Affix - Technical Documentation

Dmitry Kasatkin, Henri Ranki

Affix - Technical Documentation

by Dmitry Kasatkin, Henri Ranki

Table of Contents

List of Tables

Preface

This document describes Affix[tm] Bluetooth Protocol stack for Linux developed in Nokia Research Center at Mobile Networks Lab and released under GPL. This documentation applies to Affix version 3.2.0.

While we believe that Affix is a useful piece of software, please bear in mind that it is not an official Nokia product, but a result of the research activity of Nokia Research Center. For further details, please read the files README, COPYING and LEGAL in the tar archive.

The purpose of this document is to give a user advanced information about the Affix Bluetooth Protocol Stack for Linux.

1. Revision history

This document descents from Dmitry Kasatkin's document "Affix in a Nutshell". The name was changed to better correspond the content of the document. All the stuff for the end users was moved to Affix End User Manual. For other changes see the revision history below.

Data	Version	Description
18.10.2002	0.1	Initial version, general info
25.10.2002	0.2	Socket, HCI API description included.
02.11.2002	0.3	HCI API, Affix component diagram.
07.11.2002	0.4	Installation instruction, HCI Data structures
15.11.2002	0.5	Affix security model, Security API, tools manual (initial), SDP API (initial)
22.11.2002	0.6	Tools manual (complete), SDP API (complete)
27.11.2002	0.7	OBEX API, PAN configuration, some additions
22.01.2003	0.8	PAN extensive info.
06.03.2003	0.81	BCM2033 and 3Com PCCARD installation note.
2.7.2004	0.9	Removed Usage manual, manual pages and installation instructions. These were moved to separate Affix - End User Manual.
9.8.2004	0.91	Chapters for PAN and device cache. Added information on Python support to RFCOMM chapter. Updated L2CAP documentation to version 1.2.
31.8.2004	1.0	Update to Affix 3.2.0. Updated SDP documentation. Added a chapter about HID profile. This is an edited version Wolfgang Hansmann's report. Renamed documentation to avoid confusion with the old document.

2. Contact Information

Public mailing lists:

- Affix Developers' mailing list: <affix-devel@lists.sourceforge.net> (subscribe, archive (http://lists.sourceforge.net/lists/listinfo/affix-devel))
- Affix support mailing list: <affix-support@lists.sourceforge.net> (subscribe, archive (http://lists.sourceforge.net/lists/listinfo/affix-support))

3. Terms and Abbreviations

PAN	Personal Area Networking Profile
HCI	Host Controller Interface
HID	Human Interface Device

Chapter 1. Introduction

1.1. Overview

Affix supports core Bluetooth protocols like HCI, L2CAP, RFCOMM, SDP and various Bluetooth profiles (see bellow).

Affix features:

- Modular implementation.
- Socket interface to HCI, L2CAP and RFCOMM protocols.
- Bluetooth module interface independence.
- SMP safe.
- Multiple Bluetooth devices support.

Affix runs on major architectures:

- i386.
- ARM (e.g. Compaq iPaq).
- PowerPC (e.g. iMac).
- Sparc.

In general Affix can be run on any other architectures running Linux.

Affix supports wide range of hardware:

- USB based devices
  - 3Com, CSR, NSC, Ericsson, etc.
- PCMCIA based device
  - Nokia DTL1/DTL4, Socket CF, Anycom, Xircom, etc.
- UART based devices
  - Ericsson, Philips, etc.

Affix currently supports the following Bluetooth Profiles:

- General Access Profile
- Service Discovery Profile
- Serial Port Profile
- DialUp Networking Profile
- LAN Access Profile
- OBEX Object Push Profile
- OBEX File Transfer Profile
- PAN Profile
- HID Profile

Affix provides unified interface for all transport drivers. It makes it hardware independent.

Affix consists of following packages:

- affix-kernel
- affix

affix-kernel provides kernel modules implementing core protocols and Bluetooth device drivers. Kernel modules can be used separately from the kernel or can be linked statically into the kernel.

affix provides control tools, libraries, and server daemons.

Also a module for Python (*PyAffix*) is available. It can be downloaded from Affix web pages.

We have successfully tested Affix for interoperability with many devices, e.g:

- Nokia 6210 with Bluetooth battery pack.
- Nokia 6310/6310i.
- Nokia 7650.
- Ericsson phones (T68).
- Compaq iPaq.
- Digianswer stack for Windows.
- Bluetooth stack for Palm OS.

Chapter 2. General Architecture

2.1. Linux Core Concept

The Linux operating system implements the standard Berkeley socket API, which has its origins in the BSD Unix developments (4.2/4.3/4.4 BSD) RUSL99.

The Linux Network Subsystem architecture is shown on Figure 2-1[BTSPEC]. The standard components of the Linux network subsystem are Berkeley socket interface and Network Device Driver Interface.

Figure 2-1. Linux networking architecture

The Berkeley Socket Interface allows user space programs to open communication endpoint to remote devices. Socket is a network abstraction of the channel endpoint. Socket is associated with the protocol. Usually, the PF_INET is used to associate a socket with the TCP/IP protocol.

Network Device Driver Interface enables multiple network devices to be used at the same time. Device driver registers device in the system. And each device has appropriate type to distinguish class of devices like Ethernet, PPP, X.25, etc.

The network driver interface includes packet scheduler interface. Packet scheduler implements queuing discipline for different type of devices.

The protocol component is an actual protocol implementation. Each protocol should registers itself in the socket interface with appropriate protocol family (PF_XXX) and in the network device driver interface with the appropriate protocol type. Each packet received will be delivered to the appropriate protocol layer.

The networking layer is fairly object-oriented in its design, as indeed is much of the Linux kernel. The core structure of the networking code goes back to the initial networking and socket implementations by Ross Biro and Orest Zborowski. The key objects are:

- *Device or Interface*: A network interface is programming code for sending and receiving data packets. Usually an interface is used for a physical device like an Ethernet card; however, some devices are software only, e.g., the loop back device used for sending data to yourself.
- *Protocol:* Each protocol is effectively a different networking language. Some protocols exist purely because vendors chose to use proprietary networking schemes, while others are designed for special purposes. Within the Linux kernel each protocol is a separate module of code providing services to the socket layer.

- *Socket:* A socket is a connection endpoint providing Unix file I/O and exists as a file descriptor to the user program. In the kernel each socket is a pair of structures that represent the high level socket interface and the low level protocol interface.
- *sk_buff:* All the buffers used by the networking layers are sk_buffs. The control for these buffers is provided by core low-level library routines that are available to all of the networking system. sk_buffs provide the general buffering and flow control facilities needed by network protocols.

2.2. Affix Architecture

Generally Affix software can be divided into two components (according to protocol stack):

- Protocol component. This component implements protocol stack.
- Management component. This component controls the protocol component and represents the software intelligence.

The general software architecture is shown on Figure 2-2

Figure 2-2. Affix architectural model

The philosophy of the Unix (and Linux) operating system is the following that the kernel should provide basic mechanisms and the user space applications should implement logic [STEV97]. The bottleneck of this approach is problem of performance. In this case it is possible to move some functionality from the user space to the kernel space. The only reason to put more functionality to the kernel space is to increase the performance of the system.

Other key benefits of having more functionality in the user space are easier implementation process for the user space and greater stability of the system, which is less likely to crash due to bugs in the applications. It is possible to use wider range of debugging facilities.

According to the philosophy of the Unix operating system the most appropriate way to implement basic Bluetooth protocol stack to be running in the kernel space and management entity (later ME) to be running in the user space. Affix follows this concept.

The Bluetooth protocol stack has predefined functionality and is defined in the Bluetooth system specification. The protocol stack of different systems should have the same functionality. Otherwise interoperability is not guaranteed.

All network data traffic comes to Affix from the kernel space. Thus optimal solution for performance reason is to implement core protocols in the kernel space.

The ME has to complement the Bluetooth protocol stack functionality and can have additional functionality depending on the goal of the system. It is implemented as a user space application. The performance is not very critical here. In this case it is possible to re-implement the ME without modifying the kernel space code.

Practical Affix component diagram is shown on Figure 2-3.

Figure 2-3. Affix components

2.3. Bluetooth and Affix Security Models

Bluetooth has internal security mechanisms: authentication and encryption. And there is documentation that contains recommendation and requirements how security must be achieved. One security mechanism is based on a PIN code which user has to type to be able to connect to a device or allow other devices to connect its device.

Affix has Bluetooth security support conducted through introducing security modes and security levels.

Affix has following security modes:

- OPEN. This mode allows any device to connect to any local service without authentication (without PIN code).
- LINK. This mode performs security actions at the link level. Before link level connection (ACL) is established device must pass security check (depending on the security level). After connection is established device gets access to all services. In other words, to connect to any service security actions have to be passed.
- SERVICE. This mode performs security actions at the service level. Device can create link level connection without any security actions, but to connect to a service must pass security check. This mode is more flexible comparing to link level because it allows to specify what level of security service involves.

Affix has following security levels:

- OPEN. No security actions involved.
- AUTH. A PIN code based authentication is required to pass security check.
- ENCRYPT. A connection encryption is required. This level includes AUTH level.
- AUTHOR. An authorization is required to pass security check. User must manually confirm the action (e.g. Accepts connection).

Those mechanisms make Affix very secure and flexible. To select desired security level user has to set security mode and security level for that mode. For example, to select SERVICE mode security and require AUTH authentication and AUTHOR authorization to be involved.

Chapter 3. Application Programming Interface

Affix design allows user to implement applications at the kernel level (as kernel modules) and user level (as applications). For this reason Affix exports following interfaces:

- *Kernel programming interface*. A set of functions, which includes HCI, L2CAP, RFCOMM and UART modules.
- *Socket interface*. A first level API that is used to utilize core Bluetooth Protocols like L2CAP and RFCOMM. Use it in the same way as other type of sockets (e.g. PF_INET).
- *Shared libraries*. Various set of functions.
- *Kernel variables*. Use these to get information on the state of Affix and your Bluetooth device. Also you can set several parameters to control Affix. Read more in chapter Debugging Affix.

3.1. Affix API Components

Affix API components include following entities: incorporated in header files and libraries. Default installation path for header files and libraries is /usr/include/affix and /usr/lib respectively.

