IrDA Object Exchange Protocol

IrOBEX

Counterpoint Systems Foundry, Inc
Microsoft Corporation

January 22, 1997

Version 1.0

Authors:
Pat Megowan, Dave Suvak (Counterpoint Systems Foundry)

Contributors:
Wassef Haroun, Bei-jing Guo, Cliff Strom (Microsoft)
Kerry Lynn (Apple)
Brian McBride, Stuart Williams (Hewlett Packard)
Petri Nykanen (Nokia)
Deepak Amin (Indicus)

Editors:
Natalie A. Tyred, C.I. Clousseau, P.D.Q. Bach

Document Status: Version 1.0

INFRARED DATA ASSOCIATION (IrDA) - NOTICE TO THE TRADE -

SUMMARY:

Following is the notice of conditions and understandings upon which this document is made available to members and non-members of the Infrared Data Association.

- Availability of Publications, Updates and Notices
- Full Copyright Claims Must be Honored
- Controlled Distribution Privileges for IrDA Members Only
- Trademarks of IrDA - Prohibitions and Authorized Use
- No Representation of Third Party Rights
- Limitation of Liability
- Disclaimer of Warranty
- Certification of Products Requires Specific Authorization from IrDA after Product Testing for IrDA Specification Conformance

IrDA PUBLICATIONS and UPDATES:

IrDA publications, including notifications, updates, and revisions, are accessed electronically by IrDA members in good standing during the course of each year as a benefit of annual IrDA membership. Electronic copies are available to the public on the IrDA web site located at irda.org. IrDA publications are available to non-IrDA members for a pre-paid fee. Requests for publications, membership applications or more information should be addressed to: Infrared Data Association, P.O. Box 3883, Walnut Creek, California, U.S.A. 94598; or e-mail address: info@irda.org; or by calling John LaRoche at (510) 943-6546 or faxing requests to (510) 934-5600.

COPYRIGHT:

1. Prohibitions: IrDA claims copyright in all IrDA publications. Any unauthorized reproduction, distribution, display or modification, in whole or in part, is strictly prohibited.

2. Authorized Use: Any authorized use of IrDA publications (in whole or in part) is under NONEXCLUSIVE USE LICENSE ONLY. No rights to sublicense, assign or transfer the license are granted and any attempt to do so is void.

DISTRIBUTION PRIVILEGES for IrDA MEMBERS ONLY:

IrDA Members Limited Reproduction and Distribution Privilege: A limited privilege of reproduction and distribution of IrDA copyrighted publications is granted to IrDA members in good standing and for sole purpose of reasonable reproduction and distribution to non-IrDA members who are engaged by contract with an IrDA member for the development of IrDA certified products. Reproduction and distribution by the non-IrDA member is strictly prohibited.

TRANSACTION NOTICE to IrDA MEMBERS ONLY:

Each and every copy made for distribution under the limited reproduction and distribution privilege shall be conspicuously marked with the name of the IrDA member and the name of the receiving party. Upon reproduction for distribution, the distributing IrDA member shall promptly notify IrDA (in writing or by e-mail) of the identity of the receiving party.

A failure to comply with the notification requirement to IrDA shall render the reproduction and distribution unauthorized and IrDA may take appropriate action to enforce its copyright, including but not limited to, the termination of the limited reproduction and distribution privilege and IrDA membership of the non-complying member.

TRADEMARKS:

1. Prohibitions: IrDA claims exclusive rights in its trade names, trademarks, service marks, collective membership marks and certification marks (hereinafter collectively "trademarks"), including but not limited to the following trademarks: INFRARED DATA ASSOCIATION (wordmark alone and with IR logo), IrDA (acronym mark alone and with IR logo), IR logo, IR DATA CERTIFIED (composite mark), and MEMBER IrDA (wordmark alone and with IR logo). Any unauthorized use of IrDA trademarks is strictly prohibited.

2. Authorized Use: Any authorized use of a IrDA collective membership mark or certification mark is by NONEXCLUSIVE USE LICENSE ONLY. No rights to sublicense, assign or transfer the license are granted and any attempt to do so is void.

NO REPRESENTATION of THIRD PARTY RIGHTS:

IrDA makes no representation or warranty whatsoever with regard to IrDA member or third party ownership, licensing or infringement/non-infringement of intellectual property rights. Each recipient of IrDA publications, whether or not an IrDA member, should seek the independent advice of legal counsel with regard to any possible violation of third party rights arising out of the use, attempted use, reproduction, distribution or public display of IrDA publications.

IrDA assumes no obligation or responsibility whatsoever to advise its members or non-members who receive or are about to receive IrDA publications of the chance of infringement or violation of any right of an IrDA member or third party arising out of the use, attempted use, reproduction, distribution or display of IrDA publications.

LIMITATION of LIABILITY:

BY ANY ACTUAL OR ATTEMPTED USE, REPRODUCTION, DISTRIBUTION OR PUBLIC DISPLAY OF ANY IrDA PUBLICATION, ANY PARTICIPANT IN SUCH REAL OR ATTEMPTED ACTS, WHETHER OR NOT A MEMBER OF IrDA, AGREES TO ASSUME ANY AND ALL RISK ASSOCIATED WITH SUCH ACTS, INCLUDING BUT NOT LIMITED TO LOST PROFITS, LOST SAVINGS, OR OTHER CONSEQUENTIAL, SPECIAL, INCIDENTAL OR PUNITIVE DAMAGES. IrDA SHALL HAVE NO LIABILITY WHATSOEVER FOR SUCH ACTS NOR FOR THE CONTENT, ACCURACY OR LEVEL OF ISSUE OF AN IrDA PUBLICATION.

DISCLAIMER of WARRANTY:

All IrDA publications are provided "AS IS" and without warranty of any kind. IrDA (and each of its members, wholly and collectively, hereinafter "IrDA") EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND WARRANTY OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS. IrDA DOES NOT WARRANT THAT ITS PUBLICATIONS WILL MEET YOUR REQUIREMENTS OR THAT ANY USE OF A PUBLICATION WILL BE UN-INTERRUPTED OR ERROR FREE, OR THAT DEFECTS WILL BE CORRECTED. FURTHERMORE, IrDA DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING USE OR THE RESULTS OR THE USE OF IrDA PUBLICATIONS IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN PUBLICATION OR ADVICE OF A REPRESENTATIVE (OR MEMBER) OF IrDA SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY.

LIMITED MEDIA WARRANTY:

IrDA warrants ONLY the media upon which any publication is recorded to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of distribution as evidenced by the distribution records of IrDA. IrDA's entire liability and recipient's exclusive remedy will be replacement of the media not meeting this limited warranty and which is returned to IrDA. IrDA shall have no responsibility to replace media damaged by accident, abuse or misapplication. ANY IMPLIED WARRANTIES ON THE MEDIA, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM PLACE TO PLACE.

