МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ РАБОТАМ

«Архитектура ЭВМ и микропроцессорная техника»

Направление подготовки 09.03.04 «Программная инженерия»
Направленность (профиль) «Разработка и сопровождение программного обеспечения»
Квалификация выпускника: бакалавр

Ставрополь
2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящая методическая разработка предназначена для проведения лабораторных работ по дисциплине «Архитектура ЭВМ и микропроцессорная техника» для студентов направления 09.03.04 «Программная инженерия». Дисциплина направлена на формирование у студентов фундаментальных знаний об устройстве и принципах функционирования современных вычислительных систем, а также практических навыков низкоуровневого программирования и разработки программного обеспечения, взаимодействующего с аппаратными средствами.

Цель дисциплины является формирование у студентов теоретических знаний и практических навыков в области архитектуры вычислительных систем, низкоуровневого программирования и разработки программного обеспечения для управления аппаратными средствами, необходимых для профессиональной деятельности в области программной инженерии.

В ходе освоения курса студенты должны:

– изучить основные архитектурные концепции (фон-неймановская и гарвардская архитектуры, RISC/CISC, системы команд, способы адресации);

– освоить представление и обработку числовой информации в ЭВМ (системы счисления, форматы целых и вещественных чисел, побитовые операции);

– получить практические навыки программирования микроконтроллеров на языке С с элементами ассемблера;

– научиться настраивать и использовать периферийные модули (GPIO, таймеры, АЦП, последовательные интерфейсы UART, I2C, SPI, DMA);

– овладеть методами обработки прерываний, построения конечных автоматов и простейших планировщиков задач;

– приобрести опыт отладки встраиваемых систем с использованием JTAG/SWD, полухостинга, SWO-трассировки;

– понять принципы энергонезависимого хранения данных, износоустойчивости (wear-leveling) и эмуляции EEPROM во Flash-памяти.

Лабораторный практикум состоит из 18 работ, которые логически разделены на четыре тематических раздела, соответствующих программе курса.

Раздел 1. Основы архитектуры вычислительных систем

Работы 1–3 знакомят со средой кросс-компиляции, системой команд процессора и программной моделью портов ввода-вывода. Студенты учатся компилировать, компоновать и анализировать двоичные образы программ, исследовать ассемблерный код, а также реализовывать драйверы GPIO.

Раздел 2. Микропроцессорное ядро и программирование

Работы 4–7 охватывают таймеры и системное время, обработку прерываний, конечные автоматы для управления индикацией, а также программную генерацию ШИМ-сигналов. Эти работы дают понимание работы подсистемы прерываний и методов создания неблокирующих алгоритмов.

Раздел 3. Взаимодействие с периферией и драйверы

Работы 8–12 посвящены аналого-цифровому преобразованию, буферизации данных, протоколу UART и созданию командного интерпретатора, аппаратному и программному I2C, а также прямому доступу к памяти (DMA). Особое внимание уделяется производительности и надежности передачи данных.

Раздел 4. Системное ПО и современные тенденции

Работы 13–18 охватывают моделирование многозадачности (Round Robin, кооперативный и вытесняющий планировщики), эмуляцию EEPROM во Flash-памяти с wear-leveling, отладку через JTAG/SWD, разработку программного эмулятора периферии на ПК, создание логического анализатора на базе микроконтроллера и, наконец, комплексный проект «Программно-определяемая периферия».

Лабораторная работа №1

Среда разработки и процесс кросс-компиляции

Цель работы: освоение инструментария кросс-компиляции для ARM-микроконтроллеров, получение навыков компиляции, компоновки и анализа результирующих образов программ (hex/bin), а также изучение карты памяти и содержимого секций исполняемого файла.

Задачи

- Установить и настроить инструментарий GNU Toolchain для ARM (arm-none-eabi-gcc, binutils).
- Освоить написание Makefile для автоматизации сборки.
- Изучить этапы: препроцессинг, компиляция, ассемблирование, линковка.
- Получить выходные файлы в форматах ELF, HEX, BIN.
- Проанализировать карту памяти (map-файл) и содержимое секций (., .data, .bss, .rodata).
- Написать простейшую программу на C для целевой платформы (например, STM32F4Discovery или симулятор QEMU).

Теоретическая часть

Кросс-компиляция – это процесс компиляции программного кода для целевой платформы, архитектура или операционная система которой отличается от платформы, где выполняется компиляция. В случае микроконтроллеров ARM хост-системой обычно является x86-64 под Linux/Windows, а целевой – ARM Cortex-M без операционной системы (bare-metal).

Основные этапы сборки:

- **Препроцессинг** – обработка директив #include, #define, #ifdef.
- **Компиляция** – перевод C-кода в ассемблерный код для целевой архитектуры.

- **Ассемблирование** – преобразование ассемблера в объектный код (машинные инструкции и символы).
- **Линковка (компоновка)** – объединение объектных файлов и библиотек, разрешение символов, назначение окончательных адресов.

Форматы выходных файлов:

- **ELF (Executable and Linkable Format)** – содержит отладочную информацию, символы, секции; используется отладчиками.
- **HEX (Intel Hex)** – текстовое представление данных с адресами; загружается программаторами.
- **BIN** – чистый двоичный образ памяти; загружается по фиксированному адресу.

Секции памяти:

- . – код программы (обычно во Flash).
- .rodata – константы (только чтение).
- .data – инициализированные глобальные и статические переменные (копируются из Flash в RAM при старте).
- .bss – неинициализированные глобальные и статические переменные (обнуляются при старте).

Карта памяти (map-файл) – генерируется линковщиком и содержит:

- размещение каждой секции по абсолютным адресам,
- список входных объектных файлов,
- размеры секций,
- адреса глобальных символов.

Скрипт линковщика (linker script) – определяет расположение областей памяти (Flash, RAM), начальные адреса, выравнивание и правила размещения секций.

Методика выполнения работы

1. **Установка инструментария**

1.1. Установите компилятор arm-none-eabi-gcc и утилиты (binutils):

- Для Ubuntu/Debian: sudo apt install gcc-arm-none-eabi binutils-arm-none-eabi make
- Для Windows: установите GNU Arm Embedded Toolchain и добавьте в PATH.

1.2. Проверьте установку:

arm-none-eabi-gcc --version

arm-none-eabi-objdump --version

2. **Подготовка тестовой программы**

2.1. Создайте каталог lab1 и в нём файл main.c:

```
// Простейшая программа для bare-metal ARM
#define STACK_TOP 0x20020000   // вершина стека (для Cortex-M)

void reset_handler(void) __attribute__((naked));
void reset_handler(void) {
    __asm__ volatile (
        "ldr sp, =0x20020000\n"
        "bl main\n"
        "b .\n"
    );
}

int main(void) {
    volatile int x = 10;
    volatile int y = 20;
    volatile int z = x + y;
```

```
    return z;
}
```

2.2. Создайте простой скрипт линковщика link.ld:

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 1M
  RAM (rwx)  : ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS
{
  . : {
    *(.*)
    *(.rodata*)
  } > FLASH

  .data : {
    *(.data*)
  } > RAM AT> FLASH

  .bss : {
    *(.bss*)
    *(COMMON)
  } > RAM
}
```

3. Написание Makefile

3.1. Создайте файл Makefile:

```
TARGET = firmware
```

```
CC = arm-none-eabi-gcc
LD = arm-none-eabi-gcc
OBJCOPY = arm-none-eabi-objcopy
OBJDUMP = arm-none-eabi-objdump

CFLAGS = -mcpu=cortex-m4 -mthumb -Wall -O0 -g -ffreestanding -nostdlib
LDFLAGS = -T link.ld -Wl,-Map=$(TARGET).map,-cref -nostartfiles

OBJS = main.o

all: $(TARGET).elf $(TARGET).hex $(TARGET).bin

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

$(TARGET).elf: $(OBJS)
	$(LD) $(LDFLAGS) $^ -o $@

%.hex: %.elf
	$(OBJCOPY) -O ihex $< $@

%.bin: %.elf
	$(OBJCOPY) -O binary $< $@

clean:
	rm -f *.o *.elf *.hex *.bin *.map

list-sections:
	$(OBJDUMP) -h $(TARGET).elf
```

3.2. Запустите сборку командой make.

4. **Анализ результатов**

4.1. Изучите содержимое map-файла:

less firmware.map

Найдите расположение ., .data, .bss.

4.2. Посмотрите секции в ELF:

arm-none-eabi-objdump -h firmware.elf

4.3. Посмотрите дизассемблированный код:

arm-none-eabi-objdump -d firmware.elf

4.4. Сравните размеры .hex и .bin (откройте в текстовом редакторе). Объясните разницу.

5. **Модификация программы**

5.1. Добавьте глобальную инициализированную переменную int g_init = 42; и глобальную неинициализированную int g_bss;.

5.2. Повторно соберите проект и проследите, в какие секции попали переменные.

5.3. С помощью arm-none-eabi-nm firmware.elf выведите список символов и их адресов.

Пример выполнения задания

Входной код (main.c):

```
const char message[] = "Hello";
int data_val = 100;
int bss_val;

int main(void) {
```

```
    volatile int local = data_val + 5;
    return local;
}
```

Фрагмент карты памяти (firmware.map):

```
.          0x08000000     0x98
                0x08000000             main.o(.)
                0x08000048             main
.rodata        0x08000098       0x8
                0x08000098             message
.data          0x20000000       0x4
                0x20000000             data_val
.bss           0x20000004       0x4
                0x20000004             bss_val
```

Фрагмент дизассемблера:

```
08000048 <main>:
 8000048: b480      push {r7}
 800004a: 4b04      ldr r3, [pc, #16]
 800004c: 681b      ldr r3, [r3, #0]
 800004e: 3305      adds r3, #5
 8000050: 607b      str r3, [r7, #4]
 ...
```

Индивидуальные задания

Каждый студент получает вариант (по номеру в списке). Необходимо выполнить следующие действия:

Вариант 1. Модификация размера стека

- Измените значение STACK_TOP на 0x20001000.

- Соберите проект, проанализируйте, как изменился адрес переменных после линковки. Опишите возможные проблемы.

Вариант 2. Добавление константного массива

- Объявите const uint32_t table[256] = {0,1,2,3,...}; (заполните последовательно).
- Определите, в какую секцию он попал. Сравните размеры .bin и .hex.

Вариант 3. Использование атрибутов размещения

- Поместите функцию void delay(void) в отдельную секцию .slow_functions с помощью __attribute__((section(".slow_functions"))).
- Модифицируйте скрипт линковщика, чтобы эта секция располагалась в конце Flash.
- Покажите в отчёте изменения в map-файле.

Вариант 4. Глобальные массивы разного типа

- Объявите char big_buf1[1024] = {0}; и int big_buf2[256];.
- Проанализируйте, где располагаются эти массивы. Каков размер .bss?

Вариант 5. Использование ассемблерной вставки

- Добавьте в main ассемблерную вставку asm volatile("nop"); пять раз.
- Посмотрите, сколько байт инструкции NOP занимают в ..

Вариант 6. Принудительное выравнивание

- Объявите char small[1]; и следом int align_var __attribute__((aligned(16)));.
- Проверьте адрес align_var в map-файле. Объясните, зачем нужно выравнивание.

Вариант 7. Удаление стандартных секций

- Добавьте в CFLAGS -fdata-sections -ffunction-sections, а в LDFLAGS --gc-sections.
- Проанализируйте, изменился ли размер .. Какие неиспользуемые функции удалились?

Вариант 8. Размещение данных во внешней памяти

- Измените скрипт линковщика, добавив область EXTRAM (rw) : ORIGIN = 0x68000000, LENGTH = 64K.
- Поместите int slow_data[100] в секцию .extram с помощью __attribute__((section(".extram"))).
- Убедитесь, что адреса попали в область EXTRAM.

Вариант 9. Формирование бинарного файла с кастомным fill-значением

- В линкер-скрипте добавьте в секцию . FILL(0xFFFFFFFF).
- Сравните дамп .bin до и после изменения. Объясните, для чего используется FILL.

Вариант 10. Анализ кросс-ссылок (cross-reference)

- Создайте два файла: a.c со int a_func(void) { return 1; } и b.c, который вызывает a_func.
- С помощью arm-none-eabi-objdump -x найдите, как линковщик разрешает символы.

Вариант 11. Эмуляция в QEMU

- Настройте QEMU для Cortex-M3: qemu-system-arm -machine lm3s6965evb -kernel firmware.elf -nographic.
- Убедитесь, что программа выполняется и вызывает main. Добавьте в код бесконечный цикл.

Вариант 12. Секция инициализации (.init_array)

- Добавьте конструктор с __attribute__((constructor)) void ctor(void) { ... }.
- Изучите map-файл: появилась секция .init_array. Объясните её назначение.

Вариант 13. Анализ времени загрузки (load)

- Создайте .data размером 8 КБ: int huge_data[2048] = {42};.
- Вручную в map-файле найдите LOAD ADDRESS для .data. Вычислите объем копирования при старте.

Вариант 14. Использование weak-символов

- Определите void __attribute__((weak)) weak_func(void) { /* default */ }.

- Определите weak_func в другом файле с реализацией. Посмотрите, какой символ выберет линковщик.

Вариант 15. Формат Intel HEX

- Напишите скрипт на Python или bash, который читает .hex файл, извлекает адреса и данные, и восстанавливает .bin.
- Сравните восстановленный .bin с оригинальным.

Требования к отчёту

Отчёт должен быть оформлен в текстовом или PDF-формате и содержать:

1. Титульный лист с названием работы, ФИО студента, номером группы.
2. Цель работы и поставленные задачи.
3. Краткое теоретическое введение (своими словами, 0.5–1 страница).
4. Ход выполнения работы с указанием всех команд, Makefile (в виде листинга), скрипта линковщика.
5. Анализ полученных файлов (ELF, HEX, BIN, MAP) с пояснениями:
   - размер секций,
   - адреса загрузки и выполнения,
   - содержимое карты памяти.
6. Результаты выполнения индивидуального задания (код, map-файл или дамп, объяснение).
7. Выводы по работе (что нового освоено, какие возникли сложности, какие практические применения имеет умение анализировать карту памяти).
8. Ответы на контрольные вопросы.

Контрольные вопросы

1. Почему для микроконтроллеров используется кросс-компиляция, а не нативная?

2. Чем отличается ELF-файл от BIN-файла? Когда используется каждый формат?

3. Что такое карта памяти (map-файл) и для каких целей её анализируют?

4. Какие секции памяти присутствуют в типичной программе для bare-metal? Где они физически располагаются во Flash и RAM?

5. Зачем нужен скрипт линковщика? Что произойдёт, если его не указать?

6. Объясните разницу между AT> FLASH в описании секции .data и просто > RAM. Что такое LMA и VMA?

7. Почему в программе для микроконтроллера необходимо вручную настраивать указатель стека?

8. Как с помощью objdump просмотреть только ассемблерный код функции main?

9. Что покажет команда arm-none-eabi-size firmware.elf? Какие числа она выводит?

10. В чём преимущество генерации .hex перед .bin при программировании через загрузчик?

11. Как проверить, что глобальная неинициализированная переменная попала в .bss, а не в .data?

12. Как можно разместить критическую функцию в оперативной памяти для увеличения скорости выполнения?

13. Что такое выравнивание (alignment) и почему компилятор может добавить байты-заполнители между переменными?

14. Для чего используется опция -nostdlib? Какие стандартные функции становятся недоступны?

15. Как сгенерировать файл только с отладочной информацией (без кода)?

Лабораторная работа №2
Исследование системы команд микропроцессора на уровне С и ассемблера

Цель работы: освоение практических приёмов написания ассемблерных вставок в коде на языке С для архитектуры ARM, анализ машинного кода, генерируемого компилятором, и сравнение эффективности различных способов реализации арифметических и логических операций.

Задачи:

- Изучить синтаксис ассемблерных вставок для компилятора GCC (ARM).
- Научиться использовать регистры общего назначения и операнды (входные, выходные, разрушаемые).
- Реализовать базовые арифметические операции на ассемблере (сложение, умножение, сдвиги).
- Провести дизассемблирование С-кода и анализ инструкций, генерируемых компилятором.
- Сравнить производительность С-кода и ручной ассемблерной оптимизации.
- Изучить влияние оптимизаций компилятора (-O0, -O1, -O2, -O3, -Os) на результирующий код.

Теоретическая часть

Ассемблерные вставки в GCC (Extended Asm)

GCC предоставляет механизм встраивания ассемблерного кода непосредственно в С-программы. Общий синтаксис:

```
__asm__ volatile (
    "инструкция1\n\t"
    "инструкция2\n\t"
    : выходные_операнды
    : входные_операнды
```

```
    : разрушаемые_регистры
);
```

Спецификаторы операндов:

- "r" – любой регистр общего назначения
- "l" – конкретный регистр (например, "r0")
- "m" – операнд в памяти
- "I" – непосредственная константа (диапазон зависит от архитектуры)
- "=r" – выходной операнд (write-only)
- "+r" – операнд, который читается и изменяется

Связывание переменных с регистрами:

```
int a = 10, b = 20, result;
__asm__ (
    "add %0, %1, %2"
    : "=r" (result)
    : "r" (a), "r" (b)
);
```

Ключевое слово volatile – предотвращает оптимизацию и перемещение ассемблерной вставки компилятором.

Система команд ARM (Thumb-2 для Cortex-M)

Основные классы инструкций:

- **Арифметические:** ADD, SUB, MUL, UDIV, SDIV
- **Логические:** AND, ORR, EOR, BIC
- **Сдвиги:** LSL, LSR, ASR, ROR
- **Пересылки:** MOV, MVN, LDR, STR
- **Ветвления:** B, BL, BX, CBZ, CBNZ
- **Сравнения:** CMP, CMN, TST, TEQ

Дизассемблирование

Утилита objdump позволяет получить ассемблерный код из ELF-файла:

bash

arm-none-eabi-objdump -d firmware.elf

arm-none-eabi-objdump -S firmware.elf *# с перемежением исходного C-кода*

Влияние уровней оптимизации:

- -O0 – без оптимизации, максимальная отладочная информация
- -O1 – базовая оптимизация, баланс скорости и размера
- -O2 – агрессивная оптимизация, рекомендуется для production
- -O3 – максимальная оптимизация скорости (может увеличить размер)
- -Os – оптимизация по размеру кода

Затраты циклов на арифметические операции

Для Cortex-M4 (с плавающей точкой FPv4-SP):

- ADD/SUB – 1 цикл
- MUL (32×32→32) – 1 цикл
- UMULL/SMULL (32×32→64) – 1–2 цикла
- UDIV/SDIV – 2–12 циклов (зависит от значений)
- VLDR/VSTR (вещественные) – 2 цикла
- VADD.F32 – 1 цикл
- VMUL.F32 – 1 цикл

Методика выполнения работы

1. **Подготовка тестовой программы**

   1.1. Создайте каталог lab2 и файл arith_test.c:

   #include <stdint.h>

   #include <stdio.h>

```
#define ITERATIONS 1000000

// Функция на чистом C
uint32_t add_c(uint32_t a, uint32_t b) {
    return a + b;
}

// Функция с ассемблерной вставкой
uint32_t add_asm(uint32_t a, uint32_t b) {
    uint32_t result;
    __asm__ volatile (
        "add %0, %1, %2"
        : "=r" (result)
        : "r" (a), "r" (b)
    );
    return result;
}

// Сложная операция: умножение с накоплением
uint32_t mac_c(uint32_t acc, uint32_t a, uint32_t b) {
    return acc + (a * b);
}

uint32_t mac_asm(uint32_t acc, uint32_t a, uint32_t b) {
    uint32_t result;
    __asm__ volatile (
        "mul %0, %2, %3\n\t"
        "add %0, %0, %1"
        : "=&r" (result)
        : "r" (acc), "r" (a), "r" (b)
```

```
        : "cc"
    );
    return result;
}

// Циклическая свертка (для измерения производительности)
uint32_t convolution_c(uint32_t* data, uint32_t* coeff, int len) {
    uint32_t sum = 0;
    for (int i = 0; i < len; i++) {
        sum += data[i] * coeff[i];
    }
    return sum;
}

uint32_t convolution_asm(uint32_t* data, uint32_t* coeff, int len) {
    uint32_t sum = 0;
    __asm__ volatile (
        "mov r4, #0\n"          // sum = 0
        "mov r5, %2\n"          // счетчик
        "1:\n"
        "subs r5, r5, #1\n"     // len--
        "bmi 2f\n"              // если отрицательный - выход
        "ldr r6, [%0, r5, lsl #2]\n"  // data[i]
        "ldr r7, [%1, r5, lsl #2]\n"  // coeff[i]
        "mul r6, r6, r7\n"
        "add r4, r4, r6\n"
        "b 1b\n"
        "2:\n"
        "mov %0, r4\n"
        : "=r" (sum)
```

```
        : "r" (data), "r" (coeff), "r" (len)
        : "r4", "r5", "r6", "r7", "cc"
    );
    return sum;
}

int main(void) {
    uint32_t a = 12345, b = 67890;
    volatile uint32_t c_result, asm_result;

    c_result = add_c(a, b);
    asm_result = add_asm(a, b);

    printf("C: %u, ASM: %u\n", c_result, asm_result);

    // Тест производительности
    uint32_t data[100], coeff[100];
    for (int i = 0; i < 100; i++) {
        data[i] = i + 1;
        coeff[i] = 100 - i;
    }

    uint32_t conv_c = convolution_c(data, coeff, 100);
    uint32_t conv_asm = convolution_asm(data, coeff, 100);
    printf("Convolution C: %u, ASM: %u\n", conv_c, conv_asm);

    return 0;
}
```

1.2. Создайте Makefile для лабораторной работы:

makefile

```
TARGET = test_arith
CC = arm-none-eabi-gcc
OBJDUMP = arm-none-eabi-objdump
SIZE = arm-none-eabi-size

CFLAGS_COMMON = -mcpu=cortex-m4 -mthumb -mfloat-abi=soft -g
LDFLAGS = -T link.ld -nostdlib

all: $(TARGET)_O0.elf $(TARGET)_O1.elf $(TARGET)_O2.elf $(TARGET)_O3.elf $(TARGET)_Os.elf

$(TARGET)_O0.elf: arith_test.c
	$(CC) $(CFLAGS_COMMON) -O0 $(LDFLAGS) $< -o $@
	$(OBJDUMP) -S $@ > $(basename $@).s
	$(SIZE) $@

$(TARGET)_O1.elf: arith_test.c
	$(CC) $(CFLAGS_COMMON) -O1 $(LDFLAGS) $< -o $@
	$(OBJDUMP) -S $@ > $(basename $@).s

$(TARGET)_O2.elf: arith_test.c
	$(CC) $(CFLAGS_COMMON) -O2 $(LDFLAGS) $< -o $@
	$(OBJDUMP) -S $@ > $(basename $@).s

$(TARGET)_O3.elf: arith_test.c
	$(CC) $(CFLAGS_COMMON) -O3 $(LDFLAGS) $< -o $@
	$(OBJDUMP) -S $@ > $(basename $@).s

$(TARGET)_Os.elf: arith_test.c
	$(CC) $(CFLAGS_COMMON) -Os $(LDFLAGS) $< -o $@
```

```
	$(OBJDUMP) -S $@ > $(basename $@).s

clean:
	rm -f *.elf *.s *.o *.map

.PHONY: all clean
```

2. **Анализ ассемблерного кода C-функций**

2.1. Скомпилируйте программу с разными уровнями оптимизации:

make

2.2. Изучите сгенерированные .s файлы. Сравните код функции add_c при разных уровнях оптимизации.

2.3. Найдите в дизассемблированном коде:

- Пролог и эпилог функции (сохранение/восстановление регистров)
- Инструкцию ADD для сложения
- Инструкции пересылки из памяти в регистры

3. **Эксперимент с различными операциями**

3.1. Создайте файл operations.c:

c

```
#include <stdint.h>

// Сложение
uint32_t op_add(uint32_t a, uint32_t b) { return a + b; }

// Вычитание
uint32_t op_sub(uint32_t a, uint32_t b) { return a - b; }

// Умножение
uint32_t op_mul(uint32_t a, uint32_t b) { return a * b; }

// Деление
```

```
uint32_t op_div(uint32_t a, uint32_t b) { return a / b; }

// Левый сдвиг
uint32_t op_lsl(uint32_t a, uint32_t b) { return a << b; }

// Побитовое AND
uint32_t op_and(uint32_t a, uint32_t b) { return a & b; }

// Условный выбор (без ветвления)
uint32_t op_cond(int a, int b, int c) { return (a > b) ? c : 0; }

int main(void) { return 0; }
```

3.2. Скомпилируйте с -O2 и проанализируйте каждую функцию:

arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -O2 -c operations.c -o operations.o

arm-none-eabi-objdump -d operations.o

4. **Измерение производительности (эмуляция в QEMU)**

4.1. Модифицируйте основной файл для измерения тактов:

```
#include <stdint.h>

// Таймер SysTick для Cortex-M
void systick_init(void) {
    SysTick->LOAD = 0x00FFFFFF;  // максимальное значение
    SysTick->VAL = 0;
    SysTick->CTRL = 5;  // включить, с тактом процессора
}

uint32_t systick_get(void) {
    return SysTick->VAL;
}
```

```
uint32_t measure_add_c(uint32_t a, uint32_t b, int n) {
    uint32_t start = systick_get();
    for (int i = 0; i < n; i++) {
        volatile uint32_t x = add_c(a, b);
        (void)x;
    }
    uint32_t end = systick_get();
    return start - end;  // разница (чем больше - тем быстрее)
}
```

4.2. Запустите в QEMU и сравните количество тактов для C и ASM версий.

5. **Исследование логических и битовых операций**

5.1. Напишите функцию на ассемблере для проверки чётности числа:

```
int is_even_asm(int x) {
    int result;
    __asm__ volatile (
        "and %0, %1, #1\n\t"
        "cmp %0, #0\n\t"
        "moveq %0, #1\n\t"
        "movne %0, #0"
        : "=r" (result)
        : "r" (x)
        : "cc"
    );
    return result;
}
```

5.2. Напишите C-версию. Сравните код.

6. **Анализ работы с памятью (загрузка/сохранение)**

6.1. Создайте структуру и функции для работы с ней:

```
typedef struct {
    uint32_t x;
    uint32_t y;
    uint32_t z;
} point_t;

uint32_t sum_coords_c(point_t* p) {
    return p->x + p->y + p->z;
}

uint32_t sum_coords_asm(point_t* p) {
    uint32_t result;
    __asm__ volatile (
        "ldr r1, [%0]\n\t"      // p->x
        "ldr r2, [%0, #4]\n\t"   // p->y
        "ldr r3, [%0, #8]\n\t"   // p->z
        "add r1, r1, r2\n\t"
        "add %1, r1, r3"
        : "=r" (result)
        : "r" (p)
        : "r1", "r2", "r3"
    );
    return result;
}
```

6.2. Проанализируйте выравнивание и эффективность загрузки.

7. **Сравнение с плавающей точкой (Cortex-M4F)**

7.1. Добавьте операции с float:

c

```
float add_float_c(float a, float b) { return a + b; }
```

```
float add_float_asm(float a, float b) {
    float result;
    __asm__ volatile (
        "vadd.f32 %0, %1, %2"
        : "=t" (result)
        : "t" (a), "t" (b)
    );
    return result;
}
```

7.2. Скомпилируйте с -mcpu=cortex-m4 -mfpu=fpv4-sp-d16 -mfloat-abi=hard.

Пример выполнения задания

Исходная С-функция:

```
uint32_t multiply_add(uint32_t a, uint32_t b, uint32_t c) {
    return a * b + c;
}
```

Скомпилировано с -O0:

```
multiply_add:
    push {r7, lr}
    sub sp, sp, #12
    add r7, sp, #0
    str r0, [r7, #4]
    str r1, [r7, #0]
    str r2, [r7, #8]
    ldr r2, [r7, #4]
    ldr r3, [r7, #0]
    mul r2, r3, r2
    ldr r3, [r7, #8]
```

```
    add r3, r2, r3
    mov r0, r3
    add sp, r7, #12
    pop {r7, pc}
```

Скомпилировано с -O2:

assembly

```
multiply_add:
    mul r0, r1, r0
    add r0, r0, r2
    bx lr
```

Ассемблерная вставка (аналог):

```
uint32_t multiply_add_asm(uint32_t a, uint32_t b, uint32_t c) {
    __asm__ volatile (
        "mul %0, %1, %2\n\t"
        "add %0, %0, %3"
        : "=&r" (a)
        : "r" (a), "r" (b), "r" (c)
    );
    return a;
}
```

Ключевые наблюдения:

- -O0 генерирует много лишней работы с стеком
- -O2 помещает аргументы в регистры и не использует стек
- Компилятор генерирует идентичный код для простых операций
- Для сложных алгоритмов ручная оптимизация может дать выигрыш 10–30%

Индивидуальные задания

Вариант 1. Реализация насыщающего сложения

Напишите функцию насыщающего сложения для 16-битных чисел (результат не превышает 32767). Реализуйте на C и на ассемблере. Сравните код.

Вариант 2. Оптимизация CRC-8

Реализуйте вычисление CRC-8 на C и на ассемблере с использованием битовых операций. Сравните производительность.

Вариант 3. Поиск максимального элемента в массиве

Напишите функцию на ассемблере, которая находит максимум в массиве uint32_t. Сравните с оптимизированным C-кодом.

Вариант 4. Битовая реверсация

Реализуйте функцию, которая переворачивает порядок битов в 32-битном числе (mirror) – сначала на C, затем на ассемблере с использованием инструкции RBIT (если есть).

Вариант 5. Операция с тремя операндами

Реализуйте result = (a & b) | (~a & c) на ассемблере. Сравните с C-версией. Это аналог операции "выбор бита" (bit select).

Вариант 6. Умножение 64×64 → 128 бит

Реализуйте умножение 64-битных чисел с получением 128-битного результата, используя инструкции UMULL и UMLAL.

Вариант 7. Сравнение строк

Напишите ассемблерную вставку для побайтового сравнения двух строк до первого различия или конца. Сравните с библиотечной strcmp.

Вариант 8. Конвейеризация цикла

Преобразуйте цикл суммирования массива с использованием размотки (loop unrolling) на ассемблере. Измерьте ускорение.

Вариант 9. Деление через сдвиги

Реализуйте деление целого числа на 3 с использованием только сдвигов и сложений. Сравните код и скорость с обычным делением.

Вариант 10. Побайтовый обход

Напишите функцию, которая меняет порядок байт в 32-битном слове (endianness swap) на ассемблере, используя REV или сдвиги.

Вариант 11. Условная загрузка

Реализуйте if (x > y) z = a; else z = b; без условных переходов (используя CSEL или условное выполнение).

Вариант 12. Сдвиг с округлением

Напишите функцию сдвига вправо с округлением (как в DSP). Реализуйте на ассемблере, сравните с C-версией.

Вариант 13. Величина угла (arctan2)

Используя аппроксимацию, реализуйте быстрый арктангенс для фиксированной точки на ассемблере.

Вариант 14. Гамма-коррекция

Реализуйте табличную гамма-коррекцию для 8-битного изображения. Сравните C и ASM версии (с предвычислением таблицы).

Вариант 15. FIFO (кольцевой буфер)

Реализуйте операции чтения и записи кольцевого буфера на ассемблере (максимально эффективно). Сравните с C.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (основные понятия системы команд, синтаксис ассемблерных вставок, форматы инструкций ARM).
4. Ход выполнения работы с приведением:
   - Листингов всех разработанных функций (C и ASM)
   - Команд компиляции и дизассемблирования

- Фрагментов ассемблерного кода с пояснениями
- Таблиц сравнения производительности или размера кода

5. Результаты индивидуального задания (код, анализ, выводы).
6. Сравнительный анализ эффективности C-кода, ручных ассемблерных вставок и оптимизаций компилятора.
7. Выводы по работе (что освоено, какие особенности системы команд выявлены, когда целесообразно использовать ассемблер).
8. Ответы на контрольные вопросы.

Контрольные вопросы

1. Каков синтаксис ассемблерной вставки в GCC? Что означают разделы: ключевое слово vlatile?

2. Как компилятор связывает C-переменные с регистрами процессора в ассемблерной вставке?

3. Почему в ассемблерной вставке нужно указывать список разрушаемых регистров (clobber list)?

4. Какие ограничения накладывает архитектура ARM (Thumb-2) на использование регистров в ассемблерных вставках?

5. Чем отличается инструкция ADD от ADDS (с флагами)?

6. Как компилятор оптимизирует умножение на константу? Покажите пример замены x * 5.

7. Почему деление выполняется значительно дольше умножения на ARM Cortex-M?

8. Как влияет выравнивание данных на эффективность загрузки (LDR vs LDRH vs LDRB)?

9. Что такое условное выполнение в ARM? Приведите пример.

10. Как с помощью ассемблерной вставки реализовать атомарную операцию (например, compare-and-swap)?

11. В каких случаях ручная оптимизация на ассемблере даёт выигрыш по сравнению с C-компилятором с -O3?

12. Как проверить, что компилятор "выбросил" пустой цикл при оптимизации?

13. Что такое барьер компилятора (__asm__ volatile("" ::: "memory")) и для чего он нужен?

14. Как заставить компилятор использовать инструкции SIMD (например, SMLAD для DSP)?

15. Как интерпретировать вывод команды objdump -S с перемежением исходного кода? Почему строки C-кода могут не соответствовать инструкциям?

Лабораторная работа №3

Программная модель портов ввода-вывода

Цель работы: освоение принципов работы с периферийными устройствами микроконтроллера через memory-mapped I/O, получение навыков низкоуровневого программирования портов ввода-вывода (GPIO) на языке C, реализация библиотеки для управления светодиодами, кнопками и другими внешними устройствами.

Задачи:

- Изучить структуру регистров GPIO для микроконтроллеров семейства STM32 (ARM Cortex-M).
- Освоить прямой доступ к регистрам специальных функций (SFR) через указатели.
- Реализовать функции инициализации, установки, сброса и чтения состояния выводов.
- Научиться использовать побитовые операции для манипуляции отдельными битами в регистрах.
- Изучить различия между режимами работы GPIO: вход (pull-up/pull-down), выход (push-pull/open-drain), альтернативная функция, аналоговый режим.
- Освоить работу с подтягивающими резисторами и защитой от дребезга контактов.

Теоретическая часть

Memory-mapped I/O (MMIO)

В архитектурах ARM (и многих других) периферийные устройства отображаются в единое адресное пространство памяти. Это означает, что чтение или запись по определённому адресу физически взаимодействует с регистром периферийного устройства, а не с ОЗУ.

Для STM32 (Cortex-M) базовые адреса периферии:

- 0x40000000 – периферия APB1 (низкоскоростная)
- 0x40010000 – периферия APB2 (высокоскоростная, включая GPIO)
- 0x40020000 – периферия AHB1 (GPIO Port A–H для STM32F4)

Структура регистров GPIO (на примере STM32F4xx)

Каждый порт GPIO (GPIOA, GPIOB, и т.д.) имеет следующие 32-битные регистры:

Регистр	Смещение	Назначение
MODER	0x00	Режим работы (00: input, 01: output, 10: AF, 11: analog)
OTYPER	0x04	Тип выхода (0: push-pull, 1: open-drain)
OSPEEDR	0x08	Скорость переключения (00: low, 01: medium, 10: high, 11: very high)
PUPDR	0x0C	Подтяжка (00: no, 01: pull-up, 10: pull-down, 11: reserved)
IDR	0x10	Входные данные (только чтение)
ODR	0x14	Выходные данные
BSRR	0x18	Битовый набор/сброс (BS: биты 0–15, BR: биты 16–31)
LCKR	0x1C	Блокировка конфигурации
AFR[2]	0x20–0x24	Альтернативные функции (AFRL, AFRH)

Важные регистры для управления:

1. **MODER** – задаёт режим каждого из 16 выводов (2 бита на вывод):
   Бит 0-1: PIN0, бит 2-3: PIN1, ..., бит 30-31: PIN15
2. **ODR** – запись в этот регистр устанавливает или сбрасывает выход:
   GPIOA->ODR |= (1 << 5); *// установить PIN5 (не атомарно!)*

GPIOA->ODR &= ~(1 << 5); *// сбросить PIN5*

3. **BSRR** – атомарная установка/сброс (без чтения-модификации-записи):
   GPIOA->BSRR = (1 << 5); *// установить PIN5*
   GPIOA->BSRR = (1 << (5+16)); *// сбросить PIN5*
4. **IDR** – чтение состояния выводов (для входа):
   uint8_t state = (GPIOA->IDR >> 5) & 1; *// прочитать PIN5*

Режимы работы выводов:

- **Input (pull-up/pull-down)** – для чтения внешних сигналов (кнопки, датчики)
- **Output (push-pull)** – для управления светодиодами, реле (активно управляет уровнем)
- **Output (open-drain)** – для шин I2C, где нужна "линия с открытым стоком"
- **Alternate Function** – для подключения периферии (UART, SPI, I2C, таймеры)
- **Analog** – для АЦП/ЦАП (отключает цифровые буферы)

Побитовые операции в программировании GPIO:

Операция	Пример	Назначение
Установка бита	reg |= (1<<n)	Установить n-й бит в 1
Сброс бита	reg &= ~(1<<n)	Установить n-й бит в 0
Инвертирование	reg ^= (1<<n)	Переключить бит
Проверка бита	if (reg & (1<<n))	Проверить, установлен ли бит
Очистка поля	reg &= ~(3<<(2*n))	Обнулить 2 бита для MODER

Операция	Пример	Назначение
Установка поля	reg |= (mode << (2*n))	Записать значение в поле

Дребезг контактов (contact bounce):

Механические кнопки при нажатии создают серию ложных срабатываний. Методы подавления:

- Аппаратный: RC-фильтр, триггер Шмитта
- Программный: задержка 5–20 мс после первого срабатывания, затем повторное чтение

Пример структуры для доступа к регистрам:

```
// Определение структуры для GPIO
typedef struct {
  volatile uint32_t MODER;
  volatile uint32_t OTYPER;
  volatile uint32_t OSPEEDR;
  volatile uint32_t PUPDR;
  volatile uint32_t IDR;
  volatile uint32_t ODR;
  volatile uint32_t BSRR;
  volatile uint32_t LCKR;
  volatile uint32_t AFR[2];
} GPIO_TypeDef;

// Указатель на порт A (базовый адрес 0x40020000)
#define GPIOA ((GPIO_TypeDef *)0x40020000)
```

Методика выполнения работы

1. **Подготовка среды и создание проекта**

   1.1. Создайте каталог lab3 и файл gpio_driver.h:

```
#ifndef GPIO_DRIVER_H
#define GPIO_DRIVER_H

#include <stdint.h>

// Режимы работы вывода
typedef enum {
    GPIO_MODE_INPUT   = 0,
    GPIO_MODE_OUTPUT  = 1,
    GPIO_MODE_AF      = 2,
    GPIO_MODE_ANALOG  = 3
} GPIO_Mode_t;

// Тип выхода
typedef enum {
    GPIO_OTYPE_PUSH_PULL = 0,
    GPIO_OTYPE_OPEN_DRAIN = 1
} GPIO_OType_t;

// Скорость
typedef enum {
    GPIO_SPEED_LOW   = 0,
    GPIO_SPEED_MED   = 1,
    GPIO_SPEED_HIGH  = 2,
    GPIO_SPEED_VERY_HIGH = 3
} GPIO_Speed_t;
```

```
// Подтяжка
typedef enum {
    GPIO_PULL_NONE = 0,
    GPIO_PULL_UP   = 1,
    GPIO_PULL_DOWN = 2
} GPIO_Pull_t;

// Структура для инициализации пина
typedef struct {
    uint8_t pin;              // 0..15
    GPIO_Mode_t mode;
    GPIO_OType_t otype;
    GPIO_Speed_t speed;
    GPIO_Pull_t pull;
} GPIO_Init_t;

// Прототипы функций
void GPIO_Init(GPIO_TypeDef* port, GPIO_Init_t* config);
void GPIO_SetPin(GPIO_TypeDef* port, uint8_t pin);
void GPIO_ResetPin(GPIO_TypeDef* port, uint8_t pin);
void GPIO_TogglePin(GPIO_TypeDef* port, uint8_t pin);
uint8_t GPIO_ReadPin(GPIO_TypeDef* port, uint8_t pin);
void GPIO_WritePort(GPIO_TypeDef* port, uint16_t value);
uint16_t GPIO_ReadPort(GPIO_TypeDef* port);

#endif
```

1.2. Создайте файл gpio_driver.c с реализацией:

```
#include "gpio_driver.h"
```

```
void GPIO_Init(GPIO_TypeDef* port, GPIO_Init_t* config) {
    uint8_t pin = config->pin;
    uint32_t shift = pin * 2;

    // 1. Очищаем поле MODER для данного пина
    port->MODER &= ~(0x3 << shift);
    // 2. Устанавливаем режим
    port->MODER |= (config->mode << shift);

    // 3. Настройка типа выхода (только для OUTPUT или AF)
    if (config->mode == GPIO_MODE_OUTPUT || config->mode == GPIO_MODE_AF) {
        port->OTYPER &= ~(1 << pin);
        port->OTYPER |= (config->otype << pin);

        // 4. Настройка скорости
        port->OSPEEDR &= ~(0x3 << shift);
        port->OSPEEDR |= (config->speed << shift);
    }

    // 5. Настройка подтяжки
    port->PUPDR &= ~(0x3 << shift);
    port->PUPDR |= (config->pull << shift);
}

void GPIO_SetPin(GPIO_TypeDef* port, uint8_t pin) {
    // Атомарная установка через BSRR
    port->BSRR = (1 << pin);
}
```

```
void GPIO_ResetPin(GPIO_TypeDef* port, uint8_t pin) {
    port->BSRR = (1 << (pin + 16));
}

void GPIO_TogglePin(GPIO_TypeDef* port, uint8_t pin) {
    // Неатомарное переключение (чтение-модификация-запись)
    port->ODR ^= (1 << pin);
}

uint8_t GPIO_ReadPin(GPIO_TypeDef* port, uint8_t pin) {
    return (port->IDR >> pin) & 1;
}

void GPIO_WritePort(GPIO_TypeDef* port, uint16_t value) {
    port->ODR = value;
}

uint16_t GPIO_ReadPort(GPIO_TypeDef* port) {
    return (uint16_t)port->IDR;
}
```

1.3. Создайте файл main.c с демонстрационной программой:

```
#include "gpio_driver.h"

// Базовые адреса (для STM32F407VG)
#define GPIOA_BASE 0x40020000
#define GPIOB_BASE 0x40020400
#define GPIOC_BASE 0x40020800
#define RCC_BASE   0x40023800

#define GPIOA ((GPIO_TypeDef*)GPIOA_BASE)
```

```
#define GPIOB ((GPIO_TypeDef*)GPIOB_BASE)
#define GPIOC ((GPIO_TypeDef*)GPIOC_BASE)

// Регистр тактирования
typedef struct {
    volatile uint32_t CR;
    volatile uint32_t PLLCFGR;
    volatile uint32_t CFGR;
    volatile uint32_t CIR;
    volatile uint32_t AHB1RSTR;
    volatile uint32_t AHB2RSTR;
    volatile uint32_t AHB3RSTR;
    volatile uint32_t RESERVED0;
    volatile uint32_t APB1RSTR;
    volatile uint32_t APB2RSTR;
    volatile uint32_t RESERVED1[3];
    volatile uint32_t AHB1ENR;
    volatile uint32_t AHB2ENR;
    volatile uint32_t AHB3ENR;
    volatile uint32_t RESERVED2;
    volatile uint32_t APB1ENR;
    volatile uint32_t APB2ENR;
} RCC_TypeDef;

#define RCC ((RCC_TypeDef*)RCC_BASE)

// Функция задержки (простой цикл)
void delay(volatile uint32_t count) {
    while (count--) {
        __asm__ volatile ("nop");
```

```
    }
}

// Функция подавления дребезга
uint8_t read_button_debounced(GPIO_TypeDef* port, uint8_t pin, uint32_t debounce_ms) {
    if (GPIO_ReadPin(port, pin)) {
        delay(debounce_ms * 5000);  // грубая задержка (в реальном проекте использовать таймер)
        if (GPIO_ReadPin(port, pin)) {
            return 1;
        }
    }
    return 0;
}

int main(void) {
    // Включение тактирования GPIOA и GPIOC
    RCC->AHB1ENR |= (1 << 0);  // GPIOA
    RCC->AHB1ENR |= (1 << 2);  // GPIOC

    // Инициализация светодиода на PA5 (зеленый LED на STM32F4Discovery)
    GPIO_Init_t led_config = {
        .pin = 5,
        .mode = GPIO_MODE_OUTPUT,
        .otype = GPIO_OTYPE_PUSH_PULL,
        .speed = GPIO_SPEED_HIGH,
        .pull = GPIO_PULL_NONE
    };
```

```
    GPIO_Init(GPIOA, &led_config);

    // Инициализация кнопки на PC13 (пользовательская кнопка)
    GPIO_Init_t btn_config = {
        .pin = 13,
        .mode = GPIO_MODE_INPUT,
        .pull = GPIO_PULL_UP   // подтяжка к питанию (кнопка замыкает на землю)
    };
    GPIO_Init(GPIOC, &btn_config);

    while (1) {
        // Чтение кнопки с подавлением дребезга
        if (read_button_debounced(GPIOC, 13, 20)) {
            GPIO_TogglePin(GPIOA, 5);
            delay(500000);  // задержка для визуального подтверждения
        }
    }
}
```

2. **Изучение работы с регистрами через отладчик**

2.1. Скомпилируйте проект:

bash

```
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -O0 -g -c gpio_driver.c -o gpio_driver.o
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -O0 -g -c main.c -o main.o
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -T link.ld -nostdlib gpio_driver.o main.o -o firmware.elf
```

2.2. Запустите в отладчике (QEMU или реальная плата) и установите точки останова на функциях GPIO_Init, GPIO_SetPin.

2.3. Просмотрите значения регистров MODER, ODR, BSRR до и после выполнения.

3. **Создание тестовой программы "Бегущий огонь"**

3.1. Добавьте функцию для инициализации нескольких выводов:

```
void init_leds(void) {
    GPIO_Init_t led_cfg = {
        .mode = GPIO_MODE_OUTPUT,
        .otype = GPIO_OTYPE_PUSH_PULL,
        .speed = GPIO_SPEED_MED,
        .pull = GPIO_PULL_NONE
    };

    for (int pin = 0; pin < 8; pin++) {
        led_cfg.pin = pin;
        GPIO_Init(GPIOA, &led_cfg);
    }
}

void running_leds(void) {
    init_leds();

    while (1) {
        for (int i = 0; i < 8; i++) {
            GPIO_WritePort(GPIOA, 1 << i);
            delay(100000);
        }
    }
}
```

4. **Исследование атомарных операций**

4.1. Сравните код, сгенерированный для GPIO_SetPin (с BSRR) и для альтернативной реализации:

```
void GPIO_SetPin_nonatomic(GPIO_TypeDef* port, uint8_t pin) {
   port->ODR |= (1 << pin);
}
```

4.2. Проанализируйте дизассемблированный код. Объясните, почему BSRR безопаснее в многозадачной среде или прерываниях.

5. **Эксперимент с подтягивающими резисторами**

5.1. Измените конфигурацию кнопки на GPIO_PULL_NONE и прочитайте состояние ненажатой кнопки.

5.2. Объясните результат. Какое значение имеет "плавающий" вход?

6. **Реализация программного ШИМ (PWM) через таймер**

6.1. Добавьте простую программную ШИМ-функцию для управления яркостью светодиода:

```
void software_pwm(GPIO_TypeDef* port, uint8_t pin, uint8_t duty, uint32_t period) {
   uint32_t on_time = (period * duty) / 255;
   uint32_t off_time = period - on_time;

