Bluetooth Stack and how Linux OS handles it

Technical Report · July 2020

DOI: 10.13140/RG.2.2.10109.18402

1 author:

Alessandro Cingolani
Sapienza University of Rome
6 PUBLICATIONS 0 CITATIONS
SEE PROFILE

Bluetooth Stack and how Linux OS handles it

Alessandro Cingolani

A.Y. 2019/2020

Contents

1 Introduction

Bluetooth is a wireless technology standard used for exchanging data between fixed and mobile devices over short distances using short-wavelength UHF radio waves in the industrial, scientific and medical radio bands, from 2.402 GHz to 2.480 GHz, and building personal area networks (PANs). Due to its wide adoption, no modern Operating Systems can deny support to this interface.

The goal of this essay is to illustrate the internal organization of the Linux Bluetooth Stack, with a special focus on the support of audio devices.

Nowadays the official Linux Bluetooth stack is implemented by *BlueZ*[1]. Since 2006, this package provides supports for all core layers and protocols implementing a complete Bluetooth stack. The whole system consists of many separates modules that allows to customize and optimize kernel by inserting only the strictly required functionality (For example, an embedded device will rarely need Bluetooth High-speed features).

As already said, the kernel module is included in the Linux kernel since the 2.6 version[2], however, without the support of user-space packages these modules are quite useless since they only implement interfaces to functionalities (in other words, kernel module are just API that user-level application should use). The user-land package `bluez` provides all the tools and daemons needed by Bluetooth applications.

2 Bluetooth Working Mechanisms

Bluetooth stack shares many similarities with the classical TCP/IP stack, however, the underlying concepts are completely different. In fact, Bluetooth just focuses on communication with physically proximate devices whereas Internet protocols like TCP doesn't really care if the destination device is within a 30 meters range or on the other side of the planet. For that reason, the process of establishing a new connection depends on whether the device wants to create an *incoming* or an *outgoing* connection. In few words, a device that initiate an outgoing connection needs to choose a target device and transport protocol, establish a connection and transfer data. Similarly, in order to establish an incoming connection a device choose a transport protocol, and then listen before accepting a connection and transferring data.

2.1 Device Name

The TCP/IP stack makes use of two different addresses to identify a device in a network: the first one is the *MAC address* and it is used in the second layer of the ISO/OSI stack, and the other one is the *IP address*, that is used in every layer up to 3. In the same way, also the Bluetooth stack makes use of the hardware MAC address but, instead, it does not need an higher layer address as across the entire stack a device is identified only by its 48-bit address, referred as *Device Address*[3].
The basic data structure used in BlueZ to represent such addresses is the `bdaddr_t` structure.

Listing 1: Device address structure in `bluetooth.h`

```c
/* BD Address */
typedef struct {
        uint8_t b[6];
} __attribute__((packed)) bdaddr_t;
```

Here the 48-bit address is splitted into a byte array of six elements. The type `uint8_t` is used to explicitly specify the size of a variable: in many architecture the the value of `sizeof(int)` could vary

[1] http://www.bluez.org/
[2] Located into `net/bluetooth`
[3] Also *Bluetooth Address*

from 8 to 64 bit, so by using this type the compiler is forced to store the integer into 8 bit.
On the other hand, the "*packed*"[4] attributes is a compiler options that specifies that no *padding optimization* should be performed on the specified structure. This is probably done, apart for saving space (useful in embedded device), to avoid possible padding difference between different compilers and to optimize data transmission (in this case developers must take care of protocol's endianess). This trick will be widely used across all BlueZ code.
All Bluetooth addresses in BlueZ are stored and manipulated as `bdaddr_t` structures and, for this reason, there are two helper functions that converts this structure to a string and vice versa[5].

```
int str2ba (const char *str, bdaddr_t *ba);
int ba2str(const bdaddr_t *ba, char *str)
```

2.2 Scanning for Devices

How should a Bluetooth device advertize itself? Or how it can discover the presence of nearby devices? Differently from other radio protocols, a Bluetooth device will never "*announces*" its presence by broadcasting some packets. Instead, the only way for a device to discover the presence of another one is to conduct a *device discovery.*
In fact, for both privacy and power concerns, all Bluetooth devices have two flags that specifies how they should behave:

1. **Inquiry Scan**: determine if the device is detectable by other Bluetooth devices

2. **Page Scan**: determine if the device should accept incoming connections requests.

By setting these flags it is possible to specify the degree of interaction that a Bluetooth controller should have. For example, due to security and privacy reasons, the default configuration on Android OS have the first flag turned off and the other one enabled: in this way an device scan will not reveal the smartphone, but devices that already know the destination's bluetooth address can initiate a connection.

2.3 Types of Protocols

Similarly to the TCP/IP stack, once an addressing mechanism is put in place, there is a need for a transport protocol. The Bluetooth specifications supports several types of protocols, however only few of them are widely used in practice and worth to mention.

2.3.1 RFCOMM

The *Radio Frequency Communications* is a protocols used to create *reliable* and *stream-based* channels, being compared for this reason to the TCP. Although they are quite similar in some aspects, the RFCOMM has a smaller set of available ports (only 30) compared to TCP (65,535).
Every RFCOMM connection is actually incapsulated within a L2CAP connection.

2.3.2 L2CAP

The *Logical Link Control and Adaption Protocol* (L2CAP) is commonly considered "*the UDP of bluetooth*" and, although they share some characteristics, they are quite different in reality. The first difference is that, unlike UDP, L2CAP enforce *delivery order*[6]. In addition to that, L2CAP can be configured to achieve

[4]"packed" GCC reference
[5]Strings must be in the format "XX:XX:XX:XX:XX:XX"
[6]In the UDP protocol, if two packets were sent sequentially, there is no guarantee that those packet will arrive in the same sequential order

varying levels of reliability. This property is implemented via a transmit/acknowledgment scheme in which unacknowledge packets may or may not be retransmitted. Three possible retransmit policies are available:

- Never Retransmit
- Always retransmit until success or connection failure limit
- Drop packet and move on to the queued data

The drawback of this customization, is that any changes in L2CAP policies inevitably affects the upper RFCOMM protocol since it is encapsulated inside.
L2CAP supports far more port (called "*Protocol Service Multiplexer*") than RFCOMM, in particular it can use all odd-numbered value between 1 and 32.767.
The L2CAP could also be converted into a *best-effort* protocol, as shown in [4.3.1].

2.3.3 SDP

Previous transport protocols make use of the concept of *ports*, which is also common to the TCP/IP stack. However, the size of available port of Bluetooth protocols is quite low compared to the TCP/UDP counterpart, so a scenario where two application tries to run on the same port becomes more common. To solve this issue, Bluetooth introduced the *Service Discovery Protocol* (**SDP**): every Bluetooth device maintains an SDP server listening on a well-known port number; applications that want to use a particular port must register itself on the SDP server. Then, when a remote client connects to a device, it searches for the SDP server which will provide it all the available services.

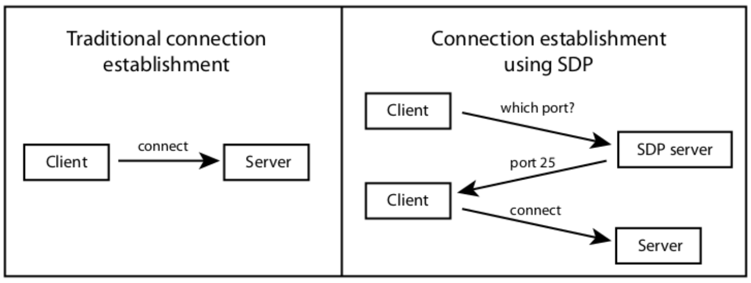


Figure 1: Comparison between traditional network applications that uses hard-coded port number and Bluetooth applications that employs SDP

2.3.4 SCO

The *Synchronous Connection Oriented* logical transport (SCO) is a best-effort packed based protocol that is exclusively used to transmit *voice-quality* audio. The main characteristic is that audio quality is fixed to exactly 64 kb/s to guarantee the "perfect" for processing human voices. Obviously such low transmission rate excludes applications such as high-quality audio (even a mid-range MP3 has a bitrate of 128 kb/s), so generally on devices such as Bluetooth headphones, audio transmission is managed by L2CAP whereas phone call transit through SCO.
In fact, despite SCO packets are not reliable and never retransmitted, the protocol provides a much more important quality of service assumption: any SCO connection is always guaranteed to have a 64 kb/s transmission rate and, if other application are contending for radio time, then SCO are prioritized.

2.3.5 ACL

The *Asynchronous Connection-oriented Logical* can be considered one of the lowest-level protocol present in Bluetooth. It is pretty similar to SCO, but ACL has a retransmission mechanism since it is used exclusively to transport data. In fact, we can see from figure [??] that ACL packets' task is basically to encapsulates all previous packets (from RFCOMM to HCI). In particular the image shows the process of fragmentation between different protocols/layers.
Generally, two Bluetooth devices cannot have more than one ACL and SCO connection opened (the latter one is to ensure QoS).

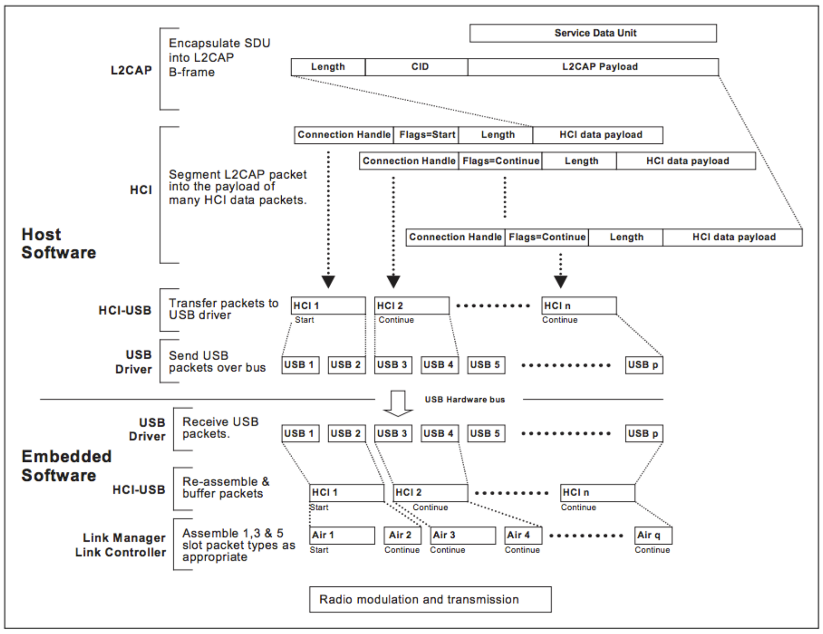


Figure 7.2: Example of fragmentation processes in a device with a BR/EDR Controller and USB HCI transport

2.4 Profiles

Bluetooth profiles can be viewed as a set of rules that enable particular services. They were developed in order to describe how implementation of user functionalities should be accomplished, avoiding the risk of interoperability problems between products of different vendors.
Profiles are placed on top of the Bluetooth stack (Application Level) because they define options for each lower-level protocol that are mandatory for the requested profile.
In addition to that, a profile can *depend* on another profile if it uses some part of it. For example, the *File Transfer Protocol* (FTP) profile is dependent on *Generic Object Exchange* (GOEP), *Serial Port* (SPP) and *Generic Access Profile* as shown in figure [2].
The official list[7] contains a lot of profiles, some of which are:

- *Advanced Audio Distribution Profile* (**A2DP**): defines how multimedia audio can be streamed from one device to another. Will be studied later at 5.4
- *Audio/Video Remote Control Profile* (**AVRCP**): provide a standard interface to control TVs, Hi-fi equipment, etc. Commonly used in conjuncton with A2DP.

