



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления (ИУ)»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии (ИУ7)»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Загружаемый модуль для устройств с интерфейсом GPIO»

Студент группы ИУ7-72Б

(Подпись, дата)

Иванов В.А.

(И.О. Фамилия)

Руководитель курсового проекта

(Подпись, дата)

Рязанова Н.Ю.

(И.О. Фамилия)

2021 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	8
1 Аналитическая часть	9
1.1 Постановка задачи	9
1.2 Актуальность проблемы	9
1.3 Метод решения	9
1.4 Критерии оптимизации	9
2 Конструкторская часть	10
3 Технологическая часть	11
4 Исследовательская часть	12
ЗАКЛЮЧЕНИЕ	13
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	14
ПРИЛОЖЕНИЕ А	15

ВВЕДЕНИЕ

В настоящее время активно развиваются технологии ”умного дома”и ”интернета вещей”. Они направлены на включение в локальную сеть дома бытовых приборов любого. Целью является их тесная интеграция, создание возможности управления и получения информации об их текущем состоянии для жильца дома.

Разработка подобных ”умных”устройств тесно связана с использованием микроконтроллеров или одноплатных компьютеров. Такие устройства имеют небольшой размер и представляют незначительные вычислительные мощности. Поэтому для них зачастую требуется применение низкоуровневого программирования.

Наиболее распространённым физическим интерфейсом подключения является GPIO (general-purpose input/output) - контакты ввода/вывода, позволяющие подключать к компьютеру широкий спектр различных устройств: датчиков, переключателей, двигателей и т.п.

Данная работа посвящена разработке загружаемого модуля для взаимодействия с устройствами с интерфейсом GPIO.

1 Аналитическая часть

1.1 Постановка задачи

В соответствии с заданием курсового проекта, требуется разработать загружаемый модуль ядра, позволяющий управлять устройствами, подключенными с помощью интерфейса GPIO.

Для достижения данной цели необходимо решить следующие задачи:

- 1) проанализировать особенности интерфейса GPIO;
- 2) определить формат передаваемых данных;
- 3) проанализировать и выбрать тип программного управления;
- 4) разработать программное обеспечение;
- 5) провести тестирование программы.

1.2 Анализ принципов работы интерфейса GPIO

GPIO - интерфейс для связи между компонентами компьютерной системы, к примеру, микропроцессором и различными периферийными устройствами[1]. Контакты GPIO могут выступать как в роли входа, так и в роли выхода — это, как правило, конфигурируется. Обычно они используются для подключения датчиков, переключателей, дисплеев и т.п.

В данной работе будет использоваться одноплатный компьютер Raspberry Pi 2B ввиду отсутствия в распоряжении других компьютеров. Рассмотрим подробнее организацию GPIO на его примере.

На рисунке 1 описывается назначение каждого из контактов (пинов) для этой модели. Можно отметить, что не все из них используются для ввода/вывода. Помимо этого есть контакты предназначенные для подачи определённого напряжения на внешние устройства (3.3V, 5V) или для заземления (GROUND).

Пин GPIO имеет два режима:

- **Вход.** Напряжение подаётся внешним устройством. От +0.0V до +1.8V считается уровнем логического нуля, +1.8V-3.3V - логическая единица.
- **Выход.** Напряжение подаётся самим Raspberry Pi. Уровень логических 0

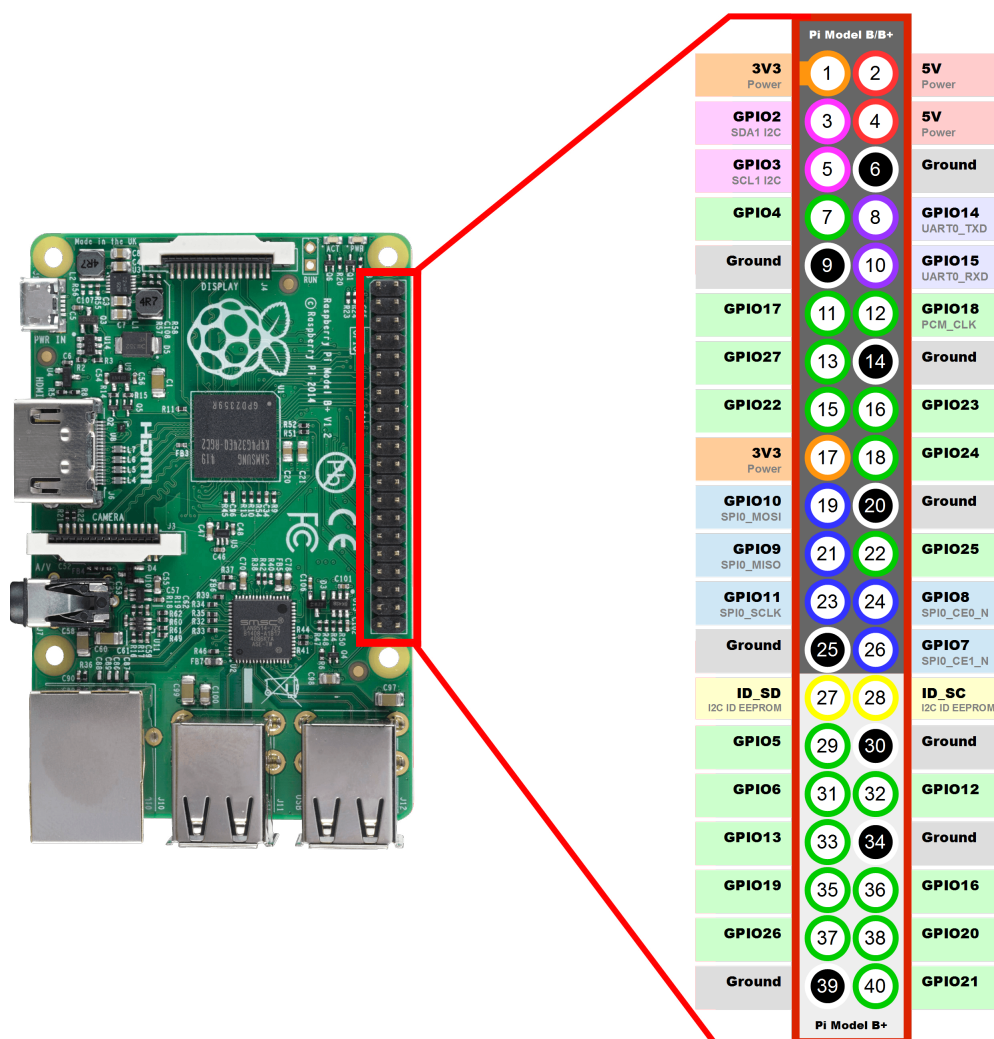


Рисунок 1 – Назначение пинов Raspberry Pi 2B

и 1 аналогичный.

По сути, передача и приём информации осуществляется только считыванием и установлением определённого напряжения. Выходы, отмеченные на схеме зелёным цветом имеют наиболее простой принцип действия: режим ввода/вывода в них устанавливается на всё время подключения устройства, а значение напряжения является относительно постоянным, так как по ним передаётся минимальное количество информации.

В отличие от них, контакты имеющие подпись I2C, SPI или UART используются для последовательной синхронной передачи данных в режиме полного или полудуплекса. Это означает, что они используются для передачи уже более большого количества данных и могут переключать режим ввода/вывода по

несколько тысяч раз за секунду.

Целью данной работы является разработка базового взаимодействия с внешними устройствами, поэтому ПО будет ориентировано на более простые интерфейсы GPIO.

1.3 Анализ способа передачи информации

Следующей задачей является определение способа и формата данных в передаче информации интерфейсу GPIO.

Для работы с пинами используется отображение в память (memory mapping). Для чтения или изменения состояния устройства требуется взаимодействовать с определённым участком оперативной памяти, имеющей постоянный физический адрес.

Рассмотрим устройство отображения GPIO в память для Raspberry Pi 2B. В листинге 1 приведены используемые константы[3].

Листинг 1: Адреса memory_mapping

```
1  #define BCM2708_PERI_BASE    0x3F000000
2  #define GPIO_BASE            (BCM2708_PERI_BASE + 0x200000)
3  /* GPIO register offsets */
4  #define GPFSEL0              0x0      /* Function Select */
5  #define GPSET0               0x1c     /* Pin Output Set */
6  #define GPCLR0              0x28     /* Pin Output Clear */
7  #define GPLEV0              0x34     /* Pin Level */
8  ...
9  #define GPPUDCLK0           0x98
```

Адрес **BCM2708_PERI_BASE**, находящийся в области памяти ядра, определяет начало участка memory mapping для периферийных устройств, участок GPIO имеет смещение 0x200000 и составляет размер в 256 байт. Далее представлены смещения участков памяти относительно базового адреса **GPIO_BASE**, каждое из которых отвечает соответственно за режим ввода/вывода, установку, сброс и чтение значения с пинов.

