

Joker 2026

Ни строчки C: Java 25 на железе без посредников

JNI и JNA → FFM API (Java 25+) → Pi4J v4

Николай Гриценко

Ведущий менеджер MWS Cloud Platform, контрибьютор Pi4J



Пара слов о себе



Инфраструктура

Строил Internal Developer Platform, работаю со сложными инфраструктурными задачами



15+ лет в IT — и всё ещё программист

Слежу за развитием языков, особенно люблю Java и её эволюцию



Open source

Автор библиотек для работы с GPIO и нативной памятью, контрибьютор Pi4J



О чём поговорим

01

Зачем Java разговаривать с железом

постановка проблемы

02

Как жили раньше: JNI, JNA, JNR

Экскурс в историю с примерами и болью

03

FFM API в Java 25+

объекты, вызовы, границы применимости, тайминги, память

04

Pi4J v4: архитектура на FFM

старая vs новая, что решилось и что сломалось

05

Бенчмарки

замеры и объяснение поведения

06

Примеры и демо

код и записи демок на реальном железе

01

Зачем Java разговаривать с железом?

Постановка проблемы: где JVM встречается с датчиками, ножками и системными вызовами



Где это нужно



Промышленная автоматизация

Датчики, контроллеры, конвейеры. Java-стек уже есть в enterprise — интеграция с ERP, мониторингом и шинами данных «из коробки»

Где это нужно



Промышленная автоматизация

Датчики, контроллеры, конвейеры. Java-стек уже есть в enterprise — интеграция с ERP, мониторингом и шинами данных «из коробки»



Серверные платформы

Взаимодействие компонентов внутри сервера: питание, охлаждение, метрики с датчиков через GPIO / I2C, управление фермой одноплатников

Где это нужно



Промышленная автоматизация

Датчики, контроллеры, конвейеры. Java-стек уже есть в enterprise — интеграция с ERP, мониторингом и шинами данных «из коробки»



Серверные платформы

Взаимодействие компонентов внутри сервера: питание, охлаждение, метрики с датчиков через GPIO / I2C, управление фермой одноплатников



Робототехника и IoT

Одноплатники как мозг устройства: JVM-экосистема, JavaFX-интерфейсы, OTA-обновления, знакомый тулинг

Где это нужно



Промышленная автоматизация

Датчики, контроллеры, конвейеры. Java-стек уже есть в enterprise — интеграция с ERP, мониторингом и шинами данных «из коробки»



Серверные платформы

Взаимодействие компонентов внутри сервера: питание, охлаждение, метрики с датчиков через GPIO / I2C, управление фермой одноплатников



Робототехника и IoT

Одноплатники как мозг устройства: JVM-экосистема, JavaFX-интерфейсы, OTA-обновления, знакомый тулинг



Системные задачи

Потребление памяти в контейнере, криптография, GPU, ML-рантаймы — всё это тоже нативные вызовы

JVM живёт в песочнице, железо — нет

Мир JVM

- Управляемая память, GC
- Безопасность типов
- Ничего не знает про пины и регистры

JVM живёт в песочнице, железо — нет

Мир JVM

- Управляемая память, GC
- Безопасность типов
- Ничего не знает про пины и регистры

Мир железа

- C ABI, указатели, структуры
- Syscalls: open / ioctl / poll
- Драйверы ядра, /dev/*

JVM живёт в песочнице, железо — нет

Мир JVM

- Управляемая память, GC
- Безопасность типов
- Ничего не знает про пины и регистры

связь?

Мир железа

- C ABI, указатели, структуры
- Syscalls: open / ioctl / poll
- Драйверы ядра, /dev/*

Хотим из Java:

- вызывать код, написанный не нами и не на Java
- читать датчики прямо с ножек контроллера и управлять периферией
- делать это быстро — без лишних копирований и накладных расходов
- и не ронять JVM при каждой ошибке

От приложения до железа

Ваше приложение

Интерфейсы периферии

GPIO

дискретный ввод-вывод, кнопки, реле, LED

I2C

медленная шина: датчики температуры, дисплеи, EEPROM

SPI

быстрая шина: матрицы, АЦП, флеш-память

PWM

ШИМ: моторы, сервоприводы, яркость

От приложения до железа

Ваше приложение



Системные вызовы `open · read · write · ioctl · poll`

Интерфейсы периферии

GPIO

дискретный ввод-вывод, кнопки, реле, LED

I2C

медленная шина: датчики температуры, дисплеи, EEPROM

SPI

быстрая шина: матрицы, АЦП, флеш-память

PWM

ШИМ: моторы, сервоприводы, яркость

От приложения до железа

Ваше приложение



Системные вызовы `open · read · write · ioctl · poll`



Ядро Linux драйверы: `gpio-cdev, i2c-dev, spidev, pwm`

Интерфейсы периферии

GPIO

дискретный ввод-вывод, кнопки, реле, LED

I2C

медленная шина: датчики температуры, дисплеи, EEPROM

SPI

быстрая шина: матрицы, АЦП, флеш-память

PWM

ШИМ: моторы, сервоприводы, яркость

От приложения до железа

Ваше приложение



Системные вызовы `open · read · write · ioctl · poll`



Ядро Linux драйверы: `gpio-cdev, i2c-dev, spidev, pwm`



Железо пины, шины, контроллеры, датчики

Интерфейсы периферии

GPIO

дискретный ввод-вывод, кнопки, реле, LED

I2C

медленная шина: датчики температуры, дисплеи, EEPROM

SPI

быстрая шина: матрицы, АЦП, флеш-память

PWM

ШИМ: моторы, сервоприводы, яркость

DMA — мимо ядра, прямо в регистры

Приложение



`mmap /dev/mem` отображаем физическую память



Регистры SoC (BCM283x) GPIO, DMA-контроллер, таймеры

Ядро Linux — остаётся в стороне

Что такое «регистры»: memory-mapped I/O

Физическое адресное
пространство

RAM

обычные данные программ

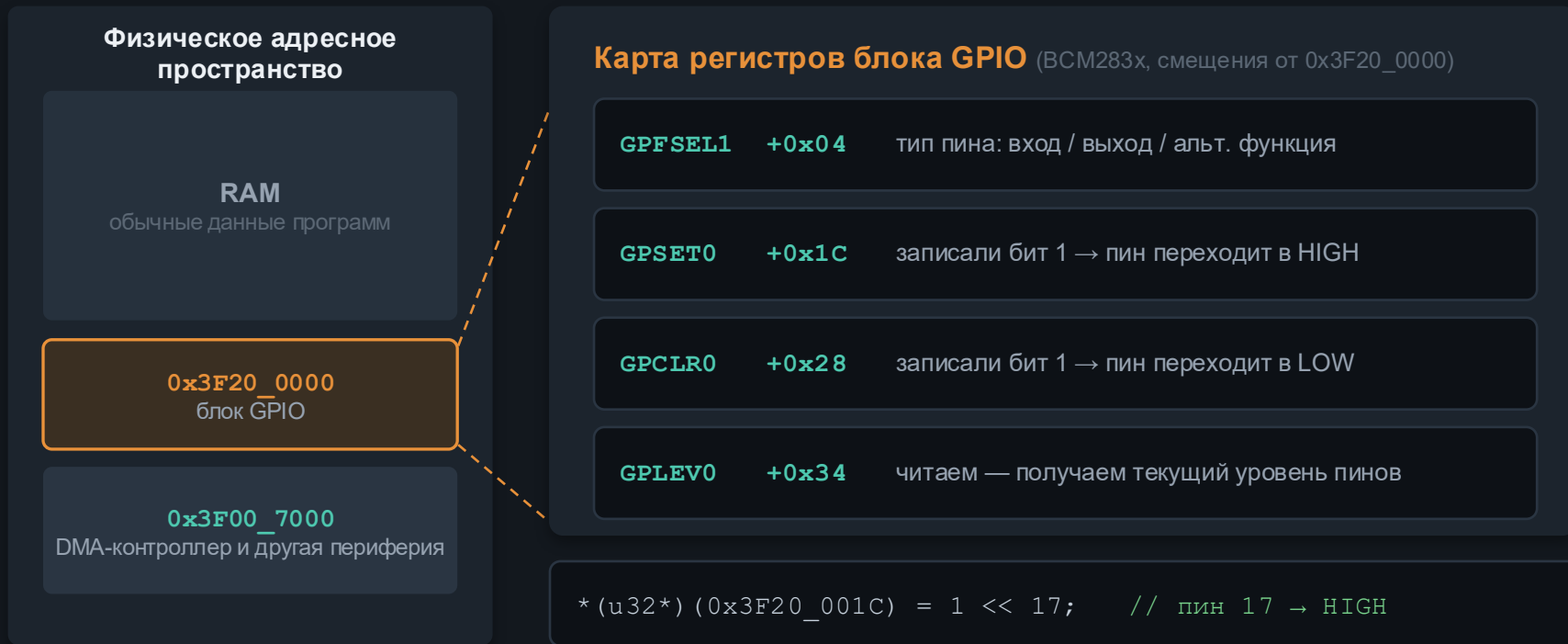
0x3F20_0000

блок GPIO

0x3F00_7000

DMA-контроллер и другая периферия

Что такое «регистры»: memory-mapped I/O



Регистр — не память, а «ручка» устройства по адресу: запись = действие. DMA-контроллер — такая же периферия: ему пишут «переливай данные сам», и он работает без участия CPU.

DMA — мимо ядра, прямо в регистры

Приложение



`mmap /dev/mem` отображаем физическую память



Регистры SoC (BCM283x) GPIO, DMA-контроллер, таймеры

Ядро Linux — остаётся в стороне



Плюсы

- Микросекундные тайминги, bit-banging, точный software-PWM
- Максимальная скорость: без syscall на каждую операцию



Минусы

- root и полный доступ к физической памяти
- Адреса регистров зависят от SoC — RPi 5 (чип RP1) сломал pigpio
- Не работает в контейнерах



02

Как это работало раньше: JNI, JNA, JNR

Экскурс в 27 лет истории нативных вызовов — и почему все дороги вели к боли

27 лет эволюции нативных вызовов

1997

2006

2010



JNI

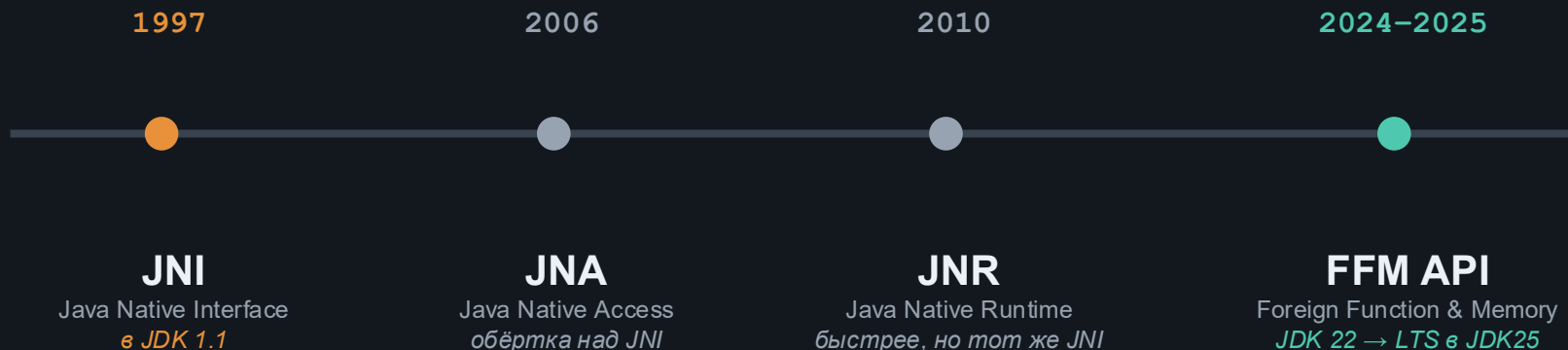
Java Native Interface

в *JDK 1.1*

27 лет эволюции нативных вызовов



27 лет эволюции нативных вызовов



14 лет развития → **17 JEP-ов**, чтобы сделать правильно:
JEP 370 · 383 · 393 память → 389 · 412 · 419 линкер → 424 · 434 · 442 превью → JEP 454 финал, JDK 22

JNI глазами его создателей



I can't remember who it was, but I was talking to one of the engineers back in the Sun days and expressing my frustration with JNI being hard to use.