   GPIO_SetPin(port, pin);
   delay(on_time);
   GPIO_ResetPin(port, pin);
   delay(off_time);
}
```

Пример выполнения задания

Исходная задача: Управление RGB-светодиодом (общий катод) через три вывода GPIO.

Конфигурация:

- RED → PA0
- GREEN → PA1
- BLUE → PA2

Реализация:

```
#define RGB_RED_PIN   0
#define RGB_GREEN_PIN 1
#define RGB_BLUE_PIN  2

void rgb_init(void) {
    GPIO_Init_t rgb_cfg = {
        .mode = GPIO_MODE_OUTPUT,
        .otype = GPIO_OTYPE_PUSH_PULL,
        .speed = GPIO_SPEED_HIGH,
        .pull = GPIO_PULL_NONE
    };

    rgb_cfg.pin = RGB_RED_PIN;
    GPIO_Init(GPIOA, &rgb_cfg);

    rgb_cfg.pin = RGB_GREEN_PIN;
    GPIO_Init(GPIOA, &rgb_cfg);

    rgb_cfg.pin = RGB_BLUE_PIN;
    GPIO_Init(GPIOA, &rgb_cfg);
}
```

```
void rgb_set_color(uint8_t red, uint8_t green, uint8_t blue) {
    if (red) GPIO_SetPin(GPIOA, RGB_RED_PIN);
    else GPIO_ResetPin(GPIOA, RGB_RED_PIN);

    if (green) GPIO_SetPin(GPIOA, RGB_GREEN_PIN);
    else GPIO_ResetPin(GPIOA, RGB_GREEN_PIN);

    if (blue) GPIO_SetPin(GPIOA, RGB_BLUE_PIN);
    else GPIO_ResetPin(GPIOA, RGB_BLUE_PIN);
}

void rgb_demo(void) {
    rgb_init();

    while (1) {
        rgb_set_color(1, 0, 0);  // Красный
        delay(500000);
        rgb_set_color(0, 1, 0);  // Зелёный
        delay(500000);
        rgb_set_color(0, 0, 1);  // Синий
        delay(500000);
        rgb_set_color(1, 1, 0);  // Жёлтый
        delay(500000);
        rgb_set_color(0, 0, 0);  // Выкл
        delay(500000);
    }
}
```

Результаты анализа:

- При использовании BSRR атомарность гарантирует, что между чтением и записью ODR не произойдёт прерывания.
- Размер кода с BSRR: 12 байт, с ODR: 16 байт (команды LDR, ORR, STR).
- Подтяжка к питанию (pull-up) для кнопки экономит внешний резистор.

Индивидуальные задания

Вариант 1. Светофор

Реализуйте программную модель светофора с тремя светодиодами (красный, жёлтый, зелёный). Тайминги: красный – 5с, красный+жёлтый – 1с, зелёный – 4с, жёлтый – 1с. Используйте разные порты для каждого цвета.

Вариант 2. 4-битный счётчик

Подключите 4 светодиода к выводам PB0-PB3. Реализуйте двоичный счётчик от 0 до 15 с интервалом 0.5 секунды. Отображайте текущее значение в двоичном виде на светодиодах.

Вариант 3. Кнопочный переключатель

Одна кнопка переключает режимы работы светодиода: 1) выключен, 2) включен постоянно, 3) мигает с частотой 1 Гц, 4) мигает 2 Гц. Реализуйте смену режимов по нажатию с подавлением дребезга.

Вариант 4. Сдвиговый регистр (эмуляция)

Используя один вывод как данные (DATA) и один как тактовый (CLK), реализуйте программную эмуляцию сдвигового регистра 74HC595 для управления 8 светодиодами через 2 вывода GPIO.

Вариант 5. "Бегущий огонь" с изменяемой скоростью

Реализуйте бегущий огонь на 8 светодиодах. Скорость движения увеличивается при нажатии кнопки и уменьшается при нажатии другой кнопки. Скорость отображается на 3 дополнительных светодиодах (градации).

Вариант 6. Сканирование матричной клавиатуры 3×3

Реализуйте драйвер для матричной клавиатуры 3×3 (строки – выходы, столбцы – входы с подтяжкой). Выводите номер нажатой клавиши через UART или на светодиоды.

Вариант 7. Светодиодная шкала уровня

Подключите 10 светодиодов в виде линейной шкалы. Потенциометр, подключённый к АЦП (аналоговый вход), определяет уровень 0–10. Включите соответствующее количество светодиодов.

Вариант 8. "Дождь" (эффект случайных светодиодов)

8 светодиодов загораются в случайном порядке на короткое время (100 мс) и гаснут. Используйте генератор псевдослучайных чисел (LFSR) на сдвиговых регистрах.

Вариант 9. Пульт управления (IR-эмуляция)

На одну кнопку запишите код (например, 0x10FF), на другую – другой код. При нажатии кнопки выводите код на светодиодный индикатор с помощью ШИМ-модуляции 38 кГц (эмуляция ИК-передатчика NEC).

Вариант 10. Измерение ёмкости (RC-цепочка)

Используя вывод GPIO в режиме входа с триггером Шмитта, измерьте время заряда конденсатора через резистор (функция измерения ёмкости). Выведите результат в условных единицах на светодиодную шкалу.

Вариант 11. 7-сегментный индикатор

Подключите один 7-сегментный индикатор (общий катод) к 7 выводам GPIO. Реализуйте функции вывода цифр 0–9 и букв A–F. Добавьте динамическую индикацию для двух индикаторов.

Вариант 12. Кодовый замок

Используйте 4 кнопки (цифры 1,2,3,4) и один светодиод. Запрограммируйте код из 4 нажатий (например, 1,2,3,4). При правильном вводе светодиод загорается на 2 секунды, при неправильном – мигает 3 раза. Сброс после 3 секунд бездействия.

Вариант 13. Управление шаговым двигателем (эмуляция)

Реализуйте последовательность для униполярного шагового двигателя в режиме full-step и half-step на 4 выводах GPIO. Добавьте кнопку для смены направления вращения.

Вариант 14. Музыкальная шкатулка

Подключите пьезоизлучатель к выводу GPIO. Используя программные задержки, сгенерируйте ноты простой мелодии (например, "В лесу родилась ёлочка"). Для каждой ноты задаются частота и длительность.

Вариант 15. "Умная" мигалка (адаптивная частота)

Светодиод мигает с частотой, обратно пропорциональной освещённости: чем темнее – тем чаще. Освещённость моделируется потенциометром на аналоговом входе (или кнопками +/-).

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (принципы MMIO, структура регистров GPIO, побитовые операции).
4. Листинги разработанного драйвера (gpio_driver.h, gpio_driver.c) с комментариями.
5. Листинг основной программы с пояснением алгоритма работы.
6. Результаты выполнения индивидуального задания:
    - Схема подключения (таблица выводов)
    - Листинг кода
    - Описание работы программы
7. Скриншоты (или текстовые дампы) содержимого регистров MODER, ODR, BSRR, IDR из отладчика.
8. Анализ атомарности операций (сравнение кода с BSRR и с ODR).
9. Выводы по работе (что освоено, какие особенности работы с GPIO выявлены).
10. Ответы на контрольные вопросы.

Контрольные вопросы

1. Что такое memory-mapped I/O? Какие преимущества и недостатки у этого подхода?

2. Каким образом происходит включение тактирования периферийного блока GPIO в STM32?

3. Для чего нужен регистр BSRR и чем он лучше прямой записи в ODR?

4. Какой регистр отвечает за режим работы вывода (вход/выход/альтернативная функция/аналог)?

5. Чем отличается push-pull выход от open-drain? В каких сценариях используется каждый тип?

6. Как сконфигурировать вывод с подтяжкой к питанию (pull-up) и к земле (pull-down)?

7. Что такое "дребезг контактов" и какие методы его подавления существуют?

8. Почему для чтения состояния вывода используется регистр IDR, а не ODR?

9. Как с помощью одного регистра BSRR одновременно установить одни биты и сбросить другие?

10. Какова максимальная скорость переключения GPIO на STM32F4? От чего она зависит?

11. Почему в структуре GPIO регистры объявлены с модификатором volatile?

12. Как с помощью GPIO реализовать программную ШИМ? Каковы ограничения такого подхода?

13. Что произойдёт, если настроить вывод как выход, но не включить тактирование порта?

14. Как реализовать чтение с нескольких выводов одновременно (например, чтение 4-битного значения)?

15. Какие меры предосторожности нужно принимать при работе с портами в обработчиках прерываний?

Лабораторная работа №4

Таймеры в режиме реального времени

Цель работы: освоение программирования таймеров микроконтроллера для создания точных временных интервалов без использования пустых циклов, изучение механизмов прерываний от таймеров, реализация системного таймера (SysTick) для организации кооперативной многозадачности и планировщика реального времени.

Задачи:

- Изучить архитектуру и регистры таймеров общего назначения (TIM2–TIM5) и системного таймера SysTick.
- Освоить настройку таймеров в различных режимах: time-base, input capture, output compare, PWM.
- Реализовать точные неблокирующие задержки на основе аппаратных таймеров.
- Создать систему временных тиков (ticks) с обработчиком прерываний.
- Реализовать простейший кооперативный планировщик задач.
- Изучить режим capture для измерения длительности импульсов.
- Научиться настраивать аппаратный ШИМ (PWM) для управления яркостью светодиода или серводвигателем.

Теоретическая часть

Таймеры в микроконтроллерах STM32

В семействе STM32F4xx присутствует несколько типов таймеров:

Тип	Количество	Разрядность	Особенности
SysTick	1	24-битный	Системный таймер ядра Cortex-M
TIM2–TIM5	4	32-битных	Таймеры общего назначения

Тип	Количество	Разрядность	Особенности
TIM1, TIM8	2	16-битных	Продвинутые (для моторов)
TIM9–TIM14	6	16-битных	Таймеры общего назначения (basic)
TIM6, TIM7	2	16-битных	Базовые (только time-base)

Архитектура таймера общего назначения (TIM2)

Ключевые компоненты:

- **Счётчик (CNT)** – текущее значение счётчика.
- **Регистр автоперезагрузки (ARR)** – значение, при достижении которого счётчик сбрасывается.
- **Регистр предделителя (PSC)** – делитель тактовой частоты (регистр предделителя +1).
- **Регистры сравнения/захвата (CCR1–CCR4)** – используются для capture и compare.

Основные режимы работы:

1. **Time-base (базовый таймер)** – счётчик тактируется от APB, при достижении ARR генерируется событие Update (прерывание или DMA).
2. **Output Compare (сравнение выхода)** – когда CNT сравнивается с CCR, можно установить/сбросить/переключить вывод.
3. **Input Capture (захват входа)** – по фронту/спаду на входе сохраняется текущее значение CNT в CCR.
4. **PWM Mode** – автоматическая генерация ШИМ-сигнала с настраиваемым периодом (ARR) и скважностью (CCR).

Формулы для расчёта времени:

Частота счётчика = Тактовая_APB1 / (PSC + 1)

Частота Update = Частота_счётчика / (ARR + 1)

Время переполнения = 1 / Частота_Update = (PSC+1)*(ARR+1) / Тактовая_APB1

Пример: Тактовая APB1 = 84 МГц, PSC = 8399, ARR = 999 → время = 8400*1000/84e6 = 0.1 сек = 100 мс.

Системный таймер SysTick

SysTick – 24-битный таймер, встроенный в ядро Cortex-M. Регистры:

- STK_CTRL – управление (бит ENABLE, TICKINT, CLKSOURCE)
- STK_LOAD – значение для автоперезагрузки
- STK_VAL – текущее значение счётчика

Преимущества: не требует тактирования от APB, присутствует во всех Cortex-M, идеален для системных тиков RTOS.

Нон-блокирующие задержки (non-blocking delay)

Вместо пустого цикла delay() используется запись текущего времени и проверка разницы:

```
uint32_t get_tick(void);  // возвращает текущее системное время в мс

void delay_ms(uint32_t ms) {
   uint32_t start = get_tick();
   while ((get_tick() - start) < ms) {
      // выполнять другие задачи (или просто ждать в энергосберегающем режиме)
      __WFI();  // Wait For Interrupt
   }
}
```

Кооперативная многозадачность

Каждая задача добровольно передаёт управление через вызов планировщика:

```
void task1(void) {
   while (1) {
      // сделать что-то
      task_yield();  // передать управление следующей задаче
   }
}
```

Планировщик хранит указатели на функции задач и циклически их вызывает.

Режим Input Capture

Используется для измерения:

- длительности импульса
- периода сигнала
- частоты

Алгоритм измерения положительного импульса:

1. Настроить канал на захват по нарастающему фронту.
2. По прерыванию сохранить CCR1 как rise_time.
3. Переключить на захват по спадающему фронту.
4. По прерыванию сохранить CCR1 как fall_time.
5. Длительность = fall_time - rise_time (с учётом переполнений).

PWM (Pulse Width Modulation)

Параметры ШИМ:

- **Период (T)** – время одного полного цикла
- **Скважность (Duty Cycle)** – время высокого уровня / период × 100%
- **Частота (f = 1/T)**

Настройка:

- ARR определяет период (значение+1 делений таймера)

- CCRx определяет длительность высокого уровня

Пример для частоты 1 кГц и тактовой 84 МГц: PSC = 83 (84-1), ARR = 999 → частота 84e6/(84*1000)=1 кГц.

Методика выполнения работы

1. **Подготовка заголовочного файла для работы с таймерами**

   1.1. Создайте каталог lab4 и файл timers.h:

```
#ifndef TIMERS_H
#define TIMERS_H

#include <stdint.h>

// Структура для таймера общего назначения (TIM2)
typedef struct {
    volatile uint32_t CR1;      // Control register 1
    volatile uint32_t CR2;      // Control register 2
    volatile uint32_t SMCR;     // Slave mode control
    volatile uint32_t DIER;     // DMA/Interrupt enable
    volatile uint32_t SR;       // Status register
    volatile uint32_t EGR;      // Event generation
    volatile uint32_t CCMR1;    // Capture/compare mode 1
    volatile uint32_t CCMR2;    // Capture/compare mode 2
    volatile uint32_t CCER;     // Capture/compare enable
    volatile uint32_t CNT;      // Counter
    volatile uint32_t PSC;      // Prescaler
    volatile uint32_t ARR;      // Auto-reload register
    volatile uint32_t RESERVED1;
    volatile uint32_t CCR1;     // Capture/compare 1
    volatile uint32_t CCR2;     // Capture/compare 2
    volatile uint32_t CCR3;     // Capture/compare 3
```

```
    volatile uint32_t CCR4;     // Capture/compare 4
    volatile uint32_t RESERVED2[4];
    volatile uint32_t DCR;      // DMA control
    volatile uint32_t DMAR;     // DMA address
} TIM_TypeDef;

// Структура для SysTick
typedef struct {
    volatile uint32_t CTRL;     // Control and status
    volatile uint32_t LOAD;     // Reload value
    volatile uint32_t VAL;      // Current value
    volatile uint32_t CALIB;    // Calibration
} SysTick_TypeDef;

#define TIM2_BASE  0x40000000
#define TIM3_BASE  0x40000400
#define TIM4_BASE  0x40000800
#define TIM5_BASE  0x40000C00
#define SysTick_BASE 0xE000E010

#define TIM2  ((TIM_TypeDef*)TIM2_BASE)
#define TIM3  ((TIM_TypeDef*)TIM3_BASE)
#define TIM4  ((TIM_TypeDef*)TIM4_BASE)
#define TIM5  ((TIM_TypeDef*)TIM5_BASE)
#define SysTick ((SysTick_TypeDef*)SysTick_BASE)

// Функции
void TIM2_Init_TimeBase(uint32_t frequency_hz);
void TIM2_Start(void);
void TIM2_Stop(void);
```

```
uint32_t TIM2_GetCounter(void);
void TIM2_DelayUs(uint32_t us);
void TIM2_DelayMs(uint32_t ms);

void SysTick_Init(uint32_t frequency_hz);
uint32_t SysTick_GetTick(void);
void SysTick_DelayMs(uint32_t ms);

#endif
```

1.2. Создайте файл timers.c с реализацией:

```
#include "timers.h"
#include "stm32f4xx.h"  // Базовые адреса и RCC

volatile uint32_t system_tick = 0;

// Инициализация TIM2 в режиме time-base
void TIM2_Init_TimeBase(uint32_t frequency_hz) {
    // Включение тактирования TIM2
    RCC->APB1ENR |= (1 << 0);  // TIM2EN

    // Сброс настроек TIM2
    TIM2->CR1 = 0;
    TIM2->CR2 = 0;
    TIM2->SMCR = 0;
    TIM2->DIER = 0;

    // Расчёт PSC и ARR для требуемой частоты (APB1 = 84 MHz)
    // Частота обновления = (84e6 / (PSC+1)) / (ARR+1) = frequency_hz
    // Выбираем коэффициент деления 1000 для удобства
```

```
    uint32_t prescaler = 8399;  // 84e6 / 8400 = 10000 Hz
    uint32_t period = (84000000 / (prescaler + 1)) / frequency_hz - 1;

    TIM2->PSC = prescaler;
    TIM2->ARR = period;
    TIM2->CNT = 0;

    // Разрешить прерывание по обновлению
    TIM2->DIER |= (1 << 0);  // UIE

    // Настройка NVIC
    NVIC_EnableIRQ(TIM2_IRQn);
    NVIC_SetPriority(TIM2_IRQn, 1);

    TIM2->CR1 |= (1 << 0);  // CEN - включить счётчик
}

void TIM2_Start(void) {
    TIM2->CR1 |= (1 << 0);
}

void TIM2_Stop(void) {
    TIM2->CR1 &= ~(1 << 0);
}

uint32_t TIM2_GetCounter(void) {
    return TIM2->CNT;
}

// Блокирующая задержка (только для обучения!)
```

```
void TIM2_DelayUs(uint32_t us) {
    TIM2->CNT = 0;
    while (TIM2->CNT < us) {
        // wait
    }
}

void TIM2_DelayMs(uint32_t ms) {
    for (uint32_t i = 0; i < ms; i++) {
        TIM2_DelayUs(1000);
    }
}

// Обработчик прерывания TIM2
void TIM2_IRQHandler(void) {
    if (TIM2->SR & (1 << 0)) {  // Update interrupt flag
        TIM2->SR &= ~(1 << 0);  // Clear flag
        // Периодическая задача (например, переключение светодиода)
    }
}

// Инициализация SysTick (тики 1 мс)
void SysTick_Init(uint32_t frequency_hz) {
    uint32_t load = (SystemCoreClock / frequency_hz) - 1;
    SysTick->LOAD = load;
    SysTick->VAL = 0;
    SysTick->CTRL = (1 << 2) | (1 << 1) | (1 << 0);   // CLKSOURCE=1,
TICKINT=1, ENABLE=1
}
```

```
uint32_t SysTick_GetTick(void) {
    return system_tick;
}

void SysTick_DelayMs(uint32_t ms) {
    uint32_t start = system_tick;
    while ((system_tick - start) < ms) {
        // idle
    }
}

// Обработчик SysTick (определение слабого символа)
void SysTick_Handler(void) {
    system_tick++;
}
```

2. **Создание неблокирующей задержки и простого планировщика**

2.1. Создайте файл scheduler.h:

```
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include <stdint.h>

#define MAX_TASKS 8

typedef struct {
    void (*task_func)(void);
    uint32_t period_ms;
    uint32_t last_run;
    uint8_t enabled;
```

```
} Task_t;

void Scheduler_Init(void);
uint8_t Scheduler_AddTask(void (*func)(void), uint32_t period_ms);
void Scheduler_Run(void);
void Scheduler_Dispatch(void);  // Вызывать из main loop

#endif
```

2.2. Создайте файл scheduler.c:

```
#include "scheduler.h"
#include "timers.h"

static Task_t tasks[MAX_TASKS];
static uint8_t task_count = 0;

void Scheduler_Init(void) {
    for (int i = 0; i < MAX_TASKS; i++) {
        tasks[i].enabled = 0;
    }
}

uint8_t Scheduler_AddTask(void (*func)(void), uint32_t period_ms) {
    if (task_count >= MAX_TASKS) return 0;

    tasks[task_count].task_func = func;
    tasks[task_count].period_ms = period_ms;
    tasks[task_count].last_run = SysTick_GetTick();
    tasks[task_count].enabled = 1;
    task_count++;
```

```
    return 1;
}

void Scheduler_Dispatch(void) {
    uint32_t current_tick = SysTick_GetTick();

    for (int i = 0; i < task_count; i++) {
        if (tasks[i].enabled) {
            if ((current_tick - tasks[i].last_run) >= tasks[i].period_ms) {
                tasks[i].last_run = current_tick;
                tasks[i].task_func();
            }
        }
    }
}

void Scheduler_Run(void) {
    while (1) {
        Scheduler_Dispatch();
        __WFI();  // Wait for interrupt (снижение энергопотребления)
    }
}
```

2.3. Создайте файл main.c с демонстрацией:

```
#include "timers.h"
#include "scheduler.h"
#include "gpio_driver.h"  // Из ЛР3

// Светодиодные задачи
```

```
void led1_task(void) {
    static uint8_t state = 0;
    state = !state;
    if (state)
        GPIO_SetPin(GPIOA, 5);
    else
        GPIO_ResetPin(GPIOA, 5);
}

void led2_task(void) {
    static uint8_t state = 0;
    state = !state;
    if (state)
        GPIO_SetPin(GPIOA, 6);
    else
        GPIO_ResetPin(GPIOA, 6);
}

void status_task(void) {
    // Вывод информации через UART (опционально)
    static uint32_t counter = 0;
    counter++;
    // Здесь можно мигать "здоровьем" системы
}

int main(void) {
    // Включение тактирования GPIOA
    RCC->AHB1ENR |= (1 << 0);

    // Инициализация светодиодов PA5 и PA6
```

```
    GPIO_Init_t led_cfg = {
        .pin = 5,
        .mode = GPIO_MODE_OUTPUT,
        .otype = GPIO_OTYPE_PUSH_PULL,
        .speed = GPIO_SPEED_HIGH
    };
    GPIO_Init(GPIOA, &led_cfg);

    led_cfg.pin = 6;
    GPIO_Init(GPIOA, &led_cfg);

    // Инициализация SysTick (тик каждую 1 мс)
    SysTick_Init(1000);

    // Инициализация планировщика
    Scheduler_Init();
    Scheduler_AddTask(led1_task, 500);      // мигает каждые 500 мс
    Scheduler_AddTask(led2_task, 1000);     // мигает каждую секунду
    Scheduler_AddTask(status_task, 5000);   // каждые 5 секунд

    // Запуск планировщика (бесконечный цикл)
    Scheduler_Run();

    return 0;
}
```

3. **Режим Input Capture для измерения длительности импульса**

   3.1. Добавьте в timers.c функции для capture:

```
// Настройка TIM2, канал 1 для input capture
void TIM2_IC_Init(void) {
```

```
    RCC->APB1ENR |= (1 << 0);  // TIM2EN

    // Настройка GPIO для TIM2_CH1 (PA0)
    GPIO_Init_t capture_cfg = {
        .pin = 0,
        .mode = GPIO_MODE_AF,
        .otype = GPIO_OTYPE_PUSH_PULL,
        .speed = GPIO_SPEED_HIGH,
        .pull = GPIO_PULL_NONE
    };
    GPIO_Init(GPIOA, &capture_cfg);

    // Альтернативная функция для PA0: AF1 (TIM2_CH1)
    GPIOA->AFR[0] |= (1 << 0);  // AFRL для PIN0

    // Настройка таймера
    TIM2->PSC = 83;  // 84 MHz / 84 = 1 MHz (1 us на тик)
    TIM2->ARR = 0xFFFFFFFF;  // 32-битный максимум

    // Настройка канала 1 в режиме input capture
    TIM2->CCMR1 &= ~(0x3 << 0);   // CC1S = 00 (выход)
    TIM2->CCMR1 |= (1 << 0);      // CC1S = 01 (вход TI1)
    TIM2->CCER |= (1 << 0);       // CC1E = 1 (включить захват)
    TIM2->CCER &= ~(1 << 1);      // CC1P = 0 (захват по нарастанию)

    TIM2->DIER |= (1 << 1);       // CC1IE = 1 (прерывание по захвату)

    TIM2->CR1 |= (1 << 0);        // CEN
}
```

```
volatile uint32_t last_capture = 0;
volatile uint32_t pulse_width = 0;
volatile uint8_t capture_ready = 0;

void TIM2_IRQHandler(void) {
   if (TIM2->SR & (1 << 1)) {  // CC1IF
      TIM2->SR &= ~(1 << 1);

      static uint32_t rise_time = 0;
      static uint8_t edge = 0;

      if (edge == 0) {
         // Нарастающий фронт
         rise_time = TIM2->CCR1;
         TIM2->CCER |= (1 << 1);   // CC1P = 1 (смена на спад)
         edge = 1;
      } else {
         // Спадающий фронт
         uint32_t fall_time = TIM2->CCR1;
         if (fall_time >= rise_time) {
            pulse_width = fall_time - rise_time;
         } else {
            // Переполнение счётчика
            pulse_width = (0xFFFFFFFF - rise_time) + fall_time;
         }
         capture_ready = 1;
         TIM2->CCER &= ~(1 << 1);  // CC1P = 0 (обратно на нарастание)
         edge = 0;
      }
   }
```

```
    if (TIM2->SR & (1 << 0)) {  // Update interrupt (переполнение)
        TIM2->SR &= ~(1 << 0);
    }
}
```

4. **Режим PWM для управления яркостью светодиода**

4.1. Добавьте функцию настройки PWM:

```
// Настройка TIM3, канал 2 (PB5) для PWM
void TIM3_PWM_Init(uint32_t frequency_hz, uint8_t duty_percent) {
    // Включение тактирования TIM3 и GPIOB
    RCC->APB1ENR |= (1 << 1);   // TIM3EN
    RCC->AHB1ENR |= (1 << 1);   // GPIOBEN

    // Настройка PB5 как AF2 (TIM3_CH2)
    GPIO_Init_t pwm_cfg = {
        .pin = 5,
        .mode = GPIO_MODE_AF,
        .otype = GPIO_OTYPE_PUSH_PULL,
        .speed = GPIO_SPEED_HIGH,
        .pull = GPIO_PULL_NONE
    };
    GPIO_Init(GPIOB, &pwm_cfg);
    GPIOB->AFR[0] |= (2 << (5 * 4));  // AFR0 для PB5: AF2

    // Расчёт PSC и ARR для требуемой частоты
    uint32_t prescaler = 83;  // 84 MHz / 84 = 1 MHz
    uint32_t period = (84000000 / (prescaler + 1)) / frequency_hz - 1;

    TIM3->PSC = prescaler;
```

```
    TIM3->ARR = period;
    TIM3->CCR2 = (period * duty_percent) / 100;

    // Настройка канала 2 в режим PWM1
    TIM3->CCMR1 &= ~(0x7 << 12);   // OC2M = 000
    TIM3->CCMR1 |= (0x6 << 12);    // OC2M = 110 (PWM mode 1)
    TIM3->CCMR1 |= (1 << 11);      // OC2PE = 1 (preload enable)
    TIM3->CCER |= (1 << 4);        // CC2E = 1 (output enable)

    TIM3->CR1 |= (1 << 7);         // ARPE = 1
    TIM3->CR1 |= (1 << 0);         // CEN
}

void TIM3_PWM_SetDuty(uint8_t duty_percent) {
    uint32_t period = TIM3->ARR;
    TIM3->CCR2 = (period * duty_percent) / 100;
}
```

Пример выполнения задания

Задача: Реализовать "дыхание" светодиода (плавное изменение яркости) с использованием аппаратного PWM.

Решение:

```
#include "timers.h"

void breathing_led_demo(void) {
    TIM3_PWM_Init(1000, 0);  // 1 кГц, начальная яркость 0%

    uint8_t duty = 0;
    uint8_t direction = 1;  // 1 = увеличение, 0 = уменьшение
```

```
    while (1) {
        TIM3_PWM_SetDuty(duty);

        if (direction) {
            duty += 5;
            if (duty >= 100) {
                duty = 100;
                direction = 0;
            }
        } else {
            duty -= 5;
            if (duty <= 0) {
                duty = 0;
                direction = 1;
            }
        }

        SysTick_DelayMs(50);  // шаг каждые 50 мс
    }
}
```

Результаты осциллографирования (теоретически):

- При duty = 0%: постоянный 0 В
- При duty = 20%: высокий уровень 3.3 В в течение 0.2 мс, 0 В в течение 0.8 мс
- При duty = 50%: импульсы длительностью 0.5 мс
- При duty = 100%: постоянный 3.3 В

Измерение длительности импульса с помощью Input Capture:

Подключите к PA0 (TIM2_CH1) внешний генератор импульсов. Программа выведет длительность в мкс:

```
int main(void) {
    TIM2_IC_Init();

    while (1) {
        if (capture_ready) {
            // pulse_width содержит длительность в микросекундах
            capture_ready = 0;
        }
    }
}
```

Индивидуальные задания

Вариант 1. Электронный секундомер

Реализуйте секундомер с точностью до 0.01 секунды. Используйте таймер в режиме input capture. Кнопка "Старт/Стоп" запускает/останавливает отсчёт, вторая кнопка "Сброс". Результат выводится на 7-сегментные индикаторы или через UART.

Вариант 2. Часы реального времени (RTC) на базе SysTick

Используя SysTick с периодом 1 мс, реализуйте программные часы с отображением часов, минут, секунд и 1/100 секунды. Добавьте функцию корректировки времени с помощью кнопок.

Вариант 3. Генератор звуков с переменной частотой

Используя PWM (или output compare) на таймере, реализуйте генератор звуковых частот (50 Гц – 10 кГц). Потенциометр через АЦП управляет частотой. Сигнал выводится на пьезоизлучатель.

Вариант 4. Измеритель частоты сигнала

Реализуйте измеритель частоты внешнего TTL-сигнала. Используйте один таймер в режиме input capture (измерение периода) или используйте второй таймер в качестве счётчика импульсов за фиксированный интервал (1 секунда). Частота выводится через UART.

Вариант 5. Планировщик с динамическими приоритетами

Модифицируйте планировщик из методики, добавив поддержку приоритетов (чем меньше приоритет – тем выше важность). Задача с более высоким приоритетом выполняется раньше при одновременной готовности.

Вариант 6. Управление серводвигателем

Реализуйте драйвер для SG90 (или аналогичного сервопривода). Период ШИМ = 20 мс (50 Гц). Длительность импульса 0.5–2.5 мс (0°–180°). Потенциометр задаёт угол поворота.

Вариант 7. Мигалка с "адаптивной частотой" по кнопке

При удержании кнопки частота мигания светодиода линейно увеличивается от 0.5 Гц до 10 Гц (за 5 секунд). После отпускания кнопки частота возвращается к исходной.

Вариант 8. Временная задержка с произвольным числом (soft timer)

Реализуйте массив программных таймеров (software timers). Каждый таймер имеет callback-функцию, которая вызывается по истечении периода (однократно или периодически). Используйте один аппаратный таймер (SysTick).

Вариант 9. Реализация протокола "OneWire" с точными задержками

Реализуйте тайминги для протокола Dallas OneWire (DS18B20). Используйте таймер для генерации точных задержек 60 мкс, 240 мкс, 480 мкс. Считайте температуру с датчика.

Вариант 10. Переливы RGB-светодиода с ШИМ

Используйте три канала ШИМ на одном таймере (например, TIM3_CH1, CH2, CH3) для управления RGB-светодиодом. Реализуйте плавное изменение цвета по радуге или случайные плавные переходы.

Вариант 11. Измерение ёмкости (как в ЛР3) с таймером

Используйте input capture для точного измерения времени заряда RC-цепи через резистор 10 кОм. Вычислите ёмкость по формуле $\tau = RC$. Результат в нанофарадах выведите через UART.

Вариант 12. Метеостанция (периодический опрос датчиков)

Используя планировщик, опрашивайте каждые 2 секунды датчик температуры (DS18B20/US18B20), каждые 5 секунд – датчик влажности (DHT11). Выводите данные через UART. Используйте неблокирующие задержки в драйверах.

Вариант 13. Эффект "бегущего огня" с ШИМ-затуханием

8 светодиодов последовательно загораются и плавно гаснут (эффект "бегущей волны"). Используйте ШИМ на одном таймере и переключайте выводы программно (или используйте 8 каналов ШИМ на двух таймерах).

Вариант 14. Таймер "Обратного отсчёта" с прерыванием

Реализуйте таймер обратного отсчёта от 99 секунд до 0. Используйте 7-сегментные индикаторы. При достижении 0 включается светодиод или зуммер. Управление: кнопка "Установить значение", "+1", "-1", "Старт".

Вариант 15. Измерение скорости вращения вентилятора

Используйте оптопару (или датчик Холла) для измерения оборотов вентилятора. Сигнал поступает на вход capture таймера. Измеряйте период и вычисляйте RPM (оборотов в минуту). Отображайте на светодиодной шкале.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (устройство таймера, формулы расчёта времени, принципы работы системного таймера и планировщика).
4. Листинги разработанных модулей (timers.h, timers.c, scheduler.h, scheduler.c) с комментариями.
5. Листинг основной программы с пояснением алгоритма.
6. Результаты выполнения индивидуального задания:

- Принципиальную схему подключения (или таблицу выводов)
- Листинг кода с подробными комментариями
- Описание работы программы

7. Временные диаграммы для генерации PWM (рассчитанные) и полученные с осциллографа (или скриншоты логического анализатора).
8. Оценку точности задержек (сравнение расчётных и реальных значений).
9. Выводы по работе (что освоено, какие преимущества дают аппаратные таймеры по сравнению с цикл-задержками).
10. Ответы на контрольные вопросы.

Контрольные вопросы

1. Чем отличаются таймеры общего назначения от системного таймера SysTick?

2. Как рассчитать PSC и ARR для получения требуемой частоты прерываний?

3. В чём преимущество неблокирующих задержек перед delay() на пустых циклах?

4. Как работает режим Input Capture? Приведите алгоритм измерения длительности положительного импульса.

5. Что такое "захват по обоим фронтам" и как его реализовать программно?

6. Объясните формулу для расчёта периода в режиме PWM.

7. Как синхронизировать несколько таймеров в STM32 (master-slave)?

8. Какие прерывания может генерировать таймер общего назначения?

9. Как реализовать задержку в микросекундах с помощью SysTick? Какую максимальную задержку он может обеспечить?

10. Что произойдёт, если в 32-битном таймере ARR установить в 0xFFFFFFFF? Как часто будет возникать прерывание?

11. Как реализовать кооперативный планировщик задач на основе системного тика?

12. В чём разница между режимами PWM1 и PWM2 на таймере?

13. Как измерить период сигнала с помощью Input Capture, если частота сигнала ниже частоты таймера?

14. Как организовать долгую задержку (например, 1 час) с помощью 16-битного таймера?

15. Почему при работе с таймером в прерывании необходимо сбрасывать флаг прерывания вручную?

16. Увеличивается ли потребление энергии при использовании таймера с высокой частотой прерываний?

17. Как можно реализовать "программный сторожевой таймер" (software watchdog) на основе SysTick?

18. Как использовать таймер для точной генерации сигнала с частотой 1 Гц (секундная метка)?

Лабораторная работа №5

Обработка прерываний. Векторная таблица

Цель работы: освоение механизма обработки прерываний в микроконтроллерах ARM Cortex-M, изучение структуры векторной таблицы, приобретение навыков написания обработчиков прерываний (ISR) на языке C, настройка приоритетов с использованием контроллера вложенных прерываний (NVIC) и создание программных прерываний.

Задачи:

- Изучить структуру и расположение векторной таблицы в памяти микроконтроллера.
- Освоить настройку NVIC (Nested Vectored Interrupt Controller) для разрешения/запрета прерываний и установки приоритетов.
- Написать собственные обработчики прерываний для периферийных устройств (таймер, кнопка, UART).
- Изучить механизм вложенных прерываний и влияние приоритетов на их поведение.
- Реализовать программное прерывание с помощью регистра NVIC->STIR.
- Исследовать критически важные участки кода и способы их защиты (запрет прерываний).

Теоретическая часть

Понятие прерывания (Interrupt)

Прерывание – это аппаратно или программно инициируемое событие, которое приостанавливает выполнение текущей программы и передаёт управление специальному обработчику (Interrupt Service Routine, ISR). После выполнения ISR возобновляется прерванная задача.

Классификация прерываний:

- **Аппаратные внешние** – от периферийных устройств (GPIO, таймеры, UART)
- **Аппаратные внутренние** – исключения (division by zero, hard fault)
- **Программные (Software Triggered Interrupt, STI)** – вызываются записью в специальный регистр

Векторная таблица (Vector Table)

Векторная таблица – это массив 32-битных указателей (адресов), расположенный в начале адресного пространства (обычно по адресу 0x00000000 во Flash или 0x20000000 в RAM). Первое слово содержит начальное значение указателя стека (MSP), последующие – адреса обработчиков исключений и прерываний.

Структура векторной таблицы для Cortex-M (первые несколько записей):

Смещение	Исключение	Назначение
0x00	Initial SP	Начальное значение Main Stack Pointer
0x04	Reset	Точка входа после сброса
0x08	NMI	Non-Maskable Interrupt
0x0C	HardFault	Аппаратная ошибка
0x10	MemManage	Ошибка управления памятью (MPU)
0x14	BusFault	Ошибка шины
0x18	UsageFault	Ошибка использования инструкций
...	...	...
0x40	TIM2_IRQ	Прерывание от TIM2 (номер 28)

NVIC (Nested Vectored Interrupt Controller)

NVIC – встроенный в ядро контроллер прерываний, обеспечивающий:

- Вложенность прерываний (прерывание с более высоким приоритетом может прервать обработку более низкого)
- Динамическое изменение приоритетов
- Маскирование прерываний (глобальное и per-interrupt)
- Генерацию программных прерываний

Регистры NVIC (для Cortex-M4):

- NVIC_ISER[0..7] – Set-Enable регистры (разрешение прерываний)
- NVIC_ICER[0..7] – Clear-Enable регистры (запрет прерываний)
- NVIC_ISPR[0..7] – Set-Pending (установка флага ожидания)
- NVIC_ICPR[0..7] – Clear-Pending (сброс флага ожидания)
- NVIC_IPR[0..59] – Priority регистры (8 бит на прерывание, но обычно используются старшие 4 бита)
- NVIC_STIR – Software Trigger Interrupt Register

Приоритеты прерываний

В Cortex-M используется 4–8 бит для задания приоритета (зависит от реализации). Меньшее числовое значение = более высокий приоритет.

Пример группировки (AIRCR регистр):

- Бит группирования PRIGROUP определяет разделение поля приоритета на групповой приоритет (прерывание с более высоким групповым может прервать) и подприоритет (используется при равенстве групповых).

Атомарность и критические секции

Для защиты общих данных между основным кодом и ISR необходимо запрещать прерывания:

__disable_irq(); *// глобальное запрещение прерываний*

// критическая секция

__enable_irq(); *// глобальное разрешение*

Или для отдельных прерываний через NVIC_ICER.

Соглашения об именах обработчиков

Для стандартной периферии STM32 в startup-файлах уже определены слабые (weak) символы обработчиков. Разработчик должен переопределить их, например:

```
void TIM2_IRQHandler(void) {
   // обработчик
}
```

Программное прерывание

Запись номера прерывания в регистр NVIC_STIR вызывает его:

```
NVIC->STIR = 28;  // программное прерывание TIM2
```

Требуется установить бит USERSETMPEND в CCR регистре ядра.

Методика выполнения работы

1. **Подготовка структуры векторной таблицы**

1.1. Создайте каталог lab5 и файл startup.c (имитация startup-файла):

```
#include <stdint.h>

// Внешние объявления обработчиков (определены в main.c)
extern void Reset_Handler(void);
extern void NMI_Handler(void);
extern void HardFault_Handler(void);
extern void TIM2_IRQHandler(void);
extern void EXTI0_IRQHandler(void);
extern void Default_Handler(void);

// Слабые обработчики по умолчанию
```

```
__attribute__((weak)) void NMI_Handler(void) { while(1); }
__attribute__((weak)) void HardFault_Handler(void) { while(1); }
__attribute__((weak)) void Default_Handler(void) { while(1); }

// Векторная таблица (размещается по адресу 0x08000000)
__attribute__((section(".isr_vector")))
const uint32_t vector_table[] = {
    (uint32_t)0x20020000,      // Initial SP
    (uint32_t)Reset_Handler,    // Reset
    (uint32_t)NMI_Handler,
    (uint32_t)HardFault_Handler,
    (uint32_t)Default_Handler,  // MemManage
    (uint32_t)Default_Handler,  // BusFault
    (uint32_t)Default_Handler,  // UsageFault
    0, 0, 0, 0,                 // Reserved
    (uint32_t)Default_Handler,  // SVCall
    0,                          // Reserved
    0,                          // Reserved
    (uint32_t)Default_Handler,  // PendSV
    (uint32_t)Default_Handler,  // SysTick
    // Периферийные прерывания
    (uint32_t)Default_Handler,  // WWDG
    (uint32_t)Default_Handler,  // PVD
    0,                          // ... многие пропущены для краткости
    (uint32_t)TIM2_IRQHandler,  // TIM2 (IRQ 28)
    (uint32_t)EXTI0_IRQHandler, // EXTI0 (IRQ 6)
    // ... остальные до 91
};
```

1.2. Создайте скрипт линковщика link.ld с размещением секции .isr_vector:

```
MEMORY {
   FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 1M
   RAM (rwx)  : ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS {
   .isr_vector : {
      KEEP(*(.isr_vector))
   } > FLASH

   . : {
      *(.*)
   } > FLASH

   .data : {
      *(.data*)
   } > RAM AT> FLASH

   .bss : {
      *(.bss*)
   } > RAM
}
```

2. **Создание обработчиков прерываний для таймера и кнопки**

2.1. Создайте файл main.c с базовой настройкой прерываний:

```
#include <stdint.h>

// Базовые адреса и структуры (упрощённо)
```

```
#define RCC_BASE      0x40023800
#define GPIOA_BASE    0x40020000
#define TIM2_BASE     0x40000000
#define EXTI_BASE     0x40013C00
#define NVIC_BASE     0xE000E100

typedef struct {
   volatile uint32_t CR;
   volatile uint32_t PLLCFGR;
   volatile uint32_t CFGR;
   volatile uint32_t CIR;
   volatile uint32_t AHB1RSTR;
   volatile uint32_t AHB2RSTR;
   volatile uint32_t AHB3RSTR;
   uint32_t Reserved0;
   volatile uint32_t APB1RSTR;
   volatile uint32_t APB2RSTR;
   uint32_t Reserved1[3];
   volatile uint32_t AHB1ENR;
   volatile uint32_t AHB2ENR;
   volatile uint32_t AHB3ENR;
   uint32_t Reserved2;
   volatile uint32_t APB1ENR;
   volatile uint32_t APB2ENR;
} RCC_TypeDef;

#define RCC ((RCC_TypeDef*)RCC_BASE)

typedef struct {
   volatile uint32_t MODER;
```

```
    volatile uint32_t OTYPER;
    volatile uint32_t OSPEEDR;
    volatile uint32_t PUPDR;
    volatile uint32_t IDR;
    volatile uint32_t ODR;
    volatile uint32_t BSRR;
    volatile uint32_t LCKR;
    volatile uint32_t AFR[2];
} GPIO_TypeDef;

#define GPIOA ((GPIO_TypeDef*)GPIOA_BASE)

typedef struct {
    volatile uint32_t CR1;
    volatile uint32_t CR2;
    volatile uint32_t SMCR;
    volatile uint32_t DIER;
    volatile uint32_t SR;
    volatile uint32_t EGR;
    volatile uint32_t CCMR1;
    volatile uint32_t CCMR2;
    volatile uint32_t CCER;
    volatile uint32_t CNT;
    volatile uint32_t PSC;
    volatile uint32_t ARR;
    volatile uint32_t RESERVED1;
    volatile uint32_t CCR1;
    volatile uint32_t CCR2;
    volatile uint32_t CCR3;
    volatile uint32_t CCR4;
```

```
} TIM_TypeDef;

#define TIM2 ((TIM_TypeDef*)TIM2_BASE)

typedef struct {
    volatile uint32_t IMR;
    volatile uint32_t EMR;
    volatile uint32_t RTSR;
    volatile uint32_t FTSR;
    volatile uint32_t SWIER;
    volatile uint32_t PR;
} EXTI_TypeDef;

#define EXTI ((EXTI_TypeDef*)EXTI_BASE)

typedef struct {
    volatile uint32_t ISER[8];
    uint32_t RESERVED0[24];
    volatile uint32_t ICER[8];
    uint32_t RESERVED1[24];
    volatile uint32_t ISPR[8];
    uint32_t RESERVED2[24];
    volatile uint32_t ICPR[8];
    uint32_t RESERVED3[24];
    volatile uint32_t IPR[60];
    uint32_t RESERVED4[644];
    volatile uint32_t STIR;
} NVIC_TypeDef;

#define NVIC ((NVIC_TypeDef*)NVIC_BASE)
```

```
// Глобальные переменные для подсчёта прерываний
volatile uint32_t tim2_interrupt_count = 0;
volatile uint32_t exti0_interrupt_count = 0;
volatile uint32_t shared_variable = 0;

// Функция задержки (цикл)
void delay(volatile uint32_t count) {
    while (count--) { __asm volatile("nop"); }
}

// Инициализация светодиода PA5
void led_init(void) {
    RCC->AHB1ENR |= (1 << 0);  // GPIOA clock
    GPIOA->MODER &= ~(3 << (5*2));
    GPIOA->MODER |= (1 << (5*2));
    GPIOA->OTYPER &= ~(1 << 5);
    GPIOA->OSPEEDR |= (3 << (5*2));
}

void led_on(void) { GPIOA->BSRR = (1 << 5); }
void led_off(void) { GPIOA->BSRR = (1 << (5 + 16)); }
void led_toggle(void) { GPIOA->ODR ^= (1 << 5); }

// Инициализация TIM2 для прерываний с периодом 500 мс
void tim2_init(void) {
    RCC->APB1ENR |= (1 << 0);  // TIM2 clock

    // Расчёт: APB1=84 MHz, PSC=8399, ARR=4999 → 500 мс
    TIM2->PSC = 8399;    // 84e6 / 8400 = 10 kHz
```

```
    TIM2->ARR = 4999;    // 10 kHz / 5000 = 2 Hz (период 500 мс)
    TIM2->CNT = 0;

    TIM2->DIER |= (1 << 0);  // Update interrupt enable

    // Настройка NVIC для TIM2 (IRQ 28)
    NVIC->ISER[0] |= (1 << 28);  // Enable TIM2 interrupt (ISER0 содержит IRQ0-31)
    NVIC->IPR[28] = (1 << 6);     // Приоритет = 64 (средний, 4-битный)

    TIM2->CR1 |= (1 << 0);  // Enable counter
}

// Инициализация кнопки на PA0 с прерыванием по нарастающему фронту
void button_init(void) {
    // Тактирование GPIOA
    RCC->AHB1ENR |= (1 << 0);

    // PA0 как вход с pull-down (кнопка подключает +3.3V)
    GPIOA->MODER &= ~(3 << (0*2));
    GPIOA->PUPDR &= ~(3 << (0*2));
    GPIOA->PUPDR |= (2 << (0*2));  // Pull-down

    // Подключение PA0 к EXTI0 (SYSCFG)
    // Упрощённо: в реальном проекте нужен SYSCFG_EXTICR1

    // Настройка EXTI0 на прерывание по нарастающему фронту
    EXTI->RTSR |= (1 << 0);   // Rising trigger
    EXTI->FTSR &= ~(1 << 0);  // Falling disable
```

```
    EXTI->IMR |= (1 << 0);    // Interrupt not masked

    // NVIC для EXTI0 (IRQ 6)
    NVIC->ISER[0] |= (1 << 6);
    NVIC->IPR[6] = (2 << 6);  // Приоритет = 128 (ниже, чем у TIM2)
}

