Affix in a Nutshell

Dmitry Kasatkin

Affix in a Nutshell

by Dmitry Kasatkin

Table of Contents

List of Tables

List of Figures

Preface

This document describes Affix^{tm} Bluetooth Protocol stack for Linux developed in Nokia Research Center at Mobile Networks Lab and released under GPL.

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

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. 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. Acknowledgements

First of all, I would like to thank Nokia Research Center and Mobile Network Laboratory for funding a project and Jari Juopperi and Asko Vilavaara for supporting and promoting it.

Also I would like to thank all these people helping to develop, test, maintain Affix:

Jakub Pavelek <jakub.pavelek@nokia.com>
- Major Affix Tester
- iPaq & Zaurus
- Affix Frontend Environment developer

Imre Deak <ext-imre.deak@nokia.com>
- USB Driver. Profiles. Testing. Fixes.

Guruprasad Krishnamurthy <kgprasad@hotmail.com>
- Original SDP implementation.

Manel Guerrero Zapata <manel.guerrero-zapata@nokia.com>
- Some coding and fixes in the SDP client.

Muller Ulrich <ulrich.muller@nokia.com> (from Bonn university/Germany)
- PAN profile implementation for NOBLIN (kernel 2.2.x Affix version)

Nicolas Delest <nicolas.delest@nokia.com>
- PAN porting to Affix

Andrei Emeltchenko <andrei.emeltchenko@nokia.com>
- Hardware List maintenance
- HOWTO

Marcel Holtmann <marcel@holtmann.org>
- bluecard_cs and bt3c_cs drivers.

Johan Hedberg <johan.hedberg@nokia.com>
- Compaq iPAQ stuff.
- Testing.

Mario Joussen <mario@joussen.org>
- debian packages maintainer

Mitja Pufic <raveer@section-one.org>
- RPM packages

Christoph Scholz <scholz@cs.uni-bonn.de>
Felix Leder <leder@informatik.uni-bonn.de>
Wolfgang Hansmann <hansmann@cs.uni-bonn.de>
- Etherreal Affix patch
- Bug reports and fixes
- PAN fixes

Albert Rybalkin <albertr@iral.com>
- Author of bt950uart_cs driver.

Miklos Quartus <miklos.quartus@nokia.com>
- WEB design

I would like to thank as well all people who use Affix and gives feedback helping to make it more powerful and robust.

4. Terms and Abbreviations

PAN	Personal Area Networking Profile
HCI	Host Controller Interface

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

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.

We successfully tested Affix for interoperability with meny 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.

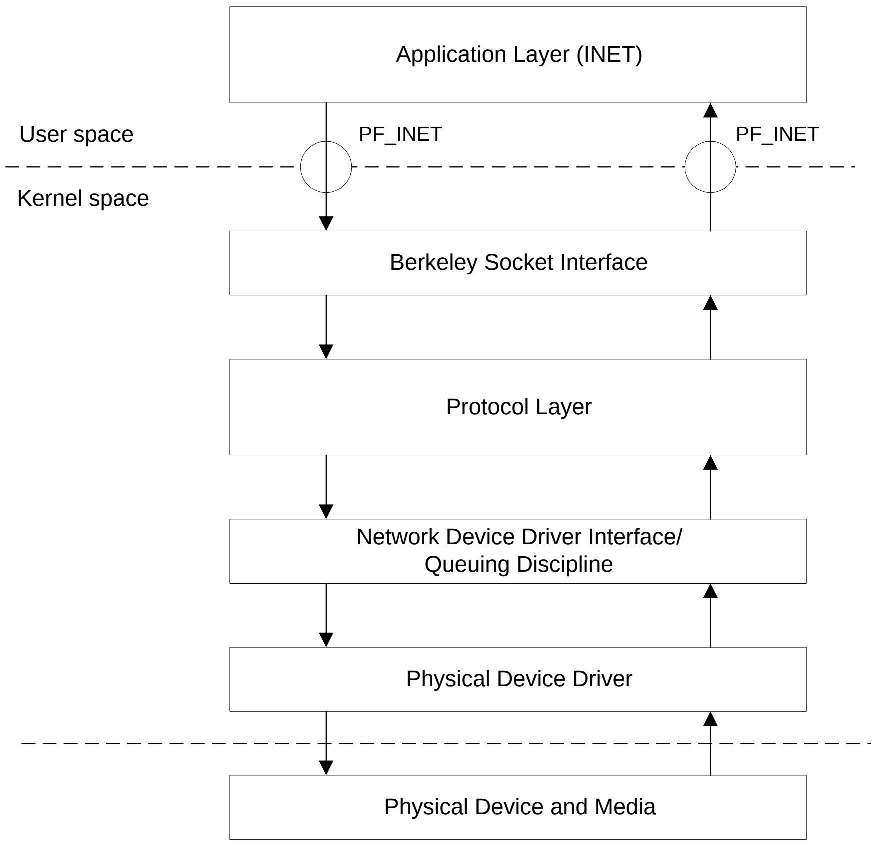
Chapter 2. General Architecture

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 loopback 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.

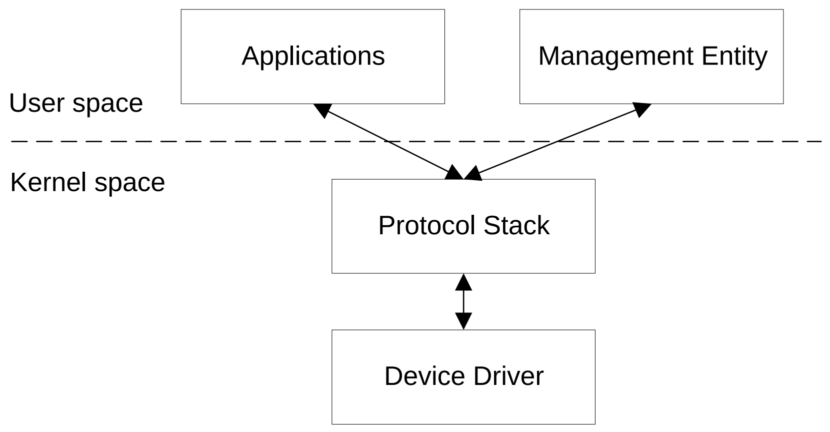
2.2. Affix architecture

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 ME to be running in the user space. Affix follows this concept.

