ПРИЛОЖЕНИЕ 1.Демон btn_sound

GPIOController.h

```
#ifndef GPIO_CLASS_H
#define GPIO_CLASS_H

#include <fstream>
#include <iostream>
#include <string.h>
#include <poll.h>                   // "poll"
#include <unistd.h>                 // "read/write/lseek"
#include <sys/stat.h>
#include <fcntl.h>                  // "O_RDONLY"
#include <linux/limits.h>           // "MAX_PATH"
#include <errno.h>                  //  codes of errors

#define DEFAULT_DIRECT "input"      // default value of direction
#define DEFAULT_EDGE   "both"       // default type of event (edge)

using namespace std;

/* GPIOController  */
class GPIOController
{
public:
    // create a GPIO object that controls GPIO with number == gpioNum
    GPIOController(int gpioNum, string direction=DEFAULT_DIRECT, string
event=DEFAULT_EDGE);
    ~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, time in
ms

    int getValue();                 // get GPIO Value
    int getGpioNum();               // return the GPIO number associated with the
instance of an object
    string getGpioDirection();      // return direction of the GPIO
    string getGpioEvent();          // return event`s type of the GPIO

private:
    string gpioNum;     // GPIO number associated with the instance of an object
    string direction;   // GPIO direction
    string event;
};

#endif
```

GPIOController.cpp

```
#include "GPIOController.h"

using namespace std;

// constructor creates and registers new GPIO with number "gpioNum" into the system
GPIOController::GPIOController(int gpioNum, string direction, string event) {
    // Instatiate GPIOController object for GPIO with current gpioNum number
    this->gpioNum = to_string(gpioNum);

    // open export file for adding new GPIO
    ofstream gpioExport("/sys/class/gpio/export");
    if (!gpioExport.is_open()){
        cout << "GPIOController Error#1: errno=" << errno << "; " <<
            "Unable to export GPIO"
             << this->gpioNum << ";" << endl;
        this->gpioNum = to_string(-1);
        return;
    }

    // add new GPIO to configuration and close file
    gpioExport << this->gpioNum ;
    gpioExport.close();

    // set dicrection for gpioNum
    setDirection(direction);

    // set type of event
    setEvent(event);
}

// destructor unregisters and removes GPIO with number "this->gpioNum" from system
GPIOController::~GPIOController() {
    // open unExport file for removing GPIO
    ofstream gpioUnExport("/sys/class/gpio/unexport");
    if (!gpioUnExport.is_open()){
        cout << "GPIOController Error#3: errno=" << errno << "; " <<
            "Unable to unexport GPIO"
             << this->gpioNum << ";" << endl;
        return;
    }

    // remove gpioNum from GPIO configuration
    gpioUnExport << this->gpioNum ;
    gpioUnExport.close();
    this->gpioNum = -1;
}

// function setDirection sets direction of GPIO work: "in" or "out" mode
int GPIOController::setDirection(string dir) {
    // open direction file for GPIO
    string gpioDirPath ="/sys/class/gpio/gpio" + this->gpioNum + "/direction";
    ofstream gpioDirFile(gpioDirPath.c_str());
```

```
    if (!gpioDirFile.is_open()){
        cout << "GPIOController Error#2: errno=" << errno << "; " <<
            "Unable to set direction for GPIO"
             << this->gpioNum << ";" << endl;
        return -1;
    }

    // write the direction to special file of direction
    gpioDirFile << dir;
    gpioDirFile.close();
    this->direction = dir;
    return 0;
}

// function setEvent sets type of event  (none, both, rising or falling)
int GPIOController::setEvent(string event) {
    // open file, which contain event of GPIO (0 or 1)
    string gpioEventPath = "/sys/class/gpio/gpio" + this->gpioNum + "/edge";
    ofstream gpioEventFile(gpioEventPath.c_str());
    if (!gpioEventFile.is_open()) {
        cout << "GPIOController Error#4: errno=" << errno << "; " <<
            "Unable to set event (edge) for GPIO"
             << this->gpioNum << ";" << endl;
        return -2;
    }

    // set type of event into file of event
    gpioEventFile << event;
    gpioEventFile.close();

    // also remember as property of GPIO
    this->event = event;
    return 0;
}

// function setValue sets value of GPIO
int GPIOController::setValue(int value) {
    // open file, which should contain value of GPIO (0 or 1)
    string gpioValuePath = "/sys/class/gpio/gpio" + this->gpioNum + "/value";
    ofstream gpioValueFile(gpioValuePath.c_str());
    if (!gpioValueFile.is_open()) {
        cout << "GPIOController Error#5: errno=" << errno << "; " <<
            "Unable to set value of GPIO"
             << this->gpioNum << ";" << endl;
        return -2;
    }

    // set value into gpioValueFile
    gpioValueFile << value;
    gpioValueFile.close();
    return 0;
}

// function getValue gets value of GPIO
```

```
int GPIOController::getValue() {
    // open file, which contain value of GPIO (0 or 1)
    string gpioValuePath = "/sys/class/gpio/gpio" + this->gpioNum + "/value";
    ifstream gpioValueFile(gpioValuePath.c_str());
    if (!gpioValueFile.is_open()){
        cout << "GPIOController Error#6: errno=" << errno << "; " <<
            "Unable to get the value of GPIO"
             << this->gpioNum << ";" << endl;
        return -2;
    }
    // read GPIO
    string tmp;
    gpioValueFile >> tmp ;
    gpioValueFile.close();

    // if 0 return 0 else return 1;
    return strcmp(tmp.c_str(), "0") == 0 ? 0 : 1;
}

// function getValue gets value of GPIO on the event (look function setEvent)
int GPIOController::getValueOnEvent(int timeOutInMSec) {
    // check (and may be set) type of event
    if (strcmp(this->event.c_str(), "both") != 0
     && strcmp(this->event.c_str(), "rising") != 0
     && strcmp(this->event.c_str(), "falling") != 0) {
        // try to reset default type of event
        if (setEvent(DEFAULT_EDGE) < 0)
            return -2;
        cout << "GPIOController Warning. Type of event is not defined." << endl
             << "Default value (" << DEFAULT_EDGE << ") is set." << endl;
    }

    char gpioValueFile[PATH_MAX];
    int fd;
    char c;
    int err;
    struct pollfd pollfd[1];

    // polling the line
    snprintf(gpioValueFile, sizeof(gpioValueFile), "/sys/class/gpio/gpio%s/value",
this->gpioNum.c_str());
    fd = open(gpioValueFile, O_RDONLY);
    if (fd < 0) {
        cout << "GPIOController Error#7: errno=" << errno << "; " <<
            "Unable to open the file with value of GPIO"
             << this->gpioNum << ";" << endl;
        return -3;
    }
    read(fd, &c, sizeof(c));

    pollfd[0].fd = fd;
    pollfd[0].events = POLLPRI | POLLERR;
    pollfd[0].revents = 0;
```

```
    // waiting of event
    errno = 0;  // clear
    err =  poll(pollfd, 1, timeOutInMSec);
    if(err != 1 && errno !=0 && errno !=4) { // 4 - Interrupted system call
        cout << "GPIOController Error#8: errno=" << errno << "; " <<
            "Unable to poll the value of GPIO"
             << this->gpioNum << ";" << endl;
        return -4;
    }

    // backing to start of value`s file
    lseek(fd, 0, SEEK_SET);
    read(fd, &c, sizeof(c));

    return c - '0';
}

// get number of current GPIO
int GPIOController::getGpioNum() {
    return stoi(this->gpioNum);
}

// get direction of current GPIO
string GPIOController::getGpioDirection() {
    return this->direction;
}

// get type of event of current GPIO
string GPIOController::getGpioEvent() {
    return this->event;
}
```