- Header files (/usr/include/affix)
- Libraries (/usr/local/lib)
- Kernel modules (/lib/modules/‘uname -r‘/kernel/{net/affix,drivers/affix}

Table 3-1. Affix header files

Header file	Description
bluetooth.h	Main header file. Has to be included in all programs. Contains main system Affix type definitions and function declarations. Provides interface between kernel and user parts.
hci_types.h	Contains all HCI type definitions. (defined according to the Bluetooth Specifications).
hci_cmds.h	Contains HCI commands implementation. (defined according to the Bluetooth Specification).
btcore.h	Contains user-level Affix system specific definitions and declarations.
utils.h	Contains declarations of common tools for the Affix.
sdp.h	Contains SDP type definitions.
sdpclt.h	Contains SDP client interface declarations.
sdpsrv.h	Contains SDP server interface declarations.
obex.h	Contains OBEX client interface declarations.

Table 3-2. Affix libraries

Library	Description
libaffix.so	Main Affix library. Contains Affix and Bluetooth core implementation. All programs must link with it.
libaffix_utils.so	Contains implementation of general purpose functions used in the Affix.
libaffix_sdp.so	Contains SDP client API implementation. Any program using SDP has to link with it.
libaffix_sdpsvc.so	contains SDP server API implementation. Only programs registering services link with it.
libaffix_obex.so	Contains Bluetooth OBEX framework implementation. Any program using OBEX must link with it.

Table 3-3. Affix kernel modules

Module	Description
affix.ko	Affix core module (HCI, L2CAP, and its socket interfaces)
affix_rfcomm.ko	RFCOMM, BTY and its socket interfaces
affix_pan.ko	PAN (BNEP, network interface)
affix_usb.ko	Bluetooth USB driver
affix_uart.ko	Bluetooth UART (H4, BCSP, TLP) driver
affix_uart_cs.ko	PCMCIA driver for UART cards
affix_hidp	HID device support implementation

3.2. Socket Interface

The Socket interface provides a standard, well-documented approach to access kernel network resources. The Affix brings to the system new protocol family PF_AFFIX and new socket address family "struct sockaddr_affix":

<affix/bluetooth.h> must be included to the source file.

Standard "socket" system call is used to create a socket of PF_AFFIX family:

```
int socket(PF_AFFIX, int type, int protocol);
```

where *type* parameter depends on the *protocol* parameter, which accepts following values:

- BTPROTO_HCIACL. Creates HCI ACL socket. Valid type: SOCK_SEQPACKET.
- BTPROTO_HCISCO. Creates HCI SCO socket. Valid type: SOCK_SEQPACKET.
- BTPROTO_L2CAP. Creates L2CAP socket. Valid type: SOCK_STREAM, SOCK_SEQPACKET.

- BTPROTO_L2CAP_RET. Create L2CAP socket using the retransmission mode (L2CAP 1.2). If the remote host does not support L2CAP 1.2 it is connected in basic mode automatically. Valid type: SOCK_STREAM.
- BTPROTO_L2CAP_FC. Create L2CAP socket using flow control (L2CAP 1.2). If the remote device does not support L2CAP 1.2. It automatically connected in basic mode. Valid type: SOCK_STREAM.
- BTPROTO_RFCOMM. Creates RFCOMM socket. Valid type: SOCK_STREAM.

3.2.1. Data Structures

The Socket interface has only one data structure as any protocol family in Unix. It is a cousin of sockaddr. The structure has a name "sockaddr_affix" and has the following prototype:

```
struct sockaddr_affix {
    sa_family_t family;
    int         devnum;
    BD_ADDR     bda;
    uint16_t    port;
};
```

All fields except port has the same meaning for all protocols inside PF_AFFIX family. The "port" field has different meaning in for different protocols:

- BTPROTO_L2CAP, BTPROTO_L2CAP_RET and BTPROTO_L2CAP_FC: "port" defines PSM value.
- BTPROTO_RFCOMM: "port" defines server channel number.
- BTPROTO_HCIACL and BTPROTO_HCISCO: "port" field is unused.

3.2.2. Function Set

The PF_AFFIX socket family allows to use standard socket functions like connect(), bind(), listen(), accept(), send(), recv() and has additional Bluetooth and Affix specific extension. More details about socket call can be found in the UNIX manual pages.

The PF_AFFIX extension practically is implemented as ioctl() and set(get)sockopt() system calls, but for more convenient it is wrapped into the well-named functions.

Table 3-4. PF_AFFIX extra API

Function	Applicability	Purpose

Function	Applicability	Purpose
int l2cap_setmtu(int fd, int mtu);	BTPROTO_L2CAP	Sets a maximum packet size that can be received on this socket. This function is used before calling connect() or bind().
int l2cap_getmtu(int fd);	BTPROTO_L2CAP	Gets a maximum packet size that can be sent through this socket.
int hci_getmtu(int fd);	BTPROTO_HCIACL, BT-PROTO_HCISCO	Gets a maximum packet size that can be sent through this socket.
int l2cap_ping(int fd, char *data, int size);	BTPROTO_L2CAP	Sends a *ping* packet and wait for response.

The RFCOMM socket has additional feature: it can be bound to TTY line. The Affix uses "/dev/btyXX" names for that, where XX is a registered line. Special functions are used to do it:

Table 3-5. RFCOMM TTY API

Function	Purpose
int rfcomm_open_tty(int fd, int line);	Attaches an RFCOMM connection to a TTY line *line*. If the *line* is -1 then it allocates a unused tty line and returns its number.
int rfcomm_close_tty(int line);	Detaches an RFCOMM connection from a TTY line *line* and close it.
int rfcomm_set_type(int fd, int type);	Changes the RFCOMM socket type to: RFCOMM_SOCKET (default) or RFCOMM_BTY. This function has to be called right after *socket()*.
int rfcomm_bind_tty(int fd, struct sockaddr_affix *sa, int line)	Binds an RFCOMM connection to a TTY line *line*. Connection is not established yet, but the line is reserved. When into the TTY is written the line is connected automatically.

3.2.3. Sample Code

Practical examples can be found in the Affix source code. Here are examples showing how to use PF_AFFIX sockets on a client and a server sides.

The following example illustrates usage of the Affix socket interface on the client side.

```
#include <stdio.h>
```

```
/* Affix includes */
#include <affix/bluetooth.h>

int main(int argc, char *argv[])
{
     struct sockaddr_affix   sa;
     int  fd, err;

     fd = socket(PF_AFFIX, SOCK_SEQPACKET, BTPROTO_L2CAP);
     if (fd < 0) {
           perror("socket() failed");
           return 1;
     }

     sa.family = PF_AFFIX;
     str2bda(&sa.bda, "00:11:22:33:44:55");   /* connect to that device */
     /* or sa.bda = other_bda; */
     sa.port = 1;                             /* connect to that port (PSM) */
     sa.local = BDADDR_ANY;
     /*
        to connect through certain device use:
         sa.local = <local device bda>
     */

     err = connect(fd, (struct sockaddr*)&sa, sizeof(sa));
     if (err) {
           perror("connect() failed");
           return 2;
     }

     /*
        here is socket is ready for communication
        any of the standard connection oriented
        transmission/receiving system calls can be used.
        - send(), sendmsg(), write()
        - recv(), recvmsg(), read()
     */


     close(fd);

     return 0;
}
```

The next example illustrates usage of the socket interface on the server side.

```
#include <stdio.h>

/* Affix includes */
#include <affix/bluetooth.h>

int main(int argc, char *argv[])
```

```
{
     struct sockaddr_affix   sa, csa;
     int   fd, cfd, err;
     socklen_t csa_len;

     /* create server socket */
     fd = socket(PF_AFFIX, SOCK_SEQPACKET, BTPROTO_L2CAP);
     if (fd < 0) {
           perror("socket() failed");
           return 1;
     }

     sa.family = PF_AFFIX;
     sa.bda = BDADDR_ANY
     sa.port = 1;            /* accept connection to that port (PSM) */
     /*
        to connect through certain device use:
         sa.local = <local device bda>
     */

     /* bind socket to address specified by "sa" parameter */
     err = bind(fd, (struct sockaddr*)&sa, sizeof(sa));
     if (err) {
           perror("bind() failed");
           return 2;
     }

     /* start listen for connection - kernel will accept connection requests */
     err = listen(fd, 5)
     if (err) {
           perror("listen() failed");
           return 3;
     }

     /* accept new connection and get its connection descriptor "cfd" */
     csalen = sizeof(csa);
     cfd = accept(fd, (struct sockaddr*)&csa, &csa_len);
     if (cfd < 0) {
           perror("accept() failed");
           return 4;
     }

     /*
        here is socket "cfd" is ready for communication
        any of the standard connection oriented
        transmission/receiving system calls can be used.
        - send(), sendmsg(), write()
        - recv(), recvmsg(), read()
     */


     close(fd);  /* close server socket */
     close(cfd); /* close client socket */
```

```
    return 0;
}
```

3.2.4. Socket Options

To set socket options use *setsockopt()* function. The definition for the function is:

```
int setsockopt(int fd, int level, int optname, void *optval, int *optlen);
```

Affix supports socket level SOL_AFFIX and the following table lists socket options.

Table 3-6. Supported socket options

Option	Purpose
BTSO_MTU	Set the maximum transfer unit.
BTSO_SECURITY	Set the socket level security. All services operate above sockets, or at least connection establishment process is done above it. The Service Level security mode allows service to set its security level. It's done through a special security API. Possible modes are HCI_SECURITY_OPEN, HCI_SECURITY_AUTH, HCI_SECURITY_ENCRYPT, HCI_SECURITY_AUTHOR. Modes can bitwise-or'd
BTSO_EVENT_MASK	Defines what events you want to handle. See possible values in hci_types.h
BTSO_PKT_MASK	Define what kind of packets socket should receive. Types are HCI_COMMAND, HCI_ACL, HCI_SCO, HCI_EVENT, HCI_PKT_OUTGOING and HCI_MGR. See definitions in hci_types.h.
BTSO_PROMISC	Allow the socket to receive all the packets.
BTSO_TYPE	Not used

3.2.5. Python Support

Affix implements socket interface also for Python programing language. *PyAffix* package contains

module for Python. It supports HCI, L2CAP, RFCOMM and non-blocking sockets. To use Affix with Python install Affix kernel module and make sure you have Python version 2.2.3 or later installed.