1.4 Структура загружаемого модуля ядра

Доступ к физической памяти возможно получить только в режиме ядра. Поэтому, для выполнения поставленной задачи требуется написать загружаемый модуль ядра. После загрузки он становится частью ОС и получает доступ к ядрённым функциям и к памяти устройств ввода/вывода, что и требуется в этом случае.

Для обеспечения доступа к устройствам принято решение использовать символьное устройство. Набор возможных действий с ним определяется структурой `file_operations`, которая частично приведена в листинге 2.

Листинг 2: Структура `file_operations`

```
1 struct file_operations {
2     struct module *owner;
3     ...
4     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
5     ...
6     int (*open) (struct inode *, struct file *);
7     int (*release) (struct inode *, struct file *);
8     ...
9 };
```

Взаимодействие с устройствами фактически производится через функцию `unlocked_ioctl`. Для её вызова используется функция **`ioctl`**. В качестве аргументов передаётся структура открытого файла символьного устройства, номер команды и указатель на данные ввода или вывода. Для задания номеров команд используются макросы описанные в листинге 3. Наличие R или W в названии указывают на то, что операция направлена на вывод или ввод соответственно. В качестве аргументов подаются тип и размер передаваемых данных, а также магическое число (уникальное число, позволяющее обнаружить ошибки некорректности команды с помощью макроса `_IOC_TYPE`).

Листинг 3: Макросы команд `ioctl`

```
1 _IO(type, nr);
2 _IOR(type, nr, size);
```

```
3 _IOW(type, nr, size);  
4 _IOWR(type, nr, size);
```

ПО должно предоставлять минимальные необходимые функции для управления контактами:

- ввод / вывод значений;
- захват / возвращения управления пином;
- установление режима работы.

Такие операции как захват и возврат управления нужны для того, чтобы обеспечить монопольный доступ на изменение для одного процесса. Таким образом, без управления пином возможно только чтение его значения.

1.5 Создание буфера ввода/вывода

Для того, чтобы буфер ввода/вывода стал доступным для использования модулем, требуется вызвать функции представленные в листинге 4 [4].

Листинг 4: Получение доступа к буферу памяти в ядре

```
1 struct resource *request_mem_region(unsigned long start, unsigned long  
len, char *name);  
2 void release_mem_region(unsigned long start, unsigned long len);  
3  
4 void *ioremap(unsigned long phys_addr, unsigned long size);  
5 void iounmap(void *addr);  
6  
7 u32 readl(const volatile void __iomem *addr);  
8 void writel(u32 b, volatile void __iomem *addr);
```

С помощью функции **request_mem_region** производится выделение участка памяти ядра для использования модулем. Однако, данный участок памяти не будет доступен напрямую.

Для того, чтобы использовать выделенный модулю буфер требуется вызвать функцию **ioremap**. Она возвращает виртуальный адрес, который используется для получения доступа к памяти ядра.

Чтение и запись по полученному виртуальному адресу производится че-

рез команды **readl, writel**.

После завершения работы модуля требуется освободить выделенный участок памяти в адресном пространстве ядра. Для этого вызываются функции **iounmap** и **release_mem_region**.

Вывод

В результате проведённого анализа были изучены особенности интерфейса GPIO, определён формат передачи данных. Было принято решение в качестве средства передачи информации между модулем ядра и пользовательскими процессами использовать символьное устройство.

2 Конструкторская часть

2.1 IDEF0 диаграмма

На рисунках 2 и 3 приведён алгоритм обработки команды ввода/вывода.