// Обработчик прерывания TIM2
void TIM2_IRQHandler(void) {
    if (TIM2->SR & (1 << 0)) {
        TIM2->SR &= ~(1 << 0);  // Clear flag
        tim2_interrupt_count++;

        // Критическая секция для shared_variable
        __disable_irq();
        shared_variable++;
        __enable_irq();

        led_toggle();  // Переключение светодиода
    }
}

// Обработчик прерывания EXTI0 (кнопка)
void EXTI0_IRQHandler(void) {
    if (EXTI->PR & (1 << 0)) {
        EXTI->PR = (1 << 0);  // Clear pending bit
        exti0_interrupt_count++;

        // Демонстрация вложенности: задержка (имитация длительной обработки)
```

```
            // В реальном проекте так делать не следует.
            delay(1000000);

            // Меняем приоритет работы светодиода (инвертируем ~50% времени)
            // Просто для демонстрации
        }
    }

    // Обработчик сброса (точка входа)
    void Reset_Handler(void) {
        // Копирование .data из Flash в RAM (упрощённо)
        // Обнуление .bss (упрощённо)

        // Инициализация
        led_init();
        tim2_init();
        button_init();

        // Главный цикл
        while (1) {
            // Демонстрация работы программного прерывания
            static uint32_t last_software_trigger = 0;
            if (shared_variable > 10 && !last_software_trigger) {
                // Вызов программного прерывания TIM2
                NVIC->STIR = 28;   // Software trigger TIM2
                last_software_trigger = 1;
            }

            // Индикация счётчиков (можно через UART)
```

```
        // (в данной версии просто пустой цикл)
    }
}
```

3. **Изучение вложенных прерываний**

3.1. Модифицируйте программу, чтобы увидеть эффект вложенности:

- Установите для TIM2 приоритет выше (меньшее число), чем для EXTI0.
- В обработчике EXTI0 добавьте длительную задержку (delay) и внутри неё наблюдайте, произойдёт ли прерывание от TIM2.
- Поменяйте приоритеты местами и повторите эксперимент.

3.2. Сравните поведение: с более высоким приоритетом TIM2 прервёт обработку кнопки, с более низким – будет ждать её завершения.

4. **Исследование критических секций и гонок данных**

4.1. В обработчике TIM2 добавьте атомарное обновление shared_variable без защиты, а в главном цикле читайте эту переменную.

4.2. Промоделируйте ситуацию, когда прерывание происходит в момент полуобновления переменной (для 32-битных переменных в Cortex-M это атомарно, но для структур или 64-битных – нет). Объясните необходимость защиты.

5. **Программное прерывание (Software Trigger)**

5.1. Добавьте в главный цикл вызов программного прерывания при нажатии кнопки (глобальная переменная-флаг с опросом).

5.2. Используйте регистр NVIC->STIR для генерации прерывания таймера без участия аппаратуры.

5.3. Проверьте, что флаг ожидания (pending) устанавливается, даже если прерывание запрещено (в NVIC_ICER). Затем при разрешении оно будет обработано.

6. **Защита критических секций**

6.1. Реализуйте простой семафор с запретом прерываний:

```
volatile uint8_t lock = 0;

void acquire_lock(void) {
   __disable_irq();
   if (lock == 0) {
      lock = 1;
   }
   __enable_irq();
}

void release_lock(void) {
   __disable_irq();
   lock = 0;
   __enable_irq();
}
```

6.2. Продемонстрируйте использование в ISR и основном коде.

7. **Анализ векторной таблицы в скомпилированном образе**

7.1. Скомпилируйте проект и с помощью arm-none-eabi-objdump -s -j .isr_vector firmware.elf просмотрите содержимое векторной таблицы.

7.2. Убедитесь, что первый DWORD (Initial SP) имеет корректное значение, а второй – адрес Reset_Handler.

Пример выполнения задания

Задача: Разработать систему, в которой прерывание от таймера TIM2 (высокий приоритет) прерывает обработку длительного прерывания от кнопки EXTI0 (низкий приоритет). Измерить количество прерываний таймера, произошедших во время задержки в обработчике кнопки.

Код (фрагмент):

```
volatile uint32_t timer_hits = 0;
volatile uint32_t button_in_progress = 0;

void EXTI0_IRQHandler(void) {
    if (EXTI->PR & (1 << 0)) {
        EXTI->PR = (1 << 0);
        button_in_progress = 1;

        // Длительная операция ~500 мс
        for (uint32_t i = 0; i < 5000000; i++) {
            __asm volatile("nop");
            // TIM2 может прервать этот цикл
        }

        button_in_progress = 0;
    }
}

void TIM2_IRQHandler(void) {
    if (TIM2->SR & (1 << 0)) {
        TIM2->SR &= ~(1 << 0);
        timer_hits++;

        if (button_in_progress) {
```

```
            led_toggle();  // визуальная индикация вложенности
        }
    }
}
```

Результаты (вывод через UART или отладчик):

- Приоритет TIM2 выше (значение 0x80), EXTI0 – ниже (0xC0): timer_hits за время обработки кнопки ~120.
- При обратном приоритете: timer_hits = 0 (прерывания таймера не обрабатываются до завершения EXTI0).

Вывод: Приоритеты NVIC управляют вложенностью; правильно настроенные приоритеты обеспечивают выполнение критичных по времени задач.

Индивидуальные задания

Вариант 1. Два таймера с разными приоритетами

Настройте TIM2 (1 Гц) и TIM3 (10 Гц). TIM3 имеет более высокий приоритет. При каждом прерывании TIM3 переключайте светодиод, TIM2 – изменяйте состояние другого светодиода. Проанализируйте, сколько раз TIM3 прерывает обработку TIM2.

Вариант 2. Защита разделяемой переменной

Два прерывания (два таймера) инкрементируют общую 32-битную переменную. В главном цикле она выводится. Продемонстрируйте, что без защиты __disable_irq() возможны ошибки (для 64-битной переменной). Реализуйте защиту через атомарные операции (__atomic_add_fetch) или через запрет прерываний.

Вариант 3. Программное прерывание от кнопки без NVIC_STIR

Используйте установку флага ожидания через NVIC->ISPR[0] = (1 << IRQn). Реализуйте два варианта: когда прерывание разрешено (ISER) – оно выполняется сразу; когда запрещено – ожидает установки ISER.

Вариант 4. Система сбора данных с защитой от потери прерываний

АЦП запускается по таймеру, прерывание по готовности данных имеет высокий приоритет. В то же время UART принимает данные (низкий приоритет). Покажите, что данные АЦП не теряются, даже если UART занят.

Вариант 5. Измерение времени входа в прерывание

Используя один таймер в режиме free-running (ARR=0xFFFFFFFF), в начале обработчика прерывания читайте CNT, в конце – снова. Вычислите накладные расходы на вызов ISR (сохранение регистров, возврат). Сравните с функцией, вызванной из основного цикла.

Вариант 6. Векторная таблица в ОЗУ и перенаправление прерываний

Скопируйте векторную таблицу в ОЗУ (по адресу 0x20000000) и перенастройте регистр VTOR на этот адрес. Динамически изменяйте обработчики прерываний (например, для разных режимов работы).

Вариант 7. Сторожевой таймер на основе прерывания SysTick

Используйте SysTick с периодом 1 мс. Если основная программа не "пинает" сторожевую переменную в течение 1 секунды, то прерывание SysTick генерирует программный сброс (или вызывает HardFault). Реализуйте системный вызов watchdog_feed().

Вариант 8. Прерывание от UART с кольцевым буфером

Реализуйте приём данных по UART через прерывание. Данные помещаются в кольцевой буфер. Основной цикл читает из буфера. Обеспечьте потокобезопасность (запрет прерываний при доступе к указателям).

Вариант 9. Детектор ложных срабатываний кнопки с таймером

При нажатии кнопки (EXTI) запускается таймер на 20 мс. Если по истечении этого времени кнопка всё ещё нажата – подтверждается событие.

Реализуйте без использования задержек: в обработчике EXTI запретите обработку кнопки на время работы таймера.

Вариант 10. Приоритеты и изменение во время работы

Реализуйте функцию, которая динамически изменяет приоритет TIM2 (через NVIC->IPR) между двумя значениями (высокий/низкий) при каждом нажатии кнопки. Демонстрируйте эффект на вложенности.

Вариант 11. Многоуровневая обработка прерываний (каскад)

Прерывание A вызывает программное прерывание B, которое вызывает C. Каждое изменяет свой светодиод. Установите приоритеты так, чтобы все они обработались в правильном порядке.

Вариант 12. Измерение максимальной частоты прерываний

Настройте таймер на максимальную частоту (PSC=0, ARR=10). В обработчике переключайте вывод и инкрементируйте счётчик. Измерьте максимальную частоту, при которой программа ещё успевает обрабатывать прерывания (без пропусков). Сравните с теоретической.

Вариант 13. Безопасное копирование данных из прерывания в main

Данные объёмом 256 байт копируются в прерывании UART. В main они обрабатываются. Используйте двойной буфер и флаг готовности, защищённый __disable_irq().

Вариант 14. Эмуляция программного прерывания с помощью таймера

Запрограммируйте таймер на однократный выстрел (one-shot) с задержкой 1 мс. По срабатыванию таймера выполняйте нужную функцию. Создайте API software_irq_register(void (*func)(void), uint32_t delay_us).

Вариант 15. Критическая секция с сохранением состояния прерываний

Напишите макросы ENTER_CRITICAL() и EXIT_CRITICAL(), которые сохраняют текущее состояние бита PRIMASK, затем запрещают прерывания, а при выходе восстанавливают. Докажите, что вложенные вызовы работают корректно.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (векторная таблица, NVIC, приоритеты, программные прерывания).
4. Ход выполнения работы с приведением:
   - Листинга файла startup.c (или определения векторной таблицы)
   - Листинга основного модуля (main.c)
   - Команд компиляции и анализа выходных файлов
5. Результаты экспериментов с вложенными прерываниями (таблица или описание поведения при разных приоритетах).
6. Результаты выполнения индивидуального задания:
   - Листинг кода
   - Описание алгоритма
   - Выводы о корректности работы с прерываниями
7. Анализ накладных расходов на вызов ISR (если измеряли).
8. Выводы по работе (что освоено, как влияют приоритеты, когда нужно использовать программные прерывания).
9. Ответы на контрольные вопросы.

Контрольные вопросы

1. Что такое векторная таблица и где она располагается в памяти Cortex-M?
2. Какое назначение первого слова в векторной таблице?
3. Как процессор находит адрес обработчика прерывания?
4. Что такое NVIC и какие функции он выполняет?
5. Как разрешить конкретное прерывание в NVIC? Как запретить?

6. Как задать приоритет прерыванию? Какое числовое значение соответствует наивысшему приоритету?

7. Что происходит, когда во время выполнения ISR поступает прерывание с более высоким приоритетом?

8. Как реализовать программное прерывание без использования STIR?

9. Почему при работе с разделяемыми данными между ISR и основным циклом нужно запрещать прерывания?

10. Что такое "критическая секция"? Как защитить короткую и длительную критическую секцию?

11. Как сохранить и восстановить состояние глобального разрешения прерываний (__disable_irq / __enable_irq)?

12. Какие исключения (прерывания) имеют фиксированные приоритеты и не могут быть изменены?

13. Что такое "pending" состояние прерывания и чем оно отличается от "active"?

14. Можно ли вызвать обработчик прерывания как обычную функцию? Какие могут быть последствия?

15. Какие регистры сохраняются автоматически при входе в прерывание, а какие – программно (PUSH)?

16. Как определить максимальную глубину стека при вложенных прерываниях?

17. Какие типичные ошибки возникают при работе с прерываниями (забыли сбросить флаг, слишком длинный ISR, приоритеты)?

18. Как использовать регистр BASEPRI для запрета прерываний ниже определённого уровня?

19. Что такое "late arrival" и "tail chaining" в контексте NVIC?

20. Зачем нужен бит PRIGROUP в регистре AIRCR и как он влияет на группировку приоритетов?

Лабораторная работа №6

Драйвер светодиодной индикации (State Machine)

Цель работы: освоение методологии конечных автоматов (state machines) при разработке драйверов периферийных устройств, реализация универсального драйвера для управления светодиодом в различных режимах индикации с использованием неблокирующих алгоритмов и аппаратного ШИМ.

Задачи:

- Изучить принципы проектирования конечных автоматов для встраиваемых систем.
- Реализовать конечный автомат, управляющий состояниями светодиода: выключен, включён постоянно, мигает с заданной частотой, плавно затухает/загорается (с ШИМ).
- Освоить переключение между режимами по командам (кнопка, UART, таймер).
- Реализовать неблокирующие задержки и временные интервалы с использованием системного таймера (SysTick).
- Интегрировать ШИМ-модуль для плавного изменения яркости (эффект «дыхания»).
- Создать модульную архитектуру драйвера с чётким разделением логики конечного автомата и аппаратного уровня.

Теоретическая часть

Конечные автоматы (Finite State Machines, FSM)

Конечный автомат – это модель поведения системы, которая в каждый момент времени находится в одном из конечного множества состояний. Переходы между состояниями происходят под действием событий и могут сопровождаться действиями.

Компоненты FSM:

- **Состояния** – режимы работы (например, LED_OFF, LED_ON, LED_BLINK, LED_FADE)
- **События** – триггеры перехода (нажатие кнопки, истечение таймера, команда по UART)
- **Переходы** – правила смены состояния
- **Действия** – выполняемый код при входе в состояние, выходе из него или во время пребывания

Типы конечных автоматов:

- **Мили (Mealy)** – действие зависит от состояния и события
- **Мур (Moore)** – действие зависит только от состояния
- **Смешанные** – наиболее распространены на практике

Реализация FSM на C:

Способ 1: switch-case (наиболее читаемый):

```
typedef enum {
    ST_OFF,
    ST_ON,
    ST_BLINK,
    ST_FADE
} LedState_t;

LedState_t current_state = ST_OFF;

void led_fsm(void) {
    switch (current_state) {
        case ST_OFF:
            led_hardware_off();
            if (event_turn_on) current_state = ST_ON;
            break;
        case ST_ON:
```

```
            led_hardware_on();
            if (event_blink) current_state = ST_BLINK;
            break;
        // ...
    }
}
```

Способ 2: Таблица переходов (более гибкий для большого числа состояний):

```
c
typedef struct {
    LedState_t next_state;
    void (*action)(void);
} Transition_t;

Transition_t transition_table[ST_COUNT][EVENT_COUNT];
```

Применение FSM в драйвере светодиода

Светодиодная индикация часто должна работать асинхронно, не блокируя основную программу. Таймер (SysTick или аппаратный) периодически (например, каждые 10 мс) вызывает функцию led_driver_tick(), которая обновляет состояние автомата и управляет выводами или ШИМ.

Режимы работы светодиода:

1. **OFF** – светодиод выключен (ШИМ- duty = 0%).
2. **ON** – светодиод горит постоянно (ШИМ-duty = 100%).
3. **BLINK** – мигание с заданным периодом (например, 500 мс включён, 500 мс выключен). Используется переменная-таймер.
4. **FADE** – плавное изменение яркости от 0% до 100% и обратно (эффект «дыхания»). Изменение duty циклически.

5. **SOS** – передача сигнала SOS азбукой Морзе (короткие, длинные вспышки, паузы).

Аппаратный ШИМ против программного

Для плавной регулировки яркости лучше использовать аппаратный ШИМ-таймер (см. ЛР4). Программный ШИМ с переключением вывода в прерывании требует точной привязки по времени и может загружать процессор.

Неблокирующие задержки

В автомате нельзя использовать delay(), так как это остановит всю систему. Вместо этого в каждом состоянии хранится timestamp следующего переключения:

uint32_t next_change_time = 0;

if (current_tick >= next_change_time) {

// выполнить действие, установить next_change_time = current_tick + interval

}

Модульная структура драйвера:

led_driver/

├── led_driver.h // интерфейс: инициализация, установка режима, команды

├── led_driver.c // реализация FSM, работа с аппаратурой (GPIO/ШИМ)

├── led_hw.h // низкоуровневые макросы для управления выводами

└── led_hw.c // реализация для конкретного микроконтроллера

Методика выполнения работы

1. **Создание базовых аппаратных функций**

   1.1. Создайте каталог lab6 и файл led_hw.h:

```
#ifndef LED_HW_H
#define LED_HW_H

#include <stdint.h>

// Инициализация вывода светодиода (PA5) и ШИМ (если используется)
void led_hw_init(void);

// Включение/выключение светодиода (для режимов ON/OFF/BLINK)
void led_hw_on(void);
void led_hw_off(void);

// Установка яркости (0..255) через ШИМ
void led_hw_set_brightness(uint8_t brightness);

// Получение текущего значения яркости (для плавных переходов)
uint8_t led_hw_get_brightness(void);

#endif
```

   1.2. Создайте файл led_hw.c (на примере STM32F4 с использованием TIM3_CH2 для PWM):

```
#include "led_hw.h"
#include "stm32f4xx.h"  // регистры (упрощённо)

static uint8_t current_brightness = 0;
```

```
void led_hw_init(void) {
    // Включение тактирования GPIOA и TIM3
    RCC->AHB1ENR |= (1 << 0);  // GPIOA
    RCC->APB1ENR |= (1 << 1);  // TIM3

    // Настройка PA5 как AF2 (TIM3_CH2)
    GPIOA->MODER &= ~(3 << (5*2));
    GPIOA->MODER |= (2 << (5*2));
    GPIOA->AFR[0] |= (2 << (5*4));

    // Настройка TIM3 для ШИМ с частотой 1 кГц
    TIM3->PSC = 83;           // 84 MHz / 84 = 1 MHz
    TIM3->ARR = 999;          // 1 MHz / 1000 = 1 kHz
    TIM3->CCR2 = 0;           // начальная яркость 0
    TIM3->CCMR1 |= (6 << 12);  // PWM mode 1
    TIM3->CCER |= (1 << 4);    // CC2 enable
    TIM3->CR1 |= (1 << 0);     // Enable TIM3
}

void led_hw_on(void) {
    TIM3->CCR2 = TIM3->ARR;    // duty = 100%
    current_brightness = 255;
}

void led_hw_off(void) {
    TIM3->CCR2 = 0;
    current_brightness = 0;
}

void led_hw_set_brightness(uint8_t brightness) {
```

```
    uint32_t duty = (TIM3->ARR * brightness) / 255;
    TIM3->CCR2 = duty;
    current_brightness = brightness;
}

uint8_t led_hw_get_brightness(void) {
    return current_brightness;
}
```

2. **Реализация конечного автомата драйвера**

2.1. Создайте файл led_driver.h:

c

```
#ifndef LED_DRIVER_H
#define LED_DRIVER_H

#include <stdint.h>
#include <stdbool.h>

// Режимы работы светодиода
typedef enum {
    LED_MODE_OFF = 0,
    LED_MODE_ON,
    LED_MODE_BLINK,
    LED_MODE_FADE
} LedMode_t;

// Инициализация драйвера (включая аппаратуру)
void led_driver_init(void);

// Установка режима
```

```
void led_driver_set_mode(LedMode_t mode);

// Установка параметров для мигания (период в миллисекундах, скважность 50%)
void led_driver_set_blink_period(uint32_t period_ms);

// Установка скорости затухания (время полного цикла "дыхания")
void led_driver_set_fade_speed(uint32_t cycle_time_ms);

// Функция, которую необходимо вызывать периодически (например, каждые 10 мс)
void led_driver_tick(uint32_t current_tick_ms);

// Получение текущего режима (для отладки)
LedMode_t led_driver_get_mode(void);

#endif
```

2.2. Создайте файл led_driver.c:

```
#include "led_driver.h"
#include "led_hw.h"

// Состояния FSM для режима BLINK
typedef enum {
    BLINK_PHASE_OFF,
    BLINK_PHASE_ON
} BlinkPhase_t;

// Состояния FSM для режима FADE
typedef enum {
```

```
    FADE_PHASE_INC,
    FADE_PHASE_DEC
} FadePhase_t;

// Текущие параметры драйвера
static LedMode_t current_mode = LED_MODE_OFF;
static uint32_t blink_period_ms = 500;   // полный период (ON+OFF)
static uint32_t fade_cycle_ms = 2000;    // время полного цикла (inc+dec)

// Переменные для неблокирующих задержек
static uint32_t next_blink_change_ms = 0;
static uint32_t next_fade_update_ms = 0;

// Состояния подрежимов
static BlinkPhase_t blink_phase = BLINK_PHASE_OFF;
static FadePhase_t fade_phase = FADE_PHASE_INC;
static uint8_t fade_brightness = 0;

void led_driver_init(void) {
    led_hw_init();
    led_hw_off();
    current_mode = LED_MODE_OFF;
}

void led_driver_set_mode(LedMode_t mode) {
    // Выход из текущего режима (сброс временных переменных)
    switch (current_mode) {
        case LED_MODE_BLINK:
            // Принудительно выключаем светодиод
            led_hw_off();
```

```
        break;
    case LED_MODE_FADE:
        // Останавливаем плавное изменение
        break;
    default:
        break;
}

current_mode = mode;

// Вход в новый режим
switch (current_mode) {
    case LED_MODE_OFF:
        led_hw_off();
        break;
    case LED_MODE_ON:
        led_hw_on();
        break;
    case LED_MODE_BLINK:
        // Начинаем с включённой фазы
        blink_phase = BLINK_PHASE_ON;
        led_hw_on();
        next_blink_change_ms = 0;  // установим при первом тике
        break;
    case LED_MODE_FADE:
        fade_phase = FADE_PHASE_INC;
        fade_brightness = 0;
        led_hw_set_brightness(0);
        next_fade_update_ms = 0;
        break;
```

```
    }
}

void led_driver_set_blink_period(uint32_t period_ms) {
    if (period_ms < 20) period_ms = 20;  // ограничение
    blink_period_ms = period_ms;
}

void led_driver_set_fade_speed(uint32_t cycle_time_ms) {
    if (cycle_time_ms < 100) cycle_time_ms = 100;
    fade_cycle_ms = cycle_time_ms;
}

void led_driver_tick(uint32_t current_tick_ms) {
    switch (current_mode) {
        case LED_MODE_BLINK:
            if (next_blink_change_ms == 0) {
                next_blink_change_ms = current_tick_ms + (blink_period_ms / 2);
            }
            if (current_tick_ms >= next_blink_change_ms) {
                // Переключение фазы
                if (blink_phase == BLINK_PHASE_ON) {
                    led_hw_off();
                    blink_phase = BLINK_PHASE_OFF;
                } else {
                    led_hw_on();
                    blink_phase = BLINK_PHASE_ON;
                }
                next_blink_change_ms = current_tick_ms + (blink_period_ms / 2);
            }
```

```
        break;

    case LED_MODE_FADE:
        if (next_fade_update_ms == 0) {
            next_fade_update_ms = current_tick_ms + (fade_cycle_ms / 256);
        }
        if (current_tick_ms >= next_fade_update_ms) {
            // Изменение яркости на 1/256 за интервал
            if (fade_phase == FADE_PHASE_INC) {
                if (fade_brightness < 255) {
                    fade_brightness++;
                } else {
                    fade_phase = FADE_PHASE_DEC;
                }
            } else {
                if (fade_brightness > 0) {
                    fade_brightness--;
                } else {
                    fade_phase = FADE_PHASE_INC;
                }
            }
            led_hw_set_brightness(fade_brightness);
            next_fade_update_ms = current_tick_ms + (fade_cycle_ms / 256);
        }
        break;

    default:
        // OFF и ON не требуют периодических действий
        break;
}
```

```
}

LedMode_t led_driver_get_mode(void) {
   return current_mode;
}
```

3. **Создание тестовой программы с переключением режимов**

   3.1. Создайте файл main.c:

   c

```
#include "led_driver.h"
#include "timers.h"     // из ЛР4 (SysTick)
#include "gpio_driver.h" // для кнопки (из ЛР3)

// Обработчик кнопки (простой конечный автомат для смены режимов)
typedef enum {
   BTN_IDLE,
   BTN_DEBOUNCE
} ButtonState_t;

static ButtonState_t btn_state = BTN_IDLE;
static uint32_t btn_press_tick = 0;

void check_button(uint32_t current_tick) {
   if (GPIO_ReadPin(GPIOC, 13) == 0) {  // кнопка нажата (pull-up, 0 = нажата)
      if (btn_state == BTN_IDLE) {
         btn_state = BTN_DEBOUNCE;
         btn_press_tick = current_tick;
      } else if (btn_state == BTN_DEBOUNCE && (current_tick - btn_press_tick) > 50) {
```

```
                // подтверждение нажатия (50 мс)
                // Смена режима циклически
                LedMode_t mode = led_driver_get_mode();
                switch (mode) {
                    case LED_MODE_OFF: led_driver_set_mode(LED_MODE_ON);
break;
                    case                                    LED_MODE_ON:
led_driver_set_mode(LED_MODE_BLINK); break;
                    case                                LED_MODE_BLINK:
led_driver_set_mode(LED_MODE_FADE); break;
                    case                                 LED_MODE_FADE:
led_driver_set_mode(LED_MODE_OFF); break;
                }
                btn_state = BTN_IDLE;
            }
        } else {
            btn_state = BTN_IDLE;  // кнопка отпущена
        }
    }

    int main(void) {
        // Инициализация системного таймера (тик каждую 1 мс)
        SysTick_Init(1000);

        // Инициализация драйвера светодиода
        led_driver_init();

        // Инициализация кнопки (GPIO PC13 с подтяжкой вверх)
        RCC->AHB1ENR |= (1 << 2);  // GPIOC
        GPIO_Init_t btn_cfg = {
```

```
        .pin = 13,
        .mode = GPIO_MODE_INPUT,
        .pull = GPIO_PULL_UP
    };
    GPIO_Init(GPIOC, &btn_cfg);

    // Задание параметров режимов
    led_driver_set_blink_period(1000);  // мигание с периодом 1 секунда
    led_driver_set_fade_speed(3000);    // полный цикл "дыхания" 3 секунды

    uint32_t last_tick = 0;

    while (1) {
        uint32_t now = SysTick_GetTick();

        // Вызов тика драйвера (периодически, но не реже 10 мс)
        static uint32_t last_driver_tick = 0;
        if ((now - last_driver_tick) >= 10) {
            led_driver_tick(now);
            last_driver_tick = now;
        }

        // Обработка кнопки (тоже каждые 10 мс)
        check_button(now);

        // Можно добавить другие задачи...
    }
}
```

4. **Расширение возможностей: добавление режима SOS (азбука Морзе)**

4.1. Добавьте новый режим LED_MODE_SOS. Логика: последовательность: точка (короткая вспышка, 200 мс), пауза 200 мс, точка, пауза, точка, пауза 400 мс, тире (длинная, 600 мс), пауза 200 мс, тире, пауза, тире, пауза 400 мс, и т.д.

4.2. Реализуйте как конечный автомат внутри led_driver_tick().

5. **Исследование нагрузки на процессор**

5.1. Измерьте, сколько времени тратится в функции led_driver_tick() при разных режимах (используйте вывод переключения GPIO в начале и конце функции, измеряйте осциллографом).

5.2. Сравните с программной реализацией ШИМ (переключение в прерывании таймера) и аппаратным ШИМ.

Пример выполнения задания

Задача: Реализовать светодиодную индикацию состояний системы:

- OFF (выключен) → постоянное горение → медленное мигание (1 Гц) → быстрое мигание (4 Гц) → «дыхание» (цикл 2 с) → обратно OFF.

Решение (фрагмент):

```
// расширение enum режимов
typedef enum {
    LED_MODE_OFF,
    LED_MODE_ON,
    LED_MODE_BLINK_SLOW,
    LED_MODE_BLINK_FAST,
    LED_MODE_FADE
} LedMode_t;

// в main()
void set_mode_by_index(uint8_t idx) {
```

```
    switch(idx) {
        case 0: led_driver_set_mode(LED_MODE_OFF); break;
        case 1: led_driver_set_mode(LED_MODE_ON); break;
        case 2:
            led_driver_set_mode(LED_MODE_BLINK_SLOW);
            led_driver_set_blink_period(1000);
            break;
        case 3:
            led_driver_set_mode(LED_MODE_BLINK_FAST);
            led_driver_set_blink_period(250);
            break;
        case 4: led_driver_set_mode(LED_MODE_FADE); break;
    }
}
```

Результат: При каждом нажатии кнопки светодиод меняет характер свечения. Выводы: конечный автомат обеспечивает чёткое поведение, легко добавлять новые режимы, не меняя основную логику. Аппаратный ШИМ при плавном затухании не нагружает процессор (в отличие от программного).

Индивидуальные задания

Вариант 1. Индикация загрузки процессора

Реализуйте режим, в котором яркость светодиода пропорциональна загрузке CPU (измеряется в фоновой задаче). Используйте FADE как базу, управляя целевой яркостью.

Вариант 2. Сигнальный маяк с чередованием цветов (RGB)

Используйте RGB-светодиод. Реализуйте конечный автомат с состояниями: красный, зелёный, синий, жёлтый, выкл. Переключение по таймеру каждые 2 секунды.

Вариант 3. Индикация уровня заряда батареи (4 уровня)

Заряд 0-25% – мигание 1 Гц, 25-50% – медленное дыхание, 50-75% – постоянное горение 50% яркости, 75-100% – постоянное горение 100% яркости.

Вариант 4. Реакция на два независимых события

Два разных события (например, датчик движения и датчик открытия двери) должны вызывать разные паттерны мигания (один – 2 коротких вспышки, другой – 1 длинную). Реализуйте очередь событий и приоритеты.

Вариант 5. «Светофор» для уведомлений

При поступлении сообщения по UART светодиод должен мигнуть 3 раза с коротким интервалом, затем вернуться к предыдущему режиму.

Вариант 6. Плавное переключение между режимами

При смене режима (например, с OFF на ON) яркость должна увеличиваться плавно за 500 мс. Реализуйте дополнительные промежуточные состояния автомата.

Вариант 7. Индикатор «Присутствие» (PIR-датчик)

В режиме ожидания – медленное дыхание. При обнаружении движения – яркая вспышка на 500 мс, затем возврат к дыханию.

Вариант 8. Режим «Ночник» (адаптивная яркость)

Уровень освещённости измеряется фоторезистором (АЦП). В режиме ON яркость светодиода автоматически подстраивается так, чтобы общая освещённость была постоянной.

Вариант 9. Кнопка с двойным нажатием

Одиночное нажатие переключает режимы, двойное нажатие – задаёт параметры текущего режима (частоту мигания или скорость затухания). Реализуйте конечный автомат для распознавания двойного нажатия.

Вариант 10. Драйвер адресной светодиодной ленты (WS2812)

Вместо одного светодиода используйте ленту WS2812. Реализуйте конечный автомат для отдельных эффектов: бегущий огонь, заливка цветом, стробоскоп. Используйте DMA таймера для генерации сигнала.

Вариант 11. Морзянка с программируемым сообщением

В режиме SOS – передача «SOS». Дополнительно реализуйте возможность задать произвольное текстовое сообщение, которое переводится в азбуку Морзе и отображается светодиодом.

Вариант 12. Эмуляция свечения накала (random flicker)

Создайте режим, имитирующий колебания яркости газонаполненной лампы (случайные изменения яркости около среднего значения). Используйте LFSR для генерации псевдослучайных чисел.

Вариант 13. Синхронизация двух светодиодов (ведущий-ведомый)

Два светодиода на разных выводах должны синхронно выполнять одинаковые режимы. Один является мастером, другой – слейвом (получает команды по шине I2C или через глобальные переменные). Реализуйте протокол синхронизации.

Вариант 14. «Музыкальный свет» (FFT)

Через АЦП оцифровывается аудиосигнал. Вычисляется амплитуда в низких, средних и высоких частотах. Три светодиода (RGB) отображают уровень каждой полосы.

Вариант 15. Программируемый паттерн мигания через UART

Пользователь посылает строку вида «500,200,300» где числа – длительности включённого состояния в мс. Драйвер циклически воспроизводит этот паттерн.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.

2. Цель и задачи лабораторной работы.

3. Теоретическую часть (конечные автоматы, неблокирующие алгоритмы, аппаратный ШИМ).

4. Листинги разработанных модулей: led_hw.c, led_driver.c, main.c.

5. Диаграмму конечного автомата (состояния, события, переходы) в виде рисунка или таблицы.

6. Описание реализации индивидуального задания с пояснением изменений в автомате.

7. Результаты измерения нагрузки на процессор (количество вызовов led_driver_tick и длительность выполнения).

8. Выводы по работе (достоинства и недостатки FSM-подхода, возможности расширения).

9. Ответы на контрольные вопросы.

Контрольные вопросы

1. Что такое конечный автомат? Приведите примеры применения во встраиваемых системах.

2. В чём отличие автомата Мура от автомата Мили? Какой удобнее для драйвера светодиода?

3. Почему в реализации драйвера используются неблокирующие задержки, а не delay()?

4. Как выполняется периодический вызов led_driver_tick()? Как часто его нужно вызывать?

5. Какие преимущества даёт разделение на аппаратный уровень (led_hw) и логику (led_driver)?

6. Как в текущей реализации организовано переключение фаз мигания? Как изменяется период?

7. Как реализован режим FADE? Как рассчитывается шаг изменения яркости?

8. Как добавить новый режим (например, стробоскоп или «бегущий огонь»)? Какие изменения нужно внести?

9. Что произойдёт, если вызов led_driver_tick() будет происходить нерегулярно (например, с большими пропусками)?

10. Как можно реализовать одновременную работу двух независимых светодиодов с разными режимами?

11. В каких случаях предпочтительнее использовать программный ШИМ, а в каких – аппаратный?

12. Как защитить разделяемые данные (например, current_mode) от одновременного доступа из прерывания и main?

13. Как реализовать «сброс» таймера при смене режима мигания, чтобы фаза не продолжалась с середины?

14. Опишите, как будет вести себя автомат при переходе из BLINK в FADE и обратно.

15. Как измерить потребляемую мощность светодиода в разных режимах? Какой режим наиболее энергоэффективен?

Лабораторная работа №7

Программная реализация ШИМ без аппаратного модуля

Цель работы: освоение метода генерации широтно-импульсной модуляции (ШИМ) программными средствами с использованием таймера и прерываний, анализ недостатков и ограничений программного подхода по сравнению с аппаратным ШИМ, оценка нагрузки на процессор и точности формирования сигнала.

Задачи:

- Изучить принципы программной генерации ШИМ-сигнала на базе таймера общего назначения.
- Реализовать драйвер программного ШИМ с поддержкой переменной скважности и частоты.
- Проанализировать зависимость точности ШИМ от частоты прерываний и тактовой частоты процессора.
- Измерить загрузку процессора при различных параметрах программного ШИМ.
- Сравнить программный ШИМ с аппаратным (из лабораторной работы №4) по следующим критериям: точность, максимальная частота, нагрузка на CPU, джиттер, простота реализации.

Теоретическая часть

Принцип программного ШИМ

Широтно-импульсная модуляция (PWM) – это метод управления мощностью нагрузки путём быстрого переключения вывода между высоким и низким уровнями с изменяемой скважностью.

При программной реализации отсутствует выделенный аппаратный модуль, генерирующий сигнал автоматически. Вместо этого:

- Таймер настраивается на периодические прерывания.
- В обработчике прерывания программно переключается вывод GPIO.

- Скважность задаётся отношением времени высокого уровня к периоду.

Методы программной генерации ШИМ

1. **Метод «один таймер + переменная»**
   Таймер генерирует прерывания с частотой, равной частоте ШИМ. В каждом прерывании сравнивается счётчик (инкрементируемый в том же прерывании) с порогом (duty). Если счётчик меньше порога – вывод HIGH, иначе LOW. При достижении периода счётчик сбрасывается.
2. **Метод «два сравнения» (более точный)**
   Используется таймер с регистром сравнения (Output Compare). Таймер тактируется от системной частоты. Прерывание по совпадению таймера с порогом переключает вывод. Требует двух сравнений на период (включить и выключить) или одного прерывания с программной установкой следующего порога.
3. **Метод «битовой маски» (для нескольких каналов)**
   Один обработчик прерывания обновляет состояние всех ШИМ-каналов, используя один общий счётчик.

Ограничения программного ШИМ

Параметр	Программный ШИМ	Аппаратный ШИМ
Частота	Ограничена скоростью обработки прерываний (~10-50 кГц для Cortex-M)	До сотен кГц – МГц (зависит от таймера)
Разрешение (число уровней)	~100-200 при частоте 1 кГц	16 бит (65536 уровней)
Загрузка CPU	~100% на высоких частотах	0% (работает независимо)
Джиттер	Есть (зависит от других прерываний)	Минимальный (аппаратный)

Параметр	Программный ШИМ	Аппаратный ШИМ
Количество каналов	Легко масштабируется (цена – CPU)	Ограничено аппаратно

Расчёт параметров программного ШИМ

Допустим, желаемая частота ШИМ – f_pwm, тактовая частота таймера (частота прерываний) должна быть:

f_timer = f_pwm * N

где N – количество дискретов на период (разрешение, максимальное значение duty). Тогда разрешение в битах: log2(N).

Пример: f_pwm = 1 кГц, N = 256 → f_timer = 256 кГц → период прерывания = 3.9 мкс. При тактовой частоте CPU 84 МГц на одно прерывание приходится ~328 тактов – достаточно для переключения вывода, но много для других задач.

Реализация на базе SysTick

SysTick – 24-битный таймер. Можно использовать его для программного ШИМ. Однако SysTick обычно занят системными тиками, лучше взять TIM2 или другой таймер общего назначения.

Управление скважностью

Скважность (duty cycle) задаётся значением от 0 до N−1 или до N. При duty = 0 – всегда LOW, при duty = N – всегда HIGH. В методах со сравнением порога: вывод HIGH пока счётчик < duty.

Джиттер (временное дрожание)

Джиттер возникает из-за того, что прерывание может быть задержано другими прерываниями с более высоким приоритетом. Для уменьшения джиттера нужно:

- Установить высший приоритет для ШИМ-таймера.
- Ограничить время выполнения других ISR.
- Использовать метод «двойного сравнения» с аппаратным переключением (если доступен Output Compare).

Сравнение с аппаратным ШИМ (ЛР4)

В ЛР4 использовался аппаратный ШИМ на TIM3: достаточно настроить регистры, и таймер сам управляет выводом. Программный ШИМ требует постоянного внимания CPU, но позволяет гибко менять частоту и количество каналов без дополнительной периферии.

Методика выполнения работы

1. **Создание базового драйвера программного ШИМ на TIM2**

   1.1. Создайте каталог lab7 и файл sw_pwm.h:

```
#ifndef SW_PWM_H
#define SW_PWM_H

#include <stdint.h>
#include <stdbool.h>

// Инициализация программного ШИМ на указанном выводе
// freq_hz - частота ШИМ в Гц (1..20000)
// resolution - разрешение (число уровней), например 256
void sw_pwm_init(uint8_t pin, uint32_t freq_hz, uint16_t resolution);

// Установить скважность в процентах (0..100)
void sw_pwm_set_duty_percent(uint8_t pin, uint8_t percent);
```

```
// Установить скважность в "сырых" отсчётах (0 .. resolution-1)
void sw_pwm_set_duty_raw(uint8_t pin, uint16_t duty);

// Остановить ШИМ (вывод в 0)
void sw_pwm_stop(uint8_t pin);

// Запустить ШИМ (возобновить)
void sw_pwm_start(uint8_t pin);

// Функция, вызываемая из прерывания таймера (один раз на период)
void sw_pwm_timer_isr(void);

#endif
```

1.2. Создайте файл sw_pwm.c:

```
#include "sw_pwm.h"
#include "gpio_driver.h"  // для управления выводами

#define MAX_PWM_CHANNELS 8

typedef struct {
    uint8_t pin;
    bool enabled;
    uint16_t duty;       // текущее значение duty (0..resolution-1)
    uint16_t counter;    // текущий счётчик (0..resolution-1)
    uint16_t resolution;
    uint32_t period_ticks; // не используется напрямую
} PWM_Channel_t;
```

```
static PWM_Channel_t channels[MAX_PWM_CHANNELS];
static uint8_t channel_count = 0;
static uint32_t pwm_freq_hz = 1000;
static uint16_t pwm_resolution = 256;

// Внешняя ссылка на таймер (будет настроен в main)
// Здесь предполагаем, что TIM2 уже настроен на частоту прерываний = pwm_freq_hz * resolution

void sw_pwm_init(uint8_t pin, uint32_t freq_hz, uint16_t resolution) {
    if (channel_count >= MAX_PWM_CHANNELS) return;

    // Сохраняем параметры для всех каналов (единые частота и разрешение)
    pwm_freq_hz = freq_hz;
    pwm_resolution = resolution;

    // Настройка вывода как выход
    GPIO_Init_t out_cfg = {
        .pin = pin,
        .mode = GPIO_MODE_OUTPUT,
        .otype = GPIO_OTYPE_PUSH_PULL,
        .speed = GPIO_SPEED_HIGH,
        .pull = GPIO_PULL_NONE
    };
    GPIO_Init(get_gpio_port_by_pin(pin), &out_cfg);

    // Заполнение структуры канала
    channels[channel_count].pin = pin;
    channels[channel_count].enabled = true;
```

```
    channels[channel_count].duty = resolution / 2;  // 50% по умолчанию
    channels[channel_count].counter = 0;
    channels[channel_count].resolution = resolution;

    channel_count++;
}

void sw_pwm_set_duty_raw(uint8_t pin, uint16_t duty) {
    for (int i = 0; i < channel_count; i++) {
        if (channels[i].pin == pin) {
            if (duty > channels[i].resolution) duty = channels[i].resolution;
            channels[i].duty = duty;
            break;
        }
    }
}

void sw_pwm_set_duty_percent(uint8_t pin, uint8_t percent) {
    for (int i = 0; i < channel_count; i++) {
        if (channels[i].pin == pin) {
            uint16_t duty = (channels[i].resolution * percent) / 100;
            channels[i].duty = duty;
            break;
        }
    }
}

void sw_pwm_stop(uint8_t pin) {
    for (int i = 0; i < channel_count; i++) {
        if (channels[i].pin == pin) {
```

```
            channels[i].enabled = false;
            // Выключить вывод
            GPIO_ResetPin(get_gpio_port_by_pin(pin), pin);
            break;
        }
    }
}

void sw_pwm_start(uint8_t pin) {
    for (int i = 0; i < channel_count; i++) {
        if (channels[i].pin == pin) {
            channels[i].enabled = true;
            break;
        }
    }
}

// Вызов из прерывания таймера (частота = pwm_freq_hz * resolution)
void sw_pwm_timer_isr(void) {
    for (int i = 0; i < channel_count; i++) {
        if (!channels[i].enabled) continue;

        // Обновляем счётчик
        channels[i].counter++;
        if (channels[i].counter >= channels[i].resolution) {
            channels[i].counter = 0;
        }

        // Устанавливаем вывод в состояние HIGH если counter < duty
        if (channels[i].counter < channels[i].duty) {
```

```
                GPIO_SetPin(get_gpio_port_by_pin(channels[i].pin),
channels[i].pin);
            } else {
                GPIO_ResetPin(get_gpio_port_by_pin(channels[i].pin),
channels[i].pin);
            }
        }
    }
```

2. **Настройка таймера для генерации прерываний**

2.1. Настройте TIM2 на частоту f_timer = pwm_freq_hz * resolution. Например, при 1 кГц и resolution=256 → 256 кГц.

```
// В main.c (или отдельном модуле)
void timer_pwm_setup(uint32_t pwm_freq, uint16_t resolution) {
    uint32_t timer_freq = pwm_freq * resolution;
    uint32_t timer_clock = 84000000;  // APB1 для TIM2
    uint32_t prescaler = (timer_clock / timer_freq) - 1;

    RCC->APB1ENR |= (1 << 0);  // TIM2 clock
    TIM2->PSC = prescaler;
    TIM2->ARR = 1;            // autoreload = 1 (прерывание каждый тик)
    TIM2->CNT = 0;
    TIM2->DIER |= (1 << 0);    // Update interrupt enable
    TIM2->CR1 |= (1 << 0);     // Enable counter

    NVIC_EnableIRQ(TIM2_IRQn);
    NVIC_SetPriority(TIM2_IRQn,  0);  // высокий приоритет для
минимизации джиттера
}
```

```
void TIM2_IRQHandler(void) {
   if (TIM2->SR & (1 << 0)) {
      TIM2->SR &= ~(1 << 0);
      sw_pwm_timer_isr();
   }
}
```

3. **Измерение нагрузки на процессор**

3.1. Перед входом в ISR установите вывод в 1, после выхода – в 0. Подключите осциллограф к этому выводу и измерьте время выполнения ISR. Нагрузка CPU = (длительность ISR) * (частота прерываний).

3.2. Альтернативно, используйте счётчик тактов:

```
uint32_t start = DWT->CYCCNT;
sw_pwm_timer_isr();
uint32_t cycles = DWT->CYCCNT - start;
```

4. **Эксперимент с различными частотами и разрешениями**

Частота ШИМ (Гц)	Разрешение	Частота прерываний (кГц)	Ожидаемая нагрузка CPU (при 84 МГц, 100 тактов на ISR)
100	100	10	0.012%
1000	256	256	0.3%
10000	256	2560	3%
20000	256	5120	6%
50000	100	5000	6%

(Расчёты приблизительные, зависит от длительности ISR)

5. **Сравнение с аппаратным ШИМ**

5.1. Используйте аппаратный ШИМ из ЛР4 на том же выводе (например, TIM3_CH2). Сравните:

- Максимальную достижимую частоту без ухудшения формы сигнала.
- Качество сигнала (замер джиттера осциллографом).
- Загрузку CPU.
- Точность задания скважности.

5.2. Зафиксируйте результаты в таблице.

6. **Многоканальный программный ШИМ**

6.1. Добавьте второй и третий канал (разные выводы). Проанализируйте, как увеличивается нагрузка CPU и возникает ли взаимовлияние каналов (джиттер из-за обработки всех каналов в одном ISR).

7. **Программный ШИМ с использованием Output Compare (более точный)**

7.1. Альтернативный метод: настройте таймер в режим сравнения. При совпадении CNT с CCR переключайте вывод и вычисляйте следующий порог. Этот метод даёт меньше джиттера, но сложнее в реализации для нескольких каналов.

