The University of Rhode Island

Bluetooth Programming in C for Linux

Features of BlueZ

- Real hardware abstraction
  - Generic and vendor specific drivers
  - Over 150 supported Bluetooth adapters
  - Up to 16 host adapters at the same time
- Machine architecture independent
  - Little and big endian
  - 32 bit and 64 bit systems
  - SMP safe
  - Hyperthreading
  - Preempt ready

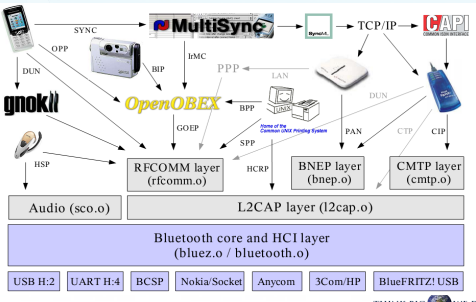
THINK BIG WE DO

More features

- BSD sockets interface
  - HCI raw socket
  - L2CAP sequential packet and datagram
  - SCO sequential packet
  - RFCOMM stream
- Complete modular design
  - Kernel code
  - User space programs and tools
- Bluetooth library with user API
  - Handling of Bluetooth addresses and devices

THINK BIG WE DO

The BlueZ architecture



THINK BIG WE DO

Supported protocols and profiles

- Current protocols
  - HCI, L2CAP, SDP,RFCOMM, OBEX, BNEP, CMTP,HIDP, HCRP,IrMC, SyncML
- Future protocols
  - TCS-BIN, AVDTP, AVCTP
- Current profiles
  - GAP, SDAP,SPP, GOEP,DUN, FAX, LAN, PUSH, SYNC, FTP,PAN, CIP,HID, HCR
- Future profiles
  - CTP, Intercom, BPP, BIP,HSP, HFP, SAPP

THINK BIG WE DO

Advantages of BlueZ

- Full source code is available under the GPL
- Socket based interfaces
- Simple API for special HCI or SDP tasks
- Access to all Bluetooth host layers
- Big user and developer community
- Very good interoperability with Bluetooth devices

THINK BIG WE DO

BlueZ host adapter setup

- Ethernet device like configuration
  - Linux Bluetooth stack specific settings
  - Host controller related configuration

```
# hciconfig
hci0:   Type: USB
        BD Address: 00:00:00:00:00:00 ACL MTU: 0:0  SCO MTU: 0:0
        DOWN
        RX bytes:0 acl:0 sco:0 events:0 errors:0
        TX bytes:0 acl:0 sco:0 commands:0 errors:0

# hciconfig hci0 up
# hciconfig -a
hci0:   Type: USB
        BD Address: 00:50:F2:7A:33:78 ACL MTU: 192:8  SCO MTU: 64:8
        UP RUNNING PSCAN ISCAN
        RX bytes:77 acl:0 sco:0 events:9 errors:0
        TX bytes:30 acl:0 sco:0 commands:8 errors:0
        Features: 0xff 0xff 0x0f 0x00
        Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
        Link policy: RSWITCH HOLD SNIFF PARK
        Link mode: SLAVE ACCEPT
        Name: 'Microsoft Bluetooth Transceiver'
        Class: 0x000100
        Service Classes: Unspecified
        Device Class: Computer, Uncategorized
        HCI Ver: 1.1 (0x1) HCI Rev: 0x1f9 LMP Ver: 1.1 (0x1) LMP Subver: 0x1f9
        Manufacturer: Cambridge Silicon Radio (10)
```