[7]Available at the official Bluetooth website

- *Human Interface Device Profile* (**HID**): provides support for devices such as mice and keyboards. It is designed to provide a low latency link, with low power requirements.[8]
- *Mesh Profile* (**MESH**): allows *many-to-many* communication over Bluetooth radio. Most suitable for IoT devices.

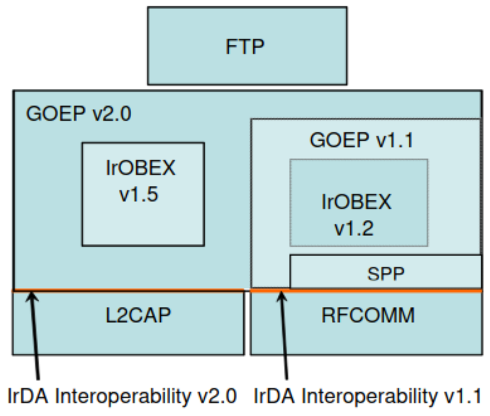


Figure 2: FTP profiles dependencies

Devices typically uses the *Service discovery protocol* (**SDP**) [2.3.3] to discover which profiles are available and what parameters are needed to connect to another client.

[8]It is also a lightweight wrapper of the human interface device protocol defined for USB

3 Driver-level Working Mechanism

This section mainly focuses on the lowest level of the Bluetooth stack. Figure [3] made a comparison between the classical ISO/OSI model and the actual Bluetooth stack. It is important to notice that the separation between the Bluetooth-controller chipset and the BlueZ modules take place at *Host Controller Interface*: in simple words, it can be considered as a bridge between the hardware and the OS that provides a uniform method of accessing the Bluetooth baseband capabilities. The discussion of this layer can be found later at [3.1].

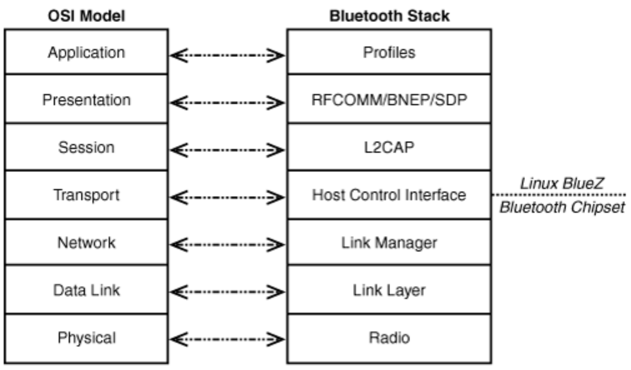


Figure 3: Bluetooth Stack compared to the ISO/OSI model

3.0.1 Radio Layer

The Bluetooth Radio layer is the lowest of the Bluetooth specification. Essentially it defines the specification of Bluetooth transceiver devices that operates in the 2.4GHz ISM band.

Bluetooth devices are actually classified into 4 classes depending on the signal range and the maximum output power:

- **Power Class 1**: designed for long-range device, max. output power is 20 dBm (range of 100m)
- **Power Class 1.5**[9]: maximum output power is 10 dBm
- **Power Class 2**: used in normal device, max. output power is 4 dBm (range of 10m)
- **Power Class 3**: designed for short-range device, max. output power is 0 dBm (range of 10cm)

Devices can optionally change its transmitted power in a link with LMP commands (see 3.0.3).

3.0.2 Link Layer

The link layer (also called "*link controller*") is mainly responsible for managing physical channels and links, but it also deals with error correction and security.
This layer handles the two link-layer protocol already presented: the *Synchronous Connection-Oriented* (**SCO**)[2.3.4] and the *Asynchronous Connection-Less* (**ACL**)[2.3.5].

[9]Available only in Bluetooth Low Energy radio controller (BLE)

3.0.3 Link Manager

The Link Manager layer is mainly responsible for link setup and configuration. Once it discovers another device remote link manager, it communicates with the device through the *Link Management Protocol* (**LMP**). This protocol is relatively simple, and essentially consists of a number of *Protocol data units* (PDU) begin exchanged from one device to another determined by the AM_ADDR field in the packet header. By looking at the header of a packet, the first bit is called *transaction ID* and specifies if the master or the slave initiated the connection., the subsequent 7 bits are used for the *operation code* that identifies the type of LMP message being sent and the rest of the packet is left for the payload.

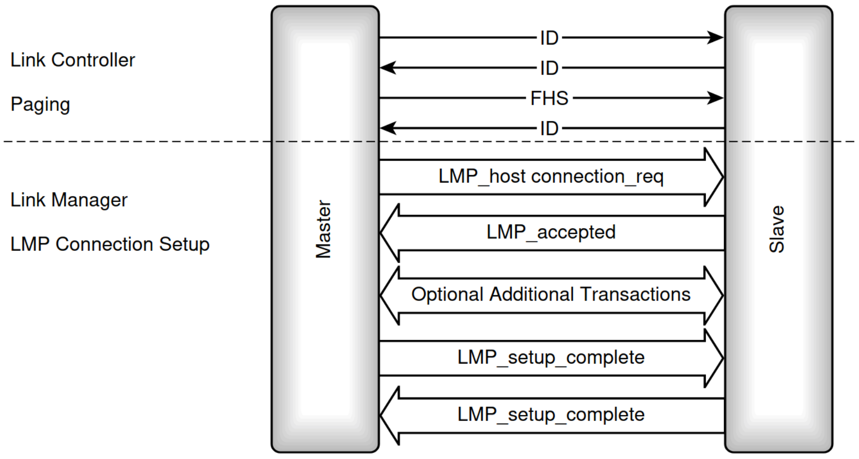


Figure 4: LMP message sequence chart for ACL link setup

In order to establish an ACL links, the Link Controller layer must establish a link between devices before LMP messages can be exchanged. Figure [4] shows all the message needed for the setup.

Apart from establishing links, this layer also deal with security: the authentication procedure is based on a challenge-response scheme, with the assumption that both devices already share a secret key. The verifier sends an LMP_au_rand PDU which contains a random number (the challange), and the other device calculates the response (that depends both on its BD_ADDR and on the secret key).

Other interesting functionalities that LMP supports are pairing, role change (master/slave), QoS and Power Control.

3.1 Host Controller Interface

The HCI provides a command interface to the baseband controller and link manager, and access to hardware status and control registers. Essentially this interface provides a uniform method of accessing the Bluetooth baseband capabilities. Later in this essay other HCI implementation will be discussued, and this is due to the fact that HCI exists across 3 sections, each of them playing different role to in the HCI system:

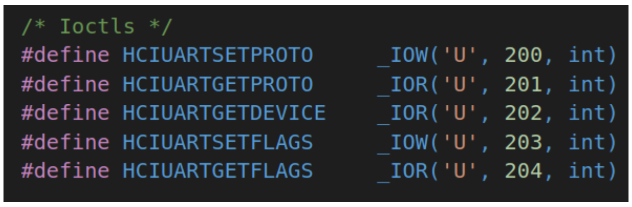
- **HCI Firmware**: located in the actual bluetooth hardware device (also called "*Host Controller*"), implements HCI commands by accessing hardware status registers, control registers, and event registers.
- **HCI Driver**: located inside the Linux kernel, it receives asynchronous notifications of HCI events and parse them. This component is extensively discussed in [4.1].

- **HCI Transport layer**: HCI Driver and Firmware communicate thought this level. In other terms, this layer should provide the ability to transfer data without without worrying about its content.

3.2 HCI Transport Layer

As already said, this layer transports *HCI commands*, ACL and SCO data between the HCI Driver and the Bluetooth chipset and vice versa.

The Bluetooth standard had defined official support three types of HCI transport layer: **UART**, **RS232** and **USB**. The kernel code that implements all these functionality can be found in `drivers/bluetooth` and, in this section, the HCI-UART layer is analyzed in detail, since it is most common on mobile and embedded devices (PCs usually employs USB).
The code for that functionality, loosely speaking, is contained in `hci_uart.h`, `hci_h4.c`, `hci_ldisc.c` and `h4_recv.h`.



```
/* Ioctls */
#define HCIUARTSETPROTO      _IOW('U', 200, int)
#define HCIUARTGETPROTO      _IOR('U', 201, int)
#define HCIUARTGETDEVICE     _IOR('U', 202, int)
#define HCIUARTSETFLAGS      _IOW('U', 203, int)
#define HCIUARTGETFLAGS      _IOR('U', 204, int)
```

Figure 5: HCI-UART ioctls

The first thing to notice in `hci_uart.h` (presented in figure [5]) is the definition of 5 *ioctls*. These function are very similar to system calls, but they are intrinsically more "*raw*". Generally ioctls are used for device-specific I/O or for other operation that cannot be expressed through a normal system call. The classical example is the CD-ROM driver: although the `open`, `read`, `write` system calls suits very well for managing data present on the disc, no existing syscall would allow user-level application to eject discs, so an ioctl was used to perform such hardware-specific operation.
Another huge difference from system calls resides in the syntax: while syscalls have well-defined parameters and names, in order to call a ioctl functionality developers must instead pass a file-descriptor of the targeted device, a *magic number*[10] that represent the requested functionality and all the other request-specific parameters[11].

Listing 2: "Prototype of the function used to invoke an iotctl request"

