Санкт-Петербургский политехнический университет Петра Великого
Кафедра компьютерных систем и программных технологий

Курсовой проект по курсу
«Проектирование ОС и их компонентов»

«Разработка демонов поддержки периферийных устройств»

Работу выполнил студент группы №: 13541/3 Чеботарёв М. М.
Работу принял преподаватель: _____ Душутина Е. В.

Санкт-Петербург
2017 г.

Оглавление

Используемая система и версия ядра

а) Windows

```
Процессор:              Intel(R) Core(TM) i5-2450 CPU @2.50GHz 2.50GHz
ОЗУ:                    8,00 Гб
Тип системы:            Windows 7 Ultimate Compact (2009) Service Pack 1. x64.
```

б) Linux

```
michael@michael-LIFEBOOK-AH531:~$ lsb_release  -a
No LSB modules are available.
Distributor ID:         Ubuntu
Description:            Ubuntu 16.04.1 LTS
Release:                16.04
Codename:               xenial


michael@michael-LIFEBOOK-AH531:~$ cat /proc/version
Linux version 4.4.0-38-generic (buildd@lgw01-58) (gcc version 5.4.0 20160609 (Ubuntu
5.4.0-6ubuntu1~16.04.2) ) #57-Ubuntu SMP Tue Sep 6 15:42:33 UTC 2016
```

1. ПО смартфона XXon

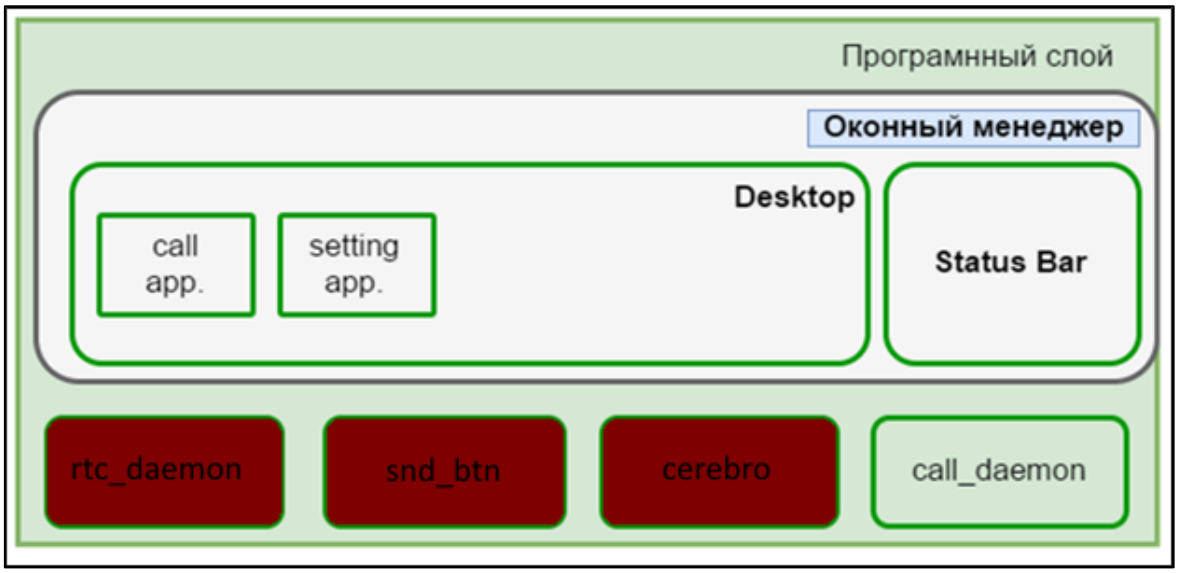

Рисунок 1. Суммарная схема разработанного ПО

Из разработанного программного обеспечения смартфона (рис.1) можно выделить следующие компоненты:

- **демоны для контроля периферии (rtc_daemon, snd_btn, cerebro);**
- демон для контроля поступивших звонков;
- оконный менеджер;
- рабочий стол, созданный в собственном оконном менеджере;
- статус бар, созданный в собственном оконном менеджере;

Курсовая работа описывает проектирование и разработку демонов контроля периферии (rtc_daemon, snd_btn, cerebro).

2. Часы реального времени (Real Time Clock)

2.1. Аппаратное подключение

Подключаем устройство в соответствии с таблицей 1.

Таблица 1. Подключение RTC-модуля (рис.2) и RaspBerry Pi*

RTC-модуль DS3231 [2]	Pi GPIO
GND	P1-06 (GND)
VCC	P1-01 (3.3V)
SDA	P1-03 (I2C SDA)
SCL	P1-05 (I2C SCL)

Рисунок 2. RTC-модуль DS3231

***Примечание*:** на момент написания курсовой все, вышедшие миникомпьютеры (RPi 1,2,3 и Zero [3]), имели одинаковую распиновку (pinout). То есть описанное подключение RTC-модуля подходит для любой из плат.

2.2. Установка зависимостей и настройка системы

Сначала необходимо **проверить/выполнить обновление системы**:

```
sudo apt-get update
sudo apt-get -y upgrade
```

Примечание: флаг **"-y"** отвечает на все вопрос об установке "yes" (если у модуля не проверен сертификат и т.п.).

Устанавливаем средства для работы с I2C шиной:

```
sudo apt-get install i2c-tools
```

Выполняем команду:

```
sudo nano /etc/modules
```

И **добавляем модули,** приведенные ниже, в конец файла:

```
snd-bcm2835
i2c-bcm2835
i2c-dev
rtc-ds1307
```

В результате файл «/etc/modules» должен содержать приблизительно следующее:

Листинг 1. Файл "/etc/modules"

```
# /etc/modules: kernel modules to load at boot time.
#
# This file contains the names of kernel modules that should be loaded
# at boot time, one per line. Lines beginning with "#" are ignored.

snd-bcm2835
i2c-bcm2835
i2c-dev
```

```
rtc-ds1307
```

Примечание: драйвер **rtc-ds1307** полностью совместим с устройством DS3231.

Для активация программного доступа к I2C-шине в самой системе (используется Raspbian) используем команду:

```
sudo raspi-config
```

В открывшемся меню проходим по следующей цепочке:

```
“Intefacing Option” -> “I2C” -> “Enable? Yes”.
```

Затем выбираем «**Finish**» и перезагружаемся (sudo reboot)

2.3. Проверка корректности настроек и подключения

2.3.1. Проверка подключения

После завершения загрузки системы проверяем подключение устройств на I2C-шине. Для этого выполняем команду:

```
sudo i2cdetect -y 1
```

Примечание: флаг -y указывает I2C шину. Обычно это 0, 1 или 2 шина.
В данном случае RPi предоставляет первую I2C-шину.