OrgOfonoCallVolumeInterface.h

```
/*
 * This file was generated by qdbusxml2cpp version 0.8
 * Command line was: qdbusxml2cpp -N -p SetProperty ofono.xml org.ofono.CallVolume
 *
 * qdbusxml2cpp is Copyright (C) 2015 The Qt Company Ltd.
 *
 * This is an auto-generated file.
 * Do not edit! All changes made to it will be lost.
 */

#ifndef SETPROPERTY_H
#define SETPROPERTY_H

#include <QtCore/QObject>
#include <QtCore/QByteArray>
#include <QtCore/QList>
#include <QtCore/QMap>
#include <QtCore/QString>
#include <QtCore/QStringList>
```

```
#include <QtCore/QVariant>
#include <QtDBus/QtDBus>

/*
 * Proxy class for interface org.ofono.CallVolume
 */
class OrgOfonoCallVolumeInterface: public QDBusAbstractInterface
{
    Q_OBJECT
public:
    static inline const char *staticInterfaceName()
    { return "org.ofono.CallVolume"; }

public:
    OrgOfonoCallVolumeInterface(const QString &service, const QString &path, const
QDBusConnection &connection, QObject *parent = 0);

    ~OrgOfonoCallVolumeInterface();

public Q_SLOTS: // METHODS
    inline QDBusPendingReply<QVariantMap> GetProperties()
    {
        QList<QVariant> argumentList;
        return asyncCallWithArgumentList(QStringLiteral("GetProperties"),
argumentList);
    }

    inline QDBusPendingReply<> SetProperty(const QString &property, const
QDBusVariant &value)
    {
        QList<QVariant> argumentList;
        argumentList << QVariant::fromValue(property) << QVariant::fromValue(value);
        return asyncCallWithArgumentList(QStringLiteral("SetProperty"),
argumentList);
    }

Q_SIGNALS: // SIGNALS
    void PropertyChanged(const QString &property, const QDBusVariant &value);
};

#endif
```

OrgOfonoCallVolumeInterface.cpp

```
/*
 * This file was generated by qdbusxml2cpp version 0.8
 * Command line was: qdbusxml2cpp -N -p SetProperty ofono.xml org.ofono.CallVolume
 *
 * qdbusxml2cpp is Copyright (C) 2015 The Qt Company Ltd.
 *
 * This is an auto-generated file.
 * This file may have been hand-edited. Look for HAND-EDIT comments
 * before re-generating it.
```

```
 */

#include "SetProperty.h"

/*
 * Implementation of interface class OrgOfonoCallVolumeInterface
 */

OrgOfonoCallVolumeInterface::OrgOfonoCallVolumeInterface(const QString &service,
const QString &path, const QDBusConnection &connection, QObject *parent)
    : QDBusAbstractInterface(service, path, staticInterfaceName(), connection,
parent)
{
}

OrgOfonoCallVolumeInterface::~OrgOfonoCallVolumeInterface()
{
}
```

ofono.xml

```
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node><interface name="org.freedesktop.DBus.Introspectable"><method
name="Introspect"><arg name="xml" type="s" direction="out"/>
</method></interface><interface name="org.ofono.Modem"><method
name="GetProperties"><arg name="properties" type="a{sv}" direction="out"/>
</method><method name="SetProperty"><arg name="property" type="s" direction="in"/>
<arg name="value" type="v" direction="in"/>
</method><signal name="PropertyChanged"><arg name="name" type="s"/>
<arg name="value" type="v"/>
</signal>
</interface><interface name="org.ofono.SimManager"><method name="GetProperties"><arg
name="properties" type="a{sv}" direction="out"/>
</method><method name="SetProperty"><arg name="property" type="s" direction="in"/>
<arg name="value" type="v" direction="in"/>
</method><method name="ChangePin"><arg name="type" type="s" direction="in"/>
<arg name="oldpin" type="s" direction="in"/>
<arg name="newpin" type="s" direction="in"/>
</method><method name="EnterPin"><arg name="type" type="s" direction="in"/>
<arg name="pin" type="s" direction="in"/>
</method><method name="ResetPin"><arg name="type" type="s" direction="in"/>
<arg name="puk" type="s" direction="in"/>
<arg name="newpin" type="s" direction="in"/>
</method><method name="LockPin"><arg name="type" type="s" direction="in"/>
<arg name="pin" type="s" direction="in"/>
</method><method name="UnlockPin"><arg name="type" type="s" direction="in"/>
<arg name="pin" type="s" direction="in"/>
</method><method name="GetIcon"><arg name="id" type="y" direction="in"/>
<arg name="icon" type="ay" direction="out"/>
</method><signal name="PropertyChanged"><arg name="name" type="s"/>
<arg name="value" type="v"/>
</signal>
```

```
</interface><interface name="org.ofono.AllowedAccessPoints"><method
name="GetAllowedAccessPoints"><arg name="apnlist" type="as" direction="out"/>
</method></interface><interface name="org.ofono.NetworkRegistration"><method
name="GetProperties"><arg name="properties" type="a{sv}" direction="out"/>
</method><method name="Register"></method><method name="GetOperators"><arg
name="operators_with_properties" type="a(oa{sv})" direction="out"/>
</method><method name="Scan"><arg name="operators_with_properties" type="a(oa{sv})"
direction="out"/>
</method><signal name="PropertyChanged"><arg name="name" type="s"/>
<arg name="value" type="v"/>
</signal>
</interface><interface name="org.ofono.SupplementaryServices"><method
name="Initiate"><arg name="command" type="s" direction="in"/>
<arg name="result_name" type="s" direction="out"/>
<arg name="value" type="v" direction="out"/>
</method><method name="Respond"><arg name="reply" type="s" direction="in"/>
<arg name="result" type="s" direction="out"/>
</method><method name="Cancel"></method><method name="GetProperties"><arg
name="properties" type="a{sv}" direction="out"/>
</method><signal name="NotificationReceived"><arg name="message" type="s"/>
</signal>
<signal name="RequestReceived"><arg name="message" type="s"/>
</signal>
<signal name="PropertyChanged"><arg name="name" type="s"/>
<arg name="value" type="v"/>
</signal>
</interface><interface name="org.ofono.VoiceCallManager"><method
name="GetProperties"><arg name="properties" type="a{sv}" direction="out"/>
</method><method name="Dial"><arg name="number" type="s" direction="in"/>
<arg name="hide_callerid" type="s" direction="in"/>
<arg name="path" type="o" direction="out"/>
</method><method name="Transfer"></method><method name="SwapCalls"></method><method
name="ReleaseAndAnswer"></method><method name="ReleaseAndSwap"></method><method
name="HoldAndAnswer"></method><method name="HangupAll"></method><method
name="PrivateChat"><arg name="call" type="o" direction="in"/>
<arg name="calls" type="ao" direction="out"/>
</method><method name="CreateMultiparty"><arg name="calls" type="ao"
direction="out"/>
</method><method name="HangupMultiparty"></method><method name="SendTones"><arg
name="SendTones" type="s" direction="in"/>
</method><method name="GetCalls"><arg name="calls_with_properties" type="a(oa{sv})"
direction="out"/>
</method><signal name="Forwarded"><arg name="type" type="s"/>
</signal>
<signal name="BarringActive"><arg name="type" type="s"/>
</signal>
<signal name="PropertyChanged"><arg name="name" type="s"/>
<arg name="value" type="v"/>
</signal>
<signal name="CallAdded"><arg name="path" type="o"/>
<arg name="properties" type="a{sv}"/>
</signal>
<signal name="CallRemoved"><arg name="path" type="o"/>
</signal>
```

```
</interface><interface name="org.ofono.MessageManager"><method
name="GetProperties"><arg name="properties" type="a{sv}" direction="out"/>
</method><method name="SetProperty"><arg name="property" type="s" direction="in"/>
<arg name="value" type="v" direction="in"/>
</method><method name="SendMessage"><arg name="to" type="s" direction="in"/>
<arg name="text" type="s" direction="in"/>
<arg name="path" type="o" direction="out"/>
</method><method name="GetMessages"><arg name="messages" type="a(oa{sv})"
direction="out"/>
</method><signal name="PropertyChanged"><arg name="name" type="s"/>
<arg name="value" type="v"/>
</signal>
<signal name="IncomingMessage"><arg name="message" type="s"/>
<arg name="info" type="a{sv}"/>
</signal>
<signal name="ImmediateMessage"><arg name="message" type="s"/>
<arg name="info" type="a{sv}"/>
</signal>
<signal name="MessageAdded"><arg name="path" type="o"/>
<arg name="properties" type="a{sv}"/>
</signal>
<signal name="MessageRemoved"><arg name="path" type="o"/>
</signal>
</interface><interface name="org.ofono.PushNotification"><method
name="RegisterAgent"><arg name="path" type="o" direction="in"/>
</method><method name="UnregisterAgent"><arg name="path" type="o" direction="in"/>
</method></interface><interface name="org.ofono.SmartMessaging"><method
name="RegisterAgent"><arg name="path" type="o" direction="in"/>
</method><method name="UnregisterAgent"><arg name="path" type="o" direction="in"/>
</method><method name="SendBusinessCard"><arg name="to" type="s" direction="in"/>
<arg name="card" type="ay" direction="in"/>
<arg name="path" type="o" direction="out"/>
</method><method name="SendAppointment"><arg name="to" type="s" direction="in"/>
<arg name="appointment" type="ay" direction="in"/>
<arg name="path" type="o" direction="out"/>
</method></interface><interface name="org.ofono.CallVolume"><method
name="GetProperties"><annotation name="org.qtproject.QtDBus.QtTypeName.Out0"
value="QVariantMap"/><arg name="properties" type="a{sv}" direction="out"/>
</method><method name="SetProperty"><arg name="property" type="s" direction="in"/>
<arg name="value" type="v" direction="in"/>
</method><signal name="PropertyChanged"><arg name="property" type="s"/>
<arg name="value" type="v"/>
</signal>
</interface><interface name="org.ofono.Phonebook"><method name="Import"><arg
name="entries" type="s" direction="out"/>
</method></interface><interface name="org.ofono.ConnectionManager"><method
name="GetProperties"><arg name="properties" type="a{sv}" direction="out"/>
</method><method name="SetProperty"><arg name="property" type="s" direction="in"/>
<arg name="value" type="v" direction="in"/>
</method><method name="AddContext"><arg name="type" type="s" direction="in"/>
<arg name="path" type="o" direction="out"/>
</method><method name="RemoveContext"><arg name="path" type="o" direction="in"/>
</method><method name="DeactivateAll"></method><method name="GetContexts"><arg
name="contexts_with_properties" type="a(oa{sv})" direction="out"/>
```

```
</method><method name="ResetContexts"></method><signal name="PropertyChanged"><arg
name="name" type="s"/>
<arg name="value" type="v"/>
</signal>
<signal name="ContextAdded"><arg name="path" type="o"/>
<arg name="properties" type="a{sv}"/>
</signal>
<signal name="ContextRemoved"><arg name="path" type="o"/>
</signal>
</interface><node name="context1"/><node name="context2"/><node
name="operator"/></node>
```