To installing Affix module in your Python do the following:

1. Download latest PyAffix package from Affix web site (http://affix.sourceforge.net/pyaffix.shtml)
2. Extract the downloaded tar ball and change to that directory
3. Give the following command:

```
python setup.py install
```

Also in Python you can use standard socket functions like connect(), bind(), listen(), accept(), send() and recv(). To create Affix socket you need to call *socket()* function like:

```
import affixsocket /* Just import Affix module and you are ready to use Affix */
affix_sock = socket(PF_AFFIX,SOCK_STREAM,BTPROTO_L2CAP)
```

Above example creates a new Affix socket object. Protocols and types are the same as listed in the beginning of this chapter. Connecting to remote host using newly created affix_sock object is done with *connect* method. The socket address information is represented in the following tuple in Python: (devnum, BTaddress, port). Where devnum is a positive integer representing the device, BTaddress is a string representing the Bluetooth address and port is the channel/psm/etc to connect to.

```
aff_sock.connect((0,"01:02:03:A0:B0:C0",1))
```

To set socket options use *setsockopt(level, optname, value)* method. See supported levels and options in chapter Section 3.2.4.

3.3. HCI API

The HCI API allows user to perform low level control of Bluetooth devices, to set it up into different modes of operation, to discover other Bluetooth devices in the radio range, and so forth. There are almost no cases when HCI functions are unused. The most common task is to make a discovery to find what devices are around.

The main notion in the HCI API is a HCI descriptor, which in fact is a Unix file descriptor. The HCI descriptor is used in many functions to specify device used with a calling function.

3.3.1. Function Set

The HCI API function set divided into two subsets: system (Affix specific) function set and HCI (Bluetooth device specific) function set. All Affix function names have low-case style and all a HCI

function names have mixed-case style and are starting from HCI_ or __HCI_. The HCI functions are implemented according to the Bluetooth System Specification [BTSPEC].

The Affix system function set is presented in Table 3-7.

Table 3-7. HCI API function set

function prototype	Description
int hci_open(char *name);	Opens device "by name" and returns HCI descriptor.
int hci_open_id(int hd);	Opens device "by id" and returns HCI descriptor.
int hci_get_devs(int *devs);	Stores HCI descriptors of registered Bluetooth devices in the system in the *devs* and returns its number.
int hci_open_event(void);	Opens a channel used to receive HCI events from any device and returns HCI descriptor.
int hci_event_mask(int fd, __u64 mask);	Sets event mask for specified device and enable event receiving of matched type. Mask value zero disable events receiving.
int hci_recv_event(int fd, void *event, int size, long timeout);	Receives HCI event from the HCI devices. Event is stored in the buffer pointed by "event" argument.
int hci_exec_cmd(int fd, __u16 opcode, void *cmd, int len, __u64 mask, int flags, void *event, int elen);	Sends HCI commands pointed by "cmd" to the device pointed by "fd", waits for the COMMAND_STATUS or COMMAND_COMPLETE event and stores it in the buffer pointed by "event". "mask" argument defines events (mask) to receive.
int hci_open_mgr(void);	Opens Affix internal control channel an returns descriptor.
int hci_add_pin(BD_ADDR *bda, int Length, __u8 *Code);	Stores a pin code in the Affix control block. It is used in the case of it will be required. It’s used by non-interactive applications. This pin code is used to authenticate device *bda*.
int hci_remove_pin(BD_ADDR *bda);	Removes pin code from the Affix control block.
int hci_add_key(BD_ADDR *bda, __u8 key_type, __u8 *key);	Stores a link key in the Affix control block. It’s used if application stores link key for later use. Link key is used to authenticate device *bda*.
int hci_remove_key(BD_ADDR *bda);	Removes link key from Affix control block.
int hci_set_mode(int fd, int mode);	Sets functional mode of Affix kernel core (mode: AFFIX_MODE_PIN - caller handles PIN, AFFIX_MODE_KEY - caller handles key).
int hci_get_conn(int fd, BD_ADDR *bda);	Retrieves handle of HCI ACL/SCO connection to device *bda*.
int hci_get_attr(char *name, struct hci_dev_attr *attr);	Reads device attributes: name, id, flags.

function prototype	Description
int hci_get_attr_id(int hd, struct hci_dev_attr *attr);	Reads device attributes: name, id, flags.
int hci_get_attr_fd(int fd, struct hci_dev_attr *attr);	Reads device attributes: name, id, flags.
int hci_set_attr(int fd, struct hci_dev_attr *attr)	Set device attributes for opened device which handle is *fd*.
int hci_get_flags_id(int devnum, int *flags);	Reads device flags. Argument devnum identifies the device.
int hci_set_secmode(int fd, int secmode);	Sets security mode: HCI_SECURITY_OPEN, HCI_SECURITY_LINK, HCI_SECURITY_AUTH, HCI_SECURITY_AUTHOR). This modes can bitwise-or'd.
int hci_set_pkttype(int fd, int pkt_type);	Sets the accepted packet types.
int hci_get_pkttype(int fd, int *pkt_type);	Get the accepted packet types.
int hci_set_role(int fd, int role);	Set connection roles.
char *hcierror(int err);	Returns HCI error message.
int hci_ioctl(int cmd, void *arg);	Control the Bluetooth device. See BTIOC_ command in bluetooth.h
int hci_open_uart(char *name, int type, int proto, int speed, int flags);	Attach a serial Bluetooth device of type "type".
int hci_close_uart(char *name);	Detach a serial bluetooth device.
int hci_setup_uart(char *name, int proto, int speed, int flags);	Sets a serial Bluetooth device up.
int hci_devnum(char *name);	Get the device number for Bluetooth device in the given interface. Argument "name" is the interface. E.g. "bt0".
int hci_get_bda_id(int devnum, BD_ADDR *bda);	Get the Bluetooth address for a device number *devnum*
int hci_start_dev(int fd);	Starts the Bluetooth device wiht handle *fd*.
int hci_stop_dev(int fd);	Stop the bluetooth device with handle *fd*.
int hci_set_dbmask(__u32 dbmask);	Set the debug mask. See the possible values in btcore.c file. Defines which debug messagas are printed.
int hci_get_dbmask(__u32 *dbmask);	Get the current debug mask.
int hci_get_version(int fd, struct affix_version *ver)	Not implemented
int hci_disconnect(struct sockaddr_affix *sa)	Disconnect from a device.
int hci_getmtu(int fd)	Get the socket's maximum transfer unit
char *bda2str(BD_ADDR *bda);	Converts Bluetooth address to string.
int str2bda(BD_ADDR *p, char *str);	Converts string to Bluetooth address.

The Major HCI functions are presented in Table 3-8. All functions accept at least one parameter - HCI

descriptor and return an error code, which can has following values:

- Zero (0) if no error occurred
- -1 in a case of system error, and errno is set appropriately. (like any Linux system call does)
- Above zero (>0) in a case of HCI error (see Bluetooth Specification for values)

General HCI function format is: int HCI_Xxxx(int hd, param, param, ...);

The rest of parameters of HCI functions depend on actual function and is defined according to Bluetooth Specification. Its description can be carefully read from there.

The following table presents only partial set of HCI functions implemented in the Affix. A complete set can be found in `hci_cmds.h`.

Table 3-8. HCI function set

Function	Description
int HCI_Inquiry(int fd, __u8 Inquiry_Length, __u8 Max_Num_Responses, INQUIRY_ITEM *Items, __u8 *Num_Responses)	Performs INQUIRY during (1.28 * Inquiry_Length) sec. and stores found Bluetooth devices info in *Items* array and number of found devices in *Num_Responses*.
int HCI_WriteScanEnable(int fd, __u8 Scan_Enable);	Sets SCAN mode (SCAN_OFF, SCAN_INQUIRY, SCAN_PAGE, SCAN_BOTH).
int HCI_WriteClassOfDevice(int fd, __u32 Class_of_Device);	Sets a Class of Device.
int HCI_ChangeLocalName(int fd, char *Name);	Sets a device name. All remote devices will see this name.
int HCI_WriteAuthenticationEnable(int fd, __u8 Authentication_Enable);	Enables authentication.
int HCI_WriteEncryptionMode(int fd, __u8 Encryption_Mode);	Sets encryption mode for ACL connections: EM_NO, EM_PP (point-to-point), EM_ALL (pp and broadcast).
int HCI_ReadBDAddr(int fd, BD_ADDR *bda);	Reads address from device.
int HCI_WritePageTimeout(int fd, __u16 Page_Timeout);	Sets page timeout.
int HCI_WriteLinkPolicy(int fd, CHANDLE Connection_Handle, __u8 Link_Policy_Settings);	Sets Link Policy.

3.3.2. Data Structures

Data structures related to the HCI interface are defined in hci_types.h header.

- struct sockaddr_affix. See definition in chapter Section 3.2.1.
- INQUIRY_ITEM

```
typedef struct {
       BD_ADDR   bda;
       __u8      PS_Repetition_Mode;
       __u8      PS_Period_Mode;
       __u8      PS_Mode;
       __u32     Class_of_Device:24;
       __u16     Clock_Offset;
}__PACK__ INQUIRY_ITEM;
```

- struct hci_dev_attr.

```
/* flags mask */

#define HCI_FLAGS_UP            0x00000002

#define HCI_FLAGS_ROLE          0x000000F0
#define HCI_ROLE_ALLOW_SWITCH   0x00000000
#define HCI_ROLE_DENY_SWITCH    0x00000010
#define HCI_ROLE_REMAIN_SLAVE   0x00000000
#define HCI_ROLE_BECOME_MASTER  0x00000020

/* Security mode flags */
#define HCI_FLAGS_SECURITY      0x0000FF00
#define HCI_SECURITY_OPEN       0x00000100
#define HCI_SECURITY_SERVICE    0x00000200
#define HCI_SECURITY_LINK       0x00000400
#define HCI_SECURITY_PAIRABLE   0x00000800

/* Security levels */
#define HCI_SECURITY_AUTH       0x00001000
#define HCI_SECURITY_ENCRYPT    0x00002000
#define HCI_SECURITY_AUTHOR     0x00004000
#define HCI_SECURITY_OUT_AUTH   0x00100000
#define HCI_SECURITY_OUT_ENCRYPT 0x00200000
#define HCI_SECURITY_OUT_AUTHOR 0x00400000
#define HCI_SECURITY_CL         0x00800000 /* connection less traffic */

struct hci_dev_attr {
    int     devnum;
    char    name[IFNAMSIZ];
    BD_ADDR bda;
    int     flags;
    int     pkt_type;
    struct hcidev_stats stats;
};
```