Результат исполнения:

```
pi@raspberrypi:~ $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

0x68 – это статический шестнадцатеричный адрес RTC [4] таймера на i2c-шине. На используемом таймере ZS-042 возможно изменение i2c-адреса посредствам физического соединения (пайки) перемычек A0, A1 и/или A2. В таком случае адресе варьируются 3 младших бита.
0x57 – это статический шестнадцатеричный i2c-адрес AT24C32-блока энергонезависимой памяти (EEPROM). Обязательная часть RTC-модуля, которая используется модулем DS3132.
Примечание: в данной работе рассматривается взаимодействие только с DS3132.

2.3.2. Проверка зависимостей и корректности настроек

На данный момент отображается адрес 0x68 – это не только указывает адрес устройства, но и говорит о том, что устройство не обслуживается никаким из драйверов, хотя необходимые драйверы уже подгружены в ядро. **Теперь необходимо указать системе, что новое устройство по адресу 0x68 – это RTC-часы, и для его обслуживания можно использовать модуль ядра «rtc-ds1307».**

Для этого выполним команду:

```
sudo /bin/sh -c "/bin/echo ds1307 0x68 > /sys/class/i2c-adapter/i2c-
1/new_device"
```

Теперь проверим как отображается устройство на шине:

```
pi@raspberrypi:~ $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- UU -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Теперь адрес устройство по адресу 0x68 (см.пересечении строки 60 и столбца 8) имеет значение UU – это означает, что RTC таймер DS3231 контролируется подгруженным модулем ядра (и это хорошо). Именной такой результат должен быть.

Теперь для окончательной проверки успешности всех подготовительных настроек, выполним команду получения системного времени и времени RTC-модуля

```
pi@raspberrypi:~ $ sudo date; sudo hwclock -r
Fri 26 May 14:55:58 UTC 2017
Fri 26 May 2017 14:55:58 UTC  -0.027346 seconds
```

Примечание: время не обязательно должно совпадать, т.к. это только проверка связи с модулем (в данном случае на модуле уже установлено точное время).

2.4. RTC демон

2.4.1. Описание демона

Рисунок 3. Алгоритм работы RTC-демона

Алгоритм работы демона следующий (рис.3):

- **Команда start.** Когда система запускается, при старте демона сначала выполняется подготовительная команда регистрации RTC-модуля в системе, и сразу же после этого происходит синхронизация часов внутреннего времени RaspBerry Pi с часами RTC-модуля. После этого демон завершает свою работу, однако из-за используемого флага RemainAfterExit=yes видится в системе как активный.

- **Команда stop.** Когда система выключается, демон выполняет обратную операцию: синхронизирует часы RTC-модуля с часами RPi (т.к. в хо де работы может произойти синхронизация времени через NTP, или же пользователь самостоятельно изменит системное время или часовой пояс). После останова, демон выполняет команду отключения RTC-модуля из системы, и модуль вновь становится не зарегистрированным.

Файл конфигурации **«rtc.service»** необходимо поместить по адресу «/etc/systemd/system/».

rtc.service [4, 6, 7, 8]

```
[Unit]
Description=Daemon for resyncing of system clock by RTC Timer DS3231
Before=getty.target

[Install]
WantedBy=multi-user.target

[Service]
Type=oneshot
RemainAfterExit=yes

ExecStartPre=/bin/sh -c "/bin/echo ds1307 0x68 > /sys/class/i2c-adapter/i2c-
1/new_device"
ExecStart=/sbin/hwclock -s

ExecStop=/sbin/hwclock -w
ExecStopPost=/bin/sh -c "/bin/echo 0x68 > /sys/class/i2c-adapter/i2c-
1/delete_device"

KillMode=process
KillSignal=SIGTERM
SendSIGKILL=yes
TimeoutStartSec=5
TimeoutStopSec=5
```

Описание конфигурации:

Before	(модуль должен загружаться ДО запуска всех TTY-терминалов);
WantedBy	(загрузка должен производиться на 3 уровне, в слое пользовательских программ);
Type	(тип демона – одиночный скрипт, который быстро завершится);
RemainAfterExit	(по завершению процесса, считать демон запущенным);
ExecStartPre	(перед запуском демона, выполнить указанную команду регистрации устройства в системе);
ExecStart	(синхронизировать время RPi, со временем RTC-модуля);
ExecStop	(синхронизировать время RTC-модуля, со временем RPi);
ExecStopPost	(после выключения демона, сделать устройство незарегистрированным);
KillMode	(отключать конкретный процесс, не контрольную группу);
KillSignal	(посылать указанный сигнал по команде stop);
SendSIGKILL	(посылать SIGKILL спустя определенное время (TimeoutStopSec), чтобы отключить сервис, который по какой-либо причине смог не выключиться);
TimeoutStartSec	(время на запуск демона);
TimeoutStopSec	(время на остановку демона).

2.4.2. Установка и запуск

Для регистрации и активации демона выполнить:

```
sudo systemctl enable rtc        # добавить в автозапуск
sudo systemctl start rtc         # запустить созданную службу
```

2.4.3. Тестирование работы

Перед началом тестирования:

- регистрируем/запускаем демон;
- отключаем NTP-демон командой **«sudo systemctl disable ntp».**

После завершения подготовительного этапа устанавливаем заведомо неправильное время на RTC-модуле, с помощью следующей команды:

```
sudo hwclock --set --date="Sat May  6 15:59:01 2020"
```

Еще раз сравниваем текущее системное время и время модуля:

```
pi@raspberrypi:~ $ sudo date; sudo hwclock -r
Sat May  6 19:36:23 UTC 2017
Sat May  6 15:59:01 2020  -0.399754 seconds
```

Команда показывает, что системное время и время RTC-таймера отличается.
Перезагружаем устройство (RTC модуль продолжает работать):

```
sudo reboot
```

