставите сторожевой таймер
Понадобится	мультиметр; по желанию логический анализатор на 8 каналов (около 10 – 15 USD) и Raspberry Pi Debug Probe

23.1 Идея урока коротко

Ошибки бывают у всех, у профессионалов тоже. Разница в том, что профессионал не гадает, а **ищет**: сначала собирает факты, потом строит гипотезу, потом проверяет её одним маленьким экспериментом. В этом уроке мы соберём инструментарий отладчика: от строки `print()` до осциллограммы шины I^2C. Вторая половина урока о надёжности: как сделать устройство, которое не просто работает на столе, а переживает сбои питания, нехватку памяти и зависания, и в самом худшем случае само перезагружается.

23.2 Методика: ошибка — это загадка, а не наказание

Новички делают одну и ту же ошибку: видят, что программа не работает, и начинают менять всё подряд. Через полчаса код уже другой, схема другая, а что было сломано — неизвестно. Научный подход иной:

1. **Воспроизведи.** Найди действия, при которых ошибка возникает *всегда*. Ошибку, которая случается раз в сутки, ловят журналом (см. ниже).
2. **Сузь.** Дели проблему пополам, как при поиске слова в словаре: отключи половину программы или половину схемы — ошибка осталась или пропала?
3. **Выдвини гипотезу** и придумай эксперимент, который её *опровергнет*. «Датчик не отвечает» — значит, `i2c.scan()` не увидит адрес. Увидит — гипотеза неверна.

4. **Меняй одно за раз.** Одно изменение — один запуск. Иначе вы не узнаете, что именно помогло.
5. **Запиши результат.** Через неделю вы не вспомните, что уже проверяли.

Прежде всего нужно понять, *к какому классу* относится неисправность: от этого зависит, чем её искать.

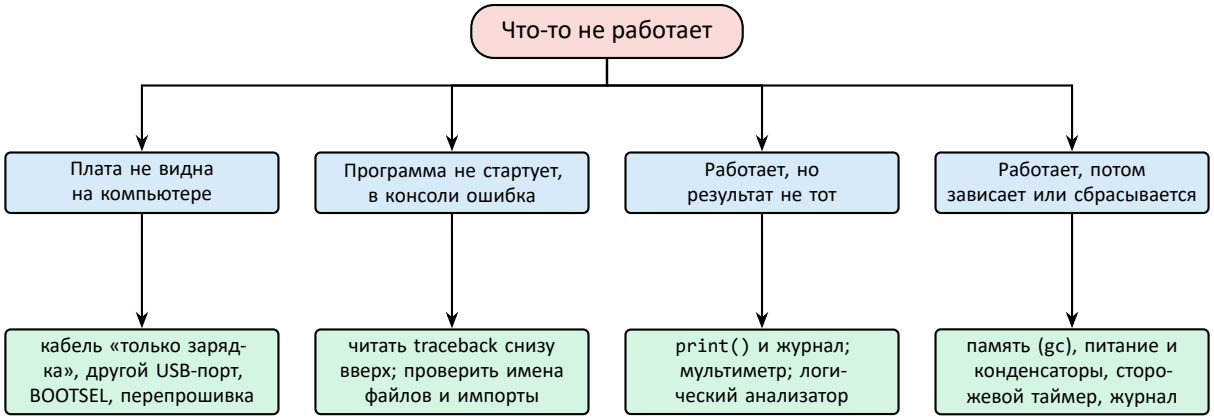


Рис. 23.1: Дерево диагностики: сначала определяем класс неисправности, потом выбираем инструмент

Совет

Метод «резиновой утки»: объясните проблему вслух любому неодушевлённому предмету, шаг за шагом. Очень часто ответ находится на середине объяснения. Это не шутка, так делают инженеры.

23.3 Читаем traceback

Когда программа падает, интерпретатор печатает **traceback** (обратную трассировку) — «цепочку вызовов» от верхнего уровня программы до места, где произошла ошибка. В нём всё, что нужно для поиска: файл, номер строки и тип ошибки.

```
Traceback (most recent call last):
  File "main.py", line 14, in <module>
  File "main.py", line 9, in read_temp
ZeroDivisionError: divide by zero
```

1. заголовок
2. откуда начали
3. где упало
4. причина

Читать **снизу вверх**: сначала тип и сообщение (строка 4), затем строка, где упало (3), затем кто её вызвал (2).

Рис. 23.2: Анатомия traceback: четыре строки, четыре ответа

Привычка номер один: **читайте последнюю строку первой**. В ней тип исключения (что случилось) и сообщение (подробности). Затем открывайте файл на строке, которую назвал traceback. В 90 % случаев этого достаточно.

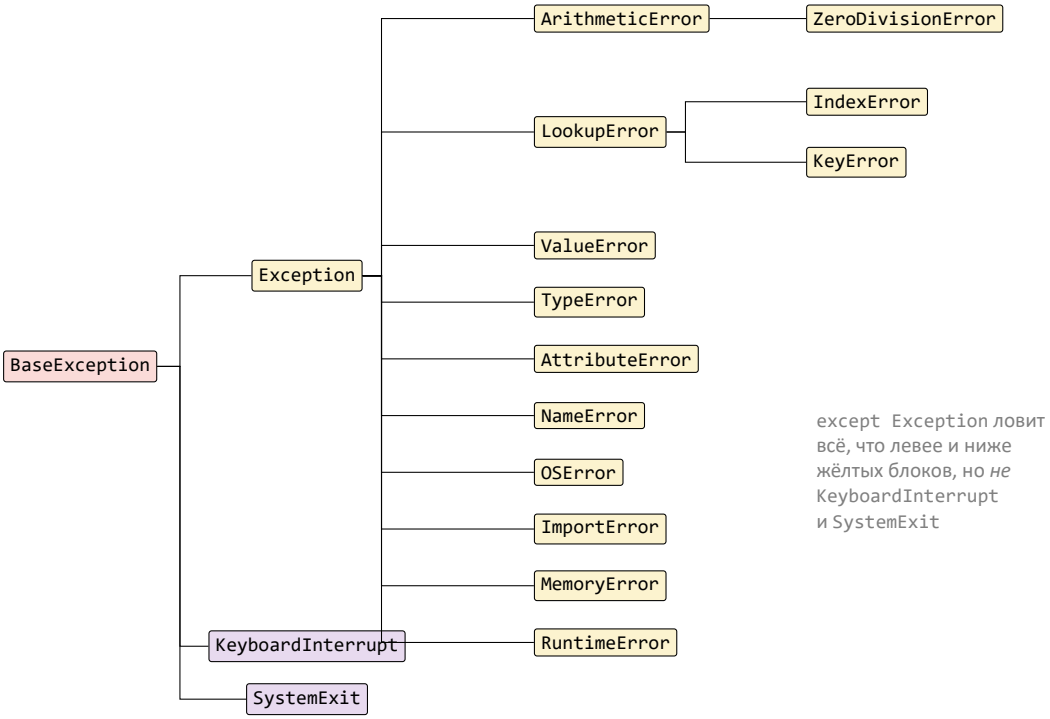


Рис. 23.3: Часть иерархии исключений Python: ловя «родителя», вы ловите всех «детей»

Исключение	Что обычно случилось	Что проверить
`SyntaxError`	нарушена грамматика: пропущено двоеточие, скобка, неверный отступ	строка выше указанной; отступы (пробелы, не табуляция)
`NameError`	имя не определено: опечатка, переменная создана позже	написание; порядок строк; `import`
`TypeError`	операция с несовместимыми типами: `"1"+1`, не тот аргумент	типы значений: `type(x)`
`ValueError`	тип верный, а значение нет: `int("abc")`; в **MicroPython** — номер вывода `Pin(99)`	сами значения: напечатать их
`AttributeError`	у объекта нет такого поля: опечатка; в **CircuitPython** — `board.GP99`	`dir(obj)`, версия библиотеки
`IndexError`, `KeyError`	выход за границы списка; нет такого ключа в словаре	длина списка, ключи словаря
`OSError`	проблема «снаружи»: нет файла (`errno 2`), шина не отвечает, диск только для чтения	провода, адрес, питание, права записи
`ImportError`	модуль или библиотека не найдены на плате	папка `lib`, имя файла, версия библиотеки
`MemoryError`	не хватило ОЗУ (264 КБ у RP2040, 520 КБ у RP2350)	`gc.mem_free()`, размеры буферов
`KeyboardInterrupt`	вы нажали Ctrl+C: это не ошибка, а просьба остановиться	ничего; ловить его не нужно

Таблица 23.1: Типичные исключения и первые подозреваемые

Программа ниже намеренно вызывает все эти ошибки и ловит их. Запустите её и сравните вывод с таблицей. Обратите внимание на два различия двух языков: неверный номер вывода в **MicroPython** даёт `ValueError`, а в **CircuitPython** — `AttributeError` (у модуля `board` просто нет такого имени); а для печати traceback пойманного исключения в **MicroPython** служит `sys.print_exception()`, в **CircuitPython** — `traceback.print_exception()`.

Листинг 23.1: MicroPython: вызываем и ловим типичные исключения

```
import sys
from machine import Pin


def trial(name, func):
    """Выполняет func и печатает, какое исключение получилось."""
    try:
        func()
        print(name, "-> ошибки нет")
    except Exception as e:                    # ловим любое «обычное» исключение
        print(name, "->", type(e).__name__, ":", e)


def bad_import():
    import no_such_module                     # ImportError


def no_file():
    open("нет_такого_файла.txt")              # OSError, errno 2 (ENOENT)


def out_of_memory():
    bytearray(10_000_000)                     # на 264 КБ ОЗУ это MemoryError


trial("int('abc')   ", lambda: int("abc"))                # ValueError
trial("10 / 0       ", lambda: 10 / 0)                    # ZeroDivisionError
trial("[1, 2][5]    ", lambda: [1, 2][5])                 # IndexError
trial("{}['ключ']   ", lambda: {}["ключ"])                # KeyError
trial("'1' + 1      ", lambda: "1" + 1)                   # TypeError
trial("Pin(99)      ", lambda: Pin(99))                   # ValueError: такого GP нет
trial("импорт       ", bad_import)
trial("открыть файл ", no_file)
trial("много памяти ", out_of_memory)

# Полный traceback, когда исключение поймано, но смотреть на него всё равно нужно
try:
    1 / 0
except ZeroDivisionError as e:
    sys.print_exception(e)                    # в MicroPython: sys.print_exception
```

Листинг 23.2: CircuitPython: то же

```
import board
import traceback


def trial(name, func):
    """Выполняет func и печатает, какое исключение получилось."""
    try:
        func()
        print(name, "-> ошибки нет")
    except Exception as e:                    # ловим любое «обычное» исключение
        print(name, "->", type(e).__name__, ":", e)

```

```

def bad_import():
    import no_such_module                    # ImportError


def no_file():
    open("/нет_такого_файла.txt")            # OSError, errno 2 (ENOENT)


def out_of_memory():
    bytearray(10_000_000)                    # на 264 КБ ОЗУ это MemoryError


def bad_pin():
    return board.GP99                        # в CircuitPython: AttributeError


trial("int('abc')   ", lambda: int("abc"))               # ValueError
trial("10 / 0       ", lambda: 10 / 0)                   # ZeroDivisionError
trial("[1, 2][5]    ", lambda: [1, 2][5])                # IndexError
trial("{}['ключ']   ", lambda: {}["ключ"])               # KeyError
trial("'1' + 1      ", lambda: "1" + 1)                  # TypeError
trial("board.GP99   ", bad_pin)
trial("импорт       ", bad_import)
trial("открыть файл ", no_file)
trial("много памяти ", out_of_memory)

# Полный traceback пойманного исключения (в CircuitPython 9 функция принимает само исключение)
try:
    1 / 0
except ZeroDivisionError as e:
    traceback.print_exception(e)
```

Осторожно

Не пишите «голый» `except:` и не прячьте исключение командой `pass`. Так вы заглушите сообщение об ошибке и потеряете единственную подсказку. Ловите *конкретный* тип, а в крайнем случае — `except Exception as e` и обязательно печатайте или записывайте `e`.

23.4 try, except, finally: пусть ошибка не убивает программу

Конструкция `try / except / finally` делится на три части: «попробуй», «если не вышло, сделай вот это» и «в любом случае убери за собой». Датчик может отвалиться — это нормально для реального мира. Задача программы — заметить это и продолжить работу или аккуратно остановиться.

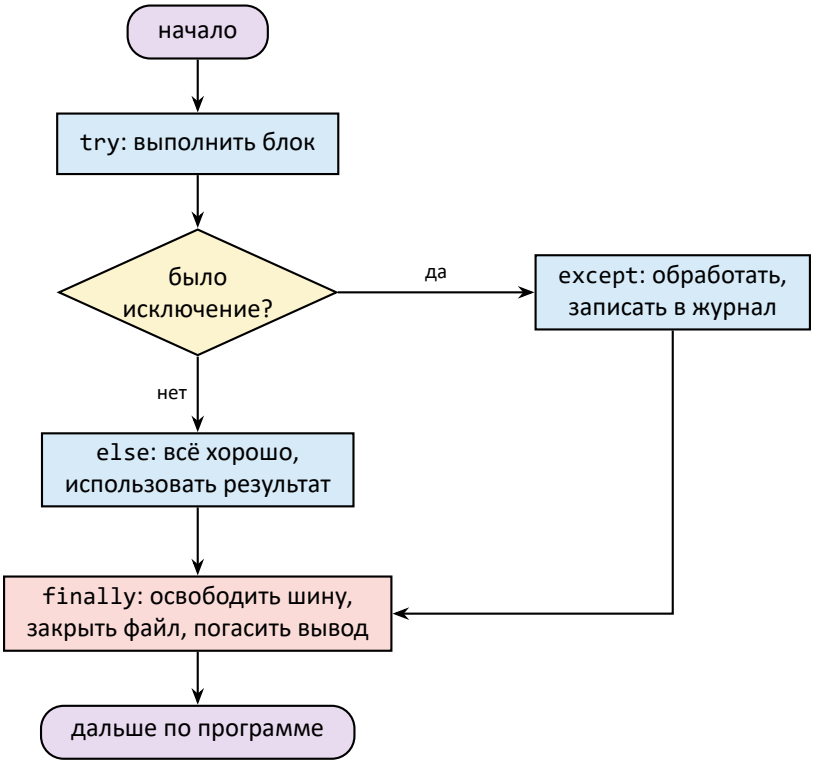


Рис. 23.4: Порядок выполнения `try`: блок `finally` срабатывает при любом исходе

В примере мы читаем идентификатор микросхемы BME280 по I²C (урок 8); регистр `0xD0` содержит число `0x60`. Если датчик не подключён, ждёт `OSError`. Какой именно номер ошибки (`errno`) будет внутри, зависит от порта и версии прошивки: чаще это 5 (`EIO`) или 19 (`ENODEV`), поэтому программа не должна опираться на конкретное число, а просто сообщает о проблеме шины.

Листинг 23.3: MicroPython: чтение датчика с обработкой ошибок

```
from machine import I2C, Pin
import time

i2c = I2C(0, sda=Pin(4), scl=Pin(5), freq=100_000)
BME_ADDR = 0x76
led = Pin("LED", Pin.OUT)


def read_chip_id(addr):
    # регистр 0xD0 у BME280 хранит идентификатор микросхемы (0x60)
    return i2c.readfrom_mem(addr, 0xD0, 1)[0]


errors = 0
for attempt in range(5):
    try:
        chip = read_chip_id(BME_ADDR)
        print("ID микросхемы: 0x%02X" % chip)
        errors = 0
    except OSError as e:
        # устройство не ответило: обрыв провода, неверный адрес, нет питания
        errors += 1
        print("Попытка", attempt + 1, "- ошибка шины I2C:", e)
    finally:
        # выполняется ВСЕГДА: и при успехе, и при ошибке
```

```
        led.toggle()
        time.sleep(0.5)

if errors >= 3:
    print("Датчик не отвечает. Проверьте провода SDA, SCL, питание и адрес.")
```

Листинг 23.4: **CircuitPython**: то же

```
import board
import busio
import digitalio
import time

BME_ADDR = 0x76
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

try:
    i2c = busio.I2C(scl=board.GP5, sda=board.GP4, frequency=100_000)
except RuntimeError as e:
    # CircuitPython проверяет подтяжки: «No pull up found on SDA or SCL»
    print("Шина I2C не создана:", e)
    raise


def read_chip_id(addr):
    buf = bytearray(1)
    while not i2c.try_lock():                # шину нужно «занять»
        pass
    try:
        i2c.writeto_then_readfrom(addr, bytes([0xD0]), buf)
    finally:
        i2c.unlock()                         # освобождаем при любом исходе
    return buf[0]


errors = 0
for attempt in range(5):
    try:
        chip = read_chip_id(BME_ADDR)
        print("ID микросхемы: 0x%02X" % chip)
        errors = 0
    except OSError as e:
        errors += 1
        print("Попытка", attempt + 1, "- ошибка шины I2C:", e)
    finally:
        led.value = not led.value
        time.sleep(0.5)

if errors >= 3:
    print("Датчик не отвечает. Проверьте провода SDA, SCL, питание и адрес.")
```

Различия: в **CircuitPython** шину нужно «занять» (`try_lock`) и обязательно освободить (`unlock`), поэтому здесь `finally` не украшение, а необходимость: без него одно исключение оставило бы шину занятой навсегда. Кроме того, при создании `busio.I2C` **CircuitPython** проверяет подтягивающие резисторы и при их отсутствии бросает `RuntimeError` с текстом «No pull up found on SDA or SCL». Это *полезная* ошибка: она сразу указывает на обрыв линии.

23.5 Учимся программировать: декораторы

Часто нужно «обернуть» много функций одним и тем же поведением: замерить время, записать вызов в журнал, поймать исключение. Копировать одни и те же три строки в каждую функцию неудобно. В Python для этого есть **декораторы**: функция, которая принимает другую функцию и возвращает её улучшенную версию. Запись `@timed` над определением — это просто сокращение для `slow_sum = timed(slow_sum)`.

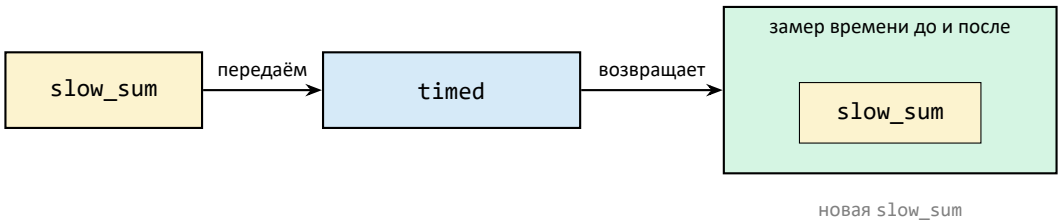


Рис. 23.5: Декоратор оборачивает функцию, не меняя её тела

Листинг 23.5: MicroPython: декоратор для замера времени

```
import time


def timed(func):
    """Декоратор: измеряет, сколько микросекунд работает функция."""
    def wrapper(*args, **kwargs):
        start = time.ticks_us()
        result = func(*args, **kwargs)
        dt = time.ticks_diff(time.ticks_us(), start)
        print("%s: %d мкс" % (func.__name__, dt))
        return result
    return wrapper


@timed                                  # то же, что slow_sum = timed(slow_sum)
def slow_sum(n):
    total = 0
    for i in range(n):
        total += i
    return total


@timed
def fast_sum(n):
    return n * (n - 1) // 2             # формула Гаусса


print(slow_sum(10000))
print(fast_sum(10000))
```

Листинг 23.6: CircuitPython: то же

```
import time


def timed(func):
    """Декоратор: измеряет, сколько микросекунд работает функция."""
    def wrapper(*args, **kwargs):
        start = time.monotonic_ns()            # наносекунды, целое число
        result = func(*args, **kwargs)
        dt = (time.monotonic_ns() - start) // 1000
        print("%s: %d мкс" % (func.__name__, dt))
        return result
    return wrapper


@timed                                         # то же, что slow_sum = timed(slow_sum)
def slow_sum(n):
    total = 0
    for i in range(n):
        total += i
    return total


@timed
def fast_sum(n):
    return n * (n - 1) // 2                    # формула Гаусса


print(slow_sum(10000))
print(fast_sum(10000))
```

Отличие только в источнике времени: в MicroPython — `ticks_us()` и безопасное вычитание `ticks_diff()` (урок 2), в CircuitPython — целочисленные наносекунды `time.monotonic_ns()`. Не используйте `time.monotonic()` для коротких интервалов: это число с плавающей точкой, а его точность падает по мере того, как плата работает всё дольше.

23.6 print-отладка и журнал

`print()` — самый старый и всё ещё самый популярный инструмент отладки. Но у него два недостатка: сообщения исчезают, как только вы отключите USB, а когда их много — в них тонет главное. Лучше завести **журнал** (log). У каждого сообщения есть **уровень важности**, а программа задаёт **порог**: всё, что ниже порога, выбрасывается.

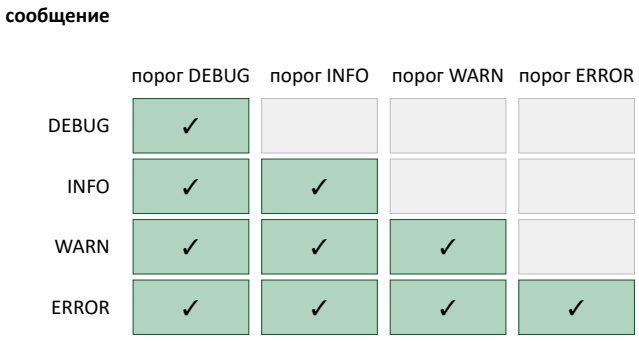


Рис. 23.6: Какие сообщения попадают в журнал при разных порогах (галочка — сообщение выводится)

Уровень	Когда использовать	Пример
DEBUG	подробности для разработчика	шаг 3, x=17
INFO	штатные события	Wi-Fi подключён
WARN	что-то необычное, но работаем	датчик ответил с третьей попытки
ERROR	операция не удалась	ошибка записи на SD

Таблица 23.2: Уровни важности сообщений

Стандартный модуль `logging` есть в наборе `micropython-lib` (устанавливается через `mip`), а для **CircuitPython** существует библиотека `adafruit_logging` (через `circup`). Но принцип так прост, что мы напишем свой маленький журнал на 40 строк. Он умеет: фильтровать по уровню, выводить в консоль, дописывать в файл и делать **ротацию**: когда файл превышает заданный размер, он переименовывается в `log.txt.1`, прежний `.1` становится `.2`, а самый старый удаляется. Так журнал никогда не займёт всю память.

Листинг 23.7: **MicroPython**: журнал с уровнями и ротацией

```
import os
import time

DEBUG, INFO, WARN, ERROR = 10, 20, 30, 40
NAMES = {DEBUG: "DEBUG", INFO: "INFO", WARN: "WARN", ERROR: "ERROR"}


class Logger:
    """Простой журнал: уровни, вывод на консоль и в файл, ротация по размеру."""

    def __init__(self, path="log.txt", level=INFO, max_bytes=4096, keep=2):
        self.path = path
        self.level = level
        self.max_bytes = max_bytes
        self.keep = keep                        # сколько старых файлов хранить

    def _rotate(self):
        try:
            size = os.stat(self.path)[6]        # [6] - размер файла в байтах
        except OSError:
            return                              # файла ещё нет
        if size < self.max_bytes:
            return
        try:
            os.remove("%s.%d" % (self.path, self.keep))
        except OSError:
            pass
        for i in range(self.keep - 1, 0, -1):  # log.txt.1 -> log.txt.2 и т. д.
            try:
                os.rename("%s.%d" % (self.path, i), "%s.%d" % (self.path, i + 1))
            except OSError:
                pass
        os.rename(self.path, self.path + ".1")

    def log(self, level, text):
        if level < self.level:                  # сообщение ниже порога - выбрасываем
            return
        line = "%8d %-5s %s" % (time.ticks_ms(), NAMES[level], text)
        print(line)
        self._rotate()
        with open(self.path, "a") as f:         # открыли, дописали, закрыли
```

```
            f.write(line + "\n")

    def debug(self, text): self.log(DEBUG, text)
    def info(self, text): self.log(INFO, text)
    def warn(self, text): self.log(WARN, text)
    def error(self, text): self.log(ERROR, text)


log = Logger(level=DEBUG)
log.info("Запуск программы")
for n in range(1, 6):
    log.debug("шаг %d" % n)
    time.sleep(0.2)
try:
    1 / 0
except ZeroDivisionError as e:
    log.error("ошибка расчёта: %s" % e)
log.warn("конец демонстрации")
```

Листинг 23.8: CircuitPython: журнал с уровнями и ротацией

```
import os
import time

DEBUG, INFO, WARN, ERROR = 10, 20, 30, 40
NAMES = {DEBUG: "DEBUG", INFO: "INFO", WARN: "WARN", ERROR: "ERROR"}


class Logger:
    """Простой журнал: уровни, вывод на консоль и в файл, ротация по размеру."""

    def __init__(self, path="/log.txt", level=INFO, max_bytes=4096, keep=2):
        self.path = path
        self.level = level
        self.max_bytes = max_bytes
        self.keep = keep

    def _rotate(self):
        try:
            size = os.stat(self.path)[6]
        except OSError:
            return
        if size < self.max_bytes:
            return
        try:
            os.remove("%s.%d" % (self.path, self.keep))
        except OSError:
            pass
        for i in range(self.keep - 1, 0, -1):
            try:
                os.rename("%s.%d" % (self.path, i), "%s.%d" % (self.path, i + 1))
            except OSError:
                pass
        os.rename(self.path, self.path + ".1")

    def log(self, level, text):
        if level < self.level:
            return
```

```
        line = "%9.2f %-5s %s" % (time.monotonic(), NAMES[level], text)
        print(line)
        try:
            self._rotate()
            with open(self.path, "a") as f:
                f.write(line + "\n")
        except OSError as e:
            # Файловая система доступна компьютеру: запись из программы запрещена.
            # См. boot.py ниже. Журнал на консоль при этом продолжает работать.
            print("(журнал в файл не пишется:", e, ")")

    def debug(self, text): self.log(DEBUG, text)
    def info(self, text): self.log(INFO, text)
    def warn(self, text): self.log(WARN, text)
    def error(self, text): self.log(ERROR, text)


log = Logger(level=DEBUG)
log.info("Запуск программы")
for n in range(1, 6):
    log.debug("шаг %d" % n)
    time.sleep(0.2)
try:
    1 / 0
except ZeroDivisionError as e:
    log.error("ошибка расчёта: %s" % e)
log.warn("конец демонстрации")
```

Осторожно

Файловая система CircuitPython. Пока плата подключена к компьютеру как диск `CIRCUITPY`, программа не может писать в файлы: попытка даёт `OSError: [Errno 30] Read-only filesystem`. Чтобы писать журнал, нужен файл `boot.py` с командой `storage.remount("/", readonly=False)`; тогда диск станет доступен компьютеру только для чтения, и файлы придётся менять другим способом (Thonny, Mu или временно убрав `boot.py`). Подробнее — урок 12. Кроме того, не пишите в файл чаще, чем нужно: флеш-память выдерживает около ста тысяч циклов стирания на сектор.

Листинг 23.9: CircuitPython: `boot.py` — разрешаем программе писать в файлы

```
# boot.py - выполняется один раз при старте, ДО подключения USB-диска.
import storage

# Отдаём файловую систему программе: она сможет писать журнал.
# Расплата: компьютер теперь видит диск CIRCUITPY только для чтения.
storage.remount("/", readonly=False)
```

23.7 Память: куча, сборщик мусора и «утечки»

Память RP2040 — всего 264 КБ, и MicroPython или CircuitPython занимают из неё заметную часть. Python выделяет память под каждое число, строку и список в области, которая называется **куча**. Освобождением занимается **сборщик мусора** (garbage collector, GC): он находит объекты, на которые больше нет ссылок, и возвращает их память. Он запускается сам, но вы можете позвать его командой `gc.collect()`.

«Утечка» в Python — это не потерянные байты, как в C, а *забытые ссылки*: например, список, в который программа дописывает измерение за измерением и никогда не очищает. Сборщик не

вправе выбросить то, на что есть ссылка, и свободная память неуклонно тает, пока не придёт `MemoryError`. Часто это случается через часы или дни работы, и на столе ошибка не видна.

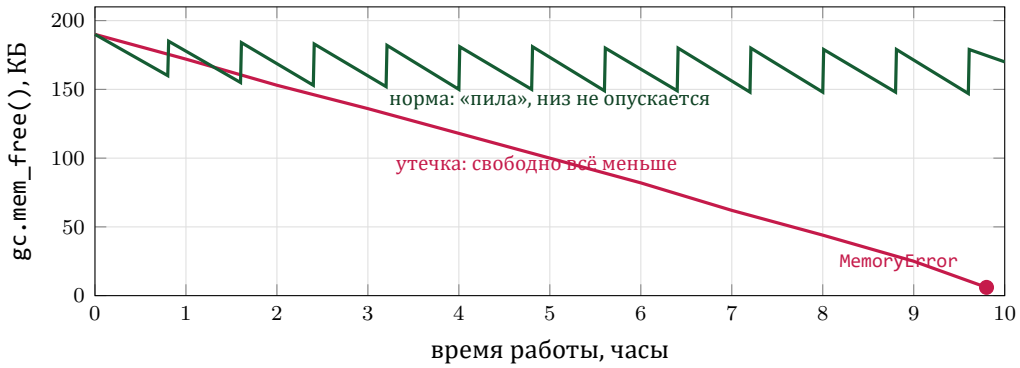


Рис. 23.7: Свободная память во времени: «пила» (норма) и непрерывное падение (утечка). Значения условные

Следить за памятью умеют две функции: `gc.mem_free()` (сколько свободно) и `gc.mem_alloc()` (сколько занято). Только в **MicroPython** есть и подробная карта кучи: `micropython.mem_info()` печатает общую статистику и строку символов, где каждый символ — блок памяти (точка — свободный, буква — занятый).
Три правила бережливой работы с памятью:

- заранее создавайте буферы (`bytearray`) и *переиспользуйте* их, а не создавайте новые в цикле;
- не накапливайте данные в списках без предела: ограничивайте размер или используйте кольцевой буфер;
- перед замером вызывайте `gc.collect()`, иначе вы измерите мусор вместе с полезными данными.

Листинг 23.10: MicroPython: замеры памяти, «плохой» список и кольцевой буфер

```
import gc
import micropython


def report(tag):
    gc.collect()                                # сначала уборка, потом честный замер
    print("%-18s свободно %6d, занято %6d" % (tag, gc.mem_free(), gc.mem_alloc()))


report("старт")

# ПЛОХО: список растёт бесконечно - «утечка» (на деле - забытые ссылки)
history = []
for i in range(40):
    history.append(bytearray(500))              # 500 байт на каждое измерение
report("после 40 измерений")

history = None                                  # отпустили ссылку - память вернётся
report("список удалён")

# ХОРОШО: буфер фиксированного размера, который переиспользуется
BUF_SIZE = 40
ring = [0] * BUF_SIZE
pos = 0
for i in range(1000):
```

```
    ring[pos] = i                               # старые значения затираются новыми
    pos = (pos + 1) % BUF_SIZE
report("кольцевой буфер")

# Подробная карта кучи (только MicroPython): каждый символ - блок памяти
micropython.mem_info()
```

Листинг 23.11: CircuitPython: то же, без `mem_info()`

```
import gc


def report(tag):
    gc.collect()                                # сначала уборка, потом честный замер
    print("%-18s свободно %6d, занято %6d" % (tag, gc.mem_free(), gc.mem_alloc()))


report("старт")

# ПЛОХО: список растёт бесконечно - «утечка» (на деле - забытые ссылки)
history = []
for i in range(40):
    history.append(bytearray(500))              # 500 байт на каждое измерение
report("после 40 измерений")

history = None                                  # отпустили ссылку - память вернётся
report("список удалён")

# ХОРОШО: буфер фиксированного размера, который переиспользуется
BUF_SIZE = 40
ring = [0] * BUF_SIZE
pos = 0
for i in range(1000):
    ring[pos] = i
    pos = (pos + 1) % BUF_SIZE
report("кольцевой буфер")

# Модуля micropython с функцией mem_info() в CircuitPython нет:
# пользуйтесь gc.mem_free() и gc.mem_alloc().
```

23.8 Мультиметр: первый прибор

Половина «программных» ошибок на самом деле электрические. Мультиметр стоит дёшево и находит их за минуту. Три режима, которых хватает в 90 % случаев:

- **Напряжение** (V, постоянное). Щупы подключаются *параллельно* участку цепи. Между пинами 3V3 и GND должно быть около $3{,}3$ V, между VBUS и GND — около 5 V. Если на выводе 0 V, а должно быть 3,3, то либо нет питания, либо вывод сконфигурирован не так, как вы думаете.
- **Прозвонка** (значок диода или динамика). Прибор пищит, когда между щупами есть проводящий путь. Так проверяют провода, дорожки, перемычки на макетной плате, замыкания между 3V3 и GND.
- **Ток** (A). Прибор включается *в разрыв* цепи, последовательно. Это самый опасный режим: вход амперметра почти не имеет сопротивления.