THINK BIG WE DO

Using the Bluetooth device

- Simple command line tools
  - Scanning for devices in range
  - Get information about devices and connections
  - Link quality, RSSI, transmit power level etc.

```
# hcitool scan
Scanning ...
        00:02:C7:1E:1D:B8       iPAQ H3970
        00:04:0E:21:06:FD       AVM BlueFRITZ! AP-X
        00:0A:D9:5C:75:57       Sony Ericsson T68i
        00:A0:57:AD:22:0F       ELSA Vianect Blue ISDN
        00:90:02:63:E0:83       Bluetooth Printer
        00:80:37:06:78:92       Ericsson T39m
        08:00:46:0E:CE:50       SONY Cyber-shot
        00:04:61:50:D4:3E       BFox BT-PM01B 80D43E
# hcitool info 00:04:0E:81:06:FD
Requesting information ...
        BD Address:  00:04:0E:21:06:FD
        Device Name: AVM BlueFRITZ! AP-X
        LMP Version: 1.1 (0x1) LMP Subversion: 0x1
        Manufacturer: AVM Berlin (31)
        Features: 0x2f 0xbe 0x05 0x00
                <3-slot packets> <5-slot packets> <encryption> <slot offset>
                <role switch> <RSSI> <channel quality> <SCO link>
                <HV2 packets> <HV3 packets> <A-law log> <CVSD>
                <power control>
```

THINK BIG WE DO

BlueZ and HCI

- From the user side
  - The BlueZ HCI API is a raw socket
  - The internal protocol is H:4
  - Only one command/event per write/read
  - API for abstracting HCI commands and events
- The kernel side
  - Easy kernel API for writing host drivers
  - Existing drivers for H:2 (USB) and H:4 (UART)
  - Currently 8 different drivers
  - The Affix stack also uses the BlueZ host driver API

THINK BIG WE DO

Bluetooth user space library

- Handling of Bluetooth device addresses
  - New data type bdaddr_t
  - The special address BDADDR_ANY
  - The functions bacpy, baswap and bacmp
  - Address conversion with str2ba, ba2str, strtoba and batostr

```
#include <bluetooth/bluetooth.h>

void main(int argc, char *argv[])
{
      bdaddr_t bdaddr;
      char *str, addr[18];

      str2ba("00:a5:b4:c3:d2:e1", &bdaddr);

      ba2str(&bdaddr, addr);
      str = batostr(&bdaddr);
      printf("%s %s\n", addr, str);
      free(str);

      bacpy(&bdaddr, BDADDR_ANY);
}
```

THINK BIG WE DO

The Bluetooth sockets

- Full socket interface
  - L2CAP
    - Connection-oriented (SOCK_SEQPACKET)
    - Connectionless (SOCK_DGRAM)
  - RFCOMM
    - Data stream (SOCK_STREAM)
    - Sockets can be converted to a TTY device
    - Uses the L2CAP in-kernel socket interface
- Complete abstraction from HCI
  - Creation and clearing of ACL connections
  - Sending and receiving of data packets

THINK BIG WE DO

Example program: rfcomm-server.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <lib/bluetooth.h>
#include <lib/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int s, client, bytes_read;
    socklen_t opt = sizeof(rem_addr);
    // allocate socket
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
    // bind socket to port 1 of the first available
    // local bluetooth adapter
    loc_addr.rc_family = AF_BLUETOOTH;
    loc_addr.rc_bdaddr = *BDADDR_ANY;
    loc_addr.rc_channel = (uint8_t) 1;
    bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));
    // put socket into listening mode
```

THINK BIG WE DO

rfcomm-server.c (Cont.)

```
    listen(s, 1);
    // accept one connection
    client = accept(s, (struct sockaddr *)&rem_addr, &opt);
    ba2str( &rem_addr.rc_bdaddr, buf );
    fprintf(stderr, "accepted connection from %s\n", buf);
    memset(buf, 0, sizeof(buf));
    // read data from the client
    bytes_read = read(client, buf, sizeof(buf));
    if( bytes_read > 0 ) {
        printf("received [%s]\n", buf);
    }
    // close connection
    close(client);
    close(s);
    return 0;
}
```

THINK BIG WE DO

Example program: rfcomm-client.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc addr = { 0 };
    int s, status;
    char dest[18] = "01:23:45:67:89:AB";
    // allocate a socket
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
    // set the connection parameters (who to connect to)
    addr.rc_family = AF_BLUETOOTH;
    addr.rc_channel = (uint8_t) 1;
    str2ba( dest, &addr.rc_bdaddr );
    // connect to server
    status = connect(s, (struct sockaddr *)&addr, sizeof(addr));
    // send a message
    if( status == 0 ) {
        status = write(s, "hello!", 6);
    }
    if( status < 0 ) perror("uh oh");
    close(s);
    return 0;
}
```