btn_sound.cpp

```
/*
    sndInc  - to increase sound if pressed (lvl.1)
    sndRdc - to reduce sound, if pressed (lvl.1)
*/

#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <pthread.h>

// QT
#include <QtCore/QStringList>
#include <QtDBus/QtDBus>

#include "GPIOController/GPIOController.h"
#include "SetProperty/SetProperty.h"

#define GPIO_SND_INC        4
#define GPIO_SND_RDC        17
#define ACTIV_LVL           0   // state for pressed button;
#define SENSITIVITY         10  // sensitivity of button`s press
                                // 1 clikc will change soundLvl on 10%
#define MAX_FREQ_EVENT_SND  250 // max frequency of event (in ms) for sound daemon


using namespace std;


int soundLvl = 100;         // in percents
bool SIGTEMT_event = false;
QString dBusPath;

void SIGINT_handler(int sig) {
    if (sig == 2) {
        cout << "\nSIGINT signal!\n" << endl;
```

```
        SIGTEMT_event = true;
    }
    else
        cout << "\nUnsupported signal!\n" << endl;
}

// function for Increasing and Reducing of the sound lvl
void *thrdSndCtrlFunction(void *arg);
// function for getting of path to SIM800L
int getDBusPath(QString &path);


int main (void) {
    // set handler for signal SIGINT
    struct sigaction sig_struct;
    sig_struct.sa_handler = SIGINT_handler;
    sig_struct.sa_flags = 0;
    sigemptyset(&sig_struct.sa_mask);

    if (sigaction(SIGINT, &sig_struct, NULL) == -1) {
        cout << "Problem with sigaction" << endl;
        cout << "Button Daemon Error#1: errno=" << errno << "; " <<
                "Could not set sigaction harder" << endl;
        exit(-1);
    }

    // thread env.
    int statusAddr;
    pthread_t thrdRdc;  // thread for reducing of Sound; Increasing occurs in main
thread;
    GPIOController *sndInc = NULL;  // obj. sndInc serves for increasing of sound
    GPIOController *sndRdc = NULL;  // obj. sndRdc serves for reducing of sound


    sndInc = new GPIOController(    // obj. sndInc serves for increasing of sound
        GPIO_SND_INC,   // number of GPIO
        "in",           // mode of work: input
        "falling"       // type of event: when input value falls from 1 to 0 (press
of button)
    );

    sndRdc = new GPIOController(    // obj. sndRdc serves for reducing of sound
        GPIO_SND_RDC,   // number of GPIO
        "in",           // mode of work: input
        "falling"       // type of event: when input value falls from 1 to 0 (press
of button)
    );


    // get dbus path
    if (getDBusPath(dBusPath) != 0) {
        cout << "Sound Daemon Error#2: errno=" << errno << "; " <<
                "Invalid DBus path;" << endl;
        delete sndInc;  // destruction of gpio objects
```

```
        delete sndRdc;
        exit(-2);
    }

    // Reducing thread. Creation and start
    if (pthread_create(&thrdRdc, NULL, thrdSndCtrlFunction, (void*) sndRdc) != 0) {
        cout << "Sound Daemon Error#3: errno=" << errno << "; " <<
                "Could not create reducing threads;" << endl;
        delete sndInc;  // destruction of gpio objects
        delete sndRdc;
        exit(-3);
    }

    // Increasing thread. Start in main
    thrdSndCtrlFunction(sndInc);

    // wainting finish of all threads (increasing and reducing)
    pthread_join(thrdRdc, (void**)&statusAddr);

    delete sndInc;  // destruction of gpio objects
    delete sndRdc;

    cout << "Succesfully completed." << endl;
    return 0;
}

// function for Increasing and Reducing of the sound lvl
void *thrdSndCtrlFunction(void *arg) {

    int value;  // current value of GPIO
    GPIOController *gpio = (GPIOController *)arg;

    QDBusConnection bus = QDBusConnection::systemBus();
    if(!bus.isConnected()){
        qDebug() << "Invalid connectnion#12" << endl;
        SIGTEMT_event = true;
        return NULL;
    }
    QDBusInterface cv("org.ofono", dBusPath, "org.ofono.CallVolume", bus);
    OrgOfonoCallVolumeInterface setPro("org.ofono", dBusPath, bus);


    while (!SIGTEMT_event) {
        // waiting the event (failing)
        value = gpio->getValueOnEvent(3000); // timeOut 3 sec

        // checking the interruption
        if (SIGTEMT_event) break;

        // checking: is event or timeOut?
        if (value != ACTIV_LVL) continue;     // it`s timeOut

        // it`s event
        if (gpio->getGpioNum() == GPIO_SND_INC) {
```

```
            soundLvl + SENSITIVITY <= 100 ? soundLvl += SENSITIVITY : soundLvl = 100;
            cout << "+soundLvl: " << soundLvl << endl;
        }
        else {
            soundLvl - SENSITIVITY > 0 ? soundLvl -= SENSITIVITY : soundLvl = 0;
            cout << "-soundLvl: " << soundLvl << endl;
        }


        // send through DBus;
        auto reply = setPro.SetProperty("SpeakerVolume",
QDBusVariant(qVariantFromValue(quint8(10))));
        reply.waitForFinished();

        // checking of errors
        if(reply.isError()) qDebug() << reply.error().name().toLatin1();

        // checking of success
        if(reply.isValid()) qDebug() << "Sound succesfully changed";

        // it`s regulate the max frequency of button`s event
        usleep(1000*MAX_FREQ_EVENT_SND);            // usleep works with microSec;
    }
    cout << "Thrd is finished." << endl;
    return NULL;
}


// function for getting of path to SIM800L
int getDBusPath(QString &path) {
    QDBusConnection bus = QDBusConnection::systemBus();
    if(!bus.isConnected()){
        qDebug() << "Invalid connectnion#1" << endl;//.value();
        return -1;
    }

    // default path
    QDBusInterface dbus_iface("org.ofono", "/", "org.ofono.Manager", bus);
    QDBusMessage modem = dbus_iface.call("GetModems");

    if(!modem.errorMessage().isNull() || !modem.errorMessage().isEmpty())
        return -2;

    QList<QVariant> outArgs = modem.arguments();
    const QDBusArgument &dbusArgs = outArgs.at(0).value<QDBusArgument>();

    dbusArgs.beginArray();
    while (!dbusArgs.atEnd()) {
        dbusArgs.beginStructure();
        while (!dbusArgs.atEnd()) {
            dbusArgs >> path;
            break;
            sleep(1);
        }
```

```
        dbusArgs.endStructure();
        break;
    }
    dbusArgs.endArray();

    return 0;
}
```

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
```

btn_sound.service

```
[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
```

ПРИЛОЖЕНИЕ 2.rtc

rtc.service

```
[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
```

ПРИЛОЖЕНИЕ 3. МК

mcu_cereb.ino

```
/*
  P0, P2 - I2C lines (taboo);
  P1, P4 - can be used as PWM;
  P2(A1), P3(A3), P4(A2), P5(A0) - can be used as input of ADC;

  Pull up resistors for GPIO are neccessary;
*/

#include "TinyWireS.h"            // wrapper class for I2C slave routines
```

```
// GPIO
#define GPIO_I2C_SDA                0     // P0. I2C Data line  (unuse definition)
#define GPIO_DISPLAY            1   // P1. PWM ping for brightness of DISPLAY
#define GPIO_I2C_SCK            2   // P2. I2C Clock line (unuse definition)
#define GPIO_BTN                3   // P3. Button
#define GPIO_RST                4   // P4 (quick LOW - reset)
#define GPIO_ADC                0   // P5 is A0; [P2(A1); P4(A2); P3(A3);]
#define GPIO_LED                1   // LED (P1). Debug.

