Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчет о лабораторной работе

Курс: Проектирование ОС и компонентов

Тема: Разработка прикладного слоя для bluetooth

Выполнил студент группы 13541/3 _____ Д.В. Круминьш
(подпись)

Преподаватель _____ Е.В. Душутина
(подпись)

Санкт-Петербург
2018 г.

Содержание

1 Цель работы

Анализ архитектуры bluetooth-драйвера в ОС Linux и принципа работы прикладного уровня. Модификация прикладного уровня, встраивание его в систему и проверка модификаций.

2 Сведения о системе

Работа производилась на виртуальной системе - **Ubuntu 16.04**, с использованием **VMware Workstation 12.5.7**.
Версия ядра - **4.13.0-38**.
Версия BlueZ - **5.37**

В качестве bluetooth-устройства использовался встроенный bluetooth-адаптер на компьютере хосте.

Для того чтобы виртуальная система имела доступ к bluetooth-адаптер, в настройках виртуальной системы необходимо включить настройку - **Share Bluetooth devices with virtual machine**.

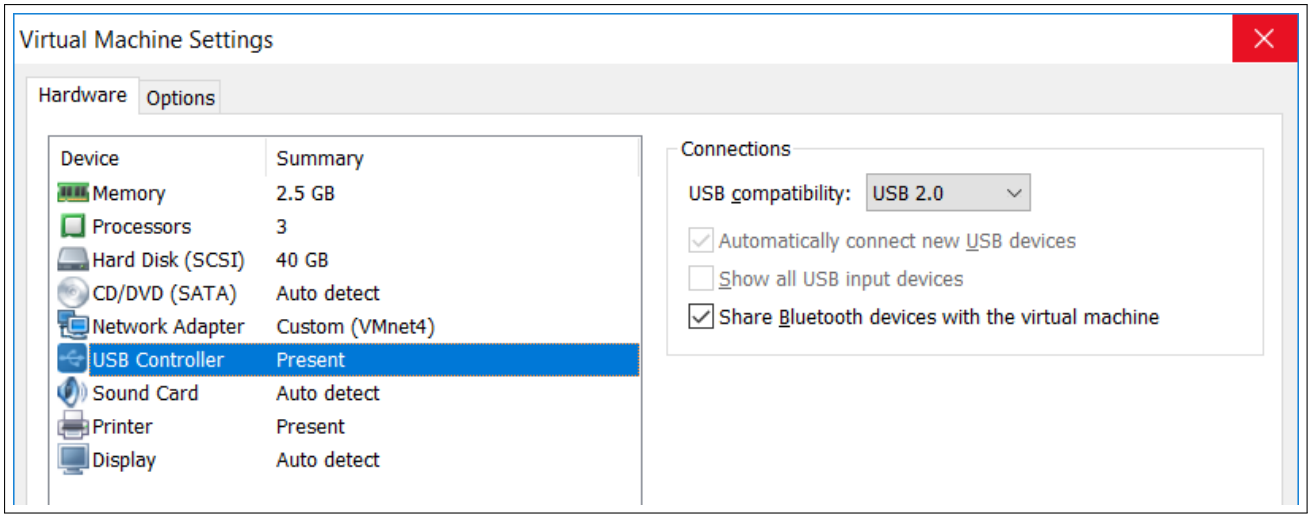

Рис. 1: Настройки виртуальной системы

После чего, на виртуальной системе будет доступен bluetooth-адаптер.

3 Выполнение работы

3.1 Обзор BlueZ

BlueZ — стек технологии Bluetooth для Linux. Его цель состоит в том, чтобы сделать реализацию спецификаций стандартов технологии Bluetooth для Linux. Стек BlueZ поддерживает все основные протоколы и уровни Bluetooth. Был первоначально разработан Qualcomm, и доступен для ядра Linux версии 2.4.6 и выше. Во всех, дружественных к пользователю, дистрибутивах Linux, он встроен по умолчанию. На рассматриваемой ОС **Ubuntu 16.04** в том числе.

Данный стек покрывает как пространство ядра так и прикладной уровень.