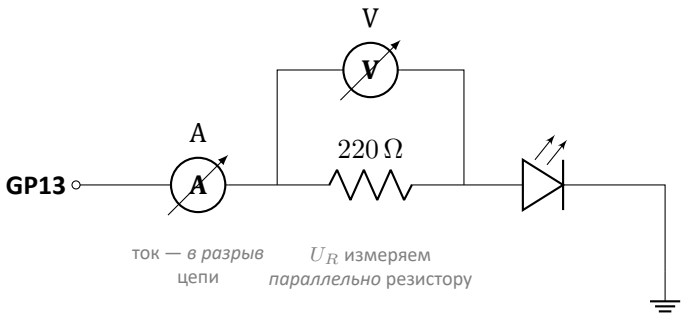


Рис. 23.8: Вольтметр подключают параллельно, амперметр — последовательно

Осторожно

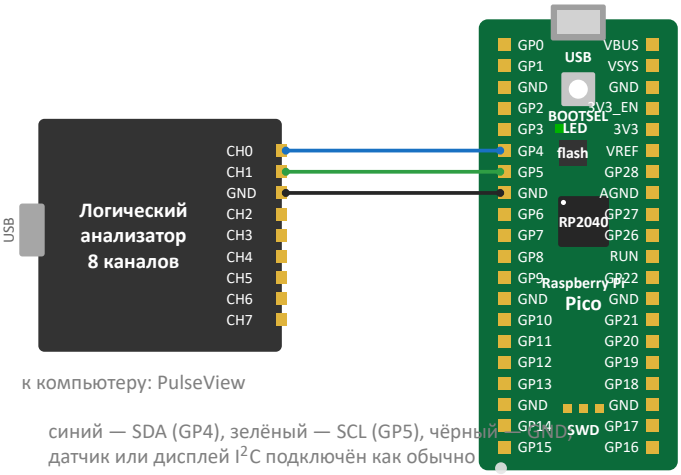
Никогда не включайте мультиметр в режим измерения тока и не подключайте щупы *параллельно* источнику питания или между 3V3 и GND: это короткое замыкание через прибор. Если вы только что мерили ток, **верните щупы в гнёзда напряжения**, пока не забыли. Щупы для измерения тока и напряжения втыкаются в разные гнёзда.

Совет

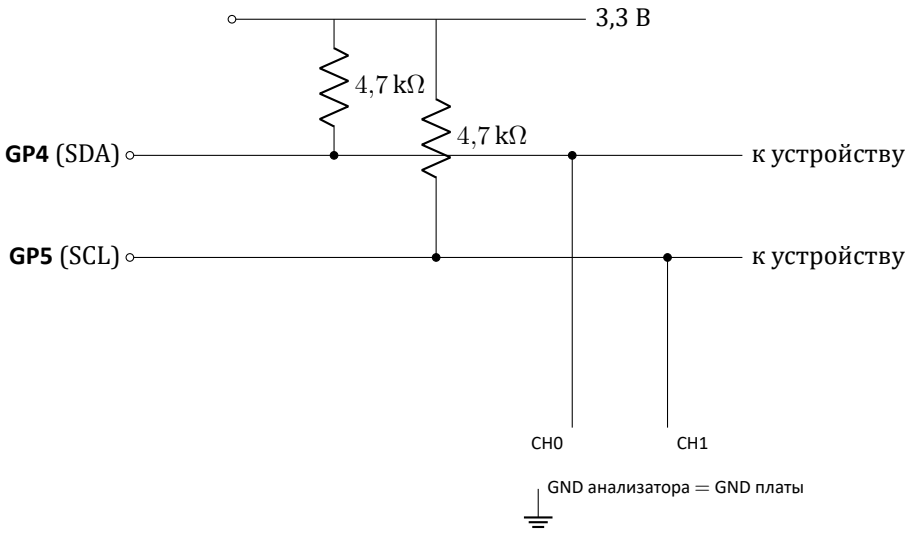
Простая проверка питания при «зависаниях»: подключите вольтметр между VSYS и GND (или 3V3 и GND) и запустите программу, потребляющую много тока (Wi-Fi, мотор). Если напряжение в этот момент проседает ниже 3 V, виновато питание (тонкий кабель, USB-порт без достаточного тока, длинные провода), а не код.

23.9 Логический анализатор: видим то, чего не видит print

Мультиметр показывает среднее напряжение. Но цифровые сигналы быстрые: I^2C на 100 kHz переключается 100 000 раз в секунду, SPI — миллионы раз. Увидеть их помогает **логический анализатор**: прибор с несколькими входами, который много раз в секунду запоминает «0 или 1» на каждом входе и показывает результат как временную диаграмму. Недорогие анализаторы на 8 каналов и 24 MHz стоят в несколько раз дешевле осциллографа и работают с бесплатной программой **PulseView** (проект **sigrok**); она умеет сразу *расшифровывать* протоколы I^2C, SPI, UART: показывает не нули и единицы, а «адрес 0x3C, запись, подтверждение получено».



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 23.9: Подключение логического анализатора к шине I²C на выводах GP4 и GP5

Правила подключения: **общий GND** обязателен (иначе уровни не с чем сравнивать); входы анализатора подключаются к проводам *параллельно* устройствам, ничего не разрывая; проверьте, что анализатор рассчитан на логику 3,3 В (большинство недорогих терпят и 5 В, но смотрите описание); частота дискретизации должна быть минимум в 4 раза, лучше в 10 раз выше частоты сигнала: для I²C на 100 kHz выбирайте 1 – 4 MHz.

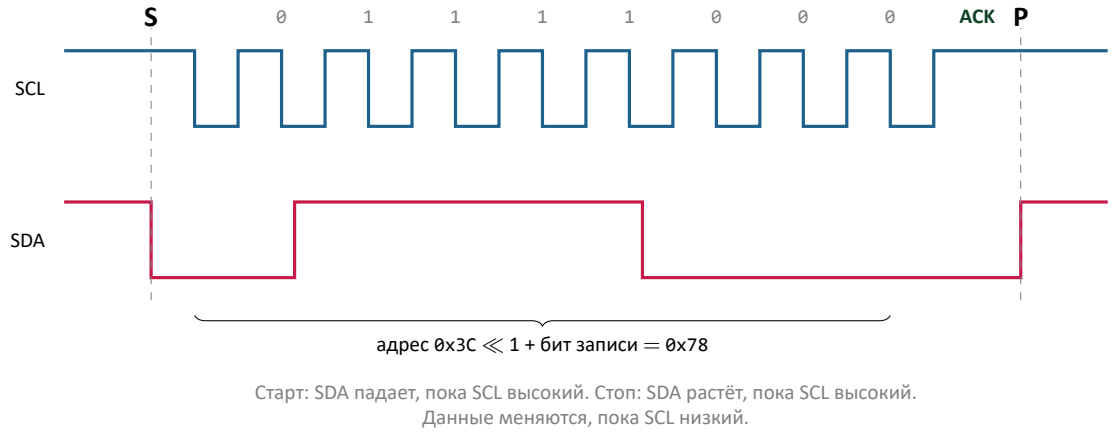


Рис. 23.10: Как выглядит начало обмена по I²C на логическом анализаторе: старт, адрес устройства 0x3C (OLED), подтверждение, стоп

Обратите внимание на девятый тактовый импульс: SDA в этот момент тянет вниз *сам дисплей* — это подтверждение (ACK). Если вместо него вы видите высокий уровень (NACK), устройство не ответило: неверный адрес, нет питания или оборван провод. Без анализатора вы бы узнали об этом только по сообщению `OSError`; с ним — видите, что именно отправила плата и почему ответа нет.

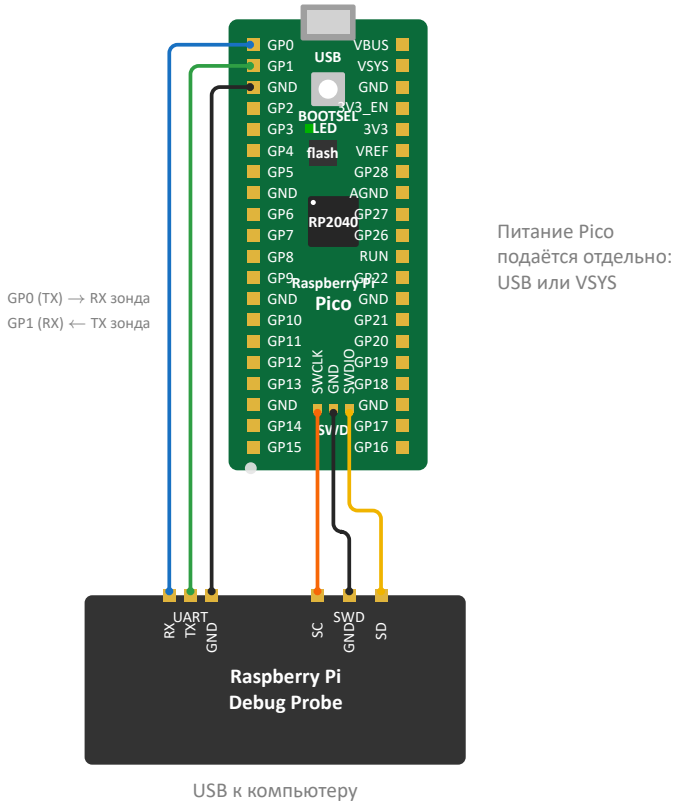
Это интересно

Саму Pico можно превратить в логический анализатор: существуют проекты (например, **sigrok-pico**), которые превращают плату в 8-канальный анализатор, совместимый с PulseView. Для задач класса «посмотреть, как переключается сигнал» этого вполне достаточно, но частоты и надёжность хуже, чем у специального прибора. Состояние проектов и поддерживаемые версии смотрите в их репозиториях.

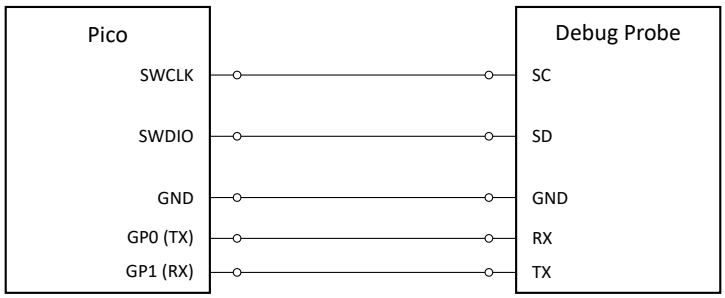
23.10 Аппаратная отладка: SWD и Raspberry Pi Debug Probe

Обычно код отлаживают через USB-консоль. Но бывают ситуации, когда USB недоступен: программа зависла так, что консоль молчит; нужно остановиться в произвольной точке и посмотреть значения; вы пишете сам загрузчик или прошивку на C. Для этого у RP2040 и RP2350 есть интерфейс **SWD** (Serial Wire Debug): два сигнальных провода (**SWCLK** — тактовый и **SWDIO** — данные) плюс земля. Пока ядро процессора *остановлено*, через SWD можно читать и писать память, ставить точки останова, шагать по инструкциям и перепрошивать плату.

Адаптер для SWD называется **отладочным зондом**. Официальный — **Raspberry Pi Debug Probe**: сам построен на RP2040, подключается по USB и умеет два дела: отладку по SWD (протокол CMSIS-DAP) и мост USB–UART. У оригинальной Pico три контакта SWD (SWCLK, GND, SWDIO) выведены на отдельный разъём-пазы под платой с надписью SWD (на платах с впаянным разъёмом удобнее всего подключаться к нему).



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 23.11: Подключение Raspberry Pi Debug Probe к Pico: SWD (три провода) и UART (по желанию)

Провода SWD к зонду идут «один в один»: SWCLK к SC, SWDIO к SD, GND к GND. Так как у зонда два сигнальных провода и земля, а UART-канал не влияет на отладку, его подключают только когда нужен вывод текста на компьютер (например, программа на C печатает в UART). У зонда есть собственный вывод общего GND.

Для работы с отладчиком используется программа **OpenOCD**: она соединяется с зондом, представляется GDB-серверу и позволяет перепрошивать плату и останавливать ядро. Типичная команда для RP2040 выглядит так (форк OpenOCD от Raspberry Pi уже содержит нужные файлы; для RP2350 используется `target/rp2350.cfg`):

```
openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed 5000"
```

Для сценариев MicroPython и CircuitPython отладка по SWD нужна редко: к вашему питоновскому коду точек останова таким способом не поставить (видна работа интерпретатора, а не вашей программы). Зато SWD спасает, когда плата «окирпичена» или нужно прошить свою C-прошивку. Для повседневной работы хватает консоли, журнала и анализатора.

Осторожно

Не подключайте провод питания зонда к плате, которая уже питается от USB или батареи: две разные цепи питания соединять нельзя. У штатного кабеля зонда отдельного провода питания нет — не добавляйте его.

23.11 Тесты: пусть компьютер проверяет вас

Каждый раз после правки запускать программу и смотреть глазами — долго и ненадёжно. Лучше написать **тесты**: маленькие функции, которые вызывают ваш код с известными входными данными и проверяют результат оператором `assert`. Если условие ложно, возникает `AssertionError` с вашим сообщением.

Тестировать удобнее всего *чистые функции*: те, что только считают и не трогают оборудование. Переводы кода АЦП в вольты, градусы, проценты; разбор строки от датчика; расчёт задержки. Железо потом проверяется глазами и приборами. Дополнительный плюс: чистые функции выполняются на компьютере тоже, поэтому тесты можно гонять за секунду.

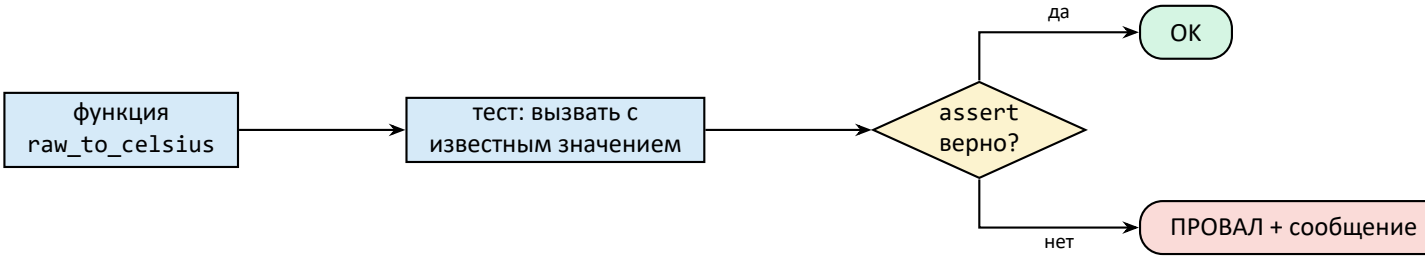


Рис. 23.12: Как работает автоматический тест

В нашем мини-раннере каждый тест — обычная функция с именем `test_...`, а раннер вызывает все по очереди и печатает «OK» или «ПРОВАЛ». Одна из проверок намеренно ложна: посмотрите, как сообщение попадает на экран. Для **MicroPython** полноценный `unittest` доступен в `micropython-lib` (`mip.install("unittest")`), но функциональность и поддерживаемые версии менялись, поэтому для маленьких проектов проще и надёжнее обойтись собственным раннером. В **CircuitPython** стандартного `unittest` нет.

Листинг 23.12: **MicroPython**: тесты на `assert` и свой раннер

```
def raw_to_celsius(raw):
    """Температура встроенного датчика RP2040 по коду АЦП (12 бит, опора 3,3 В)."""
    volts = raw * 3.3 / 4095
    return 27 - (volts - 0.706) / 0.001721


def clamp(x, low, high):
    return max(low, min(high, x))


def test_room_value():
    assert abs(raw_to_celsius(876) - 27) < 1, "около 27 градусов при коде 876"


def test_hotter_means_lower_voltage():
    # у этого датчика с ростом температуры напряжение ПАДАЕТ
    assert raw_to_celsius(850) > raw_to_celsius(900)


```

```
def test_clamp():
    assert clamp(5, 0, 10) == 5
    assert clamp(-3, 0, 10) == 0
    assert clamp(99, 0, 10) == 10


def test_that_fails():
    assert clamp(5, 0, 3) == 5, "намеренно неверное ожидание"


def run_tests(tests):
    failed = 0
    for test in tests:
        try:
            test()
            print("OK    ", test.__name__)
        except AssertionError as e:
            failed += 1
            print("ПРОВАЛ", test.__name__, "-", e)
    print("Провалено тестов:", failed, "из", len(tests))
    return failed == 0


run_tests([test_room_value, test_hotter_means_lower_voltage,
           test_clamp, test_that_fails])
```

Листинг 23.13: CircuitPython: тот же приём

```
# Чистые функции без обращения к железу работают одинаково и на компьютере,
# и в MicroPython, и в CircuitPython - их удобнее всего проверять тестами.
# Для CircuitPython формулу датчика RP2040 не используют: там есть готовое
# microcontroller.cpu.temperature, поэтому тестируем преобразование АЦП в вольты.

def raw_to_volts(raw, vref=3.3):
    """Код 16-битного analogio.AnalogIn (0..65535) в вольты."""
    return raw * vref / 65535


def clamp(x, low, high):
    return max(low, min(high, x))


def test_zero_and_full_scale():
    assert raw_to_volts(0) == 0
    assert abs(raw_to_volts(65535) - 3.3) < 1e-6, "полная шкала равна опоре"


def test_half_scale():
    assert abs(raw_to_volts(32768) - 1.65) < 0.001


def test_clamp():
    assert clamp(5, 0, 10) == 5
    assert clamp(-3, 0, 10) == 0
    assert clamp(99, 0, 10) == 10


def test_that_fails():
```

```
    assert clamp(5, 0, 3) == 5, "намеренно неверное ожидание"


def run_tests(tests):
    failed = 0
    for test in tests:
        try:
            test()
            print("OK    ", test.__name__)
        except AssertionError as e:
            failed += 1
            print("ПРОВАЛ", test.__name__, "-", e)
    print("Провалено тестов:", failed, "из", len(tests))
    return failed == 0


run_tests([test_zero_and_full_scale, test_half_scale,
           test_clamp, test_that_fails])
```

Так как функции без обращения к железу одинаковы в обоих языках, различается лишь то, что именно проверяется: у RP2040 есть формула для встроенного датчика температуры (в **MicroPython**), а **CircuitPython** даёт готовое значение `microcontroller.cpu.temperature`, поэтому тестируем перевод кода АЦП в вольты.

23.12 Сторожевой таймер: «собака», которая перезагрузит плату

Что делать, если программа всё-таки зависла, например, в бесконечном цикле или в ожидании ответа, который не придёт? Устройство, висящее в шкафу или на крыше, перезагрузить кнопкой некому. Для этого в микроконтроллер встроен **сторожевой таймер** (watchdog, «сторожевая собака»). Это обратный счётчик времени. Программа обязана периодически *«кормить собаку»*, то есть вызывать `feed()`, и счётчик начинается заново. Если программа зависла, счётчик дойдёт до нуля, и микроконтроллер **перезагрузится** сам.

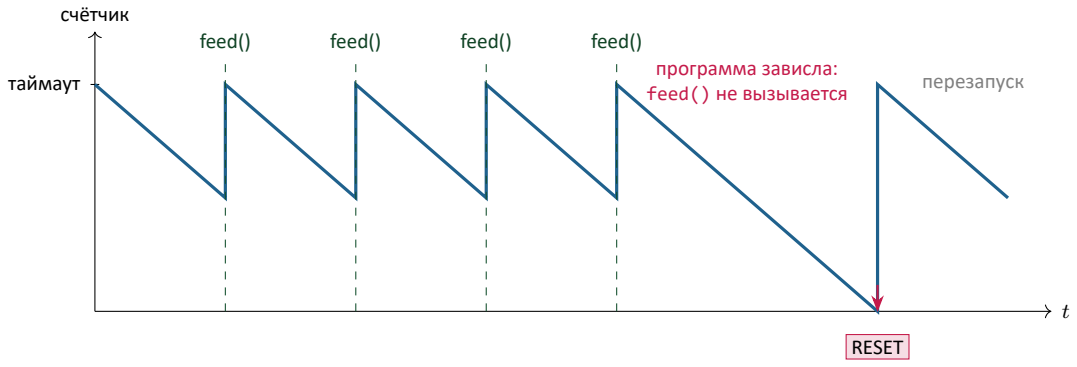


Рис. 23.13: Работа сторожевого таймера: пока программа вовремя «кормит» таймер, он не срабатывает

В **MicroPython** сторожевой таймер создаётся классом `machine.WDT(timeout=мс)`; максимальный таймаут на RP2040 составляет около 8 с (около 8388 мс). Выключить таймер после запуска *нельзя* (только перезагрузкой). В **CircuitPython** используется `microcontroller.watchdog`: задаются `timeout` в секундах и `mode = WatchDogMode.RESET`, а метод `deinit()` выключает таймер.
Хорошая программа ещё и *выясняет*, почему перезагрузилась: по причине сброса (`machine.reset_cause()` в **MicroPython**, `microcontroller.cpu.reset_reason` в **CircuitPython**) она может записать событие в журнал, зажечь индикатор или включить безопасный режим работы. В примере светодиод быстро мигает при старте после срабатывания таймера.

Листинг 23.14: MicroPython: сторожевой таймер и причина сброса

```
import machine
import time
from machine import WDT, Pin

led = Pin("LED", Pin.OUT)

# Почему плата перезагрузилась в прошлый раз?
if machine.reset_cause() == machine.WDT_RESET:
    print("Предыдущий запуск прервал сторожевой таймер!")
    for _ in range(6):                          # быстро мигаем: сигнал «был сбой»
        led.toggle()
        time.sleep(0.1)

wdt = WDT(timeout=3000)                         # 3 с; максимум на RP2040 около 8 с
print("Сторожевой таймер запущен")

for i in range(10):
    led.toggle()
    time.sleep(0.5)
    wdt.feed()                                  # «Я жива!» - счётчик начинается заново

print("Имитируем зависание: feed() больше не вызывается")
while True:
    time.sleep(0.1)                             # через 3 с плата перезагрузится сама
```

Листинг 23.15: CircuitPython: сторожевой таймер и причина сброса

```
import board
import digitalio
import microcontroller
import time
from watchdog import WatchDogMode

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

# Почему плата перезагрузилась в прошлый раз?
if microcontroller.cpu.reset_reason == microcontroller.ResetReason.WATCHDOG:
    print("Предыдущий запуск прервал сторожевой таймер!")
    for _ in range(6):
        led.value = not led.value
        time.sleep(0.1)

wd = microcontroller.watchdog
wd.timeout = 3.0                                # секунды, дробные значения допустимы
wd.mode = WatchDogMode.RESET                    # по истечении времени - перезагрузка
print("Сторожевой таймер запущен")

try:
    for i in range(10):
        led.value = not led.value
        time.sleep(0.5)
        wd.feed()                               # «Я жива!»

    print("Имитируем зависание: feed() больше не вызывается")
```

```
    while True:
        time.sleep(0.1)                          # через 3 с плата перезагрузится сама
finally:
    wd.deinit()                                  # при штатном выходе (Ctrl+C) таймер выключаем
```

Листинг 23.16: MicroPython: все причины сброса

```
import machine

# Причины сброса: имена констант зависят от порта, поэтому getattr с запасным None
NAMES = (
    ("PWRON_RESET", "включение питания"),
    ("HARD_RESET", "аппаратный сброс (RUN, кнопка)"),
    ("WDT_RESET", "сторожевой таймер"),
    ("SOFT_RESET", "программный сброс (Ctrl+D)"),
)

cause = machine.reset_cause()
for const, text in NAMES:
    if getattr(machine, const, None) == cause:
        print("Причина последнего сброса:", text)
        break
else:
    print("Причина сброса: код", cause)
```

Листинг 23.17: CircuitPython: причина сброса, причина запуска, причина безопасного режима

```
import microcontroller
import supervisor

# Причина последнего аппаратного сброса
print("reset_reason:", microcontroller.cpu.reset_reason)

# Почему запущен код: включение, автоперезагрузка при сохранении файла, Ctrl+D...
print("run_reason:  ", supervisor.runtime.run_reason)

# Если плата вошла в безопасный режим, здесь будет причина (CircuitPython 9)
print("safe_mode:   ", supervisor.runtime.safe_mode_reason)

# Следующую перезагрузку можно направить в особый режим:
#   microcontroller.on_next_reset(microcontroller.RunMode.SAFE_MODE)
#   microcontroller.reset()
```

Осторожно

Не «кормите» таймер из отдельного таймерного прерывания или фонового потока: тогда он будет кормиться, даже если основная программа зависла, и потеряет смысл. Вызывайте `feed()` в главном цикле *после* того, как все важные действия выполнены успешно. Таймаут выбирайте с запасом на самый долгий штатный шаг цикла (например, на подключение к Wi-Fi).

23.13 CircuitPython: безопасный режим и восстановление

CircuitPython старается защитить вас от самых неприятных ситуаций. Если прошивка обнаруживает серьёзную неполадку (напряжение питания упало ниже допустимого, аппаратный сбой,

внутренняя ошибка), она переходит в **безопасный режим** (safe mode): `boot.py` и `code.py` *не запускаются*, а в консоль выводится причина. Так вы можете подключиться к плате и исправить программу, а не получать бесконечную перезагрузку. Способ индикации безопасного режима (мигание светодиода) зависит от платы; причину всегда можно прочитать в консоли или узнать из программы.

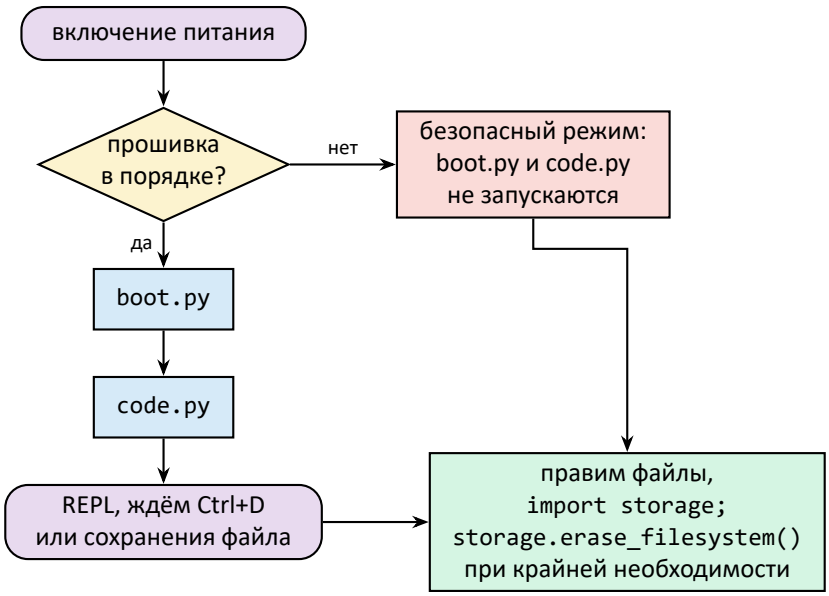


Рис. 23.14: Загрузка **CircuitPython**: нормальный путь и безопасный режим

Если вы сами хотите попасть в безопасный режим (чтобы ничего не стартовало), введите в REPL `import microcontroller`, затем `microcontroller.on_next_reset(microcontroller.RunMode.SAFE_MODE)` и `microcontroller.reset()`. Если не помогает ничего, удаление и пересоздание файловой системы выполняет `storage.erase_filesystem()` (**все файлы на плате будут стёрты!**).

В **MicroPython** специального безопасного режима нет. Если `main.py` зависает до того, как вы успели подключиться, самый простой выход — «режим спасения», вручную проверяемый в начале программы: если вывод замкнут на землю (перемычка), программа не запускается. Настоящий код при этом лежит в отдельном файле `app.py`. Приём работает в обоих языках: в **CircuitPython** вместо `main.py` используется `code.py`.

Листинг 23.18: **MicroPython**: «сторож» с режимом спасения

```
# main.py - только «сторож»: настоящая программа лежит в app.py
import sys
import time
from machine import Pin

rescue = Pin(15, Pin.IN, Pin.PULL_UP)          # перемычка GP15 - GND = режим спасения
time.sleep_ms(50)                              # даём подтяжке установиться

if rescue.value() == 0:
    print("Режим спасения: app.py не запущен, REPL свободен")
    sys.exit()                                 # main.py завершается, остаётся консоль

import app                                     # обычный запуск: выполнится app.py
```

Листинг 23.19: CircuitPython: то же

```
# code.py - только «сторож»: настоящая программа лежит в app.py
import sys
import time
import board
import digitalio

rescue = digitalio.DigitalInOut(board.GP15)    # перемычка GP15 - GND = режим спасения
rescue.direction = digitalio.Direction.INPUT
rescue.pull = digitalio.Pull.UP
time.sleep(0.05)                               # даём подтяжке установиться

if not rescue.value:
    print("Режим спасения: app.py не запущен, REPL свободен")
    rescue.deinit()
    sys.exit()                                 # code.py завершается, остаётся консоль

rescue.deinit()                                # отпускаем вывод для основной программы
import app                                     # обычный запуск: выполнится app.py
```

Полный сброс flash

Крайний случай: файловая система повреждена или в `main.py`/`code.py` ошибка, которая не даёт зайти на плату. Тогда стирают всю flash-память. Для этого с сайта Raspberry Pi скачивается файл `flash_nuke.uf2` (для Pico 2 и RP2350 используйте версию для RP2350, она лежит на той же странице документации): удерживая BOOTSEL, подключите плату к USB и скопируйте файл на появившийся диск `RPI-RP2` (у Pico 2 — `RP2350`). Плата сотрёт flash и перезагрузится в режим BOOTSEL. После этого нужно заново записать прошивку MicroPython или CircuitPython (урок 1). **Все ваши файлы пропадут**, поэтому перед такими экспериментами делайте копии программ на компьютере.

23.14 Чек-лист «ничего не работает»

Когда не помогло ничего, идите по списку сверху вниз. Он основан на статистике: самые частые причины идут первыми.

1. **Кабель и порт.** Кабель «только для зарядки»? Другой порт, другой кабель.
2. **Питание.** Горит ли светодиод платы? Мультиметром: 3V3 и GND $\approx 3{,}3$ V.
3. **Сначала запустите самое простое.** Мигает ли встроенный светодиод (урок 2)?
4. **Тот ли файл запущен?** Старый `main.py`/`code.py` мог остаться; остановите его (Ctrl+C) и запустите нужный.
5. **Читаем traceback** снизу вверх. Тип, сообщение, строка.
6. **Номера выводов.** В программе GP-номера, а не номера контактов; сверьте с распиновкой.
7. **Земля.** Общий GND у всех устройств? Нет земли — нет работы.
8. **Провода и контакты.** Прозвоните каждый провод мультиметром; переставьте деталь в другие отверстия.
9. **Библиотеки.** Версии совпадают? Файлы в `lib`? Не мешает ли локальный файл с тем же именем, что и модуль?
10. **Память.** `gc.mem_free()` не близка к нулю?

11. **Шины.** `i2c.scan()` видит устройство? Логический анализатор показывает ACK?
12. **Упростите.** Отключите всё, кроме проблемной детали; запустите минимальный пример из документации.
13. **Спросите.** Хороший вопрос содержит: плату и версию прошивки, схему, минимальный код и *полный* текст ошибки.

Если не работает

- **Ошибка пропала сама.** Это не победа: это перемежающийся дефект (плохой контакт, нехватка питания, утечка памяти). Включите журнал и ждите повторения.
- **В журнале пусто или нет файла в CircuitPython** — вы не добавили `boot.py` с `storage.remount` или сообщения ниже порога.
- **`MemoryError` при импорте большой библиотеки** — вызовите `gc.collect()` до импорта, импортируйте только нужное, перенесите тяжёлые библиотеки в `.mpy` (скомпилированный вид).
- **Плата перезагружается каждые несколько секунд** — не вызывается `feed()`, либо просадка питания (особенно при Wi-Fi или моторах), либо ошибка и автоперезапуск.
- **Логический анализатор показывает мусор** — нет общей земли, слишком низкая частота дискретизации, неверный порог напряжения или не тот канал.
- **Debug Probe не видит плату** — перепутаны SWCLK и SWDIO, плата без питания, слишком длинные провода (сократите) или не подключена земля.
- **Файл не сохраняется (CircuitPython)** — диск открыт только для чтения; файловая система занята программой или компьютером.

Итоги урока

- Отладка — это метод: воспроизвести, сузить, выдвинуть гипотезу, менять одно за раз.
- Traceback читают снизу вверх: тип и сообщение, потом строка, потом цепочка вызовов.
- `try/except/finally` не даёт ошибке убить программу, а `finally` освобождает ресурсы при любом исходе.
- Журнал с уровнями и ротацией переживает отключение USB; следите за памятью через `gc.mem_free()`.
- Мультиметр проверяет питание и контакты, логический анализатор показывает, что на самом деле творится на шинах, SWD и Debug Probe нужны для глубокой отладки и «спасения» платы.
- Тесты на `assert` проверяют чистые функции, а сторожевой таймер перезагружает зависшее устройство.
- В CircuitPython есть безопасный режим; полный сброс flash — `flash_nuke.uf2`.