// I2C
#define I2C_SLAVE_ADDR          0x26    // slave address (0x26h=38d)
// i2c.commands by master
#define I2C_CMD_BATTERY         0xBB    // "Check battery status" (frequency == 60
sec.)
#define I2C_CMD_CHECK           0xCC    // "Check general status" (frequency == 1
sec.)
#define I2C_CMD_BRIGTH          0xDD    // "Set display brightness" (async.)
#define I2C_CMD_SHUT            0x88    // "RPi shut down" (async.)

                                // OTHER commands are invalid

// GENERAL
#define DELAY                   1       // time delay between iterations in loop
(it should be: 1<=x<= 10)
#define TIME_FOR_LOADING        10000           // seconds for load of RPI ***
#define MAX_CLICK_TIME          1800    // (PressTime < MAX_CLICK_TIME) => click;

                                // (PressTime >= MAX_CLICK_TIME) => pression;

// function for incriment/decrement state counters(btnPressCnt, btnReleaseCnt)
void safeIncrement();
// poor man's display (debug)
void blink(byte led, byte times);
// check connection with RPI through I2C
void checkConnection(byte cmd);



// BUTTON
bool btnPressed = false;        // flag: true - button is pressed;
// button.(protection by tinkling of contacts)
int btnPrsCnt = 0;              // counter for containing of series of moments, when
button was Pressed
int btnRslCnt = 0;              // counter for containing of series of moments, when
button was Released

// DISPLAY
bool blockDisplay = true;       // flag: true - display is turned ON; false - display
is turned OFF (blacked)
byte blackLigth = 255;          // 0   - display is blacked;
                                // 255 - max. brightness;

// general
unsigned long time = 0;
```

```
bool shutDownEvnt = false;      // event of long pressing (User tryes to turn off the
device).
bool statusOfRPi = true;       // flag status of RPi:  true - the RPi works;
                                //                      false - the device sleeps;

// the setup routine runs once when you press reset:
void setup() {
    // initialize the digital/analog pins
    pinMode(GPIO_RST, OUTPUT);      // Reset
    pinMode(GPIO_DISPLAY, OUTPUT);  // PWM for brightness of Display
    pinMode(GPIO_BTN, INPUT);       // Button for Block/Unblock
    pinMode(GPIO_ADC, INPUT);       // ADC for Battery voltage

                                    // set GPIO_RST to 1 (default state); "quick 0" -
reset;
    digitalWrite(GPIO_RST, HIGH);

    // we will see quick blinking
    blink(GPIO_LED, 3);             // poor man's display

    // init I2C Slave mode
    TinyWireS.begin(I2C_SLAVE_ADDR);

    // for safing (waiting of finish of all initialized commands)
    delay (50);
}



// the loop routine runs over and over again forever:
void loop() {
                                    byte cmd = -1;

turboBright:
                                    // ******* I2C support *******
    // got I2C command from Master!
    if (TinyWireS.available()) {
                                        cmd = TinyWireS.receive();                  //
get the byte-command from master

        // I2C_CMD_BRIGTH-command "Set display brightness" (async.)
        if (cmd == (byte)I2C_CMD_BRIGTH) {
            if (TinyWireS.available()) {
                blackLigth = TinyWireS.receive();       // get the blackLigth value
from master
                analogWrite(GPIO_DISPLAY, blackLigth);  // update the brightness
                TinyWireS.send(blackLigth);             // send current value of
brightness back to master
            }
                                            goto turboBright;
        }

        // I2C_CMD_BATTERY-command "Check battery status" (frequency == 60 sec.)
        else if (cmd == (byte)I2C_CMD_BATTERY) {
```

```
            int batteryValue = analogRead(GPIO_ADC);
            TinyWireS.send(batteryValue & 0xFF);        // send low byte
            TinyWireS.send(batteryValue >> 8 & 0xFF);   // send high byte
                                            goto turbo;
        }

        // I2C_CMD_CHECK-command "Check general status" (frequency == 1 sec.)
        else if (cmd == (byte)I2C_CMD_CHECK) {
            byte buf = 0x00;
            buf |= blockDisplay ? 0x00 : 0x01;
            buf |= shutDownEvnt ? 0x00 : 0x02;
            TinyWireS.send(buf);                        // send it back to master
                                            shutDownEvnt = false;
                                            // reset flag, becouse RPi is warned
        }

        // I2C_CMD_SHUT-command "RPi shut down" (async.)
        else if (cmd == (byte)I2C_CMD_SHUT) {
            TinyWireS.send(cmd);
            analogWrite(GPIO_DISPLAY, 0);               // display is blacked
            statusOfRPi = false;                        // click and quick pressing
the button is ignored
        }

        // Other commands are Invalid
        else {
            //blink(GPIO_LED, 3); // debug stump ***
                                            //digitalWrite(GPIO_LED,LOW);
                                            //delay(10);
                                            TinyWireS.send(cmd);
                                        }
    }

// ******* BUTTON support *******
    btnPressed = (digitalRead(GPIO_BTN) == LOW);    // LOW - button is pressed
    safeIncrement();    // to incriment/decrement counters of Button states

    // quick Press or Click (button is pressed 30ms and it`s released 10ms yet,
    // we do our jon only by event - FALLING)
    if (statusOfRPi && btnPrsCnt >= 30/DELAY && btnRslCnt >= 10/DELAY && btnPrsCnt <
MAX_CLICK_TIME/DELAY) {
        // block/unblock the display
        blockDisplay = !blockDisplay;

        // it uses the kept value of brightness for ublocking;
        analogWrite(GPIO_DISPLAY, blockDisplay ? 0 : blackLigth);

        // button is released - empty the counter
        btnPrsCnt = 0;
    }

    // long Press (button is pressed more than MAX_CLICK_TIME mSec.)
    if (btnPrsCnt >= MAX_CLICK_TIME/DELAY) {
                                        // RPi sleeps
```

```
        if (!statusOfRPi) {
                                                digitalWrite(GPIO_RST, LOW);
                                        // set GPIO_RST to 0 (reset)
                                                delay (100);
                                                // "quick 0" (<=100ms)
                                                digitalWrite(GPIO_RST, HIGH);
                                        // back GPIO_RST to 1 (normal work)

                                                // RPi has got 10sec for starting
                                                statusOfRPi = true;
                                                // RPi works
                                                time = millis();
        }
                                        // RPi works
                                        else {
                                                // flag for sending message "TURN OFF"
through I2C to RPi
                                                shutDownEvnt = true;
                                        }
    }



turbo:

// ******* DELAY and CHECK_CONNECTION between itterations of cycle *******
    checkConnection(cmd);
    delay(DELAY);
}


// poor man's display (debug)
void blink(byte led, byte times) {
  for (byte i=0; i< times; i++) {
    digitalWrite(led,HIGH); delay (50);
    digitalWrite(led,LOW);  delay (50);
  }
}


// function for incriment/decrement state counters(btnPressCnt, btnReleaseCnt)
void safeIncrement() {
    // the button is PRESSED
    if (btnPressed) {
        btnPrsCnt++;                        // increment the counter
        btnRslCnt = 0;
    }
    // the button is RELEASED
    else {
        btnPrsCnt--;              // decrement the counters
        btnRslCnt++;              // increment the counter of Released series of
moments
    }
```

```
    // protection against an exit for time frames
    if (btnPrsCnt < 0) btnPrsCnt = 0;
    if (btnPrsCnt > 2000/DELAY) btnPrsCnt = 2000/DELAY;
    if (btnRslCnt > 2000/DELAY) btnRslCnt = 2000/DELAY;
}