```
void TIM2_IRQHandler(void) {
  if (TIM2->SR & (1 << 1)) {  // CC1IF
    TIM2->SR &= ~(1 << 1);
    // Переключение вывода и установка следующего CCR
  }
}
```

Пример выполнения задания

Задача: Реализовать программный ШИМ на частоте 1 кГц с разрешением 100 для управления яркостью светодиода. Плавно изменять скважность от 0 до 100% и обратно.

Код (main.c):

```
#include "sw_pwm.h"
#include "timers.h"  // SysTick для задержек

int main(void) {
   // Инициализация программного ШИМ на PA5
   sw_pwm_init(5, 1000, 100);

   // Настройка таймера TIM2 на 1000*100 = 100 кГц
   timer_pwm_setup(1000, 100);

   uint8_t duty = 0;
   int8_t step = 1;

   while (1) {
      sw_pwm_set_duty_percent(5, duty);
      SysTick_DelayMs(20);
      duty += step;
      if (duty >= 100) { duty = 100; step = -1; }
      if (duty <= 0) { duty = 0; step = 1; }
   }
}
```

Результаты измерений (для частоты 1 кГц, разрешение 100):

- Частота прерываний: 100 кГц (период 10 мкс).

- Длительность ISR: ~75 тактов (на 1 канал, оптимизация -O2) → 75/84e6 ≈ 0.89 мкс.
- Нагрузка CPU: 0.89 мкс * 100000 = 89 мс/секунду → 8.9%.
- Джиттер на осциллографе: ±2 мкс (из-за других прерываний, если они есть).

Сравнение с аппаратным ШИМ (TIM3, 1 кГц, 16-бит):

- Нагрузка CPU: 0% (не считая настройки регистров).
- Джиттер: < 0.1 мкс (аппаратный).
- Максимальная частота: до 84 МГц / 2 = 42 МГц (при минимальном разрешении).

Вывод: Программный ШИМ подходит для невысоких частот (до 1-5 кГц) и малого количества каналов, когда аппаратных ресурсов не хватает. Для точных и высокочастотных применений необходим аппаратный ШИМ.

Индивидуальные задания

Вариант 1. Программный ШИМ на SysTick (вместо TIM2)

Реализуйте программный ШИМ, используя SysTick. SysTick обычно занят системным временем, поэтому нужно аккуратно совмещать. Оцените максимальную частоту при разрешении 100.

Вариант 2. Зависимость максимальной частоты от разрешения

Измерьте экспериментально (с помощью осциллографа или логического анализатора) максимальную частоту ШИМ, при которой форма сигнала ещё приемлема (нет пропусков импульсов) для разрешений 10, 50, 100, 200. Постройте график.

Вариант 3. Программный ШИМ с аппаратным переключением (Output Compare)

Реализуйте метод с двумя сравнениями: таймер в режиме CTC, два регистра CCR. В прерывании по совпадению переключайте вывод и

вычисляйте следующее значение CCR (для включения и выключения). Сравните джиттер с базовым методом.

Вариант 4. ШИМ с переменной частотой

Реализуйте функцию, которая изменяет частоту программного ШИМ «на лету» (без остановки). Оцените, как это влияет на разрешение и нагрузку CPU.

Вариант 5. Многоканальный ШИМ (8 каналов)

Разверните 8 каналов программного ШИМ на разных выводах. Измерьте нагрузку CPU при частоте 1 кГц и разрешении 100. Сравните с одноканальным вариантом.

Вариант 6. Протокол передачи данных через ШИМ

Реализуйте передачу 8-битного байта данных с помощью программного ШИМ: 0 кодируется как 25% скважности, 1 – как 75%. Частота 1 кГц. На приёмной стороне (другой микроконтроллер) измеряйте скважность и декодируйте.

Вариант 7. Эмуляция ШИМ-диммера для лампы накаливания

Плавное изменение яркости с программным ШИМ, но частота выбрана 100 Гц (чтобы избежать мерцания). Разрешение 100. Добавьте «эффект накала» – экспоненциальную зависимость яркости от скважности.

Вариант 8. Программный ШИМ с прерыванием низкого приоритета

Установите для прерывания ШИМ низкий приоритет, добавьте другое прерывание (например, от UART с высоким приоритетом). Пронаблюдайте джиттер на ШИМ-сигнале. Сделайте выводы.

Вариант 9. Измерение пульсаций напряжения на нагрузке

Подключите RC-фильтр (1 кОм + 10 мкФ) к выходу программного ШИМ. Измерьте осциллографом или АЦП постоянное напряжение при разных скважностях. Вычислите погрешность по сравнению с теорией.

Вариант 10. Драйвер сервопривода (50 Гц) программным ШИМ

Сервоприводы требуют импульсы 0.5–2.5 мс с периодом 20 мс. Реализуйте программный ШИМ с периодом 20 мс и разрешением 200 (шаг 100 мкс). Сравните поведение сервы с аппаратным ШИМ.

Вариант 11. Программный ШИМ с двойной буферизацией

Добавьте возможность обновлять значения duty без "разрыва" периода (синхронно с началом следующего периода). Для этого в ISR используйте теневой регистр.

Вариант 12. Энергосбережение при программном ШИМ

Если все каналы имеют нулевую скважность, отключайте таймер и переводите выводы в 0. При изменении duty – включайте заново. Реализуйте «спящий» режим.

Вариант 13. Программный ШИМ с разрешением 16 бит при 1 кГц

Разрешение 65536 требует частоты прерываний 65.536 МГц, что невозможно для большинства МК. Предложите альтернативный метод (например, использование двух таймеров: один высокочастотный малой разрядности плюс аппаратный ШИМ).

Вариант 14. Генерация синусоидального сигнала методом ШИМ

С помощью программного ШИМ и низкочастотного фильтра сгенерируйте синусоиду 50 Гц. Таблица синуса загружена в память. Скважность изменяется в соответствии с таблицей. Оцените искажения.

Вариант 15. Программный ШИМ для управления шаговым двигателем

Шаговый двигатель требует формирования последовательности импульсов с регулируемой скоростью. Реализуйте программный ШИМ для генерации сигнала STEP (частота от 0 до 1000 Гц) и сигнала DIR. Частота должна плавно меняться (трапецеидальный профиль).

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (методы программного ШИМ, ограничения, расчёт нагрузки).
4. Листинги разработанных модулей (sw_pwm.h, sw_pwm.c, настройка таймера).
5. Результаты измерений:
   - Максимальная достигнутая частота при разрешении 100.
   - Нагрузка CPU при нескольких рабочих точках (таблица/график).
   - Осциллограммы или скриншоты логического анализатора.
6. Сравнительную таблицу программного и аппаратного ШИМ (по 5–6 критериям).
7. Результаты выполнения индивидуального задания.
8. Выводы (целесообразность применения программного ШИМ, его недостатки и область применения).
9. Ответы на контрольные вопросы.

Контрольные вопросы

1. В чём заключается принцип программной генерации ШИМ-сигнала?

2. Как связаны частота ШИМ, разрешение и частота прерываний?

3. Почему при увеличении разрешения (числа уровней) падает максимальная частота ШИМ?

4. Как измерить нагрузку на процессор, создаваемую программным ШИМ?

5. Какие факторы влияют на джиттер программного ШИМ?

6. Как можно уменьшить джиттер без перехода на аппаратный ШИМ?

7. Сравните метод «одного таймера со счётчиком» и метод «Output Compare». Какой точнее и почему?

8. Почему для программного ШИМ не рекомендуется использовать SysTick, если он уже используется для системного времени?

9. Какая максимальная частота ШИМ была достигнута в ваших экспериментах? Почему она ограничена?

10. Что произойдёт, если в ISR программного ШИМ будут выполняться слишком долгие операции?

11. Как программный ШИМ влияет на время отклика системы на другие прерывания?

12. Как организовать многоканальный программный ШИМ? Как меняется нагрузка CPU при увеличении числа каналов?

13. Почему аппаратный ШИМ практически не имеет джиттера?

14. Каким образом можно сгенерировать ШИМ с разрешением 16 бит на частоте 1 кГц программно? Реально ли это на STM32F4?

15. Как изменится форма сигнала, если прерывание от таймера будет иногда пропускать из-за занятости процессора?

16. Для каких приложений оправдано использование программного ШИМ вместо аппаратного?

17. Как реализовать плавное изменение частоты ШИМ без сбоев?

18. Что такое «мерцание» (flicker) при низкой частоте ШИМ? Какую минимальную частоту следует выбирать для управления светодиодами?

19. Можно ли использовать программный ШИМ для управления двигателем постоянного тока? Какие могут быть проблемы?

20. Как в программном ШИМ обеспечить «тихий» старт без выброса тока (плавное нарастание скважности с нуля)?

Лабораторная работа №8

АЦП: программный опрос и прерывания

Цель работы: освоение принципов работы аналого-цифрового преобразователя (АЦП) микроконтроллера, реализация драйвера АЦП с различными режимами запуска (однократный, непрерывный, по таймеру), приобретение навыков цифровой обработки сигналов (усреднение, фильтрация) и сравнение эффективности программного опроса и прерываний при сборе данных.

Задачи:

- Изучить архитектуру и регистры АЦП на примере микроконтроллера STM32F4xx.
- Настроить АЦП для работы в различных режимах: однократное преобразование, непрерывное преобразование, сканирование нескольких каналов.
- Реализовать захват данных АЦП методом программного опроса (polling) и методом прерываний.
- Освоить запуск АЦП от таймера для получения периодических выборок с фиксированной частотой дискретизации.
- Реализовать программное усреднение (скользящее среднее, медианный фильтр) для подавления шумов.
- Оценить производительность и загрузку процессора при разных методах опроса АЦП.

Теоретическая часть

Аналого-цифровой преобразователь (ADC)

АЦП преобразует непрерывный аналоговый сигнал (напряжение) в дискретный цифровой код. В STM32F4 используется 12-битный АЦП с максимальным разрешением 4096 шагов (0–4095). Опорное напряжение обычно 3.3 В, поэтому шаг квантования ≈ 0.805 мВ.

Основные параметры АЦП:

- **Разрешение** – количество бит результата (12, 10, 8, 6 бит)
- **Частота дискретизации** – количество преобразований в секунду (до ~2.4 МГц для STM32F4)
- **Время преобразования** – зависит от тактовой частоты АЦП (максимум 36 МГц) и количества циклов выборки
- **Количество каналов** – до 16 внешних + внутренние (температура, Vref)

Режимы работы АЦП:

1. **Однократный (Single conversion)** – выполняет одно преобразование по команде, затем останавливается.
2. **Непрерывный (Continuous)** – после завершения преобразования автоматически запускает следующее.
3. **Сканирование (Scan)** – последовательно преобразует группу каналов (обычно используется с DMA).
4. **Прерываемый (Injected)** – для «внеплановых» измерений без остановки регулярной группы.

Запуск АЦП (триггеры):

- Программный (SWSTART)
- Внешний сигнал (EXTI)
- Таймеры (TIM1, TIM2, TIM3, TIM4, TIM5, TIM8)

Структура АЦП в STM32F4

Каждый АЦП (ADC1, ADC2, ADC3) имеет регистры:

- SR – статус (EOC, EOS, OVR)
- CR1, CR2 – управление (разрешение, режим, выравнивание, DMA, прерывания)
- SMPR1, SMPR2 – время выборки для каждого канала (3–480 циклов)
- SQR1, SQR2, SQR3 – последовательность каналов для сканирования

- DR – регистр данных (результат преобразования)
- CCR – общий регистр для всех АЦП

Программный опрос (polling):

- Запускаем преобразование, ждём установки флага EOC в цикле.
- Просто, но процессор занят ожиданием.

Метод прерываний:

- Разрешаем прерывание по окончанию преобразования (EOC).
- В обработчике читаем данные и сохраняем в буфер.
- Позволяет процессору выполнять другую работу.

Запуск от таймера:

- Настраиваем таймер на требуемую частоту дискретизации.
- Выход таймера подключаем как внешний триггер АЦП.
- АЦП запускается автоматически, прерывание или DMA забирают результат.

Усреднение и фильтрация:

1. **Простое скользящее среднее** – сумма последних N значений, делённая на N. Подавляет высокочастотные шумы.
2. **Медианный фильтр** – берётся N отсчётов, сортируются, выбирается средний. Хорошо убирает выбросы (импульсные шумы).
3. **Экспоненциальное сглаживание** – out = α * new + (1-α) * out_prev, быстрое и без хранения массива.

DMA с АЦП:

- DMA автоматически переносит данные из DR в память без участия CPU.
- Позволяет собирать большие блоки данных на высокой скорости (например, 100 кГц).

- Требует когерентности кэша (или отключения кэша для области буфера).

Методика выполнения работы

1. **Подготовка заголовочного файла и регистров АЦП**

   1.1. Создайте каталог lab8 и файл adc_driver.h:

```
#ifndef ADC_DRIVER_H
#define ADC_DRIVER_H

#include <stdint.h>
#include <stdbool.h>

// Режимы работы АЦП
typedef enum {
    ADC_MODE_SINGLE,      // однократное преобразование
    ADC_MODE_CONTINUOUS,  // непрерывный режим
    ADC_MODE_TIMER_TRIG   // запуск от таймера
} ADC_Mode_t;

// Количество каналов для сканирования (1..16)
void adc_init(ADC_Mode_t mode, uint8_t channel, uint32_t sample_cycles);

// Запуск однократного преобразования (блокирующий)
uint16_t adc_read_single(uint8_t channel);

// Запуск в непрерывном режиме с прерываниями
void adc_start_continuous(void);
void adc_stop_continuous(void);

// Установка колбэка для обработки данных (вызывается из прерывания)
```

```
void adc_register_callback(void (*callback)(uint16_t value));

// Получение последнего значения (для опроса)
uint16_t adc_get_last_value(void);

// Настройка запуска от таймера (TIMx, частота)
void adc_start_timer_trigger(uint32_t timer_freq_hz);

// Простая фильтрация (скользящее среднее, размер окна)
void adc_enable_averaging(uint8_t window_size);
uint16_t adc_get_filtered(void);

#endif
```

1.2. Создайте файл adc_driver.c с базовой реализацией (фрагменты):

```
#include "adc_driver.h"
#include "stm32f4xx.h"

static volatile uint16_t last_adc_value = 0;
static volatile uint16_t filtered_value = 0;
static uint16_t *sample_buffer = NULL;
static uint8_t buffer_index = 0;
static uint8_t buffer_size = 0;
static bool averaging_enabled = false;
static uint32_t sample_sum = 0;

static void (*user_callback)(uint16_t) = NULL;

void adc_init(ADC_Mode_t mode, uint8_t channel, uint32_t sample_cycles)
{
```

```
// Включение тактирования ADC1 и GPIOA
RCC->APB2ENR |= (1 << 8);  // ADC1
RCC->AHB1ENR |= (1 << 0);  // GPIOA

// Настройка пина PA0 как аналоговый вход
GPIOA->MODER |= (3 << (0*2));  // Analog mode
GPIOA->PUPDR &= ~(3 << (0*2)); // No pull

// Сброс настроек ADC1
ADC1->CR2 = 0;
ADC1->CR1 = 0;

// Настройка разрешения 12 бит, выравнивание вправо
ADC1->CR1 &= ~(3 << 24);  // RES=00 (12-bit)

// Установка времени выборки для канала 0
if (channel <= 9) {
   uint32_t shift = channel * 3;
   ADC1->SMPR2 &= ~(7 << shift);
   ADC1->SMPR2 |= (sample_cycles << shift);
}

// Выбор канала в последовательности (один канал)
ADC1->SQR3 = channel;
ADC1->SQR1 = 0;  // 1 преобразование в последовательности

// Настройка режима
if (mode == ADC_MODE_CONTINUOUS) {
   ADC1->CR2 |= (1 << 1);  // CONT=1
} else if (mode == ADC_MODE_TIMER_TRIG) {
```

```
        ADC1->CR2 |= (1 << 11); // ALIGN? - нет, это EXTSEL настраивается отдельно
    }

    // Включение АЦП
    ADC1->CR2 |= (1 << 0);  // ADON
    // Калибровка (рекомендуется)
    ADC1->CR2 |= (1 << 30); // CAL
    while (ADC1->CR2 & (1 << 30));
}

uint16_t adc_read_single(uint8_t channel) {
    // Настройка канала (упрощённо, предполагаем, что канал уже выбран в SQR)
    ADC1->SR &= ~(1 << 1);  // сброс флага EOC
    ADC1->CR2 |= (1 << 30); // SWSTART (для регулярной группы)
    while (!(ADC1->SR & (1 << 1))); // ждём EOC
    return ADC1->DR;
}

void ADC_IRQHandler(void) {
    if (ADC1->SR & (1 << 1)) { // EOC
        ADC1->SR &= ~(1 << 1);
        uint16_t value = ADC1->DR;
        last_adc_value = value;

        // Фильтрация (скользящее среднее)
        if (averaging_enabled && sample_buffer != NULL) {
            sample_sum -= sample_buffer[buffer_index];
            sample_buffer[buffer_index] = value;
```

```
            sample_sum += value;
            buffer_index = (buffer_index + 1) % buffer_size;
            filtered_value = sample_sum / buffer_size;
        }

        if (user_callback) user_callback(value);
    }
}

void adc_register_callback(void (*callback)(uint16_t)) {
    user_callback = callback;
}

void adc_start_continuous(void) {
    ADC1->CR2 |= (1 << 0);  // ADON
    ADC1->CR2 |= (1 << 30); // SWSTART
    // Разрешение прерывания по EOC
    ADC1->CR1 |= (1 << 5);  // EOCIE
    NVIC_EnableIRQ(ADC_IRQn);
}

void adc_enable_averaging(uint8_t window_size) {
    if (sample_buffer) free(sample_buffer);
    sample_buffer = malloc(window_size * sizeof(uint16_t));
    buffer_size = window_size;
    buffer_index = 0;
    sample_sum = 0;
    for (int i = 0; i < window_size; i++) sample_buffer[i] = 0;
    averaging_enabled = true;
}
```

2. **Настройка запуска АЦП от таймера**

2.1. Добавьте функцию для настройки таймера (например, TIM2) для запуска АЦП:

```
void adc_start_timer_trigger(uint32_t timer_freq_hz) {
    // Включение тактирования TIM2 и настройка
    RCC->APB1ENR |= (1 << 0);  // TIM2EN

    // Расчёт PSC и ARR для частоты timer_freq_hz (примерно)
    uint32_t timer_clock = 84000000;  // APB1=84 MHz
    uint32_t prescaler = (timer_clock / timer_freq_hz) - 1;
    TIM2->PSC = prescaler;
    TIM2->ARR = 1;   // счётчик до 1 -> частота прерываний = timer_freq_hz
    TIM2->CNT = 0;

    // Подключение TIM2 как внешний триггер для ADC1
    // EXTSEL = 0010 (TIM2_TRGO)
    ADC1->CR2 &= ~(7 << 24);
    ADC1->CR2 |= (2 << 24);
    ADC1->CR2 |= (1 << 12); // EXTSEL = enable external trigger for regular group

    TIM2->CR2 |= (1 << 4);  // MMS = 0010 (Update event as TRGO)
    TIM2->CR1 |= (1 << 0);  // Enable TIM2

    // Включение АЦП в режиме запуска от триггера
    ADC1->CR2 |= (1 << 0);  // ADON
}
```

3. **Сравнение методов сбора данных**

3.1. Создайте файл main.c с демонстрацией всех режимов:

```
#include "adc_driver.h"
#include "timers.h"
#include "uart_driver.h"  // для вывода результатов (из ЛР опционально)

volatile uint32_t adc_samples_count = 0;

void adc_callback(uint16_t value) {
    adc_samples_count++;
    // Здесь можно накапливать статистику или отправлять через UART
}

void test_polling(void) {
    adc_init(ADC_MODE_SINGLE, 0, 84);  // 84 цикла выборки
    uint32_t start = SysTick_GetTick();
    for (int i = 0; i < 1000; i++) {
        uint16_t val = adc_read_single(0);
    }
    uint32_t elapsed = SysTick_GetTick() - start;
    // Вывод: 1000 преобразований заняло X мс
}

void test_interrupts(void) {
    adc_init(ADC_MODE_CONTINUOUS, 0, 84);
    adc_register_callback(adc_callback);
    adc_enable_averaging(16);
    adc_start_continuous();

    // Даём поработать 2 секунды
```

```
    SysTick_DelayMs(2000);
    adc_stop_continuous();
    // adc_samples_count содержит число преобразований
}

void test_timer_trigger(void) {
    adc_init(ADC_MODE_TIMER_TRIG, 0, 84);
    adc_register_callback(adc_callback);
    adc_start_timer_trigger(1000);  // 1 кГц частота дискретизации
    // Работает фоном
}
```

4. **Реализация фильтрации данных**

4.1. Добавьте медианный фильтр как альтернативу скользящему среднему:

```
#define MEDIAN_WINDOW 5
static uint16_t median_buffer[MEDIAN_WINDOW];
static uint8_t median_index = 0;

uint16_t median_filter(uint16_t new_sample) {
    median_buffer[median_index] = new_sample;
    median_index = (median_index + 1) % MEDIAN_WINDOW;

    // Копирование и сортировка
    uint16_t sorted[MEDIAN_WINDOW];
    for (int i = 0; i < MEDIAN_WINDOW; i++) sorted[i] = median_buffer[i];
    // Пузырьковая сортировка (для малого окна)
    for (int i = 0; i < MEDIAN_WINDOW-1; i++) {
        for (int j = i+1; j < MEDIAN_WINDOW; j++) {
            if (sorted[i] > sorted[j]) {
```

```
            uint16_t t = sorted[i]; sorted[i] = sorted[j]; sorted[j] = t;
        }
      }
    }
    return sorted[MEDIAN_WINDOW/2];
}
```

5. **Эксперимент по измерению шумов и эффективности фильтрации**

5.1. Подключите к входу АЦП источник постоянного напряжения (например, делитель с батарейки). Соберите 1000 отсчётов без фильтрации и с фильтрацией. Вычислите среднеквадратическое отклонение (СКО) для обоих случаев. Сделайте вывод об эффективности.

5.2. Используйте потенциометр для подачи переменного напряжения, измеряйте и выводите значения через UART. Нарисуйте график в Excel или Python.

Пример выполнения задания

Задача: Реализовать систему измерения температуры с помощью терморезистора (или LM35) через АЦП. Частота опроса 10 Гц, применяется скользящее среднее с окном 8. При изменении температуры более чем на 1 градус – отправлять уведомление через UART.

Решение (фрагмент):

```
#define TEMP_ADC_CHANNEL 0
#define AVERAGING_WINDOW 8

float adc_to_temperature(uint16_t adc_value) {
    // Предположим линейную зависимость: 0 В -> 0°C, 3.3 В -> 100°C
    float voltage = (adc_value * 3.3f) / 4095.0f;
    return voltage * 30.303f;  // 100 / 3.3
}
```

```
float filtered_temp = 0.0f;
float last_reported_temp = 0.0f;

void temperature_callback(uint16_t raw) {
    static uint16_t buffer[8];
    static uint8_t idx = 0;
    static uint32_t sum = 0;

    sum -= buffer[idx];
    buffer[idx] = raw;
    sum += raw;
    idx = (idx + 1) % 8;

    uint16_t avg_raw = sum / 8;
    float temp = adc_to_temperature(avg_raw);

    if (fabsf(temp - last_reported_temp) > 1.0f) {
        last_reported_temp = temp;
        printf("Temperature: %.1f C\n", temp);
    }
}

int main(void) {
    adc_init(ADC_MODE_TIMER_TRIG, TEMP_ADC_CHANNEL, 84);
    adc_register_callback(temperature_callback);
    adc_start_timer_trigger(10);  // 10 Гц
    while (1);
}
```

Результаты эксперимента:

- Без фильтрации размах шума на выходе АЦП составлял ±10 LSB (≈8 мВ, что соответствует ~0.25°C).
- Со скользящим средним (окно 8) размах шума уменьшился до ±2 LSB (±0.05°C).
- Частота обновления фильтрованного значения: 10 Гц (задержка 0.8 секунды).

Индивидуальные задания

Вариант 1. Измерение напряжения аккумулятора с усреднением

Подключите делитель напряжения (2:1) к батарее 3.7–4.2 В. Выполните 1000 измерений. Вычислите среднее, минимальное, максимальное. Реализуйте функцию battery_percent() преобразующую напряжение в проценты.

Вариант 2. Цифровой вольтметр (0-30 В) с индикацией на 7-сегментном индикаторе

Используйте внешний делитель 10:1 (макс 33 В). АЦП измеряет напряжение, отображает на трёхразрядном индикаторе. Реализуйте фильтр для устранения мерцания последнего разряда.

Вариант 3. Регистрация сигнала с датчика пульса (фотоплетизмография)

Частота дискретизации 100 Гц. Передавайте значения через UART в виде CSV. Нарисуйте график в Processing/Python. Определите частоту сердечных сокращений.

Вариант 4. Сравнение методов усреднения: скользящее среднее vs медианный фильтр

Подайте на вход АЦП сигнал с импульсными помехами (микроконтроллер генерирует на соседнем выводе короткие импульсы). Зафиксируйте, какой фильтр лучше справляется с выбросами, а какой – с гауссовским шумом.

Вариант 5. АЦП с автоматической калибровкой смещения

При запуске замыкайте вход на землю (или используйте внутренний канал Vref) для измерения смещения (offset). Корректируйте все последующие измерения, вычитая это смещение.

Вариант 6. Аудио-спектроанализатор на основе БПФ

Оцифруйте сигнал с микрофона (5000 выборок в секунду, 256 отсчётов). Вычислите БПФ (используйте библиотеку CMSIS-DSP). Определите доминирующую частоту и отобразите на светодиодной шкале.

Вариант 7. Реализация программируемого компаратора через АЦП

Подключите потенциометр к входу. Второй канал АЦП измеряет опорное напряжение. Если вход выше опорного – включается светодиод. Задержка срабатывания не более 1 мс.

Вариант 8. АЦП с управляемым усилением (программное)

Используйте два канала: прямой и через усилитель (операционный). Если сигнал мал (< 1/8 шкалы), используйте усиленный канал с компенсацией коэффициента. Расширьте динамический диапазон.

Вариант 9. Осциллограф на основе АЦП + UART (логирование)

Настройте АЦП на максимальную скорость (без прерываний, с DMA). Соберите 1024 отсчёта в буфер, затем отправьте через UART в терминал или сохраните в файл. Визуализируйте в программе (например, Putty + скрипт).

Вариант 10. Термостат с гистерезисом

Измеряйте температуру с помощью термистора. Поддерживайте температуру в заданном диапазоне с помощью реле или симистора (управление через GPIO). Используйте АЦП в непрерывном режиме с усреднением.

Вариант 11. Измерение ёмкости методом "RC-цепь + АЦП"

Заряжайте конденсатор через резистор, измеряйте время достижения 63.2% от Vcc (аналоговый компаратор или через АЦП с опросом). Вычислите ёмкость. Постройте калибровочную кривую.

Вариант 12. АЦП с дифференциальным входом (псевдодифференциальный)

Используйте два канала АЦП. Один измеряет сигнал, другой – землю поблизости от датчика. Вычитайте значения для подавления синфазной помехи.

Вариант 13. Генерация синуса с помощью АЦП (ЦАП) и DMA

Используйте ЦАП (или ШИМ+фильтр) для вывода синусоиды. АЦП измеряет её и сравнивает с таблицей. Оцените нелинейность и шум.

Вариант 14. Регистратор данных (логгер) с записью в EEPROM

Собирайте данные с АЦП каждую минуту (таймер RTC). Сохраняйте в энергонезависимой памяти (внутренняя Flash или EEPROM). При запросе по UART выгружайте журнал.

Вариант 15. АЦП в режиме сканирования (три канала)

Подключите три потенциометра к трём входам АЦП. Настройте сканирование и DMA. Выводите три значения на ЖК-дисплей или по UART. Сравните загрузку CPU с методом прерываний.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.

2. Цель и задачи лабораторной работы.

3. Теоретическую часть (принципы АЦП, режимы работы, триггеры, фильтрация).

4. Листинги разработанных модулей (adc_driver.h, adc_driver.c, main.c).

5. Таблицу и графики, демонстрирующие результаты измерений (например, зашумлённый сигнал до и после фильтрации).

6. Сравнение методов сбора данных (polling vs interrupt vs timer-trigger) по загрузке CPU, точности, максимальной частоте.

7. Результаты выполнения индивидуального задания с подробным описанием и кодом.

8. Выводы (какой метод и фильтр подходит для конкретных приложений).

9. Ответы на контрольные вопросы.

Контрольные вопросы

1. Какое разрешение имеет АЦП в STM32F4? Сколько уровней квантования?

2. Что такое время выборки (sampling time) и как оно влияет на точность?

3. В чём разница между однократным и непрерывным режимами АЦП?

4. Как настроить АЦП для запуска от таймера? Какие регистры за это отвечают?

5. Почему при работе с АЦП в прерываниях важно быстро читать регистр DR?

6. Какие преимущества даёт использование DMA с АЦП? Когда это необходимо?

7. Как рассчитать частоту дискретизации при использовании таймера в качестве триггера?

8. Что такое «антиалиасинговый фильтр» и зачем он нужен перед АЦП?

9. Чем отличается скользящее среднее от медианного фильтра? Какой лучше для импульсных помех?

10. Как оценить эффективность фильтра? Какие статистические метрики использовать?

11. Какой максимальной частоты дискретизации можно достичь на STM32F4 без пропусков?

12. Как влияет увеличение времени выборки на максимальную частоту дискретизации?

13. Что произойдёт, если частота прерываний АЦП станет выше, чем успевает обрабатывать процессор?

14. Как подключить несколько каналов АЦП? Какой регистр определяет последовательность?

15. Для чего используется регистр ADC_CCR (Common Control Register)?

16. Как выполнить калибровку АЦП программно?

17. Как перевести вывод в аналоговый режим (GPIO_MODER) и зачем это нужно?

18. Как выровнять результат АЦП (влево/вправо) и зачем?

19. Как с помощью АЦП измерить напряжение выше опорного (например, 12 В)?

20. Почему результаты АЦП могут «плавать» даже при постоянном входном напряжении? Как с этим бороться?

Лабораторная работа №9

Буферизация данных. Работа с кольцевым буфером

Цель работы: освоение принципов организации кольцевого буфера (FIFO) для асинхронного обмена данными между обработчиком прерывания и фоновой задачей, реализация потокобезопасного буфера для данных АЦП, анализ проблемы переполнения и способов её решения.

Задачи:

- Изучить структуру и алгоритмы работы кольцевого буфера (FIFO) на языке C.
- Реализовать потокобезопасный кольцевой буфер для хранения данных АЦП с поддержкой операций записи (из прерывания) и чтения (из основного цикла).
- Обеспечить защиту от гонок данных (при одновременном доступе из прерывания и main) без длительного запрета прерываний.
- Реализовать механизм обнаружения и обработки переполнения буфера (потеря данных, статистика ошибок).
- Интегрировать кольцевой буфер с драйвером АЦП из лабораторной работы №8.
- Сравнить производительность и надёжность системы с буферизацией и без неё при различных скоростях поступления данных.

Теоретическая часть

Проблема асинхронной передачи данных

В системах реального времени данные часто поступают асинхронно – от прерываний, DMA, таймеров. Основная программа (фоновый цикл) не может мгновенно обработать каждое значение, так как может быть занята другими задачами или ожидать событий. Без буферизации возможна потеря данных, если обработчик прерывания не успевает передать значение в основную программу до прихода следующего.

Кольцевой буфер (Ring Buffer / Circular Buffer / FIFO)

Кольцевой буфер – это структура данных фиксированного размера, в которой запись производится последовательно, а при достижении конца массива указатель возвращается в начало (замыкается в кольцо). Чтение также идёт последовательно.

Основные компоненты:

- buffer – массив элементов (например, uint16_t для данных АЦП)
- head – индекс (или указатель) для записи (куда писать следующий элемент)
- tail – индекс для чтения (откуда читать следующий элемент)
- count или size – текущее количество элементов в буфере
- capacity – максимальный размер буфера

Условия:

- Пустой буфер: head == tail (и count == 0)
- Полный буфер: head == tail (и count == capacity) – требуется различать состояния.

Способы различения пустого и полного буфера:

1. **Счётчик элементов** – просто и надёжно, требует атомарного обновления (или защиты).
2. **Всегда оставлять один свободный элемент** – head == tail означает пустоту, полный – когда (head+1) % capacity == tail. Экономит счётчик, но теряет один слот.
3. **Флаг «полный»** – дополнительная переменная.

В многопоточном окружении (основная программа + прерывание) необходимо обеспечить атомарность операций с общими переменными (head, tail, count).

Потокобезопасный FIFO без блокировок (lock-free) для одного писателя и одного читателя:

Если только один писатель (например, прерывание) и один читатель (основная программа), можно обойтись без запрета прерываний:

- Писатель изменяет только head.
- Читатель изменяет только tail.
- При чтении нужно сохранить локальную копию head для вычисления доступных элементов.
- Такая реализация гарантирует корректность даже при прерывании читателя писателем (при условии, что операции записи/чтения над индексами атомарны для архитектуры ARM).

Реализация базового FIFO:

```
typedef struct {
    uint16_t* buffer;
    volatile uint32_t head;   // индекс записи
    volatile uint32_t tail;   // индекс чтения
    uint32_t capacity;
    volatile uint32_t count;  // количество элементов (опционально)
} ring_buffer_t;

bool ring_buffer_write(ring_buffer_t* rb, uint16_t data) {
    uint32_t next_head = (rb->head + 1) % rb->capacity;
    if (next_head == rb->tail) {
        return false;  // буфер полон
    }
    rb->buffer[rb->head] = data;
    rb->head = next_head;
    return true;
}
```

```
bool ring_buffer_read(ring_buffer_t* rb, uint16_t* out) {
    if (rb->head == rb->tail) {
        return false;  // буфер пуст
    }
    *out = rb->buffer[rb->tail];
    rb->tail = (rb->tail + 1) % rb->capacity;
    return true;
}
```

Обработка переполнения:

Когда буфер заполняется, новые данные не могут быть записаны. Стратегии:

1. **Игнорировать новые данные** (потеря последних) – просто вернуть ошибку записи.
2. **Перезапись старых данных** (overwrite) – переместить tail вперёд, теряя самые старые данные.
3. **Приостановка источника данных** (например, отключить прерывание АЦП до освобождения места) – не всегда возможно.
4. **Увеличение размера буфера** – простое решение, но ограничено памятью.

Для диагностики переполнения полезно вести счётчик пропущенных элементов (overflow_count).

Применение с АЦП:

АЦП в непрерывном режиме или по таймеру генерирует данные с фиксированной частотой. Если основная программа обрабатывает данные медленнее, буфер заполнится. Зная скорость поступления и обработки, можно выбрать минимальный размер буфера, гарантирующий отсутствие

переполнения в худшем случае (например, при задержке обработки на время максимальной длительности другой задачи).

DMA vs FIFO:

DMA автоматически переносит данные в массив и может генерировать прерывание при заполнении половины буфера или всего буфера (двойная буферизация). FIFO – более гибкое решение, если данные обрабатываются не блоками, а по одному.

Методика выполнения работы

1. **Создание модуля кольцевого буфера**

   1.1. Создайте каталог lab9 и файл ring_buffer.h:

```
#ifndef RING_BUFFER_H
#define RING_BUFFER_H

#include <stdint.h>
#include <stdbool.h>

typedef struct {
    uint16_t* buffer;
    volatile uint32_t head;
    volatile uint32_t tail;
    uint32_t capacity;
    volatile uint32_t count;        // текущее количество элементов
    volatile uint32_t overflow_cnt;  // счётчик потерь данных
} ring_buffer_t;

// Инициализация буфера (выделяет память)
bool ring_buffer_init(ring_buffer_t* rb, uint32_t capacity);
```

// Освобождение памяти

void ring_buffer_free(ring_buffer_t* rb);

// Запись элемента (из прерывания). Возвращает true, если успешно.

bool ring_buffer_write(ring_buffer_t* rb, uint16_t data);

// Чтение элемента (из основной программы). Возвращает true, если есть данные.

bool ring_buffer_read(ring_buffer_t* rb, uint16_t* out);

// Просмотр элемента без удаления (peek)

bool ring_buffer_peek(const ring_buffer_t* rb, uint32_t index, uint16_t* out);

// Количество доступных для чтения элементов

uint32_t ring_buffer_available(const ring_buffer_t* rb);

// Очистка буфера

void ring_buffer_clear(ring_buffer_t* rb);

// Получение статистики переполнений

uint32_t ring_buffer_get_overflow(ring_buffer_t* rb);

#endif

1.2. Создайте файл ring_buffer.c:

#include "ring_buffer.h"

#include <stdlib.h>

#include <string.h>

```
bool ring_buffer_init(ring_buffer_t* rb, uint32_t capacity) {
    rb->buffer = (uint16_t*)malloc(capacity * sizeof(uint16_t));
    if (rb->buffer == NULL) return false;
    rb->capacity = capacity;
    rb->head = 0;
    rb->tail = 0;
    rb->count = 0;
    rb->overflow_cnt = 0;
    return true;
}

void ring_buffer_free(ring_buffer_t* rb) {
    if (rb->buffer) {
        free(rb->buffer);
        rb->buffer = NULL;
    }
}

bool ring_buffer_write(ring_buffer_t* rb, uint16_t data) {
    // Проверка на полный буфер
    if (rb->count >= rb->capacity) {
        // Переполнение: можно выбрать стратегию "отброс новых"
        rb->overflow_cnt++;
        return false;
    }
    rb->buffer[rb->head] = data;
    rb->head = (rb->head + 1) % rb->capacity;
    rb->count++;
    return true;
}
```

```
bool ring_buffer_read(ring_buffer_t* rb, uint16_t* out) {
    if (rb->count == 0) return false;
    *out = rb->buffer[rb->tail];
    rb->tail = (rb->tail + 1) % rb->capacity;
    rb->count--;
    return true;
}

bool ring_buffer_peek(const ring_buffer_t* rb, uint32_t index, uint16_t* out)
{
    if (index >= rb->count) return false;
    uint32_t pos = (rb->tail + index) % rb->capacity;
    *out = rb->buffer[pos];
    return true;
}

uint32_t ring_buffer_available(const ring_buffer_t* rb) {
    return rb->count;
}

void ring_buffer_clear(ring_buffer_t* rb) {
    rb->head = 0;
    rb->tail = 0;
    rb->count = 0;
    // overflow_cnt не сбрасываем, чтобы сохранить статистику
}

uint32_t ring_buffer_get_overflow(ring_buffer_t* rb) {
    return rb->overflow_cnt;
```

```
}
```

2. **Интеграция с драйвером АЦП (из ЛР8)**

2.1. Модифицируйте adc_driver.h, добавив поддержку кольцевого буфера:

```
#include "ring_buffer.h"

// Инициализация АЦП с буферизацией
void adc_init_with_buffer(uint32_t sample_cycles, uint32_t buffer_capacity);

// Получить указатель на буфер (для чтения из main)
ring_buffer_t* adc_get_buffer(void);

// Функция обратного вызова (для совместимости)
void adc_set_buffer_callback(void (*callback)(uint16_t));
```

2.2. Реализуйте в adc_driver.c (дополнения):

```
static ring_buffer_t adc_ring_buffer;
static bool use_buffer = false;

void adc_init_with_buffer(uint32_t sample_cycles, uint32_t buffer_capacity)
{
    adc_init(ADC_MODE_CONTINUOUS, 0, sample_cycles);
    if (ring_buffer_init(&adc_ring_buffer, buffer_capacity)) {
        use_buffer = true;
    }
    // Включение прерывания АЦП
    ADC1->CR1 |= (1 << 5);  // EOCIE
    NVIC_EnableIRQ(ADC_IRQn);
```

```
}

ring_buffer_t* adc_get_buffer(void) {
    return &adc_ring_buffer;
}

void ADC_IRQHandler(void) {
    if (ADC1->SR & (1 << 1)) {
        ADC1->SR &= ~(1 << 1);
        uint16_t value = ADC1->DR;

        if (use_buffer) {
            ring_buffer_write(&adc_ring_buffer, value);
        } else if (user_callback) {
            user_callback(value);
        }
    }
}
```

3. **Создание тестовой программы с буферизацией**

3.1. Файл main.c:

```
#include "adc_driver.h"
#include "ring_buffer.h"
#include "timers.h"
#include "uart_driver.h"  // или printf через semihosting

#define BUFFER_SIZE 256

void process_samples(ring_buffer_t* rb) {
    uint16_t sample;
```

```
    uint32_t processed = 0;
    // Обрабатываем все доступные данные (не блокируя)
    while (ring_buffer_read(rb, &sample)) {
        // Усреднение, фильтрация, отправка и т.д.
        static uint32_t sum = 0;
        static uint32_t count = 0;
        sum += sample;
        count++;
        if (count >= 100) {
            printf("Avg: %lu\n", sum / 100);
            sum = 0;
            count = 0;
        }
        processed++;
    }
    if (processed) {
        // Можно вывести статистику каждые N циклов
    }
}

int main(void) {
    SysTick_Init(1000);
    adc_init_with_buffer(84, BUFFER_SIZE);
    adc_start_continuous();  // запуск непрерывного режима

    uint32_t last_report = 0;

    while (1) {
        ring_buffer_t* rb = adc_get_buffer();
        process_samples(rb);
```

```
        uint32_t now = SysTick_GetTick();
        if ((now - last_report) >= 5000) {  // каждые 5 секунд
            uint32_t overflow = ring_buffer_get_overflow(rb);
            uint32_t pending = ring_buffer_available(rb);
            printf("Overflow: %lu, Pending: %lu\n", overflow, pending);
            last_report = now;
        }

        // Имитация длительной обработки (для создания переполнения)
        // SysTick_DelayMs(10);  // раскомментировать для эксперимента
    }
}
```

4. **Эксперимент с переполнением**

4.1. Установите маленький размер буфера (например, 16) и убедитесь, что при высокой частоте АЦП (непрерывный режим) возникает переполнение. Наблюдайте рост overflow_cnt.

4.2. Внесите искусственную задержку в process_samples (например, SysTick_DelayMs(5)) и пронаблюдайте переполнение. Затем увеличьте размер буфера до 512, чтобы переполнение исчезло.

4.3. Рассчитайте минимальный размер буфера: частота АЦП = 10 кГц, максимальное время, на которое основная программа может заблокировать обработку = 100 мс. За это время накопится 1000 выборок. Значит, буфер должен быть >1000.

5. **Реализация стратегии перезаписи старых данных (overwrite)**

5.1. Модифицируйте ring_buffer_write для поддержки режима перезаписи:

```
typedef enum {
    RB_DISCARD_NEW,   // потеря новых данных
    RB_OVERWRITE_OLD  // перезапись старых
} rb_policy_t;

static rb_policy_t policy = RB_DISCARD_NEW;

void ring_buffer_set_policy(rb_policy_t p) { policy = p; }

bool ring_buffer_write(ring_buffer_t* rb, uint16_t data) {
    if (rb->count >= rb->capacity) {
        rb->overflow_cnt++;
        if (policy == RB_OVERWRITE_OLD) {
            // Перезаписываем старые данные: перемещаем tail
            rb->tail = (rb->tail + 1) % rb->capacity;
            rb->count--;   // фактически, количество не меняется, но мы "выкидываем" один старый элемент
        } else {
            return false;
        }
    }
    rb->buffer[rb->head] = data;
    rb->head = (rb->head + 1) % rb->capacity;
    rb->count++;
    return true;
}
```

Обратите внимание: в OVERWRITE_OLD счётчик count остаётся равным capacity, но данные циклически заменяются. Это корректно только если читатель не отстаёт больше чем на capacity. Для простоты лучше

использовать перезапись без изменения count, а в ring_buffer_read проверять head == tail и count (но тогда нужно определять переполнение иначе). Предложите студентам обсудить.

6. **Измерение накладных расходов**

6.1. Сравните время выполнения ring_buffer_write и ring_buffer_read (через циклы DWT). Насколько они критичны для частоты дискретизации 10 кГц?

Пример выполнения задания

Задача: Реализовать сбор данных АЦП с частотой 1 кГц, буфер на 500 элементов, основная программа обрабатывает данные пакетами по 50 выборок (вычисляет среднее и отправляет по UART). При переполнении – включается светодиодная индикация ошибки.

Фрагмент кода:

```
#define ADC_FREQ 1000
#define BUFFER_SIZE 500
#define BATCH_SIZE 50

ring_buffer_t rb;
uint32_t batch_sum = 0;
uint32_t batch_cnt = 0;

void adc_callback(uint16_t value) {
   ring_buffer_write(&rb, value);
}

void process_batch(void) {
   uint16_t sample;
   while (ring_buffer_read(&rb, &sample)) {
```