- struct hci_msg_hdr.

```
#define MGR_STATE_CHANGE  0x01

struct hci_msg_hdr {
        int   opcode;
```

```
        int   length;
};
```

- struct hci_state_change.

```
/* hci device events */
#define HCIDEV_UP           0x0001
#define HCIDEV_DOWN         0x0002
#define HCIDEV_CHANGE       0x0004
#define HCIDEV_REGISTER     0x0005
#define HCIDEV_UNREGISTER   0x0006
#define HCIDEV_ATTACH       0x0100
#define HCIDEV_DETACH       0x0200

struct hci_state_change {
    struct hci_msg_hdr hdr;
    int     devnum;
    int     event;
};
```

3.3.3. Sample Code

Here is an example of the HCI API usage.

```
int main(int argc, char *argv[])
{
    int   devs[16], num, i, err;

    num = hci_get_devs(devs);
    if (num < 0) {
        printf("unable to get device list\n");
        return -1;
    }

    if (num == 0) {
        printf("No Bluetooth Adapters found\n");
        return 0;
    }

    for (i = 0; i < num; i++) {
         fd = _hci_open_id(devs[i]);
         if (fd < 0) {
             printf("Unable to open Bluetooth device: %d\n\n", devs[i]);
             return -1;
         }

         err = hci_get_attr(fd, &da);
         if (err < 0) {
             printf("Unable to get attribute for: %d\n", devs[i]);
             return -1;
         }
```

```
        printf("%s\t%s\n", da.name, bda2str(&da.bda));

        if (!(da.state & HCI_STATE_UP)) {
            printf("\tDevice is down");
            continue;
        }

        err = HCI_WriteScanEnable(fd, 0x02);
        if (err) {
            printf("unable to set scan mode\n");
            return -1;
        }

}
```

Next example illustrates how to discover surrounding devices.

```
int main(int argc, char *argv[])
{
   int    fd, i, argind = 1;
   __u32  length;
   int    err;
   INQUIRY_ITEM devs[20];
   char   *devnames[20];
   char   namebuf[248];
   __u8   num;

   if (argv[argind]) {
          sscanf(argv[argind], "%x", &length);
   } else
          length = 8;

   fd = hci_open("bt0");
   if (fd < 0) {
        printf("Unable to open device %s: %s\n", btdev, sys_errlist[errno]);
        return -1;
   }

   printf("Searching %d sec ...\n", length);
   err = HCI_Inquiry(fd, length, 20, devs, &num);
   print_hci_error(err);
   if (num == 0) {
       printf("done.\nNo devices found.\n");
       return 0;
   }
   printf("Searching done. Resolving names ...\n");
   for (i = 0; i < num; i++) {
        err = HCI_RemoteNameRequest(fd, &devs[i], namebuf);
        if (!err)
              devnames[i] = strdup(namebuf);
        else
              devnames[i] = NULL;
```

```
    }
    printf("done.\n");
    for (i = 0; i < num; i++) {
         printf("%d, bda: %s, name: %s\n", i, bda2str(&devs[i].bda), devnames[i]);
         if (devnames[i])
             free(devnames[i]);
    }
   return 0;
}
```

3.4. SDP API

Common feature for Bluetooth connections are that they are dynamic and thus also the services in the radio range varies. A special method is needed to discover the available services. The Service Discovery Protocol (SDP) provides means for that. It also defines how to find out the characteristics of the service.

Using SDP a client is able to discover services with specific attributes or just browse the available ones. SDP also defines unique identifiers for services, class of services and attributes. In addition to that dynamic attributes can be created for services and these need not be registered with some central authority.

In Affix the SDP API is divided into two pars: service client API and service provider API. The Service client API is dedicated for clients to find out what services are available on a remote device, what attributes they have and how to connect and properly use a service.

The Service provider API is dedicated for services. They can register information about themselves with SDP server. A SDP server has list of services that it can provide. All services have set of attributes. The attributes of a service include the type or class of service offered and the mechanism or protocol information needed to utilize the service. Each registered service has one service record at the server. This record contains all the service’s attributes.

A client sends a SDP request to the server to get information on the characteristics of a service. This requires communication between the server and client. L2CAP protocol is used to send SDP request and answers.

One Bluetooth device can have only one SDP server, but it can function both as client and as a server simultaneously. If several applications offer services those have to register their services with the SDP server. Every service provider cannot act as a SDP server.

SDP provides only information on services. It cannot be used to establish a connection to some certain service. Although available services changes, SDP cannot be used to notify clients about the new services. But, if needed, clients can poll a SDP server to get this information.

3.4.1. Universally Unique Identifier

An universally unique identifier (UUID) is guaranteed to be unique across all space and all time. UUIDs can be independently created in a distributed fashion. No central registry of assigned UUIDs is required. A UUID is a 128-bit value. UUIDs are used to uniquely identify service classes.

To reduce the burden of storing and transferring 128-bit UUID values, a range of UUID values has been preallocated for assignment to often-used, registered purposes. The first UUID in this preallocated range is known as the Bluetooth Base UUID and has the value 00000000-0000-1000-8000- 00805F9B34FB, from the Bluetooth Assigned Numbers document. UUID values in the preallocated range have aliases that are represented as 16-bit or 32-bit values. These aliases are often called 16-bit and 32-bit UUIDs, but it is important to note that each actually represents a 128-bit UUID value.

UUID object (see definition below) is used to store UUID, which can store 16, 32 or 128 bits UUID. To simplify interface to this object Affix provides set of functions to create UUID from a value and convert one type of UUID to another.

```
typedef struct {
    int type;
    union {
        uint16_t uuid16Bit;
        uint32_t uuid32Bit;
        uint128_t uuid128Bit;
    } value ;
} uuid_t;
```

Table 3-9. UUID helper functions (defined in sdpclt.h)

Function	Purpose
void sdp_val2uuid16(uuid_t *UUID, uint16_t value16Bit);	Convert a 16 bit UUID (integer) to 128 bit value (struct uuiid_t)
void sdp_val2uuid32(uuid_t *UUID, uint32_t value32Bit);	Convert a 32 bit UUID (integer) to 128 bit value (struct uuiid_t)
void sdp_val2uuid128(uuid_t *UUID, uint128_t *value128Bit);	Convert a 128 bit UUID (integer) to 128 bit value (struct uuiid_t)
int sdp_match_uuid(slist_t *searchPattern, slist_t *targetPattern);	Return 1 if each and every uuid_t in the search pattern exists in the target pattern.

Function	Purpose
int sdp_uuidcmp(uuid_t *u1, uuid_t *u2);	uuid_t comparison function that returns 0 if values match. Otherwise if uuidValue1 < uidValue2 returns -1. If uuidValue1 > uidValue2 returns 1.
int sdp_uuidcmp32(uuid_t *u1, uint32_t u2);	Same as above, but second argument is first converted to uuid_t.
int sdp_uuid2val(uuid_t *uuid);	Returns the UUID as integer. Result is always 32 bit long.
void sdp_print_uuid(uuid_t *uuid)	Prints the given UUID value.

3.4.2. A Service Record

Types of services varies a lot. A service can provide information, perform an action (like printing) or control some third resource. Implementation might be done with hardware or software or combining these.

A SDP server maintains information about it's services in a list of service records. A service record contains a set of attributes that describes the characteristics of the service. Affix defines service record like follows:

```
typedef struct {
    uint32_t serviceRecordHandle;
    BD_ADDR *bda; /* service owner */
    int      state;
    sdppdu_t pdu; /* PDU form and the entire service record */
    slist_t  *targetPattern; /* all possible uuid_t */
    slist_t  *attributeList;
    int      fd; /* to remove service */
} sdpsvc_t ;
```

Above structure is a composite of many attributes. Still not all of them need be present. References of those composites not present would be set to NULL. The service record handle identifies the service uniquely within the server. Note that identifiers in two separate servers do not match although the service might be the same.

There is one exception to that. A service record handle whose meaning is the same on all SDP servers. It's value is 0x00000000 and it represents the SDP server itself.

Table 3-10. Service record API

Function	Purpose
sdpsvc_t *sdp_create_svc(void);	Create an empty service record.

Function	Purpose
void sdp_free_svc(sdpsvc_t *svcRec);	Remove the given service record.
void sdp_free_svclist(slist_t **svcList);	Remove the given list of service records.
void sdp_gen_svc_pdu(sdpsvc_t *svcRec)	Serialize the given service record to sdppdu_t structure.
void sdp_print_svc(slist_t *svcAttrList)	Print all the attributes in the given list.
void sdp_add_uuid_to_pattern(sdpsvc_t *svcRec, uuid_t *uuid)	Add an UUID to service records attribute list. It is a sorted list where included attributes are found quickly.

3.4.3. Service Attributes

Each service attribute describes a single characteristic of a service. Universal attributes are defined in the SDP specification. Below is listed some of the universal attributes. In addition to those service providers can define their own attributes. An attribute consist of two components: service attribute ID and the value.

The attribute ID is 16 bit unsigned integer which distinguishes all the attributes within one service record. It also defines the type of the attributes value (see the following chapter). A service class defines a set of attributes that are common to that classes instances. All services belonging to a given service class assign the same meaning to each particular attribute ID.

The following table lists the most common universal attributes. Table has the definitions and a brief description of the purpose. You can find the rest of the definitions in file sdp.h.

Table 3-11. Universal service attributes

Attribute definition	Purpose
SDP_ATTR_SERVICE_RECORD_HANDLE	An unique service identifier (32 bits) valid only on the device that gave out the handle. The same service on two different devices might probably have different handles.
SDP_ATTR_SERVICEID	An universally unique service identifier (UUID) guaranteed to be the same on all devices where the service can be found.