CERTIFICATION and GENERAL:

Membership in IrDA or use of IrDA publications does NOT constitute IrDA compliance. It is the sole responsibility of each manufacturer, whether or not an IrDA member, to obtain product compliance in accordance with IrDA rules for compliance.

All rights, prohibitions of right, agreements and terms and conditions regarding use of IrDA publications and IrDA rules for compliance of products are governed by the laws and regulations of the United States. However, each manufacturer is solely responsible for compliance with the import/export laws of the countries in which they conduct business. The information contained in this document is provided as is and is subject to change without notice.

Contents

1. Introduction

1.1 Tasks, Platforms, and Goals

One of the most basic and desirable uses of the IrDA infrared communication protocols is simply to send an arbitrary “thing”, or data object, from one device to another, and to make it easy for both application developers and users to do so. We refer to this as object exchange (un-capitalized), and it is the subject of the protocol described in this document.

This document describes the current status of the protocol IrOBEX (for IrDA Object Exchange, OBEX for short). OBEX is a compact, efficient, binary protocol that enables a wide range of devicesto exchange data in a simple and spontaneous manner. OBEX is being defined by members of the Infrared Data Association to interconnect the full range of devices that support IrDA protocols. It is not, however, limited to use in an IrDA environment.

OBEX performs a function similar to HTTP, a major protocol underlying the World Wide Web. However, OBEX works for the many very useful devices that cannot afford the substantial resources required for an HTTP server, and it also targets at devices with different usage models from the Web. OBEX is enough like HTTP to serve as a compact final hop to a device “not quite” on the Web.

A major use of OBEX is a “Squirt” or “Slurp” application, allowing rapid and ubiquitous communications among portable devices or in dynamic environments. For instance, a laptop user squirts a file to another laptop or PDA; an industrial computer slurps status and diagnostic information from a piece of factory floor machinery; a digital camera squirts it pictures into a film development kiosk, or if lost can be queried (slurped?) for the electronic business card of its owner. However, OBEX is not limited to quick connect-transfer-disconnect scenarios - it also allows sessions in which transfers take place over a period of time, maintaining the connection even when it is idle.

PCs, pagers, PDAs, phones, auto-tellers, information kiosks, calculators, data collection devices, watches, home electronics, industrial machinery, medical instruments, automobiles, pizza ovens, and office equipment are all candidates for using OBEX. To support this wide variety of platforms, OBEX is designed to transfer flexibly defined “objects”; for example, files, diagnostic information, electronic business cards, bank account balances, electrocardiogram strips, or itemized receipts at the grocery store. “Object” has no lofty technical meaning here; it is intended to convey flexibility in what information can be transferred. OBEX can also be used for Command and Control functions - directives to TVs, VCRs, overhead projectors, computers, machinery.

OBEX consists of two major parts: a model for representing objects (and information that describes the objects), and a session protocol to provide a structure for the “conversation” between devices. OBEX is designed to fulfill the following major goals:

1. Application friendly - provide the key tools for rapid development of applications
2. Compact - minimum strain on resources of small devices
3. Cross platform
4. Flexible data handling, including data typing and support for standardized types - this will allow simpler devices for the user through more intelligent handling of data inside.
5. Maps easily into Internet data transfer protocols
6. Extensible - provide growth path to future needs like security, compression, and other extended features without burdening more constrained implementations.
7. Debuggable

2. OBEX Overview

2.1 OBEX components

OBEX is a protocol for sending or exchanging objects and control information. In its simplest form, it is quite compact and requires a small amount of code to implement. It can reside on top of any reliable transport, such as that provided by [IRDATTP] (including IrDA Lite implementations), or stream sockets. OBEX consists of the following pieces:

- an object model that carries information *about* the objects being sent, as well as containing the objects themselves. The object model is built entirely with parsable headers, similar in concept to the idea of headers in HTTP.
- a session protocol, which structures the dialogue between two devices. The session protocol uses a binary packet-based request/response model.
- an IAS definition and hint bits for the service.

Though not part of the OBEX standard proper, there are a number of appendices with supporting information.

2.2 Relation to other IrDA protocols

The following diagram illustrates where OBEX fits into the overall scheme of IrDA software.

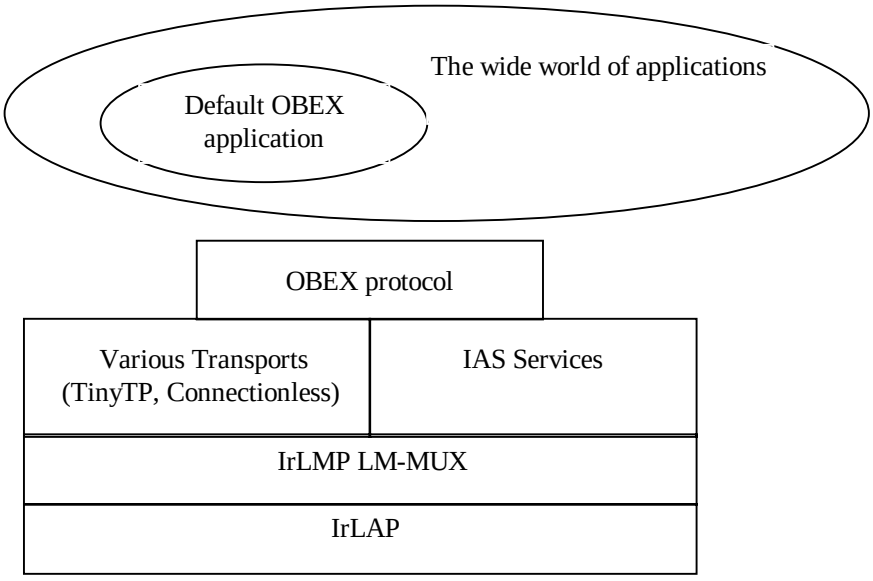

Figure 1. OBEX in the IrDA architecture

The above diagram places OBEX within the IrDA protocol hierarchy, but it could just as well appear above some other transport layer that provides reliable flow-controlled connections. Connection oriented operation over IrDA protocols uses TinyTP flow control to allow for multiple logical connections, including simultaneous OBEX client/server sessions in both directions.

When IrOBEX is used over TinyTP, the TinyTP MaxSDUSize parameter is not permitted, and segmentation and reassembly are not performed by the TinyTP layer.

2.3 OBEX applications, and the *default application*

The diagram shows a *default OBEX application*. Though not required, a default OBEX application provides a general purpose solution for sending and receiving objects, and can be extended to handle many object types with the addition of agents (software modules that know how to deal with a specific class of object). It is possible for many different applications to use IR communications through a single default OBEX service.

In a later section this document defines an IrDA IAS entry for a default OBEX server application. Having a standard IAS entry is roughly equivalent to having an assigned port number in the TCP/IP world - it allows another device to locate and connect to a service. There should be only one default OBEX server application on a device.

OBEX can also be used as a protocol within custom applications, providing the basic structure for the communication. In this case, the application may set the OBEX IAS hint bit, but it must not use the OBEX IAS entry. Instead, the custom application should define its own unique IAS entry.