```
            batch_sum += sample;
            batch_cnt++;
            if (batch_cnt >= BATCH_SIZE) {
                uint16_t avg = batch_sum / BATCH_SIZE;
                printf("Avg: %u\n", avg);
                batch_sum = 0;
                batch_cnt = 0;
            }
        }

        if (ring_buffer_get_overflow(&rb) > 0) {
            LED_On();  // сигнал переполнения
        } else {
            LED_Off();
        }
    }

    int main(void) {
        ring_buffer_init(&rb, BUFFER_SIZE);
        adc_init(ADC_MODE_TIMER_TRIG, 0, 84);
        adc_register_callback(adc_callback);
        adc_start_timer_trigger(ADC_FREQ);

        while (1) {
            process_batch();
            // другие задачи...
        }
    }
```

Результаты эксперимента:

- Без буфера: при задержке в main всего 5 мс терялось около 10% данных (пропуски).
- С буфером 500 элементов: при той же задержке потери отсутствуют (буфер заполнялся до 120 элементов, но до 500 не доходило).
- Переполнение наступало при задержке >500 мс, что ожидаемо, так как буфер 500 на частоте 1 кГц позволяет накопить данные за 0.5 секунды.

Индивидуальные задания

Вариант 1. FIFO общего назначения (типизированный)

Сделайте кольцевой буфер, который может хранить данные любого типа (используя void* и размер элемента). Реализуйте функции записи/чтения с байтовой адресацией.

Вариант 2. Два буфера: для сырых и отфильтрованных данных

АЦП пишет в буфер A. Фоновая задача читает из A, фильтрует и записывает в буфер B. Другая задача читает из B. Реализуйте потокобезопасную связь.

Вариант 3. Буфер с приоритетом (служебные данные)

При поступлении специального маркера (например, 0xFFFF) данные должны быть обработаны немедленно, даже если буфер заполнен. Реализуйте механизм «внеочередной вставки».

Вариант 4. Использование DMA + двойной буфер вместо FIFO

Настройте АЦП с DMA в режиме циклического буфера (circular mode) на 256 элементов. При заполнении половины буфера генерируется прерывание. Обработчик копирует данные во второй буфер и устанавливает флаг. Сравните с FIFO-реализацией.

Вариант 5. Буфер с плавающей границей (динамический размер)

Реализуйте кольцевой буфер, который может автоматически увеличивать свой размер (через realloc) при частых переполнениях. Ограничьте максимальный размер.

Вариант 6. Логирование телеметрии через буфер

Данные АЦП записываются в буфер. Основная программа периодически (раз в секунду) отправляет весь содержимое буфера по UART (сырой дамп). Реализуйте чтение без удаления (peek) и сброс после отправки.

Вариант 7. Буфер с временными метками

Вместе с каждым отсчётом АЦП сохраняйте 32-битную метку времени (значение SysTick). При чтении возвращайте пару (время, значение). Используйте структуру.

Вариант 8. Статистический монитор буфера

Добавьте функции для получения максимальной и средней заполненности буфера за последние N секунд. Периодически выводите эти статистики через UART.

Вариант 9. FIFO для межзадачного обмена (RTOS)

Если используется FreeRTOS, реализуйте кольцевой буфер с блокирующими операциями (запись ждёт, если буфер полон, чтение ждёт, если пуст). Используйте семафоры.

Вариант 10. Буфер с контрольной суммой

При записи вычисляйте простую контрольную сумму (XOR) для каждого блока из 16 элементов. При чтении проверяйте и сообщайте об ошибках.

Вариант 11. Энергонезависимый буфер (сохранение при сбое питания)

Храните буфер в резервной RAM (RTC backup registers) или SRAM с батарейным питанием. При включении восстанавливайте состояние.

Вариант 12. Удалённый доступ к буферу через UART (отладка)

Реализуйте команды для просмотра текущего содержимого буфера, сброса, получения статистики через простое текстовое меню по UART.

Вариант 13. Обработка переполнения звуковым сигналом

При переполнении буфера включайте зуммер на 100 мс и увеличивайте счётчик ошибок. После трёх переполнений подряд – останавливайте АЦП.

Вариант 14. Декодирование IR-сигнала с помощью буфера

Сигнал с ИК-приёмника (оцифрованный через АЦП или компаратор) записывается в буфер по прерываниям от таймера (захват). Основная программа анализирует буфер, распознаёт код NEC. Используйте кольцевой буфер для хранения длительностей импульсов.

Вариант 15. Буфер с поддержкой поиска паттерна

Реализуйте функцию, которая сканирует буфер на предмет заданной последовательности (например, синхропосылка 0xAA,0x55). При нахождении – генерирует событие и удаляет найденные данные вместе с префиксом.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (описание кольцевого буфера, проблемы асинхронной передачи данных, стратегии переполнения).
4. Листинги: ring_buffer.h, ring_buffer.c, модифицированного adc_driver.c и main.c.
5. Результаты экспериментов с разными размерами буфера и нагрузкой (таблица заполненности, частота переполнений).
6. Графики зависимости числа потерянных выборок от размера буфера и от задержки обработки.
7. Результаты индивидуального задания с кодом и пояснениями.
8. Выводы о применимости FIFO и выборе стратегии обработки переполнения.
9. Ответы на контрольные вопросы.

Контрольные вопросы

1. Почему при работе с прерываниями и основной программой необходима буферизация?

2. Что такое кольцевой буфер (FIFO)? Какие операции над ним определены?

3. Как отличить состояние «пусто» от «полно» в кольцевом буфере? Назовите два способа.

4. Почему в реализации для одного писателя и одного читателя можно обойтись без блокировок?

5. Как обеспечить атомарность обновления индексов head и tail в архитектуре ARM?

6. Какие стратегии обработки переполнения буфера вы знаете? Назовите достоинства и недостатки каждой.

7. Как выбрать размер буфера для системы с известной частотой поступления данных и максимальным временем блокировки обработчика?

8. Что произойдёт, если прерывание попытается записать данные в полный буфер, а стратегия выбрана DISCARD_NEW?

9. Какие методы отладки переполнения буфера можно использовать (аппаратные, программные)?

10. Как изменится производительность, если увеличить размер буфера в 10 раз?

11. Чем отличается кольцевой буфер от двойной буферизации (double buffering)?

12. Как реализовать буфер, который автоматически перезаписывает самые старые данные (overwrite)?

13. Какие типичные ошибки возникают при реализации FIFO (например, race condition, индексы вне диапазона)?

14. Можно ли использовать кольцевой буфер в прерывании более высокого приоритета, чем то, которое его читает? Если да, то какие меры предосторожности?

15. Как проверить, что реализация FIFO является потокобезопасной?

16. Как измерить время выполнения операций записи и чтения? Влияет ли это на реальное время?

17. Какую роль играет volatile в объявлении head, tail, count? Что произойдёт без него?

18. Как организовать буфер для данных переменной длины (например, сообщений UART разной длины)?

19. В каких случаях FIFO, реализованный программно, предпочтительнее аппаратного (DMA с двойным буфером)?

20. Как с помощью отладчика (JTAG/SWD) проследить состояние буфера в реальном времени?

Лабораторная работа №10

Протокол UART. Реализация командного интерпретатора

Цель работы: освоение протокола асинхронной последовательной связи UART, разработка драйвера приёма/передачи данных с использованием прерываний, реализация простого командного интерпретатора (CLI) для управления светодиодом, считывания АЦП и вывода системной информации через терминал.

Задачи:

- Изучить принципы работы асинхронного последовательного интерфейса UART: старт-стопные биты, скорость передачи, формат данных.
- Настроить периферийный модуль UART микроконтроллера (на примере STM32F4) на заданную скорость (9600 или 115200 бод).
- Реализовать неблокирующую передачу строк через UART с использованием прерывания (TXE/TC).
- Реализовать приём данных по прерыванию (RXNE) с накоплением строки до символа новой строки ('\n' или '\r').
- Разработать простой командный интерпретатор: разбор строки, выделение команды и аргументов, выполнение соответствующих действий.
- Интегрировать CLI с ранее разработанными модулями (GPIO, LED-драйвер, АЦП, кольцевой буфер).
- Освоить отладку через UART с использованием терминальной программы (PuTTY, minicom, screen).

Теоретическая часть

UART (Universal Asynchronous Receiver/Transmitter)

UART – один из самых распространённых последовательных интерфейсов для связи между микроконтроллером и компьютером, модулями Bluetooth, GPS, GSM.

Основные характеристики:

- **Асинхронный** – нет общего тактового сигнала, синхронизация идёт по стартовому биту.
- **Дуплексный** – передача и приём могут идти одновременно (отдельные линии TX и RX).
- **Формат кадра**: стартовый бит (0), 5–9 бит данных, необязательный бит чётности, 1–2 стоповых бита (1).
- **Скорость (baud rate)** – бит/с (обычно 9600, 19200, 38400, 115200). Оба устройства должны работать на одинаковой скорости.

Регистры UART (на примере USART2 в STM32F4):

Регистр	Назначение
SR (Status)	Флаги: TXE (регистр данных пуст), TC (передача завершена), RXNE (приняты данные)
DR (Data)	Передаваемый/принимаемый байт
BRR (Baud Rate)	Делитель для формирования скорости
CR1 (Control1)	Включение UART, разрешение прерываний (TXEIE, RXNEIE), бит чётности
CR2	Стоп-биты
CR3	Управление DMA и др.

Расчёт делителя скорости (BRR):

Для STM32F4 (тактовая частота USART2 = 84 МГц, если APB1 = 84 МГц):

BRR = 84 000 000 / baudrate

Пример: для 115200 → 84000000 / 115200 = 729,166 → BRR = 729 (целая часть).

Буферизация UART

При приёме данных на скорости 115200 бод (~11,5 Кбайт/с) байт поступает примерно каждые 87 мкс. Обработчик прерывания должен быстро прочитать байт из DR и сохранить в кольцевой буфер. Иначе следующие байты будут потеряны (флаг overrun).

Организация командного интерпретатора (CLI)

Типичный алгоритм:

1. В прерывании UART принимаются байты, помещаются в кольцевой буфер.
2. Фоновая задача читает байты из буфера, накапливает строку до символа \r или \n.
3. Полученная строка разбивается на токены (команда и аргументы).
4. По команде вызывается соответствующая функция.
5. Формируется ответ, который отправляется обратно через UART.

Формат команд (пример):

led on

led off

adc

help

Команды могут быть регистронезависимыми.

Передача строк:

Для передачи удобно использовать функцию uart_puts(char* str), которая помещает строку в буфер передачи, а прерывание TXE отправляет байт за байтом. Это не блокирует основную программу.

Кольцевые буферы для UART:

- rx_buffer – для принятых байтов (пишет обработчик RXNE, читает CLI).
- tx_buffer – для байтов на отправку (пишет uart_puts, читает обработчик TXE).

Потокобезопасность:

- Для RX: писатель – прерывание, читатель – main. Достаточно volatile-индексов, как в ЛР9.
- Для TX: писатель – main, читатель – прерывание. Аналогично.

Методика выполнения работы

1. **Подготовка модуля UART с кольцевыми буферами**

   1.1. Создайте каталог lab10 и файл uart_driver.h:

   ```
   #ifndef UART_DRIVER_H
   #define UART_DRIVER_H

   #include <stdint.h>
   #include <stdbool.h>

   // Инициализация UART2 (PA2 - TX, PA3 - RX) с заданной скоростью
   void uart_init(uint32_t baudrate);

   // Отправить строку (неблокирующая)
   void uart_puts(const char* str);

   // Отправить один символ
   void uart_putc(char c);

   // Отправить форматированную строку (обёртка для printf, если есть)
   void uart_printf(const char* fmt, ...);
   ```

```
// Проверка, есть ли принятые данные
bool uart_data_available(void);

// Чтение одного байта из приёмного буфера (неблокирующая)
bool uart_getc(char* c);

// Чтение строки (блокирующая, с таймаутом) - для CLI
bool uart_read_line(char* buffer, uint32_t max_len, uint32_t timeout_ms);

// Сброс буферов
void uart_flush_rx(void);
void uart_flush_tx(void);

#endif
```

1.2. Создайте файл uart_driver.c:

```
#include "uart_driver.h"
#include "ring_buffer.h"  // из ЛР9
#include <stdarg.h>
#include <stdio.h>

static ring_buffer_t rx_ring;
static ring_buffer_t tx_ring;
static uint8_t rx_buffer_data[256];
static uint8_t tx_buffer_data[256];
static bool tx_busy = false;

// USART2 base address (для STM32F4)
#define USART2_BASE 0x40004400
```

```
typedef struct {
    volatile uint32_t SR;
    volatile uint32_t DR;
    volatile uint32_t BRR;
    volatile uint32_t CR1;
    volatile uint32_t CR2;
    volatile uint32_t CR3;
    volatile uint32_t GTPR;
} USART_TypeDef;
#define USART2 ((USART_TypeDef*)USART2_BASE)

void uart_init(uint32_t baudrate) {
    // Включение тактирования USART2 и GPIOA
    RCC->APB1ENR |= (1 << 17);  // USART2EN
    RCC->AHB1ENR |= (1 << 0);   // GPIOA

    // Настройка PA2 (TX) как AF7, PA3 (RX) как AF7
    GPIOA->MODER &= ~(3 << (2*2));
    GPIOA->MODER |= (2 << (2*2));
    GPIOA->AFR[0] |= (7 << (2*4));

    GPIOA->MODER &= ~(3 << (3*2));
    GPIOA->MODER |= (2 << (3*2));
    GPIOA->AFR[0] |= (7 << (3*4));

    // Расчёт BRR
    uint32_t clock = 84000000;  // APB1=84 MHz
    uint32_t brr_value = clock / baudrate;
    USART2->BRR = brr_value;
```

```
    // Конфигурация: 8 бит, 1 стоп-бит, без чётности
    USART2->CR1 = (1 << 13);   // UE - UART enable
    USART2->CR2 = 0;           // 1 stop bit
    USART2->CR1 |= (1 << 2);   // RE (Receiver enable)
    USART2->CR1 |= (1 << 3);   // TE (Transmitter enable)

    // Инициализация кольцевых буферов
    ring_buffer_init(&rx_ring, 256, (uint8_t*)rx_buffer_data);
    ring_buffer_init(&tx_ring, 256, (uint8_t*)tx_buffer_data);

    // Разрешение прерываний: RXNE, TC (для TX)
    USART2->CR1 |= (1 << 5);   // RXNEIE
    USART2->CR1 |= (1 << 6);   // TCIE
    NVIC_EnableIRQ(USART2_IRQn);
    NVIC_SetPriority(USART2_IRQn, 1);
}

void uart_putc(char c) {
    // Запись в кольцевой буфер передачи
    while (!ring_buffer_write(&tx_ring, (uint8_t)c)) {
        // если буфер полон, ждём (можно использовать семафор, здесь упрощённо)
    }
    if (!tx_busy) {
        tx_busy = true;
        USART2->CR1 |= (1 << 7);  // TXEIE - разрешить прерывание на пустоту регистра
    }
}
```

```
void uart_puts(const char* str) {
    while (*str) {
        uart_putc(*str++);
    }
}

void uart_printf(const char* fmt, ...) {
    char buffer[128];
    va_list args;
    va_start(args, fmt);
    vsnprintf(buffer, sizeof(buffer), fmt, args);
    va_end(args);
    uart_puts(buffer);
}

bool uart_data_available(void) {
    return ring_buffer_available(&rx_ring) > 0;
}

bool uart_getc(char* c) {
    uint8_t byte;
    if (ring_buffer_read(&rx_ring, &byte)) {
        *c = (char)byte;
        return true;
    }
    return false;
}

bool uart_read_line(char* buffer, uint32_t max_len, uint32_t timeout_ms) {
    uint32_t start = SysTick_GetTick();
```

```
    uint32_t idx = 0;
    while ((SysTick_GetTick() - start) < timeout_ms) {
        char c;
        if (uart_getc(&c)) {
            if (c == '\r' || c == '\n') {
                buffer[idx] = '\0';
                return true;
            }
            if (idx < max_len - 1) {
                buffer[idx++] = c;
                // эхо-печать (опционально)
                uart_putc(c);
            }
        }
    }
    return false;
}

void USART2_IRQHandler(void) {
    // Прерывание по приёму
    if (USART2->SR & (1 << 5)) {  // RXNE
        uint8_t data = USART2->DR;
        ring_buffer_write(&rx_ring, data);
    }
    // Прерывание по окончанию передачи (TC)
    if (USART2->SR & (1 << 6)) {  // TC
        // Если в буфере передачи есть данные, отправляем
        uint8_t byte;
        if (ring_buffer_read(&tx_ring, &byte)) {
            USART2->DR = byte;
```

```
        } else {
            // Буфер пуст, отключаем TXEIE
            USART2->CR1 &= ~(1 << 7);
            tx_busy = false;
        }
    }
}
```

2. **Подготовка командного интерпретатора**

2.1. Создайте файл cli.h:

c

```
#ifndef CLI_H
#define CLI_H

void cli_init(void);
void cli_process(void);   // вызывается в основном цикле

#endif
```

2.2. Создайте файл cli.c:

```
#include "cli.h"
#include "uart_driver.h"
#include "led_driver.h"    // из ЛР6
#include "adc_driver.h"    // из ЛР8
#include <string.h>
#include <stdlib.h>

#define CMD_MAX_ARGS 4
#define CMD_BUFFER_SIZE 64
```

```
static char cmd_buffer[CMD_BUFFER_SIZE];

typedef struct {
    const char* name;
    void (*handler)(int argc, char** argv);
    const char* help;
} cmd_entry_t;

// Обработчики команд
static void cmd_help(int argc, char** argv);
static void cmd_led(int argc, char** argv);
static void cmd_adc(int argc, char** argv);
static void cmd_info(int argc, char** argv);
static void cmd_echo(int argc, char** argv);

static const cmd_entry_t commands[] = {
    {"help", cmd_help, "help - show this help"},
    {"led", cmd_led, "led {on|off|blink|fade} - control LED"},
    {"adc", cmd_adc, "adc [n] - read ADC (n samples, default 1)"},
    {"info", cmd_info, "info - system info"},
    {"echo", cmd_echo, "echo <> - echo back"},
    {NULL, NULL, NULL}
};

static void cmd_help(int argc, char** argv) {
    uart_puts("Available commands:\r\n");
    for (int i = 0; commands[i].name != NULL; i++) {
        uart_printf("  %s\r\n", commands[i].help);
    }
}
```

```
static void cmd_led(int argc, char** argv) {
    if (argc < 2) {
        uart_puts("Usage: led {on|off|blink|fade}\r\n");
        return;
    }
    if (strcmp(argv[1], "on") == 0) {
        led_driver_set_mode(LED_MODE_ON);
        uart_puts("LED turned ON\r\n");
    } else if (strcmp(argv[1], "off") == 0) {
        led_driver_set_mode(LED_MODE_OFF);
        uart_puts("LED turned OFF\r\n");
    } else if (strcmp(argv[1], "blink") == 0) {
        led_driver_set_mode(LED_MODE_BLINK);
        uart_puts("LED blinking\r\n");
    } else if (strcmp(argv[1], "fade") == 0) {
        led_driver_set_mode(LED_MODE_FADE);
        uart_puts("LED fading\r\n");
    } else {
        uart_puts("Unknown LED mode\r\n");
    }
}

static void cmd_adc(int argc, char** argv) {
    uint32_t samples = 1;
    if (argc >= 2) {
        samples = atoi(argv[1]);
        if (samples == 0) samples = 1;
        if (samples > 1000) samples = 1000;
    }
```

```
    uint32_t sum = 0;
    for (uint32_t i = 0; i < samples; i++) {
        sum += adc_read_single(0);
    }
    uint16_t avg = (uint16_t)(sum / samples);
    uart_printf("ADC value (avg of %lu): %u\r\n", samples, avg);
}

static void cmd_info(int argc, char** argv) {
    uart_puts("STM32F407 CLI Example\r\n");
    uart_printf("SystemCoreClock: %lu Hz\r\n", SystemCoreClock);
    // Дополнительная информация (свободная память, uptime и т.д.)
}

static void cmd_echo(int argc, char** argv) {
    if (argc >= 2) {
        uart_puts(argv[1]);
        for (int i = 2; i < argc; i++) {
            uart_putc(' ');
            uart_puts(argv[i]);
        }
        uart_puts("\r\n");
    } else {
        uart_puts("\r\n");
    }
}

// Разбор строки на команду и аргументы
static void parse_and_execute(char* line) {
    // Удаляем завершающий \r\n
```

```
    char* p = line + strlen(line) - 1;
    while (*p == '\r' || *p == '\n') *p-- = '\0';

    // Разделение на токены
    char* argv[CMD_MAX_ARGS];
    int argc = 0;
    char* token = strtok(line, " ");
    while (token != NULL && argc < CMD_MAX_ARGS) {
        argv[argc++] = token;
        token = strtok(NULL, " ");
    }
    if (argc == 0) return;

    // Поиск команды
    for (int i = 0; commands[i].name != NULL; i++) {
        if (strcmp(argv[0], commands[i].name) == 0) {
            commands[i].handler(argc, argv);
            return;
        }
    }
    uart_printf("Unknown command: %s\r\n", argv[0]);
}

void cli_init(void) {
    uart_puts("\r\nCLI Initialized. Type 'help' for commands.\r\n> ");
}

void cli_process(void) {
    // Накопление строки до \n
    static uint32_t buf_idx = 0;
```

```
    char c;
    while (uart_getc(&c)) {
        if (c == '\r' || c == '\n') {
            if (buf_idx > 0) {
                cmd_buffer[buf_idx] = '\0';
                parse_and_execute(cmd_buffer);
                uart_puts("> ");
                buf_idx = 0;
            } else {
                uart_puts("> ");
            }
        } else if (buf_idx < CMD_BUFFER_SIZE - 1) {
            // Эхо-вывод (опционально)
            uart_putc(c);
            cmd_buffer[buf_idx++] = c;
        }
    }
}
```

3. **Создание основной программы**

3.1. Создайте файл main.c:

```
#include "stm32f4xx.h"
#include "uart_driver.h"
#include "cli.h"
#include "led_driver.h"
#include "adc_driver.h"
#include "timers.h"

int main(void) {
    // Инициализация системного таймера (для задержек и тиков)
```

```
    SysTick_Init(1000);

    // Инициализация UART (115200 бод)
    uart_init(115200);

    // Инициализация светодиода
    led_driver_init();

    // Инициализация АЦП (один канал)
    adc_init(ADC_MODE_SINGLE, 0, 84);

    // Инициализация CLI
    cli_init();

    while (1) {
        cli_process();  // обработка введённых команд
        // Можно добавить другие фоновые задачи
    }
}
```

4. **Отладка и тестирование**

4.1. Скомпилируйте проект и загрузите в микроконтроллер.

4.2. Подключите USB-UART преобразователь к пинам PA2 (TX) и PA3 (RX), землю GND.

4.3. Откройте терминальную программу (PuTTY / minicom / screen) со скоростью 115200 бод, 8 бит, 1 стоп-бит, без чётности, без управления потоком.

4.4. Введите команды: help, led on, adc, led blink, echo Hello World.

5. **Расширение CLI (добавление новой команды)**

5.1. Добавьте команду uptime, которая показывает количество секунд с момента старта. Для этого в прерывании SysTick ведите глобальную переменную system_uptime_sec.

5.2. Добавьте обработчик cmd_uptime и зарегистрируйте его в таблице команд.

5.3. Продемонстрируйте работу.

Пример выполнения задания

Задача: Реализовать терминальное управление светодиодом с возможностью задавать частоту мигания командой blink <period_ms> и плавное изменение яркости с заданной скоростью fade <cycle_time_ms>.

Фрагмент расширения:

```
static void cmd_blink(int argc, char** argv) {
    if (argc >= 2) {
        uint32_t period = atoi(argv[1]);
        if (period < 20) period = 20;
        led_driver_set_blink_period(period);
        led_driver_set_mode(LED_MODE_BLINK);
        uart_printf("Blinking with period %lu ms\r\n", period);
    } else {
        uart_puts("Usage: blink <period_ms>\r\n");
    }
}

static void cmd_fade(int argc, char** argv) {
    if (argc >= 2) {
        uint32_t cycle = atoi(argv[1]);
        if (cycle < 100) cycle = 100;
        led_driver_set_fade_speed(cycle);
        led_driver_set_mode(LED_MODE_FADE);
```

```
        uart_printf("Fading cycle %lu ms\r\n", cycle);
    } else {
        uart_puts("Usage: fade <cycle_ms>\r\n");
    }
}
```

Добавление в таблицу команд:

```
c
{"blink", cmd_blink, "blink <ms> - set blink period"},
{"fade", cmd_fade, "fade <ms> - set fade cycle time"},
```

Результат работы в терминале:

```
> help
Available commands:
  help - show this help
  led on|off|blink|fade - control LED
  blink <ms> - set blink period
  fade <ms> - set fade cycle time
  adc [n] - read ADC
  info - system info
  uptime - show system uptime
  echo <> - echo back

> blink 500
Blinking with period 500 ms

> fade 2000
Fading cycle 2000 ms

> adc 100
ADC value (avg of 100): 2048
```

Индивидуальные задания

Вариант 1. Управление ШИМ-яркостью через CLI

Добавьте команду pwm <0..100>, которая задаёт скважность аппаратного ШИМ (из ЛР4) в процентах. Выводите текущую скважность.

Вариант 2. Команда для сброса микроконтроллера

Добавьте команду reset. При её вызове сбрасывайте МК программно (NVIC_SystemReset()).

Вариант 3. Команда для чтения/записи памяти (отладка)

Реализуйте команду peek <addr> для чтения 32-битного значения по адресу и poke <addr> <value> для записи. Выводите результат в HEX.

Вариант 4. История команд (history)

Храните последние 10 введённых команд в кольцевом буфере. Команда history показывает их. Команда !n повторяет n-ю команду из истории.

Вариант 5. Поддержка чисел в HEX и двоичном виде

Расширьте парсер чисел: если строка начинается с 0x – интерпретировать как шестнадцатеричную, если с 0b – как двоичную. Примените для команды poke.

Вариант 6. Команда для чтения состояния кнопки

Добавьте команду button, которая читает кнопку (GPIO) и выводит "Pressed" или "Released". Используйте обработку дребезга.

Вариант 7. Команда для настройки параметров АЦП

Реализуйте команду adc_config <sample_cycles>, где sample_cycles принимает значения 3, 15, 28, 56, 84, 112, 144, 480 циклов. Меняйте SMPR1/SMPR2.

Вариант 8. Команда для вывода графика сигнала (ASCII-графика)

Наберите adc_graph <n> – снимите n отсчётов АЦП (например, 64) и выведите их в виде гистограммы из звёздочек (каждый уровень – 1 символ). Удобно для визуализации.

Вариант 9. Команда для управления через UART с эхо-отключением

Добавьте команду echo off и echo on для включения/выключения эхо-печати вводимых символов. По умолчанию эхо включено.

Вариант 10. Команда для измерения производительности

Команда bench <n> выполняет n итераций пустого цикла, измеряет время (SysTick) и выводит среднее время итерации в микросекундах.

Вариант 11. Команда для дампа регистров UART

Выведите текущие значения регистров USART2 (SR, CR1, BRR и т.д.) в читаемом виде.

Вариант 12. Команда для работы с EEPROM (внешней)

Если подключена внешняя EEPROM по I2C, реализуйте команды eeprom_read <addr> и eeprom_write <addr> <byte>.

Вариант 13. Автодополнение команд (Tab completion)

Реализуйте обработку символа табуляции: поиск команд, начинающихся с текущего префикса, и вывод возможных вариантов.

Вариант 14. Команда для мониторинга температуры процессора Используйте внутренний датчик температуры АЦП (канал 16 или 18). Команда temp выводит температуру в градусах Цельсия (требуется калибровка).

Вариант 15. Команда для отправки данных по UART в бинарном виде (Base64)

Для передачи результатов измерений в двоичном виде, реализуйте кодирование Base64 и вывод через adc_b64 <n>.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.

2. Цель и задачи лабораторной работы.

3. Теоретическую часть (протокол UART, принципы асинхронной связи, структура кадра, расчёт скорости).

4. Листинги: uart_driver.h, uart_driver.c, cli.h, cli.c, main.c.

5. Подробное описание работы CLI (как разбираются команды, как добавляется новая команда).

6. Результаты тестирования (скриншоты или текстовые логи терминала с вводом различных команд).

7. Результаты выполнения индивидуального задания.

8. Выводы (достоинства и недостатки CLI, возможные улучшения).

9. Ответы на контрольные вопросы.

Контрольные вопросы

1. Что означает аббревиатура UART? В чём отличие UART от USART?

2. Что такое скорость в бодах (baud rate)? Как связана с битовой скоростью (bps) при 8N1?

3. Из каких частей состоит кадр UART? Сколько длится стартовый бит?

4. Как рассчитать значение регистра BRR для заданной скорости?

5. Что происходит при переполнении приёмного буфера UART (флаг ORE)?

6. Почему в обработчике прерывания UART важно быстро читать регистр DR?

7. Какие флаги используются для организации неблокирующей передачи (TXE, TC)? В чём разница?

8. Как организовать кольцевой буфер для приёма и почему он необходим?

9. Какие типичные проблемы могут возникнуть при использовании UART на высокой скорости (115200 и выше)?

10. Как в CLI происходит разделение строки на команду и аргументы?

11. Как обработать ситуацию, когда пользователь вводит команду длиннее буфера?

12. Почему команды в примере имеют обработчик cmd_help? Зачем нужна команда help?

13. Как добавить новую команду в CLI без изменения основного кода?

14. Как сделать команды регистронезависимыми?

15. Как передать число (например, период мигания) из строки в числовое значение?

16. Какие меры предосторожности нужны при использовании printf в прерывании?

17. Как измерить загрузку процессора обработкой UART-прерываний?

18. Как организовать потоковый ввод (приём большого объёма данных без потерь)?

19. Как реализовать аппаратное управление потоком (RTS/CTS) в UART?

20. Какие альтернативы UART существуют для отладки и управления (SWD, JTAG, USB CDC)? В чём их преимущества?

Лабораторная работа №11

Протокол I2C. Работа с EEPROM

Цель работы: освоение протокола I2C (Inter-Integrated Circuit) на физическом и программном уровне, реализация драйвера для работы с внешней энергонезависимой памятью EEPROM (24LCxx), приобретение навыков обработки ошибок шины и управления несколькими устройствами.

Задачи:

- Изучить архитектуру и временные диаграммы протокола I2C: стартовые и стоповые условия, передача байта, бит подтверждения (ACK/NACK).
- Настроить аппаратный модуль I2C микроконтроллера (или реализовать программный bit-banging на GPIO).
- Реализовать базовые функции: генерация START/STOP, отправка байта, чтение байта с подтверждением.
- Разработать драйвер для EEPROM стандарта 24LCxx (I2C-адрес = 0xA0/A1 для записи/чтения, внутренний адрес 8 или 16 бит).
- Реализовать функции записи байта, записи страницы, чтения текущего адреса, случайного чтения, последовательного чтения.
- Обработать типичные ошибки: отсутствие ответа (ACK failure), тайм-аут ожидания EEPROM (время записи до 5 мс), потеря синхронизации.
- Интегрировать EEPROM в систему командного интерпретатора (из ЛР10) для чтения/записи по командам.

Теоретическая часть

Протокол I2C (Inter-Integrated Circuit)

Разработан компанией Philips (ныне NXP) для связи между интегральными схемами. Использует две линии:

- **SCL (Serial Clock)** – тактовый сигнал (генерируется ведущим – master).
- **SDA (Serial Data)** – линия данных (двунаправленная, с открытым стоком).

Характеристики:

- Скорость: Standard (100 кГц), Fast (400 кГц), Fast Plus (1 МГц), High Speed (3.4 МГц).
- Топология: шинная, несколько ведущих и ведомых (адреса 7 или 10 бит).
- Подтяжка: оба вывода должны быть подтянуты к питанию через резисторы (обычно 2.2–10 кОм).

Электрические особенности:

- Логические уровни: низкий уровень формируется транзистором «открытый сток», высокий – подтягивающим резистором. Это позволяет любому устройству «вытянуть» шину в ноль.
- Арбитраж: если два ведущих пытаются передать одновременно, тот, кто первым выдаст 0 вместо 1, «проигрывает» и переходит в ведомый режим.

Формат передачи:

1. **Стартовое условие (START)**: SDA переключается из 1 в 0, пока SCL = 1.
2. **Передача байта**: 8 бит (старший бит первый), каждый бит выставляется на SDA при SCL=0 и фиксируется ведущим/ведомым по переднему фронту SCL.
3. **Бит подтверждения (ACK)**: после 8 бит ведущий отпускает SDA, ведомый притягивает SDA к 0 на 9-м такте SCL, если он готов принять следующий байт. Если ведомый не отвечает (SDA остаётся 1) – это NACK.
4. **Стоповое условие (STOP)**: SDA переключается из 0 в 1, пока SCL=1.

Адресация EEPROM (семейство 24LCxx):

Для 24LC02 (256 байт, 2 кбит) адрес устройства 7-битный: биты A2 A1 A0 задаются выводами чипа. Обычно все на земле → адрес 0b1010000 = 0x50.

На шине I2C адрес передаётся в первом байте после START: старшие 7 бит – адрес, младший бит (R/W) = 0 для записи, 1 для чтения.

Таким образом, байт адреса для записи: 0xA0 (0x50 << 1), для чтения: 0xA1.

Команды EEPROM 24LCxx:

- **Запись байта**: START → адрес устройства (запись) → ACK → внутренний адрес (8 бит для 24LC02, 16 бит для 24LC256) → ACK → данные → ACK → STOP.
  После STOP EEPROM переходит в цикл внутренней записи (до 5 мс) и не отвечает по шине.
- **Запись страницы**: до 16 байт (для 24LC02) или до 64 байт (для 24LC256) последовательно. Все байты должны быть в пределах одной страницы.
- **Чтение текущего адреса**: START → адрес устройства (чтение) → данные (байт по текущему внутреннему указателю) → ведущий генерирует NACK и STOP.
- **Случайное чтение**: START → адрес (запись) → внутренний адрес → START → адрес (чтение) → чтение байта → NACK → STOP.
- **Последовательное чтение**: после любого чтения можно продолжить выдавать такты SCL, и EEPROM будет выдавать следующие байты, инкрементируя счётчик.

Программный I2C (bit-banging) vs аппаратный:

Подход	Преимущества	Недостатки
Аппаратный I2C	Высокая скорость, малая нагрузка CPU, поддержка DMA	Фиксированные выводы, настройка регистров, может быть сложнее в отладке

Подход	Преимущества	Недостатки
Программный	Работает на любых выводах, простой код, удобен для отладки	Большая нагрузка CPU (занятие шины), меньшая скорость, чувствителен к прерываниям

В данной работе рекомендуется сначала реализовать программный I2C для понимания протокола, затем – аппаратный (по выбору).

Обработка ошибок:

1. **ACK failure** – устройство не ответило (отключено, неправильный адрес, занято). Нужно сгенерировать STOP и вернуть ошибку.
2. **Тайм-аут ожидания EEPROM** – после записи нужно повторять попытки чтения статуса (или слать START и проверять ACK) до тех пор, пока устройство не ответит.
3. **Потеря синхронизации** – если ведомое устройство «зависло» с SDA=0, нужно отправить до 9 тактов SCL, чтобы оно отпустило линию.

Методика выполнения работы

1. **Реализация программного I2C (bit-banging) на GPIO**

   1.1. Создайте каталог lab11 и файл i2c_soft.h:

   ```
   #ifndef I2C_SOFT_H
   #define I2C_SOFT_H

   #include <stdint.h>
   #include <stdbool.h>

   // Инициализация выводов I2C (SCL, SDA)
   void i2c_soft_init(uint8_t scl_pin, uint8_t sda_pin);
   ```

```
// Базовые функции протокола
void i2c_soft_start(void);
void i2c_soft_stop(void);
bool i2c_soft_write_byte(uint8_t data);
uint8_t i2c_soft_read_byte(bool send_ack);

// Комбинированные операции
bool i2c_soft_write(uint8_t dev_addr, uint8_t reg_addr, uint8_t data);
bool i2c_soft_read(uint8_t dev_addr, uint8_t reg_addr, uint8_t* data);
bool i2c_soft_write_page(uint8_t dev_addr, uint16_t mem_addr, uint8_t* data, uint16_t len);
bool i2c_soft_read_sequential(uint8_t dev_addr, uint16_t mem_addr, uint8_t* buffer, uint16_t len);

#endif
```

1.2. Создайте файл i2c_soft.c (основные функции):

```
#include "i2c_soft.h"
#include "gpio_driver.h"
#include "timers.h"   // для задержек

static uint8_t scl_pin, sda_pin;
static GPIO_TypeDef* scl_port;
static GPIO_TypeDef* sda_port;

#define I2C_DELAY_US 5  // для 100 кГц (5 мкс полупериод)

static void i2c_delay(void) {
    for (volatile int i = 0; i < 8; i++) { __asm("nop"); }  // грубо
```

```
    // В реальности использовать точные задержки на основе DWT или SysTick
    }

    static void set_sda_out(void) { /* переключить в режим выхода push-pull */ }
    static void set_sda_in(void)  { /* переключить в режим входа с подтяжкой */ }
    static void set_scl_out(void) { /* выход */ }

    static void sda_high(void) { GPIO_SetPin(sda_port, sda_pin); set_sda_in(); }
    static void sda_low(void)  { set_sda_out(); GPIO_ResetPin(sda_port, sda_pin); }
    static void scl_high(void) { GPIO_SetPin(scl_port, scl_pin); }
    static void scl_low(void)  { GPIO_ResetPin(scl_port, scl_pin); }

    static bool read_sda(void) { return GPIO_ReadPin(sda_port, sda_pin); }

    void i2c_soft_init(uint8_t scl_pin_num, uint8_t sda_pin_num) {
      scl_pin = scl_pin_num;
      sda_pin = sda_pin_num;
      // get port by pin - упрощённо: для STM32 по номеру пина определяем порт
      // Допустим, используем GPIOB для SCL и GPIOB для SDA, или отдельно.
      // Здесь примем, что оба пина на одном порту B.
      scl_port = GPIOB;
      sda_port = GPIOB;

      // Настройка как выходы push-pull, изначально высокий уровень
```

```
    GPIO_Init_t out_cfg = { .pin = scl_pin, .mode = GPIO_MODE_OUTPUT,
.otype = GPIO_OTYPE_PUSH_PULL, .speed = GPIO_SPEED_HIGH };
    GPIO_Init(scl_port, &out_cfg);
    out_cfg.pin = sda_pin;
    GPIO_Init(sda_port, &out_cfg);

    scl_high();
    sda_high();
}

void i2c_soft_start(void) {
    sda_high();
    scl_high();
    i2c_delay();
    sda_low();
    i2c_delay();
    scl_low();
}

void i2c_soft_stop(void) {
    sda_low();
    scl_high();
    i2c_delay();
    sda_high();
    i2c_delay();
}

bool i2c_soft_write_byte(uint8_t data) {
    for (int i = 0; i < 8; i++) {
        if (data & 0x80) sda_high();
```

```
        else sda_low();
        data <<= 1;
        i2c_delay();
        scl_high();
        i2c_delay();
        scl_low();
    }
    // Бит ACK
    set_sda_in();
    i2c_delay();
    scl_high();
    i2c_delay();
    bool ack = !read_sda();  // ACK = 0 (SDA низкий)
    scl_low();
    set_sda_out();
    return ack;
}

uint8_t i2c_soft_read_byte(bool send_ack) {
    uint8_t data = 0;
    set_sda_in();
    for (int i = 0; i < 8; i++) {
        data <<= 1;
        scl_high();
        i2c_delay();
        if (read_sda()) data |= 1;
        scl_low();
        i2c_delay();
    }
    // Генерация ACK или NACK
```

```
    if (send_ack) {
        set_sda_out(); sda_low();
    } else {
        set_sda_out(); sda_high();
    }
    i2c_delay();
    scl_high();
    i2c_delay();
    scl_low();
    set_sda_in();
    return data;
}

bool i2c_soft_write(uint8_t dev_addr, uint8_t reg_addr, uint8_t data) {
    i2c_soft_start();
    if (!i2c_soft_write_byte(dev_addr << 1)) { i2c_soft_stop(); return false; }
    if (!i2c_soft_write_byte(reg_addr)) { i2c_soft_stop(); return false; }
    if (!i2c_soft_write_byte(data)) { i2c_soft_stop(); return false; }
    i2c_soft_stop();
    return true;
}

bool i2c_soft_read(uint8_t dev_addr, uint8_t reg_addr, uint8_t* data) {
    i2c_soft_start();
    if (!i2c_soft_write_byte(dev_addr << 1)) { i2c_soft_stop(); return false; }
    if (!i2c_soft_write_byte(reg_addr)) { i2c_soft_stop(); return false; }
    i2c_soft_start();
    if (!i2c_soft_write_byte((dev_addr << 1) | 1)) { i2c_soft_stop(); return
false; }
    *data = i2c_soft_read_byte(false);
```

```
    i2c_soft_stop();
    return true;
}
```

2. **Реализация драйвера EEPROM 24LCxx**

2.1. Создайте файл eeprom.h:

```
#ifndef EEPROM_H
#define EEPROM_H

#include <stdint.h>
#include <stdbool.h>

#define EEPROM_ADDR 0x50    // 7-битный адрес 24LC02 (A2A1A0 = 000)

// Инициализация (настройка I2C)
void eeprom_init(void);

// Проверка присутствия EEPROM на шине
bool eeprom_ping(void);

// Запись байта по 8-битному адресу (для 24LC02)
bool eeprom_write_byte(uint8_t mem_addr, uint8_t data);

// Чтение байта по 8-битному адресу
bool eeprom_read_byte(uint8_t mem_addr, uint8_t* data);

// Запись страницы (максимум 16 байт для 24LC02)
bool eeprom_write_page(uint8_t start_addr, uint8_t* data, uint8_t len);
```

```
// Последовательное чтение (произвольной длины)
bool eeprom_read_buffer(uint8_t start_addr, uint8_t* buffer, uint16_t len);

// Ожидание завершения внутренней записи (опрос ACK)
void eeprom_wait_ready(void);

#endif
```

2.2. Создайте файл eeprom.c:

```
#include "eeprom.h"
#include "i2c_soft.h"  // или аппаратный драйвер
#include "timers.h"

void eeprom_init(void) {
    i2c_soft_init(8, 9);  // например, PB8=SCL, PB9=SDA
}

bool eeprom_ping(void) {
    i2c_soft_start();
    bool ack = i2c_soft_write_byte(EEPROM_ADDR << 1);
    i2c_soft_stop();
    return ack;
}

void eeprom_wait_ready(void) {
    while (1) {
        i2c_soft_start();
        if (i2c_soft_write_byte(EEPROM_ADDR << 1)) {
            i2c_soft_stop();
            break;
```

```
        }
        i2c_soft_stop();
        SysTick_DelayMs(1);  // пауза между попытками
    }
}

bool eeprom_write_byte(uint8_t mem_addr, uint8_t data) {
    eeprom_wait_ready();
    bool ok = i2c_soft_write(EEPROM_ADDR, mem_addr, data);
    if (ok) {
        // EEPROM переходит в цикл записи, нужно подождать
        SysTick_DelayMs(5);  // максимальное время записи (или wait_ready)
    }
    return ok;
}

bool eeprom_read_byte(uint8_t mem_addr, uint8_t* data) {
    eeprom_wait_ready();
    return i2c_soft_read(EEPROM_ADDR, mem_addr, data);
}

bool eeprom_write_page(uint8_t start_addr, uint8_t* data, uint8_t len) {
    if (len > 16) return false;  // 24LC02: страница 16 байт
    eeprom_wait_ready();
    i2c_soft_start();
    if (!i2c_soft_write_byte(EEPROM_ADDR << 1)) { i2c_soft_stop(); return
false; }
    if (!i2c_soft_write_byte(start_addr)) { i2c_soft_stop(); return false; }
    for (int i = 0; i < len; i++) {
        if (!i2c_soft_write_byte(data[i])) { i2c_soft_stop(); return false; }
```

```
    }
    i2c_soft_stop();
    SysTick_DelayMs(5);
    return true;
}

bool eeprom_read_buffer(uint8_t start_addr, uint8_t* buffer, uint16_t len) {
    eeprom_wait_ready();
    i2c_soft_start();
    if (!i2c_soft_write_byte(EEPROM_ADDR << 1)) { i2c_soft_stop(); return
false; }
    if (!i2c_soft_write_byte(start_addr)) { i2c_soft_stop(); return false; }
    i2c_soft_start();
    if (!i2c_soft_write_byte((EEPROM_ADDR << 1) | 1)) { i2c_soft_stop();
return false; }
    for (uint16_t i = 0; i < len; i++) {
        bool last = (i == len - 1);
        buffer[i] = i2c_soft_read_byte(!last);
    }
    i2c_soft_stop();
    return true;
}
```

3. **Проверка работы EEPROM через CLI**

   3.1. Добавьте команды в cli.c (из ЛР10):

```
static void cmd_eeprom_read(int argc, char** argv);
static void cmd_eeprom_write(int argc, char** argv);
static void cmd_eeprom_dump(int argc, char** argv);

// Добавить в таблицу commands[]
```

```
    {"eeprom_read", cmd_eeprom_read, "eeprom_read <addr> - read byte from EEPROM"},
    {"eeprom_write", cmd_eeprom_write, "eeprom_write <addr> <byte> - write byte to EEPROM"},
    {"eeprom_dump", cmd_eeprom_dump, "eeprom_dump <addr> <len> - dump memory"},

    static void cmd_eeprom_read(int argc, char** argv) {
        if (argc < 2) return;
        uint8_t addr = (uint8_t)atoi(argv[1]);
        uint8_t value;
        if (eeprom_read_byte(addr, &value)) {
            uart_printf("EEPROM[%u] = %u (0x%02X)\r\n", addr, value, value);
        } else {
            uart_puts("Read failed\r\n");
        }
    }

    static void cmd_eeprom_write(int argc, char** argv) {
        if (argc < 3) return;
        uint8_t addr = (uint8_t)atoi(argv[1]);
        uint8_t value = (uint8_t)atoi(argv[2]);
        if (eeprom_write_byte(addr, value)) {
            uart_puts("Write OK\r\n");
        } else {
            uart_puts("Write failed\r\n");
        }
    }

    static void cmd_eeprom_dump(int argc, char** argv) {
```

```
    if (argc < 3) return;
    uint8_t addr = (uint8_t)atoi(argv[1]);
    uint16_t len = (uint16_t)atoi(argv[2]);
    uint8_t buffer[64];
    for (uint16_t i = 0; i < len; i += 16) {
      uint16_t chunk = (len - i > 16) ? 16 : (len - i);
      if (eeprom_read_buffer(addr + i, buffer, chunk)) {
        uart_printf("%04X: ", addr + i);
        for (uint16_t j = 0; j < chunk; j++) {
          uart_printf("%02X ", buffer[j]);
        }
        uart_puts("\r\n");
      } else {
        uart_puts("Read error\r\n");
        break;
      }
    }
  }
```

4. **Аппаратный I2C (альтернатива)**

4.1. Для STM32F4 настройте I2C1 (PB6=SCL, PB7=SDA). Код настройки (упрощённо):

```
  void i2c_hw_init(void) {
    RCC->APB1ENR |= (1<<21);  // I2C1EN
    RCC->AHB1ENR |= (1<<1);   // GPIOB
    // PB6, PB7 -> AF4
    GPIOB->MODER &= ~(3<<(6*2)); GPIOB->MODER |= (2<<(6*2));
    GPIOB->MODER &= ~(3<<(7*2)); GPIOB->MODER |= (2<<(7*2));
    GPIOB->AFR[0] |= (4<<(6*4)) | (4<<(7*4));  // AF4
```

```
    // I2C1 настройка: Standard mode 100 кГц, подтяжка включена (open-drain)
    I2C1->CR2 = 84;  // APB1 frequency in MHz
    I2C1->CCR = 420;  // для 100 кГц (84e6 / 100000 / 2)
    I2C1->TRISE = 85;
    I2C1->CR1 |= (1<<0);  // PE
}
```

4.2. Сравните производительность и нагрузку CPU с программным вариантом.

5. **Обработка ошибок шины**

5.1. Добавьте в драйвер обработку тайм-аута:

```
#define I2C_TIMEOUT_MS 100
bool i2c_soft_write_byte_timeout(uint8_t data, uint32_t timeout_ms) {
    uint32_t start = SysTick_GetTick();
    while (/* ждём пока шина освободится? */) {
        if ((SysTick_GetTick() - start) > timeout_ms) return false;
    }
    // ... остальная логика
}
```

Пример выполнения задания

Задача: Записать в EEPROM по адресу 0x10 значение 0xAA, затем прочитать его и вывести через UART.

Код основной программы:

```
#include "eeprom.h"
#include "uart_driver.h"

int main(void) {
```

```
    SysTick_Init(1000);
    uart_init(115200);
    eeprom_init();

    uart_puts("EEPROM test\r\n");

    if (!eeprom_ping()) {
        uart_puts("EEPROM not found!\r\n");
        while (1);
    }

    uint8_t test_value = 0xAA;
    if (eeprom_write_byte(0x10, test_value)) {
        uart_puts("Write successful\r\n");
    } else {
        uart_puts("Write failed\r\n");
    }