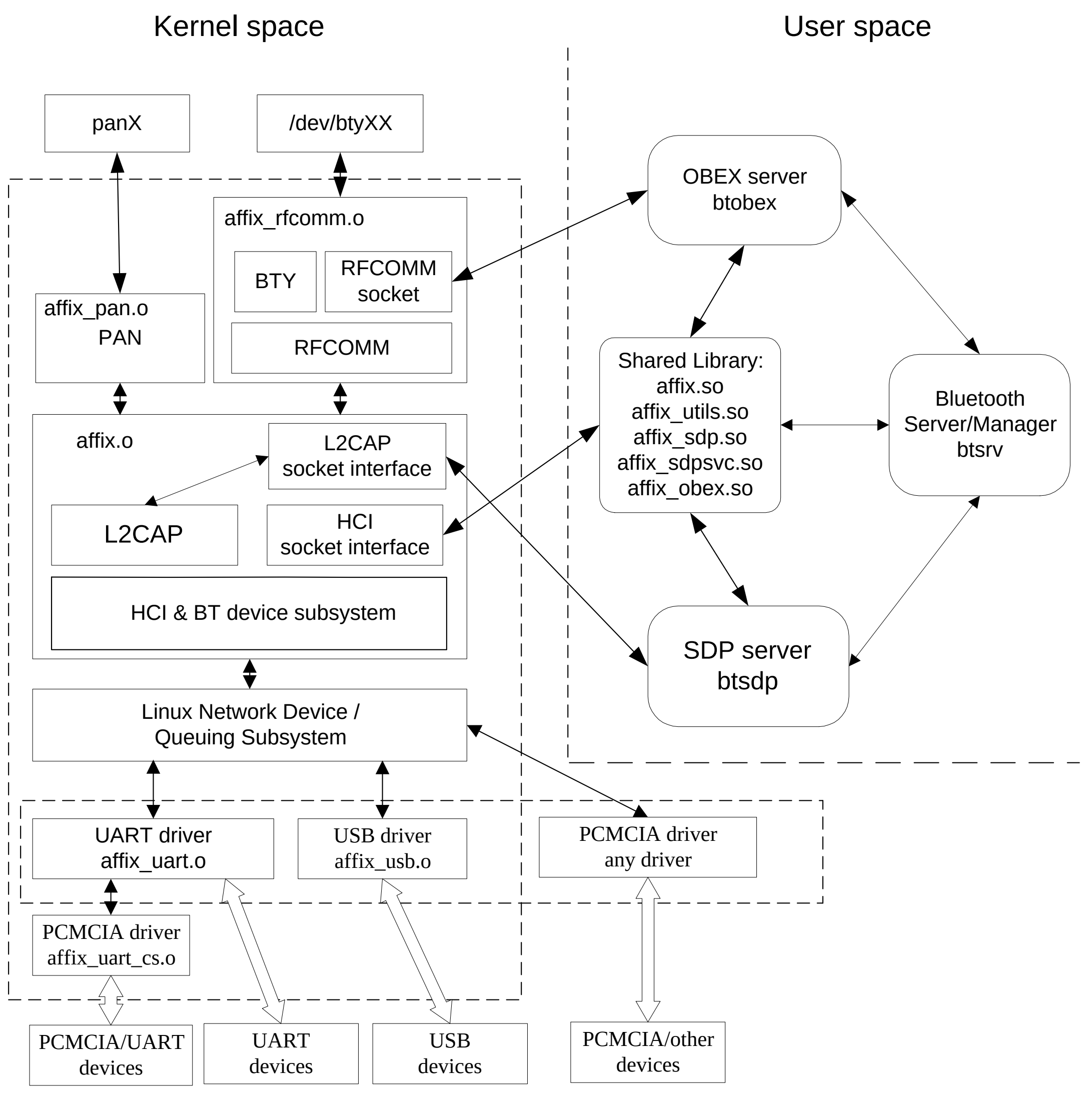
The Bluetooth protocol stack has predefined functionality and is defined in the Bluetooth system specification. The protocol stack of different systems should have the same functionality. Otherwise interoperability is not guaranteed.

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.

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/lib);
- kernel modules (/lib/modules/‘uname -r‘/kernel/{net/bluetooth,drivers/bluetooth}.

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.

Library	Description
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.o	Affix core module (HCI, L2CAP, and its socket interfaces)
affix_rfcomm.o	RFCOMM, BTY and its socket interfaces
affix_pan.o	PAN (BNEP, network interface)
affix_usb.o	Bluetooth USB driver
affix_uart.o	Bluetooth UART (H4, BCSP, TLP) driver
affix_uart_cs.o	PCMCIA driver for UART cards

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_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 - cousin of sockaddr. The structure has a name "sockaddr_affix" and has the following prototype:

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

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: "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
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, BTPROTO_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()*..

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 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.

Table 3-6. Socket security API

Function	Purpose
int setsockopt(int fd, SOL_AFFIX, BTSO_SECURITY, int *level, sizeof(int));	Sets security level: HCI_SECURITY_OPEN, HCI_SECURITY_AUTH, HCI_SECURITY_ENCRYPT, HCI_SECURITY_AUTHOR. Modes can bitwise-or’d

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 HC (Bluetooth device specific) function set. All an Affix function names have low-case style and all a HC function names have mixed-case style and are starting from HCI_ or __HCI_. The HC 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_listen_event(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 *hd);	Receives HCI event from the HCI devices. Event is stored in the buffer pointed by "event" argument and device id from where event came is stored in variable pointed by "hd".

function prototype	Description
int hci_exec_cmd(int fd, void *cmd, __u64 mask, int flags, void *event);	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_recv_msg(int fd, __u8 *param);	Receives Affix internal control message from the channel pointed by "fd" and stores data in the buffer pointed by "param" argument.
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*.
hci_get_attr(char *name, struct hci_dev_attr *attr);	Reads device attributes:. name, id, flags.
hci_get_attr_id(int hd, struct hci_dev_attr *attr);	Reads device attributes:. name, id, flags.
hci_get_attr_fd(int fd, struct hci_dev_attr *attr);	Reads device attributes:. name, id, flags.
hci_get_flags(int fd, int *flags);	Reads device flags.
hci_set_flags(int fd, int flags);	Sets device flags: HCI_FLAGS_UP
int hci_set_secmode(int fd, int mode);	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 packet type.
int hci_set_role(int fd, int role);	Set connection roles.
char *hcierror(int err);	Returns HCI error message.
char *bda2str(BD_ADDR *bda);	Converts Bluetooth address to string.
int str2bda(BD_ADDR *p, char *str);	Converts string to Bluetooth address.

The Major HC functions are presented in Table 3-8. All functions accept at least one parameter - hci descriptor and return error code, which can has following values: z,.