While a custom application will (by definition) work best when connected to a strict peer, there are cases where it may be useful to connect to a default OBEX application. In particular, if a custom application can send any objects of general interest but cannot find a peer application, it could send the data to a default OBEX application, perhaps modifying the sending format to be a common computer format such as text, GIF, JPEG, or such.

For instance, a digital camera could look for digital film development kiosks, but also want to share photos from last summer's camping trip with any PC, PDA, or smart TV. A pager might delete messages automatically after sending them to its peer application (knowing they are completely taken care of), but send them as text objects to any OBEX enabled device just for easy viewing or sharing. Any IR device whatsoever might offer its "business card", describing who and what it is.

2.4 Specification versus Implementation

This description does not specify any implementation of the protocol; only the requirements that the implementation must embody. In particular, this specification does specify APIs at either the top or bottom boundaries. The primary OBEX context assumes an lower protocol layer that supports reliable data interchange with other devices (as provided by [IRDATTP]). An appendix discussing the use of OBEX over connectionless transports is under consideration.

3. OBEX Object Model

The object model addresses the question of how objects are represented by OBEX. The model must deal with both the object being transferred and information *about* the object. It does this by putting the pieces together in a sequence of **headers**. A header is an entity that describes some aspect of the object, such as name, length, descriptive text, or the object body itself. For instance, headers for the file jumar.txt might contain the name, a type identifier of “text”, a description saying “How to use jumars to grow better tomatoes”, the file length, and the file itself.

3.1.1 OBEX Headers

Headers have the general form:

<HI, the header ID>
<HV, the header value>

HI, the header ID, is an unsigned one byte quantity that identifies what the header contains and how it is formatted. HV consists of one or more bytes in the format and meaning specified by HI. All headers are optional - depending on the type of device and the nature of the transaction, you may use all of the headers, some, or none at all. IDs make headers parseable and order independent, and allow un-recognized headers to be skipped easily. Unrecognized headers should be skipped by the receiving device.

OBEX defines a set of headers that will used quite frequently and therefore benefit from a compact representation, and then provides a mechanism to use any HTTP header, as well as user defined headers. This small set will serve the needs of file transfer on PCs as well as the needs of many other devices, while facilitating a clean mapping to HTTP when OBEX is used as a compact final hop in a web communication.

The low order 6 bits of the header identifier are used to indicate the meaning of the header, while the upper 2 bits are used to indicate the header encoding. This encoding provides a way to interpret unrecognized headers just well enough to discard them cleanly. The length prefixed header encodings send the length in network byte order, and the length includes the 3 bytes of the identifier and length.

The 2 high order bits of HI have the following meanings (shown both as bits and as a byte value):

bits 8 and 7 of HI	Interpretation
00 (0X00)	null terminated Unicode text, length prefixed with 2 byte unsigned integer
01 (0X40)	byte stream, length prefixed with 2 byte unsigned integer
10 (0X80)	1 byte quantity
11 (0XC0)	4 byte quantity - transmitted in network byte order (high byte first)

The Header identifiers are:

HI - identifier	header name	description
0xC0	Count	Number of objects (used by Connect)
0x01	Name	name of the object (often a file name)
0x42	Type	type of object - e.g. text, html, binary, manufacturer specific
0xC3	Length	the length of the object in bytes
0x44 0xC4	Time	date/time stamp - ISO 8601 version - preferred date/time stamp - 4byte version (for compatibility only)
0x05	Description	text description of the object
0x46	Target	name of service that operation is targeted to
0x47	HTTP	an HTTP 1.x header
0x48 0x49	Body End of Body	a chunk of the object body. The final chunk of the object body
0x4A	Who	identifies the application using OBEX - used in Connect to tell if talking to a peer
0x0Bto 0x2F	reserved	this range includes all combinations of the upper 2 bits
0x30 to 0x3F	user defined	this range includes all combinations of the upper 2 bits

All allowable headers and formats thereof are listed by this table. Applications must not change the upper bits on OBEX defined headers and expect anyone else to recognize them.

3.2 Header descriptions

3.2.1 Count

Count is a four byte unsigned integer to indicate the number of objects involved in the operation. It is only used by Connect.

3.2.2 Name

Name is a null terminated Unicode text string describing the name of the object.

Example: JUMAR.TXT

Though the Name header is very useful for operations like file transfer, it is optional - the receiving application may know what to do with the object based on context, Type, or other factors. If the object is being sent to a general purpose device such as a PC or PDA, this will normally be used as the filename of the received object, so act accordingly. Receivers that interpret this header as a file name must be prepared to handle Names that are not legitimate filenames on their system.

3.2.3 Type

Type is a byte stream consisting of null terminated ASCII text describing the type of the object, such as text, binary, or vcard. Type is used by the receiving side to aid in intelligent handling of the object. This header corresponds to the content-type header in HTTP.

Whenever possible, OBEX (like HTTP) uses IANA registered media types to promote interoperability based on open standards. When a registered type is used, the HTTP canonical form for the object body must also be used - in other words, if you say a thing is of type “text/html”, it must meet all the rules for representing items of type “text/html”.

If no type is specified, the assumed type is binary, and it is up to the receiving software to deal with it as best it can. This may involve simply storing it without modification of any kind under the stated name, and/or trying to recognize it by the extension on the name. For instance, a Microsoft Word file could be

sent with no type, and the receiving software, seeing the .doc suffix could choose to interpret it as a Word file.

Though the Type header is very useful for transfer of non-file object types, it is optional - the receiving application may know what to do with the object based on context, Name, or other factors.

3.2.4 Length

Length is a four byte unsigned integer quantity giving the total length in bytes of the object. If the Length is known in advance, this header should be used. This allows the receiver to quickly terminate transfers requiring too much space, and also makes progress reporting easier.

If a single object exceeds 4 gigabytes - 1 in length, its size cannot be represented by this header. Instead an HTTP content-length header should be used, which is ASCII encoded decimal and can represent arbitrarily large values. However, implementations that cannot handle such large objects are not required to recognize the HTTP header.

The Length header is optional, because in some cases the length is not known in advance, and the end of body header will signal when the end of the object is reached.

3.2.5 Time

Time is byte stream that gives the object's UTC date/time of last modification in ISO 8601 format. The Date/Time header is optional.

Note: One notable OBEX application was released before the standard became final, and uses an unsigned 4 byte integer giving the date/time of the object's last modification in seconds since January 1, 1970. Implementers may wish to accept or send both formats for backward compatibility, but it is not required. The preferred ISO 8601 format and this format can be distinguished by the high two bits of the Header Identifier—ISO 8601 uses the text HI encoding 0x44, while this one uses the 4 byte integer HI encoding 0xC4.

3.2.6 Description

Description is a null terminated Unicode text string used to provide additional description of the object or operation. The description header is optional. The description header is not limited to describing objects. For example, it may accompany a response code to provide additional information about the response.