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, ...)
```

In this context the ioctls calls are indeed used to control hardware specific functionality, like the UART's flags. Linux provides some helper macros to create an unique ioctl identifier and add the required read-/write features. These are:

- **_IO**: an ioctl with no parameters
- **_IOW**: an ioctl with write parameters, used to write data to the driver

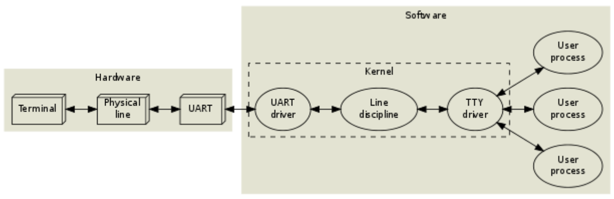
[10]Unique values with unexplained meaning that should be replaced with named constants; **here** there is a list of all Linux kernel available request

[11]The dots are typically represented as `char *argp`, but they are used in the source code to prevent type checking during compilation

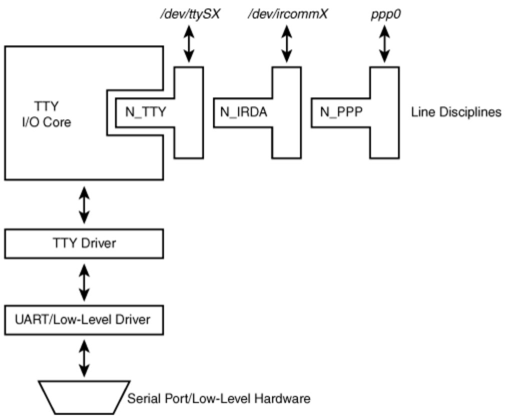
- **_IOR**: an ioctl with read parameters, used for reading data from the driver
- **_IOWR**: an ioctl with both write and read parameters.

When the HCI-UART driver is initialized in `hci_uart_init`, the structure `tty_ldisc_ops` is used to associate each protocol operation with a routine in the code: in fact by looking at the `compact_ioctl` field it is clear that all these ioctl requests are simply then managed through a `switch` statement in `hci_uart_tty_ioctl`. Finally, once the previous structure is filled with the correct function pointers, the driver is registered via `tty_register_ldisc`.

The term *tty* in the previous function name can appear unusual, since it is commonly related to old, ASCII-based teletype terminals. Despite these devices nowadays dissapeared, the TTY subsystem is central to the design of Linux, and UNIX in general.
When a device is connected through a pair of wires to a UART port on a computer, the Linux UART driver manages the physical transmission of bytes, including parity checks and flow control. After that, as shown in figure [6a], an additional layer called *"Line Disciple"*[12] must be added in order to glue together the low level device driver code with the high level generic interface routines (such as `read`, `write` and `ioctl`). In addition to that, this layer is also responsible for implementing the semantics associated with the device. The line policy is separated from the device driver so that the same serial hardware driver can be used by devices that require different data handling.
In fact, as shown in figures [6b, 6a], the terminal subsystem consists of three layers: the upper one, used to provide the character device interface, the lower hardware driver, used to communicate with the hardware, and the middle line discipline to implement behavior common to terminal devices.



(a) How Linux kernel treat UART devices



(b) The tty subsystem components

In Linux is it possible to specify the line disciple of a serial line (via its associated device file) through the `ldattach` command. When the line disciple "`HCI`" is set on a line, the function `hci_uart_tty_open` is automatically invoked. In few words, this function allocate the `hci_uart` structure of figure [7b] and set the `*tty` field to the `tty` device that correspond to the UART serial.

The `serdev_device` structure refers to a new bus controller called *"Serial Device Bus"* introduced in Linux 4.2 to overcome some limitation[13] of tty character device.

In figure [7a] it is possible to see the definition of the `hci_uart_proto` structure which contains, along with some attribute such as "*operational speed*", all the function pointer needed for the protocol. The

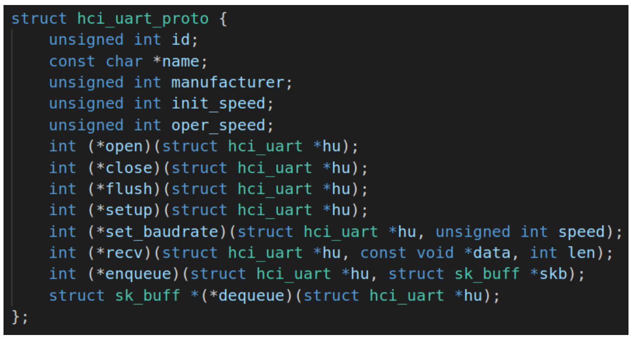
[12]More info on Line Disciple and tty is available **here**
[13]The essay won't examine in depth this structure; interested reader could find a nice overview **here**

drivers/bluetooth directory contains several files related to different bluetooth chipset (e.g. Intel, Nokia), however by defining such protocol structure it is possible to abstract hardware-specific differences and force the usage of well-defined procedures. For example, by looking at hci_h4.c code (which implement *the Bluetooth-to-Serial* H4 protocol) it is possible to see how the previous function pointers are associated to the corresponding handler (e.g. open= h4_open). The technique of specifying only function pointer inside a protocol's struct is widely used across the Linux kernel, as explained for sockets in [4.3].

An important thing to notice is that from now, upper layers will only use the hci_dev structure to interact with lower layer. In fact, when the HCI-UART driver have to pass the received frame to the HCI-Core, it employs hci_recv_frame function. This function, essentially, takes as input an HCI device and a *"socket kernel buffer"* and adds this buffer to the HCI device *"socket kernel buffer queue"* (hdev-¿rx_q)[14]. After the new frame is added to the *skb queue* the function responsible for handling frame reception is added to the device *work-queue.*[15].

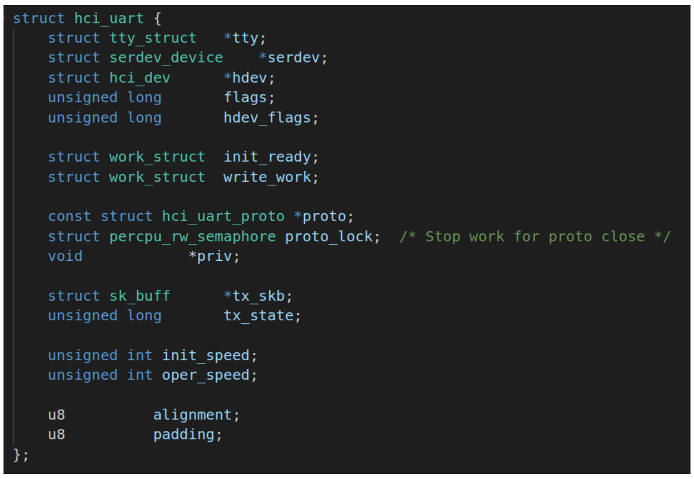
```
/* Receive frame from HCI drivers */
int hci_recv_frame(struct hci_dev *hdev, struct sk_buff *skb)
{
    ...
    Testing Packet type
    ...
        skb_queue_tail(&hdev->rx_q, skb);
        queue_work(hdev->workqueue, &hdev->rx_work);
        return 0;
}
```

Linux includes the command hciattach that, as the manpage says, is used to attach a serial UART to the Bluetooth stack as HCI transport interface. In few words this tool acts as a middle-man between UART and the HCI transport interface, the source code will not be analyzed since it is very similar to the one already discussed, but interested readers could find it inside BlueZ user-level package at tools/hciattack.h.



```
struct hci_uart_proto {
    unsigned int id;
    const char *name;
    unsigned int manufacturer;
    unsigned int init_speed;
    unsigned int oper_speed;
    int (*open)(struct hci_uart *hu);
    int (*close)(struct hci_uart *hu);
    int (*flush)(struct hci_uart *hu);
    int (*setup)(struct hci_uart *hu);
    int (*set_baudrate)(struct hci_uart *hu, unsigned int speed);
    int (*recv)(struct hci_uart *hu, const void *data, int len);
    int (*enqueue)(struct hci_uart *hu, struct sk_buff *skb);
    struct sk_buff *(*dequeue)(struct hci_uart *hu);
};
```

(a) HCI-UART protocol structure



```
struct hci_uart {
    struct tty_struct   *tty;
    struct serdev_device    *serdev;
    struct hci_dev      *hdev;
    unsigned long       flags;
    unsigned long       hdev_flags;

    struct work_struct  init_ready;
    struct work_struct  write_work;

    const struct hci_uart_proto *proto;
    struct percpu_rw_semaphore proto_lock;  /* Stop work for proto close */
    void            *priv;

    struct sk_buff      *tx_skb;
    unsigned long       tx_state;

    unsigned int init_speed;
    unsigned int oper_speed;

    u8          alignment;
    u8          padding;
};
```

(b) HCI-UART structure

[14]These structures will be further explained in the next chapter, for now they can be viewed, respectively, as a buffer and as a list of buffers used for packets

[15]This functionality will be further explained in the next chapter at [4.1], for now think about it as sort of kernel thread

4 Kernel Space Working Mechanism

4.1 Host Controller Interface

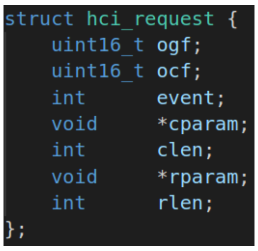
As already seen in the Figure [16b], the communication between the Bluetooth controller (the integrated circuit) and the host stack (the OS) happens through the *Host Controller Interface* (**HCI**).

A reason for this choice is due to the fact that controller deals with hard real-time requirements and contact with the physical layer, so it is good to separate this part from the host. Host is mostly responsible for more complex implementations but with less stringent in terms of timing.

The Bluetooth specification defines HCI as a set of commands and events for the host and the controller to interact with each other, along with a data packet format and a set of rules for flow control. Nowadays, most BLE chipset comes in a complete form that incorporates the complete controller, host and application in a single packet. This kind of chipset is normally known as system-on-chip (SoC). SoC reduces cost and size on the final device, hence, it is preferable to runs all three layers concurrently on a single chip for any BLE application.

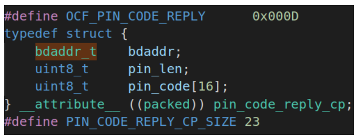
All the available HCI events, commands and functionalities are extensively defined in bluetooth documentation[16] in order to create a well-defined interface between hardware developers and the operating system. BlueZ defines all these functionalities in `bluetooth/hci_lib.h` and the related structures in `bluetooth/hci.h`. A generic HCI request [8a] is characterized by

1. *Opcode Group Filed (ogf)*: Specifies the general category of the command
2. *Opcode Command Field(ocf)*: Specifies the actual command
3. *Event parts*: composed by an event code that specifies the actual command and some (optional) event specific parameter.



```
struct hci_request {
    uint16_t ogf;
    uint16_t ocf;
    int      event;
    void     *cparam;
    int      clen;
    void     *rparam;
    int      rlen;
};
```

(a) Genering HCI Request structure



```
#define OCF_PIN_CODE_REPLY      0x000D
typedef struct {
    bdaddr_t    bdaddr;
    uint8_t     pin_len;
    uint8_t     pin_code[16];
} __attribute__ ((packed)) pin_code_reply_cp;
#define PIN_CODE_REPLY_CP_SIZE 23
```

(b) An HCI Command structure that contains its parameter

Developers that doesn't want to dirty their hand into manually packing a structure could use these two functions:

```
int hci_send_cmd(int dd, uint16_t ogf, uint16_t ocf, uint8_t plen, void *param);
int hci_send_req(int dd, struct hci_request *req, int timeout);
```

The first one just send a single command to the Bluetooth controller, while the other one also waits for a response (the timeout is specified in the parameters). All the other functions contained in `hci_lib.h` can be considered as wrappers of these two functions where, for example, some parameter are hard-coded.

We take a portion of code from `hci_code.h` that it is interesting to discuss in order to present a practical usage of most common Linux macros and data-structures.

By skipping the first three line (for the moment), we came across the definition of two *linked-list.* The first list keeps track of all the available HCI devices and thus it is then locked with a read-write spin-lock (`hci_dev_list_lock`) since the code will access that list in read mode more often than in

[16]Official Document; Vol 4 Part E

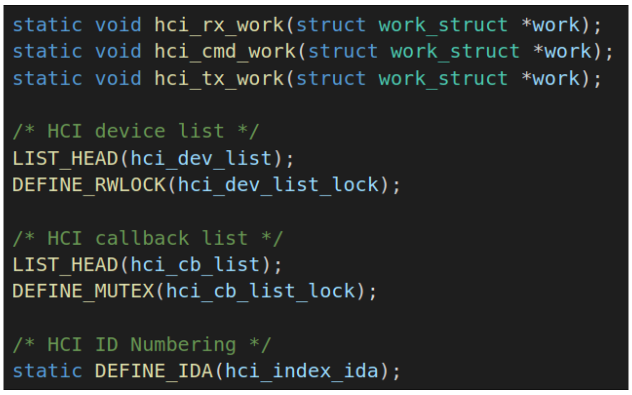

```
static void hci_rx_work(struct work_struct *work);
static void hci_cmd_work(struct work_struct *work);
static void hci_tx_work(struct work_struct *work);

/* HCI device list */
LIST_HEAD(hci_dev_list);
DEFINE_RWLOCK(hci_dev_list_lock);

/* HCI callback list */
LIST_HEAD(hci_cb_list);
DEFINE_MUTEX(hci_cb_list_lock);