That's when he said it had been developed that way to encourage people NOT to use it...

Simon Ritter, Deputy CTO of Azul

«...его разработали так, чтобы отбить желание им пользоваться». Simon Ritter, Deputy CTO Azul — про JNI со слов инженера Sun

JNI: пять шагов

Java-код
с native-методом

Gpio.java — Java-сторона

```
public class Gpio {  
    static { System.loadLibrary("gpio_jni"); }  
    public static native int readPin(int pin);  
}
```

JNI: пять шагов

Java-код
с native-методом



Генерация
header-файла



Пишем
C/C++

Gpio.java — Java-сторона

```
public class Gpio {  
    static { System.loadLibrary("gpio_jni"); }  
    public static native int readPin(int pin);  
}
```

gpio_jni.c — придётся написать и собрать

```
JNIEXPORT jint JNICALL  
Java_Gpio_readPin(JNIEnv*, jclass, jint pin) {  
    return gpiod_line_get_value(lines[pin]); // C!  
}
```

JNI: пять шагов

Java-код
с native-методом



Генерация
header-файла



Пишем
C/C++



Собираем .so
под все платформы



Запускаем
и молимся

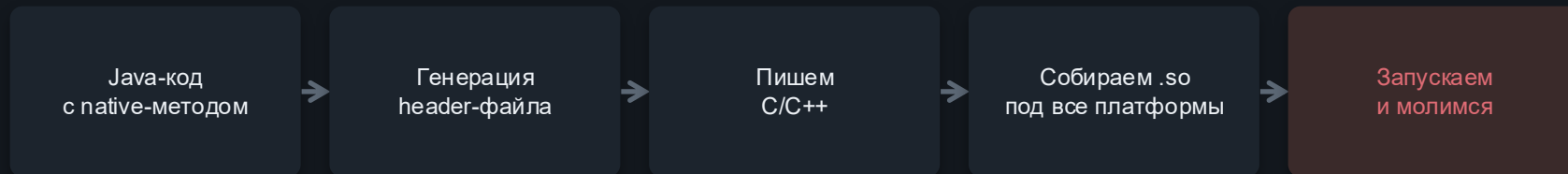
Gpio.java — Java-сторона

```
public class Gpio {  
    static { System.loadLibrary("gpio_jni"); }  
    public static native int readPin(int pin);  
}
```

gpio_jni.c — придётся написать и собрать

```
JNIEXPORT jint JNICALL  
Java_Gpio_readPin(JNIEnv*, jclass, jint pin) {  
    return gpiod_line_get_value(lines[pin]); // C!  
}
```

JNI: пять шагов



Gpio.java — Java-сторона

```
public class Gpio {  
    static { System.loadLibrary("gpio_jni"); }  
    public static native int readPin(int pin);  
}
```

gpio_jni.c — придётся написать и собрать

```
JNIEXPORT jint JNICALL  
Java_Gpio_readPin(JNIEnv*, jclass, jint pin) {  
    return gpio_line_get_value(lines[pin]); // C!  
}
```



Нужно уметь писать и собирать C/C++: исходники, кросс-компиляция под ARM, правильный тулчейн... Но мы ведь Java-разработчики!

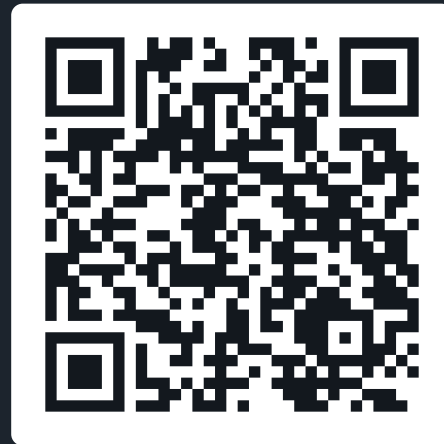
JNI и JNA под микроскопом



Здесь мы пробежались по верхам.

Как JNI устроен изнутри, чем JNA платит за удобство и как выглядят первые шаги с FFM — я разобрал в отдельном часовом докладе с примерами кода.

youtube.com/watch?v=WH5bWs34dzs



Запись доклада

Сводка подходов

	JNI	JNA / JNR
Код на C/C++	пишем сами + собираем	не пишем, но либа нужна
Под капотом	прямой native-мост	тот же JNI + Reflection / стабы
Скорость вызова	быстро, но без JIT-инлайна	медленнее из-за прослойки
Обмен данными	копирование heap ↔ offheap	копирование + маршалинг
Ошибка на границе	краш JVM	краш JVM
JVM знает о нативе	почти ничего	ничего

03

FFM API: нативный код без C-обёрток

Java 22 (JEP 454) → Java 25. Объекты, вызовы, границы
применимости, тайминги и память

FFM API: четыре кита



Arena

Управление жизненным циклом нативной памяти: global
· ofAuto · ofConfined · ofShared

FFM API: четыре кита



Arena

Управление жизненным циклом нативной памяти: global
· ofAuto · ofConfined · ofShared



MemorySegment + ValueLayout

Java-абстракция над регионом памяти (heap и off-heap):
адрес, размер, границы доступа

FFM API: четыре кита



Arena

Управление жизненным циклом нативной памяти: global
· ofAuto · ofConfined · ofShared



MemorySegment + ValueLayout

Java-абстракция над регионом памяти (heap и off-heap):
адрес, размер, границы доступа



MemoryLayout

Описание нативных структур: struct, union, массивы,
выравнивание — модель данных на стыке

FFM API: четыре кита



Arena

Управление жизненным циклом нативной памяти: global
· ofAuto · ofConfined · ofShared



MemorySegment + ValueLayout

Java-абстракция над регионом памяти (heap и off-heap):
адрес, размер, границы доступа



MemoryLayout

Описание нативных структур: struct, union, массивы,
выравнивание — модель данных на стыке



Linker + MethodHandle

Поиск символов в библиотеке и вызовы: downcall (Java
→ native) и upcall (native → Java)

Вызов ioctl — ни одной строчки C

```
PureJavaGpio.java  
  
Linker linker = Linker.nativeLinker();  
MethodHandle ioctl1 = linker.downcallHandle(  
  
);
```

Чистые Java-абстракции

Ошибка → исключение, не
краш

Не только C: Rust, Go, Zig

jextract: биндинги из .h

Вызов ioctl — ни одной строки C

```
PureJavaGpio.java

Linker linker = Linker.nativeLinker();
MethodHandle ioctl = linker.downcallHandle(
    linker.defaultLookup().find("ioctl").orElseThrow(),           // 1. символ в libc
    FunctionDescriptor.of(JAVA_INT, JAVA_INT, JAVA_LONG, ADDRESS) // 2. сигнатура
);
```

Чистые Java-абстракции

Ошибка → исключение, не
краш

Не только C: Rust, Go, Zig

jextract: биндинги из .h

Вызов ioctl — ни одной строки C

PureJavaGpio.java

```
Linker linker = Linker.nativeLinker();
MethodHandle ioctl1 = linker.downcallHandle(
    linker.defaultLookup().find("ioctl").orElseThrow(),           // 1. символ в libc
    FunctionDescriptor.of(JAVA_INT, JAVA_INT, JAVA_LONG, ADDRESS) // 2. сигнатура
);
try (Arena arena = Arena.ofConfined()) {                          // 3. время жизни памяти

}
```

Чистые Java-абстракции

Ошибка → исключение, не краш

Не только C: Rust, Go, Zig

jextract: биндинги из .h

Вызов ioctl — ни одной строки C

PureJavaGpio.java

```
Linker linker = Linker.nativeLinker();
MethodHandle ioctl = linker.downcallHandle(
    linker.defaultLookup().find("ioctl").orElseThrow(),           // 1. символ в libc
    FunctionDescriptor.of(JAVA_INT, JAVA_INT, JAVA_LONG, ADDRESS) // 2. сигнатура
);
try (Arena arena = Arena.ofConfined()) {                          // 3. время жизни памяти
    MemorySegment req = arena.allocate(GPIO_V2_LINE_REQUEST);     // 4. нативная структура
    VarHandle num = GPIO_V2_LINE_REQUEST
        .varHandle(groupElement("num_lines"));
    num.set(req, 0L, 1);
}
```

Чистые Java-абстракции

Ошибка → исключение, не краш

Не только C: Rust, Go, Zig

jextract: биндинги из .h

Вызов ioctl — ни одной строки C

```
PureJavaGpio.java

Linker linker = Linker.nativeLinker();
MethodHandle ioctl = linker.downcallHandle(
    linker.defaultLookup().find("ioctl").orElseThrow(),           // 1. символ в libc
    FunctionDescriptor.of(JAVA_INT, JAVA_INT, JAVA_LONG, ADDRESS) // 2. сигнатура
);
try (Arena arena = Arena.ofConfined()) {                          // 3. время жизни памяти
    MemorySegment req = arena.allocate(GPIO_V2_LINE_REQUEST);     // 4. нативная структура
    VarHandle num = GPIO_V2_LINE_REQUEST
        .varHandle(groupElement("num_lines"));
    num.set(req, 0L, 1);                                           // 5. пишем поле структуры
    VarHandle offs = GPIO_V2_LINE_REQUEST
        .varHandle(groupElement("offsets"), sequenceElement());
    offs.set(req, 0L, 0L, 17);                                     // offsets[0] = пин 17
    int rc = (int) ioctl.invokeExact(fd, GPIO_V2_GET_LINE_IOCTL, req); // 6. вызов!
}                                                                    // 7. память освобождена
```

Чистые Java-абстракции

Ошибка → исключение, не
краш

Не только C: Rust, Go, Zig

jextract: биндинги из .h

Время исполнения: гарантии и их отсутствие



Стоимость downcall-a

- Прогретый вызов — десятки наносекунд, как JNI и лучше JNA
- `Linker.Option.critical()` — срезаем переходы состояний для коротких вызовов
- Но: внутри `critical` нельзя блокироваться и делать `upcall`

Время исполнения: гарантии и их отсутствие



Стоимость downcall-a

- Прогретый вызов — десятки наносекунд, как JNI и лучше JNA
- `Linker.Option.critical()` — срезаем переходы состояний для коротких вызовов
- Но: внутри `critical` нельзя блокироваться и делать `upcall`