Проверь себя

1. В каком порядке читают traceback и что важнее всего в нём найти?
2. Чем отличается ловля `except Exception` от «голого» `except:` и почему `KeyboardInterrupt` лучше не ловить?
3. Что такое ротация журнала и зачем она нужна на плате с ограниченной памятью?

4. Почему при подключённом USB программа на **CircuitPython** не может записывать файлы и как это исправить?
5. Как работает сторожевой таймер и почему его нельзя «кормить» из фонового потока?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Запустите программу с исключениями и сопоставьте каждую строку вывода с таблицей типов исключений.
2. ★ Преднамеренно сделайте опечатку в имени переменной и в номере вывода; прочитайте traceback и объясните вслух, что в нём написано.
3. ★★ Добавьте в журнал уровень `CRITICAL` и режим «только ERROR и выше»; проверьте ротацию, уменьшив `max_bytes` до 300 байт.
4. ★★ Напишите декоратор `@logged`, который записывает в журнал имя функции и текст исключения, если функция упала, а затем пробрасывает исключение дальше (`raise`).
5. ★★★ Напишите три теста для собственной чистой функции (например, перевода показаний датчика освещённости в проценты), включая граничные значения; один тест должен проваливаться, пока вы не исправите ошибку.
6. ★★★★ Соберите «живучее» устройство: датчик по I^2C, журнал в файле, сторожевой таймер и режим спасения. Отключите датчик на ходу и убедитесь, что программа не падает, пишет предупреждения и восстанавливает работу, когда датчик возвращён.

Урок 24. RP2350, Pico 2 и Pico 2 W

Цель	понять, чем чип RP2350 отличается от RP2040, как выбрать плату семейства Pico, как перенести проект на Pico 2 и что нужно знать об эрратуме E9
Опирается на	уроки 1–23: прошивки, GPIO, ШИМ, АЦП, PIO, отладка (урок 14, урок 23)
Результат	скрипт определяет плату и прошивку; бенчмарк сравнивает скорость на Pico и Pico 2; тест кнопки проверяет подтяжку входа; в руках таблица выбора платы
Понадобится	Pico 2 или Pico 2 W (желательно и обычная Pico для сравнения), макетная плата, кнопка, провода

24.1 Идея урока коротко

В 2024 году появилось второе поколение микроконтроллеров Raspberry Pi: **RP2350**. Он стоит на платах **Pico 2** и **Pico 2 W**. Платы выглядят почти так же, как Pico и Pico W, у них те же 40 выводов, но внутри — более быстрые ядра, вдвое больше оперативной памяти, третий блок PIO, поддержка безопасной загрузки. Почти вся программа с Pico переносится без изменений: нужно лишь записать другую прошивку (файл `.uf2` для RP2350 и *только* для него). Остальное — детали, которые стоит знать, чтобы не споткнуться.

Осторожно

Числа и названия в этом уроке соответствуют документации на момент написания курса. Семейство RP2350 живое: выходят новые степпинги (версии кристалла), прошивки и платы сторонних производителей. Если в уроке сказано «проверьте актуальную информацию», сделайте это в даташите RP2350 и на страницах `raspberrypi.com`, `micropython.org`, `circuitpython.org`.

24.2 Что нового: RP2040 и RP2350 рядом

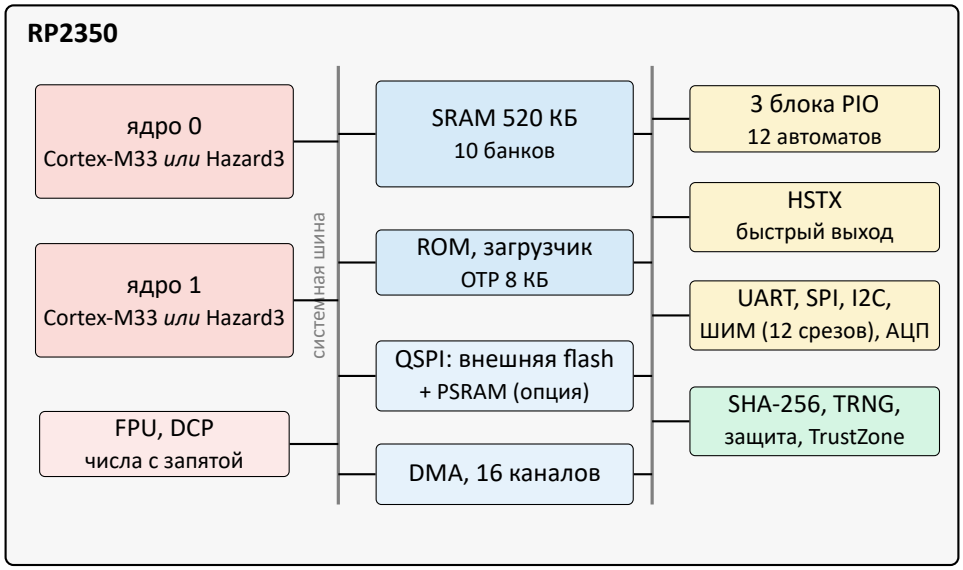


Рис. 24.1: Упрощённая структура RP2350: ядра, память и периферия на общей шине

Параметр	RP2040 (Pico, Pico W)	RP2350 (Pico 2, Pico 2 W)
Ядра	2× Cortex-M0+, 133 MHz	2× Cortex-M33 *или* 2× Hazard3 RISC-V, 150 MHz
SRAM в кристалле	264 kB	520 kB
Flash на плате Pico	2 MB	4 MB (контроллер умеет больше, плюс PSRAM у сторонних плат)
Числа с плавающей запятой	программно	аппаратный FPU одинарной точности (ядра Arm), сопроцессор для двойной
Блоки PIO	2 (8 автоматов)	3 (12 автоматов)
Срезы ШИМ	8	12
Каналы DMA	12	16
Быстрый видеовыход	нет	HSTX
Безопасность	нет	TrustZone, безопасная загрузка, OTP, SHA-256, TRNG
Выводы GPIO	30	30 (корпус A) или 48 (корпус B)
Диск BOOTSEL	`RPI-RP2`	`RP2350`

Таблица 24.1: Сравнение RP2040 и RP2350 по основным параметрам

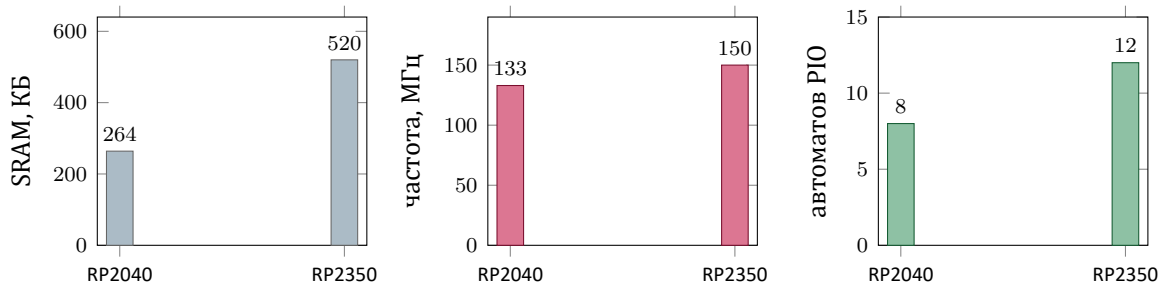


Рис. 24.2: Три числа, которые меняются сильнее всего: память, частота по умолчанию, автоматы PIO

Рост на 13 % по частоте выглядит скромно, но реальное ускорение больше: ядро Cortex-M33 выполняет больше работы за такт, чем M0+, а числа с запятой считает аппаратно. Насколько именно — вы измерите сами в бенчмарке ниже.

24.3 Два типа ядер: Arm и RISC-V

В RP2350 по два ядра *каждого* типа, но одновременно работают два ядра *одного* типа: либо пара Cortex-M33 (архитектура Arm), либо пара Hazard3 (открытая архитектура RISC-V). Какая именно пара включится, решает загрузчик в ПЗУ чипа при старте: он смотрит на заголовок программы (образ `.uf2`) и настройки в OTP. По умолчанию, если ничего не настраивали, запускается Arm.

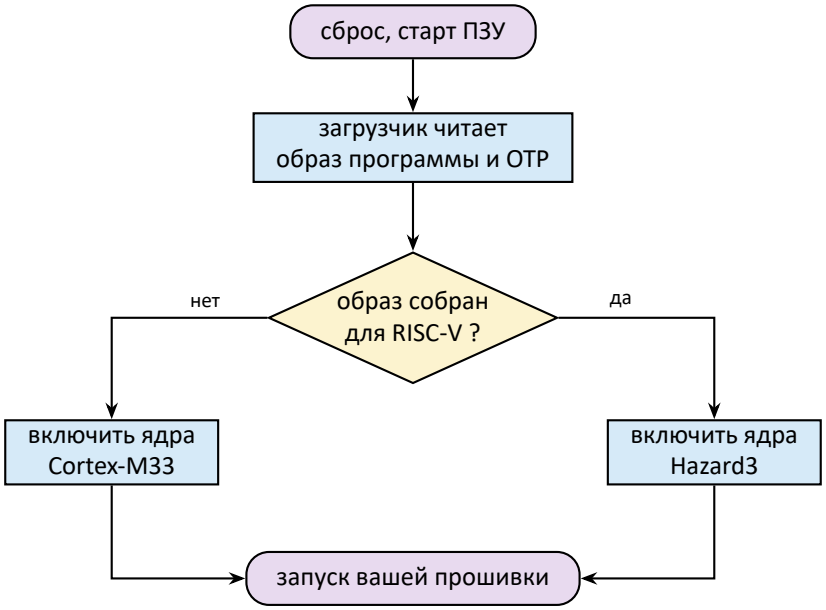


Рис. 24.3: Выбор архитектуры при загрузке RP2350 (схема упрощена)

Это интересно

Для Arm и RISC-V нужны *разные* машинные программы: образ `.uf2` для Arm-ядер не запустится на ядрах Hazard3 и наоборот. Поэтому у такого файла есть метка семейства, и загрузчик не даст записать чужой образ. Готовые прошивки **MicroPython** и **CircuitPython** для **Pico 2 и Pico 2 W собраны под Arm**. Сборки под RISC-V для **MicroPython** существуют в экспериментальном виде в исходных текстах; готового образа на странице загрузок может не быть — проверьте актуальную информацию. Для **CircuitPython** в этом курсе рассматриваем только Arm.

24.4 Новые возможности чипа

Три блока PIO и HSTX

Блок PIO мы изучали в уроке 14: маленькие автоматы, которые двигают биты независимо от процессора. В RP2350 блоков три, автоматов — двенадцать. Программы PIO для RP2040 в основном работают без изменений, а у ассемблера появились новые возможности. Следующий пример запускает программу на третьем блоке; на RP2040 такой код выдаст ошибку, потому что там автоматы нумеруются только от 0 до 7.

Листинг 24.1: MicroPython: мигание на третьем блоке PIO (RP2350)

```
import rp2
import time
from machine import Pin


@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink():
    wrap_target()
    set(pins, 1) [31]       # включить; 1 + 31 тактов
    nop() [31]
    nop() [31]
    nop() [31]
    set(pins, 0) [31]       # выключить
    nop() [31]
    nop() [31]
    nop() [31]
    wrap()


# На RP2040 автомат с номером 8 не существует (там 0..7),
# на RP2350 это первый автомат третьего блока PIO (PIO2)
sm = rp2.StateMachine(8, blink, freq=500, set_base=Pin(15))
sm.active(1)
print("PIO2 запущен, светодиод на GP15 мигает без участия процессора")

while True:
    time.sleep(1)           # процессор свободен
```

Листинг 24.2: CircuitPython: то же; блок и автомат выбираются автоматически

```
import time
import board
import rp2pio
import adafruit_pioasm

# Программа на ассемблере PIO (то же самое, что в MicroPython)
program = adafruit_pioasm.assemble("""
.program blink
.wrap_target
    set pins, 1 [31]
    nop [31]
    nop [31]
    nop [31]
    set pins, 0 [31]
    nop [31]
    nop [31]
```

```
    nop [31]
.wrap
""")

# CircuitPython сам выбирает свободный блок PIO и автомат
sm = rp2pio.StateMachine(program, frequency=500, first_set_pin=board.GP15)
print("PIO запущен, светодиод на GP15 мигает без участия процессора")

while True:
    time.sleep(1)           # процессор свободен
```

В **CircuitPython** нужна библиотека `adafruit_pioasm` (папка `lib`, устанавливается через `circup`). Номера блока и автомата вы не задаёте — об этом заботится модуль `rp2pio`.
HSTX — высокоскоростной передатчик: восемь выводов, на которых данные уходят очень быстро, например для вывода картинки на монитор по DVI. Это возможность для продвинутых проектов; обычный студенческий курс её не использует, а поддержка в **MicroPython** и **CircuitPython** на момент написания ограничена, поэтому ограничимся знанием, что она есть.

Безопасность: TrustZone, безопасная загрузка, OTP

Если устройство можно купить и унести домой, злоумышленник может попытаться подменить его программу. RP2350 предлагает несколько уровней защиты (подробности — в даташите):

- **безопасная загрузка**: загрузчик проверяет цифровую подпись программы, и неподписанная не запустится;
- **OTP** (One-Time Programmable) — память на 8 kB, в которой каждый бит можно перевести из 0 в 1 *один раз и навсегда*. Здесь хранят ключи и настройки;
- **TrustZone** (на ядрах Arm) — процессор делит мир на «безопасный» и «обычный», и обычный код не может читать секреты безопасного;
- аппаратные **SHA-256** и генератор случайных чисел **TRNG**.

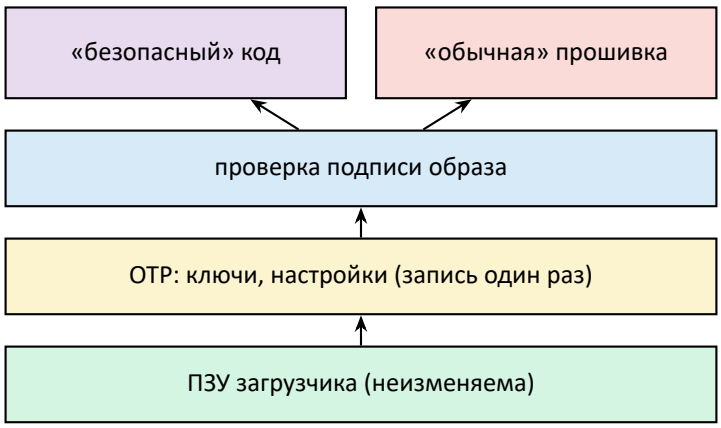


Рис. 24.4: Идея цепочки доверия: каждое звено проверяет следующее

Осторожно

Не включайте безопасную загрузку и не записывайте OTP ради эксперимента: бит OTP не стереть. Неверная настройка может навсегда превратить плату в «кирпич». В учебных проектах эти возможности нужны редко.

24.5 Корпуса RP2350A и RP2350B

Сам кристалл выпускают в двух корпусах. **RP2350A** — 60 выводов, из них 30 GPIO (GP0–GP29), и именно он стоит на Pico 2. **RP2350B** — 80 выводов и 48 GPIO (GP0–GP47). Платы с RP2350B — сторонние (у них вывода больше, чем помещается на разъёме Pico); их распиновка зависит от производителя, и номера выводов нужно брать из документации к плате.

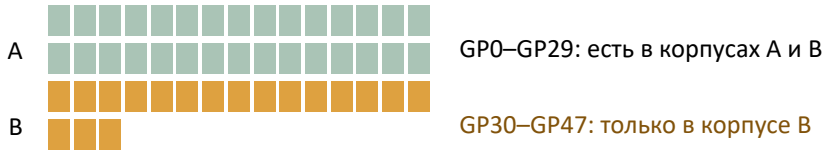


Рис. 24.5: Число доступных выводов GPIO: 30 у RP2350A, 48 у RP2350B

24.6 Прошивки для Pico 2 и Pico 2 W

MicroPython. На странице `micropython.org/download` выберите плату `RPI_PICO2` или `RPI_PICO2_W`. Поддержка RP2350 появилась в версиях **MicroPython** около 1.24 и выше; точный список версий проверьте на странице загрузок. **CircuitPython.** На `circuitpython.org/downloads` найдите «Raspberry Pi Pico 2» и «Pico 2 W»; поддержка есть в ветке 9.2 и новее. Установка такая же, как в уроке 1: удерживая BOOTSEL, подключите плату к USB. Появится диск, но теперь он называется `RP2350` (а не `RPI-RP2`); скопируйте на него файл `.uf2`.

Осторожно

Нельзя записывать на Pico 2 прошивку, собранную для Pico (и наоборот). Загрузчик отвергнет файл с чужим семейством, но путаница частая: смотрите имя файла (`RPI_PICO2...` или `adafruit_pico2...`), а не только слова «Pico» в названии.

Проверим, что запущено. Скрипт одинаково работает на всех платах семейства:

Листинг 24.3: MicroPython: какая плата и прошивка

```
import sys
import os
import gc
import machine

# Какая это плата и какая прошивка?
print("Прошивка :", sys.implementation.name, sys.implementation.version)
print("Плата    :", sys.implementation._machine)   # напр. "Raspberry Pi Pico2 with RP2350"
print("Версия   :", os.uname().version)
print("Частота  : %d МГц" % (machine.freq() // 1_000_000))

# Свободная память кучи MicroPython (это не вся SRAM чипа!)
gc.collect()
print("Куча     : свободно %d байт, занято %d байт" % (gc.mem_free(), gc.mem_alloc()))

# Уникальный номер кристалла
uid = machine.unique_id()
print("ID чипа  :", "".join("%02x" % b for b in uid))
```

Листинг 24.4: CircuitPython: то же

```
import os
import sys
import gc
import board
import microcontroller

# Какая это плата и какая прошивка?
print("Прошивка :", sys.implementation.name, sys.implementation.version)
print("Плата    :", board.board_id)              # напр. "raspberry_pi_pico2"
print("Версия   :", os.uname().version)
print("Чип      :", os.uname().machine)
print("Частота  : %d МГц" % (microcontroller.cpu.frequency // 1_000_000))

# Свободная память кучи CircuitPython (это не вся SRAM чипа!)
gc.collect()
print("Куча     : свободно %d байт, занято %d байт" % (gc.mem_free(), gc.mem_alloc()))

# Уникальный номер кристалла
print("ID чипа  :", "".join("%02x" % b for b in microcontroller.cpu.uid))
```

На Pico 2 вы увидите `150 МГц`, в строке платы — слово RP2350; `gc.mem_free()` покажет больше, чем на Pico, но не 520 КБ: куча — это лишь часть SRAM, остальное занимают стек, буферы USB и системные данные.

24.7 Миграция проекта с Pico на Pico 2

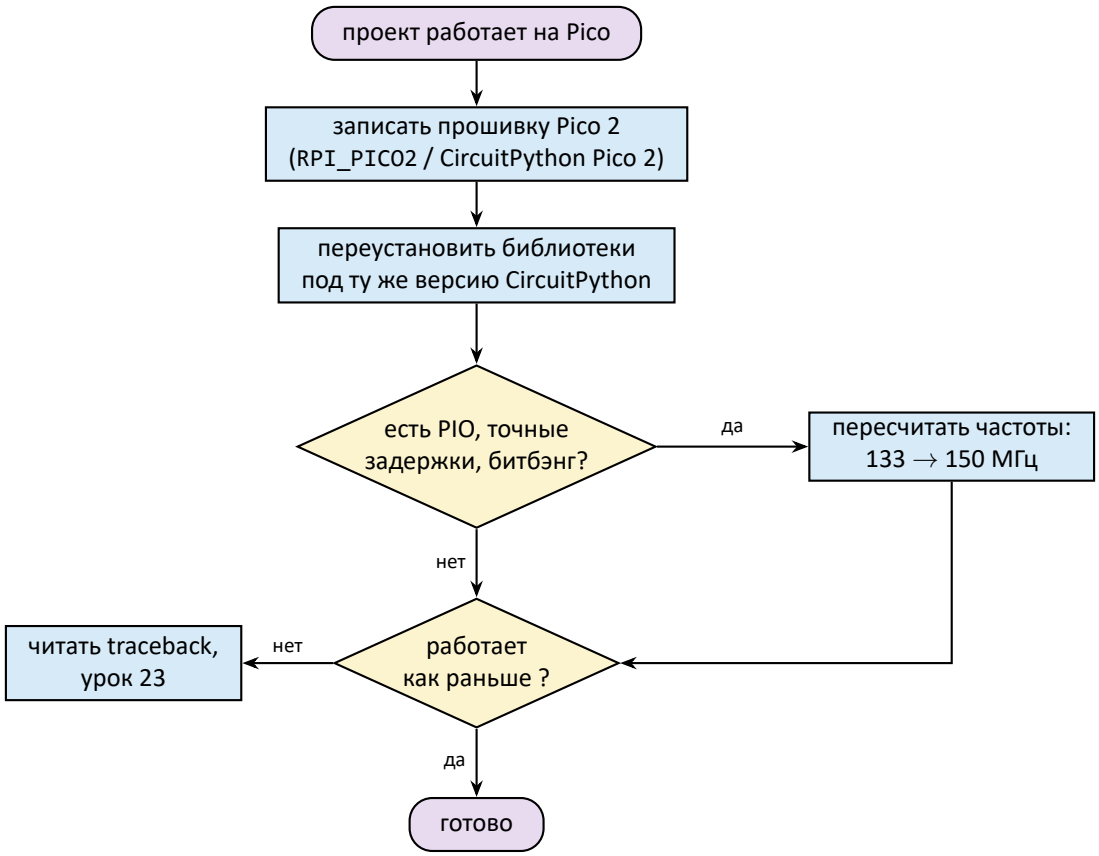


Рис. 24.6: Порядок переноса проекта на Pico 2

Что работает как есть	Что проверить или поменять
GPIO, ШИМ, АЦП, I2C, SPI, UART: те же номера GP и те же классы	Файл прошивки (`.uf2`) и версию библиотек **CircuitPython** под неё
Выводы на плате: разъём Pico 2 совместим по расположению с Pico	Встроенный светодиод: используйте `Pin("LED")` / `board.LED`, не `Pin(25)` (урок 2)
Wi-Fi на Pico 2 W (чип CYW43439, как на Pico W)	Тайминги без PIO: частота 150 вместо 133 MHz; программы со счётом циклов ведут себя иначе
Большинство программ PIO	Делители и частоты машин PIO пересчитайте (и номера автоматов теперь 0–11)
Python-код, модули `time`, `machine`, `asyncio`, `json`	Код, который предполагал малую память (264 kB); теперь ограничений меньше

Таблица 24.2: Что переносится без изменений, а что стоит проверить

24.8 Практика: бенчмарк на Pico и Pico 2

Сравним скорость честно. Правила хорошего замера: измерять время несколькими запусками и брать *лучшее*, перед замером вызывать сборщик мусора, замерять отдельно целые числа и числа с запятой.

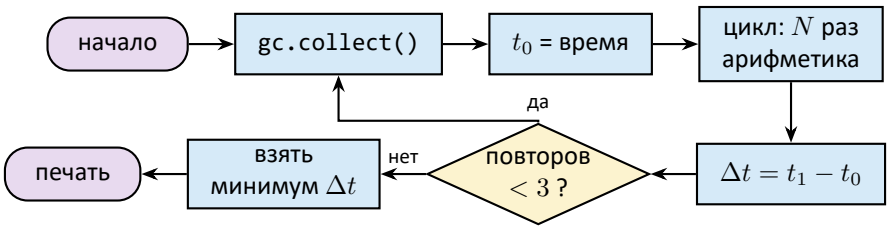


Рис. 24.7: Алгоритм бенчмарка: повторяем замер и берём лучший результат

Листинг 24.5: **MicroPython**: бенчмарк целых и дробных чисел

```
import time
import gc

N = 200_000          # число итераций цикла


def bench_int(n):
    """Целочисленная арифметика: сложение, умножение, сдвиг, XOR."""
    s = 0
    for i in range(n):
        s = ((s + i * 3) ^ (i >> 1)) & 0xFFFF
    return s


def bench_float(n):
    """Арифметика с плавающей запятой (в MicroPython float по умолчанию 32-битный)."""
    x = 0.0
    for i in range(n):
        x = x * 0.999 + i * 0.001
    return x


def measure(func, n, repeats=3):
```

```
    """Лучшее время из нескольких запусков, в миллисекундах."""
    best = None
    for _ in range(repeats):
        gc.collect()                            # убираем влияние сборщика мусора
        t0 = time.ticks_us()
        func(n)
        dt = time.ticks_diff(time.ticks_us(), t0)
        if best is None or dt < best:
            best = dt
    return best / 1000


t_int = measure(bench_int, N)
t_flt = measure(bench_float, N)
print("int   :", t_int, "мс")
print("float :", t_flt, "мс")
print("float / int = %.2f" % (t_flt / t_int))
```

Листинг 24.6: CircuitPython: тот же бенчмарк

```
import time
import gc

N = 200_000          # число итераций цикла


def bench_int(n):
    """Целочисленная арифметика: сложение, умножение, сдвиг, XOR."""
    s = 0
    for i in range(n):
        s = ((s + i * 3) ^ (i >> 1)) & 0xFFFF
    return s


def bench_float(n):
    """Арифметика с плавающей запятой (в CircuitPython float 32-битный)."""
    x = 0.0
    for i in range(n):
        x = x * 0.999 + i * 0.001
    return x


def measure(func, n, repeats=3):
    """Лучшее время из нескольких запусков, в миллисекундах."""
    best = None
    for _ in range(repeats):
        gc.collect()
        t0 = time.monotonic_ns()                # наносекундный счётчик
        func(n)
        dt = time.monotonic_ns() - t0
        if best is None or dt < best:
            best = dt
    return best / 1_000_000


t_int = measure(bench_int, N)
t_flt = measure(bench_float, N)
print("int   :", t_int, "мс")
```

```
print("float :", t_flt, "мс")
print("float / int = %.2f" % (t_flt / t_int))
```

Различия в коде небольшие: **MicroPython** считает время через `time.ticks_us()` и `ticks_diff()` (урок 4), **CircuitPython** — через `time.monotonic_ns()`. Запустите на Pico и на Pico 2 и впишите результаты в таблицу:

Плата, язык	int, мс	float, мс	float/int
Pico, **MicroPython**			
Pico 2, **MicroPython**			
Pico, **CircuitPython**			
Pico 2, **CircuitPython**			

Таблица 24.3: Бланк для результатов бенчмарка (заполните сами)

Совет

Чего ожидать: на Pico 2 заметнее всего выигрывает счёт с плавающей запятой (аппаратный FPU), но интерпретатор Python остаётся главным узким местом, поэтому разница редко бывает «в десять раз». Если нужна максимальная скорость, горячий участок переносят в PIO, в нативный модуль или пишут на C/C++.

24.9 Эрратум E9: подтяжка входа GPIO

У каждой микросхемы бывают ошибки, найденные уже после выпуска. Список таких ошибок называется **эрратой**, каждая получает номер. В первых версиях RP2350 (степпинг **A2**) нашли ошибку **E9**, связанную со входом GPIO: при определённых условиях — например, когда на входе включена внутренняя подтяжка к земле, а входной буфер открыт — вывод не ведёт себя как простой резистор: после того как внешний сигнал высокого уровня «отпущен» (кнопка разомкнута), напряжение на входе может не упасть к нулю, а «залипнуть» на промежуточном уровне, который микроконтроллер читает как 1. Из-за этого возможна утечка тока и ложные срабатывания.

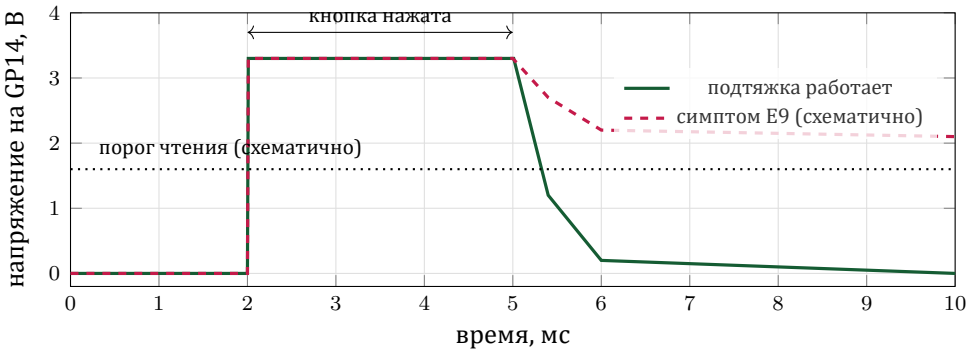
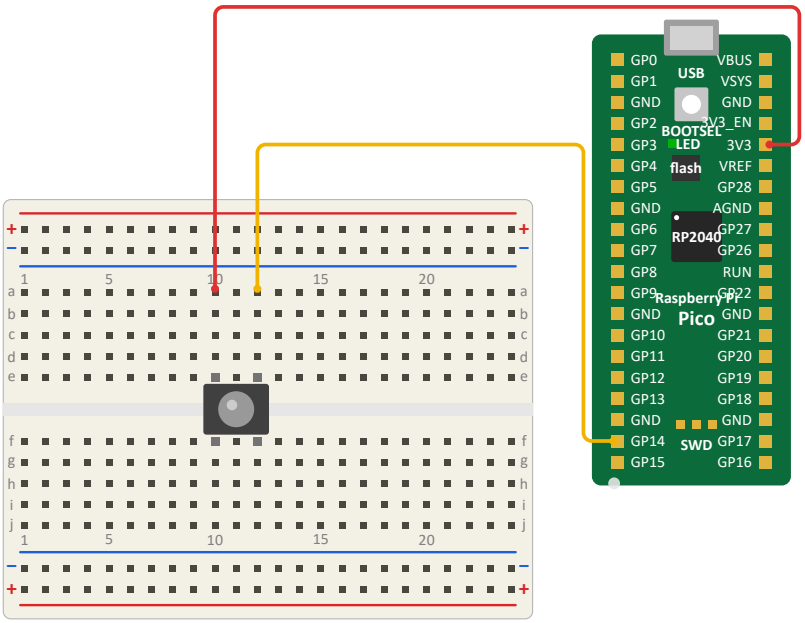


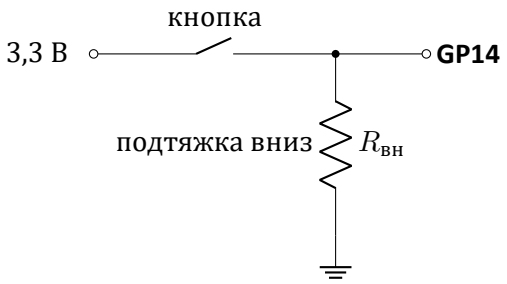
Рис. 24.8: Что должно происходить после отпускания кнопки и что описывает E9. Кривые схематичные, реальные напряжения смотрите в даташите

Что известно из открытых источников: ошибка была исправлена в новых степпингах, по сообщениям производителя и прессы — в **A4**; на старых платах и в старых партиях она может встречаться. Для обхода на платах с A2 рекомендовали внешний резистор подтяжки сравнительно небольшого номинала (порядка единиц килоом; точное значение и условия смотрите в разделе E9 даташита RP2350) и *не* полагаться на внутренние подтяжки в схемах, где вывод оставляют «висеть». **Проверьте актуальную информацию**: как называется степпинг вашей платы и что написано в последней версии даташита.

Проверить свою плату можно простой схемой: кнопка соединяет GP14 с $3{,}3$ V, а внутренняя подтяжка к земле должна возвращать 0.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 24.9: Проверка подтяжки входа: кнопка между 3V3 и GP14

Красный провод — от вывода 3V3 (на плате он правее, в правом ряду выводов) вокруг платы сверху к левому столбцу кнопки; жёлтый — от GP14 к правому столбцу кнопки. Нажатие кнопки соединяет столбцы, и **GP14** получает $3{,}3$ V. Внешнего резистора нет: проверяется именно внутренняя подтяжка.

Листинг 24.7: MicroPython: проверка возврата входа в 0 после отпускания кнопки

```
from machine import Pin
import time

# Кнопка соединяет GP14 с 3V3; внутренняя подтяжка к земле должна возвращать 0
pin = Pin(14, Pin.IN, Pin.PULL_DOWN)

last = pin.value()
print("старт, значение =", last)

while True:
    v = pin.value()
    if v != last:
        last = v
        print("изменилось на", v)
```

```
        if v == 0:
            continue
        # кнопку нажали; ждём, пока отпустят
        while pin.value() == 1:
            time.sleep_ms(10)
        # кнопка отпущена: подтяжка вниз должна быстро дать 0
        time.sleep_ms(50)
        if pin.value() == 1:
            print("ВНИМАНИЕ: после отпускания на входе осталась 1 (признак E9?)")
    time.sleep_ms(5)
```

Листинг 24.8: CircuitPython: то же

```
import time
import board
import digitalio

# Кнопка соединяет GP14 с 3V3; внутренняя подтяжка к земле должна возвращать 0
pin = digitalio.DigitalInOut(board.GP14)
pin.direction = digitalio.Direction.INPUT
pin.pull = digitalio.Pull.DOWN

last = pin.value
print("старт, значение =", last)

while True:
    v = pin.value
    if v != last:
        last = v
        print("изменилось на", v)
        if v:
            # кнопку нажали; ждём, пока отпустят
            while pin.value:
                time.sleep(0.01)
            # кнопка отпущена: подтяжка вниз должна быстро дать 0
            time.sleep(0.05)
            if pin.value:
                print("ВНИМАНИЕ: после отпускания на входе осталась 1 (признак E9?)")
    time.sleep(0.005)
```

Если на исправных платах сообщение «ВНИМАНИЕ» никогда не появляется, всё хорошо. Если оно появляется — не обязательно виноват чип: проверьте контакты кнопки и провода, повторите с другой кнопкой. Но если вход «залипает» без кнопки и без помех, вспомните про E9.