// function for checking of connection
void checkConnection(byte cmd) {
                                    if (cmd == I2C_CMD_BATTERY
                                     || cmd == I2C_CMD_CHECK
                                     || cmd == I2C_CMD_BRIGTH) {
                                        statusOfRPi = true;       // RPi works
                                        time = millis();
                                    }
                                    // if connection was lost more than 10 sec.
                                    else if (abs(millis() - time) > TIME_FOR_LOADING *
1000) {
                                        statusOfRPi = false;      // RPi sleeps
                                        delay(1000);              // sleep 1 sec.
                                    }
}
```

TinyWireS.h

```
#ifndef TinyWireS_h
#define TinyWireS_h

#include <inttypes.h>


class USI_TWI_S
{
  private:
                                    //static uint8_t USI_BytesAvail;

  public:
                                    USI_TWI_S();
    void begin(uint8_t I2C_SLAVE_ADDR);
    void send(uint8_t data);
    uint8_t available();
    uint8_t receive();
};

extern USI_TWI_S TinyWireS;

#endif
```

TinyWireS.cpp

```
extern "C" {
```

```
  #include <inttypes.h>
  #include "usiTwiSlave.h"
  }

#include "TinyWireS.h"

// Constructors ////////////////////////////////////////////////////////////////

USI_TWI_S::USI_TWI_S(){
}


// Public Methods //////////////////////////////////////////////////////////////

void USI_TWI_S::begin(uint8_t slaveAddr){ // initialize I2C lib
  usiTwiSlaveInit(slaveAddr);
}

void USI_TWI_S::send(uint8_t data){  // send it back to master
  usiTwiTransmitByte(data);
}

uint8_t USI_TWI_S::available(){ // the bytes available that haven't been read yet
  return usiTwiDataInReceiveBuffer();
}

uint8_t USI_TWI_S::receive(){ // returns the bytes received one at a time
  return usiTwiReceiveByte();
}

// Preinstantiate Objects //////////////////////////////////////////////////////

USI_TWI_S TinyWireS = USI_TWI_S();
```

usiTwiSlave.h

```
#ifndef _USI_TWI_SLAVE_H_

#define _USI_TWI_SLAVE_H_




/***************************************************************************

                                    includes
```

```
*************************************************************************/

#include <stdbool.h>



/*************************************************************************

                                    prototypes

*************************************************************************/

void    usiTwiSlaveInit( uint8_t );
void    usiTwiTransmitByte( uint8_t );
uint8_t usiTwiReceiveByte( void );
bool    usiTwiDataInReceiveBuffer( void );



/*************************************************************************

                            driver buffer definitions

*************************************************************************/

// permitted RX buffer sizes: 1, 2, 4, 8, 16, 32, 64, 128 or 256

#define TWI_RX_BUFFER_SIZE  ( 32 ) // jjg was 16
#define TWI_RX_BUFFER_MASK  ( TWI_RX_BUFFER_SIZE - 1 )
```

```
#if ( TWI_RX_BUFFER_SIZE & TWI_RX_BUFFER_MASK )
#  error TWI RX buffer size is not a power of 2
#endif

// permitted TX buffer sizes: 1, 2, 4, 8, 16, 32, 64, 128 or 256

#define TWI_TX_BUFFER_SIZE ( 32 ) // jjg was 16
#define TWI_TX_BUFFER_MASK ( TWI_TX_BUFFER_SIZE - 1 )

#if ( TWI_TX_BUFFER_SIZE & TWI_TX_BUFFER_MASK )
#  error TWI TX buffer size is not a power of 2
#endif