Источники задержек

- Холодный старт: `MethodHandle` в интерпретаторе, пока JIT не прогрелся
- Safepoint- и GC-паузы: миллисекунды могут прилететь в любой момент
- Планировщик ОС: JVM-поток — обычный процесс Linux

Время исполнения: гарантии и их отсутствие



Стоимость downcall-а

- Прогретый вызов — десятки наносекунд, как JNI и лучше JNA
- `Linker.Option.critical()` — срезаем переходы состояний для коротких вызовов
- Но: внутри `critical` нельзя блокироваться и делать `upcall`



Источники задержек

- Холодный старт: `MethodHandle` в интерпретаторе, пока JIT не прогрелся
- Safepoint- и GC-паузы: миллисекунды могут прилететь в любой момент
- Планировщик ОС: JVM-поток — обычный процесс Linux

Что это значит для железа

Soft real-time — да

события раз в миллисекунды, опрос датчиков

Hard real-time — нет

микросекундные тайминги отдаём ядру (PREEMPT_RT) или микроконтроллеру

Чаще всего задержки в шине или ядре

Время исполнения: гарантии и их отсутствие



Стоимость downcall-а

- Прогретый вызов — десятки наносекунд, как JNI и лучше JNA
- `Linker.Option.critical()` — срезаем переходы состояний для коротких вызовов
- Но: внутри `critical` нельзя блокироваться и делать `upcall`



Источники задержек

- Холодный старт: `MethodHandle` в интерпретаторе, пока JIT не прогрелся
- Safepoint- и GC-паузы: миллисекунды могут прилететь в любой момент
- Планировщик ОС: JVM-поток — обычный процесс Linux

Что это значит для железа

Soft real-time — да

события раз в миллисекунды, опрос датчиков

Hard real-time — нет

микросекундные тайминги отдаём ядру (PREEMPT_RT) или микроконтроллеру

Чаще всего задержки в шине или ядре

Рецепт: прогрев → `static final MethodHandle` → low-pause GC (ZGC)

Память: из чего складывается footprint

Java heap — то, что видит GC

`MemorySegment` $\approx$ 48 Б

обёртка: адрес + размер + scope

`Arena.ofConfined()` $\approx$ 50 Б

scope + учёт сегментов

GC видит только ~100 Б

Память: из чего складывается footprint

Java heap — то, что видит GC

`MemorySegment` $\approx$ 48 Б
обёртка: адрес + размер + score

`Arena.ofConfined()` $\approx$ 50 Б
score + учёт сегментов

GC видит только ~100 Б

Off-heap: `gpio_v2_line_request` — 592 Б (+ ~16 Б заголовок malloc)

`offsets[64]` 256 Б какие пины запрашиваем

`config` 272 Б флаги и атрибуты линий

`consumer` 32 Б имя владельца запроса

`num_lines` · `fd` · `pad` 32 Б служебные поля

Память: из чего складывается footprint

Java heap — то, что видит GC

`MemorySegment` $\approx$ 48 Б
обёртка: адрес + размер + score

`Arena.ofConfined()` $\approx$ 50 Б
score + учёт сегментов

GC видит только ~100 Б

Off-heap: `gpio_v2_line_request` — 592 Б (+ ~16 Б заголовков malloc)

`offsets[64]` 256 Б какие пины запрашиваем

`config` 272 Б флаги и атрибуты линий

`consumer` 32 Б имя владельца запроса

`num_lines · fd · pad` 32 Б служебные поля



Layout — контракт с ядром, байт в байт. Ошибка в layout — EINVAL от ioctl в лучшем случае и тихий мусор в чужом поле в худшем.

Память: из чего складывается footprint

Java heap — то, что видит GC

`MemorySegment` $\approx$ 48 Б
обёртка: адрес + размер + score

`Arena.ofConfined()` $\approx$ 50 Б
score + учёт сегментов

GC видит только ~100 Б

Off-heap: `gpio_v2_line_request` — 592 Б (+ ~16 Б заголовков malloc)

`offsets[64]` 256 Б какие пины запрашиваем

`config` 272 Б флаги и атрибуты линий

`consumer` 32 Б имя владельца запроса

`num_lines` · `fd` · `pad` 32 Б служебные поля



Layout — контракт с ядром, байт в байт. Ошибка в layout — EINVAL от ioctl в лучшем случае и тихий мусор в чужом поле в худшем.

Арена решает, когда off-heap вернётся системе. Следить: RSS процесса + NMT (-XX:NativeMemoryTracking)

`ofConfined()`
`close()` → сразу; один поток

`ofShared()`
`close()` → сразу; из любых потоков

`ofAuto()`
освободит GC —
недетерминированно

`global()`
никогда — живёт вместе с JVM

Память: правила и живой кейс



Короткая жизнь → per-call ofConfined()

iostl-запрос GPIO/I2C: аллоцировал, вызвал, закрыл —
детерминированно и дёшево

```
try (var a = Arena.ofConfined()) { } // сразу free
```

Память: правила и живой кейс



Короткая жизнь → per-call ofConfined()

ioctl-запрос GPIO/I2C: аллоцировал, вызвал, закрыл —
детерминированно и дёшево

```
try (var a = Arena.ofConfined()) { } // сразу free
```



Горячий путь → переиспользуй сегмент

Буфер SPI-кадра LED-матрицы: один сегмент на всё время жизни
устройства

```
MemorySegment frame = arena.allocate(4096); // на устройство
```

Память: правила и живой кейс



Короткая жизнь → per-call ofConfined()

ioctl-запрос GPIO/I2C: аллоцировал, вызвал, закрыл — детерминированно и дешево

```
try (var a = Arena.ofConfined()) { } // сразу free
```



Горячий путь → переиспользуй сегмент

Буфер SPI-кадра LED-матрицы: один сегмент на всё время жизни устройства

```
MemorySegment frame = arena.allocate(4096); // на устройство
```



Живой кейс: Pi4J #628

Симптом: OutOfMemoryError после часов опроса датчиков по I2C

Причина: одна глобальная арена на все вызовы — сегменты копятя и ждут GC

```
- Arena.ofAuto()           // одна на всё  
+ Arena.ofConfined()      // на вызов
```

Лечение: fix в v4.0.2 — память освобождается сразу



Pi4J/pi4j · issue #628

Память: правила и живой кейс



Короткая жизнь → per-call ofConfined()

ioctl-запрос GPIO/I2C: аллоцировал, вызвал, закрыл — детерминированно и дёшево

```
try (var a = Arena.ofConfined()) { } // сразу free
```



Горячий путь → переиспользуй сегмент

Буфер SPI-кадра LED-матрицы: один сегмент на всё время жизни устройства

```
MemorySegment frame = arena.allocate(4096); // на устройство
```



Много мелких аллокаций → одна большая + slice()

Блочное чтение регистров I2C: одна аллокация, дальше нарезка без malloc

```
var a = big.asSlice(0, 64), b = big.asSlice(64, 64);
```



Живой кейс: Pi4J #628

Симптом: OutOfMemoryError после часов опроса датчиков по I2C

Причина: одна глобальная арена на все вызовы — сегменты копятя и ждут GC

```
- Arena.ofAuto()           // одна на всё  
+ Arena.ofConfined()       // на вызов
```

Лечение: fix в v4.0.2 — память освобождается сразу



Pi4J/pi4j · issue #628

Память: правила и живой кейс



Короткая жизнь → per-call ofConfined()

ioctl-запрос GPIO/I2C: аллоцировал, вызвал, закрыл —
детерминированно и дёшево

```
try (var a = Arena.ofConfined()) { } // crazy free
```



Горячий путь → переиспользуй сегмент

Буфер SPI-кадра LED-матрицы: один сегмент на всё время жизни
устройства

```
MemorySegment frame = arena.allocate(4096); // на устройство
```



Много мелких аллокаций → одна большая + slice()

Блочное чтение регистров I2C: одна аллокация, дальше нарезка без
malloc

```
var a = big.asSlice(0, 64), b = big.asSlice(64, 64);
```



Передавай SegmentAllocator в API

API приложения принимает аллокатор — вызывающий решает, где
живёт память

```
void read(SegmentAllocator alloc) { } // решает вызывающий
```



Живой кейс: Pi4J #628

Симптом: OutOfMemoryError после часов
опроса датчиков по I2C

Причина: одна глобальная арена на все
вызовы — сегменты копятя и ждут GC

```
- Arena.ofAuto() // одна на всё  
+ Arena.ofConfined() // на вызов
```

Лечение: fix в v4.0.2 — память
освобождается сразу



Pi4J/pi4j · issue #628

FFM + GraalVM: быстрый старт, но есть нюанс

Мгновенный старт

миллисекунды вместо секунд JVM

FFM + GraalVM: быстрый старт, но есть нюанс

Мгновенный старт

миллисекунды вместо секунд JVM

Меньше памяти

десятки МБ RSS вместо сотен

FFM + GraalVM: быстрый старт, но есть нюанс

Мгновенный старт

миллисекунды вместо секунд JVM

Меньше памяти

десятки МБ RSS вместо сотен

Нет прогрева

AOT: стабильно с первого вызова

FFM + GraalVM: быстрый старт, но есть нюанс

Мгновенный старт

миллисекунды вместо секунд JVM

Меньше памяти

десятки МБ RSS вместо сотен

Нет прогрева

AOT: стабильно с первого вызова

Но: FFM в native image пока медленный

- Issue #12219: просадка ~20x (400 → 25 fps, FFM + SDL, GraalVM 25)
- Issue #8113: downcall / upcall не оптимизированы
- Arena.ofShared за флагом -H:+SharedArenaSupport

issues на GitHub



#12219



#8113

FFM + GraalVM: быстрый старт, но есть нюанс

Мгновенный старт

миллисекунды вместо секунд JVM

Меньше памяти

десятки МБ RSS вместо сотен

Нет прогрева

AOT: стабильно с первого вызова

Но: FFM в native image пока медленный

- Issue #12219: просадка ~20x (400 → 25 fps, FFM + SDL, GraalVM 25)
- Issue #8113: downcall / upcall не оптимизированы
- Arena.ofShared за флагом -H:+SharedArenaSupport

issues на GitHub



#12219



#8113

Сигнатуры downcall фиксируются на этапе сборки

```
// reachability-metadata.json:
{"foreign":{"downcalls":[{"returnType":"int","parameterTypes":["int","long","pointer"]}]}
// или Feature.duringSetup():
RuntimeForeignAccess.registerForDowncall(FunctionDescriptor.of(JAVA_INT, JAVA_INT, JAVA_LONG, ADDRESS));
```

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме
- Биндинги к Rust / Go / Zig — всё, что экспортирует C ABI

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме
- Биндинги к Rust / Go / Zig — всё, что экспортирует C ABI
- Память под контролем: арены дают детерминированное освобождение

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме
- Биндинги к Rust / Go / Zig — всё, что экспортирует C ABI
- Память под контролем: арены дают детерминированное освобождение



Осторожно / не подходит

- Прерываний в JVM нет: только poll() на дескрипторе — событие доходит за ~50 мкс, не за микросекунды

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме
- Биндинги к Rust / Go / Zig — всё, что экспортирует C ABI
- Память под контролем: арены дают детерминированное освобождение