После загрузки проверяем статус демона:

```
pi@raspberrypi:~/daemonRTC $ sudo systemctl status rtc
● rtc.service - Daemon for resyncing of system clock by RTC Timer DS3231
   Loaded: loaded (/etc/systemd/system/rtc.service; enabled)
   Active: active (exited) since Fri 2017-05-26 15:35:14 UTC; 4s ago
  Process: 1353 ExecStart=/sbin/hwclock -s (code=exited, status=0/SUCCESS)
  Process: 1347 ExecStartPre=/bin/sh -c /bin/echo ds1307 0x68 > /sys/class/i2c-
adapter/i2c-1/new_device (code=exited, status=0/SUCCESS)
 Main PID: 1353 (code=exited, status=0/SUCCESS)

May 26 15:35:14 raspberrypi systemd[1]: Started Daemon for resyncing of system
clock by RTC Timer DS3231.
```

Вновь проверяем время в системе и на модуле:

```
pi@raspberrypi:~ $ sudo date; sudo hwclock -r
Sat May  6 19:36:23 UTC 2017
Sat May  6 19:36:24 2017  -0.399754 seconds
```

Примечание: если установить системное время отличное от RTC-часов, и остановить или перезагрузить демон (можно перезагрузить устройство), то время на RTC-таймере будет изменено на время системных часов.

2.5. Подведение итогов

Время показывается правильно, и теперь, при загрузке система сразу получает точное время от таймера, а при выключении – таймер получает обновленное время от RPI (от системы) и хранит его до включения основного устройства.

3. Cerebro (Digispark MCU ATtiny85)

3.1. Аппаратное подключение

Подключение устройства DigiSpark [1] с одноплатным миникомпьютером RaspBerry Pi Zero (рис.4) производится в соответствии с таблицей 2.

Таблица 2. Подключение MCU

MCU Digispark (Attiny 85)	GPIO	Описание
GND	P1-06 (GND)	Земля
VCC	P1-01 (3.3V)	Питание латентно и к 3В и к 5В;
P0	P1-03 (I2C SDA)	I2C Data линия, соединить с GPIO3;
P1	disp_BlackLight	PWM для управления яркостью дисплея. Нужно отсоединить от GPIO27 RPI и подключить на P1 Digispark`a; идет на BlackLight-вход дисплея;
P2	P1-05 (I2C SCL)	I2C Clock линия, соединить с GPIO5;
P3	button_block	кнопка блокирования/включения экрана и включения/выключения устройства;
P4	P6-01 (RUN)	контакт включения RPI командой от Digispark;
P5	Bat+	вход АЦП.

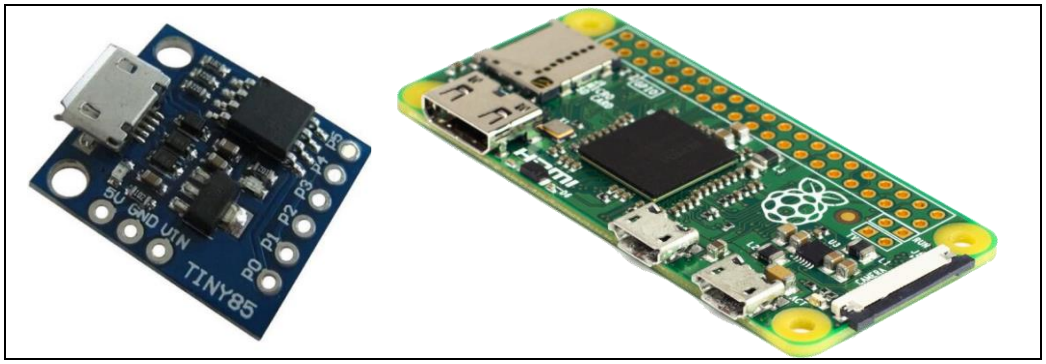

Рисунок 4. MCU Digispark (Attiny 85) и Raspberry Pi Zero

***Примечание*:** на момент написания курсовой все, вышедшие миникомпьютеры (RPi 1,2,3 и Zero), имели одинаковую распиновку (pinout). То есть описанное подключение модуля подходит для любой из плат. Приведенные ниже схемы описывают схему подключение кнопок (рис.5), и схему подключения делителя напряжения (для АЦП).

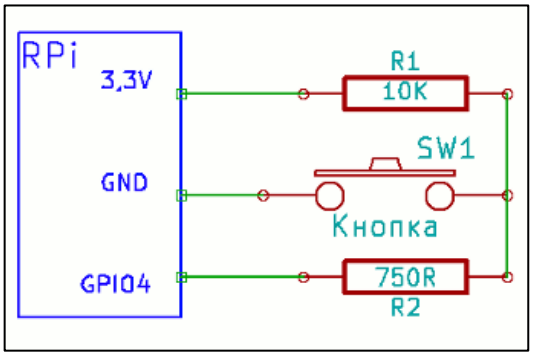

Рисунок 5. Принципиальная схема подключения кнопки

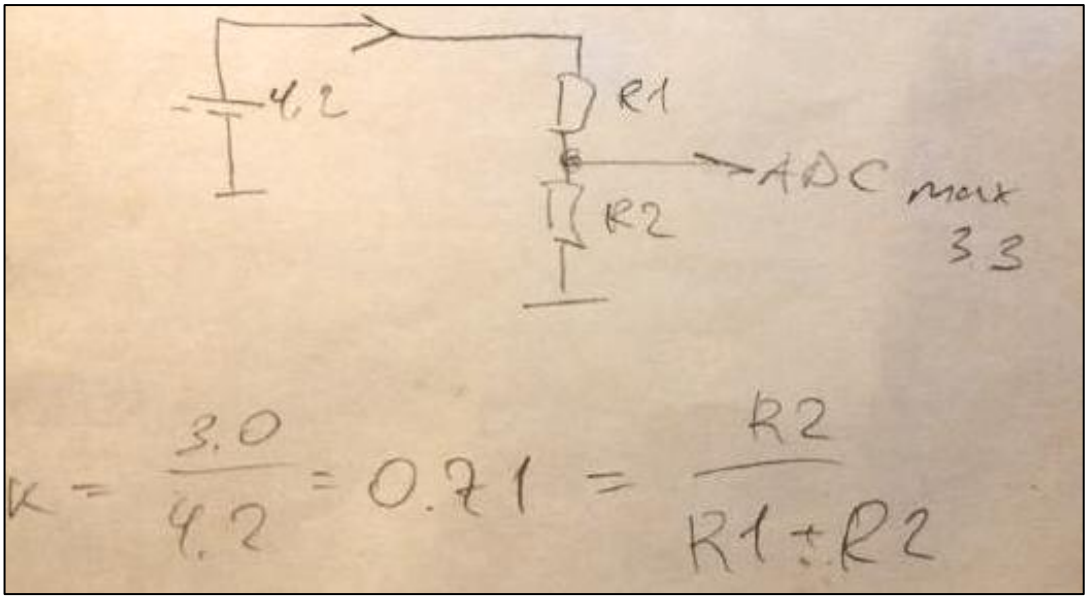

Рисунок 6. Принципиальна схема делителя (R1=12кОм; R2=30кОм)

Примечание: на рисунке 6 приведена схема делителя, позволяющая снимать напряжения с аккумулятора 4.2В средствами АЦП, диапазон снятия напряжения которого от 0В до +3.3В.