    SysTick_DelayMs(10);  // гарантия завершения записи

    uint8_t read_value;
    if (eeprom_read_byte(0x10, &read_value)) {
        uart_printf("Read back: 0x%02X\r\n", read_value);
    } else {
        uart_puts("Read failed\r\n");
    }

    while (1);
}
```

Результат в терминале:

EEPROM test

Write successful

Read back: 0xAA

Индивидуальные задания

Вариант 1. Страничная запись и проверка

Запишите 16 байт (0..15) в EEPROM начиная с адреса 0x00, затем считайте их и проверьте. Реализуйте проверку с индикацией ошибки через светодиод.

Вариант 2. Циклический буфер в EEPROM

Организуйте в EEPROM циклический буфер на 256 байт. Функции: eeprom_fifo_write(byte), eeprom_fifo_read(byte*). Используйте два байта для хранения индексов head/tail.

Вариант 3. Хранение конфигурации устройства

Определите структуру параметров (например, режим светодиода, частота мигания, скорость АЦП). Сохраняйте её в EEPROM при изменении через CLI и восстанавливайте при старте.

Вариант 4. Логгер ошибок

При каждом сбое (переполнение буфера, потеря данных) записывайте код ошибки и временную метку в EEPROM. Реализуйте команду error_log для вывода журнала.

Вариант 5. Эмуляция EEPROM во Flash МК

Если внешней EEPROM нет, используйте внутреннюю Flash-память микроконтроллера (секция для данных) с эмуляцией страничной записи. Реализуйте те же функции.

Вариант 6. Работа с 24LC256 (16-битный адрес)

Модифицируйте драйвер для поддержки 16-битного внутреннего адреса. Реализуйте запись/чтение по адресам до 32767.

Вариант 7. Драйвер I2C с использованием DMA

Настройте аппаратный I2C + DMA для чтения и записи страниц без участия CPU. Сравните скорость.

Вариант 8. Сканирование устройств на шине I2C

Напишите функцию, которая перебирает все 127 адресов и выводит список устройств, ответивших ACK. Вызовите её по команде i2c_scan.

Вариант 9. Защита от записи (Write Protect)

Если EEPROM имеет вывод WP, подключите его к GPIO и реализуйте команды wp enable и wp disable. Проверьте, что при включённой защите запись не выполняется.

Вариант 10. Ускорение программного I2C

Используйте задержки на основе цикла с учётом тактовой частоты (без делителей). Добейтесь частоты 400 кГц. Оцените максимальную скорость на вашем МК.

Вариант 11. Обработка ошибки "потеря арбитража" (в программном I2C)

Модифицируйте функцию отправки байта, чтобы она проверяла уровень SDA в середине бита и возвращала ошибку при коллизии.

Вариант 12. Энергонезависимый счётчик

Используйте EEPROM для хранения 32-битного счётчика, который инкрементируется при нажатии кнопки. При каждом включении счётчик восстанавливается. Сделайте команду counter.

Вариант 13. I2C с прерываниями на аппаратном I2C

Реализуйте неблокирующую запись/чтение с помощью прерываний I2C (главный режим). Используйте конечные автоматы состояний.

Вариант 14. Чтение уникального ID EEPROM (если поддерживается)

Некоторые EEPROM имеют уникальный серийный номер. Реализуйте чтение по специальной команде (например, для Microchip 24AAxx).

Вариант 15. Поддержка двух EEPROM на одной шине (разные адреса)

Подключите две микросхемы EEPROM (например, с адресами 0x50 и 0x52). Реализуйте команды eeprom_select <0/1> и соответствующие операции.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.

2. Цель и задачи лабораторной работы.

3. Теоретическую часть (протокол I2C: старт/стоп, передача данных, ACK/NACK, адресация EEPROM, временные диаграммы).

4. Листинги: i2c_soft.h, i2c_soft.c, eeprom.h, eeprom.c, дополнения к CLI.

5. Схема подключения EEPROM к микроконтроллеру (с указанием номиналов подтягивающих резисторов).

6. Описание тестов: ping, запись/чтение одного байта, страничная запись, дамп памяти.

7. Результаты выполнения индивидуального задания.

8. Сравнение программного и аппаратного I2C (если выполнено) по скорости, загрузке CPU, надёжности.

9. Выводы (что дало понимание протокола, типичные ошибки и способы их устранения).

10. Ответы на контрольные вопросы.

Контрольные вопросы

1. Для чего нужен протокол I2C и каковы его основные области применения?

2. Сколько линий используется в I2C? Почему нужны подтягивающие резисторы?

3. Объясните процедуру START и STOP условия.

4. Что означает бит ACK (подтверждение) и NACK? Как они формируются?

5. Какие скорости передачи данных поддерживает I2C (стандартные)?

6. Что такое 7-битный адрес устройства и как он преобразуется в первый байт на шине?

7. Как отличить запись от чтения в I2C?

8. Как выполняется случайное чтение из EEPROM (процедура)?

9. Что такое «внутренний цикл записи» EEPROM и как программа узнаёт о его окончании?

10. Какие типичные ошибки могут возникнуть на шине I2C и как их обрабатывать?

11. В чём преимущества аппаратного I2C перед программным (bit-banging)?

12. Как реализовать программный I2C без использования прерываний? Какова максимальная частота?

13. Почему в программном I2C важно соблюдать тайминги (задержки)?

14. Как влияют подтягивающие резисторы на форму сигналов и максимальную скорость?

15. Что произойдёт, если два устройства одновременно начнут передачу (арбитраж)?

16. Как организовать на шине несколько ведомых с одинаковым адресом?

17. Зачем в команде страничной записи ограничение длины страницы?

18. Какую роль играет функция eeprom_wait_ready() и почему в ней используется опрос ACK?

19. Как с помощью I2C считать данные из EEPROM, начиная с адреса 0x100 (для 16-битной адресации)?

20. Как измерить время, затрачиваемое на запись байта в EEPROM? Какое типовое значение?

Лабораторная работа №12

Прямой доступ к памяти (DMA) для высокопроизводительной передачи

Цель работы: освоение механизма прямого доступа к памяти (DMA) на примере передачи данных между UART и памятью без участия процессора, реализация кольцевого буфера с использованием DMA в циклическом режиме, сравнение производительности и загрузки CPU с традиционным методом прерываний.

Задачи:

- Изучить архитектуру контроллера DMA в микроконтроллерах STM32F4: каналы, потоковые каналы, приоритеты, флаги.
- Настроить DMA для автоматического приёма данных из UART в массив памяти.
- Реализовать два режима: одноразовая передача блока данных и циклический режим (circular mode) для непрерывного заполнения буфера.
- Организовать кольцевой буфер на основе DMA с двойным управлением: DMA пишет в буфер, основная программа читает.
- Обработать прерывание от DMA по заполнению половины буфера и полному заполнению (двойная буферизация).
- Сравнить загрузку процессора и максимальную скорость приёма для методов: опрос, прерывания, DMA.
- Интегрировать DMA-приём в командный интерпретатор UART.

Теоретическая часть

Прямой доступ к памяти (DMA, Direct Memory Access)

DMA – это аппаратный модуль, который может самостоятельно выполнять пересылку данных между периферией и памятью (или память-

память) без участия процессора. Процессор лишь инициализирует передачу (указывает источник, приёмник, длину) и получает прерывание по её завершении.

Преимущества DMA:

- Освобождает CPU от копирования каждого байта.
- Позволяет обрабатывать высокие скорости передачи (например, 1 Мбит/с и выше), где прерывания создают слишком большую нагрузку.
- Уменьшает энергопотребление (CPU может засыпать во время передачи).

DMA в STM32F4

STM32F4 имеет два контроллера DMA (DMA1 и DMA2). Каждый контроллер имеет 8 потоков (streams). Каждый поток поддерживает несколько каналов (channels), привязанных к периферии.

Например, для USART2_RX:

- DMA1, Stream 5, Channel 4 (зависит от модели, уточнить по reference manual).

Основные регистры DMA (на примере DMA1_Stream5):

- CR – управление: включение, направление, режим (циклический/одноразовый), размер данных, приоритет.
- NDTR – количество данных для передачи (счётчик).
- PAR – адрес периферийного регистра (например, &USART2->DR).
- M0AR – адрес памяти (буфера).
- M1AR – для двойного буфера (режим double buffer).
- FCR – управление FIFO.

Режимы работы:

- **Normal Mode** – после передачи NDTR байтов DMA останавливается. Требуется переинициализация для следующей передачи.
- **Circular Mode** – после достижения конца буфера DMA автоматически возвращается в начало и продолжает передачу. Идеален для кольцевого буфера.
- **Double Buffer Mode** – переключение между двумя буферами (M0AR и M1AR). Позволяет обрабатывать один буфер, пока DMA пишет в другой.

DMA для приёма UART:

UART генерирует запрос DMA по событию RXNE (принят новый байт). DMA автоматически читает байт из USARTx->DR и записывает его в буфер в памяти. Процессор получает прерывание только когда DMA заполнит заданный блок (например, половину буфера или весь буфер).

Кольцевой буфер с DMA:

Используя Circular Mode, DMA непрерывно пишет данные в буфер. В основной программе читатель использует указатели: head обновляется аппаратно (DMA), tail – программно. Нужно знать текущую позицию записи DMA – её можно получить из регистра NDTR:

current_write_index = buffer_size - DMA_Stream->NDTR;

Флаги и прерывания DMA:

- TCIF (Transfer Complete) – передача завершена.
- HTIF (Half Transfer) – передано половина буфера (удобно для двойной буферизации).
- TEIF (Transfer Error) – ошибка.
- DMEIF (Direct Mode Error).

Когерентность кэша

Если в системе используется кэш (у Cortex-M7 и выше), необходимо следить за тем, чтобы данные, которые DMA пишет в память, не лежали в кэше

CPU (или использовать SCB_CleanInvalidateDCache). Для Cortex-M4 кэша нет, поэтому проблемы нет.

Сравнение методов приёма UART:

Метод	Загрузка CPU на 115200 бод	Макс. скорость	Сложность
Опрос (polling)	100% (ждёт байт)	низкая	простая
Прерывание	~5-10%	до ~1 Мбит/с	средняя
DMA	~0-1%	до 10 Мбит/с+	выше

Методика выполнения работы

1. **Подготовка DMA-драйвера для UART RX**

   1.1. Создайте каталог lab12 и файл dma_uart.h:

```
#ifndef DMA_UART_H
#define DMA_UART_H

#include <stdint.h>
#include <stdbool.h>

// Структура кольцевого буфера на основе DMA
typedef struct {
    uint8_t* buffer;
    volatile uint32_t head;      // индекс, куда пишет DMA (аппаратный, читается из NDTR)
    volatile uint32_t tail;    // индекс, откуда читает пользователь
    uint32_t size;             // размер буфера
    volatile uint32_t overflow_cnt;
} dma_ring_buffer_t;
```

```
// Инициализация UART2 с DMA приёмом в кольцевой буфер
void dma_uart_init(uint32_t baudrate, uint8_t* buffer, uint32_t buffer_size);

// Проверка, есть ли данные для чтения
bool dma_uart_available(void);

// Чтение одного байта из буфера (неблокирующее)
bool dma_uart_getc(uint8_t* c);

// Получение количества непрочитанных байтов
uint32_t dma_uart_pending(void);

// Получить статистику переполнений
uint32_t dma_uart_overflow(void);

// Сброс буфера и счётчиков
void dma_uart_reset(void);

// Приостановить DMA (например, для изменения скорости UART)
void dma_uart_pause(void);
void dma_uart_resume(void);

#endif
```

1.2. Создайте файл dma_uart.c:

```
#include "dma_uart.h"
#include "stm32f4xx.h"
#include <string.h>
```

```
static dma_ring_buffer_t rb;
static DMA_Stream_TypeDef* dma_stream;
static uint32_t dma_channel;

// Определение DMA1 Stream5 для USART2_RX (проверьте по документации)
#define USART2_RX_DMA_STREAM    DMA1_Stream5
#define USART2_RX_DMA_CHANNEL   4
#define USART2_RX_DMA_IRQn      DMA1_Stream5_IRQn

void dma_uart_init(uint32_t baudrate, uint8_t* buffer, uint32_t buffer_size) {
   // Сохраняем параметры кольцевого буфера
   rb.buffer = buffer;
   rb.size = buffer_size;
   rb.tail = 0;
   rb.overflow_cnt = 0;

   // --- Настройка UART2 ---
   RCC->APB1ENR |= (1 << 17);  // USART2EN
   RCC->AHB1ENR |= (1 << 0);   // GPIOA

   // PA2 - TX, PA3 - RX как AF7
   GPIOA->MODER &= ~(3 << (2*2));
   GPIOA->MODER |= (2 << (2*2));
   GPIOA->AFR[0] |= (7 << (2*4));

   GPIOA->MODER &= ~(3 << (3*2));
   GPIOA->MODER |= (2 << (3*2));
   GPIOA->AFR[0] |= (7 << (3*4));
```

```
// Расчёт BRR
uint32_t clock = 84000000;
USART2->BRR = clock / baudrate;
USART2->CR1 = (1 << 13);   // UE
USART2->CR2 = 0;
USART2->CR1 |= (1 << 2);   // RE
USART2->CR1 |= (1 << 3);   // TE

// --- Настройка DMA для UART RX ---
// Включение тактирования DMA1
RCC->AHB1ENR |= (1 << 21);  // DMA1EN

dma_stream = USART2_RX_DMA_STREAM;
dma_channel = USART2_RX_DMA_CHANNEL;

// Сброс настроек потока
dma_stream->CR = 0;

// Пока DMA выключен, ждём возможного сброса
while (dma_stream->CR & 1);

// Настройка: приём от периферии в память, 8 бит, циклический режим
dma_stream->CR &= ~(0x3 << 6);   // DIR: 0 = peripheral-to-memory
dma_stream->CR |= (1 << 8);      // CIRC =1 (циклический режим)
dma_stream->CR |= (1 << 9);      // PINC =0 (адрес периферии не инкрементируется)
dma_stream->CR |= (1 << 10);     // MINC =1 (инкремент памяти)
dma_stream->CR |= (0 << 11);     // PSIZE = 0 (8 бит)
dma_stream->CR |= (0 << 13);     // MSIZE = 0 (8 бит)
```

```
    dma_stream->CR |= (1 << 16);     // PINCOS =0 (не требуется)
    dma_stream->CR |= (2 << 17);     // PL = 10 (средний приоритет)

    // Настройка адресов
    dma_stream->PAR  =  (uint32_t)&USART2->DR;      //  адрес  регистра
данных
    dma_stream->M0AR = (uint32_t)rb.buffer;    // адрес буфера
    dma_stream->NDTR = rb.size;              // количество элементов

    // Настройка канала (выбор периферийного запроса)
    dma_stream->CR &= ~(0x1F << 25);
    dma_stream->CR |= (dma_channel << 25);

    // Разрешение прерываний от DMA: Half Transfer и Transfer Complete
    dma_stream->CR |= (1 << 2);   // HTIE
    dma_stream->CR |= (1 << 4);   // TCIE
    NVIC_EnableIRQ(USART2_RX_DMA_IRQn);
    NVIC_SetPriority(USART2_RX_DMA_IRQn, 1);

    // Разрешить UART запросы DMA на приём
    USART2->CR3 |= (1 << 6);  // DMAR =1

    // Включить DMA
    dma_stream->CR |= (1 << 0);  // EN
}

// Вычисление текущего индекса записи DMA
static uint32_t get_current_head(void) {
    return rb.size - dma_stream->NDTR;
}
```

```
// Количество доступных для чтения байтов
uint32_t dma_uart_pending(void) {
    uint32_t head = get_current_head();
    uint32_t tail = rb.tail;
    if (head >= tail) return head - tail;
    else return rb.size - tail + head;
}

bool dma_uart_available(void) {
    return dma_uart_pending() > 0;
}

bool dma_uart_getc(uint8_t* c) {
    if (dma_uart_pending() == 0) return false;
    *c = rb.buffer[rb.tail];
    rb.tail = (rb.tail + 1) % rb.size;
    return true;
}

uint32_t dma_uart_overflow(void) {
    return rb.overflow_cnt;
}

void dma_uart_reset(void) {
    // Сброс буфера и указателей (останавливаем DMA)
    dma_stream->CR &= ~1;  // Disable DMA
    rb.tail = 0;
    dma_stream->NDTR = rb.size;
    rb.overflow_cnt = 0;
```

```
    dma_stream->CR |= 1;   // Enable
}

// Обработчик прерываний DMA
void DMA1_Stream5_IRQHandler(void) {
    uint32_t flags = dma_stream->LISR;   // или HISR в зависимости от номера потока

    if (dma_stream->LISR & (1 << 6)) {  // HTIF для Stream5 (бит 6)
        dma_stream->LISR |= (1 << 6);   // очистка
        // Половина буфера заполнена. Здесь можно обработать первую половину
        // Устанавливаем флаг для основной программы
    }

    if (dma_stream->LISR & (1 << 7)) {  // TCIF
        dma_stream->LISR |= (1 << 7);
        // Весь буфер заполнен (в циклическом режиме это означает переполнение)
        rb.overflow_cnt++;
    }
}
```

2. **Настройка буфера и тестирование приёма**

2.1. Создайте файл main.c с демонстрацией:

```
#include "dma_uart.h"
#include "uart_driver.h"   // для передачи (TX)
#include "timers.h"

#define DMA_BUFFER_SIZE 256
```

```
static uint8_t dma_buffer[DMA_BUFFER_SIZE];

int main(void) {
    SysTick_Init(1000);
    uart_init(115200);   // для отправки (TX)
    dma_uart_init(115200, dma_buffer, DMA_BUFFER_SIZE);

    uart_puts("DMA UART RX test. Type something...\r\n");

    while (1) {
        if (dma_uart_available()) {
            uint8_t c;
            while (dma_uart_getc(&c)) {
                uart_putc(c);   // эхо через обычный UART TX
            }
        }

        // Вывод статистики каждые 5 секунд
        static uint32_t last = 0;
        if ((SysTick_GetTick() - last) >= 5000) {
            uint32_t pending = dma_uart_pending();
            uint32_t overflow = dma_uart_overflow();
            uart_printf("\r\nPending: %lu, Overflow: %lu\r\n", pending,
overflow);
            last = SysTick_GetTick();
        }
    }
}
```

3. **Измерение производительности**

3.1. Сравните с обычным прерыванием UART (из ЛР10). Для этого отключите DMA и включите обычный RXNE-прерывание. Направьте на UART поток данных (например, через Python-скрипт, отправляющий 1 Мбайт). Замерьте время приёма (через SysTick) и загрузку CPU (по времени idle-цикла).

3.2. Используйте осциллограф на выводе, который переключается при обработке байта в прерывании и при обработке в фоновом цикле.

4. **Реализация двойной буферизации (half-transfer прерывание)**

4.1. В обработчике DMA по флагу HTIF обрабатывайте первую половину буфера, пока DMA продолжает заполнять вторую. Это позволяет избежать гонок данных.

4.2. Добавьте флаг buffer_ready[2] и обработку:

```
static volatile uint8_t active_buffer = 0;
static volatile bool half_ready = false;
static volatile bool full_ready = false;

void DMA1_Stream5_IRQHandler(void) {
   if (flags & HTIF) {
      half_ready = true;
      active_buffer = 0;  // первая половина
   }
   if (flags & TCIF) {
      full_ready = true;
      active_buffer = 1;  // вторая половина
   }
}
```

5. **Кольцевой буфер через DMA без прерываний HT/TC**

Альтернативный подход: используем только циклический DMA, а в основном цикле раз в некоторое время проверяем текущую позицию head и обрабатываем все накопившиеся байты (как в примере выше). Это самый простой способ реализации кольцевого буфера.

6. **Эксперимент с переполнением буфера**

6.1. Уменьшите размер буфера до 16. Направьте большой объём данных. Наблюдайте счётчик overflow_cnt. Он будет увеличиваться, если основная программа не успевает читать.

6.2. Рассчитайте минимальный размер буфера для скорости 115200 бод (11.5 Кбайт/с) и максимального времени бездействия основной программы 10 мс → 115 байт. Выберите 128 или 256.

Пример выполнения задания

Задача: Принимать данные через UART с помощью DMA в буфер 256 байт. Основная программа обрабатывает принятые данные пакетами по 64 байта, отправляет эхо-подтверждение. Измерить максимальную скорость без потерь.

Код основной обработки:

```
#define PACKET_SIZE 64
static uint8_t packet[PACKET_SIZE];
static uint32_t packet_index = 0;

void process_packet(void) {
   // проверка контрольной суммы, выполнение команды
   uart_printf("Packet received, %u bytes\r\n", PACKET_SIZE);
}

int main(void) {
   // init
   while (1) {
```

```
        uint8_t c;
        while (dma_uart_getc(&c)) {
            packet[packet_index++] = c;
            if (packet_index >= PACKET_SIZE) {
                process_packet();
                packet_index = 0;
            }
        }
    }
}
```

Результаты тестирования:

- При 115200 бод потери данных отсутствуют при буфере 256 байт.
- При 921600 бод (максимум для UART) потери возникают, если обработка пакетов длится более ~2 мс. Переход на более быструю обработку (меньше printf) решает проблему.
- Загрузка CPU при 115200 бод составила <1% (только фоновая обработка). При методе прерываний нагрузка была около 8%.

Индивидуальные задания

Вариант 1. DMA для передачи UART (TX)

Реализуйте передачу строки через DMA (без участия CPU). Команда dma_send отправляет буфер. Измерьте скорость и загрузку.

Вариант 2. Два буфера для приёма без прерываний

Настройте DMA в нормальном режиме на два буфера: один заполняется DMA, другой обрабатывается. По прерыванию TCIF переключайте буферы. Реализуйте без циклического режима.

Вариант 3. DMA + АЦП (непрерывный сбор)

Используйте DMA для автоматического сбора данных АЦП по таймеру. Размер буфера 512. В прерывании полузаполнения буфера обрабатывайте данные (усреднение). Сравните с методом прерываний.

Вариант 4. DMA Memory-to-Memory

Настройте DMA для копирования одного массива в другой (без периферии). Замерьте скорость копирования и сравните с memcpy().

Вариант 5. Шифрование потока (аппаратный криптоускоритель)

Используйте DMA для подачи данных в модуль AES (если есть) и приёма зашифрованного результата. Реализуйте конвейерную обработку.

Вариант 6. Обнаружение потери данных DMA

Добавьте в обработчик DMA проверку флага TEIF (Transfer Error). Выводите сообщение об ошибке.

Вариант 7. Управление DMA из CLI

Добавьте команды: dma_status (показать NDTR, текущий head, pending), dma_reset.

Вариант 8. Приём данных с переменной длиной пакета (USART idle line)

Используйте прерывание UART IDLE для определения конца сообщения. В комбинации с DMA это эффективно. Реализуйте приём пакетов без фиксации размера.

Вариант 9. DMA с таймаутом (программный)

Если после последнего принятого байта прошло более N мс, а данные в буфере есть – передать их на обработку как "пакет". Реализуйте для случая, когда терминал не отправляет символ новой строки.

Вариант 10. Анализ времени реакции DMA

Используйте вывод GPIO для измерения времени от появления байта на входе UART до момента, когда он становится доступен в памяти. Сравните с прерыванием.

Вариант 11. Динамическое изменение размера буфера

Реализуйте возможность изменить размер буфера DMA во время работы (остановить DMA, переинициализировать, скопировать старые данные).

Вариант 12. DMA для SPI-дисплея

Настройте DMA для отправки данных на SPI-дисплей (кадр). Реализуйте двойную буферизацию для анимации без мерцания.

Вариант 13. Многопоточный доступ к буферу DMA (FreeRTOS)

Если используется RTOS, реализуйте задачу-читатель, которая ожидает семафор от прерывания DMA (half/full). Задача обрабатывает данные и не мешает другим задачам.

Вариант 14. Энергосбережение + DMA

Переведите CPU в режим SLEEP после запуска DMA. Пробуждение по прерыванию DMA. Замерьте потребление тока.

Вариант 15. Передача файла через UART с протоколом XMODEM

Используйте DMA для приёма блоков XMODEM (128 байт). Реализуйте подтверждение. Сравните удобство разработки с прерываниями.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (принципы DMA, регистры, режимы циркулярного и двойного буфера, когерентность).
4. Листинги: dma_uart.h, dma_uart.c, main.c.
5. Схему подключения (только UART).
6. Таблицы и графики сравнения загрузки CPU и максимальной скорости для трёх методов (опрос, прерывания, DMA).
7. Результаты индивидуального задания.
8. Выводы о целесообразности использования DMA в различных сценариях (высокая скорость, низкое энергопотребление, многозадачность).

9. Ответы на контрольные вопросы.

Контрольные вопросы

1. Что такое DMA и для каких задач он применяется во встраиваемых системах?

2. Какие два основных режима работы DMA существуют в STM32 (normal и circular)?

3. Как в циклическом режиме DMA определить текущую позицию записи в буфере?

4. Какие прерывания может генерировать DMA?

5. Зачем нужно прерывание по полузаполнению буфера (Half Transfer)?

6. Как происходит настройка потока DMA для приёма из UART (регистр CR, PAR, M0AR, NDTR)?

7. Почему для UART RX нужно включать циклический режим и инкремент адреса памяти, но не инкремент адреса периферии?

8. Как рассчитать размер кольцевого буфера для приёма на скорости 115200 бод, если максимальная задержка обработки в основной программе составляет 50 мс?

9. В чём преимущество DMA перед прерываниями при высоких скоростях передачи?

10. Какую максимальную скорость можно достичь на STM32F4 при использовании DMA для UART? Что ограничивает скорость?

11. Как DMA влияет на энергопотребление по сравнению с методом прерываний?

12. Как решить проблему когерентности кэша при использовании DMA на ядрах Cortex-M7?

13. Какие ошибки могут возникнуть при работе DMA (TEIF, DMEIF) и как их обрабатывать?

14. Можно ли использовать один поток DMA для приёма и передачи одновременно? Почему?

15. Что произойдёт, если NDTR установить в 0?

16. Как организовать двойную буферизацию с помощью DMA без остановки потока?

17. Как изменить размер буфера DMA во время работы?

18. Как проверить, что DMA действительно освободил процессор, а не просто маскирует загрузку?

19. Как работает арбитраж между потоками DMA и доступом CPU к шине памяти?

20. В каком сценарии предпочтительнее использовать DMA, а в каком – прерывания?

Лабораторная работа №13

Моделирование многозадачности (Round Robin)

Цель работы: освоение принципов кооперативной многозадачности на примере суперцикла (Round Robin) и диспетчера задач на основе таймерных прерываний, реализация простейшего планировщика без вытеснения (non-preemptive scheduler) и сравнение его с кооперативным подходом.

Задачи:

- Изучить основные модели многозадачности во встраиваемых системах: суперцикл (бесконечный цикл), кооперативная (Round Robin с добровольной передачей управления), вытесняющая (на базе прерываний таймера).
- Реализовать простой суперцикл с последовательным вызовом функций задач.
- Реализовать кооперативный планировщик (Round Robin) с передачей управления через вызов task_yield().
- Реализовать вытесняющий планировщик на основе системного таймера (тики 1 мс), который переключает задачи по кванту времени.
- Сравнить поведение системы при разных подходах на примере задач с различной длительностью выполнения и разными требованиями к реакции на события.
- Проанализировать проблемы гонок данных и необходимость синхронизации (критические секции, семафоры) при вытеснении.
- Реализовать простую обработку событий (флаги) для связи между задачами.

Теоретическая часть

Модели многозадачности в bare-metal системах

В системах без операционной системы (bare-metal) многозадачность реализуется программно. Основные подходы:

1. **Суперцикл (Super Loop)** – бесконечный цикл, в котором последовательно вызываются функции. Каждая функция должна выполняться быстро, не блокируя другие.

```
while (1) {
   task1();
   task2();
   task3();
}
```

2. **Кооперативная многозадачность (Cooperative)** – задачи выполняются циклически, но каждая задача явно передаёт управление планировщику (например, через task_yield()). Задача сама решает, когда её прервать.
3. **Вытесняющая многозадачность (Preemptive)** – переключение задач происходит по прерыванию таймера (квант времени). Задача может быть прервана в любой момент. Требует сохранения контекста (регистры, стек).

Round Robin (карусель)

Простейший алгоритм планирования: задачи выполняются в фиксированном порядке, каждая получает одинаковый квант времени (или выполняется до добровольной уступки). В кооперативном варианте порядок циклический, в вытесняющем – добавляется квантование по таймеру.

Контекст задачи

Для вытесняющей многозадачности нужно сохранять состояние каждой задачи:

- Указатель стека
- Регистры (R4–R11, PC, LR, и др.)
- В ARM Cortex-M частичное сохранение происходит автоматически при входе в прерывание.

Простейший планировщик на тиках:

```
#define MAX_TASKS 4

typedef void (*task_func_t)(void);

task_func_t tasks[MAX_TASKS];
uint32_t task_delays[MAX_TASKS];
uint32_t current_task = 0;

void scheduler_tick(void) {  // вызывается из SysTick каждые N мс
    if (--task_delays[current_task] == 0) {
        current_task = (current_task + 1) % MAX_TASKS;
        // Вызов следующей задачи (в реальном планировщике нужно переключение контекста)
    }
}
```

Критические секции и синхронизация

При вытеснении доступ к общим данным нужно защищать запретом прерываний или использованием атомарных операций.

Кооперативный планировщик без прерываний

```
void task1(void) {
    while (1) {
        // работа
        task_yield();  // передать управление следующей задаче
    }
}
```

Планировщик хранит указатель на текущую задачу. task_yield() сохраняет текущий контекст (только PC и флаги, потому что стек остаётся) и переходит к следующей.

Методика выполнения работы

1. **Разработка простого суперцикла**

   1.1. Создайте каталог lab13 и файл superloop.c:

```
#include "led_driver.h"
#include "uart_driver.h"
#include "timers.h"
#include <stdio.h>

volatile uint32_t task1_counter = 0;
volatile uint32_t task2_counter = 0;

void task_led_blink(void) {
    static uint32_t last = 0;
    if ((SysTick_GetTick() - last) >= 500) {
        led_toggle();
        last = SysTick_GetTick();
    }
}

void task_uart_report(void) {
    static uint32_t last = 0;
    if ((SysTick_GetTick() - last) >= 2000) {
        uart_printf("Task1: %lu, Task2: %lu\r\n", task1_counter, task2_counter);
        last = SysTick_GetTick();
    }
}

void task_busy(void) {
    // Длительная операция, которая блокирует другие задачи
```

```
    for (volatile int i = 0; i < 50000; i++) {
        task1_counter++;
    }
}

int main(void) {
    SysTick_Init(1000);
    led_init();
    uart_init(115200);

    uart_puts("Superloop start\r\n");

    while (1) {
        task_led_blink();     // быстрая задача
        task_busy();          // долгая задача - задерживает другие!
        task_uart_report();   // эта задача будет выполняться редко
    }
}
```

1.2. Пронаблюдайте, как долгая задача task_busy задерживает выполнение других. Это основной недостаток суперцикла.

2. **Реализация кооперативного планировщика Round Robin (с добровольной уступкой)**

   2.1. Создайте файл cooperative_scheduler.h:

```
#ifndef COOP_SCHED_H
#define COOP_SCHED_H

#include <stdint.h>
```

```
#define MAX_TASKS 8

typedef void (*task_t)(void);

void scheduler_init(void);
uint8_t scheduler_add_task(task_t task);
void scheduler_run(void);
void scheduler_yield(void);

#endif
```

2.2. Создайте файл cooperative_scheduler.c:

```
#include "cooperative_scheduler.h"

static task_t tasks[MAX_TASKS];
static uint8_t task_count = 0;
static uint8_t current_task = 0;

void scheduler_init(void) {
    task_count = 0;
    current_task = 0;
}

uint8_t scheduler_add_task(task_t task) {
    if (task_count >= MAX_TASKS) return 0;
    tasks[task_count++] = task;
    return 1;
}

void scheduler_yield(void) {
```

```
    // Переключение на следующую задачу
    current_task = (current_task + 1) % task_count;
    // Здесь нужно перейти к выполнению следующей задачи.
    // В простейшем случае просто возврат, а планировщик вызывает следующую.
}

void scheduler_run(void) {
    while (1) {
        if (tasks[current_task]) {
            tasks[current_task]();  // выполнить задачу
            // После возврата из задачи переходим к следующей
            current_task = (current_task + 1) % task_count;
        }
    }
}
```

Однако этот код не позволяет задаче добровольно уступать внутри своего цикла. Нужно, чтобы задача содержала бесконечный цикл, внутри которого вызывает scheduler_yield(). Поэтому изменим подход: каждая задача – это бесконечный цикл, а планировщик переключает задачи после yield. Для этого нужно сохранять контекст (PC, SP) при переключении.

Упростим: сделаем задачи функциями, которые выполняются один раз за цикл, а долгие операции разбиты на шаги (конечные автоматы). Либо используем готовую библиотеку-планировщик.

Для лабораторной работы предложим реализацию на основе массива task-указателей, где каждая задача сама решает, когда вернуть управление.

```
void task_led(void) {
    static uint32_t last = 0;
    if ((SysTick_GetTick() - last) >= 100) {
```

```
        led_toggle();
        last = SysTick_GetTick();
    }
    scheduler_yield();   // уступаем после проверки
}

void task_busy_partial(void) {
    static uint32_t step = 0;
    for (int i = 0; i < 100; i++) {
        task1_counter++;
    }
    step++;
    if (step >= 500) step = 0;
    scheduler_yield();
}
```

В main: создаём задачи, запускаем scheduler_run().

3. **Вытесняющий планировщик с квантованием по таймеру (без RTOS)**

3.1. Создайте файл preemptive_scheduler.h:

```
#ifndef PREEMPT_SCHED_H
#define PREEMPT_SCHED_H

#include <stdint.h>

#define MAX_TASKS 8
#define STACK_SIZE 256

typedef void (*task_func_t)(void);

void preempt_scheduler_init(void);
```

```
uint8_t preempt_create_task(task_func_t task, uint32_t period_ticks);
void preempt_start(void);

#endif
```

3.2. Реализация (упрощённая, без переключения контекста на Cortex-M – сложна для короткой ЛР). Вместо этого можно использовать готовый пример с setjmp/longjmp для имитации переключения контекста в пользовательском режиме.

Предложим студентам использовать подход с флагами готовности и переключением в суперцикле по таймеру:

```
typedef struct {
    void (*func)(void);
    uint32_t period;
    uint32_t last_run;
    uint8_t enabled;
} task_t;

static task_t tasks[MAX_TASKS];
static uint8_t task_count;

void scheduler_tick(void) {  // из прерывания SysTick
    // Системный тик, ничего не переключает, только обновляет время
}

void scheduler_run(void) {
    while (1) {
        uint32_t now = SysTick_GetTick();
        for (int i = 0; i < task_count; i++) {
            if (tasks[i].enabled && (now - tasks[i].last_run) >= tasks[i].period) {
```

```
                tasks[i].last_run = now;
                tasks[i].func();    // функция должна быть короткой или быть конечным автоматом
            }
        }
    }
}
```

Это не вытесняющий, а скорее планировщик с временными слотами (time-triggered). Для истинного вытеснения требуется сложный механизм сохранения контекста.

Для целей ЛР13 допустим реализацию "вытеснения" на уровне эмуляции: каждая задача вызывается из таймерного прерывания с фиксированным квантом, но это опасно (стек). Лучше использовать библиотеку Protothreads или простую кооперативную многозадачность.

Так как предыдущие ЛР используют конечные автоматы, логично предложить студентам реализовать диспетчер, который циклически вызывает обработчики состояний (как в ЛР6). Это уже является моделью кооперативной многозадачности.

4. **Планировщик на основе конечных автоматов (кооперативный)**

Используем подход из ЛР6: каждая задача (модуль) имеет функцию void taskX_tick(void), которая вызывается из суперцикла или из прерывания таймера. Это наиболее практичный подход для bare-metal.

```
typedef struct {
    void (*init)(void);
    void (*tick)(void);
} module_t;

module_t modules[] = {
```

```
    {led_init, led_tick},
    {uart_init, uart_tick},
    {adc_init, adc_tick},
    {NULL, NULL}
};

void scheduler_run(void) {
    for (int i = 0; modules[i].init; i++) {
        modules[i].init();
    }
    while (1) {
        for (int i = 0; modules[i].tick; i++) {
            modules[i].tick();
        }
    }
}
```

Этот подход уже использовался в ЛР6 (драйвер светодиода). Можно развить его до полноценного кооперативного планировщика.

5. **Сравнение моделей**

5.1. Создайте три варианта программы с одинаковым функционалом (мигание светодиодом + отправка отчёта по UART + длительная имитация обработки данных). Замерьте время выполнения каждого цикла (через переключение вывода) и отклик на нажатие кнопки.

5.2. Сделайте выводы: в каком случае задержки минимальны.

6. **Реализация событий и флагов между задачами**

6.1. Добавьте простую систему событий: глобальный массив флагов. Одна задача устанавливает флаг, другая проверяет и сбрасывает. Используйте атомарные операции (запрет прерываний для защиты).

Пример выполнения задания

Задача: Реализовать кооперативный планировщик для трёх задач:

1. Мигание светодиодом с периодом 500 мс (использовать неблокирующую задержку).
2. Чтение АЦП каждые 100 мс и усреднение.
3. Отправка среднего значения по UART раз в секунду.

Решение с использованием конечных автоматов и таймера:

```
#include "timers.h"
#include "led_driver.h"
#include "adc_driver.h"
#include "uart_driver.h"

static uint32_t adc_sum = 0;
static uint32_t adc_count = 0;
static uint16_t last_avg = 0;

void task_led_tick(void) {
    static uint32_t last = 0;
    if ((SysTick_GetTick() - last) >= 500) {
        led_toggle();
        last = SysTick_GetTick();
    }
}

void task_adc_tick(void) {
    static uint32_t last = 0;
```

```
    if ((SysTick_GetTick() - last) >= 100) {
        uint16_t val = adc_read_single(0);
        adc_sum += val;
        adc_count++;
        last = SysTick_GetTick();
    }
}

void task_uart_tick(void) {
    static uint32_t last = 0;
    if ((SysTick_GetTick() - last) >= 1000) {
        if (adc_count > 0) {
            last_avg = adc_sum / adc_count;
            uart_printf("ADC average: %u\r\n", last_avg);
            adc_sum = 0;
            adc_count = 0;
        }
        last = SysTick_GetTick();
    }
}

int main(void) {
    SysTick_Init(1000);
    led_init();
    adc_init(ADC_MODE_SINGLE, 0, 84);
    uart_init(115200);

    while (1) {
        task_led_tick();
        task_adc_tick();
```

```
            task_uart_tick();
            // Можно добавить вызов __WFI() для снижения энергопотребления
        }
    }
```

Результат: Все задачи выполняются без блокировок. Даже если одна задача делала бы долгие операции, их нужно разбивать на шаги (конечный автомат). В данном примере все операции быстрые.

Индивидуальные задания

Вариант 1. Планировщик с динамическим добавлением задач

Реализуйте функции add_task(task_func, period_ms) и remove_task(task_id). Задачи могут добавляться и удаляться во время работы (например, по команде UART).

Вариант 2. Приоритеты в кооперативном планировщике

Добавьте приоритеты задачам (0 – низкий, 1 – высокий). Планировщик сначала выполняет все задачи с высоким приоритетом, затем с низким. Оцените влияние на отзывчивость.

Вариант 3. Вытесняющий планировщик на базе таймера (псевдо-Preemptive)

Используйте SysTick для вызова специальной функции планировщика, которая сохраняет контекст (просто глобальные переменные) и переключает задачи. Для упрощения – задачи не имеют собственного стека, а являются конечными автоматами, но квант прерывает их в любом месте. Обработайте защиту разделяемых данных.

Вариант 4. Семафоры для синхронизации задач

Реализуйте примитив "семафор" с функциями sem_wait() и sem_signal(). Две задачи обмениваются данными через общий буфер, синхронизируясь семафорами.

Вариант 5. Round Robin с квантом времени (кооперативный с таймерным переключением)

Используйте таймер, который через фиксированный интервал (например, 10 мс) принудительно переключает задачи (вызывает scheduler_yield()). Реализуйте сохранение контекста через setjmp/longjmp.

Вариант 6. Ограничение времени выполнения задачи (watchdog)

Если задача выполняется дольше заданного времени (например, 50 мс), планировщик принудительно прерывает её и генерирует ошибку (включает красный светодиод).

Вариант 7. Планировщик с очередью отложенных вызовов (callback queue)

Реализуйте очередь функций, которые должны быть выполнены в главном цикле. Прерывания могут добавлять функции в очередь. Планировщик выполняет их в порядке поступления.

Вариант 8. Двухуровневый планировщик (прерывания + суперцикл)

Прерывания UART помещают принятые байты в очередь. В суперцикле читается очередь и обрабатываются команды. Совместите с кооперативным планировщиком.

Вариант 9. Измерение загрузки CPU

Каждый раз перед входом в цикл задач включайте светодиод (или устанавливайте вывод), после цикла – выключайте. По коэффициенту заполнения меандра оцените загрузку CPU. Сравните разные модели планирования.

Вариант 10. Планировщик с поддержкой "спящих" задач (delay)

Реализуйте функцию task_delay(ms), которая приостанавливает выполнение текущей задачи на указанное время. Планировщик переключается на следующую задачу, а по истечении времени возвращается к задаче.

Вариант 11. Смешанная модель: кооперативная + прерывания

Прерывания (например, от кнопки) устанавливают флаг, который проверяется в главном цикле. При этом основной цикл содержит кооперативный планировщик задач. Реализуйте и оцените отклик.

Вариант 12. Планировщик для одного стека (без сохранения контекста)

Все задачи используют один стек. Переключение происходит только в точках yield, где задачи сохраняют свои локальные переменные в статических структурах. Реализуйте три задачи, переключающиеся через yield. Используйте goto или switch для эмуляции (техника Protothreads).

Вариант 13. Визуализация работы планировщика через UART

Выводите сообщения вида "Task1 start", "Task1 yield", "Task2 start" и т.д. с временными метками. Постройте график переключений.

Вариант 14. Планировщик с динамическим изменением кванта времени

Реализуйте возможность изменять квант времени для вытесняющего планировщика во время работы (команда set_quantum 10). Измерьте влияние на поведение системы.

Вариант 15. Интеграция с реальной RTOS (FreeRTOS)

Портировать простое приложение (мигание светодиодом, чтение АЦП) на FreeRTOS. Сравните сложность разработки и накладные расходы с самодельным планировщиком.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.

2. Цель и задачи лабораторной работы.

3. Теоретическую часть (модели многозадачности: суперцикл, кооперативная, вытесняющая; Round Robin, контекст, квантование).

4. Листинги реализации суперцикла, кооперативного планировщика и (опционально) вытесняющего.

5. Сравнительный анализ трех подходов (таблица: преимущества, недостатки, загрузка CPU, максимальная задержка).

6. Результаты выполнения индивидуального задания.

7. Выводы о применимости каждой модели в зависимости от требований проекта.

8. Ответы на контрольные вопросы.

Контрольные вопросы

1. В чём отличие суперцикла от кооперативной многозадачности?

2. Что такое Round Robin планирование? В каких случаях оно эффективно?

3. Какие проблемы возникают при использовании суперцикла с длительными задачами?

4. Что такое "кооперативная многозадачность" и почему она называется "кооперативной"?

5. Как реализовать передачу управления от задачи к задаче без использования прерываний?

6. В чём заключается основное преимущество вытесняющей многозадачности перед кооперативной?

7. Какие сложности возникают при реализации вытесняющего планировщика на Cortex-M (сохранение контекста, стек задач)?

8. Что такое "квант времени" и как он влияет на отзывчивость системы?

9. Как защитить разделяемые данные при вытесняющей многозадачности?

10. Как оценить загрузку процессора в самодельном планировщике?

11. Каким образом в кооперативной системе можно реализовать неблокирующие задержки?

12. Что такое "конечный автомат" и как он помогает в кооперативной многозадачности?

13. Какую роль играет системный таймер (SysTick) в планировщике реального времени?

14. В чем разница между планировщиком на основе флагов готовности и на основе квантования?

15. Как организуется обмен сообщениями между задачами (очереди, почтовые ящики)?

16. Почему в bare-metal системах часто используют комбинацию прерываний + суперцикл (background/foreground)?

17. Какие накладные расходы (память, время) у простого кооперативного планировщика по сравнению с RTOS?

18. Как проверить, что переключение задач происходит корректно (нет переполнения стека)?

19. Что произойдёт, если в вытесняющем планировщике задача не уступает управление, но таймер её прерывает? Какие риски?

20. Можно ли использовать setjmp/longjmp для реализации кооперативной многозадачности? Какие ограничения?

Лабораторная работа №14

Работа с энергонезависимой памятью (Flash-эмуляция EEPROM)

Цель работы: освоение методов записи и стирания внутренней Flash-памяти микроконтроллера, реализация программной эмуляции EEPROM для хранения пользовательских настроек, применение алгоритмов wear-leveling для увеличения срока службы памяти.

Задачи:

- Изучить архитектуру Flash-памяти микроконтроллера STM32F4: организация страниц, ограничения на число циклов стирания/записи (обычно 10 000–100 000), особенности записи (только сброс битов из 1 в 0).
- Освоить операции разблокировки, стирания страницы и программирования (побайтовой/полусловной/словной записи) через регистры Flash.
- Реализовать низкоуровневый драйвер для работы с Flash: стирание страницы, запись 32-битного слова, запись буфера.
- Разработать модуль эмуляции EEPROM с использованием двух страниц Flash (техника «двойного буфера» или «с постоянным смещением») с поддержкой wear-leveling.
- Реализовать функции сохранения и восстановления структуры параметров (например, режим светодиода, частота, калибровка АЦП).
- Интегрировать эмуляцию EEPROM в командный интерпретатор (CLI) для просмотра и изменения настроек.
- Проанализировать ограничение числа циклов записи и оценить ресурс работы эмулируемой EEPROM.

Теоретическая часть

Flash-память микроконтроллера

Встроенная Flash-память STM32F4 используется для хранения кода программы (обычно ., .rodata) и может частично отводиться для хранения пользовательских данных (эмуляция EEPROM). Основные характеристики:

- **Организация**: разбита на секторы (sectors) для STM32F4: первые 4 сектора по 16 Кбайт, затем 1 сектор 64 Кбайт, и 7 секторов по 128 Кбайт.
- **Стирание**: только целыми секторами. Нельзя стереть один байт.
- **Запись**: можно изменять биты только с 1 на 0. Для записи нового значения нужно предварительно стереть сектор (установить все биты в 1).
- **Ресурс**: 10 000 – 100 000 циклов стирания/записи на сектор.

Регистры управления Flash (STM32F4xx):

- FLASH_ACR – управление кэшем и задержками (латенси)
- FLASH_KEYR – регистр ключа для разблокировки (запись KEY1=0x45670123, KEY2=0xCDEF89AB)
- FLASH_OPTKEYR – для разблокировки опционных байт
- FLASH_SR – статус (BSY, EOP, WRPERR, PGAERR и др.)
- FLASH_CR – управление (биты: PG, SER, SNB, START, LOCK)

Типичный алгоритм записи слова во Flash:

1. Проверить, что адрес выровнен по границе слова (4 байта) и находится в пределах пользовательской области.
2. Разблокировать Flash (записать ключи в FLASH_KEYR).
3. Дождаться, пока BSY в FLASH_SR не станет 0.
4. Установить бит PG (programming) в FLASH_CR.
5. Записать слово по указанному адресу (через указатель).
6. Дождаться сброса BSY.
7. Сбросить бит PG и заблокировать Flash (установить LOCK).

Стирание сектора:

1. Разблокировать Flash.
2. Установить бит SER и выбрать номер сектора (SNB) в FLASH_CR.
3. Установить бит START.
4. Дождаться BSY=0.
5. Сбросить SER и установить LOCK.

Эмуляция EEPROM (Flash Emulation EEPROM)

Поскольку стирание Flash происходит большими блоками (секторами), а запись нужна частых небольших обновлений, применяются специальные алгоритмы, имитирующие произвольный доступ к энергонезависимой памяти.

Метод 1: Одна страница с переменным смещением (cyclic buffer in Flash)

- В одной странице хранятся записи (тег + данные). При записи новой версии переменной добавляется новая запись в конец, старая помечается как недействительная. При заполнении страницы выполняется сжатие (перенос активных записей в новую страницу) и стирание старой.

Метод 2: Две страницы (A и B) – double page emulation

- Активная страница используется для хранения текущих данных. При обновлении переменной изменения записываются в активную страницу, а когда она заполняется – активность переключается на другую страницу, в которую копируются только актуальные значения, затем старая стирается.

Метод 3: Прямое отображение структуры в фиксированном секторе (простейший)

- Выделяется один сектор Flash (например, сектор 4 размером 64 Кбайт). Структура с параметрами сохраняется целиком при каждом изменении. При каждом сохранении стирается сектор и записывается новая копия структуры. Ресурс ограничен (10 000 изменений).

Wear-leveling (выравнивание износа)

Техника распределения циклов стирания/записи по всей доступной области памяти, чтобы не «изнашивать» одни и те же физические ячейки. В контексте эмуляции EEPROM означает, что физические адреса записи варьируются, а логические адреса переменных отображаются через таблицу или алгоритмически.

Простая реализация wear-leveling для двух страниц:

- Всего выделено N страниц (обычно 2). Запись всегда происходит в следующую свободную ячейку текущей активной страницы. Когда страница заполняется, актуальные данные переносятся в следующую страницу, и индекс активной страницы меняется. Таким образом каждая страница стирается не при каждом обновлении, а через N обновлений, где N – количество записей, помещающихся на странице.

Методика выполнения работы

1. **Изучение структуры Flash-памяти и низкоуровневого доступа**

   1.1. Создайте каталог lab14 и файл flash_driver.h:

```
#ifndef FLASH_DRIVER_H
#define FLASH_DRIVER_H

#include <stdint.h>
#include <stdbool.h>

// Адрес начала пользовательской области Flash (например, сектор 4, размер 64 КБ)
// Для STM32F407VG: сектор 4 начинается с 0x08010000 (после первых 64 КБ)
#define USER_FLASH_START_ADDR   0x08010000
#define USER_FLASH_SECTOR       4
```

```
#define USER_FLASH_SIZE         0x10000  // 64 КБ

// Инициализация (проверка доступа)
void flash_init(void);

// Стереть сектор пользовательской Flash
bool flash_erase_sector(uint32_t sector_num);

// Записать 32-битное слово по адресу (адрес должен быть выровнен)
bool flash_program_word(uint32_t address, uint32_t data);