3.2.7 Target

Target is a byte stream that identifies the intended target of the operation. On the receiving end, object name and type information provide one way of performing dispatching - this header offers an alternate way of directing an operation to the intended recipient.

When used with the PUT operation, it allows for behavior analogous to the HTTP POST operation. It can also be combined with GET (get objects from a specific provider)

The sending device must provide this header in a form meaningful to the destination device. If this header is received but not recognized, it is up to the implementation to decide whether to accept or reject the accompanying object.

3.2.8 HTTP

HTTP is a byte stream containing an HTTP 1.x header. This can be used to include many advanced features already defined in HTTP without re-inventing the wheel. HTTP terminates lines using CRLF, which will be preserved in this header so it can be passed directly to standard HTTP parsing routines. This header is optional.

3.2.9 Body, End of Body

The body of an object (the contents of a file being transferred, for instance) is sent in one or more body headers. A “chunked” encoding helps make abort handling easier, allows for operations to be interleaved, and handles situations where the length is not known in advance, as with process generated data and on-the-fly encoding.

A **Body** header consists of the HI identifying it as an object body, a two byte header length, and all or part of the contents of the object itself.

A distinct HI value (**End of Body**) is used to identify the last chunk of the object body. In some cases the object body data is generated on the fly and the end cannot be anticipated, so it is legal to send a zero length end of body header. The end of body header signals the end of an object, and must be the final header of any type associated with that object.

3.2.10 Who

Who is a length prefixed byte stream used only in Connect so that peer applications may identify each other, typically to allow special additional capabilities unique to that application or class of device to come into play. This identifier must be unique, or trouble is guaranteed. A 16 byte UUID is one way of guaranteeing this uniqueness, but the method used is up to the implementer.

3.2.11 User Defined Headers

User defined headers allow complete flexibility for the application developer. Observe the use of the high order two bits to specify encoding, so that implementations can skip unrecognized headers. Obviously you cannot count on user defined headers being interpreted correctly except by strict peers of your application, so exercise due care at connect time before relying on these - the Who header can be used at connect time to identify strict peers.

4. Session Protocol

The session protocol describes the basic structure of an OBEX conversation. It consists of a format for the “conversation” between devices and a set of opcodes that define specific actions. The OBEX conversation occurs within the context of an OBEX connection. The connection oriented session allows capabilities information to be exchanged just once at the start of the connection, and allows state information to be kept (such as a target path for Put or Get operations).

OBEX follows a client/server **request-response** paradigm for the conversation format. The terms client and server refer to the originator/receiver of the OBEX connection, not necessarily who originated the low level transport connection. Requests are issued by the client (the party that initiates the OBEX connection). Once a request is issued, the client waits for a response from the server before issuing another request. The request/response pair is referred to as an **operation**.

In order to maintain good synchronization and make implementation simpler, requests and responses may be broken into multiple OBEX packets that are limited to a size specified at connection time. Each request packet is acknowledged by the server with a response packet. Therefore, an operation normally looks like a sequence of request/response packet pairs, with the final pair specially marked. In general, an operation should not be broken into multiple packets unless it will not fit into a single OBEX packet.

Each Request packet consists of an opcode (such as Put or Get), a packet length, and one or more headers, as defined in the object model chapter. A header must entirely fit within a packet - it may not be split over multiple packets. It is strongly recommended that headers containing the information *describing* the object (name, length, date, etc.) be sent before the object body itself. For instance, if a file is being sent, the file name should be sent first so that the file can be created and the receiver can be ready to store the contents as soon as they show up.

However, This does not mean that *all* descriptive headers must precede *any* body header - Description headers could come at any time with information to be presented in the user interface, and in the future intermediate headers may be used to distinguish among multiple parts in the object body.

The orderly sequence of **request** (from a client) followed by **response** (from a server) has a few exceptions

- An Abort operation may come in the middle of a request/response sequence. It cancels the current operation.

- A Command operation does not have a response - it is a uni-directional operation.

Each side of a communication link may have both client and server if desired, and thereby create a peer to peer relationship between applications by using a pair of OBEX sessions, one in each direction. However, it is not a requirement that a device have a both client and server, or that more than one session be possible at once. For example, a data collection device (say a gas meter on a house) might be a server only, and support only the GET operation, allowing the device to deliver its information (the meter reading) on demand. A simple file send applet on a PC might support only PUT.

4.1 Request format

Requests consist of one or more packets, each packet consisting of a one byte opcode, a two byte packet length, and required or optional data depending on the operation. Each request packet must be acknowledged by a response. Each opcode is discussed in detail later in this chapter, including the number and composition of packets used in the operation. The general form of a request packet is:

Byte 0	Bytes 1, 2	Bytes 3 to n
opcode	packet length	headers or request data

Every request packet in an operation has the opcode of that operation. The high order bit of the opcode is called the Final bit. It is set to indicate the last packet for the request. For example, a Put operation sending a multi-megabyte object will typically require many Put packets to complete, but only the last packet will have the Final bit set in the Put opcode.

As with header lengths, the packet length is transmitted in network byte order (high order byte first), and represents the entire length of the packet including the opcode and length bytes. The maximum packet length is 64K bytes - 1.

4.2 Response format

Responses consist of one or more packets - one per request packet in the operation. Each packet consists of a one byte response code, a two byte packet length, and required or optional data depending on the operation. Each response code is listed later in this chapter, and commonly used codes are discussed in the individual opcode sections. The general form of a request packet is:

Byte 0	Bytes 1,2	Bytes 3 to n
response code	response length	response data

The high order bit of the response code is called the Final bit. It is set to indicate the last packet of a particular response. The use of this bit is subtly different from its use in request packets - this bit is set for the last packet of *each* response, including all the intermediate responses made to each of the non-final request packets, in order to allow for multipacket responses in future versions of OBEX. This raises the question of whether a *client* must send responses to the *server* (reverse of the normal situation) in between packets *if a response requires multiple packets*. The answer to this question is no, but during such an exchange the server must listen for an abort operation from the client.

As with header and request lengths, the response length is transmitted in network byte order (high order byte first), and represents the entire length of the packet including the opcode and length bytes. The maximum response packet length is 64K bytes - 1, and the actual length in a connection is subject to the limitations negotiated in the OBEX connect operation.

The (optional) response data may include objects and headers, or other data specific to the request that was made. If a description header follows the response code prior to any headers with object specific information, it is interpreted as descriptive text expanding on the meaning of the response code. Detailed responses are discussed in the opcode sections below.

The response code contains the HTTP status code (a 3 digit ASCII encoded positive integer) encoded in the low order 7 bits as an unsigned integer (the code in parentheses has the Final bit set). See the HTTP document for complete descriptions of each of these codes. The most commonly used response codes are 0x90 (0x10 Continue with Final bit set, used in responding to non-final request packets), and 0xA0 (0x20 Success w/Final bit set, used at end of successful operation).