- 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 HC function format is: int HCI_Xxxx(int hd, param, param, ...);

The rest of parameters of HC 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 HC functions implemented in the Affix. A complete set can be found in `hci_cmds.h`.

Table 3-8. HC 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.

```
struct sockaddr_affix {
       sa_family_t   family;
       BD_ADDR       bda;
       uint16_t      port;
       BD_ADDR       local;  /* local Bluetooth adapter to connect to */
};
```

- 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_RUNNING       0x00000001
#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

#define HCI_FLAGS_SECURITY      0x0000FF00
#define HCI_SECURITY_OPEN       0x00000100
#define HCI_SECURITY_SERVICE    0x00000200
#define HCI_SECURITY_LINK       0x00000400
/* levels */
#define HCI_SECURITY_AUTH       0x00001000
#define HCI_SECURITY_ENCRYPT    0x00002000
#define HCI_SECURITY_AUTHOR     0x00004000

struct hci_dev_attr {
        int       hd;
        char      name[IFNAMSIZ];
        BD_ADDR   bda;
        int       flags;
};
```

- 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                 hd;
        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

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 to register information about itself with SDP server. it allows for a client to find it and connect to it.

3.4.1. Service Attributes

Each service attribute describes a single characteristic of a service.

Universal attributes are defined in the SDP specification. Their definition and purpose are clearly stated and well understood, and are briefly described here.

ServiceRecordHandle is a unique service identifier (32 bits) valid only on the device that gave out the handle. The same service on two different devices would have different handles.

ServiceID is a universally unique service identifier (UUID) guaranteed to be the same on all devices where the service can be found.

ServiceClassIDSequence is a sequence 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".

AccessProtocols is one or more sequences of protocol descriptors. A protocol descriptor is a sequence of attributes - UUID of protocol, version number and port number, where a service can be accessed.

Bluetooth Profile Descriptor is a sequence of standard "Bluetooth Profile" descriptions that the service is known to conform to. Bluetooth profile description contains UUID of profile, its version number.

ServiceRecordState is 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.

ServiceInfoTimeToLive is 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.

ServiceAvailability is 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.

ServiceName, ServiceDescription and *ServiceProviderName* these are user visible strings conveying information about a service. These can be provided in one (universal) or more languages.

LanguageBasedAttributeIDList contains language specific attribute identifiers for user visible strings.

BrowseGroupList consists of a sequence 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.

ClientExecutableURL location of a client platform specific (Win, Unix, Palm) application that can use the service.

DocumentationURL location of additional service documentation.

IconURL location of an icon that can be used to represent the service.

ServiceDatabaseState is an attribute specific to the service discovery server itself. If this value changes, then the service repository was recently modified.

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.

3.4.2. SDP data structures and generic API

3.4.2.1. UUID

A UUID is a universally unique identifier that 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.

To reduce the burden of storing and transferring 128-bit UUID values, a range of UUID values has been pre-allocated for assignment to often-used, registered purposes. The first UUID in this pre-allocated 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 pre-allocated 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 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.

Table 3-9. UUID API

Function	Purpose
void makeUUIDFrom16Bits(UUID *pUUID, UUID16Bit value16Bit);	Creates UUID object from 16 bit value.
void makeUUIDFrom32Bits(UUID *pUUID, UUID32Bit value32Bit);	Creates UUID object from 32 bit value.
void makeUUIDFrom128Bits(UUID *pUUID, UUID128Bit *value128Bit);	Creates UUID object from 128 bit value.
UUID *convertTo128Bits(UUID *pUUID);	Converts any type UUID to 128 bits UUID. It allocates memory to store new object.

3.4.2.2. Service Record

All of the information about a service that is maintained by an SDP server is contained within a single service record. The service record consists entirely of a list of service attributes.

ServiceRecord object is used to store information about service. It maintains a list of all service attributes.

3.4.2.3. Creating attributes

The generic data structure *SDPData* is meant to hold any attribute, universal or dynamic. Each SDPData contains one attribute. The SDPData object contains the service attribute identifier, a data type identifier (int, char *, sequence etc), followed by the value.

Table 3-10. Attribute API

Function	Purpose
SDPData *createSDPData(DataTypeDescriptor dtd, void *pValue);	Creates SDP attribute object of certain type.
int addToAttributeList(ServiceRecord *svcRec, ServiceAttributeIdentifier attrId, SDPData *pSDPData, int replaceFlag);	Adds attribute object to the attribute list of ServiceRecord object.
void freeSDPData(SDPData *pData);	Destroys entire attribute object.

3.4.3. 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 3-11. Service client SDP API

Function	Purpose
int SDPInit(0);	Initializes the SDP infrastructure.
SDPServerHandle SDPOpenServerConnection(struct sockaddr_affix *saddr);	Opens a connection to the SDP server.
void SDPCloseServerConnection(SDPServerHandle srvHandle);	Closes a connection to the SDP server.
int SDPServiceSearchRequest(SDPServerHandle srvHandle, slist_t *svcSearchList, uint16_t maxSvcRecordCount, slist_t **svcResponseList, uint16_t *handleCountInResponse);	Makes a service search request to the SDP server and returns service handle list of the services that match to search pattern. Search pattern - list of service class identifiers.

Function	Purpose
int SDPServiceAttributeRequest(SDPServerHandle srvHandle, ServiceRecordHandle svcHandle, AttributeRequestType attrReqType, slist_t *attrIDList, uint16_t maxAttrIDByteCount, ServiceRecord **_svcRec, uint16_t *maxAttrResponseByteCount);	Makes an attribute request of a service with a handle "svcHandle" and returns requested attribute list. "attrIDList" list defines attributes to return.
int SDPServiceSearchAttributeRequest(SDPServerHandle srvHandle, slist_t *svcSearchList, AttributeRequestType attrReqType, slist_t *attrIDList, uint16_t maxAttrByteCount, slist_t **svcResponseList, uint16_t *maxAttrResponseByteCount);	Makes a service search request which returns attributes of all found services. Combines previous two functions.
int __SDPServiceXXX(struct sockaddr_affix *saddr, ...);	These are variations of former three functions that take address to send request instead of ServerHandle. They open/close connection internally.
int getXXX(ServiceRecord *svcRec, ...)	Template of functions: extracts individual attributes (XXX) from service record, e.g.: service name, access port, profile id, ...

3.4.3.1. SDP client initialization

The SDP infrastructure must be initialized before any activity using SDPInit(int mode).

3.4.3.2. Connecting and disconnecting from SDP server

Before any request to the SDP server can be made, a client has to make a connection to the SDP server using following function:

SDPServerHandle SDPOpenServerConnection(struct sockaddr_affix *saddr)

This function accepts one parameter - address of the server and returns SDP server handle. An Address is given using Affix sockaddr structure.

To close a connection, call [void SDPCloseServerConnection(SDPServerHandle srvHandle);].

The ServerHandle is provided as a first argument to SDPServiceXXX() functions.

3.4.3.3. 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 service class identifiers in the pattern can be 12. SDPServiceSearchRequest() is used for that.

- 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 defined in UUID support.
- 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 setup in svcResponseList (a slist_t) and returned to the caller. The number of handles found is also set in handleCountInResponse.

3.4.3.4. 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:

- ServiceRecordHandle of the service both need to be provided to uniquely identify the service.
- The API supports requesting the entire range of attributes or a certain subset and this is specified in the request type, followed by the actual list of attributes in attrIdList (slist_t). Attribute identifiers are 16 bit unsigned integers specified in one of 2 ways. *IndividualAttributes* - 16bit individual identifiers actual attribute identifiers in ascending order. *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.
- maxAttrByteCount - this is a pointer to a 16-bit integer, which is set to indicate the number of bytes of attributes returned.
- A successful execution of the command results in E_OK being returned else a negative value indicating the type of error (timeout, invalid args etc) is returned.

3.4.3.5. 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. 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.

- The svcSearchList (slist_t) is a singly linked list containing elements of the search pattern. Each entry in the list is a UUID(DataTypeUUID_16) of the service to be searched.
- AttributeSpecification attrSpec, specifies whether the request is for a subset of attributes or the entire range as described in Service attributes request from a specific service record.
- The argument attrIDList (slist_t) is a singly linked list containing attribute identifiers desired. Every element is either a uint16_t(attrSpec = *IndividualAttributes*) or a uint32_t (attrSpec=*RangeOfAttributes*).
- The argument maxAttrIDByteCount is the byte count of the number of attribute IDs specified in the request list.
- The pointer svcResponseList (slist_t **) is set on successful return from the query, and will contain the service record handles of matching services.
- The argument maxAttrResponseByteCount is a pointer to a 16-bit integer, which is set to indicate the number of bytes of attributes returned. This pointer is set on successful return.

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

3.4.3.6. SDP client utility functions

The client API in the previous section were very basic, and do not provide mechanisms to extract attributes of individual ServiceRecord or their sub-components.