3.2. Установка зависимостей и настройка системы

Аналогично пункту 2.2.

3.3. Архитектура ПО демона

Приложение состоит из двух зависимых потоков (рис.7):

- поток «**опроса состояния MCU**» опрашивает состояние микроконтроллера (MCU) командой 0xCC (получить статус) раз в секунду и обновляет сведения о заряде батареи раз в минуту командой 0xBB.
- поток «**обслуживание запросов**» принимает подключения сторонних приложений по управлению и получению информации с МК.

Рисунок 7. Архитектура службы

Зависимость потоков заключается в том, что оба потока используют общий I2C-интерфейс для взаимодействия с МК. Решение коллизий и их синхронизация производится посредством мьютексов.

Примечание: архитектура незатейливая, но алгоритм работы «железный» (об этом ниже).

3.4. Алгоритм работы

3.4.1. Алгоритм работы потока «обслуживание запросов»

Поток «обслуживание запросов» сторонних приложений представляет собой сервер, использующий UDS для подключения клиентов. Алгоритм работы потока приведен на рисунке 8. Исходя из требований, которые были выдвинуты другими разработчика, установлено, что обращение к демону производится 1-2 приложение одновременное (не более). Так как каждая из 4 команд обслуживается довольно быстро, принято решение сделать сервер однопоточным, что позволяет не усложнять службу зазря.

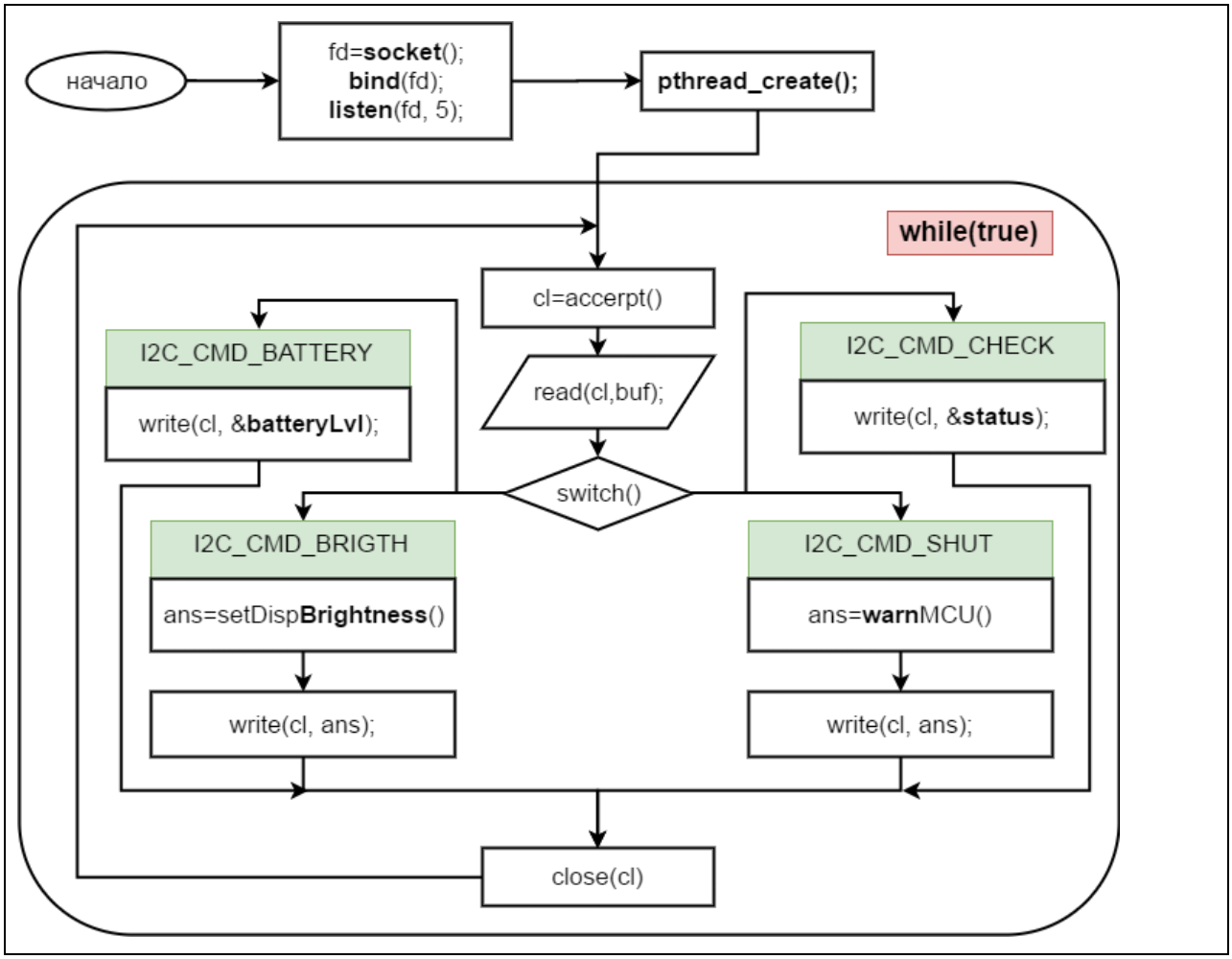

Рисунок 8. Алгоритм работы потока «обслуживания запросов»

3.4.2. Алгоритм работы потока «опроса состояния MCU»

Поток «опроса состояния MCU» опрашивает состояние микроконтроллера (MCU) командой 0xCC (получить статус), на что МК возвращает 1 байт**, нулевой бит** в котором

показывает флаг о выключении RPI (если кнопка блокировки долго нажата, то МК устанавливает флаг выключения), а **первый бит** – флаг блокировки экрана.