/* HCI ID Numbering */
static DEFINE_IDA(hci_index_ida);
```

Figure 9: `hci_code.h`

writing mode[17]. The second list, instead, employs a classical mutex since this list is used by two wrappers (`hci_register_cb` and `hci_unregister_cb`) that basically are provided as an interface to upper-layer protocols to store their callback functions, so a classical mutex is a good choice for such types of interface when the behaviour is not completely predetermined.

The last line, on the other hand, is interesting because it shows the clever usage of Linux optimizations. In fact, the `DEFINE_IDA`[18] macro a is part of the IDR library that implements the *IDR Allocator*, a particular structure that map IDs to pointers in an efficient way. The IDA variant, instead, allows to just allocate any kind IDs[19] in a safe way (automatically checks if that ID is already in use) and by using the least possible bit (each ID occupies one bit) without any sort of mapping.

By coming back to the first three lines, we see that BlueZ kernel implementation of HCI makes use of *workqueues* to manage transmission/reception and command execution of the HCI layer. Basically workqueues permit "*work*" to be deferred outside the interrupt context into the kernel process context and, differently from its predecessor (*Tasklets*), the handler function responsible for the work can sleep. This is due to the fact that *workqueues* executes as a normal process and not a interrupt, so it can hold semaphores. Obviously this causes an higher latency compared to tasklets (which always runs in the interrupt context), however their usage is preferable when speed is not a critical factor since they are more recent[20] and thus come with a richer set of API .

In its most basic form, the work queue subsystem is an interface for creating kernel threads to handle work that is queued from elsewhere. All of these kernel threads are called `worker threads`. As shown in figure (10a), the work queue are maintained by the `work_struct` [??] structure that is composed by three field: `func` is the function that will be scheduled by the `workqueue`, `data` represent all the parameters passed to the handler and `entry` is just used to insert such structures into the higher-level `workqueue_struct` as shows in figure [10a].

Listing 3: `work_struct`

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
};
```

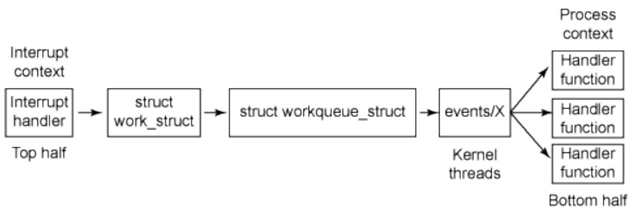
[17]Multiple reader can acquire the lock on the data structure simultaneously provided that no writer has taken the lock
[18]An article that explains the problem
[19]represented as an integer between 0 and INT_MAX
[20]Introduced in 2.5, while Tasklet were added in 2.3

By looking at figure [9], we see how the HCI module associate to each HCI device (hci_dev) several work_t structures that are statically initialized via INIT_WORK. This macro basically associate to a work its corresponding handler.

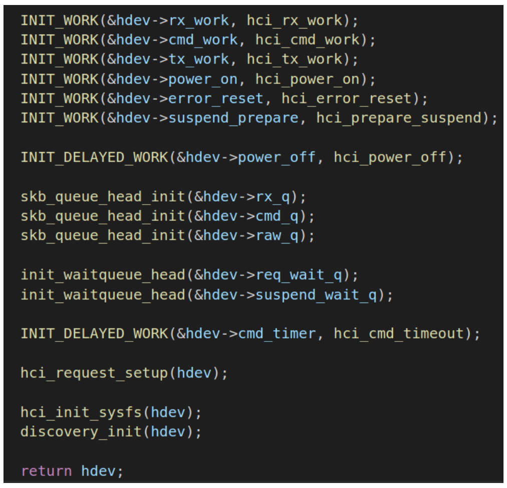
The Linux kernel provides special *per-cpu threads* that are called **kworker** that is used to schedule the deferred functions of the workqueue. The following command can be used to display BlueZ's kworkers.

```
ubuntu@server:# systemd-cgls -k | grep kworker | grep hci
173 [kworker/u9:0-hci0]
1319 [kworker/u9:2-hci0]
```

The choice of workqueues for managing trasmission/reception of packets is probably done due to their "*pausable*" functionality.



(a) How Workqueue works



```
INIT_WORK(&hdev->rx_work, hci_rx_work);
INIT_WORK(&hdev->cmd_work, hci_cmd_work);
INIT_WORK(&hdev->tx_work, hci_tx_work);
INIT_WORK(&hdev->power_on, hci_power_on);
INIT_WORK(&hdev->error_reset, hci_error_reset);
INIT_WORK(&hdev->suspend_prepare, hci_prepare_suspend);

INIT_DELAYED_WORK(&hdev->power_off, hci_power_off);

skb_queue_head_init(&hdev->rx_q);
skb_queue_head_init(&hdev->cmd_q);
skb_queue_head_init(&hdev->raw_q);

init_waitqueue_head(&hdev->req_wait_q);
init_waitqueue_head(&hdev->suspend_wait_q);

INIT_DELAYED_WORK(&hdev->cmd_timer, hci_cmd_timeout);

hci_request_setup(hdev);

hci_init_sysfs(hdev);
discovery_init(hdev);

return hdev;
```

(b) Extract from hci_alloc_dev

Later in the code, we see the initialization of one of the fundamental Linux kernel data structures: ***socket kernel buffers*** (**SKB**)[21]. When the hardware receives a packet, the kernel usually place it inside such type of structures. Later, at higher levels, applications can drain the SKB-queue through their corresponding socket via a system calls to either recv or recv_from.

Most SKBs are stored on a list, whose head is implemented by struct sk_buff_head, which is relatively simple:

```
struct sk_buff_head {
        struct sk_buff     *next;
        struct sk_buff     *prev;
        __u32              qlen;
        spinlock_t         lock;
};
```

The first two member implements a doubly linked list of SKB, the third one keeps track of how many packet are in the list and the latter one is a simple spin-lock that regulates concurrent access. As performed on [10b], this type of object must be initialized via skb_queue_head_init. Inside the HCI layer of Bluetooth, three SKB queues are initialized, two for incoming RX/TX and one for managing commands.

[21]The sk_buff structure is too complex to describe it here, anyway it is extensively commented directly on the source code

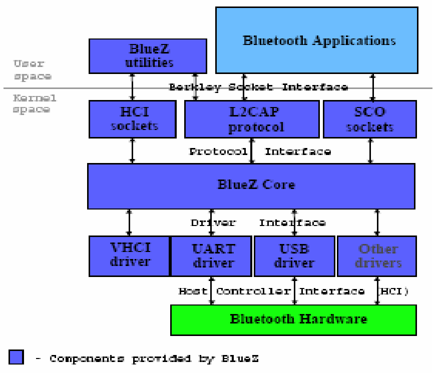
By focusing on cmd_q list, the work-handler hci_cmd_work [4.1] is analyzed to see how these buffers are managed at kernel-level.

First of all, the work handler retrieves the device's hci_dev structure by using the container_of() macro. Basically, this macro is used to cast a member of a structure out to the containing structure. This technique becomes extremely useful inside the Linux kernel because some functions (such as callbacks or work functions) have a fixed set of arguments. So by considering the code below as example, the hci_dev structure has the cmd_work member, which is used by the command handler to retrieve the higher-level structure: the first argument of the macro is a pointer to the contained struct, the second one is the type of the container struct, and the last one represent the name of the member inside that higher struct.

Then, the handler checks (through atomic_read) if there is incoming queued commands to send. In affirmative case, the socket buffer present on the head is removed (skb_dequeue) and cloned in the list of sent commands. Then, after the pending command counter is decremented (always in atomic way) the command is finally sent via the function hci_send_frame.

```
static void hci_cmd_work(struct work_struct *work)
{
        struct hci_dev *hdev = container_of(work, struct hci_dev, cmd_work);
        struct sk_buff *skb;
    ...
        /* Send queued commands */
        if (atomic_read(&hdev->cmd_cnt)) {
                skb = skb_dequeue(&hdev->cmd_q);
        ...
                kfree_skb(hdev->sent_cmd);

                hdev->sent_cmd = skb_clone(skb, GFP_KERNEL);
                if (hdev->sent_cmd) {
                        if (hci_req_status_pend(hdev))
                                hci_dev_set_flag(hdev, HCI_CMD_PENDING);
                        atomic_dec(&hdev->cmd_cnt);
                        hci_send_frame(hdev, skb);
                        if (test_bit(HCI_RESET, &hdev->flags))
                                cancel_delayed_work(&hdev->cmd_timer);
                        else
                                schedule_delayed_work(&hdev->cmd_timer,
                                                      HCI_CMD_TIMEOUT);
                } else {
                        skb_queue_head(&hdev->cmd_q, skb);
                        queue_work(hdev->workqueue, &hdev->cmd_work);
                }
        }
}
```

To conclude the discussion, we refer to figure [??]. Here we can see that the Host Controller Interface lies between the Bluetooth controller and the BlueZ core. This consequently means that other protocols like L2CAP and SCO sits over HCI. The kernel provides supports to such upper layer protocols via a socket interface, however it also allows to reach a deeper lever of the stack (HCI) by providing a similar socket interface called "*HCI socket*" that enables apps to directly send arbitrary HCI packets transparently to the hardware and call the *Bluetooth Management*



API[5.1]. The code for this functionality is written on `hci_socks.c`.
We left the discussion to Bluetooth socket to the next section.

4.1.1 Sniffing HCI Packets

BlueZ includes the tool `hcidump` that is able to sniff bluetooth packets that pass through an HCI device. First of all, the HCI device must be identified via `hciconfig`, which returns a lot of interesting information about the device like ACL and SCO MTUs along with other flags (UP, RUNNING, etc...).

```
ubuntu@server: hciconfig
hci0:    Type: Primary  Bus: UART
         BD Address: XX:XX:XX:XX:XX:XX  ACL MTU: 1021:8
SCO MTU: 64:1
         UP RUNNING PSCAN
         RX bytes:12951 acl:34 sco:0 events:269 errors:0
         TX bytes:4095 acl:34 sco:0 commands:166 errors:0
```

The subsequent output shows all the sniffed HCI packet generated by an user-level application that had called the `hci_inquiry` API in order to perform a device scan. Initially, the HCI inquiry command is sent by the kernel to the Bluetooth controller that will consequently generates three events in order to notify that the inquiry scan is started, completed and the effective scan results. [22].

```
ubuntu@server: sudo hcidump -i hci0
HCI sniffer - Bluetooth packet analyzer ver 5.53
device: hci0 snap_len: 1500 filter: 0xffffffffffffffff
< HCI Command: Inquiry (0x01|0x0001) plen 5
    lap 0x9e8b33 len 8 num 255
> HCI Event: Command Status (0x0f) plen 4
    Inquiry (0x01|0x0001) status 0x00 ncmd 1
> HCI Event: Extended Inquiry Result (0x2f) plen 255
    bdaddr XX:XX:XX:XX:XX:XX mode 1 clkoffset 0x0fa9 class 0x5a020c r
    Complete local name: 'BLUETOOTH-DEVICE-NAME'
    Complete service classes: 0x1105 ...
    Unknown type 0x05 with 0 bytes data
    Unknown type 0x07 with 128 bytes data
> HCI Event: Inquiry Complete (0x01) plen 1
    status 0x00
```

[22]Real BD address and device name are substituted respectively with 'XX:XX:XX:XX:XX:XX' and 'BLUETOOTH-DEVICE-NAME'

4.2 Bluetooth and the Linux Device Driver Model

The last thing worth to mention in figure [10b] is the initialization of a *sysfs* interface by the HCI module via hci_init_sysfs. Sysfs is a pseudo-file system provided by the Linux kernel that is usually used to export info and configure parameters of various kernel subsystem, hardware device and device drivers to user-space level.[23] The /sys directory is organized in a hierarchical way according to the "*Linux device driver model*".
The kobject is the core elenement of the device driver model: the field *name indicates the object's name that will be shown in the sysfs's directories, whereas *parent refers the object's parent in the hierarchical structure. The kobj_type field is used to connect kernel object with the user-space file operations on the sysfs file, in particular through default_attrs (which defines file's attributes), and sysfs_ops (which defines the two available operation on the attributes: store and show). An interesting thing inside the kobject structure is how the last variable is defined: the term *":1"* after the variable's name indicated the compiler that the variable is one-bit size. This types of variable are called *"bit-fields"*[24] and are extremely useful in embedded device due to memory constraint. In general, this technique is used for storing Boolean flags in the most efficient way, exactly like in this structure.