4.2.1 Response Code values

OBEX response code	HTTP status code	Definition
0x00 to 0x0F	none	reserved
0x10 (0x90)	100	Continue
0x11 (0x91)	101	Switching Protocols
0x20 (0xA0)	200	OK, Success
0x21 (0xA1)	201	Created
0x22 (0xA2)	202	Accepted
0x23 (0xA3)	203	Non-Authoritative Information
0x24 (0xA4)	204	No Content
0x25 (0xA5)	205	Reset Content
0x26 (0xA6)	206	Partial Content
0x30 (0xB0)	300	Multiple Choices
0x31 (0xB1)	301	Moved Permanently
0x32 (0xB2)	302	Moved temporarily
0x33 (0xB3)	303	See Other
0x34 (0xB4)	304	Not modified
0x35 (0xB5)	305	Use Proxy
0x40 (0xC0)	400	Bad Request - server couldn't understand request
0x41 (0xC1)	401	Unauthorized
0x42 (0xC2)	402	Payment required
0x43 (0xC3)	403	Forbidden - operation is understood but refused
0x44 (0xC4)	404	Not Found
0x45 (0xC5)	405	Method not allowed
0x46 (0xC6)	406	Not Acceptable
0x47 (0xC7)	407	Proxy Authentication required
0x48 (0xC8)	408	Request Time Out
0x49 (0xC9)	409	Conflict
0x4A (0xCA)	410	Gone
0x4B (0xCB)	411	Length Required
0x4C (0xCC)	412	Precondition failed
0x4D (0xCD)	413	Requested entity too large
0x4E (0xCE)	414	Request URL too large
0x4F (0xCF)	415	Unsupported media type
0x60 (0xE0)	500	Internal Server Error
0x61 (0xE1)	501	Not Implemented
0x62 (0xE2)	502	Bad Gateway
0x63 (0xE3)	503	Service Unavailable
0x64 (0xE4)	504	Gateway Timeout
0x65 (0xE5)	505	HTTP version not supported

4.3 OBEX Operations and Opcode definitions

OBEX operations consist of the following:

Opcode (w/high bit set)	Definition	
0x80 // high bit always set	Connect	choose your partner, negotiate capabilities
0x81 // high bit always set	Disconnect	signal the end of the session
0x02 (0x82)	Put	send an object
0x03 (0x83)	Get	get an object
0x04 (0x84)	Command	send a responseless packet
0x85 // high bit always set	SetPath	modifies the current path on the receiving side
0xFF // high bit always set	Abort	abort the current operation.
0x05 to 0x0F	Reserved	not to be used w/out extension to this specification
0x10 to 0x1F	User definable	use as you please with peer application
bit 7 of opcode means Final packet of request		
bits 5 and 6 are reserved		these bits must be set to zero

The high bit of the opcode is used as a Final bit, described in the previous sections of this chapter. Bits 5 and 6 of the opcode are reserved for future use and should be set to zero by sender and ignored by receiver.

If a server receives an unrecognized opcode, it should return 0xE1 response code (Not Implemented, with Final bit set) and ignore the operation. It may send the request to the bit bucket, save it for later analysis, or whatever it chooses.

Each operation is described in detail in the following sections.

4.3.1 Connect

This operation initiates the connection and sets up the basic expectations of each side of the link. The request format is:

Byte 0	Bytes 1 and 2	Byte 3	Byte 4	Bytes 5 and 6	Byte 7 to n
0x00	connect packet length	OBEX version number	flags	maximum OBEX packet length	optional headers

The response format is:

Byte 0	Bytes 1 and 2	Byte 3	Byte 4	Bytes 5 and 6	Byte 7 to n
response code	connect response packet length	OBEX version number	flags	maximum OBEX packet length	optional headers

The connect request and response must each fit in a single packet. Implementations are not *required* to recognize more than the first 7 bytes of these packets, though this may restrict their usage. The Connect operation is required if the OBEX session is established using connection oriented protocols such as [IRDATTP].

4.3.1.1 OBEX version number

The version numbers are the version of this OBEX specification encoded with the major number in the high order 4 bits, and the minor version in the low order 4 bits. Letters in the specification version (e.g. 0.5**a**) are not encoded in this header. See the example later in this section.

4.3.1.2 Connect flags

The flags have the following meanings:

bit	meaning
0	reserved
1	reserved
2	reserved
3	reserved
4	reserved
5	reserved
6	reserved
7	reserved

All flags are currently reserved, and must be set to zero on the sending side and ignored by the receiving side.

4.3.1.3 Maximum OBEX Packet Length

The maximum OBEX packet length is a two byte unsigned integer that indicates the maximum size OBEX packet that the device can receive. The largest acceptable value at this time is 64K bytes -1. However, even if a large packet size is negotiated, it is not required that large packets be sent - this just represents the maximum allowed by each participant. The client and server may have different maximum lengths.

There is no minimum value for OBEX packets, though common sense should be used - very small values may prevent some or all headers from being transferred, since a header must fit completely within a single packet. Missed headers may cause operations to fail. However, for some operations it will not be any problem - this issue simply needs to be evaluated carefully when building a minimal implementation.

It is permissible for an implementation to refuse to connect if the other side does not support a packet size suitable to the task. However, to encourage the widest possible interoperability, it is recommended that every implementation support a minimum packet size of 60 bytes.

4.3.1.4 Headers used in Connect

The Count, Length, and Who headers, defined in the Object Model chapter, can be used in Connect. Count and Length are used to indicate the number of objects that will be sent during this connection and their total size - this is appropriate for a “squirt” type transaction, which follows a rapid sequence of Connect-Send-Disconnect. **Who** is used to hold a unique identifier which allows applications to tell whether they are talking to a strict peer or not. Typically this is used to enable additional capabilities supplied only by an exact peer. If a Who header is used, it should be sent before any other header.

The connect request or response may include a Description header with information about the device or service. It is recommended this information be presented through the user interface on the receiving side if possible.

4.3.1.5 The Connect response

The successful response to Connect is 0xA0 (Success, with the high bit set) in the response code, followed immediately by the required fields described above, and optionally by other OBEX headers as defined above. Any other response code indicates a failure to make an OBEX connection. A fail response still includes the version, flags, and packet size information, and may include a description header to expand on the meaning of the response code value.

4.3.1.6 Example

The following example shows a connect request and response with comments explaining each component. The connect sends two optional headers describing the number of objects and total length of the proposed transfer during the connection.

Client Request:	**bytes**	**meaning**
opcode = Conn	0x80	Connect, Final bit set
	0x0011	packet length = 17
	0x10	version 1.0 of obex
	0x00	flags, all zero for this version of OBEX
	0x2000	8K is the max OBEX packet size client can accept
	0xC0	HI for Count header (optional header)
	0x00000004	four objects being sent
	0xC3	HI for Length header (optional header)
	0x0000F483	total length of hex F483 bytes
Server Response:		
Resp. Code	0xA0	Success, Final bit set
	0x0007	packet length of 7
	0x11	version 1.1 of obex
	0x00	flags
	0x0400	1K max packet size

4.3.2 Disconnect

This opcode signals the end of the OBEX session. It may include a description header for additional user readable information. The Disconnect request and response must each fit in one OBEX packet and have their Final bits set.