THINK BIG WE DO

Socket function

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

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

- For RFCOMM
  - `AF_BLUETOOTH`
    - specifies that it should be a Bluetooth socket
  - `SOCK_STREAM`
    - requests a socket with streams-based delivery semantics
  - `BTPROTO_RFCOMM`
    - specifically requests an RFCOMM socket

The socket function creates the RFCOMM socket and returns an integer which is used as a handle to that socket

THINK BIG WE DO

Addressing structures

```
// local bluetooth adapter
loc_addr.rc_family = AF_BLUETOOTH;
loc_addr.rc_bdaddr = *BDADDR_ANY;
loc_addr.rc_channel = (uint8_t) 1;
```

```
struct sockaddr_rc {
        sa_family_t rc_family;
        bdaddr_t    rc_bdaddr;
        uint8_t     rc_channel;
};
```

- The `rc_family` field specifies the addressing family of the socket, and will always be `AF_BLUETOOTH`.
- For an outgoing connection, `rc_bdaddr` and `rc_channel` specify the Bluetooth address and port number to connect to, respectively.
- For a listening socket, `rc_bdaddr` specifies the address of the local Bluetooth adapter to use and `rc_channel` specifies the port number to listen on.
- If you don't care which local Bluetooth adapter to use for the listening socket, then you can use `BDADDR_ANY` to indicate that any

THINK BIG WE DO

Establishing a connection

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

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- `sock` should be the server socket created by connect. `info` should point to a `struct sockaddr_rc` addressing structure filled in with the local Bluetooth adapter to use, and which port number to use.
- `addrlen` should always be `sizeof( struct sockaddr_rc ).`

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

```
int listen(int sockfd, int backlog);
```

- the application takes the bound socket and puts it into listening mode with the `listen` function.
- In between the time an incoming Bluetooth connection is accepted by the operating system and the time that the server application actually takes control, the new connection is put into a backlog queue. The backlog parameter specifies how big this queue should be. Usually, a value of 1 or 2 is fine.

THINK BIG WE DO

Establishing a connection (Cont.)

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

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- The `accept` function waits for an incoming connection and returns a brand new socket.
- The returned socket represents the newly established connection with a client, and is what the server application should use to communicate with the client.
- If `client_info` points to a valid `struct sockaddr_rc` structure, then it is filled in with the client's information.
- Additionally, `infolen` will be set to `sizeof(struct sockaddr_rc)`.
- The server application can then make another call to accept more connections, or it can close the server socket when finished.

THINK BIG WE DO

Using a connected socket

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

```
// send a message
if( status == 0 ) {
    status = write(s, "hello world!", 12);
}
```

```
ssize_t read (int fd, void *buf, size_t count);
```

```
ssize_t write (int fd, void *buf, size_t count);
```

- The `write` function transmits data, the `read` function waits for and receives incoming data
- Both functions take three parameter, the first being a connected Bluetooth socket.
  - For `write`, the next two parameters should be a pointer to a buffer containing the data to send, and the amount of the buffer to send, in bytes.
  - For `read`, the second two parameters should be a pointer to a buffer into which incoming data will be copied, and an upper limit on the amount of data to receive
- `write` returns the number of bytes actually transmitted, which may be less than the amount requested. In that case, the program should just try again starting from where send left off.
- `read` returns the number of bytes actually received, which may be less than the maximum amount requested.
- The special case where `read` returns 0 indicates that the connection is broken and no more data can be transmitted or received.

THINK BIG WE DO

Close the socket

- Once a program is finished with a connected socket, calling **`close`** on the socket disconnects and frees the system resources used by that connection.

THINK BIG WE DO