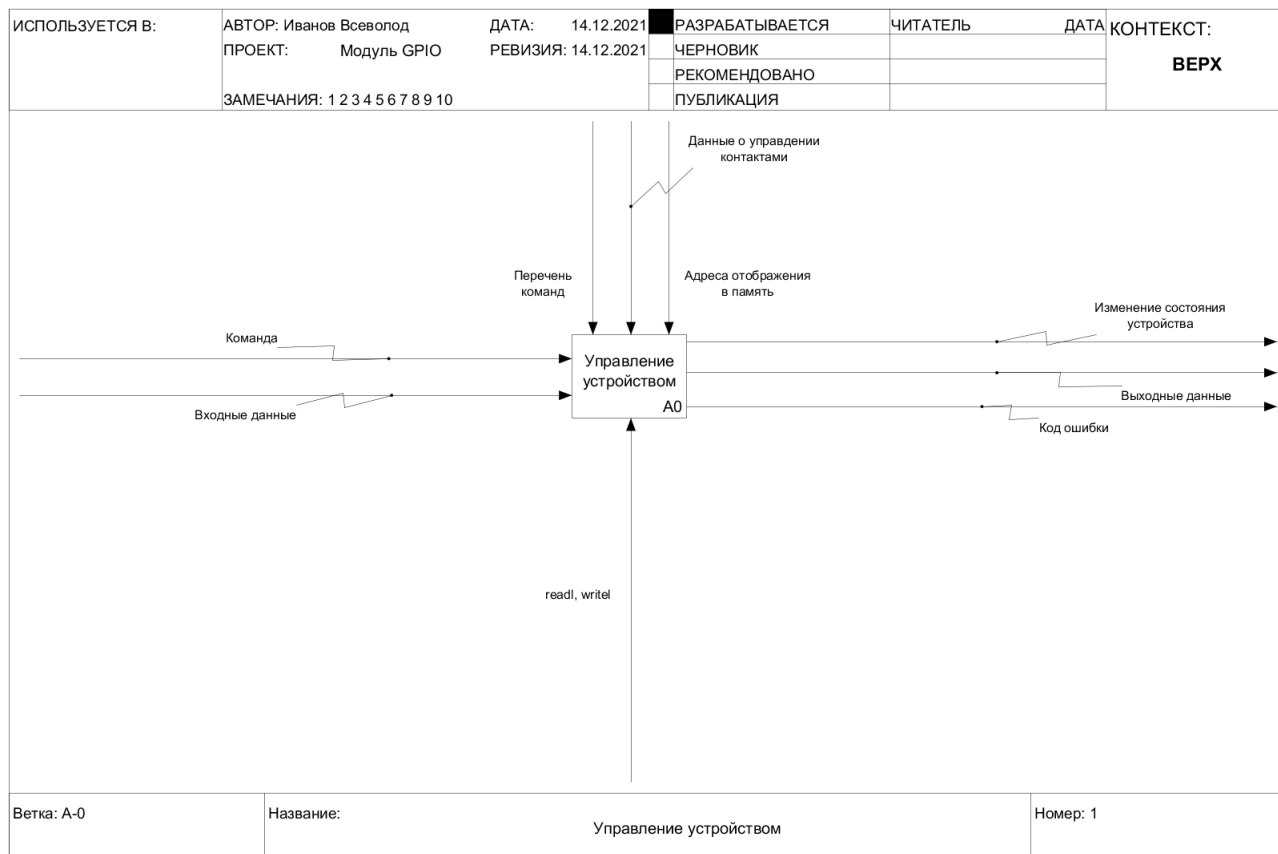


Рисунок 2 – Анализ входных и выходных данных

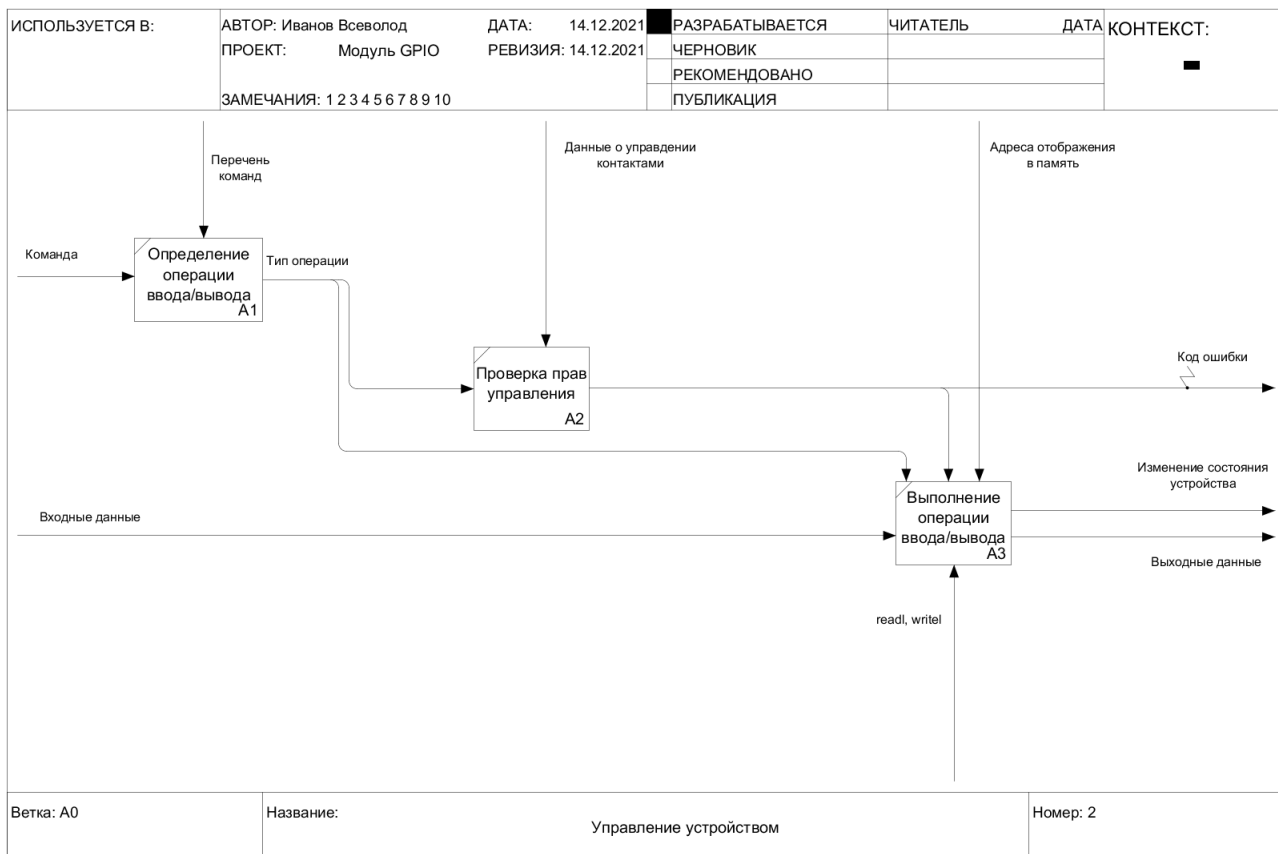


Рисунок 3 – Последовательность преобразований

2.2 Описание команд

В соответствии с выделенным в предыдущем разделе функционалом, в качестве необходимых для взаимодействия с устройством выделим команды, указанные в таблице 1 (с указанием их аргументов и возвращаемого значения):

Также для демонстрации работы данных команд необходимо разработать программу пользовательского уровня. Она должна предоставлять интерфейс для выбора команды, ввода её аргументов и отображения результата выполнения операции ввода/вывода.

Таблица 1 – Команды ввода/вывода

Команда	Аргументы	Возвращаемое значение
Чтение	№ пина	Значение пина
Запись	№ пина, значение	-
Переключение	№ пина	Значение пина
Установка / Сброс	№ пина	-
Захват владения	№ пина	-
Освобождение владения	№ пина	-
Установка режима	№ пина, режим	-

2.3 Алгоритм обработки команды

На рисунке 4 приведён алгоритм обработки команды ввода/вывода.

2.4 Алгоритмы чтения/записи значения

На рисунках 5, 6 и 7 приведены алгоритмы чтения, записи и переключения значения контакта соответственно.

2.5 Алгоритмы захвата/освобождения

На рисунках 8 и 9 приведены алгоритмы захвата и возвращения управления процессом контакта.

2.6 Алгоритмы установления режима работы

На рисунке 10 приведён алгоритм установления режима работы контакта.