Attribute definition	Purpose
SDP_ATTR_SERVICE_CLASSID_LIST	A list of UUIDs representing the hierarchy (type) of service classes a service is known to conform to. Usually specified in the order "most specific" to "most generic". Since the format and meanings of many attributes in a service record are dependent on the service class of the service, the ServiceClassIDList attribute is very important. Its value should be examined or verified before any class-specific attributes are used.
SDP_ATTR_PROTO_DESC_LIST	Protocol description is the information on what or which protocols can be used to gain access to this service. It is a sequence of attributes - UUID of protocol, version number and port number.
SDP_ATTR_PROFILE_DESC_LIST	List of standard "Bluetooth Profile" descriptions that the service is known to conform to. Bluetooth profile description contains UUID of profile, its version number.
SDP_ATTR_SERVICE_RECORD_STATE	A 32-bit integer provided to facilitate caching of service attributes. If this attribute is present in a service record, then it is guaranteed to change on any modification to the record.
SDP_ATTR_SERVICE_INFO_TTL	A 32-bit integer that contains the number of seconds for which the service record is expected not to change, but not a guarantee. The time interval is measured from the time the attribute is received from the SDP server. Client can use this value when judging polling frequency.
SDP_ATTR_SERVICE_AVAILABILITY	An 8 bit unsigned integer that represents the measure of service’s capability to handle more clients. 0x00 means completely unavailable and 0xff means maximum availability.
SDP_ATTR_BROWSE_GROUP_LIST	List of UUIDs representing the "browse group" the service belongs. Browse group is a hierarchy of service categories facilitating service discovery without "a priori" information of any service.
SDP_ATTR_EXEC_URL	Location of a client platform specific (Win, Unix, Palm) application that can use the service. The operating system type is contained in the body of the URL. The first byte with the value 0x2A (ASCII character *) is to be replaced by the client application with a string representing the desired operating environment.
SDP_ATTR_DOC_URL	Location of additional service documentation.

Attribute definition	Purpose
SDP_ATTR_ICON_URL	Location of an icon that can be used to represent the service.
SDP_ATTR_SERVICE_DB_STATE	An attribute specific to the service discovery server itself. If this value changes, then the service repository was recently modified.
SDP_ATTR_LANG_BASE_ATTRID_LIST	Language specific attribute identifiers for user visible strings. A base attribute ID is assigned to each of the natural languages used in the service record. When retrieving information in a specific language the attribute ID is composed of two elements: language's base attribute ID (read from this list) and information attribute ID offset (see below).
SDP_ATTR_SERVICE_NAME, SDP_ATTR_SERVICE_DESC, SDP_ATTR_PROVIDER_NAME	User visible strings conveying information about a service. These can be provided in one (universal) or more languages. These defines are rather attribute ID offsets. Use these together with base attribute ID. To get a string in specific language first get the languages base attribute ID from SDP_ATTR_LANG_BASE_ATTRID_LIST attribute and then add the attribute offset to it. Information provided is service's name, brief description of the service and the provider name.

In addition to universal attributes, a service could have "service specific attributes". These attributes make sense only in the context of the said service. Dynamic attributes could be either basic attributes or constructed.

The attribute value is a variable length field whose meaning is determined by the attribute ID associated with it and by the service class of the service record in which the attribute is contained. SDP uses a simple mechanism to describe the data contained within an attribute value.

A data element is a typed data representation. It has header and data fields. Header field in turn has also two fields: data type and size. This is called Data Type Descriptor (DTD). A data element type is represented as a 5-bit type descriptor. The type descriptor is contained in the most significant (high-order) 5 bits of the first byte of the data element header. Data types are defined in sdp.h and are in form SDP_DTD_<type>. Where <type> is the data type. For example:

```
#define SDP_DTD_UINT32 0x0A /* Defines an unsigned 32 bit integer*/
#define SDP_DTD_STR16 0x26 /* Defines a string which length is stored in a 16 bit integer *
```

SDP defines two special data types: sequence and alternative. Data element sequence, a data element

whose data field is a sequence of data elements. Alternative is a data element whose data field is a sequence of data elements from which one data element is to be selected.

The data element size descriptor is represented as a 3-bit size index followed by 0, 8, 16, or 32 bits. The size index is contained in the least significant (low order) 3 bits of the first byte of the data element header.

3.4.4. Attribute API

The generic data structure *sdpdata* is meant to hold any attribute, universal or dynamic. Each *sdpdata* structure contains one attribute. The *sdpdata* object contains the service attribute identifier, a Data Type Descriptor followed by the value. Protocol Data Unit (PDU) is a serialized presentation of the object.

```
typedef struct sdpdata {
    uint8_t dtd;
    uint16_t attrId;
    union {
        int8_t    int8;
        int16_t   int16;
        int32_t   int32;
        int64_t   int64;
        uint128_t int128;
        uint8_t   uint8;
        uint16_t  uint16;
        uint32_t  uint32;
        uint64_t  uint64;
        uint128_t uint128;
        uuid_t    uuid;
        char      *stringPtr;
        slist_t   *dataSeq;
    } value;
    sdppdu_t pdu;
    int unitSize;
} sdpdata_t;
```

Affix has set of functions that can be used to create and manipulate attributes.

Table 3-12. Service Attribute API

Function	Purpose
sdpdata_t *sdp_create_data(uint8_t dtd, void *pValue);	Creates a SDP attribute object of certain type.

Function	Purpose
void sdp_free_data(sdpdata_t *pData);	Destroys entire attribute object.
int sdp_attrcmp(const void *key1, const void *key2)	Compares if given two attributes are the same. Returns zero if they are the same.
void sdp_print_attr(void * value, void * userData)	Print attribute type and value. XXX userData is not used. Why this parameter is still here?
sdpdata_t *sdp_put_<type>(<type> val);	Family of functions to create a new attribute structures and puts the given value of type <type> in the new object. These functions are defined in sdpclt.h. Return a pointer to the new structure.
sdpdata_t *sdp_append<type>(sdpdata_t *seq, <type> val);	Family of functions to use with sequence type data. This adds a value of type <type> to the given sequence. These functions are defined in sdpclt.h. Return a pointer to the new structure.
<type> sdp_get_<type>(sdpdata_t *data);	Family of functions to get data value of <type> from an attribute.
sdpdata_t *sdp_create_seq(void)	Create new attribute with one value of type sequence.
sdpdata_t *sdp_append_seq(sdpdata_t *seq)	Append a new sequence type data element to attribute.
slist_t *sdp_get_seq(sdpdata_t *data);	Get the sequence type data in a linked list. Each list node presents one data item in the sequence.
sdpdata_t *sdp_create_alt(void)	Create new attribute with one value of type alternative.
sdpdata_t *sdp_append_alt(sdpdata_t *seq)	Append a new alternative type data element to attribute.

Service attributes are stored in service records. Affix has functions to add, remove and get attributes from a service record. These functions are the glue between these two structures.

Table 3-13. Functions to glue service attributes and records

Function	Purpose
sdpdata_t *sdp_append_attr(sdpsvc_t *svcRec, uint16_t attrId, sdpdata_t *data)	Adds attribute object to the attribute list of ServiceRecord object.
void sdp_remove_attr(sdpsvc_t *svcRec, uint16_t attrId)	Removes an attribute object from a service record.
sdpdata_t *sdp_get_attr(sdpsvc_t *svcRec, uint16_t attrId);	Get a pointer to the value of an attribute in a service record.
char *sdp_get_string_attr(sdpsvc_t *svcRec, uint16_t attrID)	Get the string value of given attribute. Note! Attribute have to be string type. No type conversion is done.

3.4.5. Service Client SDP API

The client SDP API is used by a client to find services provided by a remote device and to find out what attributes they have. The overall function set is shown on the following table:

Table 3-14. Service client SDP API

Function	Purpose
int sdp_init(int flags)	Initializes the SDP infrastructure. Flag is whether SDP_SERVER or SDP_CLIENT.
void sdp_cleanup(void)	Removes all data structures needed by SDP.
int sdp_connect(struct sockaddr_affix *saddr)	Opens a connection to the SDP server.
void sdp_close(int srvHandle)	Closes a connection to the SDP server.
int sdp_search_req(int srvHandle, slist_t *svcSearchList, uint16_t maxSvcRecordCount, slist_t **svcResponseList, uint16_t *handleCountInResponse)	Make a service search request. Returns E_OK on success and E_FAILURE on an error. Parameter svcSearchlist is a singly linked list containing elements of the search pattern. Each entry in the list is a uuid_t (DataType SDP_DTD_UUID16) of the service to be searched. maxSvcRecordCount sets the maximum number of entries to return. After returning svcResponseList points to the list of found services and handleCountInResponse has the number of services found.
int sdp_attr_req(int srvHandle, uint32_t svcHandle, sdp_attrreq_t attrReqType, slist_t *attrIDList, uint16_t maxAttrIDByteCount, sdpsvc_t **_svcRec, uint16_t *maxAttrResponseByteCount)	Make a request for a service's attributes. Parameter svcHandle has the handle for the requested service on a server which handle is srvHandle. attrReqType specifies in which form the attribute IDs are given and attrIDList has the actual list of attributes to request. maxAttrIDByteCount contains the maximum length of the response. Return E_OK on success and maxAttrResponseByteCount points to an integer that has the number of bytes of attributes returned.
int sdp_search_attr_req(int srvHandle, slist_t *svcSearchList, sdp_attrreq_t attrReqType, slist_t *attrIDList, uint16_t maxAttrByteCount, slist_t **svcResponseList, uint16_t *maxAttrResponseByteCount)	A service search request combined with the service attribute request. First a service class match is done and for matching service, requested attributes are extracted. Parameters and return values are as in the two functions above.
int sdp_get_<attr_name>_attr(sdpsvc_t *svcRec, <attr_type>)	Template of functions: extracts individual attributes from service record, e.g.: service name, access port, profile id. <attr_name> in the function name body specifies the received attribute. Second parameter is a pointer of the type of the attribute. It will point to the attributes value after return. See function names for different attributes in sdpclt.h.

3.4.5.1. Connecting and Disconnecting from SDP Server

First of all the SDP infrastructure must be initialized using sdp_init(SDP_CLIENT) call.

Before any request to the SDP server can be made, a client has to make a connection to the SDP server using following function: int sdp_connect(struct sockaddr_affix *saddr). This function accepts one parameter - address of the server and returns SDP server handle, that is needed later with several functions. An Address is given using Affix *sockaddr* structure.

To close a connection, call void sdp_close(int srvHandle) and give the handle got when connecting as a parameter.

3.4.5.2. Service Class Identifier(s) Based Search

Service search on the basis of given search pattern consisting of a set of service class identifiers. The maximum number of service class identifiers in the pattern is 12. Use function sdp_search_req() making the request.