The getXXX(ServiceRecord *svcRec, XXX *xxx) functions provided in sdpclt.h, are intended for extracting attributes, where XXX is an attribute type. These are not listed here, but the API definition is self-explanatory. These are complements of the setXXX(XXX *x) utility functions for the server, but packaged in a different library.

3.4.4. 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 PDUs are never sent on Bluetooth channels, but done using TCP sockets. The service registration requests and responses do 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. For details, please read the attached header files.

Table 3-12. Service provider SDP API

Function	Purpose
int SDPInit(SDP_SVC_PROVIDER);	Initializes SDP infrastructure.

Function	Purpose
SDPServerHandle SDPOpenServerConnection(struct sockaddr_affix *saddr);	Opens a connection to local SDP server.
void SDPCloseServerConnection(SDPServerHandle srvHandle);	Closes a connection to SDP server.
ServiceRecord *createServiceRecord(void);	Creates a basic Service Record object w/o attributes.
void freeServiceRecord(ServiceRecord *svcRec);	Destroys Service Record object and free all used resources.
ServiceRecord *createRFCOMMServiceRecord(int port, int flags);	Creates a Service Record object and set common RFCOMM attributes.
int registerServiceRecord(SDPServerHandle srvHandle, ServiceRecord *svcRec);	Registers a Service Record object with the SDP server.
int updateServiceRecord(SDPServerHandle srvHandle, ServiceRecord *svcRec);	Updates a Service Record object on the SDP server.
int deleteServiceRecord(SDPServerHandle srvHandle, ServiceRecord *svcRec);	Deletes a Service Record object from SDP server.
int setXXX(ServiceRecord *svcRec, ...)	Template of functions: set individual attributes (XXX) to service record, e.g.: service name, access port, profile id, ...

3.4.4.1. Initialize the SDP infrastructure

The SDP infrastructure must be initialized before any activity and this is done using SDPInit(SDP_SVC_PROVIDER).

3.4.4.2. Connecting and disconnecting from SDP server

Before any service registration can be made service provider must make a connection to local SDP server using following function.

SDPServerHandle SDPOpenLocalServerConnection(void);

The connection must be maintained the whole lifetime of the service. If connection is closed then SDP server removes all services that belongs to this connection. It prevents from the situation when service provider process terminates and forgets to delete registered services.

To close a connection, call SDPCloseServerConnection().

3.4.4.3. Create a service record

3.4.4.4. Creating/Modifying service attributes

setXXX(ServiceRecord *svcRec, XXX *value) functions provided in sdpsrv.h, are intended for creating and modifying service attributes, where XXX is an attribute type. These are not listed here, but the API definition is self-explanatory. Only point to note is that any old XXX that exists will be deleted before the new value is set.

Note that the pointers passed to setXXX() 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.

3.4.4.5. Register service record with SDP server

3.4.4.6. Update service record attributes

3.4.4.7. Delete service record from SDP server

Deletes a service record pointed to by svcRec. This deletes the SDP server's image of the service record. Returns 0 if successful or -1 if there is a failure.

int deleteServiceRecord(ServiceRecord *svcRec);

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 any object is stored in a file. Before object can be transferred it must be stored in a file. The Affix has a following function set for that:

Table 3-13. OBEX file API

Function	Purpose
obex_file_t *obex_create_file(char *name);	Creates file object from a file "file". If "file" 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 3-14. 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. Retuns 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.

Function	Purpose
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 3-15. 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".
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 transfered on GET request. (usually is used in get() callback function)

Chapter 4. Usage manual

4.1. Personal Area Networking Profile (PAN)

4.1.1. Introduction

Personal Area Networking Profile (PAN) allows Bluetooth devices to form an ad-hoc network, access a remote network through a network access point.

The ad-hoc network in the PAN profile consists of a single Bluetooth piconet. The PAN profile does not cover scatternet networking - the case where multiple piconets are involved.

In simple words, the PAN profile can be understood as Ethernet emulation over Bluetooth. Ethernet packets are encapsulated into Bluetooth special packets and are sent over Bluetooth link.

Device, participating in the ad-hoc network, is assigned one of following roles:

- **Network Access Point (NAP)**: A Bluetooth device that provides some of the features of an Ethernet bridge. The NAP device forwards Ethernet packets between each of the connected Bluetooth devices, reffered to as PAN Users (PANU). In addition, NAP forwards Ethernet packets to/from a different network connected to the NAP via Layer 2 bridging or Layer 3 routing mechanism.
- **Group Ad-Hoc Network (GN)**: A Bluetooth devices acting in the same way as NAP device with the exception that it does not provide access to any additional networks.
- **PAN User (PANU)**: A bluetooth device that uses either the NAP or the GN service.

In other words, the PAN ad-hoc network consists of one NAP or GN device and several (up to 7) PANU devices.

The PAN profile in the Affix is implemented as kernel module `affix_pan.o`. The state of the PAN module can be seen in its proc file system entry, /proc/net/affix/pan".

4.1.2. Setting up the NAP device

To enable access from PANU device to external network the NAP must be configured to have briding or routing functionality.

Bridging functionality is supported by the Linux kernel. In order to enable it, it is necessary to compile the kernel with following option enabled: [Networking Options/802.1d Ethernet Bridging].

Configuration of the Ethernet bridge is done using **brctl** utility that is distributed with the **bridge-utils** package.

First, initialize the PAN interface: "**btctl paninit nap bt0**". This command will create a new network (PAN) interface called pan0, pan1, etc.

Second, set up a bridge:

1. add a new bridge to the system: "**brctl addbr br0**". This command will create a new network interface called br0. It is used with the ifconfig command in the same way as with any ethernet interface.
2. add Ethernet interface and PAN interface to the bridge: "**brctl addif br0 eth0**" and "**brctl addif br0 pan0**".
3. activate the bridge and assign an IP address. "**ifconfig br0 up**". It may have the same IP address as the eth0 interface.
4. check and fix, if necessary, the routing table. Default gateway must point to the bridge interface br0 (route add default ...). If eth0 and br0 have the same IP address then entry pointing to eth0 must be removed from the table (route del ... eth0).

Third, activate the PAN interface: **ifconfig pan0 up**. It’s also possible to assign an IP address to the PAN interface, but as bridge has IP interface it’s not really necessary.

Active NAP PAN interface waits for connection establishment.

To enable multiple clients on one NAP, the Bluetooth device on the NAP should be configured to accept role switch when connection occurs. This is done by: "**btctl role allow|deny master**". In this case, the NAP device becomes master when a client (PANU) connects to it.

4.1.3. Setting up the GN device

The GN device is not need to be connected to a non-bluetooth network. Set up procedure is simpler then in NAP case.

First, initialize the PAN interface: "**btctl paninit gn bt0**". This command will create a new network (PAN) interface called pan0, pan1, etc.

Second, activate the PAN interface: **ifconfig pan0 up**. It’s necessary to assign an IP address to the PAN interface.

Active GN PAN interface waits for connection establishment.

4.1.4. Setting up the PANU device

First, initialize the PAN interface: "**btctl paninit panu bt0**". This command will create a new network (PAN) interface called pan0, pan1, etc.

Second, activate the PAN interface: **ifconfig pan0 up**. It's necessary to assign an IP address to the PAN interface.

If there is at least one GN or NAP device in a range, PANU will connect to it as soon as they are both activated.

4.1.5. IP address assignment

The IP address assignment for Bluetooth PAN devices SHALL be done through "*Dynamic Configuration of IPv4 link-local addresses*".

However, IP address assignment may be done also either manually or automatically using DHCP protocol.

For ad-hoc networking with GN, the GN device, running DHCP server, may provide an IP address to the PANU device using DHCP protocol.

4.1.6. Filttering

Multicast filter and protocol filter for the local device can be configured with "btctl pan". By default, no protocol filters are in use.

At a PAN User, the multicast configuration of the network device is used as multicast address filter configuration.

At a Group Ad-hoc Network/Network Access Point, no multi- cast address filters are in use by default. Setting a filter at a Group Ad-hoc Network/Network Access Point also affects packets exchanged between PAN users connected to the same Group Ad-hoc Network/Network Access Point.

The Ethernet Broadcast address FF:FF:FF:FF:FF:FF is never filtered out.

If a filter setting is rejected by the remote device, the remote device keeps its previous setting according to the specification. To avoid side effects, the current implementation always resets the affected filter in this case.

I. Manual pages

btctl

Name

`btctl` — Affix control utility

Synopsis

btctl [-i iface] [--nosdp] [interface] [command] [parameters..]

Description

btctl is the control program to configure kernel-resident Bluetooth interfaces.

If no arguments are given, *btctl* displays the status of currently active interfaces.

Device cache

btctl has a device cache. At inquiry or discovery procedure all found devices are added to the device cache. A number is assigned to each entry (according to the order in the cache), which can be used instead of Bluetooth address. btctl searches for the Bluetooth address in the device cache. The content of the cache is shown by "btctl list" command.

The Device cache is also used by btsrv to store link keys.

Commands

btctl commands can be divided in several groups:

- general commands
- device commands
- security commands
- HCI commands.
- UART commands
- RFCOMM commands
- SDP commands
- OBEX commands.
- PAN commands.

General commands

General commands group includes commands:

- help <cmdname>

  Shows command description.

- list

  Shows contents of the device cache (all known devices).

- flush

  Cleans up device cache.

- up

  Brings Bluetooth interface up.

- down

  Brings Bluetooth interface down.