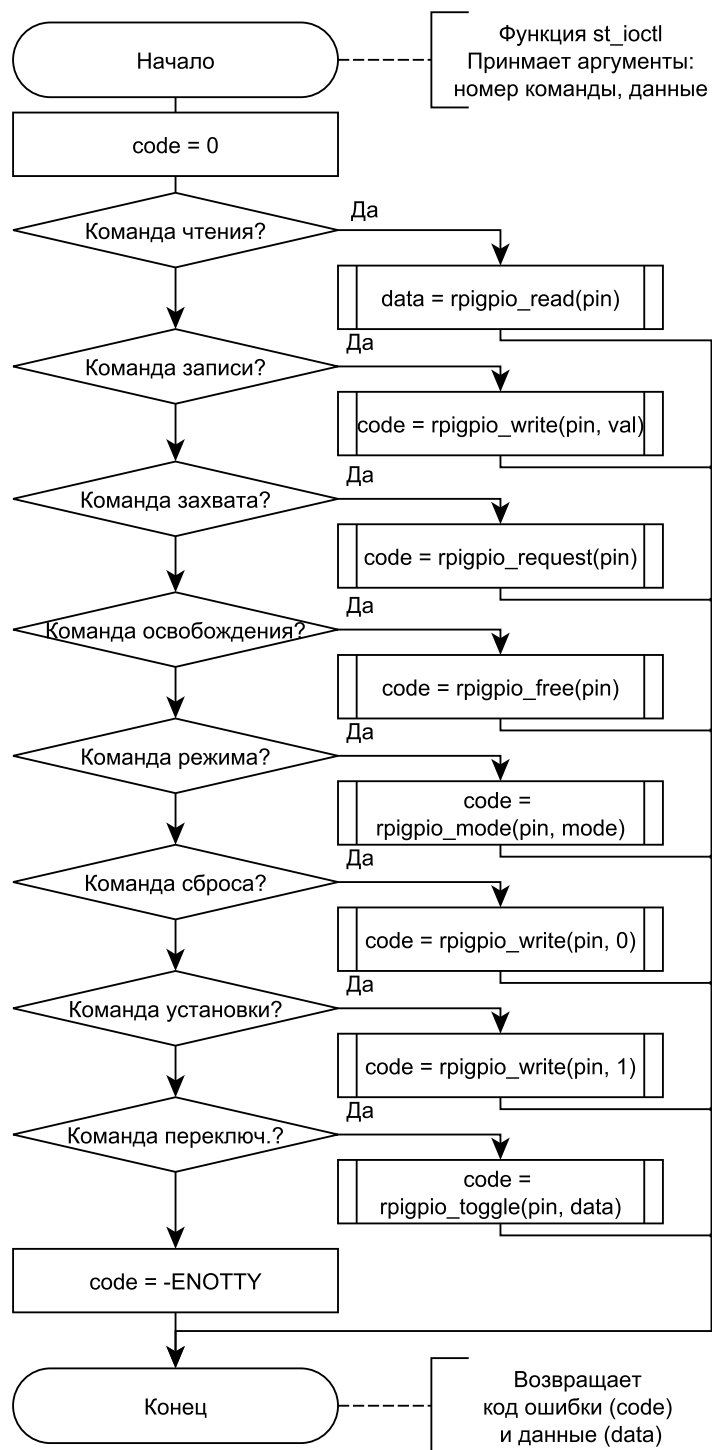


Рисунок 4 – Алгоритм обработки команды

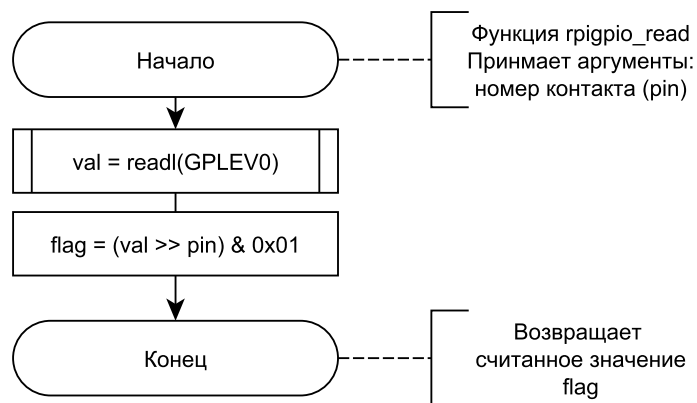


Рисунок 5 – Алгоритм чтения значения

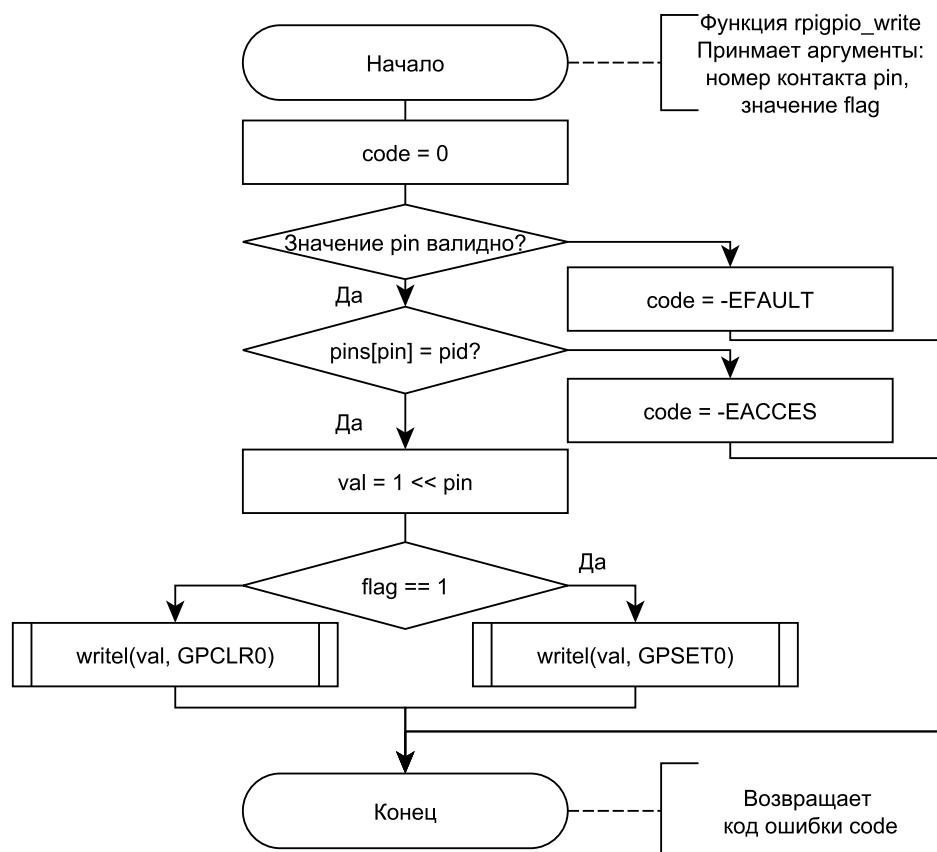


Рисунок 6 – Алгоритм установки значения

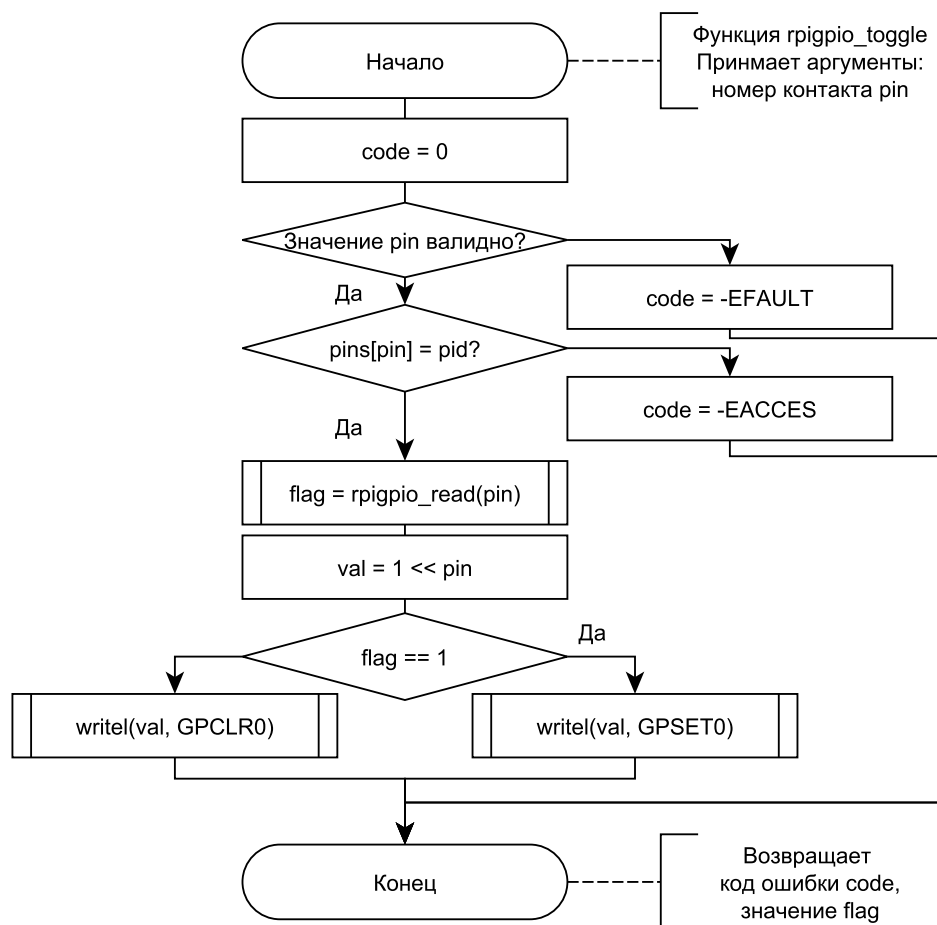


Рисунок 7 – Алгоритм переключения значения

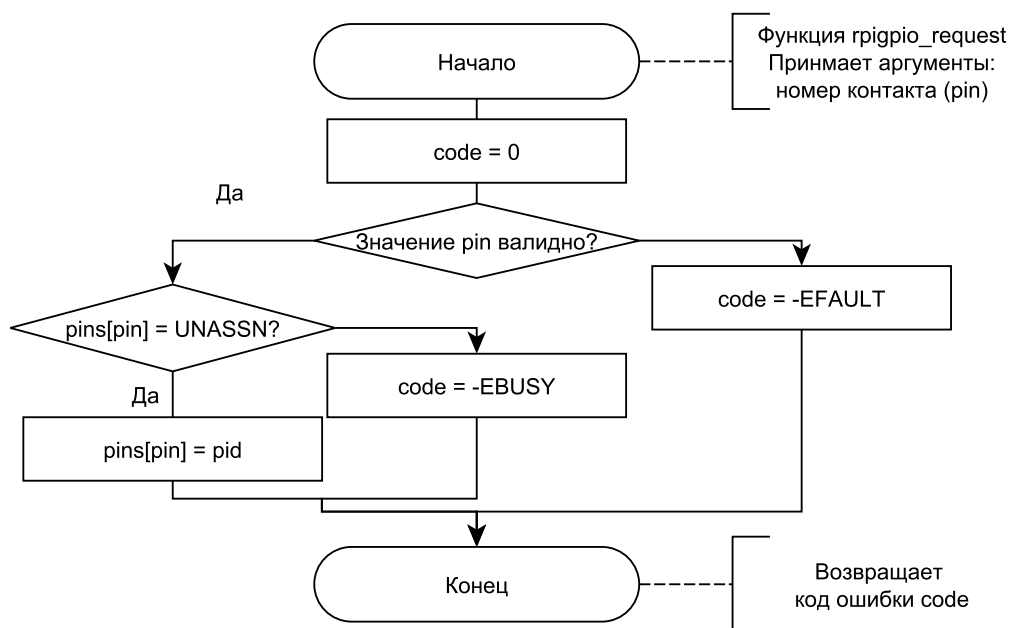


Рисунок 8 – Алгоритм захвата управления

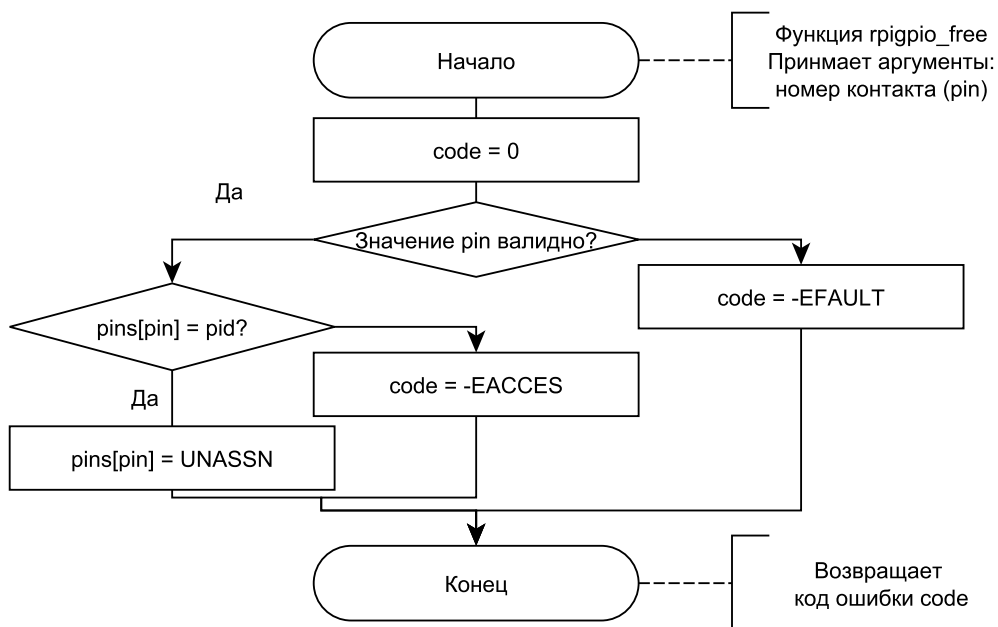


Рисунок 9 – Алгоритм возвращения управления

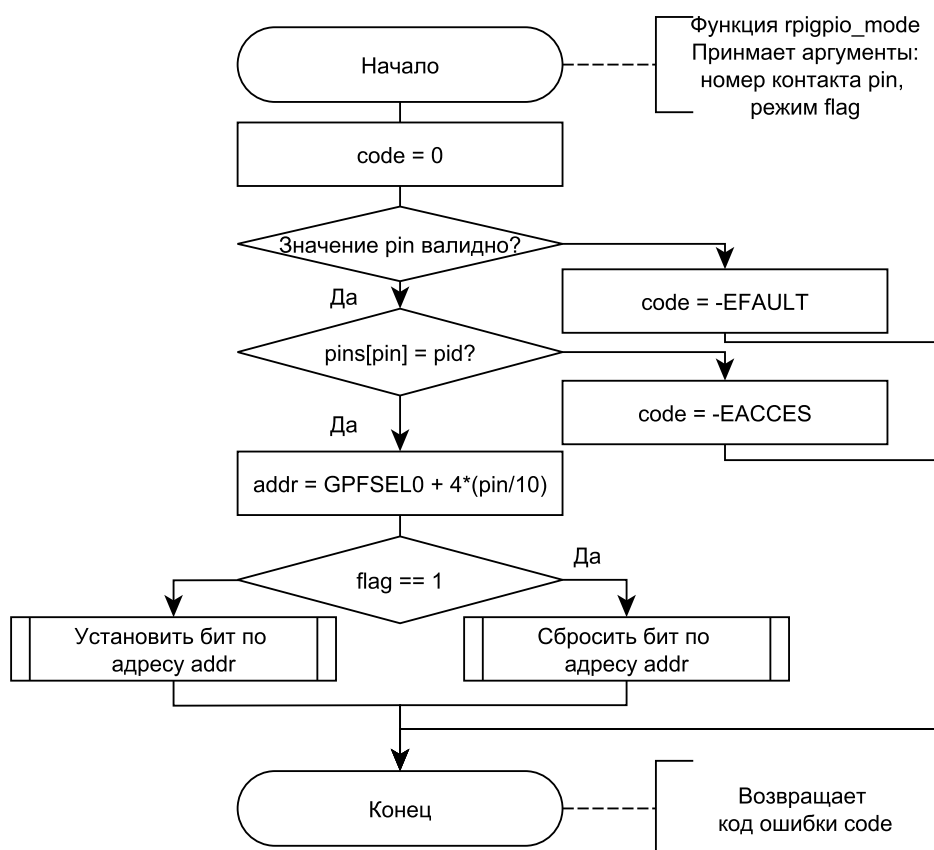


Рисунок 10 – Алгоритм установления режима работы

3 Технологическая часть

3.1 Выбор программных средств

В качестве языка программирования был выбран Си. Для сборки модуля использовалась утилита `make` и компилятор `gcc`.

Была выбрана среда разработки Visual Studio Code, так как в ней накоплен опыт работы и она обладает широким набором возможностей.

3.2 Структура загружаемого модуля

Реализованный загружаемый модуль включает в себя следующие функции:

- **`int __init rpigpio_init(void)`** – функция инициализации модуля;
- **`void __exit rpigpio_mcleanup(void)`** – функция выгрузки модуля;
- **`int st_open(struct inode*inode, struct file *filp)`** – функция открытия, описываемая в структуре `file_operations`;
- **`int st_release(struct inode *inode, struct file *filp)`** – функция закрытия, описываемая в структуре `file_operations`;
- **`long st_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)`** – функция ввода/вывода, описываемая в структуре `file_operations`;
- **`uint8_t rpigpio_read(int pin)`** – функция чтения значения контакта;
- **`long rpigpio_write(int pin, uint8_t val)`** – функция записи значения контакта;
- **`long rpigpio_toggle(int pin, uint8_t *flag)`** – функция переключения значения контакта;
- **`long rpigpio_request(int pin, int pid)`** – функция захвата управления над контактом;
- **`long rpigpio_free(int pin)`** – функция освобождения управления над контактом;
- **`long rpigpio_mode(int pin, PIN_MODE_t mode)`** – функция установления режима работы;

В Приложении А представлены листинги каждой из частей проекта.

3.3 Сборка и запуск модуля

Сборка модуля осуществляется командой `make` с использованием компилятора `gcc`. На листинге 8 приведено содержимое файла сборки.

Листинг 5: Makefile

```
1 obj-m += modgpio.o
2
3 CODE_DIR = client
4 .PHONY: client
5
6 all:
7 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
8
9 clean: bclean
10
11 bclean:
12 rm -f *.cmd *.o *.order *.mod.c *.ko
13
14 kclean:
15 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
16
17 client:
18 $(MAKE) -C $(CODE_DIR)
19
20 cclean:
21 $(MAKE) -C $(CODE_DIR) clean
22
23 install: all
24 sudo insmod modgpio.ko
25
26 uninstall:
27 -sudo rmmmod modgpio
28
29 reinstall: uninstall install
```

3.4 Функции ввода/вывода

На листинге 6 приведена реализация функции `st_ioctl`, являющейся точкой входа для .

Листинг 6: Функция ввода/вывода

```
1 static long st_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
2 {
3     int pin;
4     unsigned long ret, code;
5     uint8_t flag;
6     struct gpio_data_write wdata;    // write data
7     struct gpio_data_mode  mdata;    // mode data
8
9     switch (cmd) {
10         case GPIO_READ:
11             get_user(pin, (int __user *) arg);
12             flag = rpigpio_read(pin);
13             put_user(flag, (uint8_t __user *)arg);
14             return 0;
15
16         case GPIO_WRITE:
17             ret = copy_from_user(&wdata, (struct gpio_data_write __user *)arg
18 , sizeof(struct gpio_data_write));
19             if (ret != 0)
20             {
21                 printk(KERN_DEBUG "[WRITE] Error copying data from userspace\
22 n");
23                 return -EFAULT;
24             }
25             return rpigpio_write(wdata.pin, wdata.data);
26
27         case GPIO_REQUEST:
28             get_user (pin, (int __user *) arg);
29             return rpigpio_request(pin, current->pid);
30
31         case GPIO_FREE:
32             get_user (pin, (int __user *) arg);
33             return code = rpigpio_free(pin);
34     }
```

```

33     case GPIO_TOGGLE:
34         get_user (pin, (int __user *) arg);
35
36         if (!(code = rpigpio_toggle(pin, &flag)))
37             put_user(flag?0:1, (uint8_t __user *)arg);
38         return code;
39
40     case GPIO_MODE:
41         ret = copy_from_user(&mdata, (struct gpio_data_mode __user *)arg,
42 sizeof(struct gpio_data_mode));
43         if (ret != 0)
44         {
45             printk(KERN_DEBUG "[MODE] Error copying data from userspace\n
46 ");
47             return -EFAULT;
48         }
49
50         return rpigpio_mode(mdata.pin, mdata.data);
51
52     case GPIO_SET:
53         get_user (pin, (int __user *) arg);
54         printk(KERN_INFO "[SET] Pin: %d\n", pin);
55         return rpigpio_write(pin, 1);
56     case GPIO_CLR:
57         get_user (pin, (int __user *) arg);
58         printk(KERN_INFO "[CLR] Pin: %d\n", pin);
59         return rpigpio_write(pin, 0);
60
61     default:
62         return -ENOTTY;
63 }

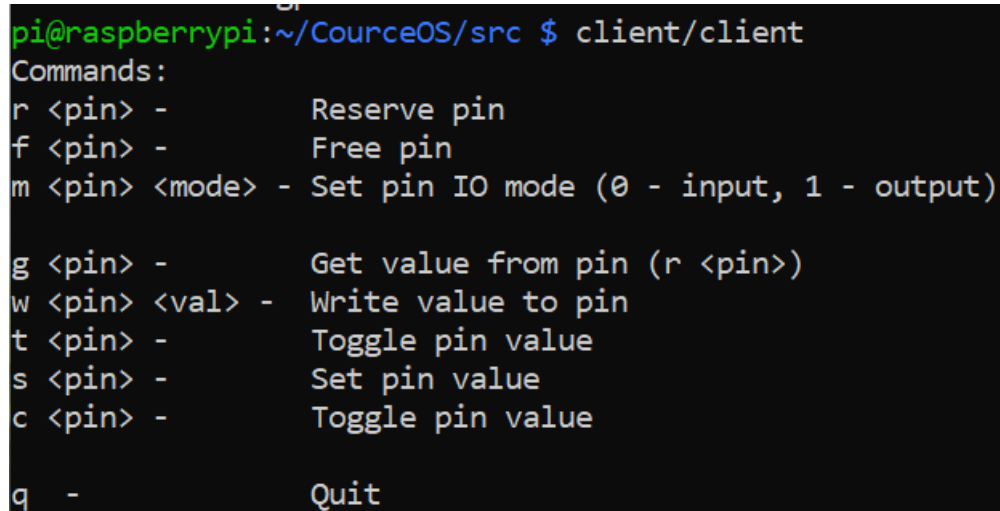
```

Вывод

Были выбраны технические средства реализации, описана общая структура программы.