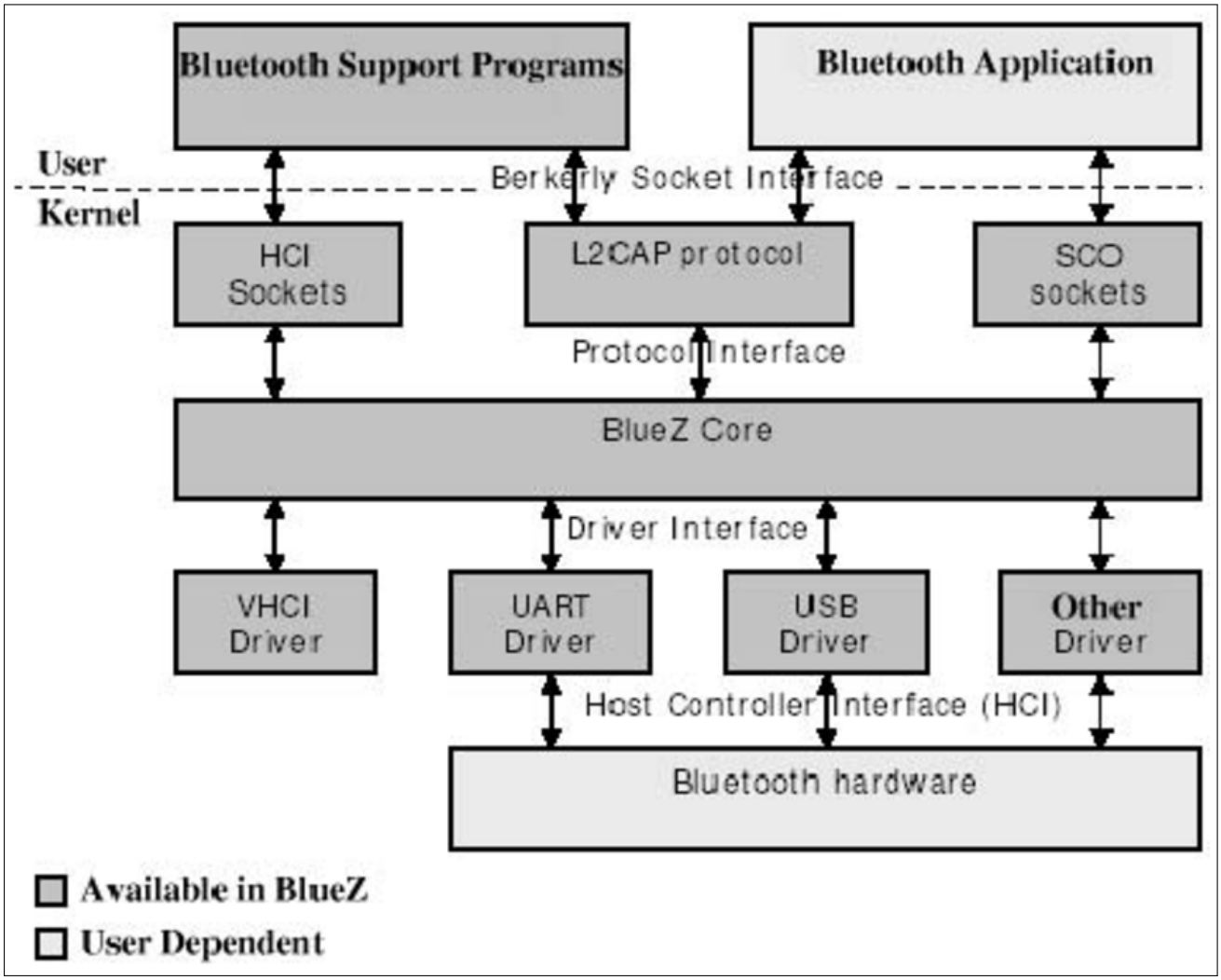

Рис. 2: Архитектура Bluez

В пространстве ядра, уровень **BlueZ Core** абстрагирует аппаратно-зависимый слой че-

рез единый интерфейс.

Bluetooth-адаптер может быть подключен через:

- **VHCI** - Virtual Host Controller Interface, имитация реального устройства;
- **UART** - Универсальный Асинхронный Приёмопередатчик;
- **USB**;
- или как-либо иначе.

То есть, как бы устройство не было подключено, интерфейс не изменится.

Интерфейс(**BlueZ Core**) используется различными протоколами:

- **HCI** - хост-контроллер;
- **L2CAP** - протокол управления логикой и адаптацией.

Стек поставляется со следующими программами:

- **bluetoothctl** - программа-интерфейся для работы с bluetooth;
- **bluetoothd** - демон bluetooth;
- **btmon** - обеспечивает доступ к инфраструктуре монитора подсистемы Bluetooth для чтения трассировки HCI;
- **hciconfig** - конфигурация bluetooth-адаптера;
- **hcidump** - считывание необработанных данных HCI, поступающих на устройство Bluetooth. Вывод на экран команд, событий и данных в удобочитаемой форме.
- ...