#endif  // ifndef _USI_TWI_SLAVE_H_
```

usiTwiSlave.c

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "usiTwiSlave.h"



/***************************************************************************

                            device dependent defines

***************************************************************************/

#if defined( __AVR_ATtiny2313__ )
#  define DDR_USI             DDRB
#  define PORT_USI            PORTB
#  define PIN_USI             PINB
#  define PORT_USI_SDA        PORTB5
#  define PORT_USI_SCL        PORTB7
#  define PIN_USI_SDA         PINB5
#  define PIN_USI_SCL         PINB7
#  define USI_START_COND_INT  USISIF
#  define USI_START_VECTOR    USI_START_vect
```

```
#  define USI_OVERFLOW_VECTOR USI_OVERFLOW_vect
#endif

#if defined( __AVR_ATtiny25__ ) | \
     defined( __AVR_ATtiny45__ ) | \
     defined( __AVR_ATtiny85__ )
#  define DDR_USI             DDRB
#  define PORT_USI            PORTB
#  define PIN_USI             PINB
#  define PORT_USI_SDA        PORTB0
#  define PORT_USI_SCL        PORTB2
#  define PIN_USI_SDA         PINB0
#  define PIN_USI_SCL         PINB2
#  define USI_START_COND_INT  USISIF //was USICIF jjg
#  define USI_START_VECTOR    USI_START_vect
#  define USI_OVERFLOW_VECTOR USI_OVF_vect
#endif

#if defined( __AVR_ATtiny26__ )
#  define DDR_USI             DDRB
#  define PORT_USI            PORTB
#  define PIN_USI             PINB
#  define PORT_USI_SDA        PB0
#  define PORT_USI_SCL        PB2
#  define PIN_USI_SDA         PINB0
#  define PIN_USI_SCL         PINB2
#  define USI_START_COND_INT  USISIF
#  define USI_START_VECTOR    USI_STRT_vect
#  define USI_OVERFLOW_VECTOR USI_OVF_vect
#endif

#if defined( __AVR_ATtiny261__ ) | \
     defined( __AVR_ATtiny461__ ) | \
     defined( __AVR_ATtiny861__ )
#  define DDR_USI             DDRB
#  define PORT_USI            PORTB
#  define PIN_USI             PINB
#  define PORT_USI_SDA        PB0
#  define PORT_USI_SCL        PB2
#  define PIN_USI_SDA         PINB0
#  define PIN_USI_SCL         PINB2
#  define USI_START_COND_INT  USISIF
#  define USI_START_VECTOR    USI_START_vect
#  define USI_OVERFLOW_VECTOR USI_OVF_vect
#endif

#if defined( __AVR_ATmega165__ ) | \
     defined( __AVR_ATmega325__ ) | \
     defined( __AVR_ATmega3250__ ) | \
     defined( __AVR_ATmega645__ ) | \
     defined( __AVR_ATmega6450__ ) | \
     defined( __AVR_ATmega329__ ) | \
     defined( __AVR_ATmega3290__ )
#  define DDR_USI             DDRE
```

```
#  define PORT_USI            PORTE
#  define PIN_USI             PINE
#  define PORT_USI_SDA        PE5
#  define PORT_USI_SCL        PE4
#  define PIN_USI_SDA         PINE5
#  define PIN_USI_SCL         PINE4
#  define USI_START_COND_INT  USISIF
#  define USI_START_VECTOR    USI_START_vect
#  define USI_OVERFLOW_VECTOR USI_OVERFLOW_vect
#endif

#if defined( __AVR_ATmega169__ )
#  define DDR_USI             DDRE
#  define PORT_USI            PORTE
#  define PIN_USI             PINE
#  define PORT_USI_SDA        PE5
#  define PORT_USI_SCL        PE4
#  define PIN_USI_SDA         PINE5
#  define PIN_USI_SCL         PINE4
#  define USI_START_COND_INT  USISIF
#  define USI_START_VECTOR    USI_START_vect
#  define USI_OVERFLOW_VECTOR USI_OVERFLOW_vect
#endif



/***************************************************************************

                        functions implemented as macros

***************************************************************************/

#define SET_USI_TO_SEND_ACK( ) \
{ \
  /* prepare ACK */ \
  USIDR = 0; \
  /* set SDA as output */ \
  DDR_USI |= ( 1 << PORT_USI_SDA ); \
  /* clear all interrupt flags, except Start Cond */ \
  USISR = \
       ( 0 << USI_START_COND_INT ) | \
       ( 1 << USIOIF ) | ( 1 << USIPF ) | \
       ( 1 << USIDC )| \
       /* set USI counter to shift 1 bit */ \
       ( 0x0E << USICNT0 ); \
}

#define SET_USI_TO_READ_ACK( ) \
{ \
  /* set SDA as input */ \
  DDR_USI &= ~( 1 << PORT_USI_SDA ); \
  /* prepare ACK */ \
  USIDR = 0; \
  /* clear all interrupt flags, except Start Cond */ \
```

```
  USISR = \
       ( 0 << USI_START_COND_INT ) | \
       ( 1 << USIOIF ) | \
       ( 1 << USIPF ) | \
       ( 1 << USIDC ) | \
       /* set USI counter to shift 1 bit */ \
       ( 0x0E << USICNT0 ); \
}

#define SET_USI_TO_TWI_START_CONDITION_MODE( ) \
{ \
  /* set SDA as input */ \
  DDR_USI &= ~( 1 << PORT_USI_SDA ); \
  USICR = \
       /* enable Start Condition Interrupt, disable Overflow Interrupt */ \
       ( 1 << USISIE ) | ( 0 << USIOIE ) | \
       /* set USI in Two-wire mode, no USI Counter overflow hold */ \
       ( 1 << USIWM1 ) | ( 0 << USIWM0 ) | \
       /* Shift Register Clock Source = External, positive edge */ \
       /* 4-Bit Counter Source = external, both edges */ \
       ( 1 << USICS1 ) | ( 0 << USICS0 ) | ( 0 << USICLK ) | \
       /* no toggle clock-port pin */ \
       ( 0 << USITC ); \
  USISR = \
        /* clear all interrupt flags, except Start Cond */ \
        ( 0 << USI_START_COND_INT ) | ( 1 << USIOIF ) | ( 1 << USIPF ) | \
        ( 1 << USIDC ) | ( 0x0 << USICNT0 ); \
}

#define SET_USI_TO_SEND_DATA( ) \
{ \
  /* set SDA as output */ \
  DDR_USI |=  ( 1 << PORT_USI_SDA ); \
  /* clear all interrupt flags, except Start Cond */ \
  USISR    =  \
       ( 0 << USI_START_COND_INT ) | ( 1 << USIOIF ) | ( 1 << USIPF ) | \
       ( 1 << USIDC) | \
       /* set USI to shift out 8 bits */ \
       ( 0x0 << USICNT0 ); \
}

#define SET_USI_TO_READ_DATA( ) \
{ \
  /* set SDA as input */ \
  DDR_USI &= ~( 1 << PORT_USI_SDA ); \
  /* clear all interrupt flags, except Start Cond */ \
  USISR    = \
       ( 0 << USI_START_COND_INT ) | ( 1 << USIOIF ) | \
       ( 1 << USIPF ) | ( 1 << USIDC ) | \
       /* set USI to shift out 8 bits */ \
       ( 0x0 << USICNT0 ); \
}
```

```
/***************************************************************************

                                typedef's

***************************************************************************/

typedef enum
{
  USI_SLAVE_CHECK_ADDRESS               = 0x00,
  USI_SLAVE_SEND_DATA                   = 0x01,
  USI_SLAVE_REQUEST_REPLY_FROM_SEND_DATA = 0x02,
  USI_SLAVE_CHECK_REPLY_FROM_SEND_DATA   = 0x03,
  USI_SLAVE_REQUEST_DATA                = 0x04,
  USI_SLAVE_GET_DATA_AND_SEND_ACK       = 0x05
} overflowState_t;



/***************************************************************************

                              local variables

***************************************************************************/

static uint8_t                  slaveAddress;
static volatile overflowState_t overflowState;


static uint8_t          rxBuf[ TWI_RX_BUFFER_SIZE ];
static volatile uint8_t rxHead;
static volatile uint8_t rxTail;

static uint8_t          txBuf[ TWI_TX_BUFFER_SIZE ];
static volatile uint8_t txHead;
static volatile uint8_t txTail;



/***************************************************************************

                              local functions

***************************************************************************/



// flushes the TWI buffers

static
void
flushTwiBuffers(
  void
)
```

```
{
  rxTail = 0;
  rxHead = 0;
  txTail = 0;
  txHead = 0;
} // end flushTwiBuffers



/********************************************************************************

                                public functions

********************************************************************************/



// initialise USI for TWI slave mode

void
usiTwiSlaveInit(
  uint8_t ownAddress
)
{

