## Object Exchange (OBEX)

- Designed to enable devices supporting infrared communication to exchange data and commands. It was adopted from the IR communication.
- Can be used to do something as simple as business cards to something as complex as synchronizing calendars. *(Synching time until next activation or exchanging current field status)*
- Based on client-server model
- Independent of the transport mechanism or API
- Client starts the communications.
- Supported Profiles
    - Synchronization *(not really that useful)*
    - File Transfer *(Generic files are transferred)*
    - Object Push *(Simply pushes a requested file to the originator of the request)*
- Supported Protocol Stacks
    - RFCOMM
    - L2CAP
    - ACL
    - Baseband
    - USB
- API are available in C, Java and Python *(partial support for Python)*
- Basis for File Transfer Profile, Generic Object Exchange Profile, Object Push Profile, Sync Profile and Basic Imaging/Printing Profile.

### Objects

OBEX works by exchanging *objects*, which are used for a variety of purposes: establishing the parameters of a connection, sending and requesting data, changing the current path or the attributes of a file.

Objects are composed of *fields* and *headers*. As an example, the following may be the object used for requesting the phonebook from a mobile:

| | | | | |
|---|---|---|---|---|
| Object | Fields | Command | GET, Final | 0x83 |
| | | Length | total length of object | 0x00 0x29 |
| | Headers | Connection ID | 1 | **0xCB** 0x00 0x00 0x00 0x01 |

| | | | | |
|---|---|---|---|---|
| | | Name | "telecom/pb.vcf" | **0x01** *0x00 0x1e* 0x00 0x74 0x00 0x65 0x00 0x6c 0x00 0x65 0x00 0x63 0x00 0x6f 0x00 0x6d 0x00 0x2f 0x00 0x70 0x00 0x62 0x00 0x2e 0x00 0x76 0x00 0x63 0x00 0x66 0x00 0x00 |

This object contains two fields (command and length) and two headers. The first field (command) specifies that is a request for data (GET). The second field is the total size of the object, including the two fields.

This object also contains two headers, specifically a "Connection ID" and a "Name". The first byte of each header is the header's name and its content type. In this case:

- 0xCB means that this header is a "Connection ID", a number obtained previously; the two highest-order bits of 0xCB are 11, and this pair specifies that this as a 4-byte quantity;
- the first byte of the second header is 0x01; this byte identifies this header as a "Name" one; the first two bits of 0x01 are 00, meaning that the content of this header is a null-terminated unicode string (in UCS-2 form), prefixed by the number of bytes it is made of (0x00 0x1e).

A possible response, containing the requested data, could be:

| | | | | |
|---|---|---|---|---|
| Response | Fields | Response code | OK, Final | 0xA0 |
| | | Length | total length of object | 0x00 0x35 |
| | Headers | End-of-Body | "BEGIN:VCARD..." | **0x49** *0x00 0x2F* 0x42 0x45 0x47 0x49 0x4e 0x3a 0x56 0x43 0x41 0x52 0x44 |

In this example, the phonebook is assumed short enough to be contained in a single response object. The only header has 0x49 as its identifier, meaning that it is an "End of Body", the last chunk of information (also the only one, in this case). The first two bits of 0x49 are 01, meaning that the content of this header is length-prefixed data: the two next bytes 0x00 0x2F tells the length of this data (in decimal, 47), the succeeding ones are the data, in this case a phonebook comprising only an empty vCard of 47 bytes.

## Serial Port Profile (SPP)

- Used to set up virtual serial ports and enable communication via Bluetooth
- Devices are labeled as A and B, where A is the initiator and B is the acceptor.
- Below is a useful Profile Structure which explains the dependencies of the profiles.