Если установлен **флаг выключения** (0 – значит установлен) – демон инициализирует приложение **xxoff**, которое выводим диалоговое окно «выключения». Если пользователь ответит «выключить», то приложение xxoff посылает сигнал 0x88 рассматриваемому демону, а тот инициализирует процесс выключения.

Если установлен флаг блокировки экрана (0 – значит установлен) – демон инициализирует приложение **xxblock**, которое блокирует любые действия пользователя, при использовании сенсора. Когда флаг сброшен (то есть равен 1) – приложению посылается сигнал об отмене блокировки.

Примечание: описание программ **xxoff** и **xxblock** не входит в состав рассматриваемой курсовой работы, т.к. программы разрабатываются другими участниками команды.

3.5. Разработка протокола взаимодействия

Для удовлетворения всех требований необходимо обеспечить поддержку следующего набора команд:

- получить текущий уровень заряд батареи;
- установить яркость экрана;
- выключить/включить миникомпьютер RPi;
- узнать текущее состояние экрана (заблокирован/разблокирован);

По той причине, что взаимодействие происходит с относительно медленным (на фоне RPi) микроконтроллером по достаточно медленной шине I2C, принято решение, что команды должны быть минималистичны: **представление команд в бинарном виде.**

3.5.1. Протокол взаимодействия по I2C-шине

На рисунке 9 приведен формат протокола взаимодействия демона «Cerebro» и MCU через I2C-шину. Инициатором всех команды является мастер на шине (Raspberry Pi). Мастер использует соответствующие 4 команды для получения статуса MCU (какие кнопки нажаты), получения текущего уровня заряда батареи, установки яркости экрана и предупреждения о своем выключении.

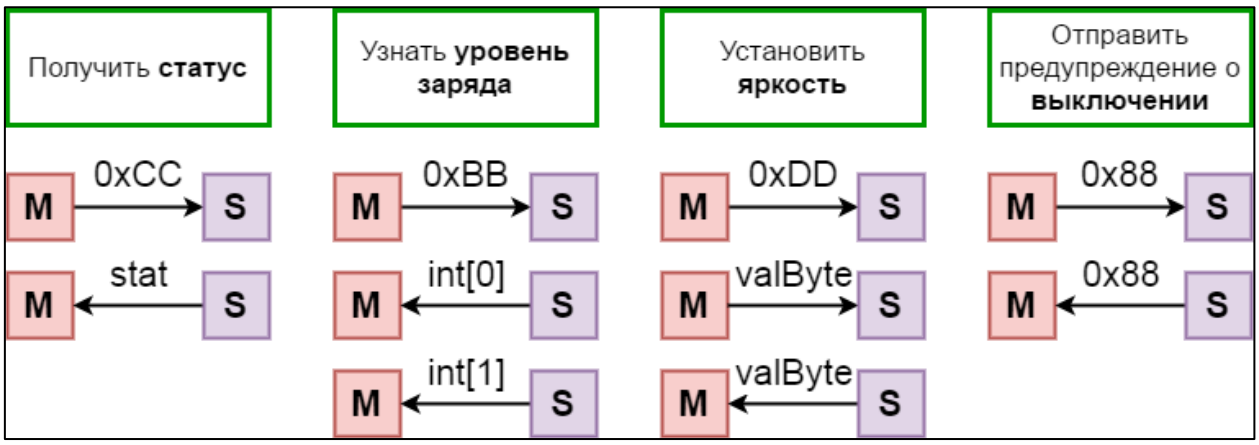

Рисунок 9. Протокол взаимодействия демона «Cerebro» и MCU через I2C-шину

3.4.2. Протокол взаимодействия с демоном

Для простоты понимания использован тот же самый протокол, но со следующими отличиям при получении ответа:

- команда «получить статус» возвращает значение из кеша, который обновляется каждую секунду;
- команда «узнать уровень заряда» возвращает значение от 0 до 100%;
- команда «установить яркость» возвращает 0, если яркость установлена корректно;
- команда 0x88 имеет иное значение: пользователь отдал команду на выключение.

Примечание: в остальном протоколы взаимодействия между демоном-приложениями и демоном-микроконтроллером полностью совпадают.

3.6. Настройка и включение

3.6.1. Конфигурационный файл

Файл конфигурации **«cerebro.service»** необходимо поместить по адресу «/etc/systemd/system/».

cerebro.service rtc.service [6, 7, 8]

```
[Unit]
Description=Daemon for control/communicate with Digispark (Cerebellum)
Before=getty.target

[Install]
WantedBy=multi-user.target

[Service]
User=root
Type=simple
ExecStart=/usr/sbin/cerebro
KillMode=process
KillSignal=SIGINT
SendSIGKILL=yes
TimeoutStartSec=5
TimeoutStopSec=5
```

Описание конфигурации:

Before	(модуль должен загружаться ДО запуска всех TTY-терминалов);
WantedBy	(загрузка должен производиться на 3 уровне, в слое пользовательских программ);
User=root	демон должен запускать от имени root-пользователя, т.к. использует создание скрытых unix-сокетов;
Type=simple	(тип демона – одиночной процесс);
ExecStart	(команда запуска процесса);
KillMode	(отключать конкретный процесс, не контрольную группу);
KillSignal	(посылать указанный сигнал по команде stop);
SendSIGKILL	(посылать SIGKILL спустя определенное время (TimeoutStopSec), чтобы отключить сервис, который по какой-либо причине смог не выключиться);
TimeoutStartSec	(время на запуск демона);
TimeoutStopSec	(время на остановку демона).

3.6.2. Установка и запуск

Для начала исходный код демона (файл «cerebro.cpp») необходимо скомпилировать с помощью Makefile.

Makefile

```
all:
	g++ cerebro.cpp -o cerebro -pthread

clean:
	rm cerebro
```

Для регистрации и активации демона выполнить:

```
sudo systemctl enable cerebro          # добавить в автозапуск
sudo systemctl start cerebro           # запустить созданную службу
```

Выполнить проверку можно командой **`systemctl status cerebro.`**

3.7. ПО МК DigiSpark (ATtiny85)

Разработка программного обеспечения микроконтроллера – это всегда настоящее искусство. В том числе, по той причине, что МК позволяет сделать выполнение некоторых задач **в режиме реального времени.**

При разработке прошивки для МК DigiSpark необходимо взять во внимание характеристики МК:

- 6 КБ постоянно памяти, из которых 1.5КБ занимает загрузчик;
- 82КБ ОЗУ;
- всего 6 контактов для взаимодействия.