Осторожно / не подходит

- Прерываний в JVM нет: только poll() на дескрипторе — событие доходит за ~50 мкс, не за микросекунды
- Hard real-time и бит-бэнгинг микросекундами — это работа ядра или МК

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме
- Биндинги к Rust / Go / Zig — всё, что экспортирует C ABI
- Память под контролем: арены дают детерминированное освобождение



Осторожно / не подходит

- Прерываний в JVM нет: только poll() на дескрипторе — событие доходит за ~50 мкс, не за микросекунды
- Hard real-time и бит-бэнгинг микросекундами — это работа ядра или МК
- C++ напрямую: name mangling, классы, исключения → нужен extern "C"

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме
- Биндинги к Rust / Go / Zig — всё, что экспортирует C ABI
- Память под контролем: арены дают детерминированное освобождение



Осторожно / не подходит

- Прерываний в JVM нет: только poll() на дескрипторе — событие доходит за ~50 мкс, не за микросекунды
- Hard real-time и бит-бэнгинг микросекундами — это работа ядра или МК
- C++ напрямую: name mangling, классы, исключения → нужен extern "C"
- Объекты: писать поля объектов нельзя — JNI пока богаче

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме
- Биндинги к Rust / Go / Zig — всё, что экспортирует C ABI
- Память под контролем: арены дают детерминированное освобождение



Осторожно / не подходит

- Прерываний в JVM нет: только poll() на дескрипторе — событие доходит за ~50 мкс, не за микросекунды
- Hard real-time и бит-бэнгинг микросекундами — это работа ядра или МК
- C++ напрямую: name mangling, классы, исключения → нужен extern "C"
- Объекты: писать поля объектов нельзя — JNI пока богаче
- GraalVM native image: downcall пока без оптимизаций

Границы применимости FFM



FFM — отличный выбор

- Вызовы C ABI: libc, syscalls, драйверы, системные библиотеки
- Структуры для ioctl описываются MemoryLayout — без C-заголовков в рантайме
- Биндинги к Rust / Go / Zig — всё, что экспортирует C ABI
- Память под контролем: арены дают детерминированное освобождение



Осторожно / не подходит

- Прерываний в JVM нет: только poll() на дескрипторе — событие доходит за ~50 мкс, не за микросекунды
- Hard real-time и бит-бэнгинг микросекундами — это работа ядра или МК
- C++ напрямую: name mangling, классы, исключения → нужен extern "C"
- Объекты: писать поля объектов нельзя — JNI пока богаче
- GraalVM native image: downcall пока без оптимизаций

Правило: FFM — это граница «Java ↔ C ABI», а не просто мост внутрь чужого рантайма



04

Pi4J v4/5: новая архитектура на FFM

Как open source проект с 2012 года избавился от JNI, JNA и Docker-сборок нативных библиотек

Pi4J: Java для Raspberry Pi с 2012 года



2012

год рождения проекта

V4

текущая мажорная версия

Java 25

минимальная версия для V4

5 провайдеров

GPIO in/out, I2C, SPI, PWM

Было (V2/V3): три цепочки, два слоя натива

Приложение пользователя

Было (V2/V3): три цепочки, два слоя натива

Приложение пользователя

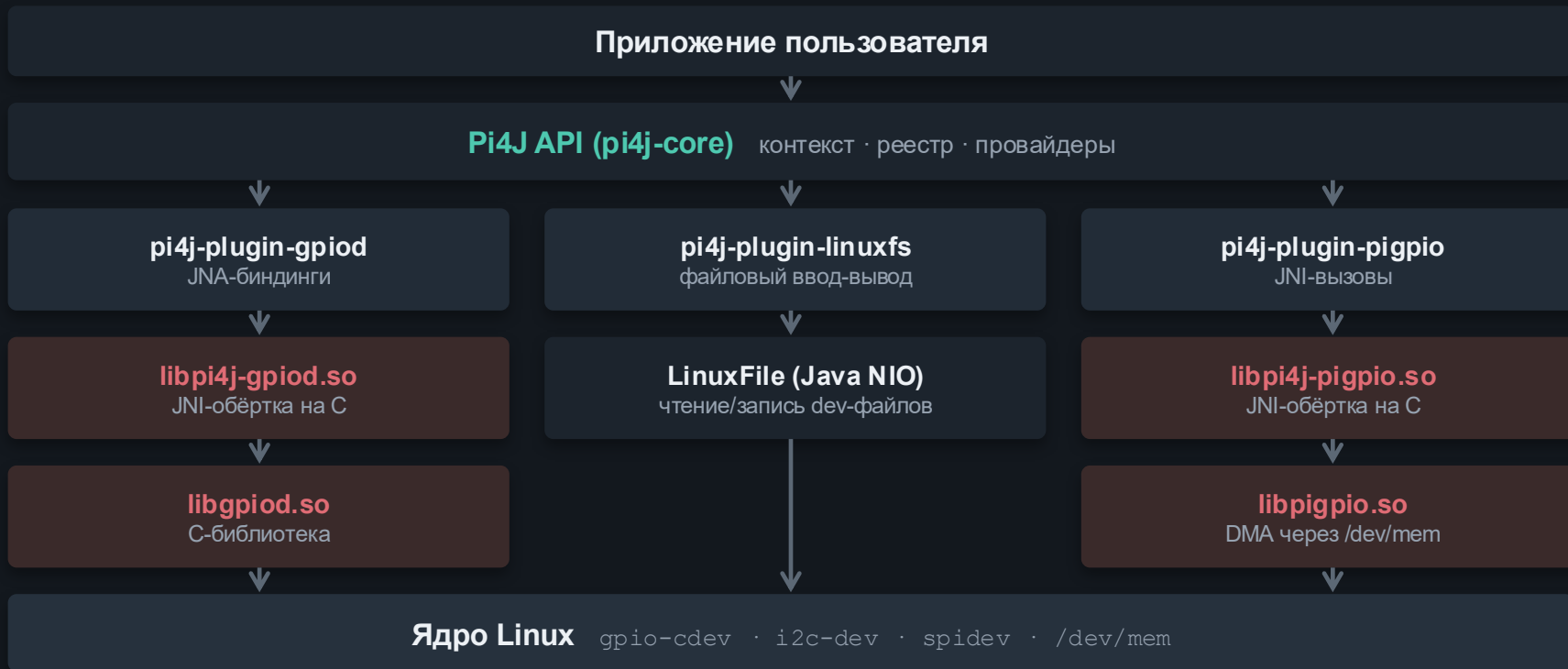


Pi4J API (pi4j-core) контекст · реестр · провайдеры

 нативный код: своя цель Docker-сборки под armv6 / armv7 / aarch64

 Java-код проекта Pi4J

Было (V2/V3): три цепочки, два слоя натива



■ нативный код: своя цель Docker-сборки под armv6 / armv7 / aarch64

□ Java-код проекта Pi4J

Стало (V4/5): Java → syscall → ядро

Приложение + Pi4J API (pi4j-core)

публичный API не изменился — миграция почти бесплатна

Стало (V4/5): Java → syscall → ядро

Приложение + Pi4J API (pi4j-core)

публичный API не изменился — миграция почти бесплатна



pi4j-plugin-ffm — чистая Java, ноль C-кода

digital in / out · i2c: direct · smbus · file · spi · pwm (hardware)

Стало (V4/5): Java → syscall → ядро

Приложение + Pi4J API (pi4j-core)

публичный API не изменился — миграция почти бесплатна



pi4j-plugin-ffm — чистая Java, ноль C-кода

digital in / out · i2c: direct · smbus · file · spi · pwm (hardware)



java.lang.foreign (FFM API)

Linker · MemoryLayout · VarHandle · Arena per-call

Стало (V4/5): Java → syscall → ядро

Приложение + Pi4J API (pi4j-core)

публичный API не изменился — миграция почти бесплатна



pi4j-plugin-ffm — чистая Java, ноль C-кода

digital in / out · i2c: direct · smbus · file · spi · pwm (hardware)



java.lang.foreign (FFM API)

Linker · MemoryLayout · VarHandle · Arena per-call



downcall в libc: ioctl · open · read · write · poll

стабильный ABI ядра — наш контракт

Стало (V4/5): Java → syscall → ядро

Приложение + Pi4J API (pi4j-core)

публичный API не изменился — миграция почти бесплатна



pi4j-plugin-ffm — чистая Java, ноль C-кода

digital in / out · i2c: direct · smbus · file · spi · pwm (hardware)



java.lang.foreign (FFM API)

Linker · MemoryLayout · VarHandle · Arena per-call



downcall в libc: ioctl · open · read · write · poll

стабильный ABI ядра — наш контракт



Ядро Linux

gpio-cdev · i2c-dev · spidev · pwm sysfs

Любой Linux с GLIBC ≥ 2.13: x86 · ARM · RISC-V — не только Raspberry Pi

Что дал переход — помимо бенчмарков



Ноль С-кода в репозитории

нечего кросс-компилировать, нечего отлаживать
на двух языках

Что дал переход — помимо бенчмарков



Ноль С-кода в репозитории

нечего кросс-компилировать, нечего отлаживать
на двух языках



Минус Docker-конвейеры сборки

цели armv6 / armv7 и их поддержка

Что дал переход — помимо бенчмарков



Ноль С-кода в репозитории

нечего кросс-компилировать, нечего отлаживать
на двух языках



Минус Docker-конвейеры сборки

цели armv6 / armv7 и их поддержка



Один артефакт на все архитектуры

тот же jar на x86, ARM и RISC-V

Что дал переход — помимо бенчмарков



Ноль С-кода в репозитории

нечего кросс-компилировать, нечего отлаживать
на двух языках



Минус Docker-конвейеры сборки

цели armv6 / armv7 и их поддержка



Один артефакт на все архитектуры

тот же jar на x86, ARM и RISC-V



Ошибка — исключение, не краш

честный Java-стектрейс вместо SIGSEGV из JNI-
обёртки

Что дал переход — помимо бенчмарков



Ноль С-кода в репозитории

нечего кросс-компилировать, нечего отлаживать на двух языках



Минус Docker-конвейеры сборки

цели armv6 / armv7 и их поддержка



Один артефакт на все архитектуры

тот же jar на x86, ARM и RISC-V



Ошибка — исключение, не краш

честный Java-стектрейс вместо SIGSEGV из JNI-обёртки



Контракт — ABI ядра

нет API чужих библиотек: libgpiod и pigpio

Что дал переход — помимо бенчмарков



Ноль С-кода в репозитории

нечего кросс-компилировать, нечего отлаживать на двух языках



Минус Docker-конвейеры сборки

цели armv6 / armv7 и их поддержка



Один артефакт на все архитектуры

тот же jar на x86, ARM и RISC-V



Ошибка — исключение, не краш

честный Java-стектрейс вместо SIGSEGV из JNI-обёртки



Контракт — ABI ядра

нет API чужих библиотек: libgriod и pigpio



Порог входа — Java

контрибьютору не нужен C-тулчейн; права через udev и группы, без root

Мигрировали за один релиз: публичный API не изменился — прикладной код Pi4J V2/V3 работает на V4 без правок