// Записать буфер данных (произвольной длины, кратной 4)
bool flash_program_buffer(uint32_t address, uint32_t* data, uint32_t word_count);

// Чтение слова из Flash (просто обращение по адресу)
static inline uint32_t flash_read_word(uint32_t address) {
   return *(volatile uint32_t*)address;
}

#endif
```

1.2. Создайте файл flash_driver.c (на основе регистров STM32F4):

```
#include "flash_driver.h"
#include "stm32f4xx.h"

void flash_init(void) {
   // Разблокировка кэша и ускорение (опционально)
   // Здесь просто убеждаемся, что Flash готова
}
```

```
bool flash_erase_sector(uint32_t sector_num) {
    // Проверка параметров (sector_num 0..11)
    if (sector_num > 11) return false;

    // Разблокировка
    FLASH->KEYR = 0x45670123;
    FLASH->KEYR = 0xCDEF89AB;

    // Ожидание готовности
    while (FLASH->SR & FLASH_SR_BSY);

    // Настройка стирания сектора
    FLASH->CR |= FLASH_CR_SER;
    FLASH->CR &= ~FLASH_CR_SNB_Msk;
    FLASH->CR |= (sector_num << FLASH_CR_SNB_Pos);
    FLASH->CR |= FLASH_CR_START;

    // Ожидание завершения
    while (FLASH->SR & FLASH_SR_BSY);

    // Проверка ошибок
    uint32_t sr = FLASH->SR;
    if (sr & (FLASH_SR_WRPERR | FLASH_SR_PGAERR |
FLASH_SR_PGPERR)) {
        FLASH->SR = sr;  // сброс ошибок
        FLASH->CR &= ~FLASH_CR_SER;
        FLASH->CR |= FLASH_CR_LOCK;
        return false;
    }
```

```
    FLASH->CR &= ~FLASH_CR_SER;
    FLASH->CR |= FLASH_CR_LOCK;
    return true;
}

bool flash_program_word(uint32_t address, uint32_t data) {
    if (address & 0x3) return false;  // выравнивание
    if (address < USER_FLASH_START_ADDR ||
        address >= USER_FLASH_START_ADDR + USER_FLASH_SIZE)
return false;

    // Разблокировка
    FLASH->KEYR = 0x45670123;
    FLASH->KEYR = 0xCDEF89AB;

    while (FLASH->SR & FLASH_SR_BSY);

    FLASH->CR |= FLASH_CR_PG;

    *(volatile uint32_t*)address = data;

    while (FLASH->SR & FLASH_SR_BSY);

    FLASH->CR &= ~FLASH_CR_PG;

    // Проверка ошибок
    if (FLASH->SR & (FLASH_SR_WRPERR | FLASH_SR_PGAERR |
FLASH_SR_PGPERR)) {
        FLASH->SR = FLASH->SR;
```

```
    FLASH->CR |= FLASH_CR_LOCK;
    return false;
  }

  FLASH->CR |= FLASH_CR_LOCK;
  return true;
}

bool flash_program_buffer(uint32_t address, uint32_t* data, uint32_t word_count) {
  for (uint32_t i = 0; i < word_count; i++) {
    if (!flash_program_word(address + i*4, data[i])) return false;
  }
  return true;
}
```

2. **Реализация эмуляции EEPROM с wear-leveling (две страницы)**

2.1. Создайте файл eeprom_emul.h:

```
#ifndef EEPROM_EMUL_H
#define EEPROM_EMUL_H

#include <stdint.h>
#include <stdbool.h>

// Определяем структуру пользовательских настроек
typedef struct {
  uint32_t magic;          // сигнатура для проверки валидности (0xDEADBEEF)
  uint16_t led_mode;     // 0=off,1=on,2=blink,3=fade
  uint16_t blink_period; // период мигания в мс
```

```
    uint16_t adc_channel;
    uint16_t threshold;
    uint32_t calibration;
    uint32_t crc;        // контрольная сумма для проверки целостности
} user_config_t;

// Инициализация эмуляции EEPROM
void eeprom_emul_init(void);

// Загрузить конфигурацию из Flash в RAM (с проверкой CRC)
bool eeprom_emul_load_config(user_config_t* config);

// Сохранить конфигурацию (с пересчётом CRC) с wear-leveling
bool eeprom_emul_save_config(const user_config_t* config);

// Восстановить заводские настройки (сброс)
void eeprom_emul_reset_defaults(user_config_t* config);

#endif
```

2.2. Создайте файл eeprom_emul.c (реализация с двумя страницами):

```
#include "eeprom_emul.h"
#include "flash_driver.h"
#include <string.h>
#include <stddef.h>

#define PAGE1_ADDR  USER_FLASH_START_ADDR
#define PAGE2_ADDR  (USER_FLASH_START_ADDR + USER_FLASH_SIZE / 2)
#define PAGE_SIZE   (USER_FLASH_SIZE / 2)
```

```
// Заголовок страницы для реализации двухстраничного алгоритма
typedef struct {
    uint32_t active;      // 0x5A5A5A5A - страница активна
    uint32_t version;     // версия данных (монотонно возрастает)
} page_header_t;

static const user_config_t default_config = {
    .magic = 0xDEADBEEF,
    .led_mode = 1,
    .blink_period = 500,
    .adc_channel = 0,
    .threshold = 2048,
    .calibration = 0,
    .crc = 0
};

// Простая CRC-32 (можно использовать аппаратный или табличный)
static uint32_t calculate_crc(const uint8_t* data, uint32_t len) {
    uint32_t crc = 0xFFFFFFFF;
    for (uint32_t i = 0; i < len; i++) {
        crc ^= data[i];
        for (int j = 0; j < 8; j++) {
            crc = (crc >> 1) ^ ((crc & 1) ? 0xEDB88320 : 0);
        }
    }
    return ~crc;
}

// Подготовка конфига перед сохранением
```

```
static void prepare_config_for_save(user_config_t* config) {
    config->magic = 0xDEADBEEF;
    config->crc = 0;
    uint32_t crc = calculate_crc((uint8_t*)config, sizeof(user_config_t) - 4); // 
без поля CRC
    config->crc = crc;
}

// Проверка валидности конфига
static bool is_config_valid(const user_config_t* config) {
    if (config->magic != 0xDEADBEEF) return false;
    uint32_t saved_crc = config->crc;
    user_config_t copy = *config;
    copy.crc = 0;
    uint32_t crc = calculate_crc((uint8_t*)&copy, sizeof(user_config_t) - 4);
    return crc == saved_crc;
}

// Определение активной страницы (возвращает адрес и версию)
static uint32_t get_active_page(uint32_t* version) {
    page_header_t h1, h2;
    h1.active = flash_read_word(PAGE1_ADDR);
    h1.version = flash_read_word(PAGE1_ADDR + 4);
    h2.active = flash_read_word(PAGE2_ADDR);
    h2.version = flash_read_word(PAGE2_ADDR + 4);

    if (h1.active == 0x5A5A5A5A && h2.active == 0x5A5A5A5A) {
        *version = (h1.version > h2.version) ? h1.version : h2.version;
        return (h1.version > h2.version) ? PAGE1_ADDR : PAGE2_ADDR;
    } else if (h1.active == 0x5A5A5A5A) {
```

```
        *version = h1.version;
        return PAGE1_ADDR;
    } else if (h2.active == 0x5A5A5A5A) {
        *version = h2.version;
        return PAGE2_ADDR;
    }
    *version = 0;
    return 0; // нет активной страницы (первый запуск)
}

void eeprom_emul_init(void) {
    // При инициализации можно проверить, есть ли валидные данные
    uint32_t version;
    uint32_t page_addr = get_active_page(&version);
    if (page_addr == 0) {
        // Нет активной страницы, записываем дефолтную конфигурацию
        user_config_t config = default_config;
        prepare_config_for_save(&config);
        eeprom_emul_save_config(&config);
    }
}

bool eeprom_emul_load_config(user_config_t* config) {
    uint32_t version;
    uint32_t page_addr = get_active_page(&version);
    if (page_addr == 0) return false;

    // Конфигурация находится по смещению после заголовка
    const user_config_t* stored = (const user_config_t*)(page_addr +
sizeof(page_header_t));
```

```
    if (!is_config_valid(stored)) return false;

    memcpy(config, stored, sizeof(user_config_t));
    return true;
}

bool eeprom_emul_save_config(const user_config_t* config) {
    user_config_t safe_config = *config;
    prepare_config_for_save(&safe_config);

    uint32_t current_version;
    uint32_t active_page = get_active_page(&current_version);
    uint32_t other_page = (active_page == PAGE1_ADDR) ? PAGE2_ADDR
: PAGE1_ADDR;

    // Стираем другую страницу
    uint32_t sector_num = (other_page == PAGE1_ADDR) ?
USER_FLASH_SECTOR : (USER_FLASH_SECTOR + 1);
    if (!flash_erase_sector(sector_num)) return false;

    // Записываем заголовок на другую страницу
    page_header_t new_header = { .active = 0x5A5A5A5A, .version =
current_version + 1 };
    if (!flash_program_word(other_page, new_header.active)) return false;
    if (!flash_program_word(other_page + 4, new_header.version)) return
false;

    // Записываем конфигурацию
    uint32_t data_addr = other_page + sizeof(page_header_t);
    // Запись побайтово (32-битными словами) структуры
```

```
    const uint32_t* words = (const uint32_t*)&safe_config;
    uint32_t word_count = sizeof(user_config_t) / 4;
    if (!flash_program_buffer(data_addr, (uint32_t*)words, word_count)) return false;

    // Деактивируем старую страницу (записываем 0 в поле active)
    if (active_page != 0) {
      flash_erase_sector( (active_page == PAGE1_ADDR) ? USER_FLASH_SECTOR : (USER_FLASH_SECTOR + 1) );
    }

    return true;
  }

  void eeprom_emul_reset_defaults(user_config_t* config) {
    *config = default_config;
    prepare_config_for_save(config);
    eeprom_emul_save_config(config);
  }
```

3. **Интеграция с CLI (командный интерпретатор)**

   3.1. Добавьте команды в cli.c:

```
static void cmd_config_save(int argc, char** argv);
static void cmd_config_load(int argc, char** argv);
static void cmd_config_reset(int argc, char** argv);
static void cmd_config_show(int argc, char** argv);

// Добавить в таблицу commands[]
{"cfg_save", cmd_config_save, "cfg_save - save current config to Flash"},
{"cfg_load", cmd_config_load, "cfg_load - load config from Flash"},
```

```
{"cfg_reset", cmd_config_reset, "cfg_reset - restore factory defaults"},
{"cfg_show", cmd_config_show, "cfg_show - display current config"},

static user_config_t current_config;

static void cmd_config_show(int argc, char** argv) {
    uart_printf("LED mode: %u\r\n", current_config.led_mode);
    uart_printf("Blink period: %u ms\r\n", current_config.blink_period);
    uart_printf("ADC channel: %u\r\n", current_config.adc_channel);
    uart_printf("Threshold: %u\r\n", current_config.threshold);
    uart_printf("Calibration: %lu\r\n", current_config.calibration);
}

static void cmd_config_save(int argc, char** argv) {
    if (eeprom_emul_save_config(&current_config)) {
        uart_puts("Config saved to Flash\r\n");
    } else {
        uart_puts("Save failed!\r\n");
    }
}

static void cmd_config_load(int argc, char** argv) {
    if (eeprom_emul_load_config(&current_config)) {
        uart_puts("Config loaded from Flash\r\n");
        cmd_config_show(0, NULL);
    } else {
        uart_puts("No valid config, loading defaults\r\n");
        eeprom_emul_reset_defaults(&current_config);
    }
}
```

```
static void cmd_config_reset(int argc, char** argv) {
    eeprom_emul_reset_defaults(&current_config);
    uart_puts("Factory defaults restored\r\n");
}
```

4. **Тестирование ресурса износа (wear-leveling)**

4.1. Напишите тестовую функцию, которая 1000 раз сохраняет конфигурацию с небольшими изменениями. Используйте светодиод для индикации прогресса. После теста прочитайте, сколько раз стёрся каждый сектор (можно вывести через отладчик). Теоретически при использовании двух страниц каждая стирается в N раз реже, где N – количество записей, помещающихся на странице (в нашем случае 1 запись на страницу, но можно хранить несколько копий).

4.2. Для демонстрации можно увеличить количество записей на одной странице до 10, используя дополнительную индексацию.

Пример выполнения задания

Задача: Сохранять пользовательские настройки (режим светодиода и период мигания) во Flash-памяти так, чтобы выдерживалось 100 000 циклов изменения настроек.

Решение с использованием двухстраничного алгоритма (каждая страница хранит одну копию структуры):

После каждого сохранения активная страница переключается, а старая стирается. Таким образом каждая страница стирается через одно сохранение. Ресурс ограничен ресурсом одной страницы (10 000–100 000). Для увеличения срока можно использовать буфер на 10 записей на страницу.

Результат работы CLI:

```
> cfg_show
LED mode: 1
```

Blink period: 500 ms

ADC channel: 0

Threshold: 2048

Calibration: 0

> cfg_save

Config saved to Flash

> cfg_reset

Factory defaults restored

> cfg_load

Config loaded from Flash

LED mode: 0

...

Индивидуальные задания

Вариант 1. Расширение объёма эмулируемой EEPROM

Вместо одной структуры реализуйте хранение массива из 256 байт (128 переменных по 2 байта). Разработайте простую файловую систему: каждая запись имеет идентификатор (тег) и данные. Реализуйте функции eeprom_write(addr, data) и eeprom_read(addr).

Вариант 2. Более эффективный wear-leveling (циклический буфер в одном секторе)

Используйте один сектор Flash как циклический буфер. Каждая запись содержит тег, данные и статус. При заполнении буфера сжимайте активные записи в начало и продолжайте. Рассчитайте, во сколько раз увеличивается ресурс.

Вариант 3. Проверка целостности с двойной копией

Сохраняйте две копии конфигурации в разных секторах. При загрузке выбирайте копию с совпадающей CRC. Если обе повреждены – сброс на defaults.

Вариант 4. Атомарное обновление конфигурации

Реализуйте механизм, при котором при сохранении сначала записывается "флаг транзакции", затем данные, затем флаг подтверждения. При сбое питания между шагами система восстановит предыдущую конфигурацию.

Вариант 5. Поддержка разных моделей микроконтроллеров

Сделайте автораспознавание объёма Flash и размера секторов через регистр FLASH_SIZE и DBGMCU. Адаптируйте код под разные семейства (STM32F1, F4, H7).

Вариант 6. Защита от несанкционированного доступа (RDP)

При первом запуске программа устанавливает уровень защиты чтения Flash (RDP Level 1). Покажите, что прочитать прошивку через отладчик становится невозможно. Обработайте последствия (отключение отладчика).

Вариант 7. Сохранение логов в энергонезависимую память

Используйте оставшуюся Flash для хранения циклического буфера логов (время, код события). При переполнении перезаписывайте самые старые записи.

Вариант 8. Измерение времени записи

Используйте DWT для замера времени стирания сектора (около 100–200 мс) и записи слова. Сравните с теоретическими значениями из даташита.

Вариант 9. Эмуляция EEPROM через внешнюю SPI Flash

Используйте внешнюю Flash-память (например, W25Q64) с алгоритмом wear-leveling. Сравните скорость и ресурс с внутренней Flash.

Вариант 10. Команда для дампа Flash

Реализуйте команду dump_flash <addr> <len>, которая выводит содержимое Flash в шестнадцатеричном виде (с защитой от чтения за пределами разрешённой области).

Вариант 11. Утилита для программирования калибровочных данных через UART

Реализуйте протокол для загрузки калибровочной таблицы (например, для коррекции АЦП) из ПК и сохранения в Flash. Используйте XMODEM или просто текстовые строки.

Вариант 12. Анти-деградация с учётом числа циклов

Ведёте счётчик сохранений в отдельной ячейке. Когда приближаетесь к пределу ресурса (например, 5000 циклов), меняйте алгоритм на более щадящий или уведомляйте пользователя.

Вариант 13. Энергонезависимый счётчик моточасов

Счётчик (32-битный) увеличивается раз в минуту (по RTC) и сохраняется в Flash. Реализуйте алгоритм с минимальным износом (например, накопление счётчика в RAM и сохранение раз в час).

Вариант 14. Периодическое сохранение по таймеру

Настройте таймер (например, раз в 10 минут), который сохраняет текущую конфигурацию автоматически (без команды пользователя). Это снижает риск потери данных при сбое питания.

Вариант 15. Конфигурация с резервными копиями (три сектора)

Используйте три сектора: последняя успешная конфигурация, предыдущая, и "рабочая". При загрузке выбирается валидная с наибольшей версией. Реализуйте откат к предыдущей при ошибке.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.

2. Цель и задачи лабораторной работы.

3. Теоретическую часть (организация Flash, операции стирания/записи, ограничения, wear-leveling, эмуляция EEPROM).

4. Листинги: flash_driver.h, flash_driver.c, eeprom_emul.h, eeprom_emul.c, изменённого cli.c и main.c.

5. Таблицу с результатами тестирования ресурса (количество сохранений до переполнения/ошибки) и расчётное увеличение срока службы.

6. Результаты выполнения индивидуального задания.

7. Выводы о применимости эмуляции EEPROM, преимуществах и недостатках по сравнению с внешней EEPROM.

8. Ответы на контрольные вопросы.

Контрольные вопросы

1. Почему нельзя произвольно записывать данные во Flash, как в RAM? Какие операции обязательны перед записью?

2. Каковы типичные ресурс (количество циклов стирания/записи) Flash-памяти для STM32?

3. Что такое «выравнивание износа» (wear-leveling) и для чего оно применяется во Flash-эмуляции EEPROM?

4. Почему для эмуляции EEPROM используют минимум две страницы (сектора) Flash?

5. Как реализована атомарность операции сохранения? Что произойдёт при потере питания во время сохранения?

6. Как проверить, что сохранённая конфигурация не повреждена (целостность данных)?

7. Для чего нужна контрольная сумма (CRC) в структуре параметров?

8. Какую роль играет поле magic (сигнатура) в конфигурации?

9. Какой алгоритм используется для определения активной страницы (с более свежими данными)?

10. Какие ограничения накладывает выравнивание адреса при записи во Flash?

11. Какова максимальная глубина программирования (количество битов, которые можно сбросить без стирания)?

12. Чем отличается эмуляция EEPROM с wear-leveling от простого сохранения структуры в фиксированном секторе?

13. Как оценить, сколько сохранится конфигурация при заданной интенсивности обновлений (например, 1 раз в минуту)?

14. Можно ли во Flash хранить код программы и пользовательские данные одновременно? Какие меры предосторожности нужны?

15. Как разблокировать Flash для записи и заблокировать после?

16. Что произойдёт, если попытаться записать данные во Flash без предварительного стирания?

17. Как проверить успешность операции записи и стирания (биты ошибок)?

18. Как выровнять двухстраничный алгоритм для большей надёжности (например, резервная копия)?

19. Как организовать хранение калибровочных таблиц большого объёма (более 1 КБ) с возможностью обновления?

20. Какие преимущества и недостатки у внешней EEPROM (24LCxx) по сравнению с эмуляцией во внутренней Flash?

Лабораторная работа №15

Отладка через JTAG/SWD. Использование трассировки

Цель работы: освоение методики аппаратной отладки встраиваемых систем с использованием интерфейсов JTAG/SWD, приобретение навыков установки точек останова (breakpoints), просмотра и модификации переменных в реальном времени, анализа стека и регистров процессора, а также применение трассировки (trace) для анализа выполнения программы.

Задачи:

- Изучить интерфейсы отладки ARM: JTAG и SWD, их отличия и подключение.
- Настроить отладчик ST-Link (или J-Link) в среде разработки (STM32CubeIDE или OpenOCD + GDB).
- Освоить базовые операции отладки: запуск/остановка, шаг с заходом (Step Into), шаг с обходом (Step Over), выход из функции (Step Return).
- Научиться устанавливать аппаратные и программные точки останова, условные точки останова.
- Освоить просмотр переменных в окне Watch/Live Expressions, изменение переменных во время выполнения.
- Изучить просмотр памяти (Memory Viewer) и регистров (Registers).
- Освоить просмотр стека вызовов (Call Stack) и анализ локальных переменных.
- Реализовать простую трассировку с использованием полухостинга (semihosting) или SWO (Serial Wire Output) для вывода отладочных сообщений без блокировки и без UART.
- Проанализировать влияние точек останова на реальное время выполнения и использовать "printf отладку" через SWO.

Теоретическая часть

Интерфейсы отладки для ARM Cortex-M

Для подключения отладчика к микроконтроллеру используются два основных интерфейса:

- **JTAG (Joint Test Action Group)** – классический интерфейс, использующий 5 линий: TMS, TCK, TDI, TDO, nTRST. Обеспечивает полный доступ ко всем отладочным функциям, включает трассировку.
- **SWD (Serial Wire Debug)** – двухпроводной интерфейс (SWDIO, SWCLK), занимает меньше выводов, обеспечивает полную отладку (но без трассировки данных – SWV). Является предпочтительным на многих отладочных платах.

Программные и аппаратные точки останова:

- **Аппаратные** (Hardware Breakpoints) – ограниченное количество (обычно 4–8 для Cortex-M), могут быть установлены в любом месте памяти, включая Flash. Используют встроенные регистры сравнения.
- **Программные** (Software Breakpoints) – используют инструкцию BKPT (0xBEAB). Требуют записи в память (заменяют инструкцию), поэтому работают только в RAM. В Flash программные breakpoints недоступны (кроме специальных эмуляторов).

Стек вызовов (Call Stack):

Позволяет просмотреть цепочку вызовов функций, приведших к текущей точке останова. Для корректного отображения требуется отладочная информация (флаг -g при компиляции).

Semihosting (полухостинг):

Механизм, позволяющий программе на целевом устройстве вызывать функции хост-компьютера (например, printf, файловый ввод-вывод) через отладчик. Требует специального кода и поддержки со стороны отладчика. Удобен для отладки, но замедляет выполнение и может быть небезопасен в production.

SWO (Serial Wire Output) / ITM:

Режим трассировки через вывод SWO (на линии SWO/SWV). Позволяет выводить отладочные сообщения с минимальными накладными расходами (без блокировки, в реальном времени), используя модуль ITM (Instrumentation Trace Macrocell). Сообщения передаются с высокой скоростью и могут быть захвачены отладчиком.

Регистры процессора:

В Cortex-M основные регистры: R0–R12 (общего назначения), SP (R13, указатель стека), LR (R14, регистр возврата), PC (R15, счётчик команд), PSR (Program Status Register), MSP (Main Stack Pointer), PSP (Process Stack Pointer). При входе в отладчик можно просмотреть их текущие значения.

Методика выполнения работы

1. **Подготовка тестовой программы**

1.1. Создайте каталог lab15 и файл debug_test.c с программой для отладки:

```
#include <stdint.h>
#include <stdio.h>

// Простые функции для отладки
static uint32_t factorial(uint32_t n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

static volatile uint32_t global_counter = 0;

void delay_loop(volatile uint32_t count) {
    while (count--) {
        __asm volatile ("nop");
    }
```

```
}

void uart_putc(char c) {
    // Заглушка для демонстрации
    (void)c;
}

void uart_puts(const char* str) {
    while (*str) uart_putc(*str++);
}

int main(void) {
    uint32_t result = 0;
    for (int i = 0; i < 10; i++) {
        result = factorial(i);
        global_counter++;
        // Искусственный дефект: неверное условие
        if (i == 5) {
            result = result / 0;  // деление на ноль (вызовет HardFault)
        }
        delay_loop(10000);
    }

    uart_puts("Done!\n");
    while (1);
}
```

1.2. Скомпилируйте программу с отладочной информацией (-g):

arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -g -O0 debug_test.c -o debug_test.elf

2. **Настройка отладчика (ST-Link + OpenOCD / STM32CubeIDE)**

2.1. В среде STM32CubeIDE:

- Подключите отладочную плату (ST-Link) через USB.
- Убедитесь, что в настройках проекта выбран отладчик ST-Link (SWD).
- Установите конфигурацию отладки: подключение к цели, частоту SWD 4 МГц.

2.2. В OpenOCD (альтернатива):

openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg

Затем в другом терминале:

arm-none-eabi-gdb debug_test.elf

(gdb) target remote localhost:3333

(gdb) load

(gdb) monitor reset init

(gdb) continue

3. **Установка точек останова и пошаговое выполнение**

3.1. Установите точку останова на функции factorial. Запустите отладку и пошагово пройдите рекурсивные вызовы.

3.2. Установите точку останова на строке с делением на ноль (result = result / 0). Запустите программу, дойдите до этой строки, проанализируйте значения i и result.

3.3. Установите условную точку останова: останов при i == 3 на строке global_counter++. В GDB:

gdb

break debug_test.c:22 if i == 3

4. **Просмотр переменных и регистров**

4.1. В режиме остановки откройте окно Watch (в IDE) или используйте print в GDB:

gdb

print i

print result

print &global_counter

4.2. Просмотрите текущие значения регистров:

gdb

info registers

4.3. Измените значение переменной global_counter вручную:

gdb

set var global_counter = 100

5. **Анализ стека вызовов (Call Stack)**

5.1. Попадите в точку останова внутри factorial (например, при рекурсивном вызове). Откройте окно Call Stack. Посмотрите, в каком порядке вызывались функции.

5.2. Используйте backtrace (или bt) в GDB.

6. **Просмотр памяти (Memory Viewer)**

6.1. Добавьте глобальный массив и просмотрите его содержимое:

uint32_t test_array[10] = {1,2,3,4,5,6,7,8,9,10};

В отладчике найдите адрес test_array и просмотрите память в шестнадцатеричном виде.

7. **Использование полухостинга (Semihosting)**

7.1. Добавьте в проект код для поддержки полухостинга (обычно требуется библиотека -lrdimon и специальный вызов initialise_monitor_handles()). В STM32CubeIDE можно включить Semihosting в настройках.

7.2. Используйте printf для вывода отладочных сообщений. Заметьте, что выполнение замедляется, но сообщения появляются в консоли отладчика.

8. **Использование SWO / ITM для трассировки (без UART)**

8.1. Настройте модуль ITM (требуется для Cortex-M3/M4). Включите вывод через SWO. Пример кода для вывода символа через ITM:

```
#define ITM_Port8(n)    (*(volatile uint8_t *)(0xE0000000 + 4*n))
#define ITM_Port32(n)   (*(volatile uint32_t *)(0xE0000000 + 4*n))

void itm_putc(char c) {
   if ((ITM_Port32(0) & 1) == 0) return;
   ITM_Port8(0) = c;
}

void itm_puts(const char* str) {
   while (*str) itm_putc(*str++);
}
```

8.2. Настройте отладчик для захвата SWO (в STM32CubeIDE: Debug Configurations → Debugger → Enable SWO, частота 2 МГц). Выводите сообщения через itm_puts.

8.3. Сравните скорость вывода и влияние на выполнение программы между printf (semihosting) и ITM. Убедитесь, что ITM работает асинхронно.

9. **Поиск и исправление ошибки (HardFault)**

9.1. Из-за деления на ноль программа попадёт в обработчик HardFault. Установите точку останова в HardFault_Handler. При останове проанализируйте стек, регистры, чтобы определить место сбоя (например, значение LR покажет, из какой функции пришли).

9.2. Исправьте ошибку (защита от деления на ноль).

Пример выполнения задания

Задача: В программе реализован бесконечный цикл, который выполняет вычисления, но иногда зависает. Используя отладчик, найти место зависания и исправить.

Исходный код с ошибкой:

```
volatile uint32_t timeout = 0;

void wait_for_flag(void) {
    while (timeout == 0) {
        // ожидание
    }
}

int main(void) {
    wait_for_flag();  // зависает, так как timeout никогда не становится !=0
}
```

Отладка:

1. Запустить программу в отладчике, через 2 секунды нажать "Break".
2. Посмотреть Call Stack: видно, что выполнение находится в wait_for_flag.
3. Просмотреть значение timeout – равно 0.
4. Поставить точку останова на изменение timeout (watchpoint).

5. Обнаружить, что нигде нет записи в timeout.
6. Исправить, добавив установку timeout = 1 по прерыванию или таймеру.

Вывод: Отладка позволила быстро локализовать проблему.

Индивидуальные задания

Вариант 1. Анализ утечки стека

Создайте программу с глубокой рекурсией (например, 1000 вызовов factorial). Установите точку останова на HardFault или MemManage. С помощью отладчика определите момент переполнения стека, сделайте скриншот содержимого регистра SP и адресов возврата.

Вариант 2. Условные точки останова для поиска редкого бага

Программа совершает ошибку только при определённом значении глобальной переменной (например, state == 7). Установите условную точку останова в функции, которая обрабатывает состояние. Покажите, как отладчик останавливается только при нужном условии.

Вариант 3. Работа с памятью (запись по некорректному адресу)

Создайте программу, которая пишет в массив по индексу, превышающему размер. Используйте Hardware Watchpoint на адрес, следующий за массивом, чтобы обнаружить момент порчи памяти.

Вариант 4. Трассировка вызовов функций через SWO (ETM)

Настройте трассировку через SWO так, чтобы в консоли отладчика отображалась последовательность вызовов функций (инструментирование с ITM). Покажите что event viewer в IDE отображает временную диаграмму вызовов.

Вариант 5. Сравнение скорости работы с полухостингом и без него

Замерьте время выполнения 1000 итераций цикла с printf (semihosting) и без него. Включите/выключите semihosting через определения. Составьте таблицу (приблизительную).

Вариант 6. Отладка прерываний (таймер)

Напишите программу с таймерным прерыванием, которое обновляет глобальную переменную. Установите точку останова внутри обработчика прерывания. Исследуйте, как изменяется состояние регистров при входе и выходе из прерывания.

Вариант 7. Измерение времени выполнения функции (DTM/ITM Synchro)

Используйте DWT (Data Watchpoint and Trace) для измерения времени выполнения функции. Установите точки останова на входе и выходе, запишите значение CYCCNT. Сравните с ручным замером.

Вариант 8. Поиск неинициализированной переменной

Создайте программу, где локальная переменная не инициализируется (содержит мусор). При отладке в режиме Watch переменная отображается как некорректное значение. Покажите использование отладчика для выявления такой ошибки.

Вариант 9. Работа с отладчиком через командную строку (GDB)

Выполните все основные операции в GDB без IDE: загрузите elf, установите breakpoints, просмотрите переменные, выполните шаг. Сделайте скриншоты (или текстовую распечатку) сессии.

Вариант 10. Использование полухостинга (semihosting) для вывода в консоль

Реализуйте вывод отладочных сообщений через printf с полухостингом. Убедитесь, что сообщения появляются в консоли отладчика. Оцените задержки при большом количестве сообщений.

Вариант 11. Трассировка событий с помощью ITM (Event Viewer)

Настройте ITM для отправки 8-битных значений (например, номера состояния). В STM32CubeIDE (или в SystemView) захватите поток событий и постройте временную диаграмму.

Вариант 12. Точка останова на запись в память (Watchpoint)

Установите watchpoint на глобальную переменную, которая не должна изменяться. При её изменении отладчик останавливается. Проверьте, срабатывает ли watchpoint, и покажите, какая инструкция её модифицирует.

Вариант 13. Отладка оптимизированного кода (gcc -O2)

Скомпилируйте программу с оптимизацией -O2. Проследите, как изменяется отображение переменных (могут быть оптимизированы, регистры переиспользованы). Используйте ассемблерный листинг в отладчике для понимания.

Вариант 14. Использование серийной отладки через UART (printf)

При отсутствии SWO/ITM используйте UART для вывода отладочных сообщений (uart_printf). Сравните простоту и накладные расходы по сравнению с полухостингом.

Вариант 15. Эмуляция отладки в QEMU

Настройте эмулятор QEMU для Cortex-M3 (LPC1768 или LM3S6965). Подключитесь к нему через GDB (QEMU с опцией -s). Выполните отладку программы в эмуляторе, используя те же команды GDB.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (JTAG/SWD, hardware/software breakpoints, стеки, регистры).
4. Листинги тестовых программ (до и после исправления ошибок, если есть).
5. Скриншоты (или описание) сессии отладчика, демонстрирующие:
   - установленную точку останова и останов на ней;
   - окно просмотра переменных (с изменением значения);
   - стек вызовов;

- просмотр памяти;
- вывод через semihosting или SWO (консоль отладчика).

6. Результаты выполнения индивидуального задания.

7. Выводы (какие возможности отладчика наиболее полезны для поиска разных классов ошибок; сравнение аппаратных и программных breakpoints).

8. Ответы на контрольные вопросы.

Контрольные вопросы

1. Чем отличается JTAG от SWD? Какой интерфейс предпочтительнее для отладки и почему?

2. Какое максимальное количество аппаратных точек останова в Cortex-M4? От чего оно зависит?

3. Почему программные точки останова (BKPT) нельзя использовать во Flash? Где они могут быть установлены?

4. Какие данные можно получить при просмотре стека вызовов (Call Stack)?

5. Как можно изменить значение переменной во время выполнения через отладчик?

6. Для чего используется просмотр регистров процессора? Приведите пример, когда это необходимо.

7. Что такое полухостинг (semihosting)? Каковы его преимущества и недостатки?

8. Как работает SWO / ITM и чем он лучше полухостинга?

9. Какие ограничения накладывает оптимизация компилятора (-O2) на отладку?

10. Что такое "точка останова на запись в память" (watchpoint) и как её установить?

11. Как с помощью отладчика найти причину HardFault?

12. В чём разница между Step Into, Step Over и Step Return?

13. Что означает «загрузка» (load) программы через отладчик? Зачем она нужна?

14. Как можно просмотреть область памяти в реальном времени (Memory Live View)?

15. Как связаны отладочная информация (-g) и дизассемблерный листинг?

16. Можно ли использовать отладчик для отладки программы на реальном оборудовании с высоким энергопотреблением? Какие особенности?

17. Как выполнить отладку прерываний, не пропустив нужное событие?

18. Что показывает регистр LR (Link Register) при входе в обработчик прерывания?

19. Какие команды GDB используются для выполнения до следующей точки останова, пошагового выполнения?

20. Как можно в GDB просмотреть содержимое всех локальных переменных текущей функции?

Лабораторная работа №16

Разработка программного эмулятора периферии на ПК

Цель работы: освоение методов создания программного эмулятора периферийных устройств микроконтроллера на языке высокого уровня (Python/C++), разработка эмулятора GPIO, UART и таймера, позволяющего отлаживать логику встраиваемого приложения без физического оборудования, и интеграция эмулятора с реальным кодом (двойная компиляция: для микроконтроллера и для ПК).

Задачи:

- Изучить принципы построения программных эмуляторов (симуляторов) периферии: цикл выполнения, обработка прерываний, моделирование времени.
- Спроектировать и реализовать на Python (или C++) эмулятор GPIO (цифровые входы/выходы) с возможностью установки/сброса линий и чтения.
- Реализовать эмулятор UART с поддержкой приёма/передачи данных через виртуальный последовательный порт или сокет.
- Реализовать эмулятор таймера (таймер общего назначения) с генерацией прерываний и моделированием времени на основе системного времени ПК.
- Обеспечить возможность компиляции одного и того же кода приложения (C) как для реального микроконтроллера (ARM), так и для ПК (через заглушки и эмуляторы периферии).
- Сравнить поведение приложения в эмуляторе и на реальном железе, выявить возможные расхождения (тайминги, прерывания, параллелизм).
- Разработать простой визуальный интерфейс командной строки для управления эмулятором (переключение входов, просмотр логов).

Теоретическая часть

Понятие программного эмулятора (симулятора) периферии

Программный эмулятор – это программа, которая воспроизводит поведение аппаратных модулей микроконтроллера (GPIO, UART, таймеры и т.д.) на персональном компьютере. Эмулятор не выполняет машинный код целевого процессора (в отличие от полносистемного эмулятора типа QEMU), а вместо этого вызывает специально написанные функции-заглушки, которые имитируют работу периферии. Это позволяет отлаживать алгоритмы и логику приложения без реального железа, с возможностью автоматического тестирования.

Основные компоненты эмулятора:

- **Регистровая модель** – структуры данных, хранящие состояние каждого регистра периферии (MODER, ODR, IDR, SR, DR и др.).
- **Функции-обработчики** – при вызове из прошивки (например, GPIO_SetPin()) изменяют состояние эмулятора и, возможно, генерируют события.
- **Моделирование времени** – использование системного таймера ПК (например, time.time()) для имитации реальных временных интервалов.
- **Прерывания** – эмуляция прерываний путём вызова функций-обработчиков по истечении времени или по событиям от эмулируемой периферии.
- **Интерфейс управления** – возможность из внешней среды (терминал, сценарий) подавать входные сигналы (нажатия кнопок, данные в UART) и наблюдать выходные.

Разница между полносистемной эмуляцией и эмуляцией периферии:

Тип	QEMU / полносистемный	Эмуляция периферии на уровне API
Выполняемый код	машинный код ARM (или др.)	C-функции, скомпилированные для ПК
Скорость	низкая (интерпретация или JIT)	высокая (нативная компиляция)
Точность	высокая (включая тайминги, конвейер)	средняя (поведенческая)
Сложность	большая	малая
Применение	тестирование ОС, сложных систем	отладка логики драйверов и приложений

Подход «двойной компиляции» (dual-target compilation):

Исходный код приложения пишется с использованием абстрактного слоя (HAL или собственного API). Для микроконтроллера используется реальная реализация (работа с регистрами), для ПК – функции-эмуляторы, которые взаимодействуют с эмулятором периферии. Условная компиляция через #ifdef _WIN32 или отдельные проекты.

Пример условного кода:

```
#ifdef PC_EMULATION
   #include "emulator/gpio_emu.h"
   #define GPIO_SetPin(pin)   emu_gpio_set(pin)
#else
   #include "stm32f4xx.h"
   #define GPIO_SetPin(pin)   GPIOA->BSRR = (1<<pin)
#endif
```

Моделирование времени:

Эмулятор должен отслеживать время выполнения операций. Например, таймер, настроенный на период 500 мс, должен через 500 мс (по часам ПК) сгенерировать прерывание. В простейшем случае можно использовать select() или time.sleep() в отдельном потоке, либо при каждом вызове функции эмулятора сравнивать текущее время с заданным.

Архитектура эмулятора (на Python):

- Emulator – главный класс, содержит экземпляры GPIO, UART, Timer.
- GpioPort – массив пинов, регистры MODER, ODR, IDR, BSRR.
- Uart – приём/передача через виртуальный последовательный порт (с использованием модуля pty или socket).
- Timer – хранит список активных таймеров с временами срабатывания, использует heapq для ближайшего события.
- Основной цикл эмулятора: while True: обработать события (таймеры, UART) и дать время на выполнение прошивки.

Методика выполнения работы

1. **Создание эмулятора GPIO на Python**

   1.1. Создайте каталог lab16 и файл emulator/gpio.py:

```
import threading
import time
from enum import Enum

class PinMode(Enum):
    INPUT = 0
    OUTPUT = 1
    AF = 2
    ANALOG = 3
```

```
class Pin:
    def __init__(self, pin_id):
        self.id = pin_id
        self.mode = PinMode.INPUT
        self.otype = 0  # 0=push-pull, 1=open-drain
        self.speed = 0
        self.pull = 0
        self.output = 0
        self.input = 0

    def set_output(self, value):
        if self.mode == PinMode.OUTPUT:
            self.output = value
            # если open-drain и value=1, то фактически входа, но упростим

    def read(self):
        if self.mode == PinMode.INPUT:
            return self.input
        elif self.mode == PinMode.OUTPUT:
            return self.output
        return 0

class GPIOPort:
    def __init__(self, port_id, pin_count=16):
        self.port_id = port_id
        self.pins = [Pin(i) for i in range(pin_count)]
        self.odr = 0
        self.idr = 0
        self.bsrr = 0
```

```
def write_odr(self, value):
    self.odr = value & 0xFFFF
    for i in range(16):
        if (self.odr >> i) & 1:
            self.pins[i].set_output(1)
        else:
            self.pins[i].set_output(0)

def read_idr(self):
    value = 0
    for i in range(16):
        if self.pins[i].read():
            value |= (1 << i)
    self.idr = value
    return value

def write_bsrr(self, value):
    # младшие 16 бит: установка, старшие: сброс
    set_mask = value & 0xFFFF
    reset_mask = (value >> 16) & 0xFFFF
    for i in range(16):
        if set_mask & (1 << i):
            self.pins[i].set_output(1)
            self.odr |= (1 << i)
        if reset_mask & (1 << i):
            self.pins[i].set_output(0)
            self.odr &= ~(1 << i)

def set_input(self, pin, value):
    if 0 <= pin < 16:
```

```
            self.pins[pin].input = 1 if value else 0
```

1.2. Создайте заголовочный файл для C-кода прошивки: emulator/gpio_emu.h:

```
#ifndef GPIO_EMU_H
#define GPIO_EMU_H

#include <stdint.h>

// Функции, которые вызывает прошивка
void emu_gpio_init(void);
void emu_gpio_set_mode(uint8_t port, uint8_t pin, uint8_t mode);
void emu_gpio_set_pin(uint8_t port, uint8_t pin);
void emu_gpio_reset_pin(uint8_t port, uint8_t pin);
uint8_t emu_gpio_read_pin(uint8_t port, uint8_t pin);

// Для эмуляции: установить входной сигнал
void emu_gpio_set_input(uint8_t port, uint8_t pin, uint8_t value);

#endif
```

2. **Реализация эмулятора UART**

2.1. Создайте emulator/uart.py:

```
import socket
import threading
import queue

class UART:
    def __init__(self, port_name, baudrate=115200):
        self.rx_queue = queue.Queue()
```

```
        self.tx_callback = None
        # Можно подключиться к виртуальному COM-порту или сокету
        # Для простоты используем локальный сокет
        self.server_socket = None

    def set_tx_callback(self, callback):
        """callback вызывается при передаче байта"""
        self.tx_callback = callback

    def receive_byte(self, byte):
        """Вызывается из внешнего интерфейса для подачи байта в приёмник"""
        self.rx_queue.put(byte)

    def read_rx(self):
        """Вызывается из эмулятора при запросе приёмника"""
        try:
            return self.rx_queue.get_nowait()
        except queue.Empty:
            return -1

    def send_byte(self, byte):
        """Вызывается из прошивки при отправке"""
        if self.tx_callback:
            self.tx_callback(byte)
```

2.2. Для интеграции с C-кодом: emulator/uart_emu.c:

```
#include "uart_emu.h"
#include <stdio.h>
```

```
static void (*tx_callback)(uint8_t) = NULL;

void emu_uart_init(uint32_t baudrate) {
    // инициализация эмулятора
}

void emu_uart_putc(char c) {
    // в консоль отладки
    putchar(c);
    if (tx_callback) tx_callback(c);
}

char emu_uart_getc(void) {
    // в полноценном эмуляторе должно быть получение из очереди
    return getchar();  // для демонстрации, блокирующий ввод
}

void emu_uart_register_tx_callback(void (*cb)(uint8_t)) {
    tx_callback = cb;
}
```

3. **Эмуляция таймера с генерацией прерываний**

3.1. emulator/timer.py:

```
import heapq
import time
import threading

class Timer:
    def __init__(self, timer_id):
        self.id = timer_id
```

```
        self.prescaler = 0
        self.arr = 0
        self.cnt = 0
        self.enabled = False
        self.irq_callback = None
        self.next_event_time = None

    def set_irq_callback(self, callback):
        self.irq_callback = callback

    def update_registers(self, psc, arr):
        self.prescaler = psc
        self.arr = arr

    def start(self):
        self.enabled = True
        self.cnt = 0
        self._schedule()

    def stop(self):
        self.enabled = False

    def _schedule(self):
        if not self.enabled or self.arr == 0:
            return
        # Расчёт периода по тактовой частоте эмуляции (условно 84 МГц)
        # Для простоты: период = ((psc+1)*(arr+1)) / 84e6 секунд
        # Здесь упростим: период = (arr+1)*0.001 (мс)
        period_sec = (self.arr + 1) * 0.001
        self.next_event_time = time.time() + period_sec
```

 # Запуск таймера в отдельном потоке или проверка в основном цикле

```
class TimerManager:
    def __init__(self):
        self.timers = []
        self.event_queue = []  # (время, timer_id)

    def add_timer(self, timer):
        self.timers.append(timer)

    def run(self):
        while True:
            current_time = time.time()
            for timer in self.timers:
                if timer.enabled and timer.next_event_time and current_time >= timer.next_event_time:
                    timer.cnt += 1
                    if timer.cnt >= timer.arr:
                        timer.cnt = 0
                        if timer.irq_callback:
                            timer.irq_callback(timer.id)
                    timer._schedule()
            time.sleep(0.001)  # 1 мс точность
```

4. **Интеграция эмулятора с С-кодом прошивки**

4.1. Напишите общий заголовочный файл hal_gpio.h:

```
#ifndef HAL_GPIO_H
#define HAL_GPIO_H
```

```
#ifdef PC_EMULATION
   #include "emulator/gpio_emu.h"
   #define GPIO_INIT()        emu_gpio_init()
   #define GPIO_SET_PIN(p)    emu_gpio_set_pin(0, p)
   #define GPIO_RESET_PIN(p)  emu_gpio_reset_pin(0, p)
   #define GPIO_READ_PIN(p)   emu_gpio_read_pin(0, p)
#else
   #include "stm32f4xx.h"
   #define GPIO_INIT()          do { RCC->AHB1ENR |= (1<<0); GPIOA->MODER |= (1<<(5*2)); } while(0)
   #define GPIO_SET_PIN(p)    GPIOA->BSRR = (1<<(p))
   #define GPIO_RESET_PIN(p)  GPIOA->BSRR = (1<<((p)+16))
   #define GPIO_READ_PIN(p)   ((GPIOA->IDR >> (p)) & 1)
#endif

#endif
```

4.2. Текст приложения пишется с использованием этих макросов.

5. **Создание эмулятора и тестирование**

5.1. Напишите на Python код, который инициализирует эмулятор, загружает (в виде динамической библиотеки) или вызывает функции C-прошивки, предварительно скомпилированной для ПК (например, в DLL или SO). Для этого можно скомпилировать прошивку с флагом -DPC_EMULATION и подключить как разделяемую библиотеку, а Python через ctypes вызывает main_loop().

5.2. Альтернатива: написать на C++ эмулятор, который включает в себя исходные файлы прошивки напрямую (больше подходит для небольших проектов).

5.3. Простой способ – создать отдельное консольное приложение на C++ (для ПК), которое подключает main.c и эмулятор через #define PC_EMULATION. Это приложение и будет эмулятором.

6. **Тестирование и валидация**

6.1. Выберите простое приложение (например, мигание светодиодом с управлением по кнопке). Скомпилируйте его для ПК с эмулятором. Запустите, эмулируйте нажатие кнопки через внешний интерфейс (например, ввод символа с клавиатуры). Убедитесь, что светодиод (виртуальный) меняет состояние и в консоли выводятся сообщения.

6.2. Сравните поведение с реальной платой (или с предыдущими лабораторными).

Пример выполнения задания

Задача: Создать эмулятор простейшей системы: светодиод на порту PA5, кнопка на PC13, при нажатии кнопки светодиод переключается (мигание). Прошивка написана с использованием условной компиляции.

Код прошивки (main.c):

```
#include "hal_gpio.h"
#include "hal_timer.h"   // эмуляция задержек

int main(void) {
   GPIO_INIT();
   while (1) {
      if (GPIO_READ_PIN(13) == 0) { // кнопка нажата
         GPIO_SET_PIN(5);
         delay_ms(100);
         GPIO_RESET_PIN(5);
         delay_ms(100);
      }
   }
```

```
}
```

Эмулятор на Python (управление):