Byte 0	Bytes 1, 2	Bytes 3 to n
0x81	packet length	optional headers

The response to Disconnect is 0xA0 (Success), optionally followed with a description header. A Disconnect may not be refused.

Byte 0	Bytes 1,2	Bytes 3 to n
0xA0	response packet length	optional response headers

It is permissible for a connection to be terminated by closing the transport connection without issuing the OBEX disconnect operation, though this precludes any opportunity to include descriptive information about the disconnection.

4.3.3 Put

The Put operation sends one object from the client to the server. The request will normally have at least the following headers: Name and Length. For files or other objects which may have several dated versions, the Date/Time header is also recommended, while the Type is very desirable for non-file object types. However, any of these may be omitted if the receiver can be expected to know the information by some other means. For instance, if the target device is so simple that it accepts only one object and prevents connections from unsuitable parties, all the headers may be omitted without harm. However if a PC, PDA, or any other general purpose device is the intended recipient, the headers are highly recommended.

A Put request consists of one or more request packets, the last of which has the Final bit set in the opcode. The implementer may choose whether to include an object body header in the first packet, or wait until the response to the initial packet is received before sending any object body chunks.

Byte 0	Bytes 1, 2	Bytes 3 to n
0x02 (0x82 when Final bit set)	packet length	sequence of headers

Each packet is acknowledged by a response from the server as described in the general session model discussion above.

Byte 0	Bytes 1,2	Bytes 3 to n
response code typical values: 0x90 for continue 0xA0 for Success	response packet length	optional response headers

4.3.3.1 Headers used in Put

Any of the headers defined in the Object model chapter can be used with Put except for Count and Who. These might include Name, Type, Description, Length, or HTTP headers specifying compression, languages, character sets, and so on. It is strongly recommended that headers describing the object body precede the object body headers for efficient handling on the receive side. If the Name header is used, it should be first, and if Type header is used it must precede all object body headers.

4.3.3.2 Put Response

The response for successfully received intermediate packets (request packets without the Final bit set) is 0x90 (Continue, with Final bit sent). The successful final response is 0xA0 (Success, with Final bit set). The response to any individual request packet must itself consist of just one packet with its Final bit set - multi-packet responses to Put are not permitted.

Any other response code indicates failure. If the length field of the response is > 3 (the length of the response code and length bytes themselves), the response includes headers, such as a description header to expand on the meaning of the response code value.

Here is a typical Final response:

Component	**bytes**	**meaning**
response code	0xA0	// success, Final bit set
// packet len	0x0003	

4.3.3.3 Put Example

For example, here is a Put operation broken out with each component (opcode or header) on a separate line. We are sending a file called jumar.txt, and for ease of reading, the example file is 4K in length and is sent in 1K chunks.

Client Request:	**bytes**	**meaning**
opcode = Put	0x02	Put, Final bit not set
	0x0422	1058 bytes is length of packet
	0x01	HI for Name header
	0x0017	Length of Name header (Unicode is 2 bytes per char)
	JUMAR.TXT	name of object, null terminated Unicode
	0xC3	HI for Length header
	0x00001000	Length of object is 4K bytes
	0x48	HI for Object Body chunk header

	0x0403 0x.....	Length of body header (1K) plus HI and header length 1K bytes of body
Server Response:		
Resp. Code	0x90 0x0003	Continue, Final bit set length of response packet
Client Request:		
opcode = Put	0x02 0x0406 0x48 0x0403 0x.....	Put, Final bit not set 1030 bytes is length of packet HI for Object Body chunk Length of body header (1K) plus HI and header length next 1K bytes of body
Server Response:		
Resp. Code	0x90 0x0003	Continue, Final bit set length of response packet

Another packet containing the next chunk of body is sent, and finally we arrive at the last packet, which has the Final bit set.

Client Request:		
opcode = Put	0x82 0x0406 0x49 0x0403 0x.....	Put, Final bit set 1030 bytes is length of packet HI for final Object Body chunk Length of body header (1K) plus HI and header length next 1K bytes of body
Server Response:		
Resp. code	0xA0 0x0003	Success, Final bit sent

4.3.3.4 Server side handling of Put objects

Servers may do whatever they wish with an incoming object - the IrOBEX protocol does not require any particular treatment. The client may “suggest” a treatment for the object through the use of the Target and Type Headers, but this is not binding on the server. Some devices may wish to control the path at which the object is stored (i.e. specify directory information such as C:\bin\pizza.txt rather than just pizza.txt). Path information is transferred using the SetPath operation, but again, this is not binding on the server.

4.3.4 Get

The **Get** operation requests that the server return an object to the client. The request is normally formatted as follows:

Byte 0	Bytes 1, 2	Bytes 3 to n
0x03 the final bit is not used in GET	packet length	sequence of headers starting with Name

The Name header can be omitted if the server knows what to deliver, as with a simple device that has only one object (e.g. a maintenance record for a machine). If the server has more than one object that fits the request, the behavior is system dependent, but it is recommended that the server return Success with the “default object” which should contain information of general interest about the device.

The final bit is used in a GET request to identify the last packet containing headers describing the item being requested, and signaling the server to start sending the object back.

A successful response for an object that fits entirely in one response packet is 0xA0 (Success, with Final bit set) in the response code, followed by the object body. If the response is large enough to require multiple GET requests, only the last response is 0xA0, and the others are all 0x90 (Continue). The object is returned as a sequence of headers just as with PUT. Any other response code indicates failure. common non-success responses include 0xC0 bad request, and 0xC3 forbidden. The response may include a description header (before the returned object, if any) to expand on the meaning inherent in the response code value.

Byte 0	Bytes 1,2	Bytes 3 to n
response code	response packet length	optional response headers

A typical multi-step GET operation proceeds as follows: the client sends a Get request that may include a Name header; server responds with 0x90 (Continue) and headers describing the name and size of the object to be returned. Seeing the Continue response code, the client sends another GET request (with final bit set and no new headers) asking for additional data, and the server responds with a response packet containing more headers (probably Body Headers) along with another Continue response code. As long as the response is Continue, The client continues to issue GET requests until the final body information (in an End of Body header) arrives, along with the response code 0xA0 Success.

4.3.4.1 The default GET object

It is recommended that every IR device (that can afford to support even a simple Get) respond to a Get that has a zero length Name header by returning its “default object”, which should be a text file, html file, or text based electronic business card that describes or identifies the device. Recommended information includes the device owner and contact information, model number and generic description.

4.3.5 Command

Command is unlike other OBEX operations in that it does not require (or allow) a response. It simply allows data to be pushed through, even in the middle of another operation. All semantics for Command are supplied by the application using OBEX. The body of this packet consists of headers (often user defined headers).