Осторожно

Не подавайте на выводы Pico 2 напряжение выше 3,3 V: допустимые 5 В по-прежнему только у вывода VBUS (питание USB). Это ограничение у RP2350 такое же, как у RP2040.

24.10 Сравнение плат семейства

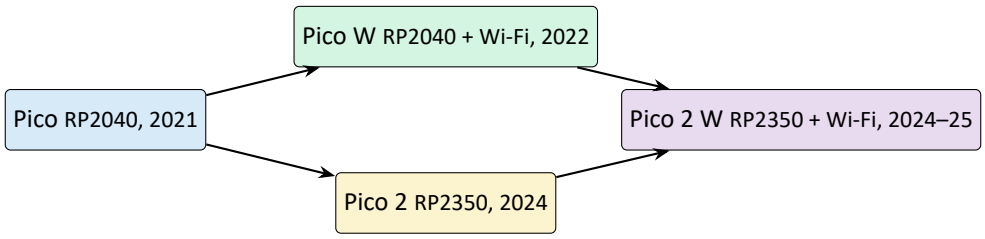


Рис. 24.10: Семейство плат Pico и пути развития (годы округлены, уточните на сайте производителя)

	Pico	Pico W	Pico 2	Pico 2 W
Чип	RP2040	RP2040	RP2350A	RP2350A
Flash	2 MB	2 MB	4 MB	4 MB
Wi-Fi, BLE	нет	да (CYW43439)	нет	да (CYW43439)
Встроенный светодиод	GP25	через радиочип	GP25	через радиочип
Прошивка MicroPython	`RPI_PICO`	`RPI_PICO_W`	`RPI_PICO2`	`RPI_PICO2_W`
Диск BOOTSEL	`RPI-RP2`	`RPI-RP2`	`RP2350`	`RP2350`

Таблица 24.4: Четыре платы Raspberry Pi: что у них общего и чем они различаются

Сторонних плат на RP2350 много; цифры у них свои, поэтому сверяйтесь с документацией производителя. Для примера: **Pimoroni Pico Plus 2** имеет заметно больше flash и PSRAM, чем Pico 2 (по данным производителя — 16 МБ flash и 8 МБ PSRAM); платы **Adafruit Feather RP2350** и **Adafruit Metro RP2350** выполнены в других форм-факторах и тоже несут PSRAM, а Metro и некоторые другие — чип RP2350B с 48 GPIO. Для таких плат в CircuitPython и MicroPython используют *свою* прошивку (имя платы указано на странице загрузок) и свои названия выводов `board.D5`, а не `board.GP5`.

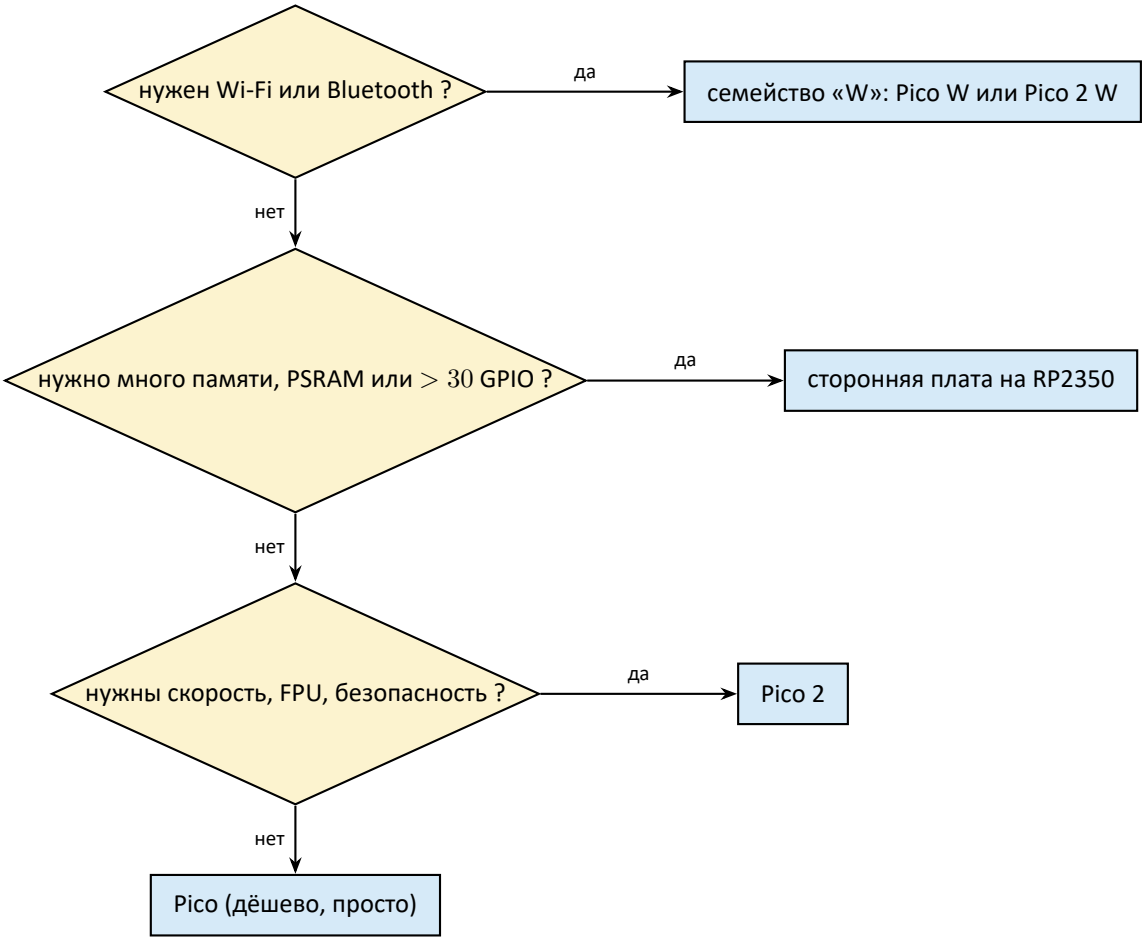


Рис. 24.11: Как выбрать плату (ориентир; решает задача и бюджет)

Задача	Рекомендация
Первые шаги, светодиоды, датчики, кнопки	Pico или Pico 2 — любая; Pico 2 даёт запас
Сайт, MQTT, Wi-Fi-датчик	Pico W или Pico 2 W
Звук, WAV, большие массивы, графика	Pico 2 (больше SRAM); при острой нехватке — плата с PSRAM
Много выводов, свои форм-факторы	сторонняя плата на RP2350B
Проект с защитой прошивки	Pico 2 и изучение безопасной загрузки (по даташиту)
Нужна совместимость со старыми библиотеками	сначала проверьте поддержку RP2350 в библиотеке

Таблица 24.5: Таблица выбора платы

24.11 Учимся программировать: определяем железо в коде

Хорошая программа сама выясняет, на какой плате работает, и подстраивается. Для этого проверяют строку-описание платы: `sys.implementation._machine` в **MicroPython** или `os.uname().machine` / `board.board_id` в **CircuitPython**. Так появляются приёмы: *условие по возможностям* («есть ли такое имя?») лучше, чем по названию платы. Например, в **MicroPython** можно написать `if "RP2350" in sys.implementation._machine:` и выбрать номер автомата PIO. Новая идея: **не прибивать константы гвоздями**. Частоту, номера выводов и размеры буферов выносите в переменные в начале файла — так программа переносится между платами

правкой одного места.

Если не работает

- **Файл `.uf2` не принимается или плата «молчит»** — прошивка не для этой платы. Сверьте имя (`RPI_PICO2` и не `RPI_PICO`).
- **Диск `RP2350` не появляется** — нужно держать BOOTSEL *до* подключения кабеля; кабель должен передавать данные.
- **Старая библиотека не импортируется в CircuitPython** — версия `.mpy` должна соответствовать версии CircuitPython (урок 1).
- **Код PIO с номером автомата 8 даёт ошибку** — вы запустили его на RP2040 (там автоматы 0–7).
- **Вход «залипает» в 1** — проверьте проводку; затем вспомните про E9 и степпинг платы; временно поставьте внешнюю подтяжку.
- **На Pico 2 W `Pin(25)` не мигает** — светодиод на радиочипе, используйте `Pin("LED")`.

Итоги урока

- RP2350: ядра Cortex-M33 (или Hazard3), 150 MHz, 520 kB SRAM, FPU, три блока PIO, HSTX, функции безопасности.
- Для Pico 2 / Pico 2 W нужны свои прошивки (`RPI_PICO2`, `RPI_PICO2_W`); диск BOOTSEL называется `RP2350`.
- Большая часть кода с Pico переносится без изменений; проверяйте встроенный светодиод, тайминги и PIO.
- Эрратум E9 касается входов GPIO в ранних степпингах; актуальные сведения — в даташите.
- Плату выбирают по задаче: Wi-Fi, память, число выводов, скорость.

Проверь себя

1. Назовите три отличия RP2350 от RP2040 и объясните, чем они полезны.
2. Что решает загрузчик в ПЗУ RP2350 и почему образы для Arm и RISC-V нельзя перепутать?
3. Что нужно изменить при переносе проекта с Pico на Pico 2 и что можно не менять?
4. Почему результат бенчмарка берут как минимум из нескольких запусков?
5. Что такое эрратум E9 и как проверить, затрагивает ли он вашу плату?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Запустите скрипт `124_info` на своей плате и запишите: прошивку, плату, частоту, свободную память.
2. ★ Заполните таблицу бенчмарка на одной плате в обоих языках. Какой язык быстрее в ваших измерениях?
3. ★★ Измените бенчмарк: добавьте тест со строками (склейка 1000 коротких строк) и

тест списков. Что медленнее всего?

4. ★★ Соберите схему кнопки из рис. 24.9, запустите проверку и опишите результат. Повторите с внешним резистором 10 kΩ к земле.
5. ★★★ Напишите универсальную программу, которая определяет плату и выбирает номер вывода светодиода и номер автомата PIO, а затем мигает светодиодом.
6. ★★★★ Составьте таблицу для вашей будущей работы: проект, выбранная плата, три причины выбора, что изменится при замене платы на другую из семейства.

Урок 25. Итоговый проект: метеостанция с экраном, Wi-Fi и MQTT

Цель	собрать законченное устройство, в котором работают вместе датчик, экран, кнопка, светодиод, Wi-Fi, MQTT, веб-страница, журнал и сторожевой таймер, и защитить проект как инженерную работу
Опирается на	уроки 3–15: кнопка, таймеры и автоматы, I2C, датчики, файлы, многозадачность, сторожевой таймер; уроки 16–18: Wi-Fi, веб-сервер, MQTT
Результат	метеостанция показывает температуру, влажность и давление на OLED, листает режимы кнопкой, отдаёт данные по Wi-Fi (страница, JSON, MQTT), ведёт журнал и сама восстанавливается после сбоев
Понадобится	Pico W или Pico 2 W, модуль BME280, OLED SSD1306, кнопка, светодиод, резистор 330 Ω, макетная плата, провода; компьютер или Raspberry Pi с брокером Mosquitto

25.1 Постановка задачи

До сих пор каждый урок давал один «кубик»: светодиод, кнопку, датчик, сеть. Настоящее устройство собирают из многих кубиков сразу, и тут возникают новые трудности: кубики мешают друг другу (один ждёт, другие простаивают), сеть пропадает, датчик иногда «молчит», память заканчивается. Итоговый проект учит справляться с этим. Мы делаем **метеостанцию**, которая:

- измеряет температуру, влажность и давление датчиком BME280 раз в две секунды;
- показывает данные на OLED-экране и листает четыре экрана по короткому нажатию кнопки;
- по долгому нажатию сбрасывает минимум и максимум температуры;
- показывает светодиодом, всё ли в порядке (по рисунку мигания);
- подключается к Wi-Fi, получает время по NTP, публикует данные в MQTT каждые 30 с;
- отвечает на запросы браузера: страница с текущими значениями и JSON;
- ведёт журнал событий во flash-памяти, не разрушая её частыми записями;
- не зависает: сторожевой таймер перезагружает плату, если программа замерла.

№	Требование	Как проверить
T1	Данные на экране обновляются не реже 4 раз в секунду; значения совпадают с эталонным термометром ± 1 °C	Сравнить с термометром, подышать на датчик
T2	Короткое нажатие меняет экран по кругу из четырёх режимов; дребезг не даёт ложных переключений	Нажать 20 раз подряд
T3	Долгое нажатие ($\geq$ 1,2 с) сбрасывает минимум и максимум	Нагреть датчик, сбросить, проверить
T4	Без Wi-Fi и без датчика устройство не «падает» и показывает причину	Выключить роутер, вынуть провод SDA
T5	После возврата Wi-Fi соединение восстанавливается само, без перезагрузки платы	Включить роутер обратно
T6	MQTT-сообщение приходит каждые 30 с в формате JSON	`mosquitto_sub`
T7	Веб-страница и `/api` открываются с телефона в той же сети	Открыть адрес с экрана «Сеть»
T8	Журнал не превышает 8 КБ и не теряется при сбросе питания	Выдернуть USB, прочитать файл
T9	Зависание программы приводит к перезагрузке за 8 с	Искусственная «петля» без feed
T10	Пароли не лежат в репозитории и не выводятся на экран	Проверить `.gitignore`

Таблица 25.1: Требования к проекту и способы проверки

Это интересно

Строка таблицы «как проверить» — главное отличие требования от пожелания. «Станция должна быть надёжной» — пожелание, его нельзя проверить. «Через 8 секунд после зависания плата перезагружается» — требование: можно поставить опыт и увидеть результат.

25.2 Архитектура

Начинаем не с кода, а с чертежа. Он отвечает на три вопроса: что с чем связано, кто главный и что случится, если один из блоков откажет.

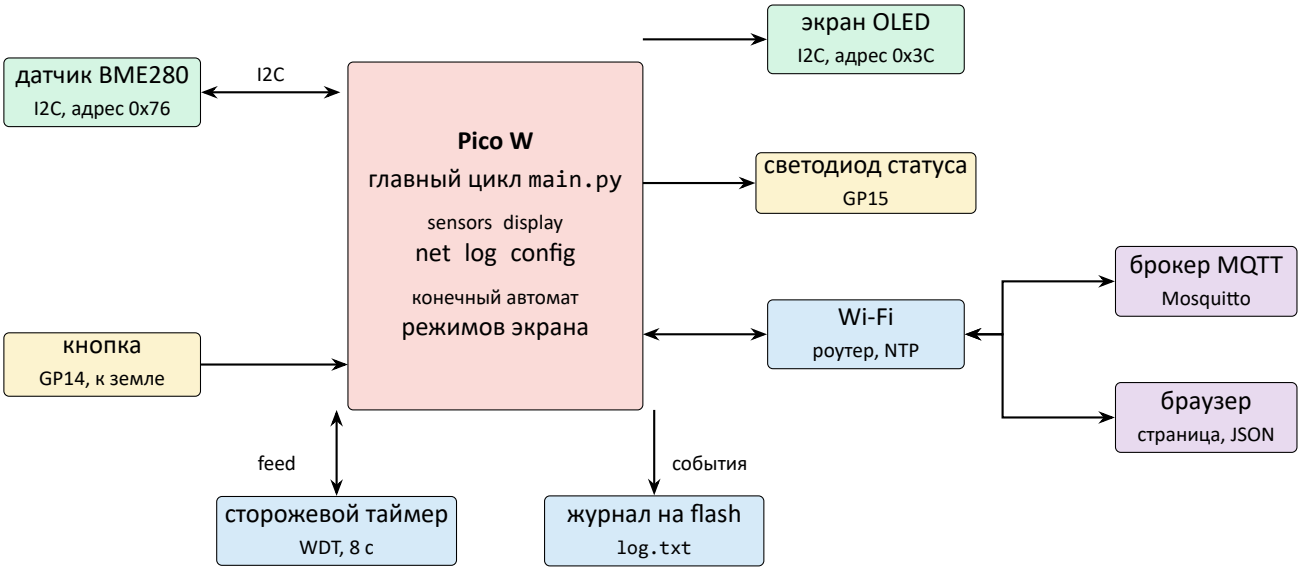


Рис. 25.1: Блок-схема метеостанции: данные идут слева направо, служебные блоки — снизу

Главный цикл — единственный «руководитель»: он по очереди спрашивает кнопку, датчик, сеть, экран и журнал, а каждому блоку отводится очень мало времени. Ни одна часть программы не имеет права ждать (`sleep`, бесконечное ожидание подключения): иначе замрёт всё остальное. Этот приём уроков 4 и 13 теперь работает на полную мощность.

25.3 Детали и бюджет ресурсов

Деталь	Кол.	Урок	Назначение
Raspberry Pi Pico W (или Pico 2 W)	1	16	мозг станции, Wi-Fi
Модуль BME280 (I2C)	1	10	температура, влажность, давление
OLED-экран 0,96" SSD1306, I2C	1	8	вывод данных
Тактовая кнопка 4-выводная	1	3	смена режима, сброс min/max
Светодиод красный или зелёный	1	2	индикация состояния
Резистор 330 Ω (или 220 Ω)	1	2	ограничение тока светодиода
Макетная плата на 400 точек	1	1	сборка без пайки
Провода «папа–папа»	15	—	соединения
Кабель micro-USB (для данных!)	1	1	питание, программирование

Таблица 25.2: Список деталей

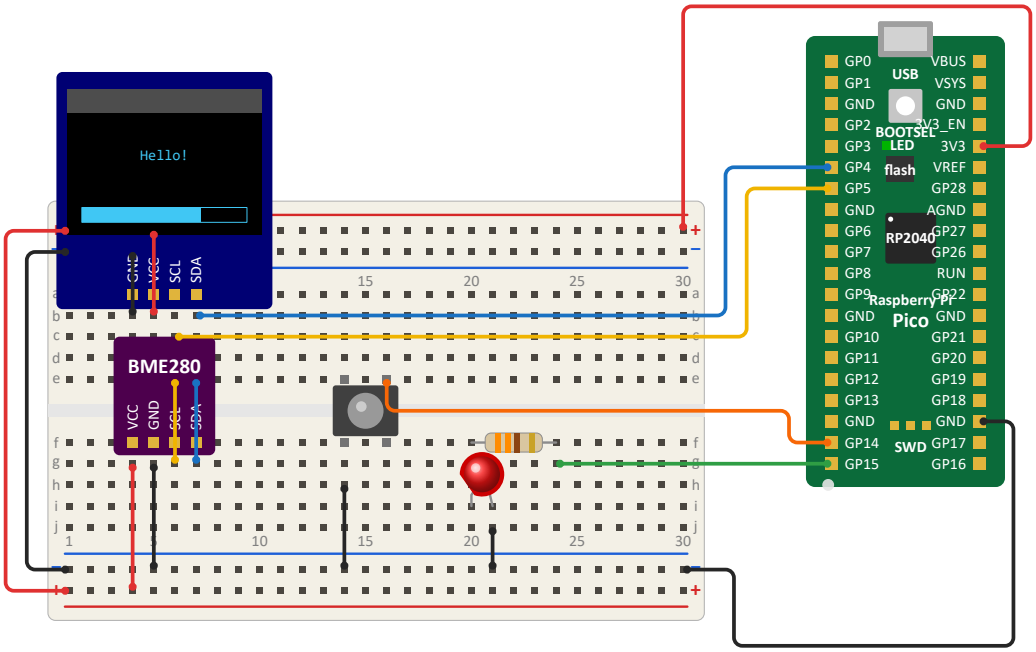
Ресурс	Бюджет проекта
Выводы GPIO	4 из 26 свободных: GP4 (SDA), GP5 (SCL), GP14 (кнопка), GP15 (светодиод)
Адреса I2C	`0x3C` (OLED), `0x76` (BME280); конфликта нет
Ток потребления	порядка 100–150 мА (в основном радиомодуль, пока Wi-Fi работает; экран — около 20 мА, датчик — доли миллиампера)
Оперативная память	из 264 КБ в **MicroPython** программе хватает с большим запасом; следим за `gc.mem_free()` на экране «Система»
Время шага цикла	обычно единицы миллисекунд; дольше всего (десятки миллисекунд) идут публикация в MQTT и обновление экрана
Flash-память	журнал не более 16 КБ (файл и его копия); запись — по событиям, а не каждую секунду
Сторожевой таймер	8 с: если цикл не вызвал `feed()`, плата перезагрузится

Таблица 25.3: Бюджет ресурсов: что мы тратим и сколько у нас есть

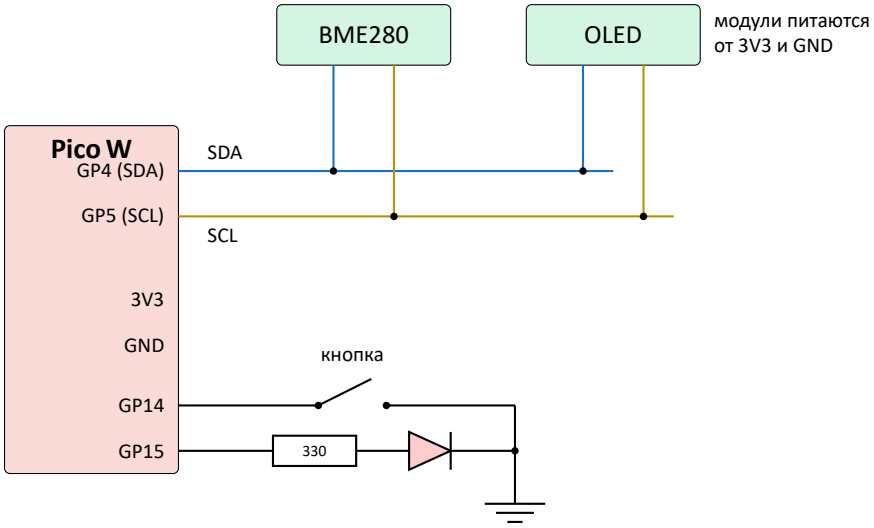
Бюджет — это привычка инженера: до начала работы выписать, чего у нас не хватит в первую очередь. Здесь такой ресурс — время цикла: каждое действие должно быть коротким.

25.4 Монтажная схема

Датчик и экран подключаются к одной шине I2C0 (**GP4** — SDA, **GP5** — SCL): у каждого устройства свой адрес, поэтому они не мешают друг другу. Питание обоим модулям — $3{,}3\,\text{V}$. Кнопка соединяет **GP14** с землёй (подтяжку к питанию включает программа), светодиод подключён к **GP15** через $330\,\Omega$.



а) Монтажная схема: как собрать на макетной плате



б) Принципиальная электрическая схема

Рис. 25.2: Метеостанция: полная схема соединений

Осторожно

Порядок выводов на модулях BME280 и OLED у разных производителей *разный*: у одних идёт «VCC, GND, SCL, SDA», у других «GND, VCC, SCL, SDA». Перед подключением прочитайте надписи на плате модуля и сверьтесь со схемой. Перепутанные питание и земля могут сразу сжечь датчик.

Если не работает

- **Экран тёмный, датчик не виден** — перепроверьте питание модулей и пару SDA/SCL (их часто путают местами). Скрипт-проверка шины: `print([hex(a) for a in i2c.scan()])` должна показать `['0x3c', '0x76']`.
- **Адрес BME280 `0x77`** — у части модулей он другой; измените `BME_ADDR` в `config.py`.

- **Кнопка «живёт своей жизнью»** — одна из ножек кнопки не в той колонке; проверьте, что пара выводов с одной стороны кнопки соединена внутри, а нажатие замыкает *разные* стороны.

25.5 Разбиение на модули

Программа из 400 строк в одном файле превращается в кашу. Поэтому проект разбит на файлы, у каждого из которых *одна* задача. Файл не должен знать лишнего: например, `display.py` ничего не знает про MQTT, а `sensors.py` — про экран.

Файл на плате	Задача	Файл книги
`config.py`	все настройки и номера выводов в одном месте	`l25_config_mp.py`
`secrets.py`	пароль Wi-Fi и данные брокера (не публиковать!)	`l25_secrets_mp.py`
`log.py`	журнал на flash с ограничением размера	`l25_log_mp.py`
`sensors.py`	драйвер BME280, минимум/максимум, счёт ошибок	`l25_sensors_mp.py`
`display.py`	рисование четырёх экранов на OLED	`l25_display_mp.py`
`net.py`	Wi-Fi, NTP, MQTT, веб-сервер	`l25_net_mp.py`
`main.py`	кнопка, автомат режимов, главный цикл, watchdog	`l25_main_mp.py`

Таблица 25.4: Модули проекта. Файлы книги называются иначе, чем на плате, чтобы не путать главы

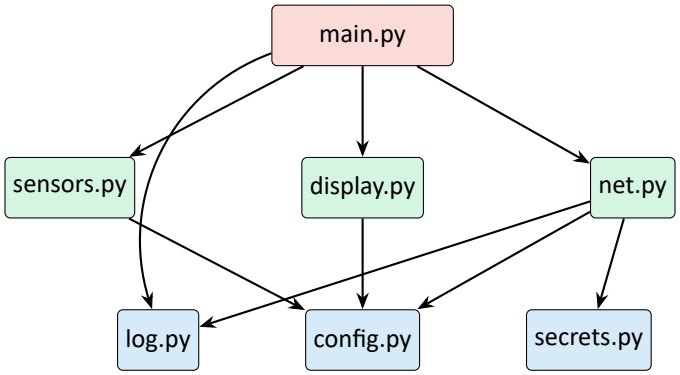


Рис. 25.3: Кто кого импортирует. Стрелки идут только «сверху вниз»: нижние модули ничего не знают о верхних

Правило «стрелки только вниз» защищает от самой частой ошибки новичков — циклических импортов, когда два модуля ждут друг друга. Если вы чувствуете, что `display.py` должен вызвать функцию из `main.py`, значит, эту функцию нужно передать параметром (так сделан веб-сервер: ему передают функции `get_data` и `get_log`).

25.6 Конечный автомат режимов экрана

Экран переключается между четырьмя режимами. Такую логику удобно описывать **конечным автоматом** (урок 4): у программы всегда ровно одно состояние, а события (нажатия) переводят её в следующее.

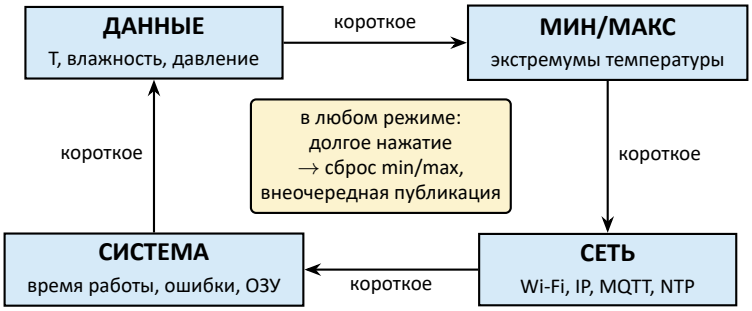


Рис. 25.4: Автомат режимов экрана: четыре состояния, события «короткое» и «долгое нажатие»

В коде состояние хранится одним числом `mode` (0–3), а переход — одной строкой `mode = (mode + 1) % N_MODES`. Это то самое «остаток от деления», которое возвращает режим 3 обратно в 0. Различать короткое и долгое нажатие учит класс `Button`: он запоминает момент нажатия и смотрит, как долго кнопку держали.

25.7 Полный код: MicroPython

Установите библиотеки командами (плата подключена к Wi-Fi, поэтому `mip` качает прямо на неё; иначе установка идёт через компьютер):

```
mpremote mip install ssd1306
mpremote mip install umqtt.simple
mpremote cp config.py secrets.py log.py sensors.py display.py net.py main.py :
```

Файлы на плате должны называться `config.py`, `log.py` и т. д. (в книге они носят имена `l25_…`). Листинги ниже идут «снизу вверх»: сначала то, от чего зависят остальные.

Листинг 25.1: MicroPython: `config.py` — настройки

```
# config.py --- настройки проекта (несекретные)
# Всё, что может понадобиться изменить, лежит здесь, а не в коде.

# --- выводы ---
I2C_ID = 0          # шина I2C0: SDA = GP4, SCL = GP5
PIN_SDA = 4
PIN_SCL = 5
PIN_BTN = 14        # кнопка на GND (внутренняя подтяжка к 3,3 В)
PIN_LED = 15        # светодиод статуса через резистор

# --- адреса устройств I2C ---
BME_ADDR = 0x76     # у некоторых модулей 0x77
OLED_ADDR = 0x3C

# --- периоды ---
SENSOR_PERIOD_MS = 2000
DISPLAY_PERIOD_MS = 250
PUBLISH_PERIOD_S = 30
LOG_PERIOD_S = 600  # раз в 10 минут записываем строку с измерениями
WIFI_RETRY_MS = 15000
LONG_PRESS_MS = 1200

# --- защита ---
USE_WDT = True      # при отладке поставьте False
WDT_TIMEOUT_MS = 8000
MAX_SENSOR_ERRORS = 30  # столько ошибок подряд --- перезагрузка

# --- прочее ---
TEMP_OFFSET = 0.0   # поправка температуры, градусы Цельсия
MQTT_BASE = "pico/weather"
LOG_FILE = "log.txt"
LOG_MAX_BYTES = 8192
WEB_PORT = 80
```

Листинг 25.2: MicroPython: `secrets.py` — пароли (не публиковать)

```
# secrets.py --- пароли и адреса. Этот файл НЕЛЬЗЯ публиковать!
# Добавьте secrets.py в .gitignore, а в репозиторий положите secrets_example.py.

WIFI_SSID = "my-network"
WIFI_PASSWORD = "my-password"

MQTT_BROKER = "192.168.1.10"   # адрес компьютера или Raspberry Pi с Mosquitto
MQTT_PORT = 1883
MQTT_USER = ""                 # пусто, если брокер без пароля
MQTT_PASSWORD = ""
```

Осторожно

Файл `secrets.py` нельзя выкладывать в интернет. Если вы храните проект в Git, добавьте строку `secrets.py` в файл `.gitignore`, а в репозиторий положите шаблон `secrets_example.py` с вымышленными значениями. Пароль, попавший в публичный репозиторий, считайте скомпрометированным и меняйте.

Листинг 25.3: MicroPython: `log.py` — журнал на flash

```
# log.py --- журнал событий на flash-памяти с ограничением размера
import os
import time
import config


def stamp():
    """Метка времени: часы, если время уже получено по NTP, иначе время работы."""
    t = time.localtime()
    if t[0] >= 2024:
        return "%02d:%02d:%02d" % (t[3], t[4], t[5])
    return "up%ds" % (time.ticks_ms() // 1000)


class Log:
    def write(self, msg):
        line = "%s %s\n" % (stamp(), msg)
        print(line, end="")                # дубль в REPL для отладки
        try:
            try:
                size = os.stat(config.LOG_FILE)[6]
            except OSError:
                size = 0
            if size > config.LOG_MAX_BYTES:  # ротация: старый журнал заменяем
                old = config.LOG_FILE + ".old"
                try:
                    os.remove(old)
                except OSError:
                    pass
                os.rename(config.LOG_FILE, old)
            with open(config.LOG_FILE, "a") as f:
                f.write(line)
        except OSError:
            pass                           # журнал не должен ронять программу

    def tail(self, nbytes=1500):
        """Последние байты журнала (для веб-страницы)."""
        try:
            size = os.stat(config.LOG_FILE)[6]
            with open(config.LOG_FILE) as f:
                f.seek(max(0, size - nbytes))
                return f.read()
        except OSError:
            return ""
```

Журнал записывается *по событиям* (запуск, потеря Wi-Fi, ошибка датчика) и раз в десять минут одной строкой с измерениями. Flash-память выдерживает десятки тысяч циклов стирания;

запись раз в секунду исчерпала бы её за считанные месяцы. Когда файл становится больше `LOG_MAX_BYTES`, он переименовывается в `log.txt.old` — получается «окно» из двух файлов, размер которого ограничен.