```
import time
from emulator.gpio import GPIOPort

# Создаём эмуляторы портов
gpioa = GPIOPort('A')
gpioc = GPIOPort('C')

# Задаём начальное состояние: кнопка не нажата (PC13 = 1, подтяжка)
gpioc.set_input(13, 1)

# Цикл эмуляции (в реальном времени)
while True:
    # Вызываем одну итерацию прошивки (если она собрана как dll)
    # Здесь упростим: вместо реального вызова симулируем логику
    if gpioc.read_idr() & (1<<13):  # кнопка не нажата
        pass
    else:
        gpioa.set_pin(5, 1)
        time.sleep(0.1)
        gpioa.set_pin(5, 0)
        time.sleep(0.1)

    # Даём возможность пользователю изменить вход
    cmd = input("Enter 'p' to press button: ")
    if cmd == 'p':
```

```
        gpioc.set_input(13, 0)
    time.sleep(0.01)
```

Результат: При нажатии клавиши 'p' виртуальный светодиод «мигает».

Индивидуальные задания

Вариант 1. Эмулятор UART с логгированием

Расширьте эмулятор UART: записывайте все переданные байты в файл (tx_log.txt). Добавьте интерфейс для подачи строки в приёмник (например, команда send "Hello"). Протестируйте на CLI из ЛР10.

Вариант 2. Эмулятор ШИМ (PWM)

Реализуйте модель таймера в режиме PWM. При изменении регистра CCR выводите в консоль информацию о скважности. Визуализируйте ШИМ-сигнал (печать символов '.' и '-').

Вариант 3. Визуализация GPIO через графический интерфейс

Используйте Tkinter или PyQt для создания панели управления: 16 кнопок для входов, 16 светодиодов для выходов. Клик по кнопке устанавливает входное значение, выходной светодиод загорается при установке вывода.

Вариант 4. Эмулятор I2C EEPROM (как в ЛР11)

Реализуйте эмулятор EEPROM 24LC02 (256 байт) на Python. Он должен обрабатывать запросы от прошивки через эмулятор I2C (функции передаются через вызовы из C). Добавьте дамп памяти по команде.

Вариант 5. Моделирование времени в эмуляторе (таймеры)

Настройте встроенный таймер (SysTick) на прерывания каждую 1 мс. В эмуляторе реализуйте очередь событий с использованием heapq и точного времени. Протестируйте работу планировщика задач из ЛР13.

Вариант 6. Разработка на C++ вместо Python

Реализуйте все классы эмулятора на C++ (GPIO, UART, Timer). Сделайте так, чтобы исходный код прошивки напрямую компилировался с ними (например, через -DPC_EMULATION и подключение emulator.hpp).

Вариант 7. Эмулятор АЦП с генерацией синусоиды

Создайте эмулятор АЦП, который при запросе преобразования возвращает значения по синусоидальному закону (или данные из заданной таблицы). Добавьте команду для установки частоты.

Вариант 8. Совмещение с реальным UART (связь с ПК)

Настройте эмулятор UART так, чтобы он подключался к реальному COM-порту (через pyserial). Тогда программа на микроконтроллере (в эмуляторе) может обмениваться данными с реальным терминалом.

Вариант 9. Эмуляция энергопотребления

В модели GPIO добавьте подсчёт количества переключений и вычисление приблизительного потребления тока (мА * время). Выводите статистику в конце эмуляции.

Вариант 10. Автоматическое тестирование (unit-test)

Используя эмулятор, напишите на Python тест, который подаёт последовательность входных сигналов и проверяет выходные состояния (например, правильность конечного автомата светодиода). Сделайте отчёт о прохождении тестов.

Вариант 11. Эмуляция прерываний с внешних событий

Реализуйте эмулятор внешнего прерывания (EXTI). При нажатии кнопки в GUI вызывайте обработчик прерывания в прошивке. Убедитесь, что обработчик вызывается без задержек.

Вариант 12. Интеграция с QEMU (полносистемная эмуляция)

Запустите образ прошивки в QEMU для Cortex-M3. Подключитесь через GDB и сравните с вашим эмулятором периферии. Оцените сложность настройки.

Вариант 13. Эмулятор flash-памяти (для сохранения настроек)

Реализуйте эмулятор внутренней Flash с поддержкой стирания страниц и записи слов. Сохраняйте содержимое в файл, чтобы оно сохранялось между сеансами.

Вариант 14. Распределённая эмуляция (клиент-сервер)

Реализуйте сервер эмулятора на Python, который принимает команды по сокету. Отдельное приложение (прошивка, скомпилированная под ПК) подключается к серверу и через протокол вызывает эмулятор. Это позволит запускать несколько экземпляров.

Вариант 15. Эмулятор для Arduino (AVR)

Перенесите подход для Arduino Uno (AVR). Напишите эмулятор периферии (PORTB, DDRB, PINB, таймеры) на Python и адаптируйте простой скетч C++ для компиляции под Linux.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.
2. Цель и задачи лабораторной работы.
3. Теоретическую часть (понятие программного эмулятора, двойная компиляция, моделирование времени, архитектура эмулятора).
4. Листинги реализованных модулей эмулятора (например, gpio.py, uart.py, timer.py) и адаптированной прошивки с использованием условной компиляции.
5. Описание архитектуры эмулятора и взаимодействия с прошивкой (диаграмма классов или последовательности).
6. Результаты тестирования: скриншоты консоли с управлением эмулятором, логи работы.
7. Сравнение поведения прошивки под эмулятором и на реальном железе (если доступно) – таблица соответствия.
8. Выводы о применимости эмуляции для отладки, её ограничениях и преимуществах.
9. Ответы на контрольные вопросы.

Контрольные вопросы

1. Чем отличается программный эмулятор периферии от полносистемного эмулятора типа QEMU?
2. Какие основные компоненты необходимы для эмуляции микроконтроллера на уровне API?
3. Каким образом реализуется «двойная компиляция» (dual-target) для одной кодовой базы?
4. Как в эмуляторе моделируются временные интервалы (например, период таймера)?
5. Какие проблемы могут возникнуть при эмуляции прерываний (например, вложенные прерывания)?
6. Как в эмуляторе обрабатывается конфликт между реальным временем ПК и временем эмулируемого устройства?
7. Преимущества и недостатки эмуляции на Python по сравнению с C++.
8. Как организовать подачу входных сигналов в эмулятор (кнопки, данные UART) в интерактивном режиме?
9. Можно ли использовать эмулятор для отладки реальной временно́й логики (например, ШИМ)? Почему?
10. Какие методы синхронизации потоков нужны при эмуляции на Python (GIL, threading)?
11. Как проверить, что эмулятор достаточно точно повторяет поведение реального оборудования?
12. Для каких типов задач эмуляция периферии особенно полезна, а для каких – недостаточно точна?
13. Как в эмуляторе реализовать поддержку DMA? Какие сложности возникают?
14. Как эмулировать аналоговые входы (АЦП) с заданной частотой дискретизации?
15. Как эмулятор может помочь в автоматическом регрессионном тестировании прошивки?

16. Можно ли с помощью эмулятора отлаживать код, который зависит от машинных инструкций (например, вставки ассемблера)?

17. Как в эмуляторе обрабатываются прямые обращения к регистрам через указатели (например, *(volatile uint32_t*)0x40020000)?

18. Каковы типичные ошибки, которые можно обнаружить только на реальном железе, а не в эмуляторе?

19. Как можно совместить эмулятор периферии с реальным отладчиком (GDB)?

20. Какие альтернативы эмулятору (например, симулятор на основе SystemC) существуют для верификации встраиваемых систем?

Лабораторная работа №17

Проект «Программный логический анализатор»

Цель работы: разработка полноценного проекта «логический анализатор» на базе микроконтроллера, способного захватывать цифровые сигналы с высоким временным разрешением (до ~10 МГц) и передавать полученные данные на персональный компьютер через UART для последующей визуализации.

Задачи:

- Изучить принципы работы логических анализаторов: захват сигналов по фронту (или по уровню), временное разрешение, глубина памяти.
- Разработать алгоритм быстрого захвата цифровых сигналов с нескольких входных линий с использованием таймера и прерываний (или DMA) для получения меток времени.
- Реализовать сжатие данных (например, RLE – Run Length Encoding) для уменьшения объёма передаваемой по UART информации.
- Организовать передачу захваченных данных на ПК через UART с использованием DMA или прерываний.
- Написать простой скрипт-приёмник на Python (или другом языке) для приёма данных, их распаковки и построения временной диаграммы (осциллограммы) нескольких каналов.
- Оценить максимальную частоту дискретизации и глубину памяти, достижимую на выбранном микроконтроллере.
- Проанализировать ограничения метода (скорость UART, размер буфера) и предложить пути улучшения (USB, сбор в память и постобработка).

Теоретическая часть

Логический анализатор

Логический анализатор – устройство для наблюдения и записи временных диаграмм цифровых сигналов. В отличие от осциллографа, он

показывает только два уровня (0/1), но может одновременно анализировать десятки каналов и иметь большую глубину памяти. Принцип работы: непрерывно или по триггеру сохраняются состояния входных линий, привязанные ко времени (обычно в виде последовательности отсчётов, взятых через равные интервалы). Полученная выборка отображается в виде графика «уровень – время».

Способы захвата:

1. **Метод равномерной дискретизации** – фиксированная частота выборки (по таймеру). Простота реализации, но большой объём данных.
2. **Метод сжатия интервалов (RLE)** – сохраняются моменты смены состояния и длительность интервала, когда состояние не менялось. Эффективен для медленно меняющихся сигналов.
3. **Захват по гистерезису (переходу)** – запись моментов фронтов (нарастающих и спадающих). Даёт минимальный объём данных для сигналов с редкими переключениями.

Ограничения при использовании UART:

Типичная скорость UART 115200 бод = ~11,5 Кбайт/с. Чтобы передавать поток необработанных отсчётов с частотой 10 МГц (10 Мбайт/с), необходимо сжатие минимум в 1000 раз. Поэтому для высоких частот нужен либо USB (CDC или HID), либо хранение в памяти с последующей передачей после захвата.

Алгоритм сжатия RLE (Run Length Encoding):

При равномерной дискретизации последовательность байтов (состояний каналов) кодируется парами: (состояние, длина серии). Например, вместо 1000 одинаковых отсчётов «0» передаётся одна пара «0, 1000».

Для многоканального анализатора часто используют побитное представление (один байт = состояние 8 входов). Приблизительная степень сжатия зависит от частоты переключений сигнала.

Аппаратная реализация на STM32:

- Генерация временно́й базы – через таймер (TIM2) в режиме «захват по обновлению» или просто прерывания с постоянной частотой.
- Захват состояния порта (например, GPIOA->IDR) в прерывании таймера и запись в кольцевой буфер.
- Передача накопленных данных через UART с использованием DMA (чтобы не тратить время на передачу).
- Триггер: захват начинается при появлении заданного условия на входе (например, фронта на канале 0).

Альтернатива: захват через DMA + таймер (PWM capture).

Можно настроить таймер в режиме захвата входа (Input Capture) и сохранять времена фронтов непосредственно аппаратно. Это даёт большую точность, но требует больше регистров.

Методика выполнения работы

1. **Подготовка проекта и базовый захчик**

   1.1. Создайте каталог lab17 и файл logic_analyzer.h:

   ```
   #ifndef LOGIC_ANALYZER_H
   #define LOGIC_ANALYZER_H

   #include <stdint.h>
   #include <stdbool.h>

   // Настройки анализатора
   #define LA_MAX_CHANNELS    8        // до 8 каналов (GPIOx 0..7)
   ```

```
#define LA_BUFFER_SIZE       4096        // размер кольцевого буфера отсчётов
#define LA_SAMPLE_RATE_HZ  1000000    // 1 МГц (для начала)

// Инициализация: настройка портов входа, таймера, UART
void la_init(void);

// Запуск захвата (с триггером или без)
void la_start_capture(uint32_t trigger_channel, uint8_t trigger_level);

// Останов захвата (возвращает количество захваченных отсчётов)
uint32_t la_stop_capture(void);

// Отправка сжатых данных на ПК через UART
void la_send_data(void);

// Проверка готовности данных
bool la_data_ready(void);

#endif
```

1.2. Создайте файл logic_analyzer.c (базовая реализация через таймер и прерывания):

```
#include "logic_analyzer.h"
#include "stm32f4xx.h"

static volatile uint8_t sample_buffer[LA_BUFFER_SIZE];
static volatile uint32_t buffer_head = 0;
static volatile bool capture_active = false;
static volatile uint32_t samples_count = 0;
```

```
static uint8_t active_channels_mask = 0x0F;  // первые 4 канала

// Таймер для генерации прерываний с частотой LA_SAMPLE_RATE_HZ
static void timer_init(void) {
    // Включение тактирования TIM2
    RCC->APB1ENR |= (1 << 0);
    // Расчёт PSC и ARR (например, APB1=84 MHz)
    uint32_t f_sample = LA_SAMPLE_RATE_HZ;
    uint32_t prescaler = (84000000 / f_sample) - 1;
    TIM2->PSC = prescaler;
    TIM2->ARR = 1;      // прерывание каждый тик
    TIM2->CNT = 0;
    TIM2->DIER |= (1 << 0);   // UIE
    NVIC_EnableIRQ(TIM2_IRQn);
    NVIC_SetPriority(TIM2_IRQn, 2);
}

void TIM2_IRQHandler(void) {
    if (TIM2->SR & (1 << 0)) {
        TIM2->SR &= ~(1 << 0);
        if (capture_active) {
            // Чтение состояния входов (пины 0..7 порта A)
            uint8_t state = (GPIOA->IDR) & active_channels_mask;
            sample_buffer[buffer_head] = state;
            buffer_head = (buffer_head + 1) % LA_BUFFER_SIZE;
            samples_count++;
            if (samples_count >= LA_BUFFER_SIZE) {
                // переполнение – останов захвата
                capture_active = false;
            }
```

```
        }
    }
}

void la_init(void) {
    // Настройка GPIOA: режим вход (первые 8 пинов)
    GPIOA->MODER &= ~(0xFFFF);   // сброс всех MODER
    GPIOA->PUPDR &= ~(0xFFFF);
    // (подтяжка по желанию)

    timer_init();
}

void la_start_capture(uint32_t trigger_channel, uint8_t trigger_level) {
    // Очистка буфера
    buffer_head = 0;
    samples_count = 0;
    capture_active = true;
    // Триггер (упрощённо: игнорируем, начинаем сразу)
    (void)trigger_channel;
    (void)trigger_level;
    TIM2->CR1 |= (1 << 0);  // запуск таймера
}

uint32_t la_stop_capture(void) {
    capture_active = false;
    TIM2->CR1 &= ~(1 << 0);
    return samples_count;
}
```

```
bool la_data_ready(void) {
    return (!capture_active && samples_count > 0);
}

// Сжатие методом RLE перед отправкой
void la_send_data(void) {
    if (!la_data_ready()) return;
    uint32_t idx = 0;
    while (idx < samples_count) {
        uint8_t cur_state = sample_buffer[idx];
        uint32_t run_len = 1;
        idx++;
        while (idx < samples_count && sample_buffer[idx] == cur_state &&
run_len < 65535) {
            run_len++;
            idx++;
        }
        // Передаём пару: состояние (1 байт) и длина (2 байта, little-endian)
        uart_putc(cur_state);
        uart_putc(run_len & 0xFF);
        uart_putc((run_len >> 8) & 0xFF);
    }
    // Сброс готовности
    samples_count = 0;
}
```

2. **Настройка UART для передачи данных**

2.1. Используйте UART с DMA, как в ЛР12, чтобы передача не отнимала время от захвата. В функции la_send_data вызывать uart_dma_send(buffer, size).

3. **Разработка скрипта-приёмника на Python**

3.1. Создайте файл la_viewer.py:

```
import serial
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
import struct
import sys

# Настройки
PORT = '/dev/ttyACM0'   # или COMx
BAUDRATE = 115200
NUM_CHANNELS = 4

ser = serial.Serial(PORT, BAUDRATE, timeout=1)

def read_packet():
    """Читает RLE-пакет: байт состояния, 16-бит длину"""
    data = ser.read(3)
    if len(data) != 3:
        return None, None
    state = data[0]
    run_len = struct.unpack('<H', data[1:])[0]
    return state, run_len

def decode_to_timeline():
    """Распаковывает RLE в массив отсчётов по времени"""
    timeline = []
    while True:
        state, length = read_packet()
```

```
            if state is None:
                break
            for i in range(length):
                timeline.append(state)
        return timeline

    def plot_timeline(timeline, channels=4):
        """Отображает цифровой сигнал для нескольких каналов"""
        # Разворачиваем по битам
        t = np.arange(len(timeline))
        fig, axes = plt.subplots(channels, 1, sharex=True)
        for ch in range(channels):
            sig = [(val >> ch) & 1 for val in timeline]
            axes[ch].step(t, sig, where='post')
            axes[ch].set_ylabel(f'CH{ch}')
        axes[-1].set_xlabel('Sample number')
        plt.show()

    if __name__ == '__main__':
        print("Waiting for data...")
        tl = decode_to_timeline()
        print(f"Received {len(tl)} samples")
        plot_timeline(tl, NUM_CHANNELS)
```

3.2. Добавьте в скрипт возможность выбора COM-порта, сохранения в файл.

4. **Добавление триггера и настройки фильтра**

4.1. Измените la_start_capture, чтобы анализировать состояние пинов до начала захвата и ждать заданного условия (например, нарастающего фронта на

канале 0). Для этого нужно предварительно читать IDR в цикле перед запуском таймера.

4.2. Реализуйте предварительный буфер (pre-trigger buffer) для сохранения некоторого количества отсчётов до момента срабатывания триггера.

5. **Тестирование на реальном сигнале**

5.1. Подключите к входам анализатора генератор прямоугольных импульсов (или простую схему с кнопкой, или ножку таймера с ШИМ). Захватите сигнал и просмотрите график в Python.

5.2. Оцените максимальную частоту дискретизации, при которой данные не теряются (буфер не переполняется). Экспериментируйте с разной частотой таймера (задавайте через LA_SAMPLE_RATE_HZ).

Пример выполнения задания

Задача: Захватить 4 канала сигналов (канал 0 – внутренний ШИМ-сигнал на частоте 100 кГц, каналы 1-3 – логические уровни от переключателей). Построить временную диаграмму на ПК.

Реализация: Использован метод равномерной дискретизации с частотой 1 МГц, буфер 4096 точек, сжатие RLE. При захвате буфер никогда не переполняется, так как данные сжимаются и отправляются после остановки. Результат – чёткая диаграмма фронтов.

Итог: Проект позволяет анализировать сигналы до 500 кГц с паузами между захватами около 0.5 с (на передачу данных). Применение DMA для UART могло бы повысить скорость передачи.

Индивидуальные задания

Вариант 1. Повышение частоты дискретизации до предела

Используйте DMA для автоматического сбора состояния порта в память (без прерываний). Включите таймер с максимальной частотой (PSC=0, ARR=1), получайте тактовую частоту 84 МГц. Оцените максимальную частоту, при которой данные успевают записываться (ограничение пропускной способности шины). Результат – несколько МГц.

Вариант 2. Сценарий визуализации с временной шкалой в секундах

Модифицируйте Python-скрипт, чтобы он отображал ось времени в секундах, используя частоту дискретизации, переданную из прошивки (например, отдельным пакетом).

Вариант 3. Реализация развёрнутого протокола передачи с контрольной суммой

Добавьте в конце передачи блока 16-битную CRC. Скрипт проверяет CRC и запрашивает повторную передачу при ошибке (команда по UART). Реализуйте простую квитирование.

Вариант 4. Захват по внешнему триггеру (внешний вывод)

Используйте вход EXTI для запуска захвата. Настройте его на фронт. При нажатии кнопки начинается запись с предысторией (захват в кольцевой буфер, по триггеру сохраняем N точек до и M точек после).

Вариант 5. Анализатор состояния шины I2C

Используйте два канала (SCL, SDA). Программа на ПК распознаёт стартовые условия, биты данных, ACK/NACK и выводит декодированные байты.

Вариант 6. Поддержка до 16 каналов со словом состояния uint16_t

Расширьте захват на 16 выводов. Увеличьте буфер. Передача RLE с 3 байтами (длина 16 бит + состояние 2 байта?) – модифицируйте формат.

Вариант 7. Запись в двоичный файл на ПК и пост-анализ

Скрипт сохраняет принятые данные в файл .bin. Дополнительная программа (или тот же скрипт) загружает файл и строит график. Это позволяет анализировать долгие процессы.

Вариант 8. Использование квадратурного энкодера

Подключите энкодер к двум входам. Реализуйте захват с высоким разрешением, а на ПК обрабатывайте данные для подсчёта положения и скорости вращения.

Вариант 9. Беспроводной логический анализатор через Bluetooth (HC-05)

Замените проводной UART на Bluetooth модуль. Принимайте данные на ПК через виртуальный COM-порт. Оцените потерю скорости.

Вариант 10. Интеграция с популярным софтом (sigrok / PulseView)

Изучите протокол подключения оборудования к libsigrok. Реализуйте минимальный драйвер для своего анализатора, чтобы он определялся в PulseView.

Вариант 11. Автоматическое определение частоты сигнала и подстройка захвата

Прошивка в начале измеряет среднюю частоту на одном из каналов (с помощью таймера в режиме захвата) и автоматически выбирает частоту дискретизации в 5–10 раз выше. Это обеспечивает наилучшую детализацию.

Вариант 12. Работа с длительными периодами (захват с глубокой памятью)

Используйте внешнюю SDRAM (если есть) для хранения миллионов отсчётов. Передача происходит после полного заполнения. Сжатие RLE значительно уменьшает время передачи.

Вариант 13. Обнаружение и отображение ширины импульсов

В скрипте на Python вычисляйте длительности положительных и отрицательных импульсов для выбранного канала и выводите гистограмму длительностей.

Вариант 14. Символьная визуализация в консоли (ASCII-art)

Без matplotlib выведите в терминал строки из символов _ и - для каждого канала. Используйте сжатый вывод.

Вариант 15. Web-интерфейс для анализатора

Запускайте на ПК веб-сервер (Flask), который принимает сырые данные от UART (через сокеты или через последовательный порт) и отображает график в браузере с помощью JavaScript (Chart.js). Это обеспечит кроссплатформенную визуализацию.

Требования к отчёту

Отчёт должен содержать:

1. Титульный лист с указанием дисциплины, темы работы, ФИО студента, группы, варианта.

2. Цель и задачи лабораторной работы.

3. Теоретическую часть (логический анализатор, методы сжатия RLE, достижимые частоты).

4. Листинги: logic_analyzer.c, logic_analyzer.h, функция передачи данных, скрипт Python для визуализации.

5. Результаты измерений: таблица максимальных частот дискретизации при разных режимах (с буфером, без сжатия, со сжатием).

6. Скриншоты временных диаграмм, полученных с помощью скрипта (хотя бы для двух каналов).

7. Оценка глубины памяти и максимальной длительности захвата для вашей реализации.

8. Результаты выполнения индивидуального задания.

9. Выводы (какие ограничения и как их обойти, перспективы использования USB/Bluetooth).

10. Ответы на контрольные вопросы.

Контрольные вопросы

1. Что такое логический анализатор? Чем он отличается от осциллографа?

2. Какие методы сжатия данных используются для уменьшения объёма передаваемой информации?

3. Какая максимальная частота дискретизации была достигнута в вашем проекте? От чего она зависит?

4. Какое преимущество даёт метод RLE (Run Length Encoding) при захвате длинных пауз?

5. Как реализован триггер в вашем анализаторе (захват по условию)?

6. Почему для высоких частот (>500 кГц) лучше использовать USB вместо UART?

7. Как можно организовать предварительный буфер (pre-trigger) для сохранения сигнала до события?

8. Какие факторы ограничивают максимальную длительность непрерывного захвата?

9. Как можно увеличить глубину памяти, используя внешнюю память (SDRAM или SPI RAM)?

10. Как в скрипте Python происходит распаковка RLE и построение графика?

11. Можно ли использовать DMA для автоматической передачи данных из памяти в UART без участия CPU? Как это ускорит процесс?

12. Что такое «захват по фронту» и как его реализовать аппаратно без дискретизации?

13. Какой формат данных удобнее передавать: сжатый RLE или сырые отсчёты? В каких случаях каждый выгоден?

14. Как вы оцените погрешность измерения временных интервалов в вашей системе?

15. Как синхронизировать несколько анализаторов для многоканального захвата (>16 каналов)?

16. Каким образом можно визуализировать до 32 каналов одновременно?

17. Как встроенная поддержка захвата через Input Capture может улучшить точность?

18. Что такое «активный пролог/эпилог»? Как они помогают при анализе протоколов?

19. Как можно автоматически определять скорость передачи UART по принимаемым данным?

20. Сравните программную реализацию (таймер+прерывания) и аппаратный захват с помощью DMA. Какие преимущества у каждого подхода?

Лабораторная работа №18

Комплексный проект: «Программно-определяемая периферия»

Цель работы

Выполнение законченного проекта по разработке программно-определяемой периферии (Software-Defined Peripheral) на базе микроконтроллера, объединяющего знания и навыки, полученные в курсе (драйверы GPIO, АЦП, UART/CLI, таймеры, DMA, конечные автоматы, буферизация, энергонезависимая память, планировщик). В результате должен быть создан демонстрационный продукт (например, «Умный переключатель» с веб-конфигурацией через ESP-01, регистратор данных на SD-карту или программный таймер с множеством каналов), демонстрирующий инженерный подход к проектированию встраиваемых систем.

Задачи:

- Самостоятельно сформулировать техническое задание на выбранный проект (функциональные требования, аппаратные ресурсы, интерфейсы).
- Разработать архитектуру программного обеспечения: модульная структура, уровни абстракции (HAL, драйверы периферии, модули прикладной логики, пользовательский интерфейс).
- Реализовать необходимые драйверы периферии (GPIO, таймеры, АЦП, UART, I2C/SPI, DMA, Flash-эмуляция EEPROM) с использованием ранее полученных наработок.
- Реализовать кооперативный или вытесняющий планировщик задач (Round Robin, конечные автоматы, тики от SysTick) для выполнения нескольких фоновых операций.
- Организовать пользовательский интерфейс: минимум – командная строка (CLI) через UART с поддержкой настройки параметров и вывода статуса, максимум – веб-интерфейс через AT-команды ESP8266 или встроенный сервер.

- Обеспечить сохранение пользовательских настроек во внутренней Flash (эмуляция EEPROM) с wear-leveling и проверкой целостности.
- Реализовать обработку ошибочных ситуаций (тайм-ауты, переполнение буферов, сбои связи) с диагностикой через CLI.
- Провести тестирование проекта на реальном оборудовании, задокументировать результаты и подготовить краткое руководство пользователя.

Теоретическая часть

Понятие программно-определяемой периферии (SDP)

В классическом встраиваемом проекте логика обработки данных жестко зашита в микроконтроллер. Программно-определяемая периферия подразумевает, что поведение устройства может быть сконфигурировано или даже перепрограммировано со стороны пользователя без изменения прошивки (через командный интерфейс, веб-страницу, мобильное приложение или скриптовый язык). Это позволяет одному и тому же устройству выполнять разные функции (например, быть таймером, логическим анализатором, ШИМ-контроллером) в зависимости от настроек.

Архитектурные принципы для комплексного проекта:

1. **Модульность** – каждый периферийный блок и задача оформлены в отдельные файлы (.c/.h) с чёткими интерфейсами.
2. **Обработка ошибок** – все функции драйверов возвращают коды ошибок (bool или enum), и приложение должно корректно на них реагировать.
3. **Настраиваемость** – параметры (пины, частота, пороги) хранятся в энергонезависимой памяти и могут быть изменены через CLI.
4. **Документирование** – каждый модуль снабжён комментариями, поясняющими назначение, форматы данных, примеры использования.

5. **Энергонезависимость** – настройки не теряются при выключении питания.
6. **Безопасность** – проверка диапазонов вводимых значений, защита от записи за пределы буферов, сторожевой таймер (опционально).

Выбор варианта комплексного проекта

Ниже приведены три типовых варианта, которые студент может взять за основу (или предложить свой). Каждый вариант охватывает разные аспекты курса.

- **Вариант А. «Программный таймер с расширенными возможностями»**

  Реализуется несколько программных таймеров (software timers) с аппаратной привязкой к SysTick. Каждый таймер можно запустить в однократном или периодическом режиме, привязать действие (включение нагрузки, звуковой сигнал, отправку сообщения). Управление через CLI.
- **Вариант Б. «Умная розетка / контроллер нагрузки с веб-интерфейсом»**

  На базе STM32 + ESP8266 (AT-команды) или готового модуля с прошивкой, реализуется реле/симистор, измерение тока/напряжения через АЦП, управление по расписанию (хранится во Flash), возможность просмотра статистики через веб-страницу (ESP8266 в режиме AP или STA).
- **Вариант В. «Регистратор данных с датчиков на SD-карту»** Используется SPI-интерфейс для работы с SD-картой (FatFs или упрощённая файловая система), периодический опрос датчиков (DS18B20, DHT11/22, BMP280 – по I2C или OneWire), запись в файл с временными метками (RTC или программный счётчик). Данные можно выгружать по UART или через карту памяти.

Интеграция наработок из предыдущих ЛР:

- ЛР1-2: кросс-компиляция, ассемблерные вставки (можно использовать для критичных участков, например, измерение времени).
- ЛР3: драйвер GPIO для управления нагрузкой.
- ЛР4: таймеры для периодических опросов и ШИМ.
- ЛР5: обработка прерываний (кнопка, таймер, UART).
- ЛР6: конечные автоматы для управления состояниями.
- ЛР7: программный ШИМ (если аппаратный занят).
- ЛР8: АЦП для измерений.
- ЛР9: кольцевые буферы для UART/АЦП.
- ЛР10: CLI для управления устройством.
- ЛР11-12: I2C (датчики) и DMA (быстрая передача).
- ЛР13: планировщик для кооперативного выполнения задач.
- ЛР14: эмуляция EEPROM для хранения настроек.
- ЛР15-16: отладка, эмуляция для тестирования логики.

Пример архитектуры проекта (вариант «Умная розетка»):

```
Проект/
├── drivers/
│   ├── gpio.c/h          (управление реле, светодиодом)
│   ├── adc.c/h           (измерение тока/напряжения)
│   ├── timer.c/h         (аппаратные таймеры)
│   ├── uart.c/h          (обмен с ESP8266)
│   ├── spi.c/h           (опционально, для SD или дисплея)
│   └── flash_eeprom.c/h  (хранение параметров)
├── middleware/
│   ├── scheduler.c/h     (простой кооперативный планировщик)
│   ├── ring_buffer.c/h   (для UART)
│   ├── cli.c/h           (командный интерпретатор)
│   └── at_parser.c/h     (парсинг ответов ESP8266)
```

```
├── app/
│   ├── main.c          (архитектура, вызов планировщика)
│   ├── web_server.c/h  (формирование веб-страниц через ESP)
│   ├── power_meter.c/h (алгоритм вычисления мощности)
│   └── schedule.c/h    (расписание включений/выключений)
└── config/
    └── settings.h      (конфигурационные макросы)
```

Методика выполнения работы

1. **Выбор темы проекта и формулировка ТЗ**

1.1. Ознакомьтесь с предложенными вариантами (А, Б, В) или предложите свой проект (согласовав с преподавателем).

1.2. Составьте техническое задание: цели, функциональные требования, список используемой периферии, форматы данных, предполагаемые способы взаимодействия с пользователем.

2. **Проектирование архитектуры**

2.1. Разделите систему на модули в соответствии с горизонтальными уровнями (HAL, драйверы, middleware, приложение).

2.2. Определите, какие модули из предыдущих работ можно переиспользовать (скопировать или адаптировать).

2.3. Нарисуйте диаграмму вызовов и потоков данных (какие модули читают/пишут в кольцевые буферы, какие прерывания их заполняют).

2.4. Определите структуру сохранения настроек (структура user_config_t) и продумайте алгоритмы wear-leveling.

3. **Разработка аппаратной части (схема подключения)**

3.1. Соберите схему на макетной плате: подключите необходимую периферию (кнопки, реле, датчики, ESP8266, SD-карту).

3.2. Убедитесь в соответствии уровней напряжений (ESP8266 – 3.3 В, многие датчики – 5/3.3 В). Используйте согласование логических уровней при необходимости.

4. **Реализация модулей по шагам (итеративная разработка)**

4.1. **Базовый каркас** – создайте проект в среде IDE, настройте кросс-компилятор, заготовку main.c с бесконечным циклом.

4.2. **Драйверы** – последовательно реализуйте и отлаживайте драйверы GPIO (для управления реле), UART (для отладки и связи с ESP), таймера (SysTick для планировщика).

4.3. **Энергонезависимые настройки** – реализуйте модуль flash_eeprom.c по методике ЛР14, сохраните и считайте структуру конфигурации.

4.4. **Планировщик** – создайте кооперативный планировщик на основе системного тика (период 1 мс). Реализуйте функции task_create(period, func), task_dispatch().

4.5. **Пользовательский интерфейс** – CLI через UART (команды помощи, установки параметров, ручного управления).

4.6. **Прикладная логика** – для варианта Б: опрос АЦП (ток/напряжение), расчёт мощности, управление реле по расписанию, веб-сервер на ESP8266.

4.7. **Отладка и тестирование** – проверка каждого модуля изолированно, затем интеграционное тестирование.

5. **Сборка, отладка и документирование**

5.1. Загрузите прошивку в МК, проверьте команды CLI, правильность сохранения настроек.

5.2. Организуйте логирование важных событий (например, через UART в терминал).

5.3. Составьте краткое руководство пользователя (формат текста).

5.4. Подготовьте итоговый отчёт по проекту (структура – см. ниже).

Пример выполнения задания

Выбран вариант Б: «Умная розетка с веб-интерфейсом (ESP8266 + STM32)».

Техническое задание (фрагмент):

- Управление нагрузкой до 2 кВт (симистор с оптроном, управление через GPIO).
- Измерение тока (трансформатор тока + АЦП) и напряжения (делитель + АЦП).
- Хранение расписания: до 10 событий (время, день недели, действие ON/OFF).
- Веб-интерфейс на ESP8266 (режим AP, IP 192.168.4.1): страница состояния (мощность, статус реле), страница установки расписания.
- Управление через CLI: power, schedule list, schedule add, relay on/off, save.
- Автоматическое переключение по расписанию.

Архитектура (упрощённый псевдокод):

```
// main.c
int main(void) {
   HAL_Init();
   SysTick_Init(1000);
   load_config();        // из Flash
   uart_init(115200);
   cli_init();
   relay_init();
   adc_init();
   esp8266_init(9600);
   scheduler_init();
```

```
    scheduler_add_task(task_check_schedule, 1000);
    scheduler_add_task(task_measure_power, 500);
    scheduler_add_task(task_update_web_data, 2000);
    scheduler_add_task(task_process_uart, 10);   // обработка команд
    while (1) {
        scheduler_run();
        // __WFI();
    }
}
```

Результат тестирования:

- Измерения мощности выводятся с погрешностью ±5% (калибровка по эталонному ваттметру).
- Расписание: добавлено событие "вт-чт 18:00 ON", в указанное время реле включилось.
- Веб-страница загружается, отображает текущую мощность и состояние.
- CLI позволяет переопределить настройки, они сохраняются и после сброса восстанавливаются.

Пример взаимодействия через CLI:

```
> power
Power: 58.3 W
> schedule add 3 18:00 ON
Schedule added
> schedule list
1: Wed 18:00 -> ON
> relay off
Relay is OFF
> save
Config saved.
```

Индивидуальные задания (варианты проектов)

Вариант 1. Умный светильник с регулировкой яркости (ШИМ + фоторезистор)

- Реализуйте автоматическую поддержку освещённости (заданный уровень LUX).
- Управление через CLI: установка целевой яркости, ручной режим, калибровка.
- Сохранение настроек во Flash.
- (Доп.) Веб-интерфейс через ESP8266 (просмотр графика освещённости за день).

Вариант 2. Двухканальный программный таймер с ЖК-дисплеем (1602 I2C)

- 4 программных таймера с точностью до 1 мс, могут быть однократными и периодическими.
- При срабатывании таймера включается зуммер или реле на заданное время.
- Настройка через кнопки и отображение на LCD.
- Сохранение конфигурации после сброса.

Вариант 3. Логгер температуры и влажности с записью в MicroSD (CSV)

- Датчик DHT22 (или SHT30 по I2C).
- Опрос каждые 10 с, сохранять в файл data.csv вместе с меткой времени (RTC DS3231 или программный счётчик от старта).
- Команды CLI: просмотр последних записей, сброс лога, форматирование карты (через SPI).
- Энергонезависимая память для хранения настроек (период опроса, смещение времени).

Вариант 4. Генератор сигналов произвольной формы (DDS на базе ЦАП + внешний усилитель)

- Использование встроенного ЦАП или внешнего (MCP4921).

- Форма сигнала задаётся таблицей в памяти (синус, меандр, пила, пользовательский массив).
- Выбор частоты (0,1 Гц – 10 кГц) и амплитуды через CLI.
- ШИМ-фильтр низких частот для получения аналогового сигнала.

Вариант 5. Система удалённого мониторинга (STM32 + HC-05 Bluetooth)

- Сбор данных с двух датчиков (АЦП на потенциометрах) и передача по Bluetooth на телефон (приложение-терминал).
- Приём команд (например, «включить светодиод»).
- Реализация простого протокола с контрольной суммой.
- CLI через UART (для отладки) и через Bluetooth приложение.

Вариант 6. Интеллектуальный дозатор жидкостей (расходомер + электромагнитный клапан)

- Измерение протока с помощью датчика Холла (поток воды).
- Управление клапаном (GPIO + реле).
- CLI: задать целевой объём, запустить дозирование (набор жидкости до заданного количества).
- Логирование каждого дозирования во Flash (эмуляция EEPROM) с временной меткой.

Вариант 7. Метеостанция с передачей на ПК через USB (VCP) и архивированием

- Датчики температуры, давления, влажности (BME280).
- Данные раз в минуту сохраняются во внешнюю EEPROM (24LC256) с временной меткой (RTC).
- При подключении к USB (VCP через ST-Link или собственный USB CDC) автоматически передавать архив в виде CSV.

Вариант 8. Синтезатор MIDI на базе STM32 (I2S аудио)

- Приём MIDI команд через UART или USB.
- Генерация звуковых импульсов (PWM с фильтром или I2S-ЦАП).
- Выбор инструмента (волновой стол) из таблиц.

- CLI для установки MIDI-канала, настройки тембра.

Вариант 9. Контроллер полива растений с тремя клапанами

- Управление по расписанию (3 независимых канала, каждому своё время и длительность).
- Датчик влажности почвы (аналоговый) – при низкой влажности включать вне графика.
- Сбор статистики (количество поливов, общее время).
- CLI или веб-интерфейс (ESP-01).

Вариант 10. Смарт-зарядное устройство (измерение тока и напряжения через АЦП, отключение по заряду)

- Заряд аккумулятора (Li-Ion 18650) через специализированный модуль TP4056.
- Мониторинг тока и напряжения на клеммах.
- При достижении 4,1 В – отключение зарядки (реле) и вывод сообщения.
- Запись профиля зарядки (ток/время) в Flash.

Вариант 11. Логический анализатор с триггером по последовательности (проект ЛР17, но без ПК)

- Захват 8 каналов, глубина 4096 отсчётов.
- Настройка триггерной последовательности через CLI (например, "0xAA" на канале 0–7).
- Отображение результатов на встроенном LCD или отправка по UART.

Вариант 12. Эмулятор клавиатуры USB (HID) для автоматизации

- STM32F4 с поддержкой USB HID.
- По командам от UART с ПК эмулирует нажатия клавиш (макросы).
- Возможность записи макросов в энергонезависимую память.

Вариант 13. Терморегулятор с PID-управлением и графическим дисплеем (ILI9341)

- Датчик температуры (DS18B20 или термопара MAX6675).
- Управление нагревателем (ШИМ через симистор).

- Отображение текущей и заданной температуры, графика изменения температуры.
- CLI для установки уставки, коэффициентов PID.

Вариант 14. Синхронный детектор (Lock-in) для прецизионных измерений на частоте 1 кГц

- Генератор опорного сигнала (DDS) и измерительный АЦП.
- Цифровая обработка: умножение на квадратурные составляющие, фильтр нижних частот.
- Вывод амплитуды и фазы через CLI.
- (Сложный проект, подходит для продвинутых).

Вариант 15. Разработка собственного проекта по выбору студента (после согласования)

- Тема может быть любой, но она должна использовать как минимум 5 различных аппаратных модулей (GPIO, таймер, АЦП, UART, I2C/SPI, внешняя память, дисплей и т.д.) и предусматривать сохранение настроек и управление через CLI или веб.

Требования к отчёту

Отчёт по комплексному проекту должен содержать:

1. Титульный лист (название дисциплины, тема проекта, ФИО, группа, вариант).

2. Техническое задание (цель, функциональные и нефункциональные требования).

3. Архитектурная схема (разбиение на модули, связи, потоки данных).

4. Электрическая принципиальная схема (или описание подключений периферии).

5. Листинги кода (в объёме, достаточном для понимания реализации, с комментариями).

6. Описание реализации ключевых алгоритмов (например, планировщик, wear-leveling, обработка команд).

7. Результаты тестирования: таблицы тестов, скриншоты терминала, фотографии работающего устройства.

8. Руководство пользователя (краткое, команды CLI, настройка подключения, типичные сценарии).

9. Анализ возникавших проблем и способов их решения.

10. Выводы по проекту (что удалось, что не удалось, перспективы развития).

11. Ответы на контрольные вопросы.

Контрольные вопросы

1. Какие этапы включает жизненный цикл разработки встраиваемого программного обеспечения, и как вы их выполнили в проекте?

2. Каким образом вы организовали сохранение пользовательских настроек? Какой алгоритм wear-leveling использовали?

3. Почему в проекте был выбран кооперативный (или вытесняющий) планировщик? Какие задачи в нём выполняются с какими периодами?

4. Как обеспечена устойчивость системы к потере питания во время записи данных во Flash?

5. Опишите используемые в проекте протоколы обмена данными (UART, I2C, SPI). Какая максимальная скорость и почему?

6. Как вы обрабатывали ошибки (переполнение буфера, тайм-ауты, некорректные команды)? Приведите пример.

7. Использовали ли вы метод конечных автоматов? Для каких задач и как это реализовано?

8. Как осуществлялась отладка проекта? Какие инструменты (JTAG/SWD, логический анализатор, UART) применялись?

9. Какие показатели энергопотребления у вашего устройства (расчёт или измерение)? Можно ли его уменьшить?

10. Как тестировалась корректность работы расписания (если оно есть)? Какие крайние случаи проверялись?

11. Как обеспечена безопасность (защита от перегрузки, от некорректных параметров, длительное нажатие кнопок)?

12. Какие модули из предыдущих лабораторных работ были использованы без изменений, а какие доработаны?

13. Как организован ввод-вывод при работе с энергонезависимой памятью? Есть ли буферизация для ускорения?

14. Какую структуру данных вы использовали для хранения расписания/списка таймеров? Почему?

15. Как вы поддерживали синхронизацию между задачами (избегали гонок данных)?

16. Какие временные параметры были важны в вашем проекте (задержки, частоты опроса)? Как вы их измеряли и настраивали?

17. Как вы подошли к модульному тестированию компонентов перед интеграцией?

18. Каковы были критерии успешного завершения проекта? Какие из них достигнуты полностью, а какие частично?

19. Если бы вы продолжили развитие проекта, что улучшили в первую очередь?

20. Сравните сложность создания данного проекта «с нуля» с использованием готового набора библиотек (HAL, RTOS). Оправдано ли использование самодельной архитектуры?

СПИСОК ЛИТЕРАТУРЫ

Основная литература

1. Максимов Н. В. Архитектура ЭВМ и вычислительных систем: учебник / Н. В. Максимов, Т. Л. Партыка, И. И. Попов. — 5-е изд., перераб. и доп. — Москва: Форум; ИНФРА-М, 2024. — 511 с. — ISBN 978-5-00091-511-0.

2. Новожилов О. П. Архитектура ЭВМ и систем: учебник для вузов / О. П. Новожилов. — 2-е изд., испр. и доп. — Москва : Юрайт, 2024. — 511 с. — (Высшее образование). — ISBN 978-5-534-18445-7.

3. Ершова Н. Ю. Организация вычислительных систем: учебное пособие / Н. Ю. Ершова, А. В. Соловьев. — 4-е изд. — Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2025. — 221 с. — ISBN 978-5-4497-0904-2.

4. Гей У. Основы STM32: руководство / У. Гей ; пер. с англ. Ю. В. Ревича. — Москва: ДМК Пресс, 2025. — 362 с. — ISBN 978-5-93700-294-5.

5. Иванов В. В. Микропроцессорная техника: учебное пособие / В. В. Иванов. — Самара: Изд-во Самар. ун-та, 2019. — on-line. — ISBN 978-5-7883-1422-8.

6. Сухатерин А. Б. Архитектура микроконтроллеров ARM серии Cortex-M: учебное пособие. — Москва: ФГБОУ ВО «МИРЭА — Российский технологический университет», 2023.

7. Ефимов А. И. Программирование микроконтроллеров ARM CORTEX-M3: учебное пособие / А. И. Ефимов, А. В. Кистрин, Д. И. Устюков. — Москва: КУРС, 2024. — 112 с.

8. Shiva S. G. Computer Organization, Design, and Architecture / S. G. Shiva. — CRC Press, 2025. — 576 p. — ISBN 978-1-032-80512-2 (print).

9. Englander I. The Architecture of Computer Hardware, Systems Software, and Networking: An Information Technology Approach / I. Englander, W. Wong. — 6th ed. — Wiley, 2024. — 680 p. — ISBN 9789363868311.

Дополнительная литература

10. Ключарёв А. А. Программирование микроконтроллеров STM32: учебное пособие. — Санкт-Петербург: ГУАП, 2023. — 196 с. — ISBN 978-5-8088-1829-3.

11. Мостовой Д. В. Программирование на языке ассемблера микроконтроллеров Cortex-M0: учебное пособие / Д. В. Мостовой, М. С. Павлова, Д. А. Серегин и др. — Москва: Изд-во МЭИ, 2025. — 61 с. — ISBN 978-5-7046-3128-6.

12. Шамров М. И. Программирование микроконтроллеров семейства CORTEX-M: учебное пособие / М. И. Шамров. — Москва: РУТ (МИИТ), 2020. — 88 с.

13. Yiu J. The Definitive Guide to Arm® Cortex®-M23 and Cortex-M33 Processors / J. Yiu. — 2025. — Focuses on the Armv8-M architecture, instruction set, programmer's model, interrupt handling, OS support, and debug features.

14. CMSIS-RTOS для микроконтроллеров с ядром Cortex-M3: методические указания. — Самара: Самар. нац. исслед. ун-т им. С. П. Королева.

Интернет-ресурсы

15. ARM Developer — Cortex-M Resources — официальная документация, технические руководства (TRM) и руководства пользователя для всех процессоров Cortex-M. — Режим доступа: https://developer.arm.com (свободный).

16. ARM Information Center — Architecture Reference Manuals для Armv6-M, Armv7-M и Armv8-M, Programmer's Model, Instruction Set. — Режим доступа: https://developer.arm.com/documentation (свободный).

17. STMicroelectronics — официальный сайт, техническая документация на микроконтроллеры STM32 (Reference Manuals, Datasheets, Application Notes). — Режим доступа: https://www.st.com (свободный).

18. GitHub — arm-education / Embedded-Systems-Fundamentals — открытый учебник по основам встраиваемых систем с ядром Arm Cortex-M и практическими лабораторными работами. — Режим доступа: https://github.com/arm-education/Embedded-Systems-Fundamentals (свободный).

19. OpenOCD User's Guide — документация по отладчику OpenOCD, работа с GDB и SWD. — Режим доступа: https://openocd.org/doc-release/html (свободный).