- The search pattern (svcSearchList) is a list of UUIDs (the UUID objects can either be 16, 32 or 128 bits). The UUID objects can be created using the helper functions described in chapter Section 3.4.1.
- The client can limit the number of service record handles it expects from the server, and this is specified in maxSvcRecordCount. If there is no limit, set this to a large value.

The server response to this request is a list of matching service record handles (if any), and this is set up in svcResponseList (a slist_t) and returned to the caller. The number of handles found is set in handleCountInResponse.

3.4.5.3. Service Attributes Request from a Specific Service Record

Search for a specified set of service attributes in a specific service. This request usually follows a previous service class identifier based search. Once a service has been identified, this request is used to find more about the service. Brief descriptions of the arguments follow:

- Service record handle of the service need to be provided to uniquely identify the service which attributes we would like to request.
- The API supports requesting the entire range of attributes or a certain subset of them. This is specified in the request type (parameter attrReqType), followed by the actual list of attributes in *attrIdList* (type *slist_t*). Attribute identifiers are 16 bit unsigned integers specified in one of 2 ways. If type is *IndividualAttributes* - 16bit individual identifiers actual attribute identifiers in ascending order. If type is *RangeOfAttributes* - 32bit identifier range; the high-order 16bits is the start of range, the low-order 16bits are the end of range; 0x0000 to 0xFFFF gets all attributes.

- maxAttrIDByteCount is the maximum byte count that the client expects to receive. The server will never return a response containing attribute byte count greater than this value.
- maxAttrResponseByteCount - this is a pointer to a 16-bit integer, which is set to indicate the number of bytes of attributes returned.
- _svcRec - this is a pointer to pointer to service record that contains the requested attributes. There are lot’s of handy functions to extract information from the service record. See the earlier chapters.
- A successful execution of the command results in E_OK being returned else a negative value indicating the type of error (timeout, invalid arguments etc) is returned.

3.4.5.4. Service Class Identifier(s) and Service Attribute(s) Based Search

This request combines the service class identifier and the service attribute search request, thus minimizing the number of request/response pairs needed to find a service. Use function sdp_search_attr_req() to make this request. The parameters contain a service search pattern and a list of attributes that need to be fetched, should a service record match the service search pattern.

First a service class match is done and for matching service, requested attributes are extracted. See the previous two chapters for explanation on the parameters

As before the method returns E_OK on successful execution of request and an error code for failures.

3.4.6. Service Provider SDP API

Bluetooth service developers can use the following API to register services with the "local" SDP server. The "on-the-wire" formats of service registrations mimic the SDP. However a key difference is that service registrations Protocol data units (PDU) are never sent on Bluetooth channels, but done using TCP sockets. The service registration requests and responses use the "reserved range" of SDP PDU Identifiers. There is no problem since it is not sent on BT channels. The following are the APIs available for service implementers to register their service with the SDP server on the device.

Table 3-15. Service provider SDP API

Function	Purpose
int sdp_init(int flags)	Initializes the SDP infrastructure. Flag is whether SDP_SERVER or SDP_CLIENT.
int sdp_connect_local(void)	Connect to a local SDP server and start it if it is not running. Returns a server handle that is needed with other functions.
void sdp_close(int srvHandle)	Closes a connection to the SDP server.

Function	Purpose
sdpsvc_t *sdp_create_rfcomm_svc(uint16_t svc_class, uint16_t generic_class, uint16_t profile, char *name, char *prov, char *desc, int port)	Creates a Service Record object and set common RFCOMM attributes.
sdpsvc_t *sdp_create_pan_svc(uint16_t svcClass, slist_t *ptype, uint16_t security, uint16_t type, uint32_t rate)	Creates a Service Record object and set common PAN attributes.
int sdp_register_service(int srvHandle, sdpsvc_t *svcRec)	Registers a Service Record object with the SDP server.
int sdp_update_service(int srvHandle, sdpsvc_t *svcRec)	Updates a Service Record object on the SDP server.
int sdp_delete_service(int srvHandle, sdpsvc_t *svcRec)	Deletes a Service Record object a service record pointed to by svcRec from SDP server.
int sdp_delete_service_handle(int srvHandle, uint32_t svcRecHandle);	Delete a service record which service handle is svcRecHandle.

3.4.6.1. Managing Service Records

Before registering or updating a service record at the server you have to fill out the service record structure. Several helper functions exists to help your work. Couple of general functions were covered in chapter Section 3.4.2. Still most of the functions are defined in sdpsrv.h and will be discussed here.

Note also that if you are coding a service provider only manipulating service record is not enough. You have to call sdp_update_service() or sdp_register_service() function the changes to have an effect on the server. Further note that the pointers passed to helper functions must not be deleted for the life-time of the service as these are stored. Copies are not made of attributes pointed to by these pointers.

The following table presents the functions you can use to manipulate service records.

Table 3-16. Service record manipulation functions

Function	Purpose
int sdp_set_info_attr(sdpsvc_t *svcRec, char *serviceName, char *providerName, char *serviceDescription)	Set the service name, description and provider attributes.
sdpdata_t *sdp_set_class_attr(sdpsvc_t *svc)	Add a service class attribute to a service record and return a pointer to it.
sdpdata_t *sdp_set_proto_attr(sdpsvc_t *svc)	Add a protocol list attribute to a service record and return a pointer to it.

Function	Purpose
sdpdata_t *sdp_set_subgroup_attr(sdpsvc_t *svc)	Add a browse group attribute to a service record and return a pointer to it.
sdpdata_t *sdp_set_lang_attr(sdpsvc_t *svcRec)	Add a language base attribute list to a service record and return a pointer to it.
sdpdata_t *sdp_set_state_attr(sdpsvc_t *svcRec, uint32_t svcRecState)	Add service record state attribute and set its value to svcRecState. Return a pointer to modified record.
sdpdata_t *sdp_set_service_attr(sdpsvc_t *svcRec, uint16_t svcUUID)	Add service ID attribute and set its value to svcUUID. Return a pointer to modified record.
sdpdata_t *sdp_set_group_attr(sdpsvc_t *svcRec, uint16_t groupUUID)	Add group ID attribute and set its value to groupUUID. Return a pointer to modified record.
sdpdata_t *sdp_set_availability_attr(sdpsvc_t *svcRec, uint8_t svcAvail)	Add availability attribute and set its value to svcAvail. Return a pointer to modified record.
sdpdata_t *sdp_set_profile_attr(sdpsvc_t *svc)	Add a profile description list attribute to a service record and return a pointer to it.
int sdp_set_url_attr(sdpsvc_t *svcRec, char *clientExecURL, char *docURL, char *iconURL)	Set executable, icon and documentation URL attributes. Returns zero on success and -1 on an error.

SDP server API provides also some functions to manage service attributes. See the following table.

Table 3-17. SDP server API for attribute manipulation

Function	Purpose
sdpdata_t *sdp_add_class(sdpdata_t *attr, uint16_t uuid)	Add class ID value to service attribute structure.
sdpdata_t *sdp_add_subgroup(sdpdata_t *attr, uint16_t uuid)	Add group ID value to service attribute structure.
sdpdata_t *sdp_add_proto_list(sdpdata_t *attr)	Add protocol description list to attribute structure.
sdpdata_t *sdp_add_proto(sdpdata_t *attr, uint16_t uuid, uint16_t portNumber, int portSize, uint16_t version)	Add a protocol description to attribute structure.
int sdp_add_lang(sdpdata_t *attr, uint16_t lang, uint16_t encoding, uint16_t offset)	Add a language base information to attribute structure.

In the following example code first a service class attribute is added to the service record. Then the value is set to SDP_UUID_SDP_SERVER. A service record has also a match list of service classes it supports. When user looks for a certain service he sends a list of classes that specify the service. This list will be matched with the class list in the service record. This is why we have to add the class also to that list.

```
attr = sdp_set_class_attr(sdpSvcRec);
```

```
sdp_add_class(attr, SDP_UUID_SDP_SERVER);
sdp_add_uuid16_to_pattern(sdpSvcRec, SDP_UUID_SDP_SERVER);
```

3.5. OBEX API

The Affix provides a flexible API to implement OBEX server and OBEX client applications. The OBEX API is very simple. Any object is stored in a file before/after transfer. The Affix OBEX API provides easy way to deal with it.

3.5.1. Data Structures

All data structure definitions are located in the <affix/obex.h> header file and they include: obexclt_t, obexsrv_t, obex_file_t.

3.5.2. OBEX File API

The main concept of the Affix OBEX API is that before object can be transferred it must be stored in a file. Affix has a following function set for that:

Table 3-18. OBEX file API

Function	Purpose
obex_file_t *obex_create_file(char *name);	Creates file object from a file "name". If "name" is NULL then it creates a file with randomly generated name.
void obex_close_file(obex_file_t *file);	Closes file.
int obex_open_file(obex_file_t *file);	Reopens a file referenced by the file object "file".
void obex_destroy_file(obex_file_t *file, int del);	Closes file and destroys file object. If "del" is not zero then file is removed.
char *obex_map_file(obex_file_t *file);	Maps file referenced by file object "file" to memory location and returns pointer to that.

3.5.3. Client API

The OBEX client API includes two sets of functions: general purpose and file transfer oriented.

Table 3-19. OBEX client API

Function	Purpose
obexclt_t *obex_connect(struct sockaddr_affix *addr, obex_target_t *target, int *err);	Opens connection to OBEX server. Accepts sockaddr_affix address and target. Reruns OBEX client object.
int obex_disconnect(obexclt_t *clt);	Closes connection to OBEX server.
int obex_get(obexclt_t *clt, char *local, char *remote, char *type);	Receives object from the server. Target object specified by "remote" and "type" arguments. It is stored in the file "local".
int obex_put(obexclt_t *clt, char *local, char *remote, char *type);	Sends object from a file "local" to the server. Target object identified by "remote" and "type".
obexclt_t *obex_connect_file(struct sockaddr_affix *addr, int *err);	Opens connection to FTP service on the server.
int obex_get_file(obexclt_t *clt, char *local, char *remote);	Receives file "remote" from the server and stores it to the file "local". If "local" is NULL then file is stored with the name "remote" to the current working directory.
int obex_put_file(obexclt_t *clt, char *local, char *remote);	Sends file "local" to the server and stores it in the file "remote". If "remote" is NULL then file is stored with the name "local" to the server current working directory.
int obex_browse(obexclt_t *clt, char *local, char *name);	Gets "name" folder listing from the server and stores it in the file local.
int obex_setpath(obexclt_t *clt, char *path);	Changes working directory on the server to "path".
int obex_mkdir(obexclt_t *clt, char *path);	Creates new directory with path "path" on the server.