4 Исследовательская часть

Для взаимодействия с загружаемым модулем был разработана программа пользовательского режима. Её интерфейс представлен на рисунке 11.



```
pi@raspberrypi:~/CourceOS/src $ client/client
Commands:
r <pin> - Reserve pin
f <pin> - Free pin
m <pin> <mode> - Set pin IO mode (0 - input, 1 - output)

g <pin> - Get value from pin (r <pin>)
w <pin> <val> - Write value to pin
t <pin> - Toggle pin value
s <pin> - Set pin value
c <pin> - Toggle pin value

q - Quit
```

Рисунок 11 – Интерфейс программы

Для тестирования работы программы к компьютеру были подключены реле и кнопка (номера контактов 17 и 22 соответственно).

Первым тестом является изменение состояния реле 12. В результате теста значение было успешно изменено - при установке логической единицы на 17-м контакте реле было открыто.

Вторым тестом является проверка изменения значения при нажатии кнопки 13. В результате теста считываемое значение было изменено во время зажатия кнопки.

Третьим тестом является проверка контроля захвата управления контактами 14. Действия выполняются сначала в правом терминале, потом в левом. Как можно видеть, операция чтения позволена без захвата контроля. При попытке второго процесса захватить управление или совершить операцию вывода на уже используемый пин, ему возвращается ошибка.

```

> r 17
Reserved pin 17
> w 17 0
Pin 17 value=17
Wrote 0 to pin 17
> g 17
Pin 17 value=0
> t 17
Toggled pin 17 to 1
> g 17
Pin 17 value=1

```

Рисунок 12 – Тест вывода

```

> g 22
Pin 22 value=0
> g 22
Pin 22 value=1
> g 22
Pin 22 value=0

```

Рисунок 13 – Тест ввода

```

> g 17
Pin 17 value=0
> r 17
Reserved pin 17
> t 17
Toggled pin 17 to 1
> f 17
Freed pin 17
>

```

```

> g 17
Pin 17 value=0
> r 17
ioctl: Device or resource busy
> t 17
ioctl: Permission denied
> r 17
Reserved pin 17
> t 17
Toggled pin 17 to 0
>

```

Рисунок 14 – Тест контроля захвата и возвращения управления

ЗАКЛЮЧЕНИЕ

В ходе проделанной работы были проанализированы особенности интерфейса GPIO. Также был определён формат передаваемых данных: описаны участки памяти в адресном пространстве ядра, используемые для отображения устройства в память, их назначение и метод чтения/записи информации. В качестве средства передачи информации между модулем ядра и пользовательскими процессами выбрано символьное устройство. Реализован загружаемый модуль ядра, выполняющего управление над устройствами, подключенными к компьютеру по интерфейсу GPIO. Созданное программное обеспечение соответствует заявленным требованиям и, как было проверено в процессе тестирования, корректно выполняет свои функции.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. GPIO [Электронный ресурс] Режим доступа: <http://ru.wikipedia.org/wiki/GPIO> (дата обращения 10.12.2021).
2. Знакомство с GPIO в Raspberry Pi Режим доступа: <https://ph0en1x.net/86-raspberry-pi-znakomstvo-s-gpio-perekluchatel-i-svetodiod.htmlgpio-header-layout-pins> (дата обращения 10.12.2021).
3. pinctrl-bcm2835.c Режим доступа: <https://elixir.bootlin.com/linux/latest/source/drivers/pinctrl/bcm2835.cL49> (дата обращения 10.12.2021).
4. Использование памяти ввода/вывода Режим доступа: http://dmilvdv.narod.ru/Translate/LDD3/ldd_using_io_memory.html (дата обращения 10.12.2021).

ПРИЛОЖЕНИЕ А

Листинг 7: modgpio.h

```
1 #ifndef __RPI_GPIO_H__
2 #define __RPI_GPIO_H__
3 //magical IOCTL number
4 #define GPIO_IOC_MAGIC 'k'
5
6 typedef enum {MODE_INPUT=0, MODE_OUTPUT} PIN_MODE_t;
7
8 struct gpio_data_write {
9     int pin;
10    char data;
11 };
12
13 struct gpio_data_mode {
14     int pin;
15     PIN_MODE_t data;
16 };
17
18
19 /// GPIO MEMORY
20 #define BCM2708_PERI_BASE    0x3F000000
21 #define GPIO_BASE            (BCM2708_PERI_BASE + 0x200000)
22 // memory locations defines
23 #define GPFSEL0              0x00
24 #define GPSET0               0x1C
25 #define GPCLR0               0x28
26 #define GPLEV0               0x34
27
28 /// PIN ROLES
29 #define RELAIS_PIN    17
30 #define BUTTON_PIN    22
31
32 /// DEFINES OF ioctl
33 //in: pin to read    //out: value           //the value read on the pin
34 #define GPIO_READ    _IOWR(GPIO_IOC_MAGIC, 0x90, int)
35 //in: struct(pin, data)    //out: NONE
```

```

36 #define GPIO_WRITE                _IOW(GPIO_IOC_MAGIC, 0x91, struct gpio_data_write
    )
37 //in: pin to request              //out: success/fail      // request exclusive
    modify privileges
38 #define GPIO_REQUEST              _IOW(GPIO_IOC_MAGIC, 0x92, int)
39 //in: pin to free
40 #define GPIO_FREE                 _IOW(GPIO_IOC_MAGIC, 0x93, int)
41 //in: pin to toggle              //out: new value
42 #define GPIO_TOGGLE              _IOWR(GPIO_IOC_MAGIC, 0x94, int)
43 //in: struct (pin, mode[i/o])
44 #define GPIO_MODE                 _IOW(GPIO_IOC_MAGIC, 0x95, struct gpio_data_mode)
45 //in: pin to set //set the pin (same as write 1)
46 #define GPIO_SET                  _IOW(GPIO_IOC_MAGIC, 0x96, int)
47 //in: pin to clear //clear the pin (same as write 0)
48 #define GPIO_CLR                  _IOW(GPIO_IOC_MAGIC, 0x97, int)
49
50
51 #endif //__RPI_GPIO_H__

```

Листинг 8: modgpio.c

```

1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/stat.h>
5 #include <linux/device.h>
6 #include <linux/fs.h>
7 #include <linux/err.h>
8 #include <linux/semaphore.h>
9 #include <linux/gpio.h>
10 #include <linux/interrupt.h>
11 #include <linux/ioctl.h>
12 #include <linux/io.h>
13 #include <linux/sched.h>
14
15 #include "modgpio.h"
16
17 #define IO_OFFSET                  0x00
18 #define __IO_ADDRESS(x)           ((x) + IO_OFFSET)
19 #define IO_ADDRESS(pa)            IOMEM(__IO_ADDRESS(pa))
20 #define __io_address(n)           ((void __iomem *)IO_ADDRESS(n))

```

```

21
22
23 #define RPIGPIO_MOD_AUTH      "Ivanov Vsedolod"
24 #define RPIGPIO_MOD_DESC      "OS source work (GPIO access control for
    Raspberry Pi)"
25 #define RPIGPIO_MOD_SDEV      "RPiGPIO"
26 #define MOD_NAME               "rpi gpio"
27
28 #define PIN_NULL_PID          1    // invalid pin
29 #define PIN_UNASSN            0    // pin available
30 #define PIN_ARRAY_LEN        32
31
32 struct gpiomod_data {
33     int irq;
34     int mjr;
35     struct class *cls;
36     void __iomem *regs;
37     spinlock_t lock;
38     uint32_t pins[PIN_ARRAY_LEN];
39 };
40
41 static struct gpiomod_data std = {
42     .mjr = 0,
43     .cls = NULL,
44     .regs = NULL,
45     .pins = {
46         PIN_NULL_PID,
47         PIN_NULL_PID,
48         PIN_UNASSN,
49         PIN_UNASSN,
50         PIN_UNASSN,
51         PIN_NULL_PID,
52         PIN_NULL_PID,
53         PIN_UNASSN,
54         PIN_UNASSN,
55         PIN_UNASSN,
56         PIN_UNASSN,
57         PIN_UNASSN,
58         PIN_NULL_PID,
59         PIN_NULL_PID,