Листинг 25.4: MicroPython: `sensors.py` — драйвер BME280 и учёт ошибок

```
# sensors.py --- датчик BME280 (температура, влажность, давление)
import struct
import time
import config


def _s8(b):
    return b - 256 if b > 127 else b


class BME280:
    def __init__(self, i2c, addr=0x76):
        self.i2c = i2c
        self.addr = addr
        if i2c.readfrom_mem(addr, 0xD0, 1)[0] != 0x60:   # идентификатор BME280
            raise OSError("это не BME280")
        # калибровочные коэффициенты, записанные на заводе в датчик
        (self.T1, self.T2, self.T3, self.P1, self.P2, self.P3, self.P4,
         self.P5, self.P6, self.P7, self.P8, self.P9) = struct.unpack(
            "<HhhHhhhhhhhh", i2c.readfrom_mem(addr, 0x88, 24))
        self.H1 = i2c.readfrom_mem(addr, 0xA1, 1)[0]
        d = i2c.readfrom_mem(addr, 0xE1, 7)
        self.H2 = struct.unpack("<h", d[0:2])[0]
        self.H3 = d[2]
        self.H4 = (_s8(d[3]) << 4) | (d[4] & 0x0F)
        self.H5 = (_s8(d[5]) << 4) | (d[4] >> 4)
        self.H6 = _s8(d[6])

    def read(self):
        """Один замер в режиме forced: (T в °C, влажность в %, давление в гПа)."""
        w = self.i2c.writeto_mem
        w(self.addr, 0xF2, b"\x01")       # влажность: передискретизация x1
        w(self.addr, 0xF4, b"\x25")       # T x1, P x1, режим forced (один замер)
        time.sleep_ms(15)                 # замер занимает около 10 мс
        d = self.i2c.readfrom_mem(self.addr, 0xF7, 8)
        adc_p = (d[0] << 12) | (d[1] << 4) | (d[2] >> 4)
        adc_t = (d[3] << 12) | (d[4] << 4) | (d[5] >> 4)
        adc_h = (d[6] << 8) | d[7]

        # температура (формулы из документации Bosch, целые числа)
        v1 = (((adc_t >> 3) - (self.T1 << 1)) * self.T2) >> 11
        v2 = (((((adc_t >> 4) - self.T1) * ((adc_t >> 4) - self.T1)) >> 12)
              * self.T3) >> 14
        t_fine = v1 + v2
        temp = ((t_fine * 5 + 128) >> 8) / 100

        # давление
        v1 = t_fine - 128000
        v2 = v1 * v1 * self.P6
        v2 += (v1 * self.P5) << 17
        v2 += self.P4 << 35
        v1 = ((v1 * v1 * self.P3) >> 8) + ((v1 * self.P2) << 12)
        v1 = (((1 << 47) + v1) * self.P1) >> 33
        if v1 == 0:
            raise OSError("плохие данные давления")
        p = 1048576 - adc_p
        p = (((p << 31) - v2) * 3125) // v1
        v1 = (self.P9 * (p >> 13) * (p >> 13)) >> 25
        v2 = (self.P8 * p) >> 19
        p = ((p + v1 + v2) >> 8) + (self.P7 << 4)
        pres = p / 25600

        # влажность
        v = t_fine - 76800
        v = (((((adc_h << 14) - (self.H4 << 20) - (self.H5 * v)) + 16384) >> 15)
             * (((((((v * self.H6) >> 10) * (((v * self.H3) >> 11) + 32768))
                    >> 10) + 2097152) * self.H2 + 8192) >> 14))
```

```
        v -= (((((v >> 15) * (v >> 15)) >> 7) * self.H1) >> 4)
        v = max(0, min(v, 419430400))
        hum = (v >> 12) / 1024
        return temp, hum, pres


class Weather:
    """Хранит последние измерения, минимум и максимум; считает ошибки."""

    def __init__(self, i2c):
        self.i2c = i2c
        self.chip = None
        self.errors = 0          # ошибок подряд
        self.total_errors = 0
        self.t = self.h = self.p = None
        self.reset_minmax()

    def reset_minmax(self):
        self.tmin = self.tmax = None

    def ok(self):
        return self.chip is not None and self.errors == 0

    def update(self):
        """Обновляет данные. Возвращает True, если замер удался."""
        try:
            if self.chip is None:                       # датчик ещё не найден
                self.chip = BME280(self.i2c, config.BME_ADDR)
            t, h, p = self.chip.read()
            if not (-40 <= t <= 85 and 0 <= h <= 100 and 300 <= p <= 1100):
                raise OSError("значения вне диапазона")
        except OSError:
            self.errors += 1
            self.total_errors += 1
            self.chip = None                            # при следующей попытке инициализируем заново
            return False
        self.errors = 0
        self.t, self.h, self.p = t + config.TEMP_OFFSET, h, p
        if self.tmin is None or self.t < self.tmin:
            self.tmin = self.t
        if self.tmax is None or self.t > self.tmax:
            self.tmax = self.t
        return True

    def as_dict(self):
        return {"temp": round(self.t, 1), "hum": round(self.h, 1),
                "press": round(self.p, 1)}
```

Формулы пересчёта «сырых» чисел датчика в градусы, проценты и паскали взяты из его технической документации; нам важно не запоминать их, а видеть общий принцип: у каждого датчика на заводе записаны индивидуальные калибровочные коэффициенты, и программа обязана ими воспользоваться. Обратите внимание на три приёма защиты: проверка идентификатора микросхемы (`0x60`), проверка диапазонов значений и счётчик ошибок подряд (`errors`).

Листинг 25.5: MicroPython: `display.py` — четыре экрана

```
# display.py --- экран OLED 128x64 (SSD1306). Шрифт 8x8: 16 символов в строке.
# Библиотека ssd1306 ставится командой: mpremote mip install ssd1306
import ssd1306
import config

# шрифт экрана знает только латиницу, поэтому заголовки английские
TITLES = ("WEATHER", "MIN / MAX", "NETWORK", "SYSTEM")


class Screen:
    def __init__(self, i2c):
        self.oled = ssd1306.SSD1306_I2C(128, 64, i2c, config.OLED_ADDR)

    def _page(self, mode, lines):
        o = self.oled
```

```
        o.fill(0)
        o.fill_rect(0, 0, 128, 10, 1)            # белая шапка с заголовком
        o.text(TITLES[mode], 2, 1, 0)
        o.text("%d/4" % (mode + 1), 98, 1, 0)
        for i, s in enumerate(lines):
            o.text(s[:16], 0, 14 + i * 10)
        o.show()

    def draw(self, mode, w, net, uptime_s, free_mem):
        if mode == 0:
            if w.t is None:
                self._page(mode, ["No data yet", "check sensor"])
            else:
                self._page(mode, ["T  %6.1f C" % w.t, "H  %6.1f %%" % w.h,
                                  "P  %6.1f hPa" % w.p])
        elif mode == 1:
            if w.tmin is None:
                self._page(mode, ["No data yet"])
            else:
                self._page(mode, ["Tmin %6.1f C" % w.tmin, "Tmax %6.1f C" % w.tmax,
                                  "", "Hold button:", "reset min/max"])
        elif mode == 2:
            self._page(mode, ["WiFi: " + net.wifi_state(), net.ip(),
                              "MQTT: " + ("ok" if net.mqtt_ok else "--"),
                              "NTP:  " + ("ok" if net.ntp_done else "--")])
        else:
            h, m = divmod(uptime_s // 60, 60)
            self._page(mode, ["Up %dh %02dm" % (h, m),
                              "Sensor err: %d" % w.total_errors,
                              "WiFi drops: %d" % net.drops,
                              "Free RAM %dK" % (free_mem // 1024)])

    def message(self, text):
        self.oled.fill(0)
        self.oled.text(text[:16], 0, 28)
        self.oled.show()
```

Листинг 25.6: MicroPython: `net.py` — Wi-Fi, MQTT, веб-сервер

```
# net.py --- Wi-Fi, время (NTP), MQTT и маленький веб-сервер
import json
import network
import socket
import time
import ntptime
from umqtt.simple import MQTTClient    # mpremote mip install umqtt.simple
import config
import secrets


class Net:
    def __init__(self, log):
        self.log = log
        self.wlan = network.WLAN(network.STA_IF)
        self.wlan.active(True)
        # чтобы первая попытка началась сразу, «отодвигаем» прошлую попытку в прошлое
        self.last_try = time.ticks_add(time.ticks_ms(), -config.WIFI_RETRY_MS)
        self.was_connected = False
        self.drops = 0
        self.ntp_tries = 0
        self.ntp_done = False
        self.mqtt = None
        self.mqtt_ok = False

    # ---------- Wi-Fi ----------
    def connected(self):
        return self.wlan.isconnected()

    def wifi_state(self):
        return "ok" if self.connected() else "no link"

    def ip(self):
```

```
        return self.wlan.ifconfig()[0] if self.connected() else "-"

    def poll(self):
        """Вызывается в каждом шаге главного цикла. Не блокирует программу."""
        now = time.ticks_ms()
        if self.wlan.isconnected():
            if not self.was_connected:
                self.was_connected = True
                self.log.write("WiFi ok " + self.ip())
            if not self.ntp_done and self.ntp_tries < 3:
                self.ntp_tries += 1
                try:
                    ntptime.settime()          # время UTC
                    self.ntp_done = True
                    self.log.write("NTP ok")
                except OSError:
                    self.log.write("NTP fail")
            return
        # связи нет
        if self.was_connected:
            self.was_connected = False
            self.drops += 1
            self.mqtt_ok = False
            self.mqtt = None
            self.log.write("WiFi lost")
        if time.ticks_diff(now, self.last_try) >= config.WIFI_RETRY_MS:
            self.last_try = now
            self.wlan.connect(secrets.WIFI_SSID, secrets.WIFI_PASSWORD)  # возвращается сразу

    # ---------- MQTT ----------
    def publish(self, data):
        """Публикует измерения. При любой ошибке сети просто возвращает False."""
        if not self.connected():
            return False
        try:
            if self.mqtt is None:
                self.mqtt = MQTTClient("pico-weather", secrets.MQTT_BROKER,
                                       secrets.MQTT_PORT,
                                       secrets.MQTT_USER or None,
                                       secrets.MQTT_PASSWORD or None,
                                       keepalive=60)
                # «последняя воля»: брокер сам сообщит, если станция пропала
                self.mqtt.set_last_will((config.MQTT_BASE + "/status").encode(),
                                        b"offline", retain=True)
                self.mqtt.connect()
                self.mqtt.publish((config.MQTT_BASE + "/status").encode(),
                                  b"online", retain=True)
                self.log.write("MQTT ok")
            self.mqtt.publish((config.MQTT_BASE + "/state").encode(),
                              json.dumps(data).encode(), retain=True)
            self.mqtt_ok = True
            return True
        except OSError as e:
            self.mqtt_ok = False
            if self.mqtt is not None:
                try:
                    self.mqtt.sock.close()
                except Exception:
                    pass
                self.mqtt = None
            self.log.write("MQTT error %s" % e)
            return False

    def mqtt_ping(self):
        """Поддерживает соединение между публикациями."""
        if self.mqtt is not None and self.mqtt_ok:
            try:
                self.mqtt.ping()
            except OSError:
                self.mqtt_ok = False
                self.mqtt = None


PAGE = """<!doctype html><html><head><meta charset="utf-8">
```

```
<meta name="viewport" content="width=device-width,initial-scale=1">
<meta http-equiv="refresh" content="5"><title>Pico weather</title>
<style>body{font-family:sans-serif;text-align:center;margin-top:2em}
b{font-size:2.2em}</style></head><body>
<h2>Метеостанция Pico</h2>
<p>Температура: <b>%s</b> &deg;C</p>
<p>Влажность: <b>%s</b> %%</p>
<p>Давление: <b>%s</b> гПа</p>
<p><a href="/api">JSON</a> &middot; <a href="/log">журнал</a></p>
</body></html>"""


class Web:
    """Неблокирующий веб-сервер: в главном цикле вызываем poll()."""

    def __init__(self, get_data, get_log, port=80):
        self.get_data = get_data       # функции, которые отдаёт main.py
        self.get_log = get_log
        self.sock = socket.socket()
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.bind(("0.0.0.0", port))
        self.sock.listen(2)
        self.sock.setblocking(False)

    def _send(self, cl, ctype, body, status="200 OK"):
        body = body.encode()
        head = ("HTTP/1.0 %s\r\nContent-Type: %s\r\nContent-Length: %d\r\n"
                "Connection: close\r\n\r\n" % (status, ctype, len(body)))
        cl.send(head.encode() + body)

    def poll(self):
        try:
            cl, _ = self.sock.accept()     # нет клиента --- OSError, это нормально
        except OSError:
            return
        try:
            cl.settimeout(2)
            req = cl.recv(512).split(b"\r\n")[0].split()
            path = req[1] if len(req) > 1 else b"/"
            d = self.get_data()
            if path == b"/api":
                self._send(cl, "application/json", json.dumps(d))
            elif path == b"/log":
                self._send(cl, "text/plain; charset=utf-8", self.get_log())
            elif path == b"/":
                self._send(cl, "text/html; charset=utf-8",
                           PAGE % (d.get("temp", "--"), d.get("hum", "--"),
                                   d.get("press", "--")))
            else:
                self._send(cl, "text/plain", "404", "404 Not Found")
        except Exception:
            pass
        finally:
            cl.close()
```

Листинг 25.7: MicroPython: `main.py` — главный цикл

```
# main.py --- метеостанция: датчик, экран, кнопка, Wi-Fi, MQTT, веб, журнал, watchdog
import gc
import time
import machine
from machine import Pin, I2C, WDT
import config
from log import Log
from sensors import Weather
from display import Screen
from net import Net, Web

MODE_VALUES, MODE_MINMAX, MODE_NET, MODE_SYSTEM = 0, 1, 2, 3   # режимы экрана
N_MODES = 4


```

```
class Button:
    """Кнопка с защитой от дребезга; события «short» и «long»."""

    def __init__(self, pin):
        self.pin = Pin(pin, Pin.IN, Pin.PULL_UP)
        self.last = 1
        self.t_change = 0
        self.down_at = None
        self.long_done = False

    def update(self, now):
        v = self.pin.value()
        event = None
        if v != self.last and time.ticks_diff(now, self.t_change) > 30:
            self.last = v
            self.t_change = now
            if v == 0:                                  # нажали
                self.down_at = now
                self.long_done = False
            else:                                       # отпустили
                if self.down_at is not None and not self.long_done:
                    event = "short"
                self.down_at = None
        if (self.down_at is not None and not self.long_done and
                time.ticks_diff(now, self.down_at) >= config.LONG_PRESS_MS):
            self.long_done = True
            event = "long"
        return event


def due(last, now, period_ms):
    return time.ticks_diff(now, last) >= period_ms


def led_level(now, net, sensor_ok):
    """Рисунок мигания: что происходит с системой, видно издалека."""
    if not sensor_ok:
        return 1                                # горит постоянно: беда с датчиком
    if not net.connected():
        return (now // 150) % 2                 # быстро мигает: нет Wi-Fi
    if not net.mqtt_ok:
        return (now // 500) % 2                 # медленно мигает: нет MQTT
    return 1 if now % 3000 < 80 else 0          # короткая вспышка раз в 3 с: всё хорошо


def run():
    log = Log()
    log.write("START")
    i2c = I2C(config.I2C_ID, sda=Pin(config.PIN_SDA), scl=Pin(config.PIN_SCL),
              freq=400000)
    led = Pin(config.PIN_LED, Pin.OUT)
    button = Button(config.PIN_BTN)
    weather = Weather(i2c)
    net = Net(log)
    try:
        screen = Screen(i2c)
        screen.message("Starting...")
    except OSError:
        screen = None
        log.write("OLED not found")
    web = None
    try:
        web = Web(lambda: weather.as_dict() if weather.t is not None else {},
                  log.tail, config.WEB_PORT)
    except OSError as e:
        log.write("Web error %s" % e)

    wdt = WDT(timeout=config.WDT_TIMEOUT_MS) if config.USE_WDT else None

    mode = MODE_VALUES
    now = time.ticks_ms()
    t_sensor = t_display = t_pub = t_log = time.ticks_add(now, -10 ** 6)
    force_pub = False
    sensor_was_ok = True
```

```
90
91      while True:
92          now = time.ticks_ms()
93          if wdt:
94              wdt.feed()
95
96          # 1. кнопка: автомат режимов экрана
97          ev = button.update(now)
98          if ev == "short":
99              mode = (mode + 1) % N_MODES
100             t_display = time.ticks_add(now, -10 ** 6)    # перерисовать сразу
101         elif ev == "long":
102             weather.reset_minmax()
103             force_pub = True
104             log.write("min/max reset")
105
106         # 2. датчик
107         if due(t_sensor, now, config.SENSOR_PERIOD_MS):
108             t_sensor = now
109             weather.update()
110             if weather.ok() != sensor_was_ok:            # записываем только смену состояния
111                 sensor_was_ok = weather.ok()
112                 log.write("sensor " + ("ok" if sensor_was_ok else "ERROR"))
113             if weather.errors >= config.MAX_SENSOR_ERRORS:
114                 log.write("too many errors: reset")
115                 machine.reset()
116
117         # 3. сеть
118         net.poll()
119         if web:
120             web.poll()
121         if weather.t is not None and (force_pub or
122                                       due(t_pub, now, config.PUBLISH_PERIOD_S * 1000)):
123             t_pub = now
124             force_pub = False
125             net.publish(weather.as_dict())
126
127         # 4. экран
128         if screen and due(t_display, now, config.DISPLAY_PERIOD_MS):
129             t_display = now
130             try:
131                 screen.draw(mode, weather, net, now // 1000, gc.mem_free())
132             except OSError:
133                 log.write("OLED error")
134
135         # 5. журнал с измерениями раз в LOG_PERIOD_S
136         if weather.t is not None and due(t_log, now, config.LOG_PERIOD_S * 1000):
137             t_log = now
138             log.write("T=%.1f H=%.0f P=%.0f" % (weather.t, weather.h, weather.p))
139
140         led.value(led_level(now, net, weather.ok()))
141
142
143 try:
144     run()
145 except KeyboardInterrupt:
146     raise                       # Ctrl+C при отладке не должен перезагружать плату
147 except Exception as e:
148     Log().write("FATAL %r" % e)
149     time.sleep(2)
150     machine.reset()
```

Как устроен главный цикл

Цикл `while True` делает пять дел, и каждое устроено одинаково: «если с прошлого раза прошло достаточно времени — сделай и запомни момент». Для этого есть функция `due()`. Благодаря этому:

- кнопка опрашивается тысячи раз в секунду, поэтому нажатие не пропадает;
- датчик читается раз в 2 с, экран обновляется раз в 250 мс, MQTT — раз в 30 с, журнал —

раз в 10 минут;

- `wdt.feed()` вызывается в каждом шаге цикла: если какой-то блок «зависнет», кормление прекратится и сторожевой таймер перезагрузит плату;
- `try/except` вокруг `run()` записывает причину аварии в журнал и перезагружает плату.

Осторожно

Сторожевой таймер RP2040 нельзя отключить, пока он запущен. Если вы остановите программу в Thonny клавишами Ctrl+C, через 8 секунд плата перезагрузится и снова запустит `main.py`. На время отладки поставьте `USE_WDT = False` в `config.py`. Кроме того, самые длинные блокирующие вызовы (подключение к MQTT, NTP) не должны быть дольше времени таймера; брокер в локальной сети отвечает за доли секунды, но недоступный адрес может «подвесить» вызов на несколько секунд — поэтому таймер выбран с запасом.

25.8 Версия для CircuitPython

В **CircuitPython** та же станция получается короче благодаря готовым библиотекам Adafruit, но проще: нет собственного веб-сервера, журнала и режима «Система» (их можно добавить, см. задания). Секреты лежат в файле `settings.toml`, который читается функцией `os.getenv()`.

Листинг 25.8: **CircuitPython**: `settings.toml` (диск CIRCUITPY)

```
# settings.toml --- на диске CIRCUITPY; в репозиторий не добавляем
WIFI_SSID = "my-network"
WIFI_PASSWORD = "my-password"
MQTT_BROKER = "192.168.1.10"
MQTT_PORT = "1883"
```

Листинг 25.9: **CircuitPython**: `code.py` — упрощённая метеостанция

```
# code.py --- упрощённая метеостанция на CircuitPython 9.x
# Нужны библиотеки в папке lib: adafruit_bme280, adafruit_ssd1306,
# adafruit_minimqtt, adafruit_bus_device, adafruit_framebuf (всё ставится через circup)
import os
import time
import json
import board
import busio
import digitalio
import wifi
import socketpool
import microcontroller
from watchdog import WatchDogMode
from adafruit_bme280 import basic as adafruit_bme280
import adafruit_ssd1306
import adafruit_minimqtt.adafruit_minimqtt as MQTT

# ---------- настройки (секреты --- в settings.toml) ----------
SENSOR_PERIOD = 2        # секунды
PUBLISH_PERIOD = 30
TOPIC = "pico/weather/state"

i2c = busio.I2C(scl=board.GP5, sda=board.GP4)
bme = adafruit_bme280.Adafruit_BME280_I2C(i2c, address=0x76)
oled = adafruit_ssd1306.SSD1306_I2C(128, 64, i2c, addr=0x3C)

button = digitalio.DigitalInOut(board.GP14)
button.switch_to_input(pull=digitalio.Pull.UP)
led = digitalio.DigitalInOut(board.GP15)
led.direction = digitalio.Direction.OUTPUT

wdt = microcontroller.watchdog
wdt.timeout = 8          # секунды (максимум около 8,3 с)
```

```
wdt.mode = WatchDogMode.RESET

pool = socketpool.SocketPool(wifi.radio)
mqtt = MQTT.MQTT(
    broker=os.getenv("MQTT_BROKER"),
    port=int(os.getenv("MQTT_PORT", "1883")),
    socket_pool=pool,
    is_ssl=False,
)

MODES = 3               # 0 --- данные, 1 --- мин/макс, 2 --- сеть
mode = 0
tmin = tmax = None
mqtt_ok = False
last_btn = True
t_sensor = t_pub = -1000.0


def wifi_ok():
    return wifi.radio.ipv4_address is not None


def draw(t, h, p):
    oled.fill(0)
    if mode == 0:
        oled.text("WEATHER", 0, 0, 1)
        oled.text("T %5.1f C" % t, 0, 16, 1)
        oled.text("H %5.1f %%" % h, 0, 28, 1)
        oled.text("P %6.1f hPa" % p, 0, 40, 1)
    elif mode == 1:
        oled.text("MIN / MAX", 0, 0, 1)
        oled.text("Tmin %5.1f" % tmin, 0, 16, 1)
        oled.text("Tmax %5.1f" % tmax, 0, 28, 1)
    else:
        oled.text("NETWORK", 0, 0, 1)
        oled.text("WiFi " + ("ok" if wifi_ok() else "--"), 0, 16, 1)
        oled.text("MQTT " + ("ok" if mqtt_ok else "--"), 0, 28, 1)
    oled.show()


t, h, p = bme.temperature, bme.relative_humidity, bme.pressure
tmin = tmax = t

while True:
    wdt.feed()
    now = time.monotonic()

    # кнопка: переключение режима экрана по нажатию (фронт «отпущено -> нажато»)
    pressed = not button.value
    if pressed and last_btn:
        mode = (mode + 1) % MODES
        time.sleep(0.05)                # простая защита от дребезга
    last_btn = not pressed

    # Wi-Fi: пробуем подключиться, если связи нет
    if not wifi_ok():
        try:
            wifi.radio.connect(os.getenv("WIFI_SSID"), os.getenv("WIFI_PASSWORD"))
        except (ConnectionError, OSError):
            pass                        # повторим на следующем круге
        mqtt_ok = False

    # датчик
    if now - t_sensor >= SENSOR_PERIOD:
        t_sensor = now
        try:
            t, h, p = bme.temperature, bme.relative_humidity, bme.pressure
            tmin = t if tmin is None else min(tmin, t)
            tmax = t if tmax is None else max(tmax, t)
        except OSError:
            print("ошибка датчика")

    # публикация в MQTT
    if wifi_ok() and now - t_pub >= PUBLISH_PERIOD:
```

```
        t_pub = now
        try:
            if not mqtt_ok:
                mqtt.connect()
                mqtt_ok = True
            mqtt.publish(TOPIC, json.dumps(
                {"temp": round(t, 1), "hum": round(h, 1), "press": round(p, 1)}),
                retain=True)
        except Exception as e:            # любая сетевая ошибка: переподключимся позже
            print("MQTT:", e)
            mqtt_ok = False

    draw(t, h, p)
    led.value = wifi_ok() and mqtt_ok     # светодиод горит, когда всё в порядке
    time.sleep(0.1)
```

Сравните два варианта. В **CircuitPython** драйвер датчика и экрана — готовые библиотеки (`circup install adafruit_bme280 adafruit_ssd1306 adafruit_minimqtt`); в **MicroPython** драйвер BME280 мы написали сами, а для экрана взяли `ssd1306`. Принцип главного цикла одинаков: опросить, обновить, показать. Различается сторожевой таймер: в **CircuitPython** время задаётся в секундах и обслуживается объектом `microcontroller.watchdog`.

25.9 Ошибки и переподключения

Устройство на столе у разработчика работает всегда. Устройство в доме у пользователя сталкивается с пропавшим Wi-Fi, выключенным брокером, грязным контактом и отключением электричества. Поэтому для каждого узла продумываем, *что именно* сломается и *что* программа сделает.

Что сломалось	Как это видно	Реакция программы
Нет Wi-Fi при старте	светодиод мигает быстро, экран «Сеть»: `no link`	цикл работает; раз в 15 с новая попытка `connect()`
Wi-Fi пропал позже	запись `WiFi lost` в журнале, счётчик обрывов	сброс MQTT, повторные попытки, показания продолжаются
Брокер недоступен	светодиод мигает медленно	`publish()` ловит `OSError`, закрывает сокет, создаёт клиент заново
Нет ответа NTP	`NTP fail` в журнале	до трёх попыток, дальше метки времени — «время работы»
Датчик не отвечает	светодиод горит постоянно	значения замораживаются, счётчик ошибок; через 30 ошибок подряд — перезагрузка
Экран отключился	`OLED error` в журнале	станция продолжает публиковать данные без экрана
Журнал переполнен	размер файла $>$ 8 КБ	старый файл заменяется, работа не прерывается
Программа зависла	—	через 8 с срабатывает сторожевой таймер и плата перезагружается
Неисправимая ошибка в коде	`FATAL` в журнале	пауза 2 с и `machine.reset()`

Таблица 25.5: Карта отказов: что ломается и как реагирует станция

Общее правило: сетевая ошибка — это *нормальная жизнь*, а не авария. Ловим её узко (`except OSError`), записываем и продолжаем работу. «Голый» `except:`, поглощающий *любые* ошибки, прячет настоящие баги и мешает отладке.



Рис. 25.5: Сторожевой таймер: пока программа кормит его, всё в порядке; зависание заканчивается перезагрузкой

25.10 Веб-страница, JSON и MQTT

Данные станция отдаёт тремя способами. Адрес в локальной сети показывает экран «Сеть».

Адрес / тема	Что отдаёт	Для кого
`http://192.168.1.57/`	HTML-страница, обновляется каждые 5 с	человек с телефоном
`http://192.168.1.57/api`	JSON: `{"temp": 22.4, "hum": 41.0, "press": 1013.2}`	другая программа
`http://192.168.1.57/log`	последние строки журнала (текст)	разработчик
`pico/weather/state`	то же JSON, `retain=True`	Home Assistant, Node-RED, график
`pico/weather/status`	`online` / `offline` (последняя воля)	мониторинг

Таблица 25.6: Интерфейсы станции (IP-адрес — пример, у вас будет свой)

Проверить MQTT можно с компьютера, на котором работает брокер Mosquitto:

```
mosquitto_sub -h 192.168.1.10 -t "pico/weather/#" -v
```

Каждые 30 секунд должна появляться строка вида `pico/weather/state {"temp": 22.4, "hum": 41.0, "press": 1013.2}`.

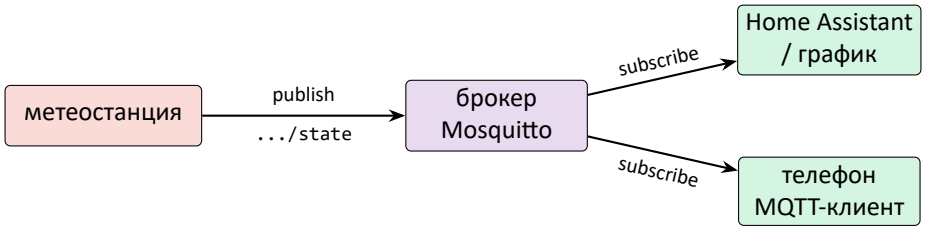


Рис. 25.6: Схема публикации–подписки: станция ничего не знает о получателях

Это интересно

Флаг `retain=True` просит брокер хранить последнее сообщение. Новый подписчик получит его сразу, не дожидаясь следующей публикации через 30 с. А «последняя воля» (`set_last_will`) — сообщение, которое брокер сам отправит, если станция вдруг пропадёт без прощания: так видно, что устройство `offline`.

25.11 Журнал на flash

Файл `log.txt` можно прочитать тремя способами: по адресу `/log` в браузере, командой `mpremote cat :log.txt` или в Thonny. Пример содержимого:

```
up1s START
up6s WiFi ok 192.168.1.57
up7s NTP ok
12:04:11 MQTT ok
12:14:11 T=22.4 H=41 P=1013
12:31:40 WiFi lost
12:31:55 WiFi ok 192.168.1.57
```

По журналу видно историю жизни устройства: когда оно перезапускалось, когда пропадала связь. Именно такие записи позволяют разобраться, почему станция «странно вела себя ночью».

25.12 Этапы сборки и тестирования

Не собирайте всё сразу. Двигайтесь этапами и после каждого проверяйте результат — тогда ошибка всегда в последнем шаге.

1. Собрать схему, запустить `i2c.scan()` — должны появиться оба адреса.
2. Прочитать BME280 из REPL, сравнить с эталонным термометром.
3. Вывести значения на OLED одним экраном.
4. Добавить кнопку и автомат режимов.
5. Добавить светодиод статуса.
6. Подключить Wi-Fi и показать IP на экране «Сеть».
7. Включить веб-страницу, потом MQTT.
8. Добавить журнал, затем сторожевой таймер.
9. Провести испытания на отказ (чек-лист ниже).

	Испытание	Т
☐	Датчик и экран найдены сканером I2C: `0x3C`, `0x76`	Т1
☐	Температура отличается от эталона не более чем на 1 °C	Т1
☐	Четыре экрана листаются по кругу, 20 нажатий без пропусков и «двойных» срабатываний	Т2
☐	Долгое нажатие сбрасывает минимум/максимум, короткое — нет	Т3
☐	Роутер выключен: станция работает, светодиод мигает быстро	Т4
☐	Роутер включён: Wi-Fi возвращается без перезагрузки платы	Т5
☐	Брокер остановлен и запущен: публикации возобновляются сами	Т5, Т6
☐	Датчик отсоединён и подключён обратно: значения снова обновляются	Т4
☐	Телефон открывает страницу и `/api`	Т7
☐	Размер `log.txt` не превышает 8 КБ после суток работы	Т8
☐	Питание выдернуто на ходу: после включения станция стартует сама	Т8
☐	Искусственная бесконечная петля: плата перезагружается через 8 с	Т9
☐	В репозитории нет паролей, `secrets.py` в `.gitignore`	Т10

Таблица 25.7: Чек-лист приёмочных испытаний (колонка «Т» — номер требования)

25.13 Критерии оценивания

Критерий	Что смотрим	Баллы
Работоспособность	выполнены требования Т1–Т3, устройство собрано и запускается само при включении	20
Схема и монтаж	схема соответствует чертежу, аккуратные провода, продуманы цвета, нет перепутанной полярности	10
Структура кода	модули с одной задачей, настройки вынесены, нет «магических чисел», понятные имена	15
Надёжность	отказ Wi-Fi, брокера, датчика не останавливают станцию; работает watchdog (Т4, Т5, Т9)	15
Сеть и данные	рабочая веб-страница, JSON, MQTT (Т6, Т7)	15
Журнал и безопасность	ограниченный журнал, нет паролей в коде и репозитории (Т8, Т10)	10
Тестирование	заполненный чек-лист, описание найденных и исправленных ошибок	5
Защита проекта	рассказ, демонстрация, ответы на вопросы	10
Итого		**100**

Таблица 25.8: Критерии оценивания итогового проекта

Для оценки «5» нужно набрать не менее 85 баллов, для «4» — не менее 65, для «3» — не менее 45. Дополнительные баллы дают идеи расширения.

25.14 Идеи расширения

- **Дисплей на электронных чернилах (e-ink).** Такой экран подключается по SPI (урок 9), потребляет энергию только при обновлении картинки и сохраняет изображение без питания. Обновлять его нужно редко (раз в несколько минут), поэтому он хорошо подходит для станции на батарее.
- **Солнечная панель и сон.** Станции, питающейся от аккумулятора, не нужно работать постоянно: замерить, отправить данные и уснуть (урок 15). Самые экономные режимы сна «забывают» программу, и после пробуждения всё начинается с начала. Подумайте, как вернуть нужные настройки и как экономить энергию Wi-Fi.
- **Home Assistant.** Станция уже публикует JSON в MQTT, остаётся описать датчики в Home Assistant (через конфигурацию или автообнаружение MQTT) и получить графики, уведомления и автоматизации (например, включить вентилятор при влажности выше 70 %).
- **История на экране.** Храните последние 60 измерений в кольцевом буфере и рисуйте график температуры линией (`oled.line`).
- **Второй датчик.** Датчик освещённости или CO_2 на той же шине I2C и новая тема в MQTT.
- **Безопасность.** Пароль на веб-страницу, `TLS` для MQTT, проверка длины запроса.
- **Управление с телефона.** Подписка на тему `pico/weather/cmd`: команда «сбросить min/max» или «сменить экран».