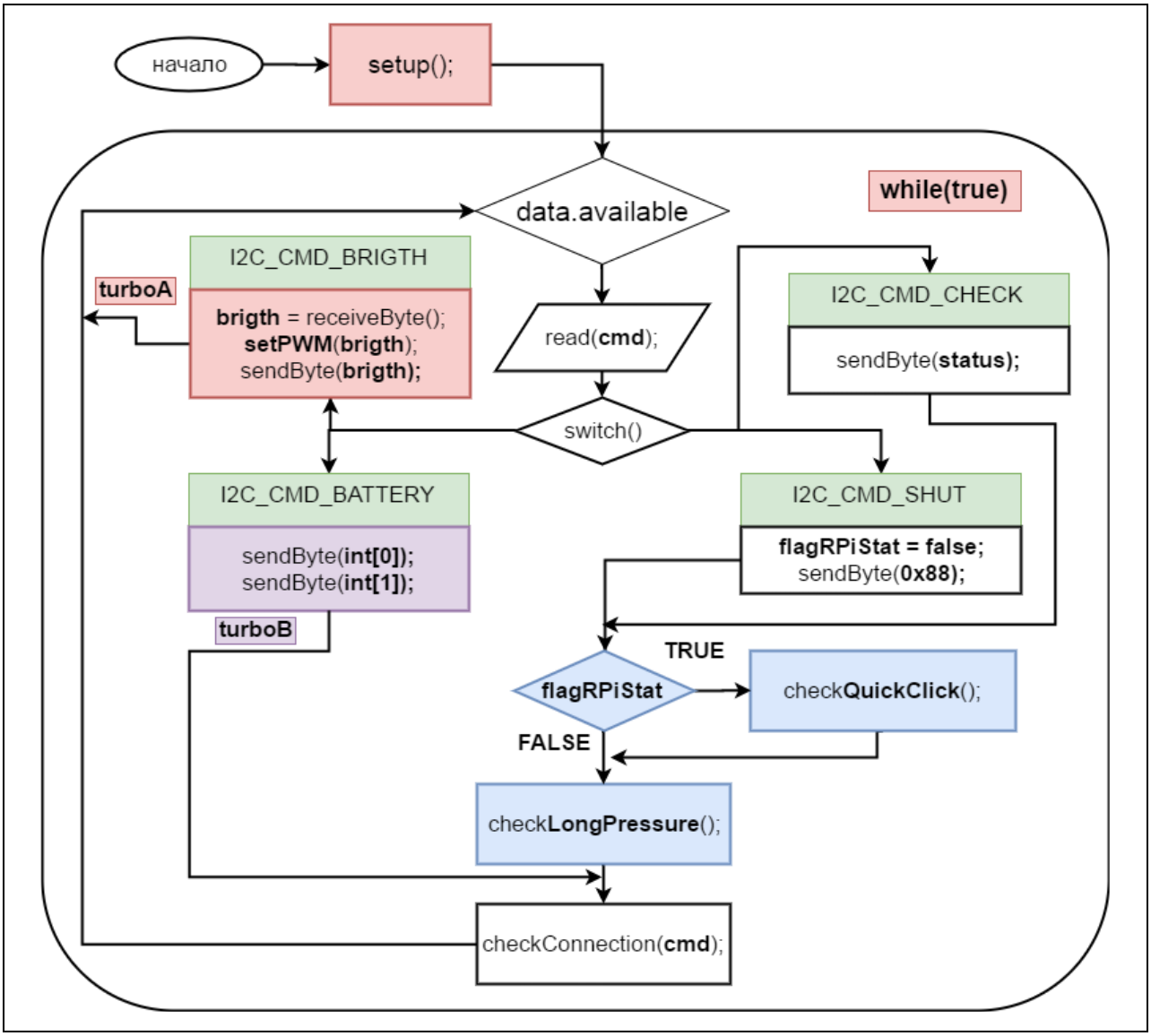

Рисунок 10. Алгоритм работы ПО MCU

На рис. 10 приведен полный алгоритм работы ПО МК. Как видно, МК поддерживает, аналогичные демону, **четыре I2C-команды:**

1. задать яркость дисплея;
2. выдать показания с батареи;
3. выдать текущий статус;
4. считать RPI выключенным.

Наиболее важной командой из всех считает команда установить яркость, поэтому она имеет максимальный приоритет и при поступлении команды I2C_CMD_BRIGHT – все остальные команды откладывают, и МК переходит в режим ожидания следующей команды.

Второй по приоритетности является команда I2C_CMD_BATTERY , т.к. команда является самой дорогостоящей из всех: время срабатывания АЦП на хороших МК может варьироваться от 12мс до 29мс. МК выполняет блокирование работы, до завершения оцифровки, и затем по 1 байту передает данные. Это может вызвать задержку на I2C-шине, поэтому часть ответственная за анализ нажатия кнопки – опускается.

Примечание: МК выполняет **все задачи в одном потоке.**

3.8. Описание тестирования

Для тестирования работы демона и МК разработан набор из 8 тестов (рис.11): 4 для МК и 4 для демон+МК, каждый из которых запускает свою отдельную команду и проверяет результат работы МК и демона+МК.

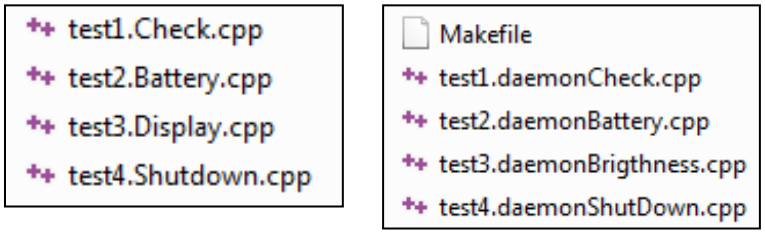

Рисунок 11. Разработанные тесты testMCU (слева) и testCerebro (справа)

Для тестирования каждой команды используется 1 тест для проверки отклика МК, 1 тест для проверки корректности работы демон (+МК). Алгоритм работы тестов приведен на рисунке 12: тесты из набора testMCU напрямую взаимодействуют с микроконтроллером и не нуждаются в наличии стороннего ПО (демона Cerebro); тесты из набора testCerebro тестируют работу демона. Разумеется для проведения всех тестов необходимо наличие МК DigiSpark.