  flushTwiBuffers( );

  slaveAddress = ownAddress;

  // In Two Wire mode (USIWM1, USIWM0 = 1X), the slave USI will pull SCL
  // low when a start condition is detected or a counter overflow (only
  // for USIWM1, USIWM0 = 11).  This inserts a wait state.  SCL is released
  // by the ISRs (USI_START_vect and USI_OVERFLOW_vect).

  // Set SCL and SDA as output
  // DDR_USI |= ( 1 << PORT_USI_SCL ) | ( 1 << PORT_USI_SDA );

  // set SCL high
  PORT_USI |= ( 1 << PORT_USI_SCL );

  // set SDA high
  PORT_USI |= ( 1 << PORT_USI_SDA );

  // set SCL as output
  DDR_USI |= ( 1 << PORT_USI_SCL );

  // Set SDA as input
  DDR_USI &= ~( 1 << PORT_USI_SDA );

  USICR =
       // enable Start Condition Interrupt
       ( 1 << USISIE ) |
       // disable Overflow Interrupt
       ( 0 << USIOIE ) |
```

```
       // set USI in Two-wire mode, no USI Counter overflow hold
       ( 1 << USIWM1 ) | ( 0 << USIWM0 ) |
       // Shift Register Clock Source = external, positive edge
       // 4-Bit Counter Source = external, both edges
       ( 1 << USICS1 ) | ( 0 << USICS0 ) | ( 0 << USICLK ) |
       // no toggle clock-port pin
       ( 0 << USITC );

  // clear all interrupt flags and reset overflow counter

  USISR = ( 1 << USI_START_COND_INT ) | ( 1 << USIOIF ) | ( 1 << USIPF ) | ( 1 <<
USIDC );

} // end usiTwiSlaveInit



// put data in the transmission buffer, wait if buffer is full

void
usiTwiTransmitByte(
  uint8_t data
)
{

  uint8_t tmphead;

  // calculate buffer index
  tmphead = ( txHead + 1 ) & TWI_TX_BUFFER_MASK;

  // wait for free space in buffer
  while ( tmphead == txTail );

  // store data in buffer
  txBuf[ tmphead ] = data;

  // store new index
  txHead = tmphead;

} // end usiTwiTransmitByte



// return a byte from the receive buffer, wait if buffer is empty

uint8_t
usiTwiReceiveByte(
  void
)
{

  // wait for Rx data
  while ( rxHead == rxTail );
```

```
  // calculate buffer index
  rxTail = ( rxTail + 1 ) & TWI_RX_BUFFER_MASK;

  // return data from the buffer.
  return rxBuf[ rxTail ];

} // end usiTwiReceiveByte



// check if there is data in the receive buffer

bool
usiTwiDataInReceiveBuffer(
  void
)
{

  // return 0 (false) if the receive buffer is empty
  return rxHead != rxTail;

} // end usiTwiDataInReceiveBuffer



/************************************************************************

                            USI Start Condition ISR

************************************************************************/

ISR( USI_START_VECTOR )
{

  // set default starting conditions for new TWI package
  overflowState = USI_SLAVE_CHECK_ADDRESS;

  // set SDA as input
  DDR_USI &= ~( 1 << PORT_USI_SDA );

  // wait for SCL to go low to ensure the Start Condition has completed (the
  // start detector will hold SCL low ) - if a Stop Condition arises then leave
  // the interrupt to prevent waiting forever - don't use USISR to test for Stop
  // Condition as in Application Note AVR312 because the Stop Condition Flag is
  // going to be set from the last TWI sequence
  while (
       // SCL his high
       ( PIN_USI & ( 1 << PIN_USI_SCL ) ) &&
       // and SDA is low
       !( ( PIN_USI & ( 1 << PIN_USI_SDA ) ) )
  );


  if ( !( PIN_USI & ( 1 << PIN_USI_SDA ) ) )
```

```
  {

    // a Stop Condition did not occur

    USICR =
         // keep Start Condition Interrupt enabled to detect RESTART
         ( 1 << USISIE ) |
         // enable Overflow Interrupt
         ( 1 << USIOIE ) |
         // set USI in Two-wire mode, hold SCL low on USI Counter overflow
         ( 1 << USIWM1 ) | ( 1 << USIWM0 ) |
         // Shift Register Clock Source = External, positive edge
         // 4-Bit Counter Source = external, both edges
         ( 1 << USICS1 ) | ( 0 << USICS0 ) | ( 0 << USICLK ) |
         // no toggle clock-port pin
         ( 0 << USITC );

  }
  else
  {

    // a Stop Condition did occur
    USICR =
         // enable Start Condition Interrupt
         ( 1 << USISIE ) |
         // disable Overflow Interrupt
         ( 0 << USIOIE ) |
         // set USI in Two-wire mode, no USI Counter overflow hold
         ( 1 << USIWM1 ) | ( 0 << USIWM0 ) |
         // Shift Register Clock Source = external, positive edge
         // 4-Bit Counter Source = external, both edges
         ( 1 << USICS1 ) | ( 0 << USICS0 ) | ( 0 << USICLK ) |
         // no toggle clock-port pin
         ( 0 << USITC );

  } // end if

  USISR =
       // clear interrupt flags - resetting the Start Condition Flag will
       // release SCL
       ( 1 << USI_START_COND_INT ) | ( 1 << USIOIF ) |
       ( 1 << USIPF ) |( 1 << USIDC ) |
       // set USI to sample 8 bits (count 16 external SCL pin toggles)
       ( 0x0 << USICNT0);

} // end ISR( USI_START_VECTOR )



/************************************************************************

                                USI Overflow ISR

Handles all the communication.
```

```
Only disabled when waiting for a new Start Condition.

****************************************************************************/

ISR( USI_OVERFLOW_VECTOR )
{

  switch ( overflowState )
  {

    // Address mode: check address and send ACK (and next USI_SLAVE_SEND_DATA) if OK,
    // else reset USI
    case USI_SLAVE_CHECK_ADDRESS:
      if ( ( USIDR >> 1 ) == slaveAddress )
      {
          if ( USIDR & 0x01 )
        {
          overflowState = USI_SLAVE_SEND_DATA;
        }
        else
        {
          overflowState = USI_SLAVE_REQUEST_DATA;
        } // end if
        SET_USI_TO_SEND_ACK( );
      }
      else
      {
        SET_USI_TO_TWI_START_CONDITION_MODE( );
      }
      break;

    // Master write data mode: check reply and goto USI_SLAVE_SEND_DATA if OK,
    // else reset USI
    case USI_SLAVE_CHECK_REPLY_FROM_SEND_DATA:
      if ( USIDR )
      {
        // if NACK, the master does not want more data
        SET_USI_TO_TWI_START_CONDITION_MODE( );
        return;
      }
      // from here we just drop straight into USI_SLAVE_SEND_DATA if the
      // master sent an ACK

    // copy data from buffer to USIDR and set USI to shift byte
    // next USI_SLAVE_REQUEST_REPLY_FROM_SEND_DATA
    case USI_SLAVE_SEND_DATA:
      // Get data from Buffer
      if ( txHead != txTail )
      {
        txTail = ( txTail + 1 ) & TWI_TX_BUFFER_MASK;
        USIDR = txBuf[ txTail ];
      }
      else
```

```
      {
        // the buffer is empty
        SET_USI_TO_TWI_START_CONDITION_MODE( );
        return;
      } // end if
      overflowState = USI_SLAVE_REQUEST_REPLY_FROM_SEND_DATA;
      SET_USI_TO_SEND_DATA( );
      break;

    // set USI to sample reply from master
    // next USI_SLAVE_CHECK_REPLY_FROM_SEND_DATA
    case USI_SLAVE_REQUEST_REPLY_FROM_SEND_DATA:
      overflowState = USI_SLAVE_CHECK_REPLY_FROM_SEND_DATA;
      SET_USI_TO_READ_ACK( );
      break;

    // Master read data mode: set USI to sample data from master, next
    // USI_SLAVE_GET_DATA_AND_SEND_ACK
    case USI_SLAVE_REQUEST_DATA:
      overflowState = USI_SLAVE_GET_DATA_AND_SEND_ACK;
      SET_USI_TO_READ_DATA( );
      break;

    // copy data from USIDR and send ACK
    // next USI_SLAVE_REQUEST_DATA
    case USI_SLAVE_GET_DATA_AND_SEND_ACK:
      // put data into buffer
      // Not necessary, but prevents warnings
      rxHead = ( rxHead + 1 ) & TWI_RX_BUFFER_MASK;
      rxBuf[ rxHead ] = USIDR;
      // next USI_SLAVE_REQUEST_DATA
      overflowState = USI_SLAVE_REQUEST_DATA;
      SET_USI_TO_SEND_ACK( );
      break;

  } // end switch

} // end ISR( USI_OVERFLOW_VECTOR )
```