```

```

60     PIN_UNASSN,
61     PIN_UNASSN,
62     PIN_NULL_PID,
63     PIN_UNASSN,
64     PIN_UNASSN,
65     PIN_NULL_PID,
66     PIN_NULL_PID,
67     PIN_NULL_PID,
68     PIN_UNASSN,
69     PIN_UNASSN,
70     PIN_UNASSN,
71     PIN_UNASSN,
72     PIN_NULL_PID,
73     PIN_UNASSN,
74     PIN_UNASSN,
75     PIN_UNASSN,
76     PIN_UNASSN,
77     PIN_UNASSN,
78 }
79 };
80
81
82 static int st_open(struct inode*inode, struct file *filp)
83 {
84     return 0;
85 }
86
87 static int st_release(struct inode *inode, struct file *filp)
88 {
89     int i;
90
91     spin_lock(&std.lock);
92     for (i = 0; i < PIN_ARRAY_LEN; i++) {
93         if (std.pins[i] == current->pid) {
94             printk(KERN_DEBUG "[FREE] Pin:%d From:%d\n", i, current->pid);
95             std.pins[i] = PIN_UNASSN;
96         }
97     }
98     spin_unlock(&std.lock);
99     return 0;

```

```

100 }
101
102 /// READ/WRITE FUNCTIONS
103 static uint8_t rpigpio_read(int pin)
104 {
105     uint32_t val;
106     uint8_t flag;
107
108     val = readl(__io_address(std.regs + GPLEV0));
109     flag = val >> pin;
110     flag &= 0x01;
111     printk(KERN_DEBUG "[READ] Pin: %d Val:%d\n", pin, flag);
112
113     return flag;
114 }
115
116 static long rpigpio_write(int pin, uint8_t val)
117 {
118     spin_lock(&std.lock);
119     if (pin > PIN_ARRAY_LEN || pin < 0 || std.pins[pin] == PIN_NULL_PID) {
120         // validate pins
121         spin_unlock(&std.lock);
122         return -EFAULT; // bad request
123     } else if (std.pins[pin] != current->pid) {
124         spin_unlock(&std.lock);
125         return -EACCES; // pin reserved by another process
126     }
127     spin_unlock(&std.lock);
128
129     printk(KERN_INFO "[WRITE] Pin: %d Val:%d\n", pin, val);
130     if (val)
131         writel(1 << pin, __io_address(std.regs + GPSET0)); // set
132     else
133         writel(1 << pin, __io_address(std.regs + GPCLR0)); // clear
134
135     return 0;
136 }
137
138 static long rpigpio_toggle(int pin, uint8_t *flag)
139 {

```

```

139     spin_lock(&std.lock);
140     if (pin > PIN_ARRAY_LEN || pin < 0 || std.pins[pin] == PIN_NULL_PID) { //
141         validate pins
142         spin_unlock(&std.lock);
143         return -EFAULT; // bad request
144     } else if (std.pins[pin] != current->pid) {
145         spin_unlock(&std.lock);
146         return -EACCES; // permission denied
147     }
148     spin_unlock(&std.lock);
149
150     *flag = rpigpio_read(pin);
151     if (*flag)
152         writel(1 << pin, __io_address(std.regs + GPCLR0)); // clear
153     else
154         writel(1 << pin, __io_address(std.regs + GPSET0)); // set
155     printk(KERN_DEBUG "[TOGGLE] Pin:%d %.1d -> %.1d\n", pin, *flag, *flag
156             ?0:1);
157     return 0;
158 }
159
160 /// OWN PIN FUNCTIONS
161 static long rpigpio_request(int pin, int pid)
162 {
163     spin_lock(&std.lock);
164     // validate pins
165     if (pin > PIN_ARRAY_LEN || pin < 0 || std.pins[pin] == PIN_NULL_PID) {
166         spin_unlock(&std.lock);
167         return -EFAULT; // bad request
168     } else if (std.pins[pin] != PIN_UNASSN) {
169         spin_unlock(&std.lock);
170         return -EBUSY; // pin already reserved
171     }
172
173     std.pins[pin] = pid;
174     spin_unlock(&std.lock);
175
176     printk(KERN_DEBUG "[REQUEST] Pin:%d Assigned To:%d\n", pin, pid);
177     return 0;
178 }

```

```

177
178 static long rpigpio_free(int pin)
179 {
180     spin_lock(&std.lock);
181     //validate pins
182     if (pin > PIN_ARRAY_LEN || pin < 0 || std.pins[pin] == PIN_NULL_PID) {
183         spin_unlock(&std.lock);
184         return -EFAULT; // bad request
185     } else if (std.pins[pin] != current->pid) {
186         spin_unlock(&std.lock);
187         return -EACCES; // pin reserved by another process
188     }
189     std.pins[pin] = PIN_UNASSN;
190     spin_unlock(&std.lock);
191
192     printk(KERN_DEBUG "[FREE] Pin:%d From:%d\n", pin, current->pid);
193     return 0;
194 }
195
196 /// MODE FUNCTIONS
197 static long rpigpio_mode(int pin, PIN_MODE_t mode)
198 {
199     spin_lock(&std.lock);
200     // validate pin
201     if (pin > PIN_ARRAY_LEN || pin < 0 || std.pins[pin] == PIN_NULL_PID) {
202         spin_unlock(&std.lock);
203         return -EFAULT; // bad request
204     } else if (std.pins[pin] != current->pid) {
205         spin_unlock(&std.lock);
206         return -EACCES; // permission denied
207     }
208     spin_unlock(&std.lock);
209
210     // clear the bits (sets to input)
211     writel(~(7<<((pin %10)*3)) & readl(__io_address(std.regs + GPFSEL0 + (0
x04)*(pin /10))),
212     __io_address(std.regs + GPFSEL0 + (0x04)*(pin/10)));
213
214     switch (mode)
215     {

```



```

216         case MODE_INPUT:
217             printk(KERN_DEBUG "[MODE] Pin %d set as Input\n", pin);
218             break;
219
220         case MODE_OUTPUT:
221             writel(1<<((pin % 10)*3) | readl(__io_address(std.regs + GPFSEL0 + (0
x04)*(pin/10))),
222                 __io_address(std.regs + GPFSEL0 + (0x04)*(pin/10)));    // Set pin as
output
223             printk(KERN_DEBUG "[MODE] Pin %d set as Output\n", pin);
224             break;
225
226         default:
227             return -EINVAL;
228     }
229
230     return 0;
231 }
232
233
234 static long st_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
235 {
236     int pin;                //used in read, request, free
237     unsigned long ret, code;
238     uint8_t flag;
239     struct gpio_data_write wdata;    // write data
240     struct gpio_data_mode  mdata;    // mode data
241
242     switch (cmd) {
243         case GPIO_READ:
244             get_user(pin, (int __user *) arg);
245             flag = rpigpio_read(pin);
246             put_user(flag, (uint8_t __user *)arg);
247             return 0;
248
249         case GPIO_WRITE:
250             ret = copy_from_user(&wdata, (struct gpio_data_write __user *)arg,
sizeof(struct gpio_data_write));
251             if (ret != 0) {
252                 printk(KERN_DEBUG "[WRITE] Error copying data from userspace\n");

```

```

253         return -EFAULT;
254     }
255     return rpigpio_write(wdata.pin, wdata.data);
256
257     case GPIO_REQUEST:
258         get_user (pin, (int __user *) arg);
259         return rpigpio_request(pin, current->pid);
260
261     case GPIO_FREE:
262         get_user (pin, (int __user *) arg);
263         return code = rpigpio_free(pin);
264
265     case GPIO_TOGGLE:
266         get_user (pin, (int __user *) arg);
267
268         if (!(code = rpigpio_toggle(pin, &flag)))
269             put_user(flag?0:1, (uint8_t __user *)arg);
270         return code;
271
272     case GPIO_MODE:
273         ret = copy_from_user(&mdata, (struct gpio_data_mode __user *)arg,
274             sizeof(struct gpio_data_mode));
275         if (ret != 0) {
276             printk(KERN_DEBUG "[MODE] Error copying data from userspace\n");
277             return -EFAULT;
278         }
279         return rpigpio_mode(mdata.pin, mdata.data);
280
281     case GPIO_SET:
282         get_user (pin, (int __user *) arg);
283         printk(KERN_INFO "[SET] Pin: %d\n", pin);
284         return rpigpio_write(pin, 1);
285     case GPIO_CLR:
286         get_user (pin, (int __user *) arg);
287         printk(KERN_INFO "[CLR] Pin: %d\n", pin);
288         return rpigpio_write(pin, 0);
289
290     default:
291         return -ENOTTY;