- ping <bda> <size>

  Sends a L2CAP echo request packet.

- debug [+|-][<module>|<type>|<detail>|<all>]

  Enables or disables debugging messages for certain module or certain type.

  + or - enables or disables certain features.

  module - hcicore|afhci|hcisched|hcimgr|hcilib|hci | pl2cap|afl2cap|l2cap | prfcomm|afrfcomm|bty|rfcomm |usb|uart|drv | allmod type - init | xmit | xmite | recv | recve | ctrl | ctrle | timer | allerror | alltype

detail - dump | chardump | parse | fname | func | alldetail

Security commands

Security commands control

- security <mode>

  Sets security mode for the device.

- addpin [<address>|default] <pin>

  Adds a PIN code to the Affix control block. This PIN code is used for authentication of a device with address <address>. If "default" is used instead of address then it will be used if no dedicated PIN code found.

- rmpin [<address>|default]

  Removes a PIN code from Affix control block.

- unbond <address>

  Removes a paring with device with address <address>. Practically it means that the link key is removed from Affix control block and device cache as well.

- pair <address>

  Creates a paring with the device. If btsrv is running as "btsrv --managet-keys" then the link is added to device cache as well.

  Some devices require to be pared first (like Ericsson T68).

HCI commands

HCI commands are used to manage Bluetooth device.

- inquiry [length]

  Makes an inquiry during length "length" to find neighbor devices. It shows device address and device class of found devices. If no length provided then default length is used - 8 secs.

  Each found device is shown with an order number that later can be use instead of address to identify remote device.

- discovery [length]

  Makes a discovery during length "length" to find neighbor devices. In addition to "inquiry" it also resolves name of the found device and shows with address and class. If no length provided then default length is used - 8 secs.

  Each found device is shown with order number which later can be use instead of address to identify remote device.

- remotename <bda>

  Resolves a name of the remove device. Probably not really useful because discovery does it as well.

- bdaddr

  Reads device address from the card. Probably not really useful because btctl shows it.

- name [name]

  If "name" is given then sets new name for device. Otherwise reads current name.

- role <allow|deny> <master|slave>

  Sets connection roles. Has two arguments and defines connection roles for incoming and outgoing connections. First argument defines roles for outgoing connections: allow or deny. allow - allow role switching, deny - deny role switching. Second argument defines roles for incoming connections: slave or master. slave - remain slave, master - become the master.

- class [<class> | <major> <minor> <service1> ... <servicen>]

  Sets the device class. If hexadecimal value is given then it is set as a device class. Otherwise the class is given in textual format and has the following syntax:

  Major can be one of: misc, computer, phone, audio, lap, peripheral. It defines what this device actually is.
  Computer devices (major - computer) : desktop, server, laptop, handheld, palm.
  Phone devices (major - phone): cellular, cordless, smart, modem.
  Audio devices (major - audio): headset.
  Service Class can be set of following: networking, information, audio, transfer, rendering, capturing, telephony.

  If no arguments are given for the command then it shows current device class.

- pkt_type [<0xXXX> | <mnemonic>]

  Sets packet types, which are used to create connections. It accepts hexadecimal or literal arguments: DM1, DH1, DM3, DH3, DM5, DH5.

- scan [+|-][disc|conn]

  Sets a scanning mode for the device: disc or/and conn. disc tells that device will be discoverable (visible for remote devices). conn tells that device has to be connectible (allow remote device to connect to local).

UART commands

UART commands are used to attach and detach HCI interface to Bluetooth device connected to the host via serial cable or device which use legacy UART interface to communicate with it (in fact mostly it is handled by affix_uart_cs.o).

- open_uart <tty_device> <type> [speed] [flags]

  Attaches HCI interface to UART device "tty_device".

  "type" is type of the device and can take following values: h4 - for standard H4 devices, tlp - for Nokia chip based devices, bcsp - for BCSP devices, csr - for CSR H4 devices, ericsson - for Ericsson H4 devices, digi - for Digianswer H4 devices, swave - for Sillicon Wave H4 devices. Device type depends on the protocol which is used by device (H4, BCSP, TLP) or if the device requires special initialization.

"speed" - non-default speed the UART will work on (57600, 115200, 230400, 460800, 921600).

"flags" - non-default UART flags: ctl - hardware flow control, pareven - even parity, parodd - odd parity, stopb - 2 stop bits, low - low UART latency (preferred), ring - enable RI processing (only for Nokia chip based devices)

- close_uart <tty_device>

  Detaches HCI interface from UART device "tty_device"

RFCOMM commands

RFCOMM commands are used to create, check and destroy RFCOMM connections and bind it to virtual TTY lines /dev/btyXX.

- connect <address> [<channel> | [service_type]]

  Creates an RFCOMM connection to device with address <address> to service channel <channel> or service <service_type>. service_type can be one of following: serial, dun, lan, headset. If channel or service_type omitted then "serial" is assumed.

  After connection is established it's bind to virtual TTY line /dev/btyXX.

- status

  Shows RFCOMM connections and bound TTY lines.

- disconnect <line>

  Destroys an RFCOMM connection and releases TTY line.

SDP commands

SDP commands are used to browse services available on the device.

- browse <address>

Browses available services on a device with address <address>. It shows information about service: service name, provider name, description, protocol info (access ports), profile info.

Services represented as hierarchy (tree) that is started from the root (BrowseGroupRoot). Browse command does request to get whole service hierarchy. Not all devices support it. (Bluetooth stack for Windows from Digianswer does not).

If "local" is provided instead of <address> then it shows local services.

- search <address>

  This command does the same as "browse" but it tries to search for all known services instead of requesting whole service hierarchy. This command is used if "browse" does now show any services.

OBEX commands

OBEX commands are used for transferring files and objects to/from devices.

- ftp

  Starts btctl in interactive mode.

- open <address> [channel]

  Opens OBEX connection to FTP service (File Transfer Profile). If channel is given then use it instead of looking for it in SDP database.

  This command works only in interactive mode.

- close

  Closes existing OBEX connection.

  This command works only in interactive mode.

- ls [<address> [<channel>]] [dir]

  Shows content of the directory on the OBEX server.

In interactive mode connection must be opened by "open" command first and address and channel parameters must not be provided.

- put [<address> [<channel>]] <local-file> [remote-file]

  Sends a file to the server.

  In interactive mode connection must be opened by "open" command first and address and channel parameters must not be provided.

- get [<address> [<channel>]] <remote-file> [local-file]

  Receives a file from the server.

  In interactive mode connection must be opened by "open" command first and address and channel parameters must not be provided.

- push <address> [<channel>] <local-file> [remote-file]

  Sends a file to the server (using Object Push Profile).

- rm [<address> [<channel>]] <remote-file>

  Deletes a file on the server.

  In interactive mode connection must be opened by "open" command first and address and channel parameters must not be provided.

- cd <dir name>

  Changes an active directory.

- mkdir <dir name>

  Creates a new directory on the server.

PAN commands

PAN commands are used to initialize and configure PAN interfaces.

- paninit [panu|nap|gn]

  Initializes the PAN interface (pan0, pan1, ..) and assign it role: panu, nap or gn.

- panstop

  Releases the PAN interface.

- panctl <interface> [m (start_addr stop_addr)*] [p (range_start range_stop)*]

  Sets multicast or protocol filters for PAN interface <interface> (pan0, pan1, ...). If now filters are given then shown current filter settings.

EXAMPLES

Following examples explains commands usage.

1. btctl bt1 up

   Brings interface bt1 up

2. btctl b0 down

   Brings bt0 down

btsrv

Name

`btsrv` — Affix ”super-server”

Synopsis

btsrv [-d] [-v] [--manage-keys] [--nosvc] [--nosdp | -s] [--nogui | -g] [--config *`config_file`* | -C *`config_file`*]

Description

btsrv is the "super-server" daemon which listens for connections on certain Bluetooth sockets and invokes program to service the request. In other words it manages profiles.

btsrv also handles service registration/deregistration with SDP server. In order for client to find and connect to service it first connects to SDP server and retrieves from it information about available services.

Upon execution, btsrv reads its configuration from a configuration file, which, by default, is /etc/affix/btsrv.conf. It has service and device configuration information.

OPTIONS

"-d" option starts btsrv in a background.

"-v" option turns on debugging.

"--manage--keys" option turns on key management functionality. This means that btsrv will store all known keys in cache file and later use them for authentication purpose. Hence there is no need to type the PIN code always when Affix runs.

"--nosvc" option disables start of services. It is useful when Affix works only as a client (nobody connects to it).

"--nosdp" option disables SDP usage. In fact should not be used, because all Bluetooth enabled systems must use SDP.

"--nogui" option disables graphical PIN helper. It is used if you add PIN code non-interactively (using "btctl addpin") and does not use X windows system.

"--config" options is used to provide other then standard configuration file.

btsrv.conf

Name

`btsrv.conf` — Affix ”super-server” (btsrv) configuration file

Description

Upon execution, btsrv reads its configuration from a configuration file, which, by default, is /etc/affix/btsrv.conf. It contains various configuration entries. Comments are denoted by a ’#’ at any place of the line.

Each entry is represented in XML style and starts with element <entryname> and ends with element </entryname>. Starting element can have attributes. Each entry has certain parameters depending on the semantic of the entry. Currently two entries are defined in configuration file: *service* and *device*.

service entry contains parameters related to the service and has the following syntax:

```
<service name>
      param1
      param2; param3
      param4 arg1, arg2, .., argn
      param5 arg1 arg2 .. argn
</service>
```

Different parameters can be placed on the same line and separated by a semicolon. Each parameter can have arguments followed by a parameter and can be separated by a comma or white space.

Service entry has the following parameters:

- *name.* It has a string type and defines service name. Service name in SDP database is set to this value.
- *profile.* It has a word type and defines profile to which this service belongs to: SerialPort, DialupNetworking, LANAccess, OBEXFileTransfer, OBEXObjectPush.
- *provider.* It has a string type and defines service provider name. Service provider name in SDP database is set to this value.
- *descript.* It has a string type and defines service description. Service description in SDP database is set to this value.
- *security.* It has a list type and defines security level to access a service. This parameter accepts following arguments: *open*, *auth*, *author*, *encrypt*. *open* argument defines that no security control has to be involved to access this service (no authentication, authorization and encryption requires). *auth* tells that authentication requires to access this service. *author* tells that authorization requires to access this service. *encrypt* tells that connection encryption requires for accessing this service.

- *class*. It has a list type and defines Bluetooth service class that this service belongs to. This parameter affects "Class of Device" which is available after "inquiry" or "Discovery" procedure. It accepts the same values as service class for device (see in device entry description).
- *flags*. It has a list type and controls special behavior of the incoming connection. This parameter accepts following arguments: tty, std. *tty* tells that connection has to be bound to TTY line (/dev/btyXX) and allows applications to use legacy TTY device to communicate through Bluetooth. *std* tells that connection has to be multiplexed to stdin/stdout streams (it allows applications working on stdin/stdout communicate through Bluetooth).
- *exec*. It has string argument and defines command that is to be executed when request is found. Command line can have special format characters with preceded "%": %a, %l, %i, %c. "%a" replaced by remote device address, "%l" - TTY line number of virtual device (/dev/btyXX), "%i" - connection identifier (file descriptor), "%c" - RFCOMM server channel.

Device entry contains parameter related to device configuration and has the following syntax:

```
<device ifname>
      param1
      param2; param3
      param4 arg1, arg2, .., argn
      param5 arg1 arg2 .. argn
</device>
```

Different parameters can be placed on the same line and separated by a semicolon. Each parameter can have arguments followed by a parameter and can be separated by a comma or white space.

Service entry has the following parameters:

- *ifname*. Defines interface name to which this entry belongs to (*, bt0, bt1, ...).
- *name*. Has a string argument and defines Bluetooth device name. This name is available for remote devices after discovery procedure.
- *class*. Has a list type and defines Bluetooth "Class of Device". This class is available for remote devices after inquiry or discovery procedure. Class arguments consist of 3 groups: major class, minor class and service class. minor class takes values depending on the major and defines device type inside certain major class.

  Major can be one of: misc, computer, phone, audio, lap, peripheral. It defines what this device actually is.
  Computer devices (major - computer): desktop, server, laptop, handheld, palm.
  Phone devices (major - phone): cellular, cordless, smart, modem.
  Audio devices (major - audio): headset.
  Service Class can be set of following: networking, information, audio, transfer, rendering, capturing, telephony.
- *scan*. Has list argument and defines scanning mode for device: disc, conn. *disc* tells that device will be discoverable (visible for remote devices). *conn* tells that device has to be connectible (allow remote device to connect to local).

- *security*. Has a list type and defines what security mode device will use. It accepts following arguments: open, link, service, auth, author, encrypt. *open* mode means that no security actions has to be taken and any device can connect without authentication, authorization and encryption does not required for the connection. *link* mode means that security actions has to be taken at link level. Before connection is created certain actions are taken. It depends on the rest of parameters (auth, author and encrypt) and has the same meaning as for security option in service entry except that extends to link level. service means
- *role*. Has two arguments and defines connection roles for incoming and outgoing connections. First argument defines roles for outgoing connections: allow or deny. allow - allows role switching, deny - denies role switching. Second argument defines roles for incoming connections: slave or master. slave - remain slave, master - become the master.
- *pkt_type*. Has a list type and defines what packets to use for connections: DM1, DH1, DM3, DH3, DM5, DH5.

Examples

The following example shows how to configure service and device bt0.