25.15 Защита проекта

Защита — это не пересказ кода, а рассказ инженера о работе. Рассчитайте 5–7 минут:

1. **Задача** (30 с): что делает станция, кому она нужна.

2. **Архитектура** (1 мин): блок-схема, модули, как устроен главный цикл.
3. **Демонстрация** (2 мин): работа экранов, кнопка, страница в телефоне, сообщение MQTT.
4. **Испытание на отказ** (1 мин): выключить Wi-Fi или отсоединить датчик и показать реакцию.
5. **Что не получилось и чему научились** (1 мин): честный рассказ о найденных ошибках.
6. **Планы развития** (30 с): одна идея расширения.

Вопросы для подготовки к защите.

- Почему в программе нет `sleep()` и как вместо него отсчитывается время?
- Что произойдёт, если выключить Wi-Fi, и откуда вы это знаете (покажите по журналу)?
- Зачем нужен сторожевой таймер и чем опасно его слишком маленькое время?
- Почему журнал нельзя писать каждую секунду?
- Как программа различает короткое и долгое нажатие кнопки?
- Зачем секреты вынесены в отдельный файл?
- Что изменится, если добавить второй датчик на ту же шину I2C?

Если не работает

- **`ImportError: no module named 'ssd1306'` / `'umqtt'`** — библиотеки не установлены; выполните команды `mip install` из начала раздела.
- **`OSError: [Errno 19] ENODEV` при запуске** — устройство I2C не отвечает; запустите сканер шины и проверьте адреса, провода SDA/SCL и питание.
- **Станция перезагружается каждые 8 с** — главный цикл не доходит до `wdt.feed()` (где-то блокирующий вызов) либо вы остановили программу Ctrl+C при включённом сторожевом таймере.
- **Wi-Fi не подключается** — пароль введён неверно, сеть 5 ГГц (Pico W понимает только 2,4 ГГц), либо в имени сети лишние пробелы.
- **Страница не открывается** — телефон в другой сети (например, гостевой), либо вы ввели IP-адрес с `https://` вместо `http://`.
- **Значения давления «прыгают»** — проверьте, что не перепутаны калибровочные коэффициенты и что датчик не лежит рядом с источником тепла (радиатор, зарядка).
- **Файловая система переполнена** — проверьте размер `log.txt` и `log.txt.old`.

Итоги урока

- Большой проект начинается с требований, проверяемых опытом, чертежа и бюджета ресурсов.
- Один главный цикл без `sleep()` опрашивает кнопку, датчик, сеть и экран по расписанию; режимы экрана — конечный автомат.
- Код делят на модули с одной задачей; настройки и секреты хранят отдельно от логики.
- Любая сетевая ошибка — норма: ловим узко, записываем в журнал, повторяем позже; сторожевой таймер страхует от зависания.
- Журнал пишут по событиям и ограничивают по размеру, чтобы не «убить» flash.
- Устройство отдаёт данные тремя способами: HTML, JSON и MQTT; готовность проверяют чек-листом.

Проверь себя

1. Чем требование отличается от пожелания и как проверить требование «станция надёжна»?
2. Почему главный цикл станции не содержит `sleep()` и как он вместо этого отмеряет время?
3. Как автомат режимов экрана реагирует на короткое и на долгое нажатие кнопки?
4. Что делает станция, когда пропал Wi-Fi, и что — когда датчик перестал отвечать?
5. Зачем журнал на flash ограничивают по размеру и пишут не каждую секунду?

Задания для самостоятельной работы

★ повторить ★★ изменить ★★★ создать своё ★★★★ вызов

1. ★ Соберите схему по рисунку и добейтесь, чтобы сканер I2C показал адреса `0x3C` и `0x76`.
2. ★ Запустите станцию и заполните чек-лист испытаний; найденные отклонения запишите.
3. ★★ Добавьте на экран «Данные» строку с текущим временем (по NTP) и поясните, что показывается до синхронизации.
4. ★★ Измените светодиод статуса: добавьте отдельный рисунок мигания для состояния «Wi-Fi есть, NTP ещё нет».
5. ★★★ Добавьте пятый режим экрана с графиком температуры за последние 60 измерений; нарисуйте новую диаграмму автомата.
6. ★★★★ Реализуйте в **MicroPython** подписку на тему `pico/weather/cmd`: команда `reset` сбрасывает минимум и максимум, команда `next` переключает экран. Не используйте блокирующее ожидание.

Приложение А. Справочник

Это приложение — «шпаргалка»: таблицы, к которым удобно возвращаться во время работы. Данные приведены для Raspberry Pi Pico, Pico W и Pico 2 (RP2350); отличия моделей указаны отдельно. Значения токов и напряжений — ориентировочные; точные цифры смотрите в технической документации (раздел «Полезные ссылки» в конце).

А.1 Распиновка Pico

Контакты нумеруются по физическому расположению: USB сверху, левый ряд читается сверху вниз (1–20), правый — снизу вверх (21–40). В программе всегда указывают номер **GP**, а не номер контакта. Распиновка у Pico, Pico W и Pico 2 *одинаковая*.

№	Имя	№	Имя	№	Имя	№	Имя
1	GP0	11	GP8	21	GP16	31	GP26 (ADC0)
2	GP1	12	GP9	22	GP17	32	GP27 (ADC1)
3	GND	13	GND	23	GND	33	AGND
4	GP2	14	GP10	24	GP18	34	GP28 (ADC2)
5	GP3	15	GP11	25	GP19	35	ADC_VREF
6	GP4	16	GP12	26	GP20	36	3V3 (выход)
7	GP5	17	GP13	27	GP21	37	3V3_EN
8	GND	18	GND	28	GND	38	GND
9	GP6	19	GP14	29	GP22	39	VSYS
10	GP7	20	GP15	30	RUN	40	VBUS

Таблица А.1: Контакты платы Pico по порядку

Функции выводов по таблице

Любой вывод GPIO может быть простым входом или выходом; встроенные устройства (UART, SPI, I2C, PWM) подключаются только к определённым выводам. Если нужная функция «заняла» вывод, выбирайте другой вариант из этой таблицы.

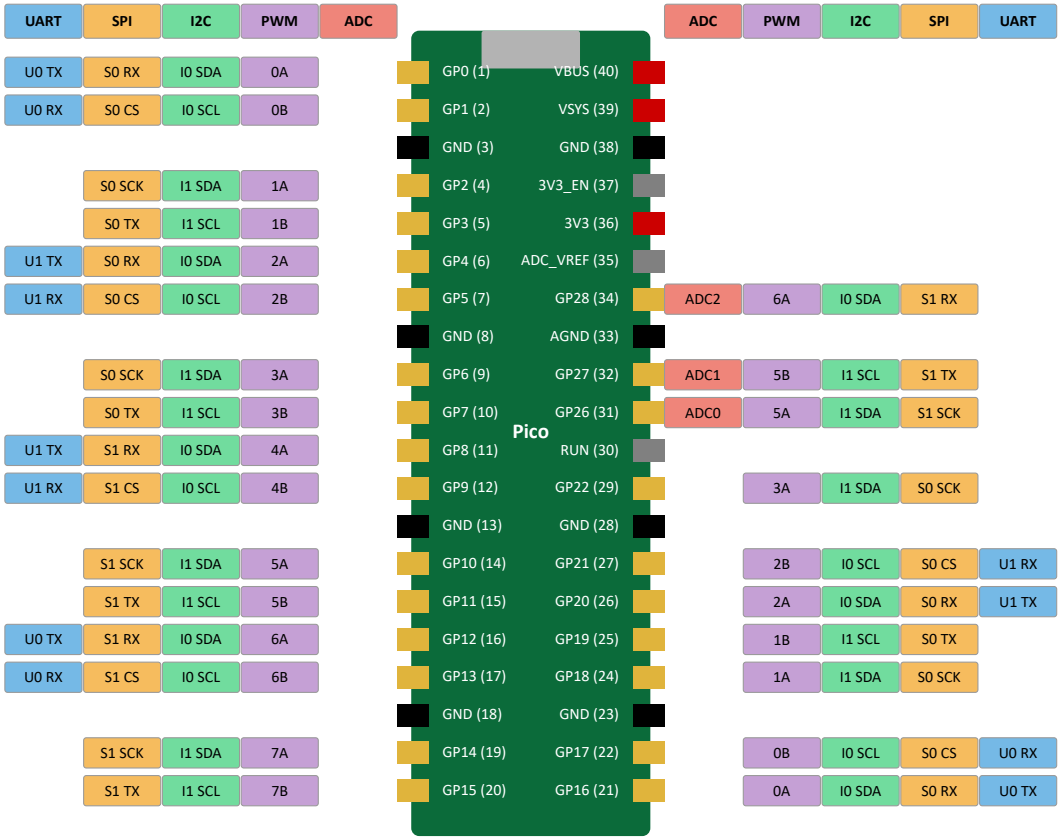


Рис. А.1: Карта выводов Pico: рядом с каждым GPIO цветом показано, какие функции на нём доступны. Синий — UART, оранжевый — SPI, зелёный — I2C, фиолетовый — PWM (слайс и канал), красный — АЦП. Чёрные контакты — земля, красные — питание

GPIO	Контакт	UART	SPI	I2C	PWM	ADC
GP0	1	U0 TX	S0 RX	I0 SDA	0A	—
GP1	2	U0 RX	S0 CS	I0 SCL	0B	—
GP2	4	—	S0 SCK	I1 SDA	1A	—
GP3	5	—	S0 TX	I1 SCL	1B	—
GP4	6	U1 TX	S0 RX	I0 SDA	2A	—
GP5	7	U1 RX	S0 CS	I0 SCL	2B	—
GP6	9	—	S0 SCK	I1 SDA	3A	—
GP7	10	—	S0 TX	I1 SCL	3B	—
GP8	11	U1 TX	S1 RX	I0 SDA	4A	—
GP9	12	U1 RX	S1 CS	I0 SCL	4B	—
GP10	14	—	S1 SCK	I1 SDA	5A	—
GP11	15	—	S1 TX	I1 SCL	5B	—
GP12	16	U0 TX	S1 RX	I0 SDA	6A	—
GP13	17	U0 RX	S1 CS	I0 SCL	6B	—
GP14	19	—	S1 SCK	I1 SDA	7A	—
GP15	20	—	S1 TX	I1 SCL	7B	—
GP16	21	U0 TX	S0 RX	I0 SDA	0A	—
GP17	22	U0 RX	S0 CS	I0 SCL	0B	—
GP18	24	—	S0 SCK	I1 SDA	1A	—
GP19	25	—	S0 TX	I1 SCL	1B	—
GP20	26	U1 TX	S0 RX	I0 SDA	2A	—
GP21	27	U1 RX	S0 CS	I0 SCL	2B	—
GP22	29	—	S0 SCK	I1 SDA	3A	—
GP26	31	—	S1 SCK	I1 SDA	5A	ADC0
GP27	32	—	S1 TX	I1 SCL	5B	ADC1
GP28	34	—	S1 RX	I0 SDA	6A	ADC2

Таблица А.2: Функции выводов GPIO (Pico, Pico W, Pico 2). U0/U1 — UART0/1, S0/S1 — SPI0/1 (RX = MISO, TX = MOSI, CS = CSn), I0/I1 — I2C0/1; PWM: номер слайса и канал A/B

Совет

Правило для запоминания. Роль вывода в SPI определяет остаток от деления номера на 4: 0 — RX, 1 — CS, 2 — SCK, 3 — TX. В I2C чётные выводы — SDA, нечётные — SCL. В PWM слайс равен $(n \bmod 16) / 2$ (дробная часть отбрасывается), чётные выводы — канал A, нечётные — B. GP0 и GP1 — один слайс (общая частота), а GP0 и GP16 — один и тот же канал: одновременно их использовать нельзя.

Отличия моделей

	Pico	Pico W	Pico 2 / Pico 2 W
Микроконтроллер	RP2040, 2 ядра Cortex-M0+	RP2040	RP2350, 2 ядра Cortex-M33 или Hazard3
Частота (стандартная)	125 МГц	125 МГц	150 МГц
ОЗУ	264 КБ	264 КБ	520 КБ
Flash	2 МБ	2 МБ	4 МБ
Wi-Fi и Bluetooth	нет	есть (CYW43)	нет / есть (версия W)
Встроенный светодиод	GP25	на радиочипе, `"LED"`	GP25 / на радиочипе
Внешние GPIO	26 (GP0–GP22, GP26–GP28)	те же	те же
Контроллер USB	встроенный	встроенный	встроенный

Таблица А.3: Сравнение моделей

А.2 Электрические характеристики

Параметр	Значение	Комментарий
Напряжение логики GPIO	$3{,}3$ V	выводы *не* рассчитаны на 5 V
Допустимое напряжение на выводе	$0 \ldots 3{,}3$ В (предел — около 3,6 В)	превышение — риск повреждения
Ток одного вывода (по умолчанию)	4 мА	программно 2, 4, 8 или 12 мА
Суммарный ток всех выводов	порядка 50 мА	для мощных нагрузок — транзистор или драйвер
Питание по USB (VBUS)	5 V	берётся с USB-порта
Питание VSYS (внешнее)	примерно $1{,}8$ – $5{,}5$ V	подключается через диод к VBUS
Выход 3V3 (вывод 36)	$3{,}3$ V	ток — порядка десятков–сотни мА, остальное потребляет плата
Опорное напряжение АЦП	$3{,}3$ V (вывод ADC_VREF)	АЦП измеряет $0 \ldots 3{,}3$ В
АЦП	12 бит (0–4095), 4 входа + температура	в **MicroPython** `read_u16()` масштабирует до 0–65535
Частота ядра	125 МГц (RP2040), 150 МГц (RP2350)	можно менять, но не нужно новичкам
Подтяжки (pull-up/pull-down)	около $50\,\text{k}\Omega$	включаются программно
Рабочая температура	около $-20 \ldots +85$ °C	на платах и кристалле
Потребление платы (без нагрузки)	десятки мА	Wi-Fi добавляет десятки мА, пики — больше

Таблица А.4: Основные электрические характеристики (ориентировочно)

А.3 MicroPython и CircuitPython: сравнение API

В таблицах — самые частые задачи. Всё, что связано с платой, в **CircuitPython** собрано в модуле `board` (например, `board.GP15`), в **MicroPython** используются числа.

Задача	MicroPython	CircuitPython
Импорт	`from machine import Pin`	`import board, digitalio`
Выход	`led = Pin(15, Pin.OUT)`	`led = digitalio.DigitalInOut(board.GP15); led.direction = digitalio.Direction.OUTPUT`
Включить	`led.value(1)` или `led.on()`	`led.value = True`
Переключить	`led.toggle()`	`led.value = not led.value`
Вход с подтяжкой	`b = Pin(14, Pin.IN, Pin.PULL_UP)`	`b = digitalio.DigitalInOut(board.GP14); b.switch_to_input(pull=digitalio.Pull.UP)`
Прочитать вход	`b.value()`	`b.value`
Встроенный светодиод	`Pin("LED", Pin.OUT)`	`board.LED`
Прерывание	`b.irq(trigger=Pin.IRQ_FALLING, handler=f)`	нет прерываний; keypad или опрос

Таблица А.5: Цифровые входы и выходы

Задача	MicroPython	CircuitPython
ШИМ (создание)	`pwm = PWM(Pin(15), freq=1000, duty_u16=0)`	`pwm = pwmio.PWMOut(board.GP15, frequency=1000)`
Скважность	`pwm.duty_u16(32768)` (0–65535)	`pwm.duty_cycle = 32768` (0–65535)
Частота	`pwm.freq(2000)`	`pwm.frequency = 2000`
Выключить ШИМ	`pwm.deinit()`	`pwm.deinit()`
АЦП (создание)	`adc = ADC(Pin(26))`	`adc = analogio.AnalogIn(board.GP26)`
Прочитать АЦП	`adc.read_u16()`	`adc.value`
Напряжение	`adc.read_u16() * 3.3 / 65535`	`adc.value * adc.reference_voltage / 65535`

Таблица А.6: ШИМ и АЦП

Задача	MicroPython	CircuitPython
I2C (создание)	`i2c = I2C(0, sda=Pin(4), scl=Pin(5), freq=400000)`	`i2c = busio.I2C(scl=board.GP5, sda=board.GP4)`
Сканировать шину	`i2c.scan()`	`i2c.try_lock(); i2c.scan(); i2c.unlock()`
Записать байты	`i2c.writeto(0x3C, b"\x00")`	`i2c.writeto(0x3C, bytes([0]))` (внутри lock)
Прочитать байты	`i2c.readfrom(0x76, 2)`	`i2c.readfrom_into(0x76, buf)`
Регистр датчика	`i2c.readfrom_mem(0x76, 0xD0, 1)`	`I2CDevice` из `adafruit_bus_device`
SPI (создание)	`spi = SPI(0, baudrate=1_000_000, sck=Pin(2), mosi=Pin(3), miso=Pin(4))`	`spi = busio.SPI(board.GP2, MOSI=board.GP3, MISO=board.GP4)`
Выбор устройства (CS)	`cs = Pin(5, Pin.OUT, value=1); cs(0) … cs(1)`	`digitalio`, как выход
Запись SPI	`spi.write(b"\x01\x02")`	`spi.try_lock(); spi.write(buf); spi.unlock()`
UART (создание)	`uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))`	`uart = busio.UART(board.GP0, board.GP1, baudrate=9600)`
Отправить	`uart.write(b"hello")`	`uart.write(b"hello")`
Принять	`uart.read()`, `uart.any()`	`uart.read(10)`, `uart.in_waiting`

Таблица А.7: Шины I2C, SPI и UART

Задача	MicroPython	CircuitPython
Wi-Fi (включить)	`wlan = network.WLAN(network.STA_IF); wlan.active(True)`	`import wifi` (радиомодуль уже есть)
Подключиться	`wlan.connect(ssid, pw)`	`wifi.radio.connect(ssid, pw)`
Проверить связь	`wlan.isconnected()`	`wifi.radio.ipv4_address is not None`
IP-адрес	`wlan.ifconfig()[0]`	`wifi.radio.ipv4_address`
Сокет	`socket.socket()`	`socketpool.SocketPool(wifi.radio)`, затем `pool.socket()`
Время (мс)	`time.ticks_ms()`, `time.ticks_diff(a, b)`	`time.monotonic()` (секунды, дробные)
Пауза	`time.sleep(1)`, `time.sleep_ms(10)`	`time.sleep(1)`
Время и дата	`time.localtime()`, `ntptime.settime()`	`rtc.RTC().datetime` + библиотека `adafruit_ntp`

Таблица А.8: Wi-Fi и время

Задача	MicroPython	CircuitPython
Файлы (чтение)	`open("a.txt").read()`	то же, но диск по умолчанию доступен для записи только с компьютера
Файлы (запись)	`with open("a.txt", "w") as f: f.write("x")`	нужен `storage.remount("/", readonly=False)` в `boot.py`
Список файлов	`os.listdir()`	`os.listdir()`
Сторожевой таймер	`WDT(timeout=5000)`, `wdt.feed()`	`microcontroller.watchdog: .timeout = 5, .mode = WatchDogMode.RESET`
Лёгкий сон	`machine.lightsleep(1000)`	`alarm.light_sleep_until_alarms(alarm.time.TimeAla`
Глубокий сон	`machine.deepsleep(1000)`	`alarm.exit_and_deep_sleep_until_alarms(...)`
Перезагрузка	`machine.reset()`	`microcontroller.reset()`
Свободная память	`gc.mem_free()`	`gc.mem_free()`
Установка библиотек	`mpremote mip install имя`	`circup install имя`
Файл запуска	`main.py` (и `boot.py`)	`code.py` (и `boot.py`, `settings.toml`)

Таблица А.9: Файлы, сон, сторожевой таймер, служебное

А.4 Коды цветов резисторов

На резисторе чаще всего четыре или пять полос. Для четырёх полос: первая и вторая — цифры, третья — множитель, четвёртая — точность. Резистор читают со стороны, ближней к краю (полоса точности обычно золотая или серебряная и стоит отдельно).

Цвет	Название	Цифра	Множитель	Точность
	чёрный	0	×1	—
	коричневый	1	×10	±1 %
	красный	2	×100	±2 %
	оранжевый	3	×1 к	—
	жёлтый	4	×10 к	—
	зелёный	5	×100 к	±0,5 %
	синий	6	×1 М	±0,25 %
	фиолетовый	7	×10 М	±0,1 %
	серый	8	—	±0,05 %
	белый	9	—	—
	золотой	—	×0,1	±5 %
	серебряный	—	×0,01	±10 %

Таблица А.10: Цветовая маркировка резисторов

Примеры. Красный–красный–коричневый–золотой: $22 \times 10 = 220\,\Omega$, ±5 %. Коричневый–чёрный–красный: $10 \times 100 = 1\,\text{k}\Omega$. Оранжевый–оранжевый–красный: $33 \times 100 = 3{,}3\,\text{k}\Omega$. Жёлтый–фиолетовый–оранжевый: $47 \times 1000 = 47\,\text{k}\Omega$.

А.5 Частоты нот

Для звука (урок 21) нужны частоты нот. Каждая следующая октава вдвое выше; ноту получают умножением частоты предыдущей на $\sqrt[12]{2} \approx 1{,}0595$.

Нота	Латиница	3-я окт., Гц	4-я окт., Гц	5-я окт., Гц	6-я окт., Гц
до	C	131	262	523	1047
до♯	C#	139	277	554	1109
ре	D	147	294	587	1175
ре♯	D#	156	311	622	1245
ми	E	165	330	659	1319
фа	F	175	349	698	1397
фа♯	F#	185	370	740	1480
соль	G	196	392	784	1568
соль♯	G#	208	415	831	1661
ля	A	220	440	880	1760
ля♯	A#	233	466	932	1865
си	B	247	494	988	1976

Таблица А.11: Частоты нот равномерно темперированного строя (ля первой октавы $= 440$ Гц; «4-я октава» — от до$_4$ = 262 Гц)

А.6 Таблица ASCII

Символы в компьютере хранятся числами. Например, строка `"Hi"` — это байты 72 и 105. Русские буквы в стандарте ASCII отсутствуют: в UTF-8 они занимают по два байта.

Код	HEX	Символ	Код	HEX	Символ	Код	HEX	Символ	Код	HEX	Символ
32	20	пробел	56	38	8	80	50	P	104	68	h
33	21	!	57	39	9	81	51	Q	105	69	i
34	22	"	58	3A	:	82	52	R	106	6A	j
35	23	#	59	3B	;	83	53	S	107	6B	k
36	24	$	60	3C	<	84	54	T	108	6C	l
37	25	%	61	3D	=	85	55	U	109	6D	m
38	26	&	62	3E	>	86	56	V	110	6E	n
39	27	'	63	3F	?	87	57	W	111	6F	o
40	28	(	64	40	@	88	58	X	112	70	p
41	29	)	65	41	A	89	59	Y	113	71	q
42	2A	*	66	42	B	90	5A	Z	114	72	r
43	2B	+	67	43	C	91	5B	[	115	73	s
44	2C	,	68	44	D	92	5C	\	116	74	t
45	2D	-	69	45	E	93	5D	]	117	75	u
46	2E	.	70	46	F	94	5E	^	118	76	v
47	2F	/	71	47	G	95	5F	_	119	77	w
48	30	0	72	48	H	96	60	`	120	78	x
49	31	1	73	49	I	97	61	a	121	79	y
50	32	2	74	4A	J	98	62	b	122	7A	z
51	33	3	75	4B	K	99	63	c	123	7B	{
52	34	4	76	4C	L	100	64	d	124	7C	|
53	35	5	77	4D	M	101	65	e	125	7D	}
54	36	6	78	4E	N	102	66	f	126	7E	~
55	37	7	79	4F	O	103	67	g	127	7F	DEL

Таблица А.12: Печатные символы ASCII (коды 32–126). Управляющие коды: 8 — Backspace, 9 — Tab, 10 — перевод строки (LF), 13 — возврат каретки (CR), 27 — Esc

А.7 Единицы измерения и приставки

Приставка	Обозначение	Множитель	Величина	Единица	Обозначение
пико	п (p)	10^{-12}	напряжение	вольт	В (V)
нано	н (n)	10^{-9}	ток	ампер	А (A)
микро	мк (µ)	10^{-6}	сопротивление	ом	Ω
милли	м (m)	10^{-3}	ёмкость	фарад	Ф (F)
кило	к (k)	10^{3}	частота	герц	Гц (Hz)
мега	М (M)	10^{6}	мощность	ватт	Вт (W)
гига	Г (G)	10^{9}	энергия	джоуль	Дж (J)

Таблица А.13: Приставки СИ и электрические величины

Для памяти: 1 КБ в программировании — 1024 байта, 1 МБ — 1024 КБ; 1 Мбит/с — миллион бит в секунду.

А.8 Основные формулы

Закон	Формула	Пояснение
Закон Ома	$I = \dfrac{U}{R}$	ток через резистор
Мощность	$P = U \cdot I = I^2 R = \dfrac{U^2}{R}$	нагрев резистора; резистор на 0,25 Вт
Резистор для светодиода	$R = \dfrac{U_{\text{пит}} - U_F}{I}$	U_F — прямое напряжение светодиода
Делитель напряжения	$U_{\text{вых}} = U_{\text{вх}} \dfrac{R_2}{R_1 + R_2}$	потенциометр, датчики с резистором
Резисторы последовательно	$R = R_1 + R_2$	сопротивления складываются
Резисторы параллельно	$R = \dfrac{R_1 R_2}{R_1 + R_2}$	общее меньше меньшего
RC-цепочка	$\tau = R \cdot C$	время заряда конденсатора (за τ — 63 %)
Частота и период	$f = \dfrac{1}{T}$	Гц и секунды
Скважность ШИМ	$D = \dfrac{t_{\text{имп}}}{T}$	среднее напряжение $\approx D \cdot 3{,}3$ В
Значение АЦП в вольтах	$U = \dfrac{N}{4095} \cdot 3{,}3$	N — 12-битное значение
Скорость UART	$T_{\text{бит}} = \dfrac{1}{\text{бод}}$	при 9600 бод бит длится около 104 мкс

Таблица А.14: Формулы для расчётов в проектах

А.9 Команды инструментов

mpremote (MicroPython)

Команда	Что делает
`mpremote connect list`	показать подключённые платы и порты
`mpremote`	открыть REPL (выход — Ctrl+])
`mpremote run файл.py`	выполнить файл на плате, не сохраняя
`mpremote cp файл.py :`	скопировать файл на плату
`mpremote cp :main.py .`	скопировать файл с платы на компьютер
`mpremote ls`	список файлов на плате
`mpremote cat :main.py`	вывести содержимое файла
`mpremote rm :main.py`	удалить файл на плате
`mpremote mkdir :lib`	создать каталог
`mpremote mip install имя`	установить пакет из репозитория (`ssd1306`, `umqtt.simple`)
`mpremote reset`	перезагрузить плату
`mpremote bootloader`	перевести плату в режим BOOTSEL

Таблица А.15: Часто используемые команды mpremote

circup (CircuitPython)

Команда	Что делает
`circup list`	какие библиотеки установлены на плате и их версии
`circup install имя`	установить библиотеку в `CIRCUITPY/lib`
`circup update`	обновить устаревшие библиотеки
`circup freeze`	список библиотек в формате файла зависимостей
`circup uninstall имя`	удалить библиотеку
`circup show`	показать, какие библиотеки есть в наборе Adafruit

Таблица А.16: Команды circup

picotool (аналог esptool для Pico)

Если вы знакомы с платами ESP, то `esptool` для них — то же, что `picotool` для Raspberry Pi Pico: программа для прошивки и просмотра платы, подключённой к компьютеру. Плату нужно перевести в режим BOOTSEL (кнопка при подключении USB) или использовать ключ `-f`.

Команда	Что делает
`picotool info`	сведения о плате и загруженной программе
`picotool load -f файл.uf2`	записать прошивку (`-f` — при необходимости перезагрузить плату в BOOTSEL)
`picotool save -a файл.uf2`	сохранить содержимое flash в файл
`picotool verify файл.uf2`	сверить flash с файлом
`picotool reboot -f`	перезагрузить плату

Таблица А.17: Основные команды picotool

Для прошивки без инструментов достаточно перетащить файл `.uf2` на диск `RPI-RP2` (у Pico 2 — `RP2350`), который появляется при включении платы с нажатой кнопкой BOOTSEL.

А.10 Полезные ссылки

Названия сайтов приводятся без «рекомендаций магазинов»; адреса могут измениться, поэтому ищите по названию.

- `https://micropython.org` — сайт MicroPython, прошивки для Pico, Pico W и Pico 2;
- `https://docs.micropython.org` — документация MicroPython: модули `machine`, `network`, `time`;
- `https://circuitpython.org` — сайт CircuitPython, прошивки и список библиотек;
- `https://datasheets.raspberrypi.com` — официальные технические описания RP2040, RP2350, Pico и Pico W;
- `https://www.raspberrypi.com/documentation` — документация по микроконтроллерам Raspberry Pi;
- `https://learn.adafruit.com` — руководства Adafruit по CircuitPython и датчикам.

Приложение В. Детали и инструменты

Для курса не нужна дорогая лаборатория. Основа — плата Raspberry Pi Pico и «наборчик» дешёвых деталей. В приложении собраны три комплекта: чем больше уроков вы хотите пройти, тем шире комплект. Цены и магазины сознательно не указаны: они меняются, а детали стандартные и продаются везде, где продают радиоэлектронику. Ищите по названиям из таблиц.

В.1 Какую плату выбрать

- **Pico** — самая простая, годится для уроков 1–12 и 20–23.
- **Pico W** — то же и Wi-Fi/Bluetooth; нужна для уроков 16–19 и итогового проекта (урок 25).
- **Pico 2 / Pico 2 W** — более быстрый и ёмкий микроконтроллер RP2350 (урок 24); версии с W получают те же сети.
- **Исполнение с распаянными штырями («H»).** Для макетной платы нужны штыри на плате. Если штырей нет, их нужно припаять; без пайки удобнее купить плату сразу с ними.

В.2 Каталог деталей

Картинки ниже — условные: настоящие детали выглядят разнообразнее, но общий облик такой.

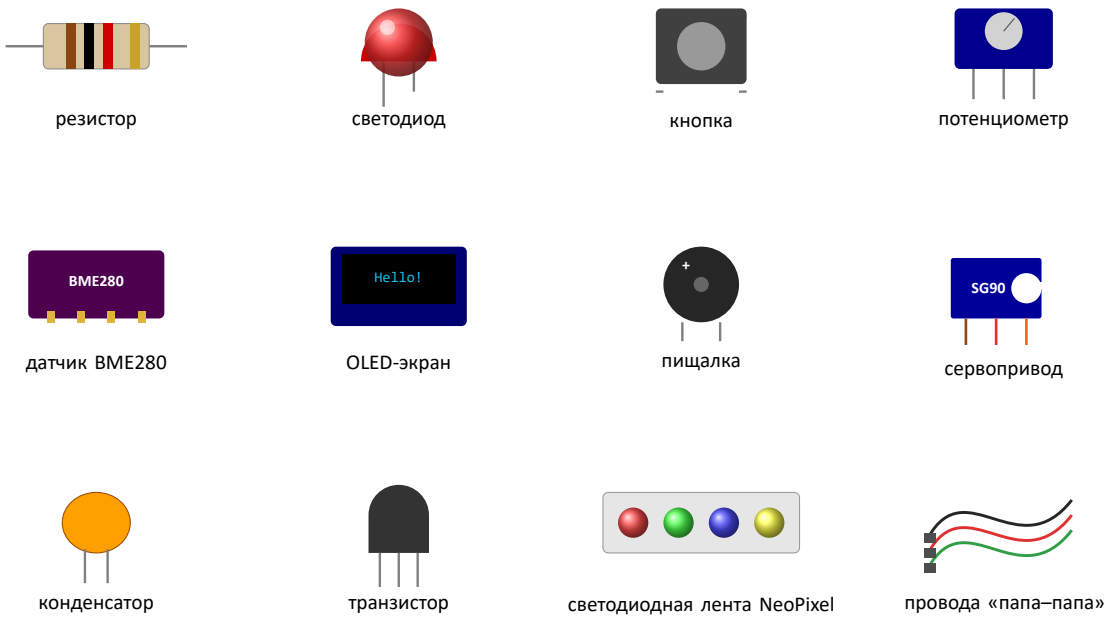


Рис. В.1: Каталог деталей курса (условные изображения)

В.3 Комплекты

Столбцы Мин., Станд., Расш. — количество деталей в минимальном, стандартном и расширенном комплекте. «Уроки» — где деталь используется.