Byte 0	Bytes 1, 2	Bytes 3 to n
0x04 0x84 when Final bit is set	packet length	headers

4.3.6 Abort

The Abort request is used when the client decides to a terminate a multipacket operation (such as Put) before it would normally end. Abort does not apply to any information sent in a Command packet - all Command semantics, including termination of operations carried in Command packets, is supplied by layers above OBEX.

The Abort request and response each always fit in one OBEX packet and have the Final bit set. An Abort operation may include headers for additional information, such as a description header giving the reason for the abort.

Byte 0	Bytes 1, 2	Bytes 3 to n
0xFF	packet length	optional parameters

The response to Abort is 0xA0 (success), indicating that the abort was received and the server is now resynchronized with the client. If anything else is returned, the client should disconnect.

Byte 0	Bytes 1,2	Bytes 3 to n
0xA0	response packet length	optional response headers

4.3.7 SetPath

The SetPath operation is used to set the "current directory" on the receiving side in order to enable transfers that need additional path information. For instance, when a nested set of directories is sent between two machines, SetPath is used to create the directory structure on the receiving side. The Path name is contained in a Name header.

Byte 0	Bytes 1 and 2	Byte 3	Byte 4	Byte 5 to n
0x85	packet length	flags	constants	Name header, optional headers

Byte 0	Bytes 1, 2	Bytes 3 to n
response code	response packet length	optional response headers

Servers are not required to store objects according to SetPath request, though it is certainly useful on general purpose devices such as PCs or PDAs. If they do not implement SetPath, they may return C0 (Bad Request) or C3 (Forbidden), and the client may decide whether it wishes to proceed.

4.3.7.1 Flags

The flags have the following meanings:

bit	meaning
0	backup a level before applying name (equivalent to ../ on many systems)
1	reserved
2	reserved
3	reserved
4	reserved
5	reserved
6	reserved
7	reserved

The unused flag bits must be set to zero by sender and ignored by receiver.

4.3.7.2 Constants

The constants byte is entirely reserved at this time, and must be set to zero by sender and ignored by receiver.

4.3.7.3 Name header

Allowable values for the Name header are:

- <name> - go down one level into this directory name, relative to the current directory
- <null> - reset to the default directory

In addition, the name header may be omitted when flags or constants indicate the entire operation being requested (for example, back up one level, equivalent to "cd .." on some systems.

The receiving side normally starts out in a default inbox/outbox directory. When sending a directory, the client will start by sending a SetPath request containing the name of the directory. For example, if the full path for the directory on a PC is C:\fred\notes, the client will send “notes” as the Name header of a SetPath operation. A PC server would create or switch to the “notes” directory relative to the default inbox directory (or perhaps create a mangled name if “notes” already exists and that was the preferred behavior). As subdirectories of “notes” are encountered by the sending routine, additional SetPath requests with the subdirectory names will be sent. As directories are completed (all objects in the directory sent, the client will send a SetPath (generally with no Name header) and the “back up a level” flag set.

5. IrDA OBEX IAS entries and service hint bit

5.1 IAS entry

A default IrDA OBEX server application must have an IAS entry with the following attribute.

Class OBEX {
Integer IrDA:TinyTP:LsapSel IrLMP LSAP selector, legal values from Ox00 to 0x6F
}

There should be only one such entry on a system, or it may be impossible for an incoming connection to decide who to connect to except by flipping a coin.

The OBEX classname should be used only by implementations that are intended as default IrOBEX servers, and in particular that have some provision for dealing with multiple object types. Special purpose programs that use this protocol should define unique IAS classnames that they alone will use. However, there is one very common subset of object exchange application - applications that are strictly for file exchange. They should use a classname of OBEX:IrXfer.

A general purpose OBEX implementation should look first for an OBEX IAS object, and if not found then look for an OBEX:IrXfer entry if the object can reasonably be represented as a file. Similarly, a special purpose program using OBEX should look first for a strict peer using whatever classname the peer normally uses, and if not found should look for the default OBEX server to accept the transfer.

5.1.1 IrDA:TinyTP:LsapSel

This attribute contains the TinyTP/IrLMP MUX channel number for the service. This attribute must be present for connection oriented use, whatever the classname is.

5.2 Service Hint bits

The OBEX IrLMP service hint bit has a value of 0x20 in the second hint byte. See section 3.4.1.1 in [IRDALMP] specification for the definition of service hint bits, and [IRDAIAS] for the complete listing of service hint bits.

6. Appendices

6.1 Appendix 0: Minimum level of service

Almost all elements of OBEX are optional, to allow resource constrained implementations to do the bare minimum while allowing reasonably rich interactions between more capable devices. Variations on this theme will be illustrated in the examples below. The only definite requirements are that connection oriented versions of OBEX (such as those running over the IrDA TinyTP protocol) must use a Connect operation. Obviously at least one other operation will be needed to have a useful application.

A minimal default OBEX server application would support Connect and either Put or Get.

6.2 Extending OBEX

The headers, opcodes, flags, and constants defined for OBEX have ranges reserved for future use and ranges for user defined functions. Reserved ranges must not be used without official extensions to this specification. Please contact the authors or IrDA if you have proposed extensions with broad applicability. As a general rule, reserved flags or constants should be set to zero by senders and ignored by receivers.

User defined opcodes and headers are freely available for any kind of use. Obviously they are only likely to make sense if both sides of the connection interpret them the same way, so the sides must somehow identify themselves to ensure compatibility. Recommended methods using the IrDA transport protocols are to use unique IAS Class names, or optional headers (in particular the Who header) in the connect packet.

When object Type headers are used, values should be IANA registered types when available, and when not available the value should have an excellent chance of being unique in order to avoid confusion (e.g. “thing” is not a very safe object type, but “Lupine_Design_zinger_ver1.0_thing” is pretty safe).

6.3 Appendix 1: Examples

6.3.1 Simple Put - file/note/ecard transfer

This example describes a simple Put operation, a scenario with many applications. It illustrates the basic request/response cycle, the exchange of version and capability information at the start of the connection, and the use of the Name and Length headers in the Put operation.

Ms. Manager worked up an outline for a presentation last night on her laptop, and needs to give it to a staffer to expand on. She sets her laptop down next to an infrared adapter attached to the staffer’s desktop machine, drags the Word document containing the outline to her OBEX app, and ...

A connection is made after her OBEX client queries staffer’s IAS to find the required LSAP of the staffer’s OBEX application. In the first two examples, we will show the transaction byte by byte. For ease of reading, the file and packet sizes are kept simple - file data is sent 1K at a time.