```

```

292     }
293 }
294
295 static char *st_devnode(struct device *dev, umode_t *mode)
296 {
297     if (mode) *mode = 0666;
298     return NULL;
299 }
300
301
302 static const struct file_operations gpio_fops = {
303     .owner          = THIS_MODULE,
304     .open           = st_open,
305     .release        = st_release,
306     .unlocked_ioctl = st_ioctl,
307 };
308
309 static irqreturn_t button_int (int irq, void *dev_id)
310 {
311     uint8_t flag;
312     printk(KERN_DEBUG "[IRQ] %d\n", irq);
313     rpigpio_toggle(RELAIS_PIN, &flag);
314     return IRQ_HANDLED;
315 }
316
317 static int __init rpigpio_init(void)
318 {
319     struct device *dev;
320
321     printk(KERN_INFO "[GRIO] Startup\n");
322     spin_lock_init(&(std.lock));
323
324     std.mjr = register_chrdev(0, MOD_NAME, &gpio_fops);
325     if (std.mjr < 0) {
326         printk(KERN_ALERT "[GRIO] Cannot Register");
327         return std.mjr;
328     }
329     printk(KERN_INFO "[GRIO] Major #%d\n", std.mjr);
330
331     std.cls = class_create(THIS_MODULE, "std.cls");

```

```

332     if (IS_ERR(std.cls)) {
333         printk(KERN_ALERT "[GPIO] Cannot get class\n");
334         unregister_chrdev(std.mjr, MOD_NAME);
335         return PTR_ERR(std.cls);
336     }
337     std.cls->devnode = st_devnode;
338
339     dev = device_create(std.cls, NULL, MKDEV(std.mjr, 0), (void*)&std,
MOD_NAME);
340     if (IS_ERR(dev)) {
341         printk(KERN_ALERT "[GPIO] Cannot create device\n");
342         class_destroy(std.cls);
343         unregister_chrdev(std.mjr, MOD_NAME);
344         return PTR_ERR(dev);
345     }
346
347     release_mem_region(GPIO_BASE, 0xb4);
348     if(!request_mem_region(GPIO_BASE, 0x40, MOD_NAME))
349     {
350         unregister_chrdev(std.mjr, MOD_NAME);
351         return -ENODEV;
352     }
353     std.regs = ioremap(GPIO_BASE, 0x40);
354
355     printk(KERN_INFO "[GPIO] %s loaded\n", MOD_NAME);
356
357     std.irq = gpio_to_irq(BUTTON_PIN);
358     if (!rpigpio_mode(BUTTON_PIN, MODE_INPUT) ||
359         !rpigpio_mode(RELAIS_PIN, MODE_OUTPUT) ||
360         request_irq(std.irq, button_int, IRQF_TRIGGER_RISING, "button_int", &std.
mjr))
361     {
362         unregister_chrdev(std.mjr, MOD_NAME);
363         return -1;
364     }
365
366     return 0;
367 }
368
369 static void __exit rpigpio_mcleanup(void)

```

```

370 {
371     synchronize_irq(std.irq);
372     free_irq(std.irq, &std.mjr);
373
374     iounmap(std.regs);
375     release_mem_region(GPIO_BASE, 0x40);
376     device_destroy(std.cls, MKDEV(std.mjr, 0));
377     class_destroy(std.cls);
378     unregister_chrdev(std.mjr, MOD_NAME);
379
380     printk(KERN_NOTICE "[GRIO] %s removed\n", MOD_NAME);
381 }
382
383 module_init(rpigpio_init);
384 module_exit(rpigpio_mcleanup);
385
386 MODULE_LICENSE("GPL");
387 MODULE_AUTHOR(RPIGPIO_MOD_AUTH);
388 MODULE_DESCRIPTION(RPIGPIO_MOD_DESC);
389 MODULE_SUPPORTED_DEVICE(RPIGPIO_MOD_SDEV);

```

Листинг 9: gpioctlent.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <sys/ioctl.h>
9 #include <stdint.h>
10 #include <readline/readline.h>
11 #include <readline/history.h>
12 #include "modgprio.h"
13
14 int main(int argc, char * argv[])
15 {
16     int fd;
17     int ret;
18     int v=0;

```

```

19     int pin = 17;
20     char* buf = NULL;
21     char* found = NULL;
22     struct gpio_data_write mydwstruct;
23     struct gpio_data_mode mydmstruct;
24
25     using_history();
26
27     fd = open("/dev/rpiggpio", O_RDWR);
28     if(!fd) {
29         perror("open(O_RDONLY)");
30         return errno;
31     }
32
33     printf("Commands:\n");
34     printf("r <pin> - \t Reserve pin\n");
35     printf("f <pin> - \t Free pin\n");
36     printf("m <pin> <mode> - Set pin IO mode (0 - input, 1 - output)\n");
37     printf("\n");
38     printf("g <pin> - \t Get value from pin (r <pin>)\n");
39     printf("w <pin> <val> - Write value to pin\n");
40     printf("t <pin> - \t Toggle pin value\n");
41     printf("s <pin> - \t Set pin value\n");
42     printf("c <pin> - \t Toggle pin value\n");
43     printf("\n");
44     printf("q - \t\t Quit\n");
45     printf("\n");
46
47     while (1) {
48         if (buf != NULL)
49             free(buf);
50
51         buf = readline("> ");
52         if (!buf) {
53             break;
54         }
55         add_history(buf);
56
57         //accept commands without a space between ctrl char and value
58         v=atoi(&buf[2]);

```

```

59
60 // Action based on input
61 switch (buf[0])
62 {
63     case 'g':
64     case 'G':
65         pin = v;
66         ret = ioctl(fd, GPIO_READ, &pin);
67         if (ret < 0) {
68             perror("ioctl");
69         }
70         printf("Pin %d value=%d\n", v, pin);
71         break;
72
73     case 's':
74     case 'S':
75         pin = v;
76         ret = ioctl(fd, GPIO_SET, &pin);
77         if (ret < 0)
78             perror("ioctl");
79         else
80             printf("Set pin %d\n", pin);
81         break;
82
83     case 'c':
84     case 'C':
85         pin = v;
86         ret = ioctl(fd, GPIO_CLR, &pin);
87         if (ret < 0)
88             perror("ioctl");
89         else
90             printf("Clear pin %d\n", pin);
91         break;
92
93     case 'w':
94     case 'W':
95         found = strstr(&buf[3], " ");
96         if (found == NULL) {
97             printf("Missing 2nd parameter. Usage \"%w <pin> <val>\\\"\\n\"");
98             continue;

```

```

99         }
100
101         mydwstruct.pin = v;
102         mydwstruct.data = atoi(found);
103         printf("Pin %d value=%d\n", pin, v);
104
105         ret = ioctl(fd, GPIO_WRITE, (unsigned long)&mydwstruct);
106
107         if (ret < 0)
108             perror("ioctl");
109         else
110             printf("Wrote %d to pin %d\n", mydwstruct.data, mydwstruct.pin);
111         break;
112
113         case 'r':
114         case 'R':
115             pin = v;
116             ret = ioctl(fd, GPIO_REQUEST, &pin);
117             if (ret < 0)
118                 perror("ioctl");
119             else
120                 printf("Reserved pin %d\n", pin);
121             break;
122
123         case 'f':
124         case 'F':
125             pin = v;
126             ret = ioctl(fd, GPIO_FREE, &pin);
127             if (ret < 0)
128                 perror("ioctl");
129             else
130                 printf("Freed pin %d\n", pin);
131             break;
132
133         case 't':
134         case 'T':
135             pin = v;
136             ret = ioctl(fd, GPIO_TOGGLE, &pin);
137             if (ret < 0)
138                 perror("ioctl");

```



```

139         else
140             printf("Toggled pin %d to %d\n", v, pin);
141             break;
142
143         case 'm':
144         case 'M':
145             found = strstr(&buf[3], " ");
146             if (found == NULL) {
147                 printf("Missing 2nd parameter. Usage \"m <val> <pin>\"\n");
148                 continue;
149             }
150
151             mydmstruct.pin = v;
152             mydmstruct.data = atoi(found)?MODE_OUTPUT:MODE_INPUT;
153
154             ret = ioctl(fd, GPIO_MODE, &mydmstruct);
155             if (ret < 0)
156                 perror("ioctl");
157             else
158                 printf("Set pin %d as %s\n", mydmstruct.pin, mydmstruct.data?"
OUTPUT":"INPUT");
159             break;
160
161         case 'q':
162         case 'Q':
163             break;
164         default:
165             printf("Unknown Command\n");
166     }
167
168     if ((buf[0]=='q') || (buf[0]=='Q')) {
169         break;
170     }
171 }
172 clear_history();
173 printf("\n");
174 close(fd);
175 return 0;
176 }

```