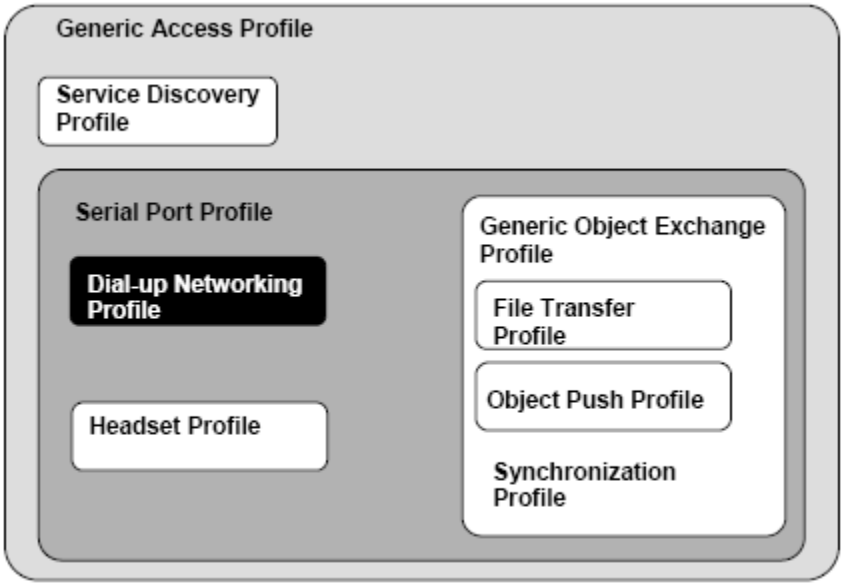


*Figure 1.1: Bluetooth Profiles*

- Applications are not aware of the serial port emulation, so they need an intermediary Bluetooth-aware application for setting up the emulated “cables”.
- Both sides must conform to the Bluetooth profile to exchange information.
- Multiple instances of this profile can run concurrently with data rates of up to 128kbps *(can be increased)*
- Can be made more secure using encryption, authentication, authorization, bonding.
- Master-Slave roles are not fixed.
- Devices can be sent into low power modes to save energy.
- Relies heavily on proper implementation of RFCOMM and L2CAP protocols.

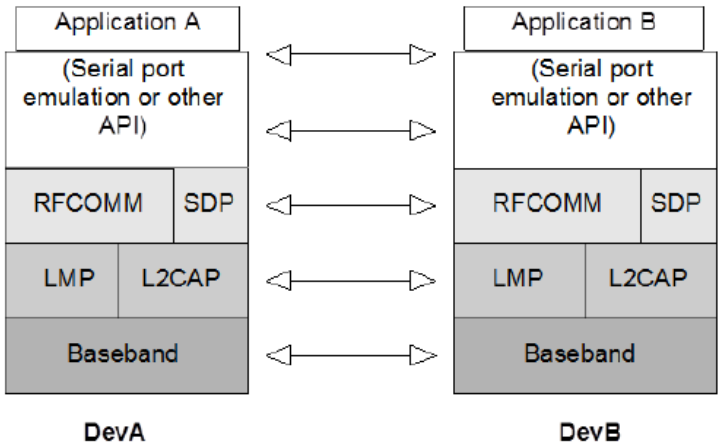


*Figure 2.1: Protocol model*

### Establish Link and Set up Virtual Serial Connection

This procedure refers to performing the steps necessary to establish a connection to an emulated serial port (or equivalent) in a remote device. The steps in this procedure are:

1. Submit a query using SDP to find out the RFCOMM Server channel number of the desired application in the remote device. This might include a browsing capability to let the user select among available ports (or services) in the peer device. Alternatively, if it is known exactly which service to contact, it is sufficient look up the necessary parameters using the Service Class ID associated with the desired service.
2. Optionally, require authentication of the remote device to be performed. Also optionally, require encryption to be turned on.
3. Request a new L2CAP channel to the remote RFCOMM entity.
4. Initiate an RFCOMM session on the L2CAP channel.
5. Start a new data link connection on the RFCOMM session, using the aforementioned server channel number.

After step 5, the virtual serial cable connection is ready to be used for communication between applications on both sides.

**Note:** If there already exists an RFCOMM session between the devices when setting up a new data link connection, the new connection must be established on the existing RFCOMM session. (This is equivalent to skipping over steps 3 and 4 above.)

### Accept Link and Establish Virtual Serial Connection

This procedure refers to taking part in the following steps:

1. If requested by the remote device, take part in authentication procedure and, upon further request, turn on encryption.
2. Accept a new channel establishment indication from L2CAP.
3. Accept an RFCOMM session establishment on that channel.
4. Accept a new data link connection on the RFCOMM session. This may trigger a local request to authenticate the remote device and turn on encryption, if the user has required that for the emulated serial port being connected to (and authentication/encryption procedures have not already been carried out).

**Note:** steps 1 and 4 may be experienced as isolated events when there already exists an RFCOMM session to the remote device.

### Register Service Record in Local SDP Database

This procedure refers to registration of a service record for an emulated serial port (or equivalent) in the SDP database. This implies the existence of a Service Database, and the ability to respond to SDP queries.

For further details refer to the SPP PDF that will be/has been uploaded.