3.5.4. Server API

The Affix provides a very simple API to implement OBEX servers also. OBEX server works in request-response way: waits for a request from client, processes it and sends response back.

The main idea behind API is "call-back" function. OBEX server exports certain functions, which are called by the Affix OBEX library when certain request arrives. Server implementation has to create a server object and initialize callback function pointers (see bellow) and to call btsrv_run() to start request processing loop.

Table 3-20. OBEX server API

Function	Purpose
int obexsrv_run(obexsrv_t *srv, int rfd, int wfd);	Starts server’s main request processing loop. It reads request from the endpoint referenced by the file descriptor "rfd" and writes responses to the endpoint referenced by the file descriptor "wfd".

Function	Purpose
int (*connect)(obexsrv_t *srv, obex_target_t *target);	Function is called on connection request. "target" points to "connection target".
void (*disconnect)(obexsrv_t *srv);	Function is called on disconnection.
int (*put)(obexsrv_t *srv, char *file, char *name, char *type, int flags);	Function is called on PUT request. File "file" contains object. The object identified by "name" and "type" on the server.
int (*get)(obexsrv_t *srv, char *name, char *type);	Function is called on GET request. An Object identified by "name" and "type".
int (*setpath)(obexsrv_t *srv, char *path, int flags);	Function is called on SETPATH request. "path" points to path to set.
void obexsrv_set_file(obexsrv_t *srv, char *name, int del);	Sets a file to be transferred on GET request. (usually is used in get() callback function)

3.6. PAN API

Personal Area Network consist of one GN (Group Network) or NAP (Network access Point) and 1 to 7 PANUs (Personal Area Network User). These form a piconet that uses Bluetooth for communicating between hosts. GN or NAP performs network bridge functions using Bluetooth Network Encapsulation Protocol (BNEP) to transfer Ethernet packets over Bluetooth connection.

To participate in piconet a host’s Bluetooth device has to be initialized. During the initialization the host’s role is set. After the initialization is completed the host can connect to another host or wait others to connect to it. Usually PANUs connect to GNs or NAPs.

Ethernet packets are transmitted as a single L2CAP packet. They are transmitted in the payload using BNEP. According to PAN profile specification payload can be encrypted (profile secrecy mode 2), but Affix does not support this yet.

Creating a PAN connection with Affix is very straightforward and the API is simple. See the example below.

Table 3-21. Personal area network API

Function	Purpose
int affix_pan_init(char *name, int mode);	Initialize the given Bluetooth device for PAN connection. Mode is one of AFFIX_PAN_PANU, AFFIX_PAN_NAP, AFFIX_NAP_GN, AFFIX_PAN_AUTO.

Function	Purpose
int affix_pan_connect(struct sockaddr_affix *sa, int role);	Connect to a remote host. PAN have to be initialized before calling this. Role parameter defines peer's role. It is one of the following: AFFIX_PAN_NAP, AFFIX_NAP_GN. This function is also used to disconnecting the device. When disconnecting set remote address to BDADDR_ANY.

The following simple example illustrates creating and closing a PAN connection.

```
char btdev[IFNAMSIZ] = "bt0";
char remote_host[] = "00:0c:76:b1:03:2d";
int  err, role = AFFIX_PAN_NAP;
BD_ADDR bda;
struct sockaddr_affix saddr;

/* Initialize this host as an user */
err = affix_pan_init(btdev, AFFIX_PAN_PANU);
if (err) {
    BTERROR("Error while initalizing PAN\n");
    fprintf(stderr, "%s\n", hci_error(err));
    exit(1);
}

/* PAN initialized. Now, try connecting */
/* Parse the string presentation of the address */
err = btdev_get_bda(&bda, remote_host);
if (err) {
    fprintf(stderr, "Incorrect address\n");
    return 1;
}

saddr.family = PF_AFFIX;
saddr.bda = bda;
saddr.devnum = hci_devnum(btdev);

err = affix_pan_connect(&saddr, role);
if (err) {
    if (errno == EINVAL)
        fprintf(stderr, "This role is not valid.\n");
    else
        fprintf(stderr, "Creating connection failed!\n");
        exit(1);
} else
    fprintf(stderr, "Connection established. \n");


/* Now the connections is established. In this example we
are going to close it now*/
```

```
saddr.bda = BDADDR_ANY; /* BDADDR_ANY means disconnect */
err = affix_pan_connect(&saddr, 0);
if (err) {
    fprintf(stderr, "Closing the connection failed\n");
    exit(1);
} else
    fprintf(stderr, "Connection closed.\n");
```

3.6.1. Traffic Filtering

As piconets bandwidth is limited it is often good idea to filter out some unnecessary traffic. Affix implements this filtering to reduce wasted bandwidth. For example video transmission and IPv6 packets may be dropped at NAP.

Look at http://www.iana.org/assignments/ethernet-numbers for protocol types. Multicast addresses for IPv4 are listed at http://www.iana.org/assignments/multicast-addresses. For IPv6 see http://www.iana.org/assignments/ipv6-multicast-addresses and http://www.rfc-archive.org/getrfc.php?rfc=2373.

3.6.2. Data Structures

You need to fill specific data structures when setting filters. Both protocol and multicasting filters has their own structures. Multicast filter structure is defined as follows.

```
typedef struct {
    __u16      count; /* number of filter entrys, 0 = no filter */
    ETH_ADDR   multicast[MULTICAST_FILTER_MAX][2] /* start_range -
               stop_range pairs in network byte order */
} multicast_filter;
```

Protocol filter definition is:

```
typedef struct {
    __u16   count; /* number of filter entrys, 0 = no filter */
    __u16   protocol[PROTOCOL_FILTER_MAX][2]; /* start_range -
            stop_range pairs in network byte order */
} protocol_filter;
```

Maximum amount of filter ranges are defined with MULTICAST_FILTER_MAX and PROTOCOL_FILTER_MAX.

3.6.3. Function API

You should use ioctl() function call to set filters. The function call interface is

```
ioctl(file_descriptor, type, value);
```

First you have to open a socket to set or get the filter values. The type parameter defines the type of setting that is set or got. See the table below for filter settings. The value is a pointer to *ifreq* struct. When setting a filter it holds the filter information. When getting the filter information the structure pointed by this is filled.

Table 3-22. Filter types for ioctl() function

Define	Purpose
SIOCSFILTERPROTOCOL	Set protocol filter
SIOCSFILTERMULTICAST	Set multicast filter
SIOCGFILTERPROTOCOL	Get protocol filter
SIOCGFILTERMULTICAST	Get multicast filter

Follows example of setting protocol filter.

```
struct ifreq ifr;
protocol_filter pf;
int fd;
char interface[] = "bt0";

/* Create socket */
fd = socket(PF_INET, SOCK_DGRAM, IPPROTO_IP);
if (fd < 0) {
    fprintf(stderr, "error opening socket");
    return -1;
}

memset(&pf, 0, sizeof(pf));
strncpy(ifr.ifr_name, interface, IFNAMSIZ);

/* Set filter information. Allow only IPv4 traffic. */
pf.count = 1;
pf.protocol[0][F_START] = 0x800;
pf.protocol[0][F_STOP] = 0x800;

/* Set the filter */
```

```
ifr.ifr_ifru.ifru_data = (char*) &pf;
if ((result = ioctl(fd, SIOCSFILTERPROTOCOL, &ifr))) {
    fprintf(stderr, "ioctl error when setting protocol filter ");
}
```

3.7. Human Interface Device API

The Bluetooth Human Interface Device (HID) profile facilitates the use of Bluetooth as a bearer for wireless human interface peripheral devices. The HID device class comprises a large set of devices, such as keyboards, mice, pointer, remote control, etc. That can make use of Bluetooth to provide user input to a host device. HID is a subclass of USB devices. The Bluetooth HID profile fills the gap opened by USB and enables the use of Bluetooth as a wireless bearer for devices conforming to the HID device class specification. This chapter discuss about Affix HID interface.

3.7.1. Basic Architecture

The topology consists of simple wireless point-to-point links between a host device and HID device. In most cases, the host will be master of a Bluetooth piconet with the HID devices connected to as slave devices. However, for dedicated scenarios it is possible for the HID device to take over the role as Bluetooth master. This holds for situations in which the HID device only generates HID reports sporadically, which means that regularly polling the HID device would waste the possibly limited power resources on the device.

Another scenario in which it makes sense that the HID device becomes Bluetooth master is the case the HID device has to generate input for more than one device. Acting as a Bluetooth master the device is able to join all related devices in one piconet and therefore, no scatternet functionality or forwarding of L2CAP data must be supported or used.

For the common case that the host is piconet master, connection establishment is typically initialized by the host. A HID device can be temporarily brought into limited discoverable mode, in which the device answers to a host’s inquiry procedure. Once discovered, the host sets up a connection in a page procedure, binding the HID device into its piconet. However, the HID profile also supports the possibility of a HID device to reinitiate a connection, which can be beneficial in case of a host reboot that led to the loss of HID device connection state at the host.

3.7.2. HID Profile

The HID profile species a new protocol on top of L2CAP that encapsulates the HID reports and

facilitates other control communication between the HID device and driver for the HID device class on the host. HIDP maintains two L2CAP connections between host and HID device - a control channel, and an interrupt channel. This separation directly emerges from the original HID specification.

The control channel is used to signal status information, to set the power saving and operational states of the HID device, and to query the resource records of a device. The interrupt channel is a data channel, over which the host expects input reports from the HID device, or transmits output reports towards the HID device. As the L2CAP MTU may not suffice to carry complex HID record descriptors, HIDP also supports segmentation and reassembly.

3.7.3. Affix HID API

The kernel-space module affix_hidp has two main tasks. Firstly, it has to connect to the HID device, manage the connection with the device and care for the transmission of the HID reports. Secondly, the kernel module has to take these HID reports, parse them, and put the extracted input events into the input subsystem of the Linux kernel. The two tasks are realized in two different files affix_hid.c and hid.c. However, these files are compiled into one module.