Но у медали две стороны: новые проблемы



Порог версии: нужен JDK 25+
есть свежая Java под ARM

Старые образы и дистрибутивы для плат отсекаются — не везде

Но у медали две стороны: новые проблемы



Порог версии: нужен JDK 25+

есть свежая Java под ARM

Старые образы и дистрибутивы для плат отсекаются — не везде



Native access теперь явный

шаг настройки

--enable-native-access=... или манифест: безопасность, но ещё один

Но у медали две стороны: новые проблемы



Порог версии: нужен JDK 25+

есть свежая Java под ARM

Старые образы и дистрибутивы для плат отсекаются — не везде



Native access теперь явный

шаг настройки

--enable-native-access=... или манифест: безопасность, но ещё один



Memory-safety — наша ответственность

Ошибки в аренах и layout-ах — теперь баги Java-кода

Но у медали две стороны: новые проблемы



Порог версии: нужен JDK 25+

есть свежая Java под ARM

Старые образы и дистрибутивы для плат отсекаются — не везде



Native access теперь явный

шаг настройки

--enable-native-access=... или манифест: безопасность, но ещё один



Memory-safety — наша ответственность

Ошибки в аренах и layout-ах — теперь баги Java-кода



Структуры ядра и версии поддерживаем сами

должны успевать за ABI

ядро эволюционирует — layout-ы в Java-коде

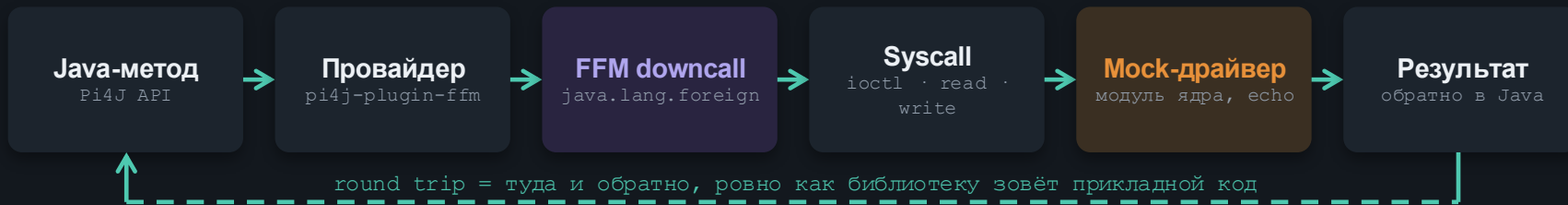
05

Бенчмарки: что дал переход

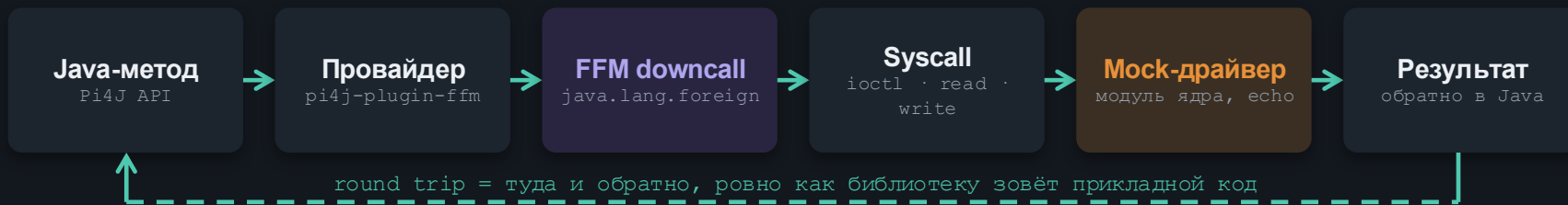
Прогонь на amd64, arm64 и riscv64. И объяснение, почему
цифры такие



Методика: что именно мы меряем



Методика: что именно мы меряем



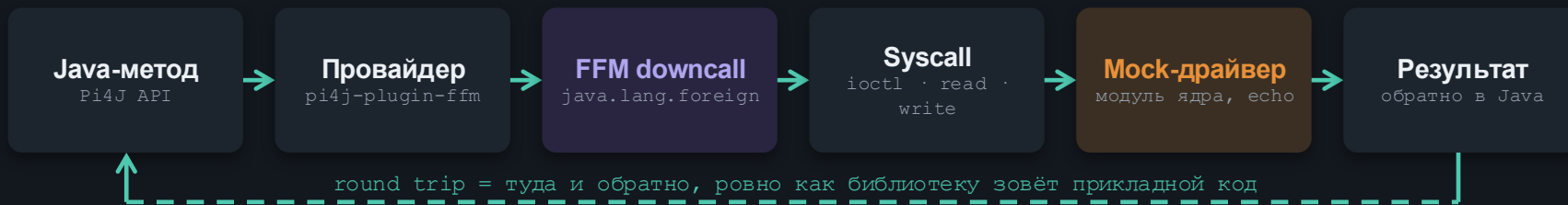
Mock-драйверы (echo): модули ядра из репозитория Pi4J — настоящие /dev/gpiochipN, i2c-dev, spidev, pwm с состоянием в памяти: что записали, то и читаем обратно. Измеряем мост, а не физику шины



Параметры запуска

JMH 1.37 · 3 fork · warmup 5 × 2 s · measure 5 × 2 s · AverageTime + SampleTime (p99 / p99.9)
 --enable-native-access=ALL-UNNAMED -Xms512m -Xmx512m -XX:+UseG1GC · taskset -c 2,3 ·
 governor=performance

Методика: что именно мы меряем



Mock-драйверы (echo): модули ядра из репозитория Pi4J — настоящие /dev/gpiochipN, i2c-dev, spidev, pwm с состоянием в памяти: что записали, то и читаем обратно. Измеряем мост, а не физику шины



Параметры запуска

`JMH 1.37 · 3 fork · warmup 5 × 2 s · measure 5 × 2 s · AverageTime + SampleTime (p99 / p99.9)`
`--enable-native-access=ALL-UNNAMED -Xms512m -Xmx512m -XX:+UseG1GC · taskset -c 2,3 ·`
`governor=performance`



Этим цифрам не стоит верить на слово. Другой SoC, ядро и JDK дадут другие числа — прогоните сами в репозитории



[github.com/
digitalsmile/
pi4j-bench](https://github.com/digitalsmile/pi4j-bench)

Стенд: три платформы, один jar

amd64

AMD Ryzen 7 5800X · 8 ядер · 32 ГБ

Linux 7.0 · Temurin JDK 25

arm64

Raspberry Pi 5 · BCM2712, 4× Cortex-A76 · 16 ГБ

Raspberry Pi OS, Linux 6.8 · Temurin JDK 25.0.4

riscv64

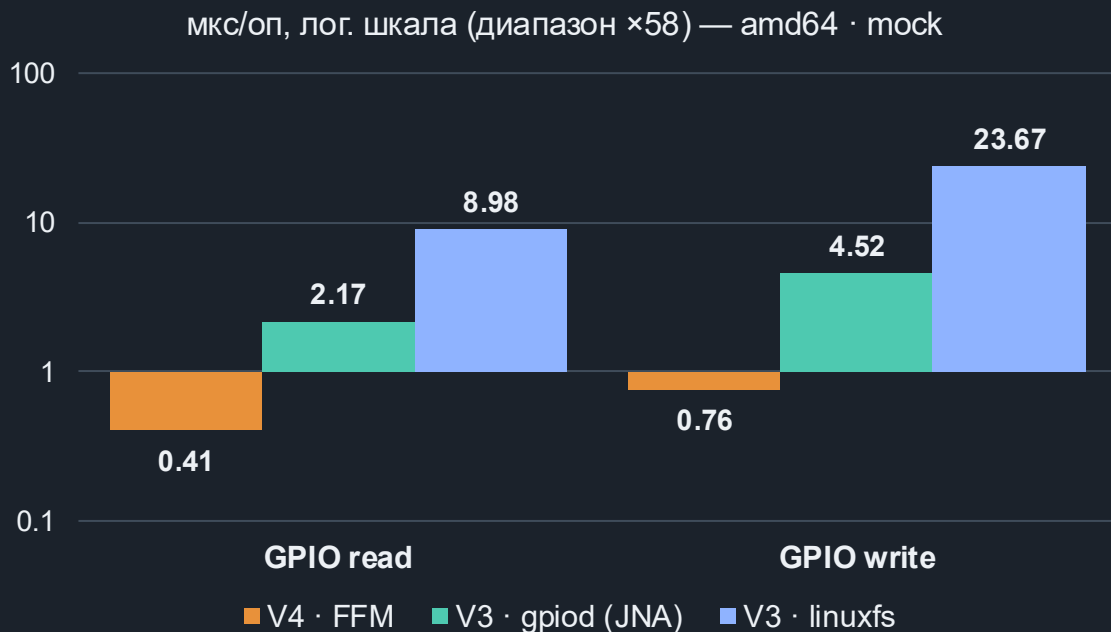
Milk-V Mars · StarFive JH7110 · 4 ядра · 8 ГБ

Ubuntu 24.04, Linux 6.17 · Temurin JDK 25.0.3

Один и тот же артефакт **Pi4J V4** на всех трёх — меняется только
плата и JDK под неё



GPIO: V4 против V3

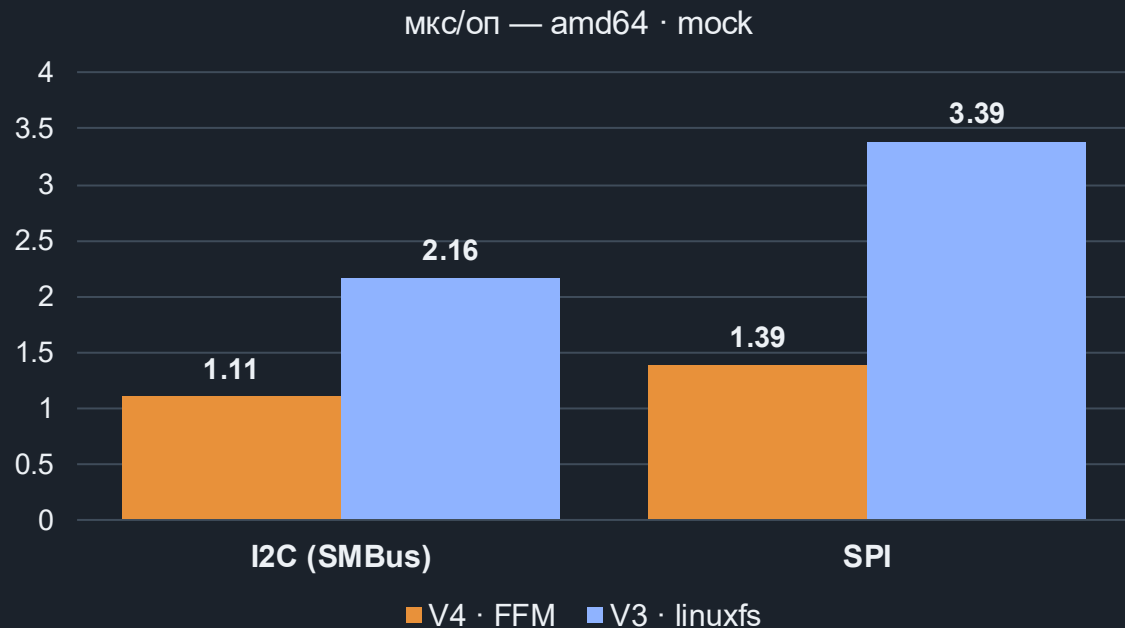


Итоги

- ×5,3 на чтении и ×6,0 на записи против gpiod
- ×22...31 против linuxfs (sysfs)