В данной работе будет рассмотрен **bluetoothctl** - интерфейс bluetooth на прикладном уровне.

3.2 Обзор bluetoothctl

Для запуска утилиты, в консоли необходимо ввести **bluetoothctl**. Далее приведено выполнение команд: **version, help, scan**.

```
1 psaer@ubuntu:/usr/src/linux-headers-4.13.0-36$ bluetoothctl
2 [NEW] Controller 34:F6:4B:36:FD:43 ubuntu [default]
3 [bluetooth]# version
4 Version 5.37
5 [bluetooth]# help
```

```
Available commands:
  list                       List available controllers
  show [ctrl]                Controller information
  select <ctrl>              Select default controller
  devices                    List available devices
  paired-devices             List paired devices
  power <on/off>             Set controller power
  pairable <on/off>          Set controller pairable mode
  discoverable <on/off>      Set controller discoverable mode
  agent <on/off/capability>  Enable/disable agent with given capability
  default-agent              Set agent as the default one
  set-scan-filter-uuids [uuid1 uuid2 ...] Set scan filter uuids
  set-scan-filter-rssi [rssi] Set scan filter rssi, and clears pathloss
  set-scan-filter-pathloss [pathloss] Set scan filter pathloss, and clears rssi
  set-scan-filter-transport [transport] Set scan filter transport
  set-scan-filter-clear      Clears discovery filter.
  scan <on/off>              Scan for devices
  info [dev]                 Device information
  pair [dev]                 Pair with device
  trust [dev]                Trust device
  untrust [dev]              Untrust device
  block [dev]                Block device
  unblock [dev]              Unblock device
  remove <dev>               Remove device
  connect <dev>              Connect device
  disconnect [dev]           Disconnect device
  list-attributes [dev]      List attributes
  select-attribute <attribute> Select attribute
  attribute-info [attribute] Select attribute
  read                       Read attribute value
  write <data=[xx xx ...]>   Write attribute value
  notify <on/off>            Notify attribute value
  register-profile <UUID ...> Register profile to connect
  unregister-profile         Unregister profile
  version                    Display version
  quit                       Quit program
[bluetooth]# scan on
Discovery started
[CHG] Controller 34:F6:4B:36:FD:43 Discovering: yes
[NEW] Device 14:F4:2A:7B:1E:D9 GT-N7100
[bluetooth]# devices
Device 14:F4:2A:7B:1E:D9 GT-N7100
[bluetooth]# quit
[DEL] Controller 34:F6:4B:36:FD:43 ubuntu [default]
```

Листинг 1: Лог bluetoothctl

Внимание стоит уделить команде **scan**, которая была запущена с ключем **on**, то есть было запущено сканирование.

Далее, на телефоне был включен bluetooth, и в логе появилась запись

[NEW] Device 14:F4:2A:7B:1E:D9 GT-N7100

То есть сканер, успешно обнаружил мой, личный телефон, вывел его mac-адрес и название устройства.

Для того, чтобы разобраться в функционировании прикладного уровня, предполагается сделать следующие модификации:

- Изменение версии;
- При сканировании устройств, автоматическое определение mac-адреса личного телефона, и вывод соответствующего сообщения.

3.3 Модификация прикладного уровня - bluetoothctl

Был скачан и распакован архив, по следующей ссылке:

http://www.kernel.org/pub/linux/bluetooth/bluez-5.37.tar.xz

Далее, необходимо понять, где находится отправная точка для модификации. Полный список команд, из предудыщего листинга был выведен не просто-так. Возьмем какую-либо специфичную команду, например - **set-scan-filter-clear**. Далее применим следующую команду:

sudo grep -rnw '/home/psaer/bluez-5.37' -e 'set-scan-filter-clear'

Которая рекурсивно ищет среди содержимого файлов нужную строку, в данном случае, выбранууую специфичную команду.

```
1 psaer@ubuntu:/usr/src/linux-headers-4.13.0-36$ sudo grep -rnw '/home/psaer/
    ↪ bluez-5.37' -e 'set-scan-filter-clear'
2 Binary file /home/psaer/bluez-5.37/client/bluetoothctl matches
3 Binary file /home/psaer/bluez-5.37/client/main.o matches
4 /home/psaer/bluez-5.37/client/main.c:1716:   { "set-scan-filter-clear", "",
    ↪ cmd_set_scan_filter_clear,
```