Figure [11] shows how the kobject structure is the basic structure of the Linux Device Model, in fact structures in the higher levels of the model are bus_type, device and device_driver.

```
struct kobject {
 const char  *name;
 struct list_head entry;
 struct kobject  *parent;
 struct kset  *kset;
 struct kobj_type *ktype;
 struct sysfs_dirent *sd;
 struct kref  kref;
 ...
 unsigned int  state_initialized:1;
 unsigned int  state_in_sysfs:1;
 unsigned int  state_add_uevent_sent:1;
 unsigned int  state_remove_uevent_sent:1;
 unsigned int  uevent_suppress:1;
};
```

Buses are used to implement a communication channel between the processor and IO devices. Each device is connected to the CPU via such buses and it appears at /sys/bus directory. In Linux, a bus is represented through a bus_type structure which include as its member a struct device: in fact any device in the system has a struct device structure associated with it. Devices are discovered by different kernel methods (hotplug, device drivers, system initialization) and are registered in the system: in fact, each device present in the kernel has an entry in /sys/devices. Inside hci_sysfs.c, the Bluetooth device is firstly initialized through device_initialize() and then registered via device_add()[25], which internally adds the device to the kobject hierarchy via kobject_add(), adds it to the global for the device and then adds it to the other relevant subsystems of the driver model..

As shown in figure [11], drivers for the device (represented by a device_driver structure) must be linked to a device via a bus. This operation is performed by the bus that will try to *bind* the device

[23]The sysfs virtual filesystem was introduced in 2.6, in previous version the same functionality was provided by procfs
[24]GCC official documentation regarding "bitfields"
[25]Respectively inside hci_init_sysfs and hci_conn_add_sysfs

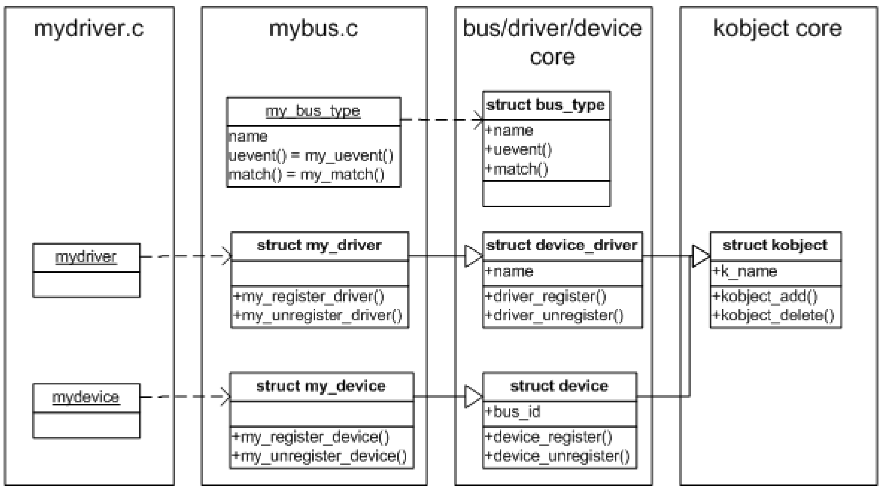
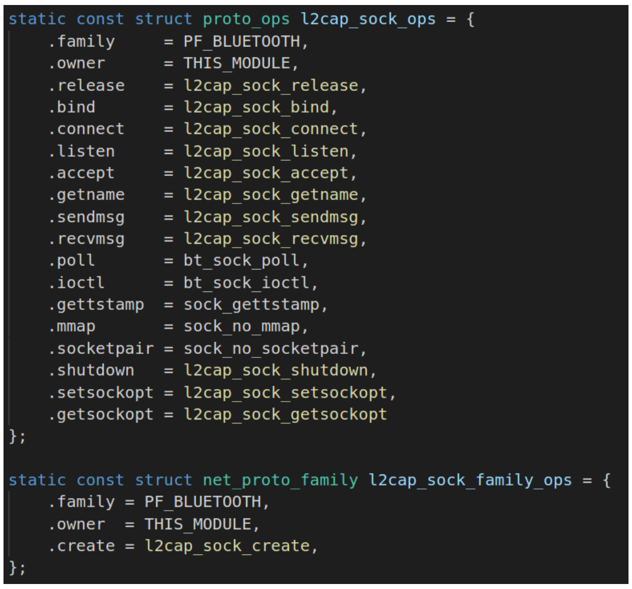


Figure 11: Basic components of Linux Device Model

connected to in with every drivers that have been registered.
All low-level structure that manages drivers employs this model, in fact also the `hci_dev` structure used at driver level has a `struct device` field.

To conclude the discussion, the `bt_sysfs_init` creates a sysfs *class* (via `class_create`) called "*bluetooth*". Classes are a kernel abstraction that contains all the devices with similar functionalities and, in fact, Bluetooth information is exposed at `/sys/class/bluetooth`.

```
ubuntu@server:/sys/class/bluetooth/hci0#: ll
lrwxrwxrwx 1 root root    device -> ../../ttyAMA0/
drwxr-xr-x 2 root root    power/
drwxr-xr-x 3 root root    rfkill1/
lrwxrwxrwx 1 root root    subsystem -> ../../class/bluetooth/
-rw-r--r-- 1 root root    uevent
ubuntu@server:/sys/class/bluetooth/hci0/device# ll
-r--r----- 1 root root    close_delay
-r--r----- 1 root root    closing_wait
-r--r----- 1 root root    custom_divisor
-r--r--r-- 1 root root    dev
lrwxrwxrwx 1 root root    device -> ../../../3f201000.serial/
-r--r----- 1 root root    flags
drwxr-xr-x 4 root root    hci0/
-r--r----- 1 root root    io_type
-r--r----- 1 root root    iomem_base
-r--r----- 1 root root    iomem_reg_shift
-r--r----- 1 root root    irq
-r--r----- 1 root root    line
-r--r----- 1 root root    port
drwxr-xr-x 2 root root    power/
lrwxrwxrwx 1 root root    subsystem -> ../../../../../../class/tty/
-r--r----- 1 root root    type
-r--r----- 1 root root    uartclk
-rw-r--r-- 1 root root    uevent
-r--r----- 1 root root    xmit_fifo_size
```



```
static const struct proto_ops l2cap_sock_ops = {
    .family     = PF_BLUETOOTH,
    .owner      = THIS_MODULE,
    .release    = l2cap_sock_release,
    .bind       = l2cap_sock_bind,
    .connect    = l2cap_sock_connect,
    .listen     = l2cap_sock_listen,
    .accept     = l2cap_sock_accept,
    .getname    = l2cap_sock_getname,
    .sendmsg    = l2cap_sock_sendmsg,
    .recvmsg    = l2cap_sock_recvmsg,
    .poll       = bt_sock_poll,
    .ioctl      = bt_sock_ioctl,
    .gettstamp  = sock_gettstamp,
    .mmap       = sock_no_mmap,
    .socketpair = sock_no_socketpair,
    .shutdown   = l2cap_sock_shutdown,
    .setsockopt = l2cap_sock_setsockopt,
    .getsockopt = l2cap_sock_getsockopt
};

static const struct net_proto_family l2cap_sock_family_ops = {
    .family = PF_BLUETOOTH,
    .owner  = THIS_MODULE,
    .create = l2cap_sock_create,
};
```

Figure 12: Extract from `l2cap_socks.c`

4.3 Bluetooth Sockets

As already said in the previous section, the BlueZ framework employs *sockets* to pass data between user-level and the kernel. Everything works similarly to TCP/UDP socket programming, with the only exception of the flags passed to the `socket()` system call.

```
s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
```

Here the code refers to the opening of a RFCOMM socket, the first parameter (AF_BLUETOOTH) specifies the family of protocol that will be used for communication, the second one the specifies the *communication semantics* (a stream based transport protocol as already said in 2.3.1) and the last one sets the effective protocol to use.
In the next section we are going to analyze two of the main protocol related to audio transmission.

4.3.1 L2CAP

In order to implement a customized socket in the kernel, developers have to fill two structure: `proto_ops`, to register a new protocol, and `net_proto_family`, that defines the new protocol family. Then, to finallt register the new socket a call to `sock_register` must be issued.
From figure [12] it is possible to see how the interface is defined, mapping each socket API to the corresponding L2CAP handler.

Below we have a simple application that employs L2CAP sockets in order to create a Bluetooth server. By default, L2CAP connections provide reliable datagram-oriented connections with packets delivered in order, so the socket type is SOCK_SEQPACKET, and the protocol is BTPROTO_L2CAP.
An important thing to note, is that the addressing structure changes between different protocols: the RFCOMM `sockaddr_rc` structure has an `uint8_t` filed that specifies the connection's channel, whereas L2CAP `sockaddr_l2` structure uses an `unsigned short l2_psm` to represent the *Protocol Service Multiplexer* at a different position with respect to the previous one, so a direct conversion is not possible.
In addition to that, since Bluetooth byte-ordering is little-endian, a set of function equivalent to the TCP/IP `htons` is provided by BlueZ:

```
unsigned short int htobs( unsigned short int num );
unsigned short int btohs( unsigned short int num );
unsigned int htobl( unsigned int num );
unsigned int btohl( unsigned int num );
```

The term BDADDR_ANY is defined as a Bluetooth address filled with zeroes, and means that the socket must be opened on the first available Bluetooth controller.

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>

int main(int argc, char **argv)
{
    struct sockaddr_l2 loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int s, client, bytes_read;
    socklen_t opt = sizeof(rem_addr);

    // allocate socket
    s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);

    // bind socket to port 0x1001 of the first available Bluetooth adapter
    loc_addr.l2_family = AF_BLUETOOTH;
    loc_addr.l2_bdaddr = *BDADDR_ANY;
    loc_addr.l2_psm = htobs(0x1001);

    bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));

    // put socket into listening mode
    listen(s, 1);

    // accept one connection
    client = accept(s, (struct sockaddr *)&rem_addr, &opt);

    ba2str( &rem_addr.l2_bdaddr, buf );
    fprintf(stderr, "accepted connection from %s\n", buf);

    memset(buf, 0, sizeof(buf));

    // read data from the client
    bytes_read = read(client, buf, sizeof(buf));
    if( bytes_read > 0 ) {
        printf("received [%s]\n", buf);
    }

    close(client);
    close(s);
}
```

L2CAP connection options can be configured/viewed through the classical `getsockopt` and `setsockopt`, which accepts an `l2cap_options` as input. At kernel level the procedure for changing such options is managed by `l2cap_sock_setsockopt`, that essentially locks the socket (via `lock_sock`), copy the new configuration from user-space with either `copy_to_user` or `get_user`[26] and perform the needed configuration before unlocking the socket.

```
struct l2cap_options {
    uint16_t     omtu;
    uint16_t     imtu;
    uint16_t     flush_to;
    uint8_t      mode;
};
```

Advanced developers could also customize the semantics of L2CAP connection, switching from a *reliable* to a *best-effort* protocol. This operation is not supported through sockets, so the only way to instruct the Bluetooth controller is via HCI commands that adjust the *flush timeout*. This parameter specifies the maximum time to wait for a packet acknowledgment before moving to the next packet. The only drawbacks of this configuration is that the flush timeout is only present at the low-level ACL connection, so changing its value will affect all upper layer protocol[27].

An useful user-level tool for testing L2CAP connection is `l2ping`. It basically sends echo packets to another Bluetooth device and waits for a response, giving timing information and regarding how long it takes to send/receive packets of a certain size.