ПРИЛОЖЕНИЕ 4. cerebro

cerebro.cpp

```
#include <sys/socket.h>
#include <sys/un.h>

#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
```

```
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <math.h>
#include <pthread.h>

pthread_mutex_t lockI2C; // mutex for critical section

// i2c.general
#define I2C_BUS                         "/dev/i2c-1"      // path to i2c bus
#define                             I2C_DIGISPARK_ADR      0x26    // The I2C
slaveAdress
// i2c.commands
#define I2C_CMD_BATTERY           0xBB    // "Check battery status" (frequency == 60
sec.)
#define I2C_CMD_CHECK             0xCC    // "Check general status" (frequency == 1
sec.)
#define I2C_CMD_BRIGTH            0xDD    // "Set display brightness" (async.)
#define I2C_CMD_SHUT              0x88    // "RPi shut down" (async.)


// function for getting "Battery charge level" (int percent 0-100)
char getBatteryChargeLvl(void);
// function for setting of display`s brightness (in percent)
int setDispBrightness(char brightValue);
// function for warning the MCU about power off
int warnMCU(void);
```

```
// function for Shutdown the System
void powerOff(void);
// function for execution of request to User
void askTheUser();

void *thrdCheckFun(void *arg);

//char *socket_path = "./socket";
char *socket_path = "\0hidden";
char status = 0;
int batteryCheckCnt = 0;
char batteryChargeLvl = 100;

int main(int argc, char *argv[]) {
                                    struct sockaddr_un addr;
                                    char buf[100];
                                    int fd,cl,rc;
                                    pthread_t thrdChecker;  // thread

                                    if (argc > 1) socket_path=argv[1];

                                    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
{
                                        perror("socket error");
                                        exit(-1);
```

```
                                    }

                                    memset(&addr, 0, sizeof(addr));
                                    addr.sun_family = AF_UNIX;
                                    if (*socket_path == '\0') {
                                        *addr.sun_path = '\0';
                                        strncpy(addr.sun_path+1, socket_path+1,
sizeof(addr.sun_path)-2);
                                    } else {
                                        strncpy(addr.sun_path, socket_path,
sizeof(addr.sun_path)-1);
                                        unlink(socket_path);
                                    }

                                    if (bind(fd, (struct sockaddr*)&addr,
sizeof(addr)) == -1) {
                                        perror("bind error");
                                        exit(-1);
                                    }

                                    if (listen(fd, 5) == -1) {
                                        perror("listen error");
                                        exit(-2);
                                    }

                                    batteryChargeLvl = getBatteryChargeLvl();  // from
0 to 100
                                    // thread. Creation and start
                                    if (pthread_create(&thrdChecker, NULL,
thrdCheckFun, NULL) != 0) {
                                        printf("Error#3: errno=%d; Could not create
check-thread;", errno);
                                        exit(-3);
                                    }

                                    while (true) {
                                        if ( (cl = accept(fd, NULL, NULL)) == -1) {
                                               perror("accept error");
                                               continue;
                                        }

                                        while ((rc=read(cl,buf,sizeof(buf))) > 0) {
                                               switch (buf[0]) {
                                                      // I2C_CMD_BATTERY-command "Check
battery status" (frequency == 60 sec.)
                                                      case I2C_CMD_BATTERY:
                                                             printf("I2C_CMD_BATTERY,
batteryChargeLvl=%d\n", batteryChargeLvl);   // from 0 to 100
                                                             if (write(cl,
&batteryChargeLvl, 1) != 1)                  // send info about Battery charge to
client
                                                                    printf("write
error;\n");
                                                             break;
```

```
                                                // I2C_CMD_CHECK-command "Check
general status" (frequency == 1 sec.)
                                                // "status" - result of
I2C_CMD_CHECK-command executed in another thread
                                                case I2C_CMD_CHECK:
                                                      printf("I2C_CMD_CHECK,
status=%d\n", status);
                                                      if (write(cl, &status, 1)
!= 1)                                  // send Status to client
                                                            printf("write
error;\n");
                                                      break;


                                                // I2C_CMD_BRIGTH-command "Set
display brightness" (async.)
                                                case I2C_CMD_BRIGTH:
                                                      printf("I2C_CMD_BRIGTH, ");


                                  pthread_mutex_lock(&lockI2C);       // lock I2C-
bus
                                                      buf[0] =
setDispBrightness(buf[1]);        // from 0 to 100

                                  pthread_mutex_unlock(&lockI2C);           //
unlock I2C-bus

                                  printf("setDispBrightness()=%d\n", buf[0]);

                                                      if (write(cl, buf, 1) != 1)
                                          // send result of operation to client
                                                            printf("write
error;\n");
                                                      break;


                                                // I2C_CMD_SHUT-command "RPi shut
down" (async.)
                                                case I2C_CMD_SHUT:
                                                      printf("I2C_CMD_SHUT, ");


                                  pthread_mutex_lock(&lockI2C);       // lock I2C-
bus
                                                      buf[0] = warnMCU();
                                          // warn the MCU about power off

                                  pthread_mutex_unlock(&lockI2C);           //
unlock I2C-bus
                                                      printf("warnMCU()=%d\n",
buf[0]);
```

```
                                                        if (write(cl, buf, 1) != 1)
                                                // send result of operation to client
                                                                printf("write
error;\n");

                                                        if (buf[0] == 0)
                                                // only if MCU is warned - turn off the
RPi
                                                                powerOff();
                                                        break;

                                                // INVALID command
                                                default:
                                                        printf("default\n");
                                                        write(cl, buf, 1);
                                                // resend bad command to client
                                                        break;
                                                }
                                                printf("read %u bytes: %.*s\n", rc, rc,
buf);
                                        }


                                        if (rc == -1) {
                                                //perror("read");
                                                exit(-1);
                                        }
                                        else if (rc == 0) {
                                                //printf("EOF\n");
                                                close(cl);
                                        }
                                }

                                // wainting finish of checker-thread
                                pthread_join(thrdChecker, NULL);
                                return 0;
}

void *thrdCheckFun(void *arg) {
                                char buf = 0;
    int i2cBusDesc = -1;
                                int ans = -1;
                                while (true){
                                        buf = I2C_CMD_CHECK;


                                // *** lockI2C ***
                                        pthread_mutex_lock(&lockI2C);

                                        // open bus
                                        if ((i2cBusDesc = open(I2C_BUS, O_RDWR)) < 0)
{
                                                printf("Failed to open the bus.\n");
```

```
                                        goto waiting;
                                    }

                                    // get access to slave with address
I2C_DIGISPARK_ADR
                                    if (ioctl(i2cBusDesc, I2C_SLAVE,
I2C_DIGISPARK_ADR) < 0) {
                                        printf("Failed to get access to the
slave (%x).\n", I2C_DIGISPARK_ADR);
                                        goto waiting;
                                    }

                                    // write cmd
                                    if (write(i2cBusDesc, &buf, 1) != 1) {
                                        printf("Failed to write to the i2c
bus.\n");
                                        goto waiting;
                                    }
                                    usleep(100000);

                                    // read answer
                                    if (read(i2cBusDesc, &buf, 1) != 1) {
                                        printf("Failed to read to the i2c
bus.\n");
                                        goto waiting;
                                    }
                                    pthread_mutex_unlock(&lockI2C);
                                // *** unlocked ***

                                    // update the status of MCU
                                    status = buf;
                                    // try to power off the RPi
                                    if ((buf & 0x02) != 0x02)
                                        askTheUser();

                                    // check battery status each 60 sec
                                    if (batteryCheckCnt >= 60) {
                                        pthread_mutex_lock(&lockI2C);
                                    // lock I2C-bus
                                        batteryChargeLvl =
getBatteryChargeLvl();          // from 0 to 100
                                        pthread_mutex_unlock(&lockI2C);
                                    // unlock I2C-bus
                                        batteryCheckCnt = 0;
                                    }

                                    // power off the RPi if Battery is finished
                                    if (batteryChargeLvl <= 10)
                                        powerOff();

waiting:
                                    usleep(1000000);   // 1sec
                                    batteryCheckCnt++;
                                }
```

```
}


// function for getting "Battery charge level" (int percent 0-100)
char getBatteryChargeLvl(void) {
                                    int i2cBusDesc = -1;    // descriptor of MCU
Digispark on the I2C-bus
                                    int batteryLvl = -1;    // battery charge level
                                    char buf[2] = {0};            // buffer for
transmission through I2C

                                    // open bus
                                    if ((i2cBusDesc = open(I2C_BUS, O_RDWR)) < 0)
                                        return -1;   // Failed to open the bus

                                    // get access to slave with address
I2C_DIGISPARK_ADR
                                    if (ioctl(i2cBusDesc, I2C_SLAVE,
I2C_DIGISPARK_ADR) < 0)
                                        return -2;   // Failed to get access to the
slave (I2C_DIGISPARK_ADR)

                                    // write cmd
                                    buf[0] = (int)I2C_CMD_BATTERY;
                                    if (write(i2cBusDesc, buf, 1) != 1)
                                        return -3;   // Failed to write to the i2c
bus.

                                    // time-delay (waiting ADC)
                                    usleep(400000);

                                    // read answer
                                    if (read(i2cBusDesc, buf, 2) != 2)
                                        return -4;   // Failed to read to the i2c bus

                                    // convert ADC data and return "Battery charge
level"
                                    batteryLvl = ((int)buf[1] & 0x00FF) | ((int)buf[0]
<< 8 & 0xFF00);                     // ADC data (0-1024)
                                    return (char)(((double) batteryLvl) / 10.24);
                                    // "Battery charge level"
}


// function for setting of display`s brightness (in percent)
int setDispBrightness(char brightValue) {
                                    // check asserts
                                    if (brightValue > 100)  brightValue = 100;
                                    if (brightValue < 0)    brightValue = 0;



                                    int i2cBusDesc = -1;    // descriptor of MCU
Digispark on the I2C-bus
```

```
                                    char buf[2] = {0};            // buffer for
transmission through I2C
                                    int converter = 255*brightValue/100;

                                    // open bus
                                    if ((i2cBusDesc = open(I2C_BUS, O_RDWR)) < 0)
                                        return -1;   // Failed to open the bus

                                    // get access to slave with address
I2C_DIGISPARK_ADR
                                    if (ioctl(i2cBusDesc, I2C_SLAVE,
I2C_DIGISPARK_ADR) < 0)
                                        return -2;   // Failed to get access to the
slave (I2C_DIGISPARK_ADR)

                                    // clear i2c bus
                                    for (int i = 0; i<10; i++)
                                        read(i2cBusDesc, buf, 1);

                                    // write cmd
                                    buf[0] = (int)I2C_CMD_BRIGTH;
                                    buf[1] = (char) converter;//( ((int)brightValue) *
255) / 100;
                                    if (write(i2cBusDesc, buf, 2) != 2)
                                        return -3;   // Failed to write to the i2c
bus.

                                    // time-delay (waintig of execution)
                                    usleep(300000);

                                    // read answer
                                    if (read(i2cBusDesc, buf, 1) != 1)
                                        return -4;   // Failed to read to the i2c bus

                                    // check result
                                    if (buf[0] != (char) converter) {
                                        printf("Brigness: try to set  - %d\n",
(char)converter);
                                        printf("Brigness: as a result - %d\n",
(char)buf[0]);
                                        return -5;   // Failed to set brightness
                                    }
                                    // time-delay (measures for protection of I2C-bus)
                                    usleep(200000);

                                    return 0;
}


// function for warning the MCU about power off
int warnMCU(void) {
                                    int i2cBusDesc = -1;    // descriptor of MCU
Digispark on the I2C-bus
```

```
                                    char buf[2] = {0};            // buffer for
transmission through I2C
                                    int ans = -1;                 // answer by MCU. It
has to be an equivalent of "brightValue"

                                    // open bus
                                    if ((i2cBusDesc = open(I2C_BUS, O_RDWR)) < 0)
                                        return -1;   // Failed to open the bus

                                    // get access to slave with address
I2C_DIGISPARK_ADR
                                    if (ioctl(i2cBusDesc, I2C_SLAVE,
I2C_DIGISPARK_ADR) < 0)
                                        return -2;   // Failed to get access to the
slave (I2C_DIGISPARK_ADR)

                                    // write cmd
                                    buf[0] = (int)I2C_CMD_SHUT;
                                    if (write(i2cBusDesc, buf, 1) != 1)
                                        return -3;   // Failed to write to the i2c
bus.

                                    // time-delay (waintig of execution)
                                    usleep(300000);

                                    // read answer
                                    if (read(i2cBusDesc, buf, 1) != 1)
                                        return -4;   // Failed to read to the i2c bus

                                    // check result
                                    if ((int)buf[0] != I2C_CMD_SHUT)
                                        return -5;   // Failed to power off

                                    // time-delay (measures for protection of I2C-bus)
                                    usleep(200000);

                                    return 0;
}


// function for Shutdown the System
void powerOff(void) {
                                    printf("shutdown -h now");
                                    system("shutdown -h now");    // stump
}


// function for execution of request to User
void askTheUser() {
                                    printf("Are you shure? (*stump)");
                                    return;
}
```

cerebro.service

```
[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
```

make

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

clean:
	rm cerebro
```