```
<service "Serial Port">
       profile    SerialPort;
       provider   "Nokia";
       descript   "Serial Port Profile";
       security   open
       class      information networking;
       flags      tty;
       exec       "/etc/affix/serial %l %i %a %c";
</service>

<device bt0>
       name       "Selma"
       class      computer handheld, information, audio, networking, transfer;
       scan       disc, conn
       security   service
       role       allow, slave
       pkt_type DM1, DH1, DM3, DH3, DM5, DH5
</device>
```

Appendix A. Installation instructions

The Affix distribution consists of two separate packages: *affix* and *affix-kernel*. *affix* contains source files to compile and install user mode programs and *affix-kernel* contains source file to compile kernel modules.

A.1. affix-kernel package

A.1.1. Package components

The *affix-kernel* package compiles and installs kernel-modules and developers header files.

Table A-1. affix-kernel components

Module	Description
net/bluetooth/affix.o	Affix core module (HCI, L2CAP, and its socket interfaces).
net/bluetooth/affix_rfcomm.o	RFCOMM, BTY and its socket interface.
net/bluetooth/affix_pan.o	PAN (BNEP, network interface).
drivers/bluetooth/affix_usb.o	Bluetooth USB driver.
drivers/bluetooth/affix_uart.o	Bluetooth UART (H4, BCSP, TLP) driver.
drivers/bluetooth/affix_uart_cs.o	PCMCIA driver for UART cards.
/usr/include/affix/bluetooth.h	Main Affix header file: Affix data structures, system calls, types.
/usr/include/affix/hci_types.h	HCI data structures and identifiers.
/usr/include/affix/hci_cmds.h	HCI commands implementation (used by "affix" package).

A.1.2. System requirements

- USB support. In order to use USB Bluetooth adapter it’s necessary to enable USB support in the kernel.
- PCMCIA support. In order to use PCMCIA (PCCARD) Bluetooth adapter it’s necessary to enable PCMCIA support in the kernel.
- DEVFS support (optional, recommended). Provides automatic creation of /dev/btyXX device files.
- HOTPLUG support (optional, recommended). Used in a case of USB support. Provides automatic way to load drivers for USB modules on demand.
- modutils support (optional, recommended). Provides automatic updates for module information from user specific files. In order for the system to load modules automatically on demand modules information has to be added to /etc/modules.conf. modutils does it for you.

- In oder to use Broadcomm BCM2033 USB dongle and 3Com Bluetooth PCCARD install Bluetooth Firmware Loader (http://bluez.sourceforge.net/download/bluez-bluefw-0.9.tar.gz).

A.1.3. Configuration and compilation

In order to properly compile affix-kernel package please follow these steps:

1. Copy affix-kernel-xxx.tgz package to any directory (e.g. /usr/src/).
2. Unpack it: tar xvfz affix-kernel-xxx.tgz
3. Configure it: make config

   Configuration script offers configuration options to select. Answer y/n to the questions depending on the configuration you desire.

4. Compile it: make all
5. Install it: make install

   This step requires root privilege.

6. Update system configuration.
   - Update protocol aliases.

     It’s required to automatically load Affix modules when application requires the Affix support. Installation script installs /etc/modutils/affix which contains necessary aliases. It works fine in a Debian distribution. If you do not have modutuls support then you need manually add following lines to your /etc/modules.conf.

     ```
     alias net-pf-27       affix
     alias char-major-60   affix_rfcomm
     ```

   - Update PCMCIA config file.

     Installation script installs etc/pcmcia/affix and etc/pcmcia/affix.conf files to the /etc/pcmcia. affix.conf is read by the PCMCIA subsystem automatically. But some time ago it was necessary to add entries to /etc/pcmcia/config manually. Please check you system and if necessary then add contents of the affix.conf file to your pcmcia config file (/etc/pcmcia/config or /etc/pcmcia/config-2.4).

     Run "killall -HUP cardmgr" in order to tell PCMCIA manager to reload configuration.

A.1.4. Usage

Modules has to be loaded into the memory if your system does not do it automatically. Run "modprobe <module name>" for that.

A.2. affix package

A.2.1. Package components

The *affix* package installs programs, libraries, system scripts, configuration files and developer header files. The complete set is the following:

Table A-2. Affix components

Component	Description
/usr/bin/btctl	Main Affix control program.
/usr/bin/btsrv	Affix server. Manage Bluetooth devices, services, link keys, PIN codes.
/usr/bin/btobex	OBEX server: provides Bluetooth FTP and Object Pushing.
/usr/bin/btftp	OBEX client: provides Bluetooth FTP and Object Pushing.
/usr/bin/btmodem	Modem emulator (for emulating DUN over LAN).
/usr/lib/libaffix.so	Core library: HCI API, Affix System API.
/usr/lib/libaffix_utils.so	Affix utils.
/usr/lib/libaffix_sdp.so	SDP service client library.
/usr/lib/libaffix_sdpsvc.so	SDP service provider library.
/usr/lib/libaffix_obex.so	OBEX library.
/etc/affix/hwinit	Hardware initialization script.
/etc/affix/hwinit.map	Device configuration database.
/etc/affix/hwinit.opts	Device initialization parameters: name, class, security, scanning.
/etc/affix/btsrv.conf	Affix server configuration program.
/etc/affix/btsrv-gui	Affix server graphical helper (called by "btsrv").
/etc/affix/ppp_server	DUN and LAN PPP server initialization script (called by "btsrv").
/etc/affix/serial	Serial Port Profile server script (called by "btsrv").
/usr/include/affix/btcore.h	Core user space API data structures and functions declarations.
/usr/include/affix/utils.h	Generic data structures and functions declarations.
/usr/include/affix/sdp.h	SDP data types and identifiers.
/usr/include/affix/sdpclt.h	SDP service client functions.

Component	Description
/usr/include/affix/sdpsrv.h	SDP service provider functions.
/usr/include/affix/obex.h	OBEX data structures and functions.

A.2.2. System requirements

The following requirements have to be fulfilled in order to install and use Affix package.

- Install "affix-kernel" package. It installs necessary header files to /usr/include/affix directory, required to compile "affix" package.
- Install "openobex" package. It is required if you would like to use Affix OBEX functionality.
- Install "python" and "python GTK" library. It is used by PIN code helper application (/etc/affix/btsrv-gui).

A.2.3. Configuration and compilation

In order to properly compile affix-kernel package please follow these steps:

1. Copy affix-xxx.tgz package to any directory (e.g. /usr/src/).
2. Unpack it: tar xvfz affix-xxx.tgz
3. Configure it: ./configure [options]

   configure script accepts following options (--enable-FEATURE):

   - enable audio support (default): --enable-audio
   - enable SDP support (default): --enable-sdp
   - enable OBEX support (default): --enable-obex
   - enable Hands Free support - experimental: --enable-hfp
   - enable debugging support: --enable-debug

4. Compile it: make
5. Install it: make install

   This step requires root privilege.

A.2.4. Usage

btctl - main control tool. So please consult "btctl" manual page for that.

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.