Деталь	Мин.	Станд.	Расш.	Назначение	Уроки
Основа					
Pico, Pico W или Pico 2 (со штырями)	1	1	2	мозг проекта; вторая плата для сети и отладки	все

Деталь	Мин.	Станд.	Расш.	Назначение	Уроки
Кабель micro-USB (для данных) / USB-C для Pico 2	1	1	2	питание и программирование	1
Макетная плата 400 точек	1	1	2	сборка без пайки	2–25
Провода «папа–папа»	20	40	65	соединения	все
Провода «папа–мама»	—	10	20	подключение модулей с штырями	10
Светодиоды, кнопки, резисторы					
Светодиоды разных цветов	5	10	20	индикация, бегущий огонь	2, 5
Резисторы 220 Ω / 330 Ω	10	20	40	ограничение тока светодиодов	2
Резисторы 1 kΩ, 10 kΩ	5	10	20	подтяжки, делители, базы транзисторов	3, 6
Резисторы 4,7 kΩ	2	4	6	подтяжки I2C (если нет на модулях)	8
Тактовые кнопки	2	4	8	ввод, дребезг, автоматы	3, 4
Аналоговые детали					
Потенциометр 10 kΩ	1	2	3	АЦП, регулировка яркости	6
Фоторезистор	—	1	3	измерение освещённости	6
Термистор NTC 10 kΩ	—	1	2	температура через делитель	6
Конденсаторы 100 nF (керамические)	2	5	10	развязка питания, RC-цепочки	6, 15
Конденсатор 100 µF (электролитический)	—	1	2	сглаживание питания моторов и сервоприводов	20
Датчики и модули					
Модуль BME280 (T, влажность, давление), I2C	—	1	2	метеостанция	10, 25
Датчик температуры DS18B20	—	1	2	1-Wire, влагозащищённый вариант	10
Ультразвуковой дальномер HC-SR04	—	1	2	расстояние, парковочный датчик	10
OLED-экран 0,96" SSD1306 (I2C)	1	1	2	вывод информации	8, 25
Адресные светодиоды WS2812 (лента, кольцо)	—	1	2	NeoPixel, PIO	11, 14
Модуль часов реального времени DS3231	—	—	1	время при отключённом питании	8
Модуль карты памяти microSD (SPI)	—	—	1	хранение больших данных	9, 12
Адаптер USB-UART (3,3 В)	—	1	2	связь с компьютером и GPS	7
Звук и движение					
Пьезопищалка (пассивная)	1	2	3	звук, мелодии	21
Сервопривод SG90	—	1	3	угол поворота ШИМом	5, 20
Моторчик постоянного тока + драйвер (например, L298N, TB6612)	—	1	2	управление скоростью и направлением	20
Шаговый двигатель 28BYJ-48 с драйвером ULN2003	—	—	1	шаговое управление, PIO	20, 14
Транзисторы NPN (например, 2N2222, BC337) и диоды 1N4148/1N4007	—	3	6	ключ для нагрузки, защита от выбросов	20
Реле-модуль	—	—	1	управление сетевыми приборами (только с взрослыми)	20
Питание					
Батарейный отсек 3×AA/AAA или литий-ионный аккумулятор с модулем зарядки	—	—	1	автономная работа	15
Повышающий/понижающий преобразователь	—	—	1	питание нагрузки от аккумулятора	15

Таблица B.2: Три комплекта деталей: минимальный (для уроков 1–6), стандартный (для всего курса) и расширенный (с запасом и дополнительными проектами)

Совет

Резисторы, светодиоды, провода и кнопки покупайте *с запасом*: это самые «расходные» детали. Наборы «стартовый комплект для Raspberry Pi Pico» часто уже содержат большую часть списка; сверьтесь со списком из этого приложения.

B.4 Инструменты

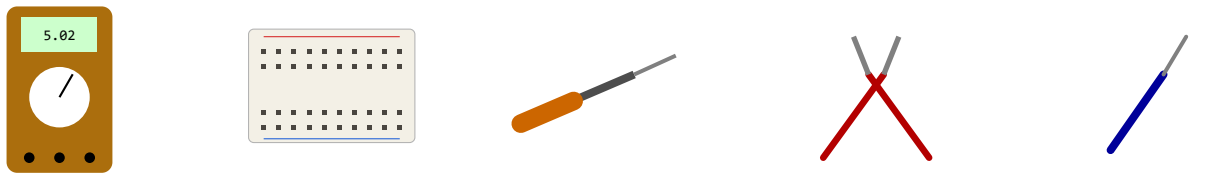


Рис. B.2: Инструменты для сборки и проверки схем

Инструмент	Зачем нужен	Комплект
Мультиметр	измерить напряжение, сопротивление, прозвонить цепь и найти обрыв; главный прибор для отладки схем	мин.
Макетная плата и провода	сборка без пайки	мин.
Компьютер с USB-портом	программирование (Thonny, mpremote)	мин.
Паяльник (25–40 Вт), припой с канифолью, подставка	припаять штыри, провода и модули	станд.
Бокорезы и стриппер	обрезка выводов и зачистка проводов	станд.
Пинцет	мелкие детали и SMD-компоненты	станд.
«Третья рука» или тиски	держать плату при пайке	расш.
Лабораторный блок питания	питание моторов и ламп отдельно от платы	расш.
Осциллограф или логический анализатор (например, на базе Pico)	посмотреть сигналы I2C, SPI, UART и ШИМ	расш.
Защитные очки	при пайке и обрезке выводов	станд.

Таблица B.3: Инструменты

Осторожно

Паяльник нагревается примерно до 350 °C. Работайте на открытом пространстве с хорошей вентиляцией, подставляйте паяльник только в подставку, не дотрагивайтесь до жала и не паяйте включённую плату. Школьникам — только вместе со взрослым.

B.5 Советы по покупке

1. **Проверьте напряжение.** Датчики и экраны для платы Pico должны работать от $3{,}3\,\text{V}$ (или быть совместимы с ним). Модуль, которому нужно $5\,\text{V}$ на сигнальных линиях, без преобразователя уровней подключать нельзя.

2. **Смотрите на интерфейс.** Для курса удобнее всего модули с I2C (всего четыре провода) и SPI.
3. **Адрес I2C.** У большинства датчиков адрес можно изменить перемычкой; если на шине два одинаковых датчика, это пригодится.
4. **Штыри.** Если модуль приходит с отдельными штырями, их нужно припаять; если пайки нет, ищите модули «с распаянными штырями».
5. **Кабель.** Многие дешёвые кабели micro-USB умеют только заряжать и не передают данные: плата с таким кабелем не будет видна в компьютере (урок 1).
6. **Не гонитесь за самым дешёвым.** Для датчиков неизвестных производителей могут встретиться поддельные микросхемы (например, «BMP280» под видом «BME280»: у них нет датчика влажности). Проверка простая: запустите сканер I2C и прочитайте идентификатор микросхемы.
7. **Покупайте наборами.** Стартовый набор обходится дешевле по штуке, чем отдельные детали, но проверьте, что в нём есть всё из вашего комплекта.
8. **Запасайтесь.** Светодиоды, резисторы и провода расходуются быстро; купите с запасом.
9. **Безопасность.** Работайте только с напряжением до 5 V; сетевое напряжение 220 V в этом курсе *не используется*.

B.6 Организация рабочего места

Заведите коробку с отделениями для деталей: резисторы сортируйте по номиналу и подписывайте, светодиоды — по цвету. Макетную плату держите на листе картона или на подставке, чтобы провода не цеплялись. Рядом с компьютером положите распечатку распиновки (приложение A) — она пригодится чаще всего.

Приложение C. Словарь терминов

Короткие определения терминов курса. Сначала идут латинские обозначения и названия по алфавиту, затем русские термины. Если слова нет в словаре, поищите его в оглавлении или в предметном тексте урока.

C.1 Латинские обозначения и названия

3V3 — вывод платы Pico, на котором доступно напряжение 3,3 V для питания датчиков и модулей; выход встроенного стабилизатора.

ADC — аналого-цифровой преобразователь: превращает напряжение на входе в число. У Pico 12 бит (0–4095), опорное напряжение 3,3 В, входы GP26–GP28.

AP (Access Point) — точка доступа Wi-Fi: устройство, к которому подключаются другие. Pico W может сама работать точкой доступа.

ASCII — таблица кодов символов, в которой коды 32–126 — печатные знаки (латиница, цифры, знаки препинания).

BLE (Bluetooth Low Energy) — «экономный» Bluetooth для датчиков и браслетов; обменивается небольшими порциями данных. Доступен на Pico W.

BME280 — популярный датчик температуры, влажности и давления с интерфейсом I2C или SPI.

BOOTSEL — кнопка на плате Pico; если держать её при подключении USB, плата превращается в диск для записи прошивки.

boot.py — файл, который запускается один раз при включении до `main.py` (в CircuitPython — до `code.py`); в нём настраивают режим файловой системы.

CDC — класс USB «виртуальный последовательный порт»: через него компьютер видит REPL платы как COM-порт.

circup — программа для установки и обновления библиотек CircuitPython на плате одной командой.

CircuitPython — версия Python для

микроконтроллеров от Adafruit; плата показывается как USB-диск `CIRCUITPY`, программа лежит в `code.py`.

code.py — главный файл программы в CircuitPython; запускается автоматически после включения платы.

CPOL / CPHA — два параметра режима SPI: полярность (CPOL) и фаза (CPHA) тактового сигнала. Вместе дают четыре режима (0–3); ведущий и ведомый должны использовать один и тот же.

CS (chip select) — линия выбора устройства в SPI; ведущий опускает её в 0, чтобы «обратиться» к нужному ведомому. Обозначается также CSn или SS.

DHCP — служба роутера, автоматически выдающая устройствам IP-адреса.

DMA — прямой доступ к памяти: аппаратура сама переносит данные между памятью и периферией, не отвлекая процессор.

DNS — «телефонная книга интернета»: превращает имя сайта в IP-адрес.

duty cycle — см. «скважность».

FIFO — очередь «первым вошёл — первым вышел»; так устроены буферы UART и каналы PIO.

flash — энергонезависимая память платы, в которой лежат прошивка и файлы; выдерживает ограниченное число циклов записи.

GATT — набор правил BLE: устройство описывается службами (services) и характеристиками (characteristics), которые можно читать и писать.

GND — «земля», общий провод, от которого отсчитываются все напряжения.

GP, GPIO — General Purpose Input/Output — вывод общего назначения, который можно сделать входом или выходом; в программе его называют номером GP (например, GP15).

HID — Human Interface Device — класс USB-устройств ввода: клавиатуры, мыши, геймпады. Pico может притвориться клавиатурой.

HTTP — протокол, по которому браузер получает веб-страницы; работает поверх TCP.

I2C — двухпроводная шина (SDA — данные, SCL — такт): к ней подключают много датчиков и экранов, у каждого свой адрес.

IP-адрес — числовой адрес устройства в сети, например `192.168.1.57`.

IRQ — запрос прерывания (Interrupt ReQuest): сигнал, заставляющий процессор на мгновение отложить текущую работу и выполнить обработчик.

JSON — текстовый формат обмена данными: `{"temp": 22.4, "hum": 41}`. Легко читается и людьми, и программами.

LDO — линейный стабилизатор напряжения; на Pico превращает 5 В USB в 3,3 В.

LED — светодиод.

LittleFS — файловая система MicroPython на Pico: устойчива к внезапному отключению питания и равномерно изнашивает flash.

main.py — главный файл программы в MicroPython; запускается автоматически после включения платы.

mip — менеджер пакетов MicroPython: `mip.install("имя")` или `mpremote mip install имя`.

MicroPython — компактная версия Python для микроконтроллеров. Урезана по сравнению с обычным Python, но знакома по синтаксису.

MISO / MOSI — линии SPI: «ведущий принимает» и «ведущий отправляет». Иначе называются RX и TX.

mpremote — программа для компьютера: копирует файлы на плату, запускает программы, открывает REPL, ставит пакеты.

MQTT — лёгкий протокол публикации и подписки для «интернета вещей»: устройства отправляют сообщения в темы брокера, а подписчики их получают.

NeoPixel — название адресных светодиодов WS2812 в экосистеме Adafruit.

NTP — протокол синхронизации времени по сети: Pico W может узнать точное время у сервера.

Pico W / Pico 2 W — версии платы Pico с радиочипом Wi-Fi и Bluetooth.

picotool — программа для компьютера, умеющая смотреть содержимое платы, записывать и сохранять прошивку.

PIO — Programmable I/O: небольшие аппаратные «процессоры» внутри RP2040/RP2350, выполняющие свои короткие программы и формирующие любые цифровые протоколы.

PWM — широтно-импульсная модуляция: быстрое включение и выключение выхода; доля времени «включено» (скважность) задаёт среднее напряжение.

QoS — «качество обслуживания» в MQTT: 0 — отправить один раз, 1 — хотя бы один раз, 2 — ровно один раз.

REPL — Read–Eval–Print Loop: интерактивная командная строка Python на плате; вы вводите команду и сразу видите результат.

retain — флаг MQTT-сообщения: брокер хранит последнее такое сообщение и выдаёт его новым подписчикам.

RP2040 — двухъядерный микроконтроллер Raspberry Pi на ядрах Cortex-M0+, 125 МГц, 264 КБ ОЗУ; сердце Pico и Pico W.

RP2350 — следующее поколение: двухъядерный микроконтроллер на Cortex-M33 (или Hazard3) с 520 КБ ОЗУ; стоит на Pico 2.

RTC — часы реального времени: хранят время и дату; на Pico ими можно пользоваться после установки времени.

RX / TX — «приём» и «передача»; линия TX одного устройства подключается к RX другого.

SCL / SDA — линии такта и данных шины I2C.

SCK — тактовая линия SPI.

SPI — быстрая четырёхпроводная шина (SCK, MOSI, MISO, CS) для экранов, карт памяти, радиомодулей.

SSD1306 — популярный контроллер OLED-экранов 128×64 или 128×32.

SSID — имя сети Wi-Fi.

SWD — Serial Wire Debug: двухпроводной интерфейс отладки; на плате Pico выведен тремя контактами внизу.

TCP / UDP — протоколы передачи данных в сети: TCP надёжный (с подтверждениями), UDP быстрый, но без гарантий.

Thonny — простая среда разработки на Python, подходящая для MicroPython и CircuitPython.

TLS — шифрование соединений (то, что скрывается за `https`).

UART — асинхронный последовательный интерфейс (две линии TX и RX); применяется для связи с GPS, Bluetooth-модулями и компьютером.

UF2 — формат файла прошивки; его достаточно скопировать на диск `RPI-RP2`.

USB — универсальная последовательная шина; через неё плата получает питание, показывается как диск или COM-порт.

VBUS — вывод платы, на котором присутствует напряжение 5 В от USB.

VSYS — системное питание платы: сюда подаётся питание от батареи или внешнего источника.

watchdog (WDT) — сторожевой таймер: перезагружает плату, если программа вовремя не «покормила» его командой `feed()`.

WLAN — беспроводная локальная сеть; в MicroPython так называется класс `network.WLAN`.

WS2812 — адресный светодиод с встроенным контроллером; цвет передаётся по одному проводу особыми импульсами.

C.2 Русские термины

Автомат (конечный) — модель, при которой программа в каждый момент находится в одном из состояний, а события переводят её из состояния в состояние.

Анод — положительный вывод светодиода или диода (у светодиода — более длинная ножка).

Аналоговый сигнал — сигнал, который может принимать любое значение в диапазоне, например напряжение датчика.

Бод — единица скорости передачи: символов (изменений сигнала) в секунду; для UART примерно равна числу бит в секунду.

Библиотека — набор готовых функций и классов, который подключают командой `import`.

Брокер — сервер MQTT, принимающий сообщения от публикаторов и раздающий их подписчикам (например, Mosquitto).

Бит, байт — бит — 0 или 1; байт — 8 бит, число от 0 до 255.

Блокирующий вызов — функция, которая останавливает программу до завершения: например, `sleep()` или ожидание ответа сервера.

Датчик — устройство, превращающее физическую величину (температуру, свет, расстояние) в электрический сигнал.

Делитель напряжения — два резистора, включённые последовательно; со средней точки снимают часть напряжения.

Диод — деталь, пропускающая ток только в одну сторону.

Дребезг — многократные быстрые замыкания и размыкания контактов кнопки при нажатии; программа должна его отфильтровать.

Драйвер — программа (или микросхема), которая управляет устройством; например, драйвер экрана или драйвер мотора.

Железо (hardware) — физическая часть: плата, провода, датчики — в противоположность программам.

Исключение — ошибка времени выполнения; её можно перехватить конструкцией `try/except`.

Катод — отрицательный вывод светодиода или диода (короткая ножка).

Конденсатор — деталь, накапливающая заряд; сглаживает пульсации питания и участвует в RC-цепочках.

Кортеж — неизменяемая последовательность значений: `(1, 2, 3)`.

Куча (heap) — область оперативной памяти, из которой программа берёт память под объекты.

Макетная плата — плата с отверстиями для сборки схем без пайки.

Мультиметр — прибор для измерения напряжения, тока и сопротивления.

Напряжение — «давление», толкающее ток по цепи; измеряется в вольтах.

Неблокирующий — способ писать программы без ожидания: время сравнивают с моментом последнего действия.

Опрос (polling) — периодическая проверка состояния (например, кнопки) в главном цикле, в отличие от прерывания.

Отладка — поиск и исправление ошибок в программе или схеме.

Периферия — устройства, встроенные в микроконтроллер: таймеры, UART, SPI, I2C, АЦП.

Подтяжка (pull-up / pull-down) — резистор, который соединяет вход с питанием (pull-up) или с землёй (pull-down), чтобы вход не «висел в воздухе».

Прерывание — механизм, при котором событие (нажатие кнопки, окончание передачи) вызывает обработчик немедленно.

Прошивка — программа, записанная во flash-память микроконтроллера, и сам процесс записи.

Протокол — договорённость о формате и порядке обмена данными между устройствами.

Резистор — деталь, ограничивающая ток; сопротивление измеряют в омах.

Сборщик мусора (GC) — механизм MicroPython, освобождающий память, занятую ненужными объектами; запускается командой `gc.collect()`.

Сервопривод — моторчик, который по ШИМ-команде поворачивается в заданный угол.

Скважность — отношение длительности импульса к периоду ШИМ; от 0 до 100 %.

Словарь — набор пар «ключ — значение»: `{"temp": 22}`.

Сокет (socket) — «розетка» для сетевого обмена: через неё программа отправляет и принимает данные по TCP или UDP.

Сопротивление — мера сопротивления току; измеряется в омах (Ω).

Список — изменяемая последовательность значений: `[1, 2, 3]`.

Таймер — аппаратный счётчик времени; умеет вызывать функцию через заданные промежутки.

Ток — количество заряда, проходящего через цепь за секунду; измеряется в амперах.

Транзистор — электронный ключ или усилитель: малым током управляет большим; позволяет включать моторы и лампы с вывода GPIO.

Частота — число повторов в секунду; единица — герц (Гц).

Шина — набор проводов, по которым несколько устройств обмениваются данными (I2C, SPI).

Шаговый двигатель — мотор, который поворачивается на строго определённый угол за каждый импульс.

Светодиод — диод, излучающий свет; ограничивается резистором.

Интерпретатор — программа, выполняющая код Python построчно; на плате — это сама прошивка MicroPython.

Модуль — файл `.py` с функциями и классами, подключаемый командой `import`.

Пин (вывод) — контакт платы или микросхемы.

Логический уровень — «0» (около 0 В) и «1» (около 3,3 В) на цифровом выводе.

Эхо-запрос (ping) — проверка доступности устройства в сети коротким пакетом.

Ключ (транзисторный) — схема, где транзистор работает в режимах «включён/выключен», коммутируя нагрузку.

Тема (MQTT topic) — «адрес» сообщения в MQTT, например `pico/weather/state`.

Подписчик, публикатор — участники MQTT: публикатор отправляет сообщения в тему, подписчик их принимает.

Сторожевая схема — см. «watchdog».

Кольцевой буфер — массив фиксированного размера, в который данные пишутся по кругу, затирая самые старые.

Передискретизация — усреднение нескольких измерений для снижения шума; например, в BME280.

Калибровка — подгонка показаний датчика под эталон с помощью поправок или коэффициентов.

Эмуляция клавиатуры — режим, при котором Pico вводит символы как настоящая USB-клавиатура.

Приложение D. Python за один вечер

Это приложение для тех, кто никогда не программировал. Его можно прочитать за один вечер с платой в руках: вводите примеры в REPL (урок 1) и смотрите результат. Python — язык, в котором программа читается почти как английский текст, поэтому его выбрали для школ и для микроконтроллеров. Примеры здесь написаны на обычном Python; в конце сказано, чем **MicroPython** и **CircuitPython** от него отличаются.

D.1 Как запускать программы

- **REPL** — командная строка с приглашением `>>>`. Набираете строку, нажимаете Enter и сразу видите ответ. Удобно для опытов: `>>> 2 + 2` даёт `4`.
- **Файл** с расширением `.py`. Программа выполняется сверху вниз, строка за строкой. На плате **MicroPython** файл `main.py` запускается сам, в **CircuitPython** — `code.py`.

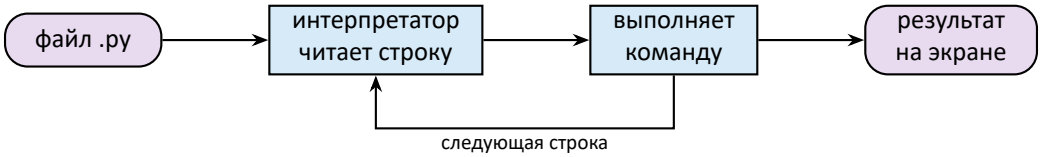


Рис. D.1: Интерпретатор выполняет программу по одной строке сверху вниз

Всё после знака `#` в строке — **комментарий**, записка для человека; компьютер её игнорирует. Отступы (четыре пробела) в Python *обязательны*: они показывают, какие строки входят в условие, цикл или функцию.

D.2 Переменные и типы

Переменная — это имя, под которым хранится значение. Представьте коробку с наклейкой: наклейка — имя, внутри — значение. Команда `age = 14` клеит наклейку `age` на коробку с числом 14.

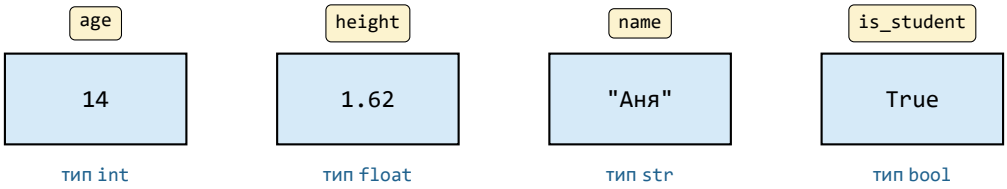


Рис. D.2: Переменные: имя (наклейка), значение (содержимое коробки) и тип значения

Тип	Что хранит	Примеры	Пример переменной
`int`	целые числа	0, 42, −7	`count = 3`
`float`	числа с точкой (в записи — точка!)	3.14, −0.5	`temp = 22.4`
`str`	текст	`"Привет"`, `'a'`	`name = "Аня"`
`bool`	«да» или «нет»	`True`, `False`	`led_on = True`
`list`	изменяемый список	`[1, 2, 3]`	`temps = []`
`tuple`	неизменяемый набор	`(3, 4)`	`pos = (3, 4)`
`dict`	словарь «ключ—значение»	`{"a": 1}`	`cfg = {}`
`None`	«ничего»	`None`	`result = None`

Таблица D.1: Основные типы данных Python

Листинг D.1: Python: переменные и арифметика

```
# Переменные и типы
age = 14                  # целое число (int)
height = 1.62             # число с точкой (float)
name = "Аня"              # строка (str)
is_student = True         # логическое значение (bool)

print(type(age), type(height), type(name), type(is_student))

# переменную можно менять, и её тип может измениться
x = 10
x = x + 5                 # теперь x равно 15
x = "десять"              # теперь x --- строка (так делать не стоит!)

# арифметика
a, b = 17, 5
print(a + b)              # 22  сложение
print(a - b)              # 12  вычитание
print(a * b)              # 85  умножение
print(a / b)              # 3.4 деление всегда даёт float
print(a // b)             # 3   целая часть от деления
print(a % b)              # 2   остаток от деления
print(a ** 2)             # 289 степень
print(round(3.14159, 2)) # 3.14
```

Имена переменных пишут латиницей, маленькими буквами, слова соединяют знаком подчёркивания: `sensor_value`. Имя должно объяснять, что внутри. Знак = в Python — не «равно», а «присвоить»: справа вычисляется значение, слева — имя, под которым его запоминают.

Осторожно

Деление `/` всегда даёт число с точкой, даже если делится без остатка (`10 / 2` равно `5.0`). Для целой части используйте `//`, для остатка — `%`. Остаток от деления пригодится для «закольцовывания» (например, `(mode + 1) % 4` возвращает 3 обратно в 0).

D.3 Строки, форматирование, ввод и вывод

Строка — текст в кавычках (одинарных или двойных). Вывод на экран делает `print()`, ввод с клавиатуры — `input()`. Чтобы вставить значения в текст, используют **f-строки**: перед открывающей кавычкой ставят букву `f`, а переменные пишут в фигурных скобках.

Листинг D.2: Python: строки, форматирование, ввод

```
# Строки, форматирование, ввод и вывод
name = "Аня"
temp = 22.456

print("Привет, " + name)                  # склеивание строк
print(f"{name}: {temp:.1f} градуса")      # f-строка: Аня: 22.5 градуса
print("{}: {:.1f}".format(name, temp))    # то же через format
print("%s: %.1f" % (name, temp))          # старый стиль (используется в MicroPython)

s = "метеостанция"
print(len(s))             # 12 --- длина
print(s[0], s[-1])        # м я --- первый и последний символ
```

```
print(s[0:4])            # мете --- срез
print(s.upper())         # МЕТЕОСТАНЦИЯ
print("ст" in s)         # True

# ввод с клавиатуры (в REPL)
answer = input("Сколько тебе лет? ")     # input всегда возвращает строку!
age = int(answer)                        # превращаем в число
print("Через год тебе будет", age + 1)
```

Формат	Значение	Пример (для 22.456)
`{x:.1f}`	число с одним знаком после точки	`22.5`
`{x:6.2f}`	ширина 6, два знака	`22.46`
`{n:03d}`	целое с нулями слева (для `n = 7`)	`007`
`{n:x}`	в шестнадцатеричном виде (для `n = 255`)	`ff`
`{n:08b}`	в двоичном виде (для `n = 5`)	`00000101`

Таблица D.2: Часто используемые форматы в f-строках и `format()`

D.4 Условия

Условный оператор `if` выбирает, какой кусок программы выполнить. Условие — выражение, которое даёт `True` или `False`. Операции сравнения: `==` (равно), `!=` (не равно), `<`, `>`, `<=`, `>=`. Объединяют условия словами `and`, `or`, `not`.

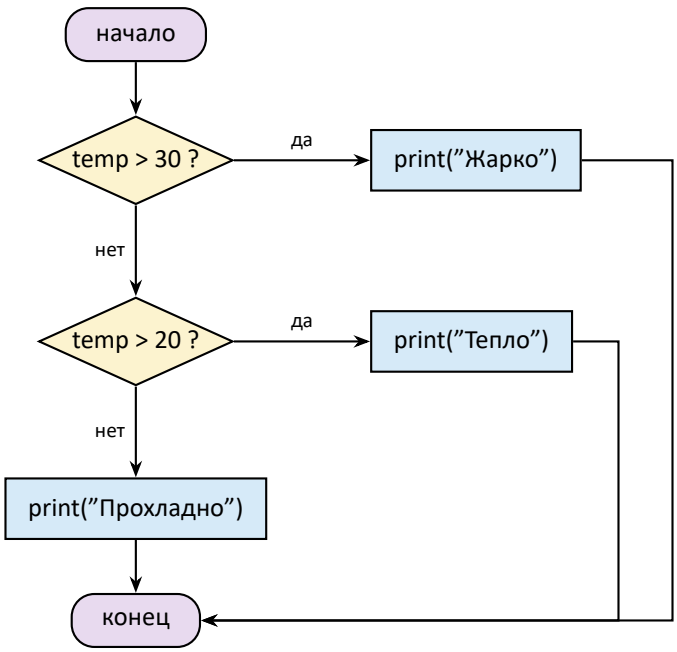


Рис. D.3: Блок-схема цепочки `if / elif / else`: выполняется только одна ветка

Листинг D.3: Python: условия

```
# Условия
temp = 27

if temp > 30:
    print("Жарко")
elif temp > 20:
    print("Тепло")        # выполнится эта ветка
else:
    print("Прохладно")

# сравнения: == != < > <= >=; логика: and, or, not
if 20 <= temp <= 30 and not temp == 25:
    print("Комфортно")

# условное выражение в одну строку
state = "включить" if temp > 25 else "выключить"
print(state)
```

D.5 Циклы

Цикл повторяет действия. Цикл `for` перебирает элементы (например, числа из `range`), а цикл `while` повторяется, пока условие истинно. `range(5)` даёт числа 0, 1, 2, 3, 4 (правая граница не включается!).

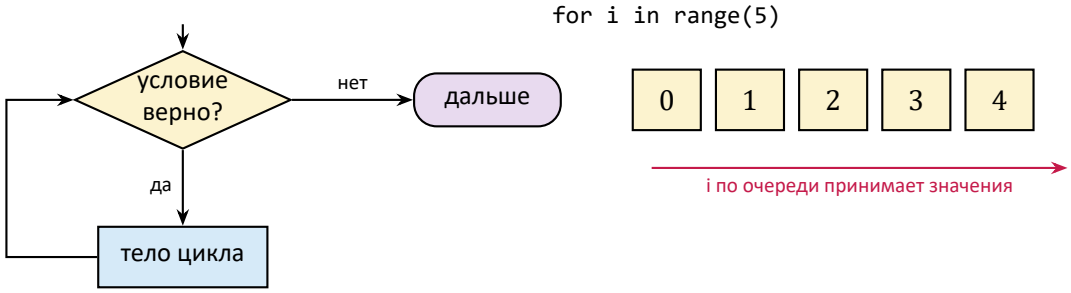


Рис. D.4: Слева: схема цикла `while`. Справа: цикл `for` перебирает числа из `range(5)`

Листинг D.4: Python: циклы

```
# Циклы
for i in range(5):                # 0, 1, 2, 3, 4
    print("шаг", i)

total = 0
for n in range(1, 11):            # сумма чисел от 1 до 10
    total += n
print(total)                      # 55

count = 3
while count > 0:                  # пока условие истинно
    print("осталось", count)
    count -= 1

# break выходит из цикла, continue переходит к следующему шагу
for n in range(10):
```

```
    if n % 2 == 0:
        continue                # пропускаем чётные
    if n > 7:
        break                   # заканчиваем цикл
    print(n)                    # 1 3 5 7
```

Осторожно

Если в цикле `while` условие никогда не станет ложным, цикл будет вращаться вечно. В программах для плат это нормально (`while True`), но на компьютере остановить программу можно клавишами Ctrl+C.

D.6 Списки, кортежи и словари

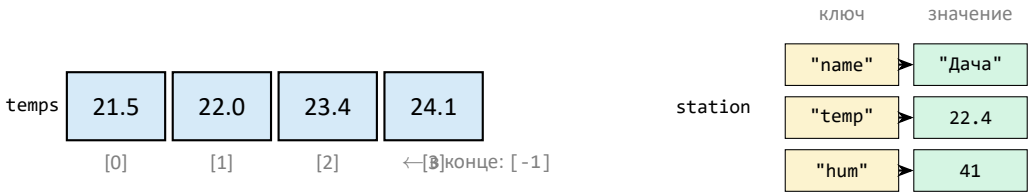


Рис. D.5: Список хранит значения по номерам (с нуля), словарь — по ключам

Листинг D.5: Python: списки, кортежи, словари

```
# Списки, кортежи, словари
temps = [21.5, 22.0, 23.4]      # список: изменяемый
temps.append(24.1)              # добавить в конец
print(temps[0], temps[-1], len(temps))
for t in temps:
    print(t)
print(sum(temps) / len(temps))  # среднее
squares = [n * n for n in range(5)]   # [0, 1, 4, 9, 16] --- списковое включение

point = (3, 4)                  # кортеж: неизменяемый
x, y = point                    # распаковка

station = {"name": "Дача", "temp": 22.4}   # словарь: ключ -> значение
station["hum"] = 41             # добавить пару
print(station["name"])
print(station.get("press", "нет данных"))  # безопасное чтение
for key, value in station.items():
    print(key, "=", value)
```

	Список	Кортеж	Словарь
Запись	`[1, 2, 3]`	`(1, 2, 3)`	`{"a": 1}`
Менять элементы	да	нет	да
Доступ	по номеру `s[0]`	по номеру `t[0]`	по ключу `d["a"]`
Когда нужен	набор однотипных значений	пара координат, «запись»	настройки, данные датчика

Таблица D.3: Сравнение коллекций

D.7 Функции

Функция — это «чёрный ящик» с именем: на вход получает данные (аргументы), на выходе выдаёт результат. Один раз описали — вызываем сколько угодно. Функции делают программу короче и понятнее.