Самая дорогая прослойка ушла: один прогретый downcall в ioctl вместо JNA → JNI-обёртка → libgpiod

Шины I2C и SPI: V4 против V3

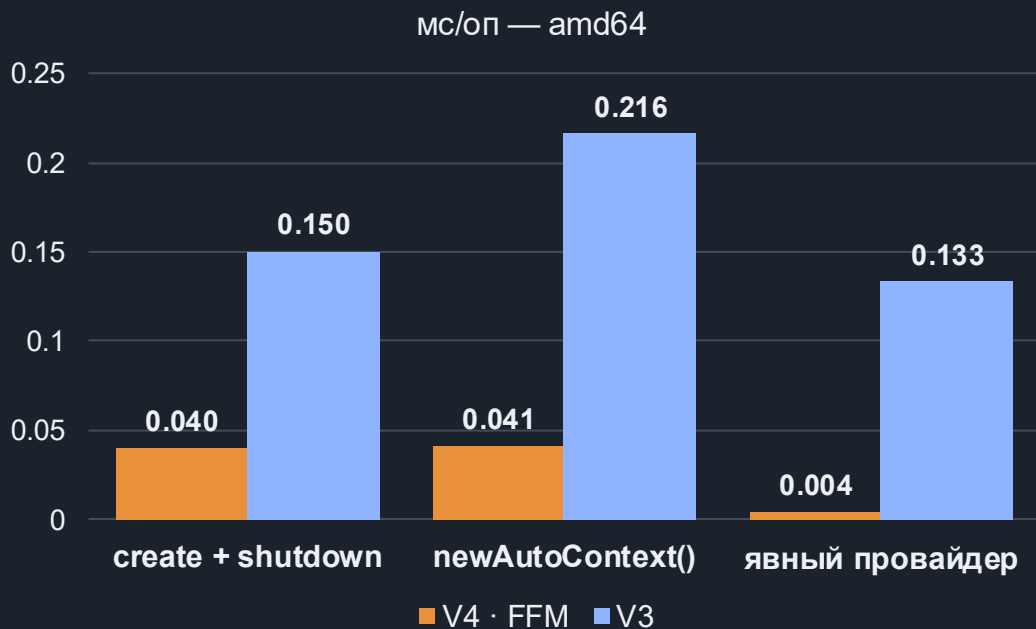


Итоги

- I2C $\times 2,0$, SPI $\times 2,4$
- Внутри V4: SMBus 1,1 · file 1,2 · direct 2,1 мкс
- На живой шине транзакция — сотни мкс: мост $< 1\%$

Здесь чистая физика: выигрыш V4 на шинах — в надёжности и поддержке, а не в микросекундах

Жизненный цикл



× 3,8

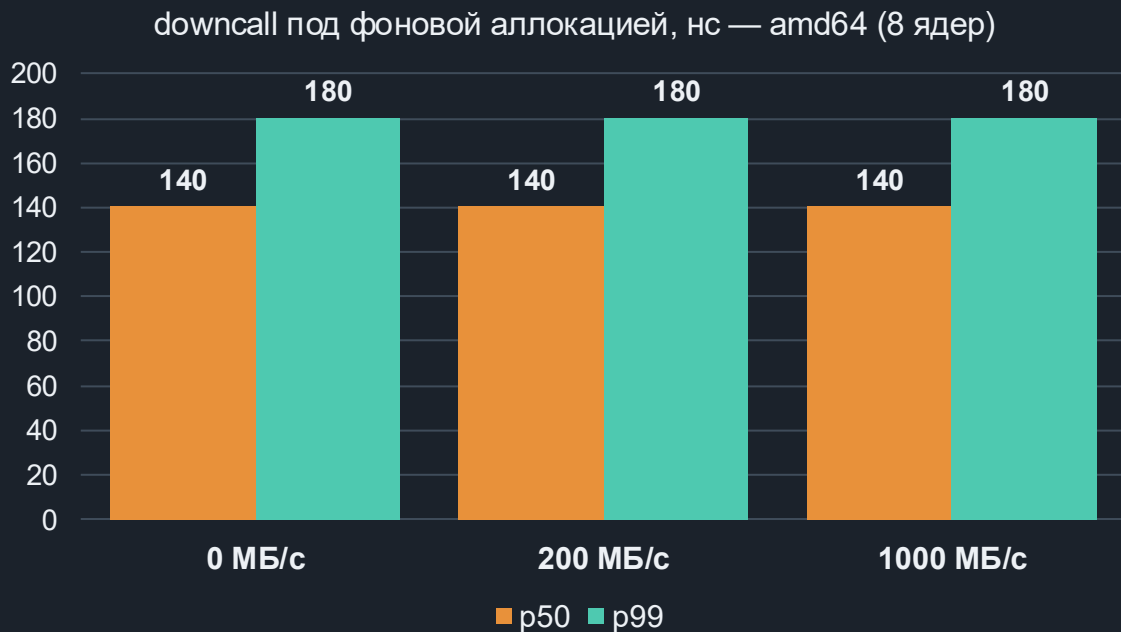
быстрее: создание + остановка провайдера

× 33

быстрее: контекст с явным провайдером

V4 быстрее: ранний тест из PR #458 давал 1,6 мс против 0,25 — инициализацию плагина с тех пор переписали

GC-давление

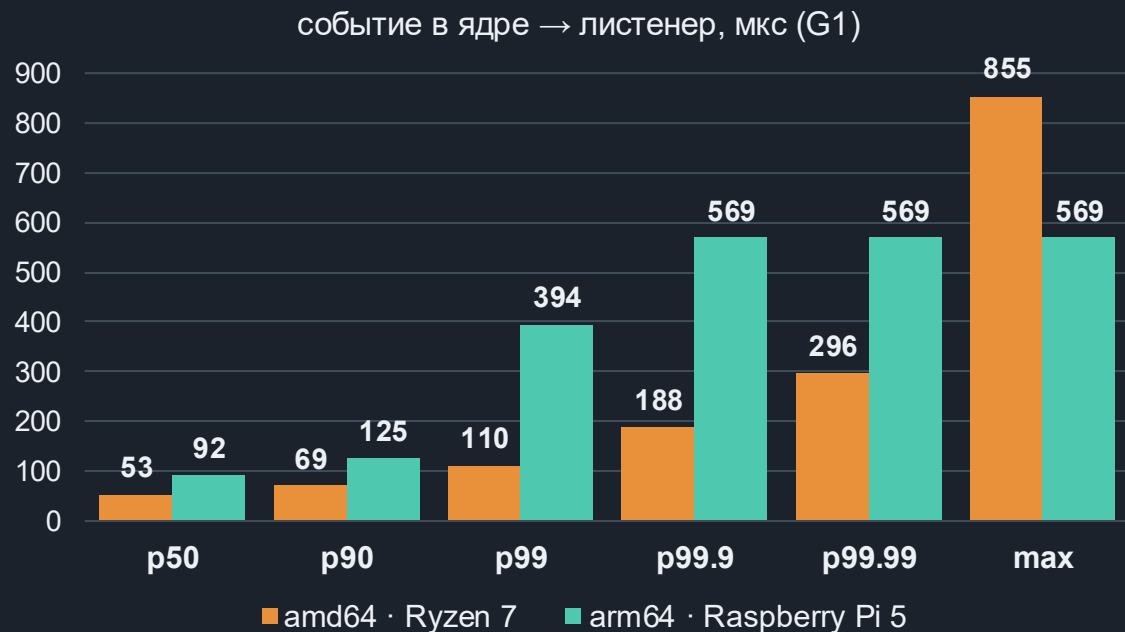


Итоги

- p50 140 нс — одинаково при 0, 200 и 1000 МБ/с
- p99 180 нс — тоже без изменений
- p99.9 $\approx$ 13 мкс есть даже без аллокаций (планировщик и safepoint)

FFM не добавляет чувствительности к GC: хвост принадлежит сборщику и планировщику, а не мосту

arm64: латентность событий на Raspberry Pi 5



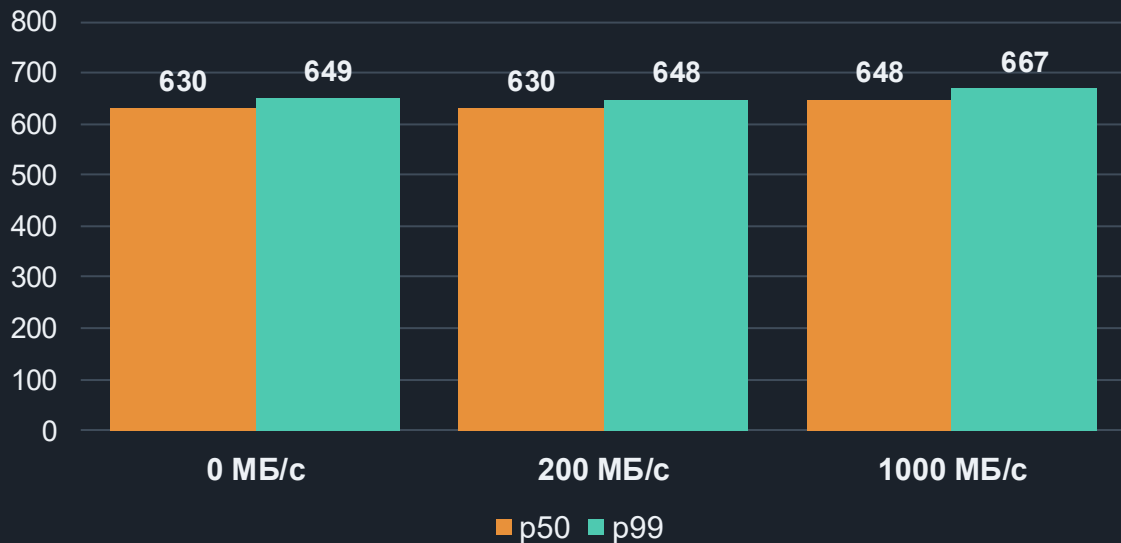
Итоги

- p50 92 мкс на Pi 5 против 53 на десктопе
- p99 394 мкс, max 569 мкс
- 600 событий на Pi 5 — p99.9 = max
- Ни одного пропущенного события

Soft real-time на JVM: миллисекунды — да, микросекунды — ядру или микроконтроллеру

arm64: мост, GC-давление и память

downcall под фоновой аллокацией, нс — Raspberry Pi 5 (4 ядра)