Листинг 2: Лог поиска

Как видно из лога, данную запись содержит файл **/client/main.c** в строке 1716.

```
1674 static const struct {
1675     const char *cmd;
1676     const char *arg;
1677     void (*func) (const char *arg);
1678     const char *desc;
1679     char * (*gen) (const char *text, int state);
1680     void (*disp) (char **matches, int num_matches, int max_length);
1681 } cmd_table[] = {
1682     { "list",         NULL,       cmd_list, "List available controllers" },
1683     { "show",         "[ctrl]",   cmd_show, "Controller information",
1684                                   ctrl_generator },
1685     { "select",       "<ctrl>",   cmd_select, "Select default controller",
1686                                   ctrl_generator },
1687     { "devices",      NULL,       cmd_devices, "List available devices" },
1688     { "paired-devices", NULL,     cmd_paired_devices,
1689                         "List paired devices"},
1690     { "system-alias", "<name>",   cmd_system_alias },
1691     { "reset-alias",  NULL,       cmd_reset_alias },
```

Листинг 3: .../original/client/main.c

Как видно из отрывка кода, в 1674 строке происходит инициализация структуры по обработке cmd команд. Соответственно для каждого названия функции, приведена функция с реализацией.

3.3.1 Модификация версии

За вывод версии отвечает функция **cmd_version**, откроем её.

```
1598 static void cmd_version(const char *arg)
1599 {
1600     rl_printf("Version %s\n", VERSION);
1601 }
```

Листинг 4: .../original/client/main.c

Как видно из реализации, используется лишь функция **rl_printf**. Использование данной функции вместо стандартного **printf** обусловлено лучшей работой при асинхронных вызовах.

Модифицируем функцию следующим образом:

```
1603 static void cmd_version(const char *arg)
1604 {
```

```
1605     rl_printf("Custom Version %s\n", VERSION);
1606 }
```

Листинг 5: .../modified/client/main.c

3.3.2 Модификация по определению личного телефона

Рассмотрим команду **cmd_scan**.

```
880 static void cmd_scan(const char *arg)
881 {
882     dbus_bool_t enable;
883     const char *method;
884 
885     if (parse_argument_on_off(arg, &enable) == FALSE)
886         return;
887 
888     if (check_default_ctrl() == FALSE)
889         return;
890 
891     if (enable == TRUE)
892         method = "StartDiscovery";
893     else
894         method = "StopDiscovery";
895 
896     if (g_dbus_proxy_method_call(default_ctrl, method,
897                 NULL, start_discovery_reply,
898                 GUINT_TO_POINTER(enable), NULL) == FALSE) {
899         rl_printf("Failed to %s discovery\n",
900                     enable == TRUE ? "start" : "stop");
901         return;
902     }
903 }
```

Листинг 6: .../original/client/main.c

Как видно из реализации, спервa выполняются различные проверки и подготовки, после чего вызывается функция **g_dbus_proxy_method_call**.

Рассмотрим эту функцию.

```
839 gboolean g_dbus_proxy_method_call(GDBusProxy *proxy, const char *method,
840                 GDBusSetupFunction setup,
841                 GDBusReturnFunction function, void *user_data,
842                 GDBusDestroyFunction destroy)
```

```
843 {
844     struct method_call_data *data;
845     GDBusClient *client;
846     DBusMessage *msg;
847     DBusPendingCall *call;
848 
849     if (proxy == NULL || method == NULL)
850         return FALSE;
851 
852     client = proxy->client;
853     if (client == NULL)
854         return FALSE;
855 
856     data = g_try_new0(struct method_call_data, 1);
857     if (data == NULL)
858         return FALSE;
859 
860     data->function = function;
861     data->user_data = user_data;
862     data->destroy = destroy;
863 
864     msg = dbus_message_new_method_call(client->service_name,
865                 proxy->obj_path, proxy->interface, method);
866     if (msg == NULL) {
867         g_free(data);
868         return FALSE;
869     }
870 
871     if (setup) {
872         DBusMessageIter iter;
873 
874         dbus_message_iter_init_append(msg, &iter);
875         setup(&iter, data->user_data);
876     }
877 
878     if (g_dbus_send_message_with_reply(client->dbus_conn, msg,
879                     &call, METHOD_CALL_TIMEOUT) == FALSE) {
880         dbus_message_unref(msg);
881         g_free(data);
882         return FALSE;
883     }
884 
885     dbus_pending_call_set_notify(call, method_call_reply, data, g_free);
886     dbus_pending_call_unref(call);
887 
```

```
888     dbus_message_unref(msg);
889
890     return TRUE;
891 }
```

Листинг 7: .../original/gdbus/client.c

В данной реализации, многое завязано на **d-bus(системе межпроцессного взаимодействия)**. Дальнейший анализ затруден, так как из кода ясно с какими компонентами происходит дальнейшие взаимодействия, поэтому продолжим анализ не заглядывая так глубоко.