```
# l2ping -c 5 XX:XX:XX:XX:XX:XX
Ping: XX:XX:XX:XX:XX:XX from YY:YY:YY:YY:YY:YY (data size 44) ...
44 bytes from XX:XX:XX:XX:XX:XX id 0 time 1022.71ms
44 bytes from XX:XX:XX:XX:XX:XX id 1 time 49.71ms
44 bytes from XX:XX:XX:XX:XX:XX id 2 time 32.24ms
44 bytes from XX:XX:XX:XX:XX:XX id 3 time 57.27ms
44 bytes from XX:XX:XX:XX:XX:XX id 4 time 43.48ms
```

4.3.2 SCO

Similarly to L2CAP and RFCOMM, the SCO protocol can be used through sockets. The socket type and the protocol is respectively SOCK_SEQPACKET and BTPROTO_SCO, and this time the `sockaddr_sco` addressing structure has not field related to port or channel. The semantics of SCO sockets are nearly the same of all other protocol, but with two major exception regarding *audio format* and MTU. In fact, before establishing a SCO connection, the application should check and set the audio format used for future SCO connections via the following API:

```
int hci_read_voice_settings(int dd, uint16_t *vs, int to);
int hci_write_voice_settings(int dd, uint16_t *vs, int to);
```

To conclude the discussion, the main function employed to send an SCO frame is analyzed. Just as L2CAP sockets, SCO sockets must be registered by populating the `struct proto_ops sco_sock_ops` and `struct net_proto_family sco_sock_family_ops` . The next code refers to the routine called when the `connect` system call is called on a SCO socket.
Initially, the HCI device (identified with an `hci_dev` structure) corresponding to the Bluetooth adapter

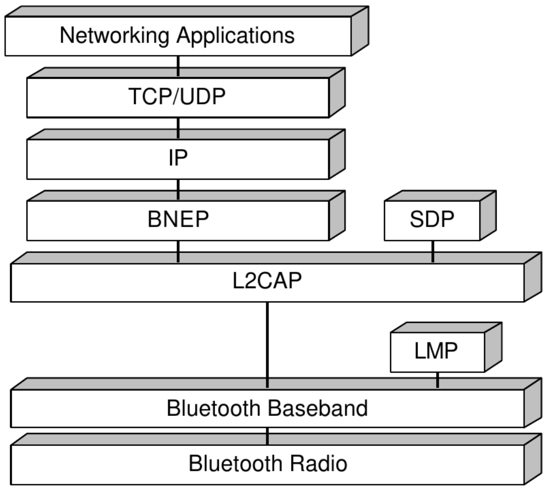
[26]The first one is used to copy a block of data while the latter simply fetch a variable from user space
[27]Also considering that between two device only a single ACL connection is allowed

is retrieved via hci_get_route and then locked with hci_dev_lock. Then, some of the macro defined at hci_core.h is used to manage LMP commands [3.0.3]. In particular the routine checks via lmp_esco_capable if the Bluetooth controller supports the *Enhanced SCO* (**eSCO**), an improved protocol that even supports packet retransmissions to achieve reliability, and set the link type accordingly. At this point, an HCI connection is created through hci_connect_sco (which basically perform all low-level commands and procedures to setup the link) and then converted to an sco_conn structure via sco_conn_add. Finally, the SCO connection is applied to the socket through sco_chan_add and, eventually, the socket's timer is cleared when the device is fully connected.
At this point becomes clear how Bluetooth sockets are just a wrapper interface that implements under the hood all the necessary HCI and LMP commands necessary to setup a connection.

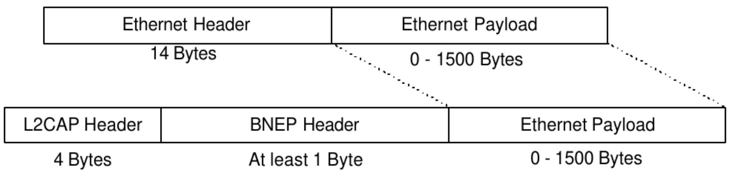
```
static int sco_connect(struct sock *sk)
{
        struct sco_conn *conn;
        struct hci_conn *hcon;
        struct hci_dev  *hdev;
        int err, type;

        hdev = hci_get_route(&sco_pi(sk)->dst, &sco_pi(sk)->src, BDADDR_BREDR);
    ...

        hci_dev_lock(hdev);

        if (lmp_esco_capable(hdev) && !disable_esco)
                type = ESCO_LINK;
        else
                type = SCO_LINK;
    ....
        hcon = hci_connect_sco(hdev, type, &sco_pi(sk)->dst,
                               sco_pi(sk)->setting);
    ...

        conn = sco_conn_add(hcon);
    ...

        /* Update source addr of the socket */
        bacpy(&sco_pi(sk)->src, &hcon->src);

        sco_chan_add(conn, sk, NULL);
    ...

        if (hcon->state == BT_CONNECTED) {
                sco_sock_clear_timer(sk);
                sk->sk_state = BT_CONNECTED;
        } else {
                sk->sk_state = BT_CONNECT;
                sco_sock_set_timer(sk, sk->sk_sndtimeo);
        }

done:
        hci_dev_unlock(hdev);
        hci_dev_put(hdev);
        return err;
}
```

4.4 BNEP

The *Bluetooth Network Encapsulation Protocol* (**BNEP**) encapsulates packets from various networking protocols, which are transported directly over the Bluetooth Logical Link Control and Adaptation Layer Protocol (L2CAP) protocol. In other words, L2CAP provides a data link layer for Bluetooth. The Bluetooth Personal Area networking profile describes how BNEP shall be used to provide networking capabilities for Bluetooth devices.



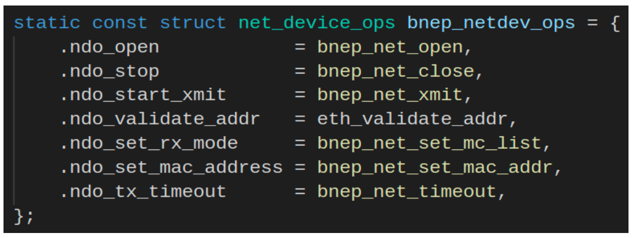
(a) Overview of the BNEP stack



(b) How Ethernet packet is encapsulated in BNEP

By using BNEP, Bluetooth devices can supports network capabilities and thus employ most of the TCP/IP stack as shown in figure [13a]. Ethernet packets are transparently encapsulated into an L2CAP packet where, as explained in figure [13b], the Ethernet header is converted into a BNEP header.
Kernel code for this functionality can be found at `net/bluetooth/bnep`, and for now the content of `netdev.c` is briefly studied. In order to transparently implement Ethernet over Bluetooth, the BNEP code allocated a new network device called bnepX via `bnep_add_connection` which internally issue a call to `alloc_netdev` and `register_netdev`.
These two function are used within the Linux kernel to create a new network interface: each interface is defined by a `struct net_device` item that, similarly to what happens with sockets, has a `struct net_device_ops` that defines the available procedure to export, as shown in figure [??].

The BNEP effectively creates an Ethernet interface via `ether_setup` inside the function `bnep_net_setup`.
Whenever an Ethernet packet is received, the BNEP layer essentially puts it into an SKB list that will be later drained by the L2CAP layer (in `bnep_net_xmit`).



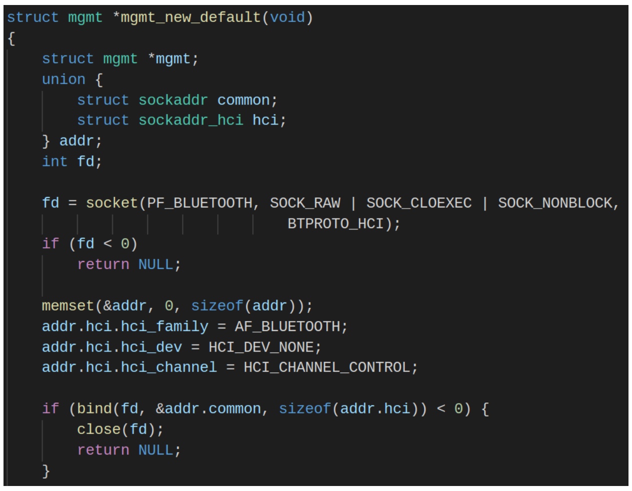
```
static const struct net_device_ops bnep_netdev_ops = {
    .ndo_open            = bnep_net_open,
    .ndo_stop            = bnep_net_close,
    .ndo_start_xmit      = bnep_net_xmit,
    .ndo_validate_addr   = eth_validate_addr,
    .ndo_set_rx_mode     = bnep_net_set_mc_list,
    .ndo_set_mac_address = bnep_net_set_mac_addr,
    .ndo_tx_timeout      = bnep_net_timeout,
};
```

Figure 14: `net_device` available operations

5 User Level Working Mechanism

5.1 Bluetooth Management API

Apart from passing through the Bluetooth daemon[28], user-level application can communicate with the kernel via the so-called *"Bluetooth Management socket"*. These sockets can be created by setting the `hci_channel` member of `struct sockaddr_hci` to HCI_CHANNEL_CONTROL when creating a raw HCI socket. These API are even used by the Bluetooth daemon at "`mgmt.c`" and figure [15] shows the creation of the HCI socket used to communicate with the kernel, as explained in section 4.1. The API documentation can be found at `mgmt-api.txt`, however the BlueZ package provides a command-line utilty called `btmgmt` that easily associate commands with the corresponding API.
By running `hcidump` while executing `btmgmt` commands becomes clear how a single `mgmt` request relates to many HCI commands and events at kernel level.



```
struct mgmt *mgmt_new_default(void)
{
    struct mgmt *mgmt;
    union {
        struct sockaddr common;
        struct sockaddr_hci hci;
    } addr;
    int fd;

    fd = socket(PF_BLUETOOTH, SOCK_RAW | SOCK_CLOEXEC | SOCK_NONBLOCK,
                                BTPROTO_HCI);
    if (fd < 0)
        return NULL;

    memset(&addr, 0, sizeof(addr));
    addr.hci.hci_family = AF_BLUETOOTH;
    addr.hci.hci_dev = HCI_DEV_NONE;
    addr.hci.hci_channel = HCI_CHANNEL_CONTROL;

    if (bind(fd, &addr.common, sizeof(addr.hci)) < 0) {
        close(fd);
        return NULL;
    }
```

Figure 15: Extract from `mgmt.c`

5.2 The Bluetooth daemon

The bluetoothd daemon manages all the Bluetooth devices. `bluetoothd` itself does not accept many command-line options, as most of its configuration is done via files present in `/etc/bluetooth`.

The `main.conf` file is read by `bluetoothd` on start-up and contains general configurations. The daemon can also provide a number of services via the D-Bus message bus system, and the security policies for that subsystem can be specified inside the file "`bluetooth.conf`".

5.2.1 D-Bus API

The Bluetooth daemon exposes a lot of API to user-space application through its D-Bus interface. The complete list of available functionalities can be found on the BlueZ code inside the `doc` directory.

D-Bus is a system for *interprocess communication* (IPC) which consists, architecturally, of two fundamental part: the `libdbus` library that allows two application to connect to each other and exchange messages, and the *D-Bus Daemon*, which acts as a router where all application connect to, that is responsible of routing messages. Normally two instances of this daemon are present, the heavily secured machine-global bus (`system`) and the one generated for each user login session (`session`).

[28]These sockets are completely independent from the daemon, since they communicate directly with the kernel

Essentially, each D-Bus application has its own service name ("`org.bluez`" in this case) that uses to expose its *objects*. Similarly, objects are uniquely identified by an *object path* (e.g. "`/hci0/dev_XX_XX`) and each of them implement one or several *interfaces*. This last element is identified by an unique names that uses dots (e.g. "`org.bluez.Device1`") and contains *properties*, *methods* and *signals*.
"Properties" are direct accessible typed fields that can be read/written; differently "methods" consists in a remote procedure call from one process to another (in this case from the client bluetooth application to the bluetooth daemon) that can return valeus/objects. "Signals", instead, are sort of message/notifications that are sent to every client that is listening for them.

Linux provides several utilities to manage the D-Bus subsystem, two of which are `dbus-monitor`, that sniffs all exchanged messages on a bus, and `dbus-send`, that allows to invoke methods or get/set object properties. The command below instruct the bluetooth daemon to start a discovery on the `hci0` adapter. Adapter's documentation gives all the necessary parameter to communicate with the daemon, like the interface `org.bluez.Adapter1` and the object path. The flag "`system`" indicates that the BlueZ daemon works at system bus rather than on a session bus.