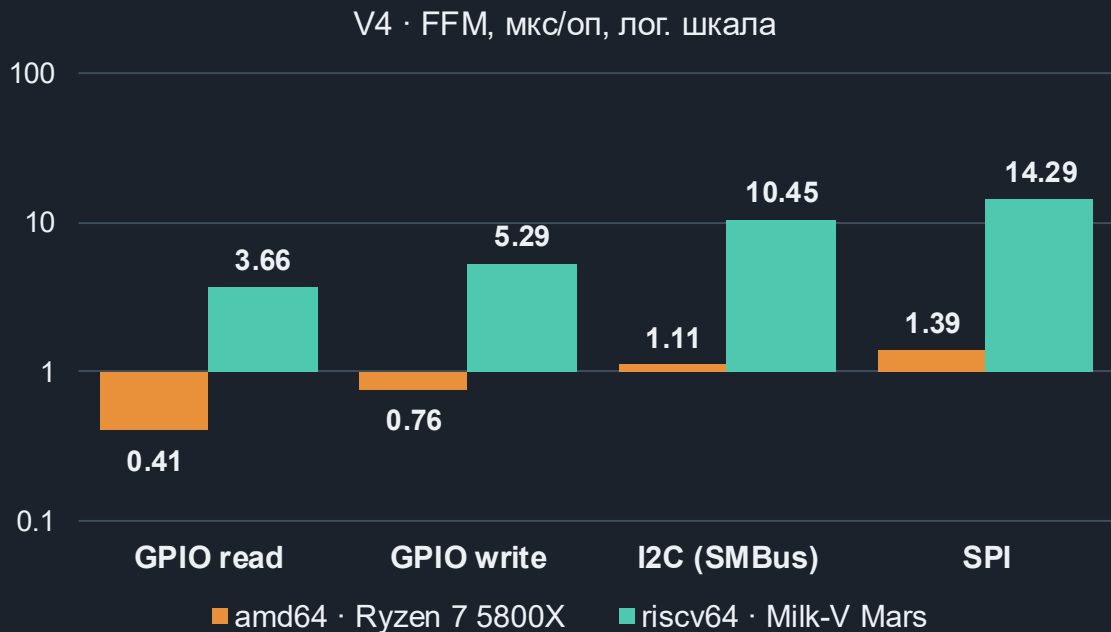


Итоги

- Мост 16–28 нс; syscall JNI 587 ≈ FFM 591; JNA +18 %
- Копия 64 Б: FFM 15 против 50 (JNI) и 68 (JNA) нс
- VarHandle ≈ ByteBuffer ≈ byte[] (0,55 / 0,56 / 0,51 мкс)

4 ядра, но давление не видно — JIT и планировщик держат p99 на месте

riscv64: V4 на Milk-V Mars



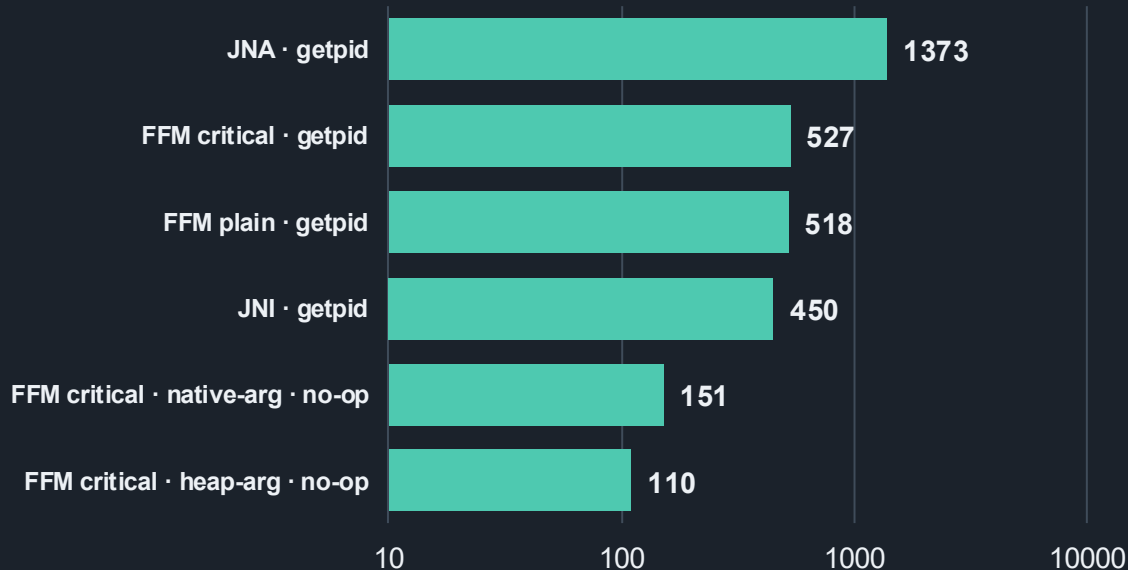
Итоги

- ×9...×10 медленнее десктопа
- Тот же яг, что на amd64 — без сборки натива

RISC-V из коробки: тот же артефакт Pi4J V4, что на x86 — ни одной нативной сборки под новую архитектуру

riscv64: цена моста

downcall, нс (лог. шкала) — riscv64

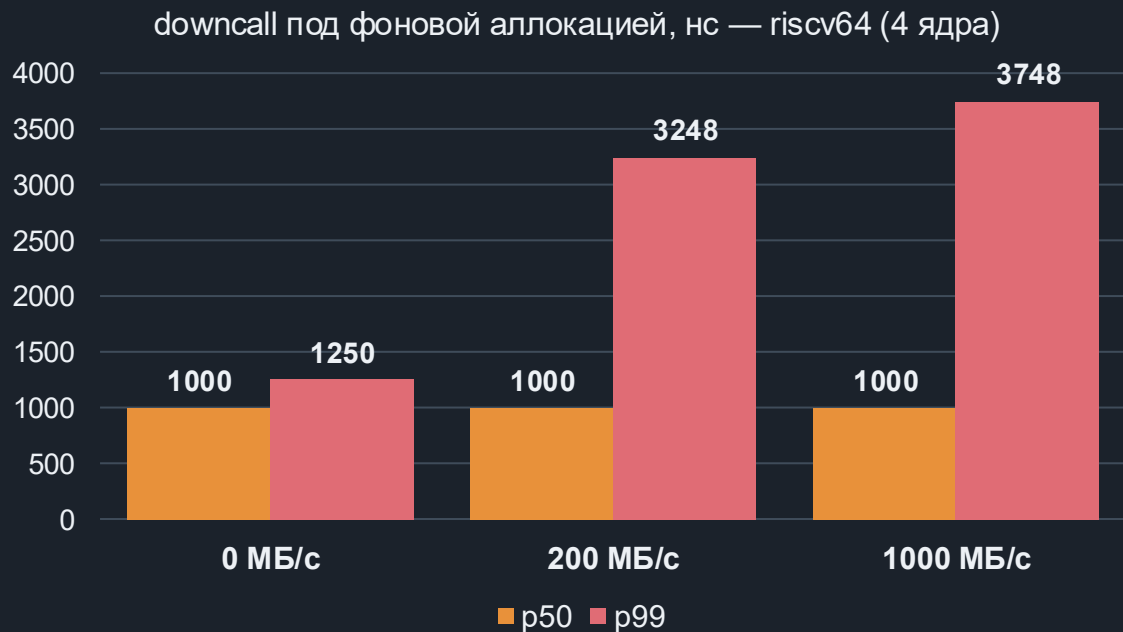


Итоги

- Мост 110–151 нс против 4–6 на amd64
- JNI 450 $\approx$ FFM 518 нс; JNA 1373 — $\times 2,7$
- Копия 64 Б: FFM 79 против JNI 346 и JNA 466 нс
- На 1 МиБ все $\approx$ 0,7 мс — память платы

Соотношения те же, что на x86: JNI $\approx$ FFM, JNA заметно позади, копирование у FFM быстрее всех

riscv64: GC-давление и доступ к памяти

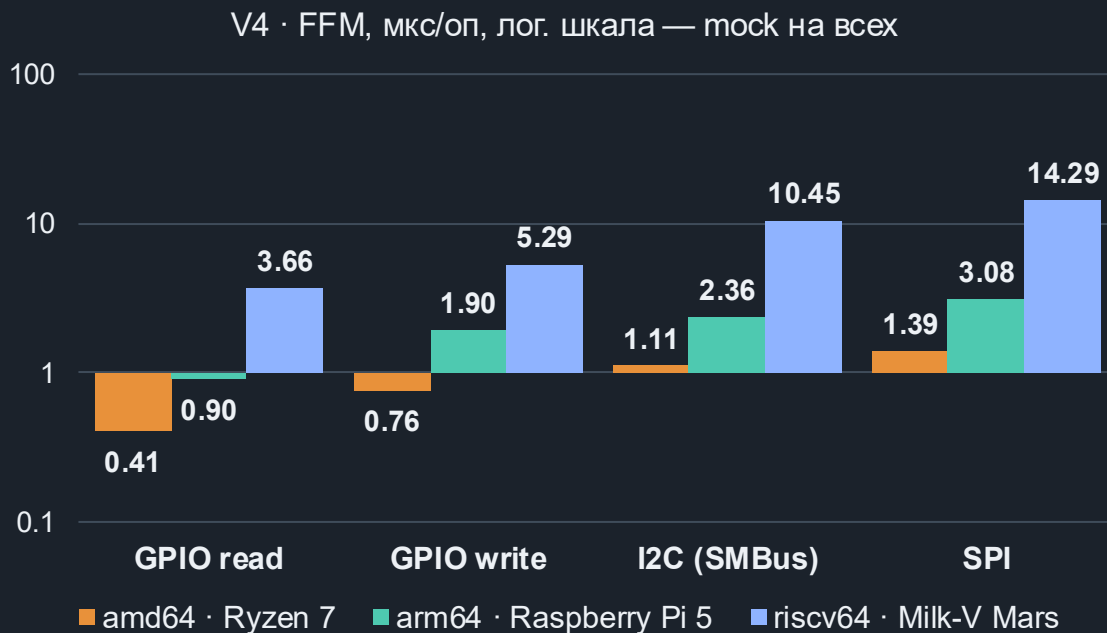


Итоги

- p50 1000 нс стабилен; p99 1250 → 3748 нс
- 4 ядра: аллокаторы отбирают CPU — планировщик, не мост
- ByteBuffer-цикл 29 мкс против VarHandle 1,4 — ×20: порту не хватает интринсиков

На слабой плате давление видно: p99 ×3 при 1000 МБ/с — планируйте потоки аллокации, а VarHandle держите static final

V4 на трёх платформах: amd64, arm64, riscv64



Итоги

- Pi 5: $\times 2,2 \dots \times 2,2$ к десктопу — форма та же
- Milk-V: $\times 9 \dots \times 10$ — JIT моложе

Один артефакт — три архитектуры: плата меняет абсолютные цифры, но не соотношения между интерфейсами

Выводы



Работает на любом железе, где есть Linux и Java

Один и тот же jar на amd64, arm64 и riscv64 — без нативной сборки

Выводы



Работает на любом железе, где есть Linux и Java

Один и тот же jar на amd64, arm64 и riscv64 — без нативной сборки



Новая реализация быстрее предыдущей

V4 против V3: GPIO $\times 5,3 \dots \times 6,0$ · I2C $\times 2,0$ · SPI $\times 2,4$ · жизненный цикл $\times 3,8 \dots \times 33$

Выводы



Работает на любом железе, где есть Linux и Java

Один и тот же jar на amd64, arm64 и riscv64 — без нативной сборки



Новая реализация быстрее предыдущей

V4 против V3: GPIO $\times 5,3 \dots \times 6,0$ · I2C $\times 2,0$ · SPI $\times 2,4$ · жизненный цикл $\times 3,8 \dots \times 33$