При вызове функции **g_dbus_proxy_method_call**, одним из её аргуменов является функция - **start_discovery_reply**

```
863 static void start_discovery_reply(DBusMessage *message, void *user_data)
864 {
865     dbus_bool_t enable = GPOINTER_TO_UINT(user_data);
866     DBusError error;
867
868     dbus_error_init(&error);
869
870     if (dbus_set_error_from_message(&error, message) == TRUE) {
871         rl_printf("Failed to %s discovery: %s\n",
872                 enable == TRUE ? "start" : "stop", error.name);
873         dbus_error_free(&error);
874         return;
875     }
876
877     rl_printf("Discovery %s\n", enable == TRUE ? "started" : "stopped");
878 }
```

Листинг 8: .../original/client/main.c

Исходя из реализации, функция ожидает успешного или не успешного **discovery** - действия означающего успешный запуск сканирования.

На этом весь прямой поиск заканчивается, более никаких функций, вызывающих интерес не найдено, поэтому применим обратный поиск. Найдем функцию, отвечающую за вывод нового найденного устройства.

Данной функцией оказалась - **print_device** из файла **/client/main.c**.

```
133 static void print_device(GDBusProxy *proxy, const char *description)
134 {
135     DBusMessageIter iter;
```

```
136     const char *address, *name;
137
138     if (g_dbus_proxy_get_property(proxy, "Address", &iter) == FALSE)
139         return;
140
141     dbus_message_iter_get_basic(&iter, &address);
142
143     if (g_dbus_proxy_get_property(proxy, "Alias", &iter) == TRUE)
144         dbus_message_iter_get_basic(&iter, &name);
145     else
146         name = "<unknown>";
147
148     rl_printf("%s%s%sDevice %s %s\n",
149                 description ? "[" : "",
150                 description ? : "",
151                 description ? "] " : "",
152                 address, name);
153 }
```

Листинг 9: .../original/client/main.c

Как видно из реализации, происходит межпроцессное взаимодействие, в ходе которого идет идентификация устройства: получения его адреса, и названия.

Данную функцию вызывает функция **proxy_added** из файла **/client/main.c**.

```
324 static void proxy_added(GDBusProxy *proxy, void *user_data)
325 {
326     const char *interface;
327
328     interface = g_dbus_proxy_get_interface(proxy);
329
330     if (!strcmp(interface, "org.bluez.Device1")) {
331         if (device_is_child(proxy, default_ctrl) == TRUE) {
332             dev_list = g_list_append(dev_list, proxy);
333
334             print_device(proxy, COLORED_NEW);
335         }
336     } else if (!strcmp(interface, "org.bluez.Adapter1")) {
337         ctrl_list = g_list_append(ctrl_list, proxy);
338
339         if (!default_ctrl)
340             default_ctrl = proxy;
341
342         print_adapter(proxy, COLORED_NEW);
343     } else if (!strcmp(interface, "org.bluez.AgentManager1")) {
```

```
344         if (!agent_manager) {
345             agent_manager = proxy;
346 
347             if (auto_register_agent)
348                 agent_register(dbus_conn, agent_manager,
349                             auto_register_agent);
350         }
351     } else if (!strcmp(interface, "org.bluez.GattService1")) {
352         if (service_is_child(proxy))
353             gatt_add_service(proxy);
354     } else if (!strcmp(interface, "org.bluez.GattCharacteristic1")) {
355         gatt_add_characteristic(proxy);
356     } else if (!strcmp(interface, "org.bluez.GattDescriptor1")) {
357         gatt_add_descriptor(proxy);
358     } else if (!strcmp(interface, "org.bluez.GattManager1")) {
359         gatt_add_manager(proxy);
360     }
361 }
```

Листинг 10: .../original/client/main.c

Как видно из реализации, происходит сравнение интерфейсов, далее либо вывод информации в конось или дополнительные обработки нового устройства.