```
ubuntu@server: dbus-send --system --print-reply --type=method_call
--dest=org.bluez /org/bluez/hci0 org.bluez.Adapter1.StartDiscovery
```

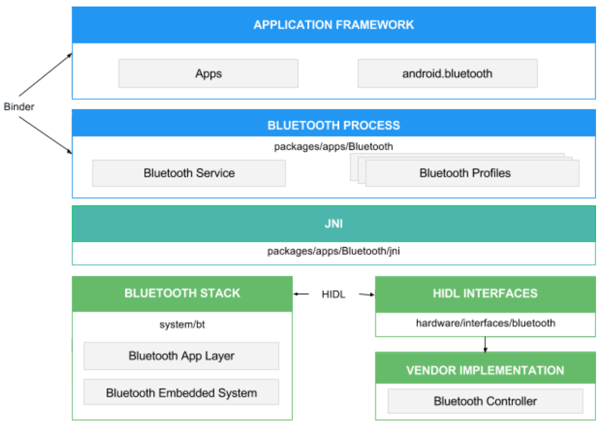
By analyzing the `bluetoothd` code, it becomes clear that the daemon basically open the D-Bus bus and, once a message is received, it parse it by following the API specification (interface, object etc..) and execute the corresponding functionality through the use of Bluetooth sockets (as shown in figure [16b]. The same comparison can be made with the popular `bluetoothctl` tool[29]: essentially is parses commands and instruct the Bluetooth daemon via the D-Bus corresponding API(s).

Android Bluetooth Stack comparison The Android operating system (that is based on Linux) also used BlueZ till 2012, when it switched to the Brodacom developed *BlueDroid*[30]. Despite the official reason for this change is still unknown, probably it was due to the fact that BlueZ uses the *D-Bus* IPC system in order to implement its API[31], while the Android OS does support it. In fact we can see from picture [16a] that the Android stack makes use of its own IPC system called "*Binder IPC*"[32] in order to communicate with the bluetooth stack.

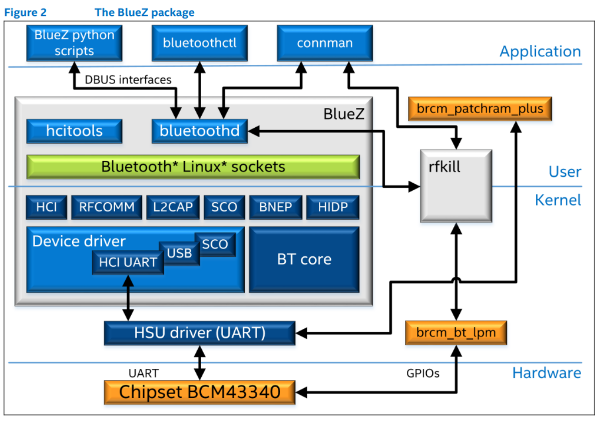
5.3 Rfkill

Many computer systems contain radio transmitters, including Wi-Fi, Bluetooth, and 3G devices. These devices consume power, which is wasted when the device is not in use. RFKill is a subsystem in the Linux kernel that provides an interface through which radio transmitters in a computer system can be queried, activated, and deactivated. When transmitters are deactivated, they can be placed in a state where software can reactive them (a *soft block*) or where software cannot reactive them (a *hard block*). The RFKill core provides the application programming interface (API) for the subsystem. Kernel drivers that have been designed to support RFkill use this API to register with the kernel, and include methods for enabling and disabling the device. Additionally, the RFKill core provides notifications that user applications can interpret and ways for user applications to query transmitter states. The RFKill interface is located at `/dev/rfkill`, which contains the current state of all radio transmitters on the system. Each device has

[29]Located at `bluez/client`
[30]Now called *Fluoride*
[31]D-Bus Bluetooth API: https://git.kernel.org/pub/scm/bluetooth/bluez.git/tree/doc
[32]Documentation



(a) Fluoride Android architecture

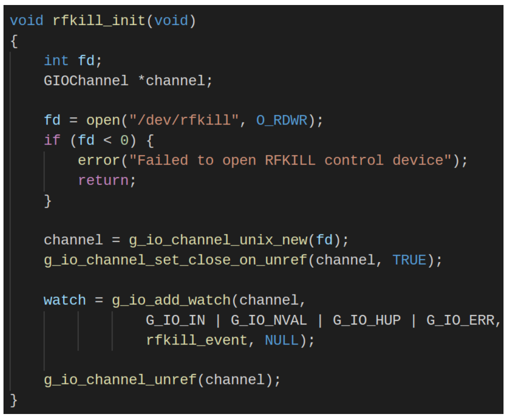


(b) Bluez stack taken from Intel Edison docs

its current RFKill state registered in `sysfs`. `rfkill` is a command-line tool with which can be queried to obtain and/or change the status of rfkill-enabled devices on the system.

At user-space level, the reccomended way to interface with `rfkill` subsystem is through the `/dev/rfkill` character device, which allows to obtain and set the state of rfkill devices via the simple `read`/`write` system calls . For example, figure [??] shows a portion of code where the `bluetoothd` daemon opens the rfkill device in order to grab information about devices present on the system. Later on, the corresponding file descriptor is wrapped around a *GLib IO Channel*[33], a data structure used to provide a portable method for using file descriptors, pipes and sockets. In the code a new channel to the rfkill file is created and then a watchdog that handles rfkill events is attached to monitor each changes of such file (similarly to `poll()` system call).

When the `rfkill_event` watchdog is called, it reads a `rfkill_event` structure from the character device and consult the ”`/sys/class/rfkill/`” directory to grab information on the device. Then, it calls `btd_adapter_restore_powered` which, at low level, perform a `mgmt`[5.1] request to change device's power options.



```
void rfkill_init(void)
{
    int fd;
    GIOChannel *channel;

    fd = open("/dev/rfkill", O_RDWR);
    if (fd < 0) {
        error("Failed to open RFKILL control device");
        return;
    }

    channel = g_io_channel_unix_new(fd);
    g_io_channel_set_close_on_unref(channel, TRUE);

    watch = g_io_add_watch(channel,
                G_IO_IN | G_IO_NVAL | G_IO_HUP | G_IO_ERR,
                rfkill_event, NULL);

    g_io_channel_unref(channel);
}
```

Figure 17: `bluetoothd` uses `rfkill` to discover available controller

In fact, by running `btmon`[34] when an device is deactivated via ”rfkill”, the output shows that first an HCI ”*Intel SW RF Kill*” command is issued and then the `mgmt` event *”Class Of Device Changed”* is received (indicating that the device where powered off).