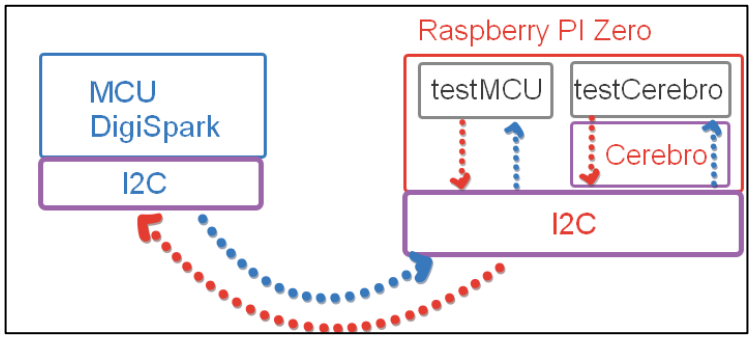

Рисунок 12. Алгоритм работы тестового пакета

Примечание: все тесты пройдены успешно и помогли скорректировать несколько серьезных ошибок (то есть **от тестирования получен качественный результат).**

4. Демон контроля звука (btn_sound)

4.1. Аппаратное подключение

Необходимыми и обязательными компонентами демона являют кнопки **увеличения** и **уменьшения** громкости звука. Подключение кнопок производится в соответствии с таблицей 3.

Таблица 3. Подключение кнопок контроля звука

Кнопки	GPIO RPi	Описание
sound+	P1-04	кнопка для увеличения громкости звука в режиме разговора или воспроизведения звука;
sound-	P1-17	кнопка для уменьшения громкости соответственно.

***Примечание*:** на момент написания курсовой все, вышедшие миникомпьютеры (RPi 1,2,3 и Zero), имели одинаковую распиновку (pinout). То есть описанное подключение RTC-модуля подходит для любой из плат.

Примечание: принципиальная схема подключения кнопки приведена на рис.5.

4.2. Алгоритм работы

Общий алгоримт работы демона представлен на рисунке 13. При запуске демона **производится резервирование** аппартной части – GPIO, которое служит для подключения кнопок.

Примечание: Для корректной работы демона обязательно подключение кнопок по схеме приведенной на рисунке 5. Если это не произведено – шум, постоянно присутствующий на GPIO типа input приводит к ложным срабатываниям ПО (демон считает, что кнопку беспрерывно нажимают), что приводит заполнению DBus-шины и «захламлению» логов.

После успешной инициализации GPIO (за это отвечает разработанны класс GPIOControl), демон переходит **в режим ожидания нажатия кнопки или истечения таймаута** этого события. Очень важно отметить, что за ожидание собтытия так же отвечает метод класса GPIOControl - **`getValueOnEvent`**, который позволяет детектировать события наGPIO, а не опрашивать состояние GPIO с определенной частотой. То есть событие носит асинхронный характер.

Если событие наступило, в дело вступает класс OrgOfonoCallVolumeInterface, который предоставляет методы взаимодействия с демоном Ofono (который отвечает за совершение любых действий с чипом связи SIM800L) по системной DBus-шине (не путать с сеансной dbus-шиной). После передачи командый демона некоторое время (250мс) не реагирует на нажатия кнопок, а затем вновь переходит в режим ожидания события/таймаута.

Если событие не наступило, а произошел таймаут – демон повторно переходит в режим ожидания события.

Если поступило прерывание (по сигналу SIGINT) – производится освобождение аппаратных ресурсов (GPIO) и завершение работы.

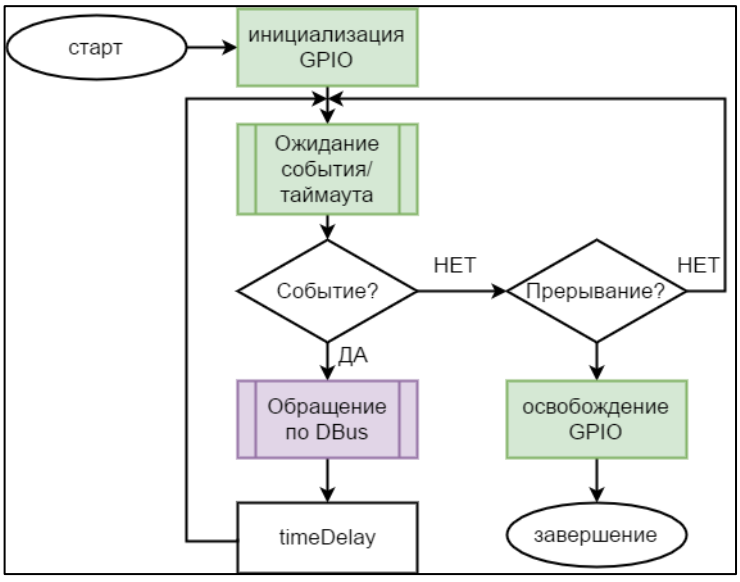

Рисунок 13. Алгоритм работы демона **btn_daemon**

4.3. Архитектура ПО демона

Приложение состоит из двух независимых потоков: **потоки «увеличения»/ «уменьшения» громкости** звука. Данный демон является самым обширным с точки зрения файловой архитектуру, хотя и не самым богатым с функциональной точки зрения. Файловая архитектура сервиса приведена на рисунке 14.

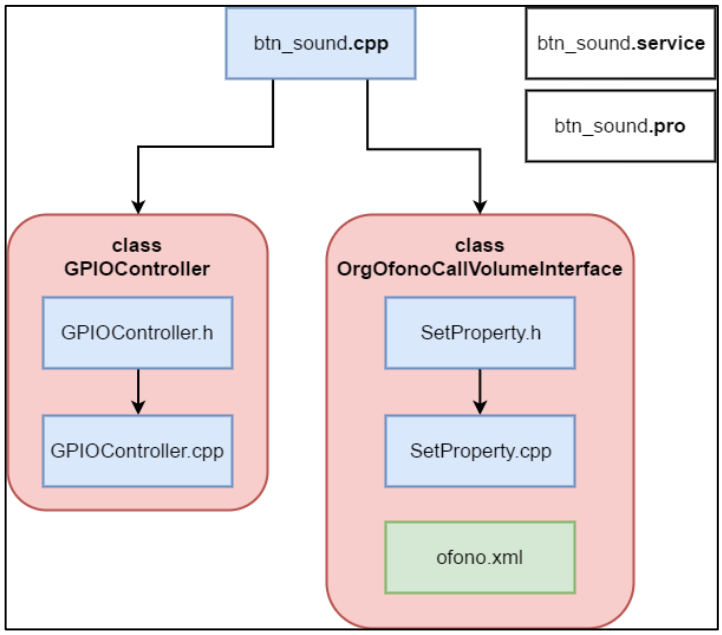

Рисунок 14. Файловая архитектура

4.3.1. Класс GPIOController

Этот класс отвечает за инициализацию, обработку и освобождение GPIO-контактов Raspberry Pi. Отличительной особенностью класса является метод **`getValueOnEvent`**, позволяющий устанавливать обработчик по событию, возникающему на контакте. Среди поддерживаемых событий:

- rising (переход из 0 в 1);
- failing (переход из 1 в 0);
- both (одновременно оба вышеперечисленных метода);
- none (отключения реакции на любые события).