Client Request:	**bytes**	**meaning**
opcode = Conn	0x80	Connect, Final bit set
	0x0007	7 bytes is length of packet
	0x10	version 1.0 of obex
	0x01	flags
	0x2000	8K max packet size
Server Response:		

Resp. Code	0xA0	Success, Final bit set
	0x07	packet length of 7
	0x11	version 1.1 of obex
	0x00	flags
	0x0400	1K max packet size
Client Request:	**bytes**	**meaning**
opcode = Put	0x02	Put, Final bit not set
	0x0422	1058 bytes is length of packet
	0x01	HI for Name
	0x000D	Length of Name header
	THING.DOC	name of object, null terminated
	0xC3	HI for Length
	0x00006000	Length of object is 0x6000 bytes
	0x48	HI for Object Body chunk
	0x0403	Length of body header (1K) plus HI and header length
	0x.....	1K bytes of body
Server Response:		
Resp. Code	0x90	Continue, Final bit set
	0x0003	length of response packet
Client Request:		
opcode = Put	0x02	Put, Final bit not set
	0x0406	1030 bytes is length of packet
	0x48	HI for Object Body chunk
	0x0403	Length of body header (1K) plus HI and header length
	0x.....	next 1K bytes of body
Server Response:		
Resp. Code	0x90	Continue, Final bit set
	0x0003	length of response packet

A number of packets containing chunk of file body are sent, and finally we arrive at the last packet, which has the Final bit set.

Client Request:		
opcode = Put	0x82	Put, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for final Object Body chunk
	0x0403	Length of body header (1K) plus HI and header length
	0x.....	next 1K bytes of body
Server Response:		
Resp. code	0xA0	Success, Final bit sent
	0x0003	

The transaction is complete, so the OBEX client disconnects. 3 seconds have passed, and Ms. Manager heads down the hall to a meeting. No Type header was used, so server assumes binary file and stores exactly as is.

6.3.2 Simple Get - field data collection

This example illustrates a Get operation with user defined headers, in this case for application specific password and version information.

A meter keeps tabs on electricity used, and is intermittently visited by a meter reader. The meter reader points a collection device at the meter and presses a button which causes the following:

Client Request:	**bytes**	**meaning**
opcode = Conn	0x80	Connect, Final bit set
	0x001A	26 bytes is length of packet
	0x10	version 1.0 of obex
	0x00	no special flags
	0x0800	2K max packet size
	0x70	HI for a user defined, length prefixed header
	0x0013	length of header
	XUseElectricityX	actual contents of user defined header
Server Response:		
Resp. Code	0xA0	Success, Final bit set
	0x000B	packet length of 11
	0x11	version 1.1 of obex
	0x00	no flags set
	0x0040	64 byte max packet size
	0xF0	HI for user defined 4 byte header
	0x00000603	meter version info
Client Request:	**bytes**	**meaning**
opcode = Get	0x83	Get, Final bit set
	0x0003	length of Get packet
Server Response:		
Resp. Code	0xA0	Success, Final bit set
	0x0038	length of Get response packet
	0x49	HI for final Object Body chunk
	0x0035	Length of body header
	0x.....	0x32 bytes of meter information

A security code (“XUseElectricityX”) was passed in a user defined header during connection to keep prying eyes at bay. The Get operation specified no name or other headers, so the meter returned a reading. A header entry at this point might have instructed the meter to return service information. No type or name headers were needed on the returned object because the client application knows just what it is getting.

6.3.3 Combined Get and Put - paying for the groceries

Now that you have the idea of what the byte stream looks like, we will represent the remaining examples in a more readable format.

Smiling vacantly, the checkout clerk says “$45.12, please”. You take out your bit-fold (electronic bill-fold). You point it at the IR window at the checkstand, and press the “Do-it” key. An encrypted IR connection is made to the register. During the connection negotiation, you discover that the store takes Visa and the local bank’s debit card.

Request:
<opcode=Connect>
version info
encryption information

Response:
<response code = success>
version and capabilities info
encryption information
Accepted-payment-forms: Visa, MasterCard, ...

Request:
<opcode=Get> // Get with no arguments tells register to return itemized amount

Response:
<response code = success>
Type: text/itemized-receipt
Len: 2562 // wow, you bought a lot of items

<body of receipt in form standardized by finance industry.
It includes date, a UUID for the transaction, and the total amount due>

You examine the list of items, making sure you got the sale price on asparagus, and make sure your coupons were deducted. You press the Accept key, and select Visa from the offered payment choices...

Request:
<opcode=Put>
Type: payment/Visa
Len: 114
Payment-id: <UUID of payment sent to you in receipt>

<body of standardized Visa Payment>

The cash register chimes pleasantly as your payment is verified.

Response:
<response code = success>
Type: text
Len: 123

<hey, it says you got some frequent flyer miles!>

Connection disconnects, you pocket the bit-fold and head out past the stacks of dog food and charcoal briquets. At home you set the bitfold down by your PC and press the “Reconcile” key, and it connects to your PC, sends the days purchases with itemized receipts over to the OBEX server. With a home accounting program you make short work of the financial record keeping, since the data is all entered for you, free of errors. Of course, it is harder to hide the purchase of the Twinkies from your spouse...

6.4 Appendix 2: Proposed Additions to OBEX

1. Add flexible acking, permitting the client to specify which packets it wants a response on. This permits faster operation by reducing the necessity to turn the IR link around frequently.
2. Add single byte encoding for Type header, covering common types
3. Add Dir operation - (like HTTP head, returns object metainformation)
4. Add Offer operation - enables receiving side to preview and approve proposed objects
5. Add Negotiate operation - in-band negotiation

6. Add content negotiation - all sides to propose multiple options and have other side select
7. Add security/password/encryption
8. Add compression hooks
9. Create object type companion document discussing common object types

6.5 Appendix 3: Using OBEX over IrDA Ultra-Lite (Connectionless use)

OBEX is constructed to take advantage of the benefits of a connection oriented transport, for instance by exchanging capabilities and version information just once in the Connect packet. However, for transfers of small objects OBEX can be useful in a connectionless environment, such as that provided by the IrDA UltraLite proposals. A number of issues arise and are discussed in the following sections.

6.5.1 assuring reliable delivery

- CRCs
- Out of band feedback (visual/audible)

6.5.2 how long to wait for a response

- In the range of 3 to 5 seconds, optionally totally at application discretion.
- Allow responseless operation for Put (unidirectional link)

6.5.3 exchanging capabilities

- Put all version/capability items in headers, allow them in Put/Get packets.

6.5.4 condensing operations into a minimal exchange

- streaming multiple small objects in one packet.
- No Connect or Disconnect - just go straight to Put/Get

6.6 Appendix 4: References

IRDALAP	Serial Infrared Link Access Protocol, IrLAP, Version 1.1, Infrared Data Association
IRDALMP	Link Management Protocol, IrLMP, Version 1.1, Infrared Data Association
IRDACOM	Serial and parallel port emulation, IrCOMM, Version 1.0, Infrared Data Association
IRDATTP	Tiny Transport Protocol, TinyTP, Version 1.0, Infrared Data Association
IRDAIAS	IrLMP Hint Bit Assignments and Known IAS Definitions, Ver 1.0, IrDA
HTTP1.1	HTTP v1.1, HTTP 1.1 working group
IANAREG	IANA media type registry .....
MIME	Multipurpose Internet Mail Extensions .....