```
struct rfkill_event {
        uint32_t idx;
        uint8_t  type;
        uint8_t  op;
        uint8_t  soft;
        uint8_t  hard;
};
```

[33] Although designed for powerful and complex GNOME applications, Glib is actually a separate library from GNOME and readily usable from any C application

[34] A tool similar to `hcidump` that also supports `mgmt` commands

5.4 Bluetooth Audio

The Bluetooth standard specifies three audio profiles, each of them further divided into two roles:

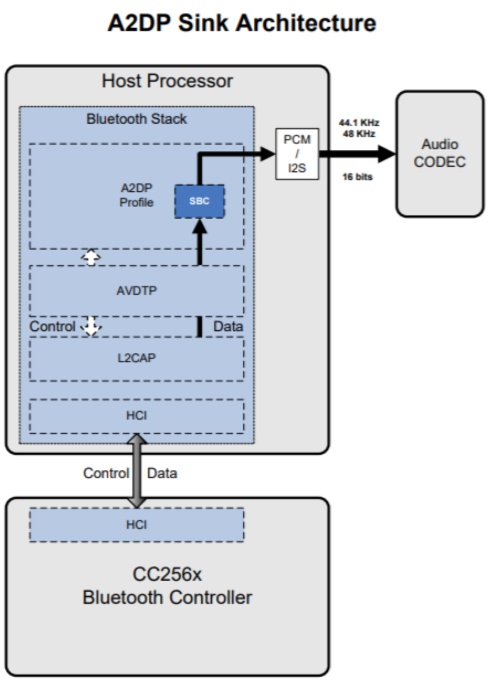
- **A2DP** (*Advanced Audio Distribution Profile*): high-quality audio playback, appropriate for listening to music.
  - *Source role*: the device that sends audio.
  - *Sink role*: the device that receives audio
- **HSP** (*HeadSet Profile*): phone-quality audio playback and recording, appropriate for phone calls
  - *Audio Gateway role*: the device that the headset is connected to. The HSP profile is typically used in phone calls, and this is the device that is connected to the cellular network
  - *Headset role*: the headset, obviously. This is where the speakers and microphone are.
- **HFP** (*Hands-Free Profile*): same as HSP, but with additional functionality for managing phone calls.
  - *Audio Gateway role*: the device that the hands-free device is connected to. The HFP profile is typically used for cellular phone calls, and this is the device that is connected to the cellular network.
  - *Hands-Free Unit role*: the device with the speakers and microphone.

The *Advanced Audio Distribution Profile* is the Bluetooth stereo profile which defines how high quality stereo audio can be transmitted from one device to another over a Bluetooth connection. Despite other Bluetooth audio profiles exists, they serves different purpose (e.g. only AD2P can provide stereo audio). At low-level, A2DP uses ACL packets, whereas voice calls (*Hands-free profile* **HFP**) uses SCO packets and thus cannot be to stream music. The AD2P profile is usually used in conjunction with the *Audio/Video Remote Control Profile* (**AVRCP**) to support remote control functionalities like pause or volume changes, which is implemented over L2CAP, whereas the *Headset* and *HandsFree* profiles employs RFCOMM for control channel.

Linux implements Bluetooth audio by combining BlueZ with the `PulseAudio` package: this one acts as a *sound server*, which essentially is a background process accepting sound input from one or more sources (processes, capture devices, etc.) and that later redirects these sound sources to one or more **sinks** (sound cards or other processes). In the case of Bluetooth, PulseAudio communicates with the Bluetooth daemon [5.2] provided by BlueZ and create a new virtual sound card.

By looking at the source code of Pulse Audio, it is possible to see the definition of the BlueZ's D-Bus classes and interfaces inside `bluez5-util.c` [5.2.1]. In fact, the Pulse Audio daemon interacts with Bluetooth daemon through D-Bus messages via the `Media` interface.

On the BlueZ side, the part of the code responsible for implementing A2DP can be found inside `profiles/audio`. The official Audio/Video Bluetooth specifications introduces three main protocol for managing media:



1. **A2DP**: used to manage audio, implemented at `a2dp.c`
2. **AVDTP**: used to distribute media, implemented at `avdtp.c`
3. **AVRCP**: used for remote control, implemented at `avrcp.c`

The Bluetooth daemon contains the configuration relative to A2DP profiles (along with general audio and HFP config) inside the "`/etc/bluetooth/audio.conf`"[35] file. Here is it possible to enable/disable support for specific services and configure profiles options (like the audio coded used for A2DP).

Once a Bluetooth device that supports A2DP is connected, PulseAudio should automatically create a virtual sound card like in the following output , which refers to a pair of wireless headphones, that shows how the sound-server view the device. The properties sections specifies the data needed to interact with the Bluetooth D-Bus API, while in the *Profiles* section two on them are available. The first one is the A2DP sink, which is used to output audio. The latter one, instead, is the *Headset Head Unit*, a protocol based on SCO which is used for calls. In fact, by viewing the device *Ports section*, the `headset_head_unit` is present both in output and input profile (a call is a bi-directional communication) whereas A2DP only appear in outputs.

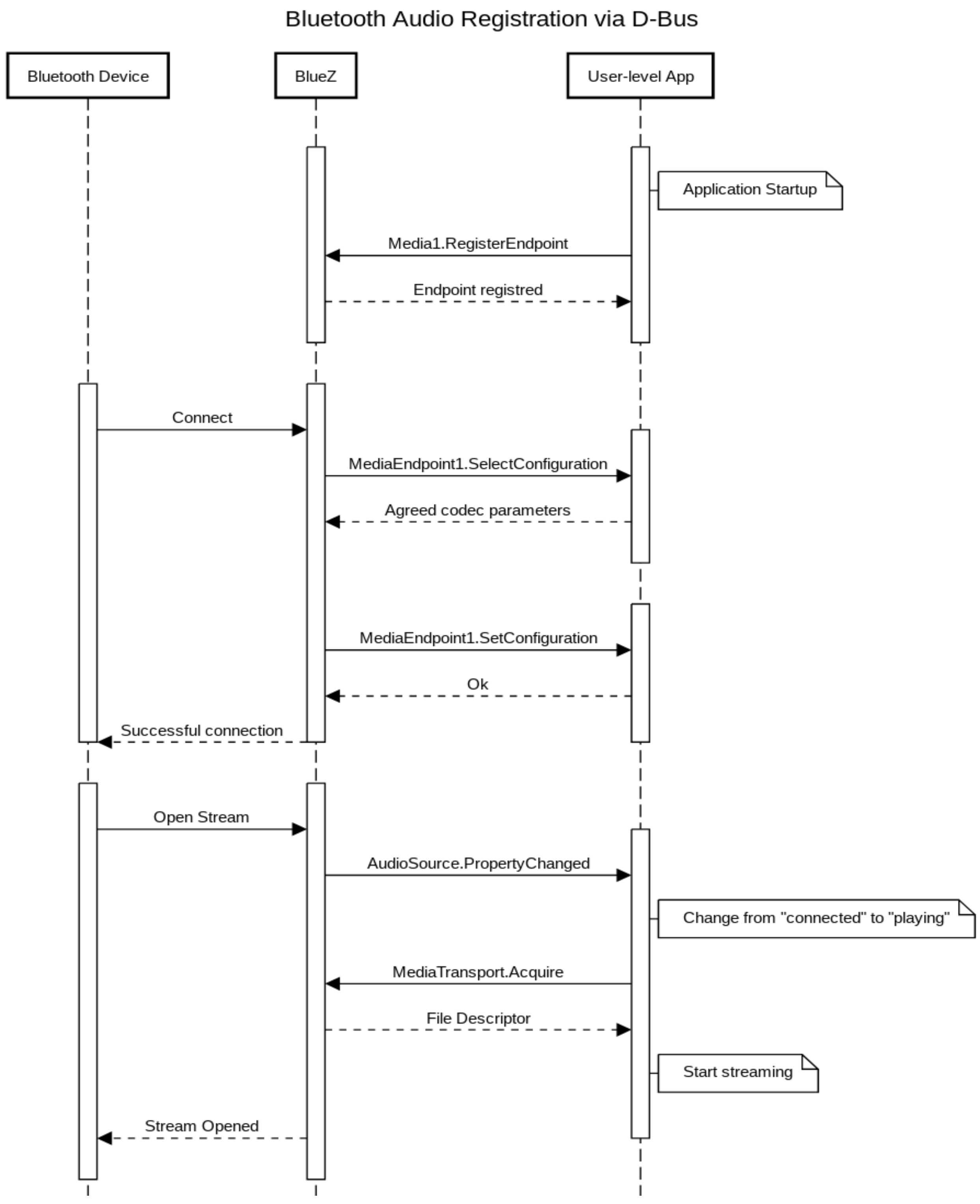
```
ubuntu@server:# pactl list cards
Card #0
  Name: bluez_card.XX_XX_XX_XX_XX_XX
  Driver: module-bluez5-device.c
  Owner Module: 23
  Properties:
        device.description = "Bluetooth Wireless Headphones"
        device.string = "XX:XX:XX:XX:XX:XX"
        device.api = "bluez"
        device.class = "sound"
        device.bus = "bluetooth"
        device.form_factor = "headphone"
        bluez.path = "/org/bluez/hci0/dev_XX_XX_XX_XX_XX_XX"
        bluez.class = "0x240418"
        bluez.alias = "Bluetooth Wireless Headphones"
        device.icon_name = "audio-headphones-bluetooth"
  Profiles:
        headset_head_unit: Headset Head Unit (HSP/HFP) (...)
        a2dp_sink: High Fidelity Playback (A2DP Sink) (...)
        off: Off (...)
  Active Profile: headset_head_unit
  Ports:
        headphone-output: Headphone (...)
                Part of profile(s): headset_head_unit, a2dp_sink
        headphone-input: Bluetooth Input (...)
                Part of profile(s): headset_head_unit
```

At this point, the card profile should be set from the available options (in this case `headset_head_unit` or `a2dp_sink`) by using the following command, where CARD_ID is the ID of the Bluetooth device sound card outputted by the previous command:

```
pactl set-card-profile CARD_ID a2dp_sink
```

[35]In BlueZ-5 this file is no longer present, and all the needed configuration must be written directly into the `/etc/bluetooth/main.conf` file

Below there is a scratch of the D-Bus interaction between the Bluetooth controller, BlueZ and an application that wants to use A2DP for sending/receiving audio.



When an user-level app starts, it has to tell BlueZ that it needs to handle A2DP sinks or sources. This is done by calling the Media method `RegisterEndpoint`, where both a profile UUID and an audio codec must be specified. Currently BlueZ implements four types of codecs [36], and usually **SBC** (*Low-complexity subband codec*) is the most popular option due to the fact that it is the default codec and its implementation is mandatory for devices supporting the A2DP profile. Anyway, vendors are free to add their own codecs to match their needs (as shown in the last line of [5.4]).

```
#define A2DP_CODEC_SBC              0x00
#define A2DP_CODEC_MPEG12           0x01
#define A2DP_CODEC_MPEG24           0x02
#define A2DP_CODEC_ATRAC            0x04
#define A2DP_CODEC_VENDOR           0xFF
```

At this point the app waits until a Bluetooth A2DP device connects to the adapter. When this happens, BlueZ will automatically notify the app again by using the `MediaEndpoint.SelectConfiguration` event, which contains the device's available codecs and other parameters. The user-space application will then have to chose the right parameters according to its need (obviously trying to provide the best-audio quality). If everything succeded, BlueZ will issue an event `MediaEndpoint.SetConfiguration` that should contain exactly the same codec parameters that the app chose in the previous step. Apart from this check, the event contains a parameter "*transport path*" which will be needed later.
Once the remote device opens an audio streams, BlueZ will trigger an `AudioSource.PropertyChanged` event: user-space applications must listen for such events and keep track of its "*state*"[37] property (which in this case switches from "*connected*" to "*playing*").
At this point the application should call `MediaTransport.Acquire` method passing the previously fetched "*transport path*" parameter. BlueZ will then answer with a file descriptor where the app can read from along with some other parameters such as the read MTU value.
Internally, BlueZ handles these transmission through a `media_transport` structure which contains all data necessary to act as a middleman between the remote device and the appliation, such as a `btd_device`[38] member that is used to communicate with the remote device, the file descriptor `fd` that is returned to the user-space application and the input/output MTU value.

```
struct mediatransport –
        char     *path; /* Transport object path */
        struct btddevice    *device;          /* Transport device */
        const char *remoteendpoint; /* Transport remote SEP */
        struct mediaendpoint *endpoint;           /* Transport endpoint*/
        struct mediaowner        *owner;                /* Transport owner */
        int          size;                /* Transport configuration size*/
        int          fd;                /* Transport file descriptor */
        uint16t      imtu;                /* Transport input mtu */
        uint16t      omtu;                /* Transport output mtu */
        transportstatet    state;
    ...
        guint    (*resume) (struct mediatransport *transport,
                                struct mediaowner *owner);
        guint    (*suspend) (struct mediatransport *transport,
                                struct mediaowner *owner);
        void     (*cancel) (struct mediatransport *transport, guint id);
    ...
;
```

[36] Defined at `profiles/audio/a2dp-codecs.h`
[37] A complete list of possible value is documented at audio-api.txt
[38] Defined inside `src/device.c`

6 Conclusion

To conclude the discussion, the output of `dmesg` is presented in order to show the order in which Bluetooth stack components are initialized by the kernel[39].
On line 2 the Bluetooth protocol (identified by the number 31 in `/etc/protocols`) is registered in the kernel. Immediately after this (line 3-4) the HCI layer is initialized and, consequently, also the L2CAP and SCO sockets are (line 5-6). Once the software layer of the Bluetooth stack are ready, the OS loads the HCI UART driver [3.2] needed to communicate with the Bluetooth chipset installed on the device[40] (line 7-17). Consequently, all the higher layer protocol like BNEP and RFCOMM are initialized (line 18-23) and then, at user-level, the Bluetoot daemon is started.

```
[1] Bluetooth: Core ver 2.22
[2] NET: Registered protocol family 31
[3] Bluetooth: HCI device and connection manager initialized
[4] Bluetooth: HCI socket layer initialized
[5] Bluetooth: L2CAP socket layer initialized
[6] Bluetooth: SCO socket layer initialized
[7] Bluetooth: HCI UART driver ver 2.3
[8] Bluetooth: HCI UART protocol H4 registered
[9] Bluetooth: HCI UART protocol BCSP registered
[10] Bluetooth: HCI UART protocol LL registered
[1] Bluetooth: HCI UART protocol ATH3K registered
[12] Bluetooth: HCI UART protocol Three-wire (H5) registered
[13] Bluetooth: HCI UART protocol Intel registered
[14] Bluetooth: HCI UART protocol Broadcom registered
[15] Bluetooth: HCI UART protocol QCA registered
[16] Bluetooth: HCI UART protocol AG6XX registered
[17] Bluetooth: HCI UART protocol Marvell registered
[18] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[19] Bluetooth: BNEP filters: protocol multicast
[20] Bluetooth: BNEP socket layer initialized
[21] Bluetooth: RFCOMM TTY layer initialized
[22] Bluetooth: RFCOMM socket layer initialized
[23] Bluetooth: RFCOMM ver 1.11
```

[39]The output taken from a Raspberry Pi 3b running Ubuntu Core version

[40]The Raspberry Pi has the Bluetooth chipset connected through UART, on different devices the driver might change (e.g. HCI-USB)