Небольшие миллисекундные задержки работают хорошо

Raspberry Pi 5: p99 394 мкс, max 569 мкс — ни одного пропущенного события

Выводы



Работает на любом железе, где есть Linux и Java

Один и тот же jar на amd64, arm64 и riscv64 — без нативной сборки



Новая реализация быстрее предыдущей

V4 против V3: GPIO $\times 5,3 \dots \times 6,0$ · I2C $\times 2,0$ · SPI $\times 2,4$ · жизненный цикл $\times 3,8 \dots \times 33$



Небольшие миллисекундные задержки работают хорошо

Raspberry Pi 5: p99 394 мкс, max 569 мкс — ни одного пропущенного события



Давление GC мало влияет на производительность

amd64 и arm64: p50 и p99 на месте при аллокации до 1000 МБ/с; на riscv64 растёт только p99

Выводы



Работает на любом железе, где есть Linux и Java

Один и тот же jar на amd64, arm64 и riscv64 — без нативной сборки



Новая реализация быстрее предыдущей

V4 против V3: GPIO $\times 5,3 \dots \times 6,0$ · I2C $\times 2,0$ · SPI $\times 2,4$ · жизненный цикл $\times 3,8 \dots \times 33$



Небольшие миллисекундные задержки работают хорошо

Raspberry Pi 5: p99 394 мкс, max 569 мкс — ни одного пропущенного события



Давление GC мало влияет на производительность

amd64 и arm64: p50 и p99 на месте при аллокации до 1000 МБ/с; на riscv64 растёт только p99



Скорость FFM $\approx$ JNI, а ускорение — от устранения промежуточных слоёв

Pi 5: syscall JNI 587 $\approx$ FFM 591 нс; вместо JNA $\rightarrow$ JNI-обёртка $\rightarrow$ libgpiod — один downcall в ioctl



06

Примеры и демо

Код с pi4j.com, стенд Elecrow CrowPi и демки на реальном железе

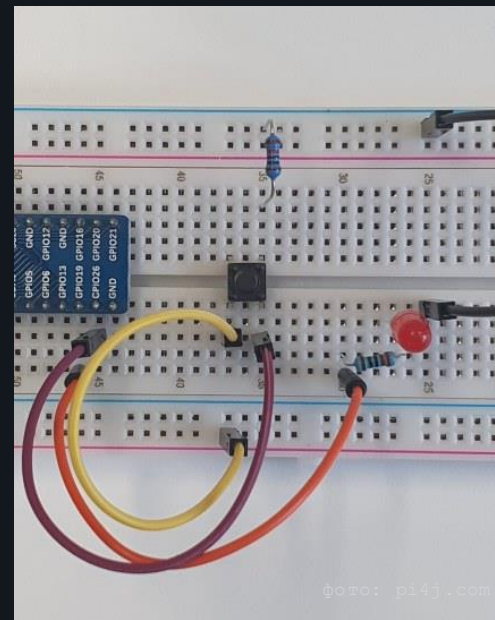
Пример 1: LED и кнопка

MinimalExample.java — pi4j.com/getting-started

```
var pi4j = Pi4J.newAutoContext();           // подхватит pi4j-plugin-ffm
var led = pi4j.digitalOutput().create(22); // PIN 15 = BCM 22

var cfg = DigitalInput.newConfigBuilder(pi4j)
    .id("button").name("Press button").bcm(24)
    .pull(PullResistance.PULL_DOWN).debounce(3_000L);
var button = pi4j.create(cfg);

button.addListener(e -> {                    // poll() в ядре
    if (e.state() == DigitalState.LOW) led.toggle();
});
```



LED

минимальный create по номеру линии



Кнопка

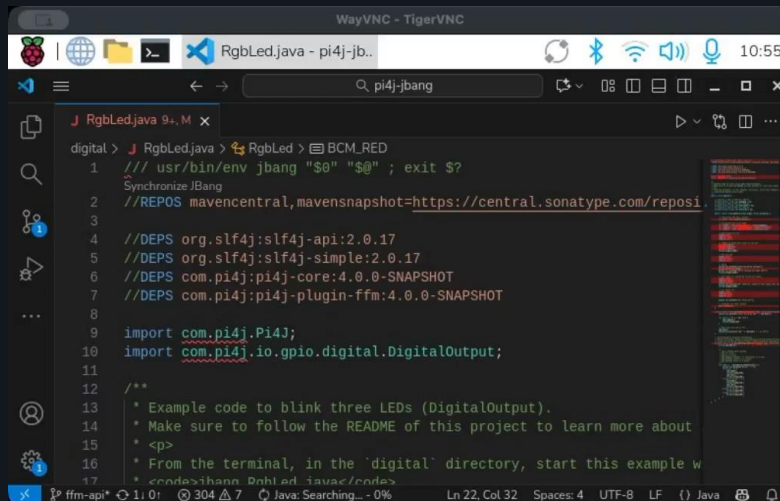
builder: pull, debounce, id



Листенер

poll() в ядре → callback в Java

Видео: азбука морзе



```
WayVNC - TigerVNC
RgbLed.java - pi4j-jb...
10:55

digital > j RgbLed.java > RgbLed > BCM_RED
1  /// usr/bin/env jbang "$@" && exit $?
   Synchronize JBang
2  //REPOS mavencentral,mavensnapshot=https://central.sonatype.com/reposi...
3
4  //DEPS org.slf4j:slf4j-api:2.0.17
5  //DEPS org.slf4j:slf4j-simple:2.0.17
6  //DEPS com.pi4j:pi4j-core:4.0.0-SNAPSHOT
7  //DEPS com.pi4j:pi4j-plugin-ffm:4.0.0-SNAPSHOT
8
9  import com.pi4j.Pi4j;
10 import com.pi4j.io.gpio.digital.DigitalOutput;
11
12 /**
13  * Example code to blink three LEDs (DigitalOutput).
14  * Make sure to follow the README of this project to learn more about
15  * <p>
16  * From the terminal, in the 'digital' directory, start this example w...
17  * <code>shang Dnblid javac/codes</code>
18  */
```



Пример 2: I2C и SPI — датчик и LED-матрица

I2C — pi4j.com/documentation/providers/ffm

```
var cfg = I2C.newConfigBuilder(pi4j)
    .bus(1).device(0x76)
    .i2cImplementation(I2CImplementation.SMBUS)
    .build(); // DIRECT | FILE | SMBUS
var bme280 = pi4j.create(cfg);
int id = bme280.readRegister(0xD0); // 0x60
```

SPI · MAX7219 — pi4j.com/documentation/io-types/spi

```
var cfg = Spi.newConfigBuilder(pi4j).id("matrix")
    .bus(0).chipSelect(0).baud(1_000_000)
    .build();
var spi = pi4j.create(cfg);
spi.write(CMD_SHUTDOWN, (byte) 0x01); // вкл
spi.write((byte) (ROW0 + row), buffer[row]);
```

Пример 2: I2C и SPI — датчик и LED-матрица

I2C — pi4j.com/documentation/providers/ffm

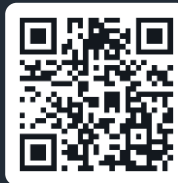
```
var cfg = I2C.newConfigBuilder(pi4j)
    .bus(1).device(0x76)
    .i2cImplementation(I2CImplementation.SMBUS)
    .build(); // DIRECT | FILE | SMBUS
var bme280 = pi4j.create(cfg);
int id = bme280.readRegister(0xD0); // 0x60
```

SPI · MAX7219 — pi4j.com/documentation/io-types/spi

```
var cfg = Spi.newConfigBuilder(pi4j).id("matrix")
    .bus(0).chipSelect(0).baud(1_000_000)
    .build();
var spi = pi4j.create(cfg);
spi.write(CMD_SHUTDOWN, (byte) 0x01); // вкл
spi.write((byte) (ROW0 + row), buffer[row]);
```



Drivers Library (2026): готовые драйверы датчиков и дисплеев поверх того же API — BME280, Sense HAT, MAX7219; плюс IOChecker (jbang) для проверки, что I2C/SPI включены на плате



[github.com/
Pi4J/
pi4j-drivers](https://github.com/Pi4J/pi4j-drivers)

pi4j.com/drivers · pi4j.com/examples/jbang · github.com/pi4j/pi4j-os (IOChecker)



Pi4J в бою: уже не только хобби



Soft real-time PLC на Stolch

Промышленный ПЛК на Java поверх Raspberry Pi: stolch.li использует Pi4J как слой доступа к железу в коммерческих инсталляциях



SmartCoop — серийный продукт

Автоматизация птичников от ex-инженера электроники RAAF: I2C, SPI, Serial; продаётся как готовое устройство (smartcoop.tech)



CocktailPi

Зрелый open source продукт для коктейль-машин с большим сообществом; его автор создал GpioD-провайдер для Pi4J и RPi 5



Pi4J в бою: уже не только хобби



Soft real-time PLC на Stolch

Промышленный ПЛК на Java поверх Raspberry Pi: stolch.li использует Pi4J как слой доступа к железу в коммерческих инсталляциях



SmartCoop — серийный продукт

Автоматизация птичников от ex-инженера электроники RAAF: I2C, SPI, Serial; продаётся как готовое устройство (smartcoop.tech)



CocktailPi

Зрелый open source продукт для коктейль-машин с большим сообществом; его автор создал GpioD-провайдер для Pi4J и RPi 5

Февраль 2026: Pi4J вступил в Commonhaus Foundation —
юридическая база и курс на широкое применение

Итоги: что забрать с собой



JNI/JNA работали, но цена — C-код, сборки и краши JVM на каждом чихе

Итоги: что забрать с собой



JNI/JNA работали, но цена — C-код, сборки и краши JVM на каждом чихе



FFM = граница «Java ↔ C ABI»: чистые Java-абстракции, безопасность, скорость JNI

Итоги: что забрать с собой



JNI/JNA работали, но цена — C-код, сборки и краши JVM на каждом чихе



FFM = граница «Java ↔ C ABI»: чистые Java-абстракции, безопасность, скорость JNI



Тайминги: soft real-time — да, hard real-time — отдаём ядру и микроконтроллерам

Итоги: что забрать с собой



JNI/JNA работали, но цена — C-код, сборки и краши JVM на каждом чихе



FFM = граница «Java ↔ C ABI»: чистые Java-абстракции, безопасность, скорость JNI



Тайминги: soft real-time — да, hard real-time — отдаём ядру и микроконтроллерам



Память: выбирайте арену осознанно — правильный выбор спасает от утечек памяти

Итоги: что забрать с собой



JNI/JNA работали, но цена — C-код, сборки и краши JVM на каждом чихе



FFM = граница «Java ↔ C ABI»: чистые Java-абстракции, безопасность, скорость JNI



Тайминги: soft real-time — да, hard real-time — отдаём ядру и микроконтроллерам



Память: выбирайте арену осознанно — правильный выбор спасает от утечек памяти



Pi4J v4: минус C-код и Docker-сборки — работает на любом Linux, даже с RISC-V

Пробуйте FFM с железом и ничего не бойтесь!

Вопросы?

Николай Гриценко

Ведущий менеджер MWS Cloud и немножко программист



github.com/Pi4J/pi4j
исходники Pi4J v4 и FFM-плагина



pi4j.com
документация и гайды



github.com/digitalsmile/pi4j-bench
бенчмарки — прогоните сами



LinkedIn
[in/nick-gritsenko](https://in.linkedin.com/in/nick-gritsenko)