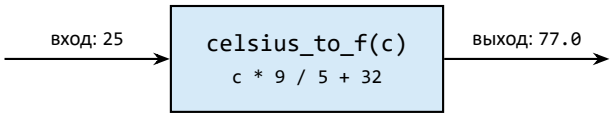


Рис. D.6: Функция как чёрный ящик: на вход аргументы, на выходе результат (`return`)

Листинг D.6: Python: функции и модули

```python
# Функции и модули
def celsius_to_f(c):
    """Переводит градусы Цельсия в Фаренгейта."""
    return c * 9 / 5 + 32


def clamp(value, low=0, high=100):      # значения по умолчанию
    if value < low:
        return low
    if value > high:
        return high
    return value


print(celsius_to_f(25))                 # 77.0
print(clamp(120))                       # 100
print(clamp(-5, low=-10))               # -5

# модули
import time
from math import sqrt
print(sqrt(16))
time.sleep(0.1)
```

Модуль — файл `.py` с функциями; его подключают командой `import`. Форма `from модуль import имя` берёт из модуля только нужное. Собственные модули (например, `config.py`) подключаются так же, как стандартные.

D.8 Исключения

Ошибка во время работы (деление на ноль, неверное число) вызывает **исключение**, и программа аварийно останавливается. Конструкция `try/except` позволяет перехватить ошибку и продолжить работу.

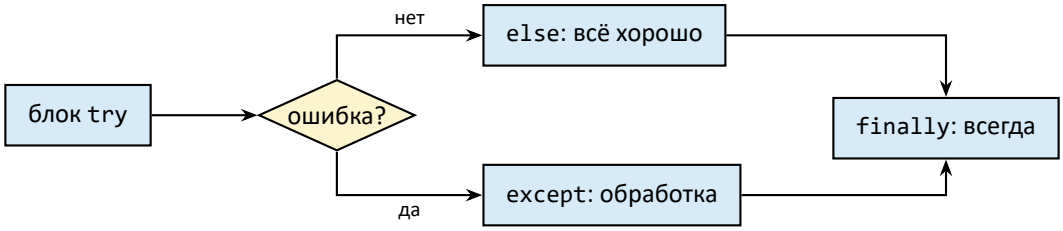


Рис. D.7: Порядок выполнения `try / except / else / finally`

Листинг D.7: Python: исключения

```
# Исключения
try:
    n = int(input("Число: "))
    print(10 / n)
except ValueError:                  # ввели не число
    print("Это не число")
except ZeroDivisionError:           # деление на ноль
    print("На ноль делить нельзя")
else:
    print("Всё хорошо")             # выполнится, если ошибок не было
finally:
    print("Конец")                  # выполнится всегда
```

Правило: перехватывайте конкретные ошибки (`except ValueError`), а не все подряд: иначе вы спрячете настоящие баги. В программах для плат так ловят сетевые ошибки (`OSError`) и продолжают работу.

D.9 Классы (основы)

Класс — чертёж, по которому делают **объекты**. Класс описывает, какие данные (атрибуты) и какие действия (методы) есть у объекта. Когда вы пишете `Pin(15, Pin.OUT)`, вы создаёте объект класса `Pin`.

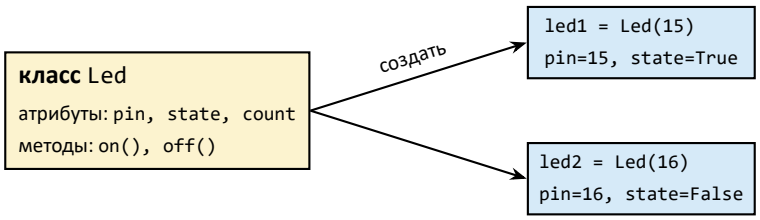


Рис. D.8: Один класс — много объектов с одинаковым набором методов, но своими значениями атрибутов

Листинг D.8: Python: класс

```
# Классы
class Led:
    """Светодиод со счётчиком включений."""

    def __init__(self, pin):        # конструктор
        self.pin = pin
        self.state = False
        self.count = 0

    def on(self):
        self.state = True
        self.count += 1

    def off(self):
        self.state = False

    def __str__(self):
        return "Led(pin=%d, state=%s)" % (self.pin, self.state)
```

```
19
20
21 led = Led(15)                      # создаём объект
22 led.on()
23 led.on()
24 print(led, led.count)              # Led(pin=15, state=True) 2
```

Слово `self` внутри класса означает «сам этот объект»; метод `__init__` вызывается при создании объекта.

D.10 Чем MicroPython отличается от обычного Python

MicroPython написан так, чтобы работать на микроконтроллере с 264 КБ памяти. Синтаксис тот же, но среда другая.

	Обычный Python	**MicroPython**
Стандартная библиотека	огромная	урезанная: нужные модули (`os`, `json`, `time`, `struct`) есть, многих нет
Установка пакетов	`pip install`	`mip` (`mpremote mip install имя`); пакетов меньше
Память	гигабайты	сотни килобайт; следите за `gc.mem_free()`
Целые числа	любые по размеру	тоже любые, но большие замедляют и занимают память
Числа с точкой	64-битные	на Pico — 32-битные (около 7 значащих цифр)
Многопоточность	`threading`	нет; есть `asyncio` и второе ядро (`_thread`)
Специальные модули	—	`machine` (выводы и шины), `network`, `rp2` (PIO)
Время	`time.time()` (секунды)	`ticks_ms()`, `ticks_diff()`
Запуск	по команде	`boot.py` и `main.py` стартуют сами
Ошибка памяти	редко	`MemoryError`; лечится меньшими буферами и `gc.collect()`

Таблица D.4: **MicroPython** и обычный Python

Листинг D.9: **MicroPython**: особенности

```
1  # Что нужно помнить в MicroPython
2  import gc
3  import time
4
5  # 1. нет pip: пакеты ставит mip (mpremote mip install имя)
6  # 2. память мала: следим за ней
7  print(gc.mem_free())
8  gc.collect()                      # освободить мусор
9
10 # 3. время в миллисекундах и безопасная разность
11 t0 = time.ticks_ms()
12 time.sleep_ms(100)
13 print(time.ticks_diff(time.ticks_ms(), t0))   # около 100
14
15 # 4. числа: int может быть очень большим, а float на Pico --- 32-битный (около 7 цифр)
16 print(2 ** 100)
```

```
print(0.1 + 0.2)                  # 0.3 (в обычном Python: 0.30000000000000004)

# 5. часть модулей урезана: не всё из обычной стандартной библиотеки есть
try:
    import json                   # json есть
    import threading              # а вот threading нет
except ImportError as e:
    print("нет модуля:", e)
```

D.11 Чем отличается CircuitPython

CircuitPython — ещё одна реализация Python для плат (на основе **MicroPython**), ориентированная на простоту для новичков.

- Плата подключается как USB-диск `CIRCUITPY`; программа — файл `code.py`, его достаточно сохранить, и плата сама перезапустится.
- Вместо `machine` используются модули `board`, `digitalio`, `analogio`, `busio`, `pwmio` (таблицы в приложении А).
- Библиотеки лежат в папке `lib` и ставятся через `circup`.
- Время — `time.monotonic()` (секунды), а не `ticks_ms()`.
- Секреты принято хранить в `settings.toml`.
- Нет прерываний `Pin.irq`; многозадачность — через `asyncio`.

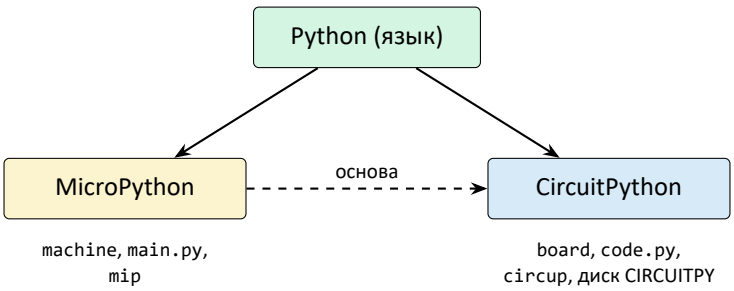


Рис. D.9: Один язык — две реализации для плат; программы похожи, но библиотеки для работы с «железом» разные

D.12 Упражнения

Выполните упражнения в REPL, а потом сверьтесь с ответами ниже.

1. Чему равны значения `17 // 5`, `17 % 5`, `2 ** 10`, `7 / 2`?
2. Напишите программу, которая просит ввести радиус круга и выводит площадь с двумя знаками после точки ($\pi \approx 3{,}14159$).
3. Выведите числа от 1 до 20, но кратные трём замените словом `"бум"`.
4. Дан список `t = [20.5, 22.0, 19.5, 23.5]`. Найдите минимум, максимум и среднее без использования `min`, `max`.
5. Напишите функцию `is_even(n)`, которая возвращает `True` для чётных чисел.
6. Что выведет программа?
   `d = {"a": 1, "b": 2}; d["c"] = d["a"] + d["b"]; print(len(d), d["c"])`
7. Опишите класс `Counter` с методом `inc()`, увеличивающим счётчик на единицу, и методом `reset()`.

Ответы.

1. `3`, `2`, `1024`, `3.5`.
2. `r = float(input("Радиус: "))`
   `print(f"Площадь: {3.14159 * r * r:.2f}")`
3. `for n in range(1, 21):`
   `print("бум" if n % 3 == 0 else n)`
4. Идея: взять первый элемент как начальное значение, потом в цикле сравнивать. `lo = hi = t[0]; total = 0`; `for x in t`: обновить `lo` и `hi`, добавить `x` к `total`; среднее — `total / len(t)`. Результат: 19.5, 23.5, 21.375.
5. `def is_even(n):`
   `return n % 2 == 0`
6. `3 3` (в словаре стало три пары, `d["c"]` равно 3).
7. `class Counter:`
   `def __init__(self): self.n = 0`
   `def inc(self): self.n += 1`
   `def reset(self): self.n = 0`

Итоги урока

- Переменная — имя значения; типы: `int`, `float`, `str`, `bool`, `list`, `tuple`, `dict`.
- Ветвления (`if/elif/else`) и циклы (`for`, `while`) управляют порядком выполнения; отступы обязательны.
- Функции и модули делают программу понятной; исключения перехватываются `try/except`.
- Класс — чертёж, объект — его экземпляр: `Pin(15, Pin.OUT)` создаёт объект класса `Pin`.
- **MicroPython** и **CircuitPython** — урезанный Python для плат со своими модулями и ограничениями памяти.

Приложение E. Ответы на вопросы для самопроверки

Ответы даны кратко: это подсказка, с чем сверить своё объяснение, а не образец для заучивания. Порядок ответов соответствует порядку вопросов в блоках «Проверь себя» каждого урока.

Урок 1. Первый запуск

1. Кнопка BOOTSEL переводит плату в режим загрузки: она появляется как USB-диск. Нужно зажать её и, не отпуская, подключить USB.
2. Прошивка — интерпретатор Python, заливается один раз; программа — ваш файл (`main.py` или `code.py`).
3. У Pico W есть радиочип и другое описание платы (светодиод, Wi-Fi); прошивка для Pico этого не содержит.
4. REPL — консоль «прочитал–выполнил–напечатал»: команды выполняются сразу, удобно пробовать идеи.
5. MicroPython запускает `main.py`, CircuitPython — `code.py`.

Урок 2. Цифровой выход

1. Резистор ограничивает ток: $R = (U_{\text{пит}} - U_F)/I$, например $(3{,}3 - 2)/0{,}006 \approx 220\,\Omega$.
2. В программе используется логическая нумерация GP; номер контакта на плате другой (GP13 — контакт 17).
3. Во время `sleep()` программа стоит и не может опрашивать кнопки и датчики.
4. Счётчик переполняется и начинается заново; обычное вычитание даёт неверный результат, а `ticks_diff()` это учитывает.
5. Через имя `"LED"`: `Pin("LED", Pin.OUT)` (в CircuitPython — `board.LED`).

Урок 3. Цифровой вход

1. Вход, не подключённый ни к 0, ни к 1: ловит наводки и читается случайно.
2. Pull-up подтягивает к 3,3 В: нажатая кнопка (замыкает на GND) читается как 0. Pull-down подтягивает к земле: нажатая кнопка читается как 1.
3. Контакты дребезжат несколько миллисекунд: на осциллограмме серия быстрых переходов 0–1 вместо одного.
4. Изменение принимается, только если новое значение держится стабильно заданное время (например, 20 мс).
5. Обработчик выполняется внутри прерывания и блокирует остальное; нельзя долго ждать, много печатать, выделять память — ставят флаг и обрабатывают его в основном цикле.

Урок 4. Время и конечные автоматы

1. `ticks_diff()` правильно работает при переполнении счётчика.
2. `PERIODIC` срабатывает регулярно, `ONE_SHOT` — один раз. В колбэке нельзя делать долгие операции, `sleep()`, много выделять памяти.
3. Так ошибки не накапливаются: расписание привязано к плану, а не к моменту фактического срабатывания.
4. Конечный автомат — модель с конечным набором состояний, событий и переходов (и действий при переходах).
5. Вызвать функцию много раз и разделить суммарное время на число вызовов; использовать `ticks_us()`.

Урок 5. ШИМ

1. Доля периода во включённом состоянии; $U_{\text{ср}} = D \cdot U$ (при $D = 0{,}5$ и 3,3 В получается около 1,65 В).
2. Глаз не замечает мерцаний выше 50–100 Гц.
3. У пары каналов одного слайса общая частота: например, GP0 и GP1 (а также GP16, GP17).
4. Глаз воспринимает яркость нелинейно: без гаммы середина шкалы кажется слишком яркой.
5. Импульсом 0,5–2,5 мс с периодом 20 мс; серво потребляет большой ток, 3V3 такой нагрузки не потянет — питание отдельное, земля общая.

Урок 6. АЦП

1. Дискретизация сигнала по уровню; ступенька $3{,}3/4096 \approx 0{,}8$ mV.
2. GP26, GP27, GP28 (каналы ADC0–ADC2); не более 3,3 В.
3. $U_{\text{вых}} = U_{\text{вх}} R_2/(R_1 + R_2)$. Для 9 В подойдёт, например, 20 кОм и 10 кОм (на выходе 3 В).
4. Медиана отбрасывает одиночные выбросы, усреднение их «размазывает»; медиана лучше при редких сильных помехах.
5. Два порога (гистерезис) не дают свету «дребезжать» около границы.

Урок 7. UART

1. Общего такта нет: стороны заранее договариваются о скорости, а старт-бит синхронизирует приём.
2. $0\text{x}41 = 01000001$; идёт старт-бит 0, затем биты от младшего: 1,0,0,0,0,0,1,0, затем стоп-бит 1.
3. $100 \cdot 10/9600 \approx 0{,}104$ s.
4. Передатчик должен идти к приёмнику (TX→RX); 5 В на входе Pico может её повредить.
5. Байты приходят порциями; строку собирают, пока не придёт символ конца строки.

Урок 8. I²C

1. Устройства только «прижимают» линию к нулю, а единицу создают резисторы: так нет короткого замыкания при спорах.
2. Старт: SDA падает при высоком SCL; бит: SDA меняется при низком SCL, читается при высоком; стоп: SDA растёт при высоком SCL.
3. Они ответят одновременно, данные будут испорчены: нужно сменить адрес, использовать вторую шину или мультиплексор.
4. $47 = 0100\,0111_{\text{BCD}} = 0\text{x}47$. Число `0x23` в обычной записи равно 35, в BCD — 23.
5. Рисование идёт в буфер в памяти; `show()` отправляет его на экран.

Урок 9. SPI

1. SPI: четыре провода и отдельный CS на каждое устройство; I²C: два провода и адреса.
2. Полярность тактового сигнала и фаза выборки данных; режим указан в даташите устройства.
3. Обмен идёт как сдвиг в двух регистрах; чтобы просто прочитать, отправляют фиктивные байты.
4. Защёлка переносит данные в выходы одновременно, поэтому они не мигают во время загрузки.
5. Файл может не дописаться, файловая система повредится; подходит FAT (чаще FAT32).

Урок 10. Датчики

1. Длительностью высокого уровня: короткий (около 26–28 мкс) — 0, длинный (около 70 мкс) — 1.
2. Шина с открытым стоком, нужна подтяжка (4,7 кОм); у каждого датчика свой 64-битный адрес.

3. $d = v\,t/2 = 343 \cdot 0{,}00466/2 \approx 0{,}8$ m.
4. ECHO выдаёт 5 В; делитель даёт $5 \cdot 2/3 \approx 3{,}3$ V.
5. Среднее сглаживает, но сдвигается от выброса; медиана выброс игнорирует и лучше при редких скачках.

Урок 11. WS2812

1. По длительности высокого импульса: короткий — 0, длинный — 1.
2. До 30×60 mA $= 1{,}8$ А; нужен блок питания 5 В не менее 2 А.
3. Конденсатор сглаживает всплеск при включении, резистор защищает вход данных.
4. В HSV цвет задаётся углом тона; радуга — плавное изменение H от 0 до 360.
5. Глаз нелинеен; гамму реализуют таблицей или возведением в степень.

Урок 12. Память и файлы

1. SRAM быстрая и теряет данные без питания; flash медленнее и сохраняет данные.
2. У ячеек flash ограниченное число стираний (порядка 10^5 на сектор); при записи раз в минуту в один сектор это около двух с половиной месяцев, но LittleFS распределяет износ.
3. Запись во временный файл и последующее переименование: старая версия не теряется при сбое.
4. Диском владеет компьютер; в `boot.py` вызывают `storage.remount("/", readonly=False)`.
5. Сборщик мусора освобождает неиспользуемые объекты; фрагментация — нет цельного свободного куска.

Урок 13. Многозадачность

1. Кооперативные задачи переключаются только в точках `await`; настоящая параллельность — два ядра одновременно.
2. Два потока одновременно меняют общий счётчик, и часть увеличений теряется; нужна блокировка.
3. Это остановит весь цикл событий; нужно `await asyncio.sleep()`.
4. Один дополнительный поток; он выполняется на втором ядре.
5. Через общий «ящик» (список) и событие `asyncio.Event`.

Урок 14. PIO

1. Четыре автомата и 32 команды на блок.
2. $125/15{,}625 = 8$ MHz, то есть 125 нс на такт; `nop() [3]` занимает 4 такта, 500 нс.
3. Side-set устанавливает выводы одновременно с командой, не тратя отдельный такт.
4. Autopull сам догружает данные; 24-битный цвет сдвигают на 8, чтобы он занимал старшие разряды слова.
5. Счётчиком в цикле известной длины; точность равна длительности одного шага цикла.

Урок 15. Питание, сон, WDT

1. VBUS — 5 В от USB; VSYS — основной вход питания (1,8–5,5 В); 3V3 — выход. Батареи подключают к VSYS.
2. Диод не даёт 5 В от USB попасть в батареи и наоборот.
3. Время = ёмкость / средний ток; во сне ток на порядки меньше.
4. Лёгкий сон сохраняет состояние, глубокий часто перезапускает программу; состояние хранят во flash или специальной памяти.
5. Если программа зависла, она перестаёт кормить таймер, и плата перезапускается; вынесенный в таймер `feed()` продолжал бы работать и при зависании.

Урок 16. Wi-Fi

1. STA подключается к чужой сети, AP создаёт свою; первое — для выхода в интернет, второе — для настройки и прямого доступа.
2. Код страны задаёт допустимые каналы и мощность; пароль хранят отдельно, чтобы он не попал в публикуемый код.
3. IP — адрес платы, маска — границы сети, шлюз — выход наружу, DNS — перевод имён в адреса; всё выдаёт роутер по DHCP.
4. Незакрытые ответы занимают память и сокеты; сеть ненадёжна, ошибки нужно перехватывать.
5. NTP отдаёт время UTC; к нему прибавляют смещение часового пояса.

Урок 17. Веб-сервер

1. Запрос: строка запроса, заголовки, пустая строка, тело; ответ: строка состояния, заголовки, тело. 200 — успех, 404 — не найдено, 403 — запрещено.
2. `bind` занимает порт, `listen` создаёт очередь, `accept()` ждёт и возвращает новое соединение.
3. Обновляется только нужная часть, трафика меньше, страница не мигает.
4. Это заблокирует весь цикл событий и остальные запросы.
5. HTTP передаётся открытым текстом; помогают локальная сеть, HTTPS, VPN и настоящая авторизация.

Урок 18. MQTT

1. В MQTT клиенты не знают друг друга, а общаются через брокера, который хранит подписки и рассылает сообщения.
2. `+` — один уровень, `#` — любое число уровней (в конце). `home/+/temp` подпишется на `home/kitchen/temp`, `home/room/temp` и т.п.
3. QoS 0 — не более одного раза, QoS 1 — не менее одного с подтверждением. Retain нужен, чтобы новый подписчик сразу получил последнее значение.
4. Брокер публикует «завещание», если клиент пропал (истёк keepalive).
5. Подписки живут вместе с сессией; одинаковые `client_id` выбрасывают друг друга.

Урок 19. BLE

1. BLE экономичнее и рассчитан на небольшие порции данных (датчики, браслеты), а не на потоковый звук.
2. GAP — реклама и соединение, GATT — структура данных; central ищет и подключается, peripheral рекламирует себя.
3. Профиль → сервисы → характеристики → дескрипторы; UUID однозначно называет сервис и характеристику.
4. Read: клиент запрашивает; write: клиент записывает; notify: устройство само присылает изменения.
5. Они делят радиочастоты и антенну; помогает сопряжение с шифрованием.

Урок 20. Моторы

1. Ток мотора намного больше допустимого для GPIO; при торможении $I = 3/3 = 1$ А.
2. Диод включают параллельно мотору в обратном направлении: он гасит импульс самоиндукции.
3. Для запаса берут ток базы в 2–3 раза больше минимального: $I_B \approx 5$ мА, $R_B \approx (3{,}3 - 0{,}7)/0{,}005 \approx 500\,\Omega$, берут 470 Ом.
4. Реверс — сменой полярности (входы в другом порядке); быстро остановить — торможением (оба входа в одинаковом состоянии).
5. Земля задаёт общий опорный уровень сигналов; без неё управление работать не будет.

Урок 21. Звук

1. Частота определяет высоту, скважность влияет на громкость (в пределах возможностей излучателя).
2. $f = 440 \cdot 2^{n/12}$; «до» первой октавы (до^1) ≈ 262 Hz.
3. Активный содержит генератор и издаёт только один тон; пассивному частоту задаёт плата.
4. Зазор разделяет ноты; пауза — нулевая громкость или частота на время паузы.
5. Динамику нужен больший ток; резистор базы ограничивает ток базы, диод защищает от выбросов катушки.

Урок 22. USB

1. Хост управляет обменом и сам начинает все передачи; устройство только отвечает.
2. Дескриптор описывает устройство; класс определяет, какой драйвер нужен хосту.
3. `boot.py` выполняется при включении и жёстком сбросе, а не при сохранении файла.
4. `release_all()` отпускает клавиши; если забыть, клавиша «залипнет».
5. Специальной сборкой TinyUSB, библиотеками `usb-device`, внешним USB-модулем; ограничения зависят от версии.

Урок 23. Отладка

1. Читают с последней строки (тип и сообщение ошибки), затем вверх по цепочке вызовов до строки своего файла.

2. Голый `except:` ловит всё, включая `KeyboardInterrupt`, и мешает остановить программу по Ctrl+C.
3. Когда журнал велик, его переименовывают и начинают новый: размер ограничен.
4. Диском владеет компьютер; в `boot.py` вызывают `storage.remount`.
5. Если его не кормить, плата перезапустится; кормить из фонового потока нельзя, так как он продолжит работать при зависании главной программы.

Урок 24. RP2350

1. Например: ядра Cortex-M33 с FPU, вдвое больше ОЗУ, больше автоматов PIO.
2. Загрузчик выбирает архитектуру по образу; код Arm и RISC-V различается, поэтому образы несовместимы.
3. Нужна прошивка для Pico 2; программа обычно не меняется, нужно проверить выводы и памяти.
4. Результат зависит от помех и сборки мусора, поэтому берут минимум или медиану.
5. Эрратум E9 — особенность входов GPIO на ранних ревизиях; проверьте ревизию чипа и описание в документации Raspberry Pi.

Урок 25. Итоговый проект

1. Требование измеримо и проверяется («работает 72 часа без перезапуска»), пожелание — нет.
2. Главный цикл отмеряет время по счётчику (`ticks_ms`), чтобы всегда оставаться отзывчивым.
3. Короткое и долгое нажатия переводят автомат экранов в разные состояния; смотрите диаграмму состояний в уроке.
4. При потере Wi-Fi станция продолжает работать и пытается подключиться снова; при сбое датчика — показывает пометку и пробует его перезапустить.
5. Flash мало и она изнашивается, поэтому журнал ограничивают и пишут редко.

Приложение F. Для педагога: планирование и организация курса

Приложение адресовано педагогу. Оно не заменяет программу объединения, но помогает быстро собрать тематическое планирование, подготовить класс и оценить результаты.

F.1 Для кого и на сколько часов рассчитан курс

Книга рассчитана на обучающихся 12–17 лет без опыта программирования. В ней двадцать пять уроков и итоговый проект. Каждый урок содержит от трёх до шести подразделов с опытами; на урок обычно уходит **два академических часа** (одно занятие по 90 минут или два по 45).

Вариант	Часов	Состав
Базовый	36	уроки 1–10 и итоговый проект (метеостанция без Wi-Fi или с готовым кодом)
Стандартный	72	уроки 1–19 и итоговый проект: полный путь до Wi-Fi, MQTT и BLE
Углублённый	108	все 25 уроков, самостоятельный проект ученика, защита на конференции

Совет

Если Wi-Fi-платы (Pico W, Pico 2 W) есть не у всех, уроки 16–19 можно провести демонстрацией одной платы на экране проектора, а остальные успеть на обычной Pico.

F.2 Структура занятия

Типичное занятие по 90 минут удобно строить так:

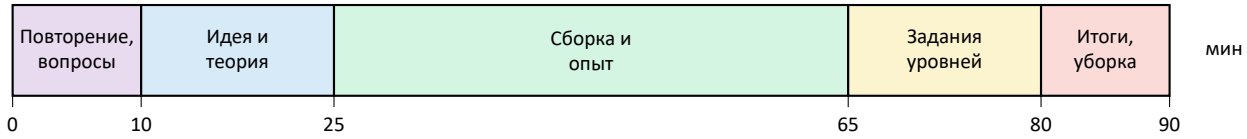


Рис. F.1: Примерная раскладка 90 минут

1. **Повторение** (10 мин): вопросы «Проверь себя» предыдущего урока, разбор домашнего задания.
2. **Идея и теория** (15 мин): раздел «Идея урока коротко», рисунки, живой пример в REPL.
3. **Сборка и опыт** (40 мин): ученики собирают схему по рисунку и запускают пример; педагог обходит класс. Рекомендуется парная работа: один собирает, второй проверяет по схеме.
4. **Задания** (15 мин): задания уровней 1–2 в классе, уровни 3–4 — дома или на следующем занятии.
5. **Итоги и порядок** (10 мин): блок «Итоги урока», отключение питания, уборка рабочих мест.

F.3 Тематическое планирование (стандартный вариант, 72 часа)

№	Тема	Часы	Результат и оборудование
Вводные главы			
—	Техника безопасности, макетная плата, мультиметр	2	знают правила; собрали первую цепь «3V3–резистор–светодиод–GND»
—	Плата Pico, чипы RP2040/RP2350, распиновка	2	ориентируются в распиновке; понимают разницу Pico, Pico W, Pico 2
1	Первый запуск: прошивка, Thonny, REPL	2	прошита плата; запущен `main.py`
2	Цифровой выход: светодиод	2	Blink, бегущий огонь; резисторы
3	Цифровой вход: кнопка	2	антидребезг, прерывания
4	Время и конечные автоматы	2	светофор без `sleep()`
5	ШИМ: яркость, звук, серво	3	«дыхание» светодиода, серво
6	АЦП: аналоговые входы	3	потенциометр, фоторезистор, термистор
7	UART и последовательный обмен	2	обмен между платами, разбор команд
8	Шина I^2C: OLED, датчики	3	OLED, BME280

№	Тема	Часы	Результат и оборудование
9	Шина SPI: SD-карта, дисплеи	2	запись на SD, сравнение шин
10	Датчики: температура, расстояние, движение	3	DHT22, DS18B20, HC-SR04
11	Адресные светодиоды WS2812	2	радуга, эффекты
12	Память и файлы	2	журнал данных, JSON
13	Многозадачность: ядра, `asyncio`	3	параллельные задачи
14	PIO: программируемый ввод-вывод	3	свой протокол на PIO
15	Питание, сон, сторожевой таймер	2	автономная работа от батареи
16	Wi-Fi	3	подключение, HTTP, NTP
17	Веб-сервер	3	панель управления с телефона
18	MQTT и интернет вещей	3	публикация и подписка
19	Bluetooth Low Energy	2	управление с телефона
20	Моторы, реле, драйверы	3	ключ на транзисторе, мотор, шаговик
21	Звук и музыка	2	мелодии, пианино
22	USB: клавиатура, мышь	2	макропад
23	Отладка и надёжность	2	методика поиска ошибок
24	RP2350 и Pico 2	2	перенос проекта, сравнение производительности
25	Итоговый проект	8	метеостанция; защита проекта

В стандартном варианте уроки 20–24 заменяются свободным временем проекта; точное число часов подбирайте под свою группу.

F.4 Оборудование на группу

Подробные списки деталей — в приложении B. Практический минимум на одну пару учеников:

- плата Pico W или Pico 2 W (лучше с распаянными контактами), кабель micro-USB *с данными*;
- макетная плата 830 точек, набор перемычек «папа-папа»;
- светодиоды разных цветов, резисторы 220 Ом, 1 кОм, 10 кОм, кнопки, потенциометр;
- дисплей SSD1306, датчик BME280, DHT22, фоторезистор;
- мультиметр на группу из 4–6 человек;
- компьютер с Thonny (Windows, macOS, Linux) и доступом в интернет для скачивания прошивок.

Осторожно

Заранее проверьте у всех кабели: в классах часто встречаются кабели «только для зарядки». Держите несколько заведомо рабочих кабелей и пару запасных плат: плата может оказаться неисправной, но чаще виноваты кабель, перепутанная полярность или неправильная прошивка (Pico и Pico W!).

F.5 Подготовка класса: чек-лист

1. Скачайте и положите на общий ресурс файлы `.uf2` для всех моделей плат группы.
2. Установите и проверьте Thonny на каждом компьютере (в учебных классах часто нет прав записи в профиль).
3. Заранее прошейте платы выбранной прошивкой и проверьте REPL.
4. Подготовьте распечатки распиновки (рис. 0.7) на каждую пару.
5. Настройте точку доступа Wi-Fi 2,4 ГГц для уроков 16–19 (Pico не работает в сети 5 ГГц).
6. Разверните брокер MQTT в локальной сети (урок 18), чтобы ученики не отправляли данные в публичные сервисы.

F.6 Критерии оценивания

Критерий	Достаточный	Хороший	Отличный
Сборка схемы	собрана по рисунку с помощью педагога	собрана самостоятельно, работает	собрана аккуратно, схему можно проверить и объяснить
Программа	запущен готовый пример	пример изменён под задачу	написана своя программа без `sleep()` там, где нужно
Понимание	называет детали и назначение	объясняет принцип работы	находит и исправляет ошибки, обосновывает выбор решений
Задания	уровни 1–2	уровни 1–3	уровни 1–4 или собственная идея
Культура работы	порядок на столе, правила ТБ	помогает соседу	ведёт документацию (схема, комментарии)

F.7 Итоговый проект и защита

Итоговый проект (урок 25) удобно оценивать по таблице в самом уроке. Защита занимает 5–7 минут: демонстрация работающего устройства, показ схемы и кода, ответы на два вопроса о принципе работы. Рекомендуется приглашать на защиту других групп.

F.8 Типичные трудности и что делать

Ситуация	Подсказка педагога
«Плата не видна»	кабель, порт, режим BOOTSEL, закрыт ли другой терминал
«Светодиод не горит»	полярность, номер GP вместо номера контакта, резистор в цепи
«Работало, а теперь нет»	не остался ли в плате старый `main.py`; не перепутаны ли провода
«Программа зависла»	Ctrl+C; если не помогает — `flash_nuke.uf2`
«Wi-Fi не подключается»	сеть 2,4 ГГц, пароль, `rp2.country()`, расстояние до точки доступа
Ученик быстро справился	задания уровня 4, помощь соседу, самостоятельный мини-проект
Ученик отстаёт	паре с сильным соседом; задания уровня 1; готовая схема без доработок

F.9 Работа с двумя языками

Книга содержит параллельные примеры на MicroPython и CircuitPython. Практика показывает два удачных сценария:

- **Один язык на группу.** Выберите один и пользуйтесь только его листингами; второй язык показывайте как расширение в конце курса (урок 24 и итоговый проект).

- **По полугодиям.** Первое полугодие — CircuitPython (проще входить и работать с готовыми драйверами), второе — MicroPython (PIO, потоки, глубина).

Смешивать языки внутри одного занятия не рекомендуется: прошивки взаимоисключающие, а переключение отнимает время.