Теперь узнаем где используется функция **proxy_added**.

```
2059     g_dbus_client_set_proxy_handlers(client, proxy_added, proxy_removed,
2060                                 property_changed, NULL);
```

Листинг 11: .../original/client/main.c

Как видно, данная функция используется в функции **g_dbus_client_set_proxy_handlers** по межпроцессному взаимодействию. То есть, при инициализации программы, на уровне межпроцессного взаимодействия инициализируется обработчик нового устройства.

Для модификации, достаточно изменить функцию **print_device**.

```
134 static void print_device(GDBusProxy *proxy, const char *description)
135 {
136     DBusMessageIter iter;
137     const char *address, *name;
138 
139     if (g_dbus_proxy_get_property(proxy, "Address", &iter) == FALSE)
140         return;
141 
142     dbus_message_iter_get_basic(&iter, &address);
143 
```

```
144     if (g_dbus_proxy_get_property(proxy, "Alias", &iter) == TRUE)
145         dbus_message_iter_get_basic(&iter, &name);
146     else
147         name = "<unknown>";
148 
149     if (strcmp(address, "14:F4:2A:7B:1E:D9")==0){
150         rl_printf("[DENIS] My phone is connected!\n");
151     }
152 
153     rl_printf("%s%s%sDevice %s %s\n",
154                 description ? "[" : "",
155                 description ? : "",
156                 description ? "] " : "",
157                 address, name);
158 }
```

Листинг 12: .../modified/client/main.c

Также, в этот же файл было добавлено подключение **#include <string.h>**.

```
36 #include <string.h>
```

Листинг 13: .../modified/client/main.c

С помощью функции **strcmp** происходит сравнение адреса найденного устройства, с предварительно заданным устройством(адресом телефона). И в случае совпадения адресов, выводится соответствующее сообщение.

3.4 Сборка модифицированного прикладного уровня

Перед сборкой, необходимо установить следующие зависимости:

```
1 sudo apt-get install libdbus-1-dev
2 sudo apt-get install libudev-dev
3 sudo apt-get install libical-dev
4 sudo apt-get install libreadline-dev
```

Листинг 14: Зависимости

Далее, с помощью команд **./configure, make, make install** происходит конфигурация, сборка и установка. Логи приложены.

3.5 Тестирование

```
psaer@ubuntu:/usr/src/linux-headers-4.13.0-36$ bluetoothctl
[NEW] Controller 34:F6:4B:36:FD:43 ubuntu [default]
[bluetooth]# version
Custom Version 5.37
[bluetooth]# scan on
Discovery started
[CHG] Controller 34:F6:4B:36:FD:43 Discovering: yes
[NEW] Device D0:66:7B:29:49:A3 DTVBluetooth
[DENIS] My phone is connected!
[NEW] Device 14:F4:2A:7B:1E:D9 GT-N7100
[bluetooth]# quit
[DEL] Controller 34:F6:4B:36:FD:43 ubuntu [default]
```

Листинг 15: Лог bluetoothctl

Как видно из лога:

- текст версии утилиты изменился;
- при включении bluetooth на телефоне, вывелось соответствующее сообщение.

Вывод

В данной работе была рассмотрена структура драйвера bluetooth. Был рассмотрен прикладной слой, а также произведены модификации кода, с последующим тестированием.

В отличии от драйвера символьного устройства, в данном случае, на мой взгляд, поддержка драйвера является более сложной задачей, в частности из-за отсутствия документации или каких-либо комментариев в коде. С чем я и столкнулся в данной работе(межпроцессное взаимодействие). В таком случае, для понимания функционирования, необходима отладка и много времени.

В целом, архитектура **BlueZ** является универсальной, то-есть прикладной интерфейс останется не изменнным, при различных подключениях bluetooth-адаптера.

Список литературы

[1] BlueZ-5.37. — URL: http://www.linuxfromscratch.org/blfs/view/7.9/general/bluez.html (дата обращения: 2018-05-26).

[2] BlueZ about. — URL: http://www.bluez.org/about/ (дата обращения: 2018-05-26).

[3] BlueZ 5 API introduction and porting guide. — URL: http://www.bluez.org/bluez-5-api-introduction-and-porting-guide/ (дата обращения: 2018-05-26).

[4] Bluetooth Development Notes. — URL: http://rrbluetoothx.blogspot.ru/2016/04/rr-bluetooth-compile-bluez-539.html (дата обращения: 2018-05-26).