Класс предоставляет методы, приведенные в листинге ниже:

Методы класса GPIOController

```
    // create a GPIO object that controls GPIO with number == gpioNum
    GPIOController();
    ~GPIOController();              // no comments

    int setDirection(string dir);   // set GPIO Direction (values: in/out)
    int setEvent(string edge);      // set type of event - type of edge (values:
none/both/rising/falling)
    int setValue(int value);        // set GPIO Value (only for output GPIO)

    int getValueOnEvent(int timeOutInMSec);   // get GPIO Value on the Event (ms)
    int getValue();                 // get GPIO Value
    int getGpioNum();               // return the GPIO number associated with the…
    string getGpioDirection();      // return direction of the GPIO
    string getGpioEvent();          // return event`s type of the GPIO
```

4.3.2. Класс OrgOfonoCallVolumeInterface

Данный класс является реализацией **паттерна «адаптер»**: класс и*мплементирует интерфейс* QDBusAbstractInterface [5], предоставляя необходимый для задания громкости звука во время разговора (через QDBus демону Ofono) метод **SetProperty().**

4.3.3. Файлы для сборки/установки

btn_sound.pro – файл для сборки Makefile`a, который уже произведет компиляцию бинарного файла;

btn_sound.service – конфигурационный файл для регистрации сервиса btn_sound.

4.4. Настройка и включение

4.4.1. Конфигурационный файл

Файл конфигурации **«btn_sound.service»** необходимо поместить по адресу «/etc/systemd/system/».

btn_sound.service rtc.service [6, 7, 8]

```
[Unit]
Description=Daemon for handling of events from sound buttons
Before=getty.target

[Install]
WantedBy=multi-user.target

[Service]
Type=simple
ExecStart=/usr/sbin/btn_sound
KillMode=process
KillSignal=SIGINT
SendSIGKILL=yes
TimeoutStartSec=5
TimeoutStopSec=5
```

Описание конфигурации:

Before	(модуль должен загружаться ДО запуска всех TTY-терминалов);
WantedBy	(загрузка должен производиться на 3 уровне, в слое пользовательских программ);
User=root	демон должен запускать от имени root-пользователя, т.к. использует создание скрытых unix-сокетов;
Type=simple	(тип демона – одиночной процесс);
ExecStart	(команда запуска процесса);
KillMode	(отключать конкретный процесс, не контрольную группу);
KillSignal	(посылать указанный сигнал по команде stop);
SendSIGKILL	(посылать SIGKILL спустя определенное время (TimeoutStopSec), чтобы отключить сервис, который по какой-либо причине смог не выключиться);
TimeoutStartSec	(время на запуск демона);
TimeoutStopSec	(время на остановку демона).

4.4.2. Установка и запуск

Для начала необходимо собрать Makefile. Для этого используем команду **qmake**.

btn_sound.pro

```
QT += core dbus
QT -= gui
CONFIG += c++11 qtquickcompiler warn_off

HEADERS += SetProperty/SetProperty.h \
SOURCES += btn_sound.cpp \
           GPIOController/GPIOController.cpp \
           SetProperty/SetProperty.cpp \

target.path = $$[QT_INSTALL_EXAMPLES]
sources.files = $$SOURCES $$HEADERS btn_sound.pro
sources.path = $$[QT_INSTALL_EXAMPLES]
INSTALLS += target sources
```

Далее, исходный код демона необходимо скомпилировать с помощью команды **make**. Для регистрации и активации демона выполнить:

```
sudo systemctl enable btn_sound          # добавить в автозапуск
sudo systemctl start btn_sound           # запустить созданную службу
```

Выполнить проверку можно командой **`systemctl status cerebro.`**

5. Вывод

#I2C #Dbus #QDbus #RTC #services #MCU #gpio #systemd

Курсовая работа раскрывает тему разработки системного ПО: пройдены этапы проектирования, разработки и тестирования. Тот факт, что тесты помогли выявить несколько ошибок в коде, указывает на состоятельность созданных тестов, и качество их практического применения. Примечательно так же, что разработанные программы напрямую взаимодействую с аппаратным слоем обеспечения: разработан распределенный комплект ПО, обеспечивающий надежное соединение между компонентами. Демон Cerebro позволяет взаимодействовать Raspberry Pi и MCU Digispark. Другой демон – btn_sound отвечает за программно-эффективное реагирование на состояние кнопок. Для передачи данных использовалась DBus-шина. Третий демон – rtc, реализован так же по всем канонам сервисов, использует нативный драйвер RTC-модуля и синхронизирует время в обоих направлениях.

Примечание: код приведен в отдельном файле «Приложение 1».

Источники

1. Подключение и программирование Digispark
http://lesson.iarduino.ru/page/podklyuchenie-i-programmirovanie-digispark/

2. Спецификация RTC-модуля DS3231
https://datasheets.maximintegrated.com/en/ds/DS3231.pdf

3. Introducing the Raspberry Pi Zero
https://cdn-learn.adafruit.com/downloads/pdf/introducing-the-raspberry-pi-zero.pdf

4. Exploring Raspberry Pi® Interfacing to the Real World with Embedded Linux® Derek Molloy

5. Qt D-Bus Examples
http://doc.qt.io/qt-5/examples-dbus.html

6. About types of targets
https://www.freedesktop.org/software/systemd/man/systemd.special.html

7. About systemd during 5 minutes
https://habrahabr.ru/company/centosadmin/blog/255845/

8. Systemd.unit — Unit configuration
https://www.freedesktop.org/software/systemd/man/systemd.unit.html