The kernel module is controlled via the *ioctl* interface. The kernel module implements a set of commands that can be accessed by invoking the *ioctl* system call on an Affix HCI socket. The interface between kernel-space module and user-space utility contains three *ioctl* calls. The structures that are given as data by the argument parameter are defined in affix-kernel/include/bluetooth.h. The following *ioctl* commands are specified:

3.7.3.1. BTIOC_HIDP_MODIFY

The BTIOC_HIDP_MODIFY call is used when a device should be added to the kernel list or a connection to a device should be established. If the device is already known, the existing list entry will be modified. If it is not known, a new list entry will be added. The *ioctl* call expects a pointer to a *hidp_ioc* structure. The structure contains various information fields obtained during the SDP query and is defined as follows:

```
struct hidp_conn_info {
    __u16 parser;
    __u16 rd_size;
    __u8 *rd_data;
    __u8 country;
    __u16 vendor;
    __u16 product;
    __u16 version;
    __u32 flags;
    __u32 idle_to;
    char name[128];
};
```

```
struct hidp_ioc {
    struct sockaddr_affix saddr;
    struct hidp_conn_info conn_info;
    __u16 status;
    __u16 cnum;
};
```

The struct contains the Bluetooth device address, information that have been received by the SDP request , a status field, and a connection number. The status field is a bit mask comprising the flags listed in table below.

Table 3-23. Values of the status bit mask in the hidp_ioc struct

Flag	Value
HIDP_STATUS_ACTIVE_ADD	0x01
HIDP_STATUS_CONNECTED	0x02

The ACTIVE_ADD flag is used for connection initiation. Many HID devices reopen a connection themselves e.g. When a button is hit after a long idle time. For some others the connection has to be initiated by the peer. If this flag is set, the host will actively open a connection to the HID device. Otherwise, it will wait for connection initiation of the HID device.

The CONNECTED flag is used for returning the status of the connection to a HID device known to the kernel module. If set, the device is currently connected, if not, the device is known, but currently not connected.

3.7.3.2. BTIOC_HIDP_DELETE

The BTIOC_HIDP_DELETE command deletes the device from the kernel's list of known HID devices. Thus, a connection is not possible anymore. The only way to reconnect is adding the device again to the list. As with the BTIOC_HIDP_MODIFY command, a pointer to a hidp_ioc struct is expected. In case there is a connection when this call is performed, the connection with the device is closed before.

3.7.3.3. BTIOC_HIDP_GET_LIST

The BTIOC_HIDP_GET_LIST command returns a list of all devices to user-space. It expects a pointer to a hidp_ioc_getlist struct which contains an array of hidp_ioc records to be filled out by the kernel. The number of available hidp_ioc entries must be given in count which returns the number of entries written upon completion. In case the supplied array is not large enough to cover all HID devices, the left field contains the number of devices that do not fit.

The struct returned as argument has the following format:

```
struct hidp_ioc_getlist {
    int count; /* # of devices in list */
    int left; /* != 0 if more than count devices in list */
    struct hidp_ioc list[0]; /* first element in list */
};
```

3.8. Device Cache

Using inquiry to find Bluetooth devices around your host takes usually at least several seconds. To make accessing the remote Bluetooth equipment more efficient and simpler for the end user Affix implements a specific device cache. This is cache stores information about Bluetooth devices that are found around you during the device inquiry. Note that also devices that are not currently in the radio range are preserved in the device cache until it is flushed or they are explicitly removed.

The end users get the list of cached devices fast and it is also more convenient to refer to devices cache entry number that all the time write the whole Bluetooth address. A single device is presented in the following data structure.

```
typedef struct {
    int flags;
    int state; /* Flag indicating is the device in the range. */

    BD_ADDR bda; /* Address of the device */
    uint32_t cod;
    char name[248];

    __u8 key_type;
    __u8 link_key[16];
    __u8 pin_length;
    __u8 pin[16];
} btdev_struct;
```

The devices in the cache are stored as a linked list where each element contains one device in a structure described above. In addition to device entries cache structure contains the mount of the devices in the list, flag for locking and the file name where the cache will be stored etc. See the definition below.

```
typedef struct {
    slist_t *head;
    int count;
    char *file;
    int lock;
    time_t stamp;
} btdev_list;
```

In the following table are the functions that are needed to access device cache. All function definitions are in btcore.h file.

Table 3-24. Device cache API

Function	Purpose
btdev_struct *btdev_cache_lookup(BD_ADDR *bda);	Get device information of the device which address is given as a parameter. Returns btdev_struct or null if the address is not found in the cache.
int btdev_cache_del(btdev_struct *entry);	Delete a device from the list of cached devices. Always returns zero. Device to remove is identified by a pointer to it's btdev_structure.
btdev_struct *btdev_cache_add(BD_ADDR *bda);	Initialize a new btdev_structure and insert it to the device cache. Set the given address for the device. Returns a pointer to the new structure.
int btdev_cache_load(int locked);	Load cache information from the disk. The file name is stored in btdev_list structure's file member (see above). The member is initialized in btdev_cache_init() function. If *locked* parameter is non zero the function leaves cache locked after it exits. On error a value below zero is returned. Returning zero indicates success.
int btdev_cache_reload(void);	The same as btdev_cache_load, but leaves cache locked by default.
void btdev_cache_purge(void);	Flush device cache. Note! Also the cache file on the disk is emptied.
int btdev_cache_save(void);	Stores the whole device cache into a file on a disk. The file name is stored in btdev_list structure's file member (see above). Function returns zero on success and -1 on error.
void btdev_cache_free(void);	Removes device cache and frees all memory allocated for it's structures.
int btdev_cache_init(void);	Initializes device cache. The storage file's name is copied from global variable affix_cachefile that is defined in btcore.c.
void btdev_cache_retire(void);	Retire cache by marking all devices out of the range. Set their state to DEVSTATE_GONE
void btdev_cache_print(int state);	Print all the devices with a specific state in the cache to standard output. State is given as a parameter for the function. This way you can print first devices that are reachable in the radio range.

Function	Purpose
int btdev_cache_resolve(BD_ADDR *bda, int id);	Resolve the device address according to the cache entry number. Address is stored in a structure pointed by bda. Memory allocation is left for the caller. Parameter id holds the cache index. Returns zero if resolving was OK or -1 on error.
int btdev_get_bda(BD_ADDR *bda, char *arg);	Converts string presentation of a Bluetooth device address to binary form. Also seeks for the address from the device cache if a cache entry index was given instead of device address. Parameter *bda* is a pointer to structure where the resolved address is stored. Parameter arg is the address or the index number in a string form. Return zero if resolving the address was OK.
int btdev_cache_lock(void);	Lock device cache.
void btdev_cache_unlock(void);	Unlock device cache.
int btdev_cache_init(void)	Initialize device cache. Make sure you have freed existing cache before calling this. This is done with btdev_cache_free() function.

3.9. Debugging Affix

Affix uses system log for printing debug messages and notifying for errors and warnings. To read messages give the following command as a root:

```
tail -f /var/log/syslog
```

The granularity of debug messages can be set with **btctl** command. The format is following:

```
btctl debug [+|-][<module>|<detail>|all]
```

Use plus sign to turn messages for specified module on. Special name "all" allows all messages to be shown. This results typically flood of messages and should be used carefully.

Table 3-25. List of possible modules and details

DBHCI	DBL2CAP	DBALLMOD
DBAFHCI	DBAFL2CAP	DBCTRL
DBHCIMGR	DBRFCOMM	DBPARSE
DBHCISCHED	DBAFRFCOMM	DBCHARDUMP
DBHCI	DBBTY	DBHEXDUMP
DBAFHCI	DBRFCOMM	DBFNAME

DBHCIMGR	DBAFRFCOMM	DBFUNC
DBHCISCHED	DBBTY	DBALLDETAIL
DBL2CAP	DBPAN	
DBAFL2CAP	DBDRV	

3.9.1. Capturing Traffic

You can also capture network packets transmitted through a Bluetooth device. This allows you to debug data transmitted over the Bluetooth line. Using this method is fairly simple. First all the traffic is captured to a file you specify. Use the following command to start capturing:

```
btctl capture file_name
```

When enough data is transmitted stop capturing pressing ctrl-c. Now you can use nice ethereal tool to make analyzing the traffic easier. Give command:

```
ethereal capture_file
```

3.9.2. Kernel Variables

Affix has several kernel variables in /proc/net/affix and /proc/sys/net/affix directories. Use these to get information of the state of Affix and your Bluetooth devices. You can also use some variables to control Affix functioning.

Affix 3.1.0 adds support for L2CAP 1.2. Support includes two new protocols L2CAP in retransmission and flow control modes. These protocols are also controlled using kernel variables. Six new variables were introduced. Those are located under /proc/sys/net/affix.

Table 3-26. L2CAP 1.2 retransmission and flow control kernel variables

Variable	Usage
l2cap_max_transmit	Sets the maximum number of times that a packet can be retransmitted. (Range from 1 to 255).
l2cap_mon_timeout	Sets the monitor timeout time (in milliseconds) (Range 100 - 65535)
l2cap_mps	Sets the Maximum payload size (max size of packet that l2cap can deal with) (Range MTU - 65535). (MTU is the Maximum transmission unit)
l2cap_ret_timeout	Sets the retransmission timeout time (in milliseconds) (Range 400 - 65535)

Variable	Usage
l2cap_supported_modes	This variable sets the value of the extended features supported by Affix. (Default: 7) The values are 4 bit value. Where: bit 0: Flow control mode (set 1 if feature is supported), bit 1: Retransmission mode (set 1 if feature is supported), bit 2: QoS (set 1 if feature is supported), bits 3-31 are set 0.
l2cap_txwindow	Sets the size of the transmission window for flow control and retransmission mode. (Range 1 - 32)

The status of each HID device can be read from /proc/net/affix/hidp. Each line contains two tab-delimited fields where the first field is a Bluetooth device address and the second field represents the status of the HID device (connected/not connected).

References

[BTSPEC] *Bluetooth Specification Book*, Bluetooth SIG.

[BTPROF] *Bluetooth Profile Book*, Bluetooth SIG.

[RUSL99] *The Linux Kernel*, Devid Rusling, 1999.

[STEV97] *"UNIX Networking Programming, Volume 1: Networking APIs - Sockets and XTI"*, 1009 pages, Prentice Hall, October 17, 1997.