IrDA Object Exchange Protocol

OBEX™

Version 1.5

August 14. 2009

iAnywhere Solutions, Inc.
Extended Systems
Microsoft Corporation

Copyright ©, 2009 Infrared Data Association® (IrDA®)

Petitioner has added a romanette to page (i). Otherwise, it leaves the original page numbering.

MICROSOFT CORP.
EXHIBIT 1278

Authors:
Pat Megowan, Dave Suvak, Doug Kogan (Extended Systems, Inc.)
Kevin Hendrix (iAnywhere Solutions, Inc.)

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:
Doug Kogan (Extended Systems, Inc.)
Kevin Hendrix (iAnywhere Solutions, Inc.)

Document Status: Version 1.5
Major changes from Version 1.4 to 1.5:

- Incorporate Errata from June 2009

Major changes from Version 1.3 to 1.4:

- Incorporate Errata from April 2003
- Incorporate Errata from July 2007

Major changes from Version 1.2 to 1.3 :

- Incorporate Errata from October 1999, January, April and July 2000 IrDA meetings.
- Incorporate Errata from January 2001 IrDA meeting.
- Incorporate Errata from November and December 2002.
- Incorporate *Session Capabilities in OBEX*™ version 0.14 document.
- Removal of the OBEX Test Specification section in favor of the complete *OBEX Test Specification* version 1.0 document.

Major changes from Version 1.1 draft to 1.2:

- Incorporate the OBEX Errata approved at the January 1999 IrDA meeting.

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 Lawrence Faulkner at (925) 944-2930 or faxing requests to (925) 943-5600.

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 ... 9

1.1 Tasks, Platforms, and Goals ... 9
1.2 OBEX components ... 9
1.2.1 OBEX Session Protocol ... 10
1.2.2 OBEX Application Framework ... 11
1.3 Relation to other IrDA protocols ... 12
1.4 Specification versus Implementation ... 12
1.4.1 Mapping OBEX packets to TinyTP/IrLMP packets ... 12
1.5 References ... 13

2. OBEX OBJECT MODEL ... 14

2.1 OBEX Headers ... 14
2.2 Header descriptions ... 16
2.2.1 Count ... 16
2.2.2 Name ... 16
2.2.3 Type ... 16
2.2.4 Length ... 16
2.2.5 Time ... 17
2.2.6 Description ... 17
2.2.7 Target ... 17
2.2.8 HTTP ... 17
2.2.9 Body, End-of-Body ... 18
2.2.10 Who ... 18
2.2.11 Connection Identifier ... 18
2.2.12 Application Request-Response Parameters ... 19
2.2.13 Authenticate Challenge ... 19
2.2.14 Authenticate Response ... 19
2.2.15 Creator ID ... 19
2.2.16 WAN UUID ... 20
2.2.17 Object Class ... 20
2.2.18 Session-Parameters ... 20
2.2.19 Session-Sequence-Number ... 22
2.2.20 Action Identifier ... 23
2.2.21 DestName ... 23
2.2.22 Permissions ... 23
2.2.23 Single Response Mode (SRM) ... 24
2.2.24 Single Response Mode (SRM) Parameters ... 25
2.2.25 User Defined Headers ... 28

3. SESSION PROTOCOL ... 29

3.1 Request format ... 31
3.2 Response format ... 31
3.2.1 Response Code values ... 32
3.3 OBEX Single Response Mode (SRM) ... 33
3.4 OBEX Operations and Opcode definitions ... 35
3.4.1 Connect ... 35
3.4.2 Disconnect ... 38
3.4.3 Put ... 39
3.4.4 Get ... 41
3.4.5 Abort ... 42
3.4.6 SetPath ... 42
3.4.7 Session ... 43
3.4.8 Action ... 50

3.5 Packet Timeouts **53**
3.6 Authentication Procedure **53**
3.6.1 Digest Challenge 54
3.6.2 Digest Response 55
3.6.3 Hashing Function 56
3.6.4 Authentication Examples 56
3.7 Multiplexing with OBEX **58**
3.7.1 Connections multiplexed at the OBEX Command level 58
3.7.2 Connections multiplexed at the OBEX Transport layer 59

4. OBEX APPLICATION FRAMEWORK **60**

4.1 The Default OBEX Server **60**
4.2 The Inbox Service **60**
4.3 Inbox Connection **61**
4.4 Capability Service **61**
4.5 Custom OBEX applications **61**
4.6 Directed Operations and Connections **62**
4.6.1 Directed Connection Mechanics 62
4.6.2 Target Header Processing 62
4.7 OBEX Access Methods **62**
4.7.1 File Access 63
4.7.2 Database Access 63
4.7.3 Process / RPC 63

5. USING OBEX OVER IRDA ULTRA-LITE (CONNECTIONLESS USE) **64**

6. IRDA OBEX IAS ENTRIES, SERVICE HINT BIT AND TCP PORT NUMBER **65**

6.1 IAS entry **65**
6.1.1 IrDA:TinyTP:LsapSel 65
6.2 Service Hint bits **65**
6.3 TCP port number **65**

7. OBEX EXAMPLES **66**

7.1 Simple Put - file/note/ecard transfer **66**
7.2 Simple Get - field data collection **67**
7.3 Example Get of the Capability Object **68**
7.4 Connect using Target, Who and Connection Id headers **69**
7.5 Combined Get and Put - paying for the groceries **70**
7.6 Create Session Followed by a Directed Connection **71**
7.7 Suspend a Session **72**
7.8 Immediate Suspend Session during an Operation **72**
7.9 Resume a Session **73**
7.10 Resume Session for an interrupted PUT operation using SRM **74**
7.11 Resume Session for an interrupted GET operation using SRM **75**
7.12 Close a Session **76**
7.13 Negotiate a Timeout **77**
7.14 SRM during a Single PUT Operation **77**
7.15 CONNECT for Single Response Mode (SRM) **78**
7.15.1 Client Initiates Single Response Mode 78
7.15.2 Server Initiates Single Response Mode 78
7.16 PUT during Single Response Mode (SRM) **79**
7.17 GET during Single Response Mode (SRM) **80**
7.18 PUT and GET during Single Response Mode (SRM) **81**
7.19 Abort a PUT operation during Single Response Mode (SRM) **82**
7.20 Abort a GET operation during Single Response Mode (SRM) **82**
7.21 Abort of PUT operation while GET outstanding during Single Response Mode (SRM)84

7.22 PUT during Single Response Mode (SRM) using SRM Parameters Header ... 86

8. OBEX SERVICES AND PROCEDURES ... 87

8.1 Folder Browsing Service ... 87
8.1.1 Exchanging Folder Listings ... 87
8.1.2 Navigating Folders ... 88
8.1.3 Security ... 88
8.2 Simple OBEX Put file transfer (+ SetPath) ... 89
8.3 Telecom/IrMC Synchronization Service ... 89
8.4 OBEX Get Default Object ... 89
8.4.1 Get default vCard example ... 90
8.5 Capability Service ... 90
8.5.1 The Capability Object/Database ... 90
8.5.2 The Object Profile Database ... 91
8.5.3 Locating the Capability Service ... 91

9. OBEX OBJECTS ... 94

9.1 The Folder Listing Object ... 94
9.1.1 Element Specification ... 94
9.1.2 Folder Listing Details ... 97
9.1.3 Encoding Folder Listing Objects ... 98
9.1.4 XML Document Definition ... 99
9.2 Generic File Object ... 100
9.2.1 Introduction ... 100
9.2.2 Commonly Used Headers ... 100
9.2.3 Response Codes Commonly Used in File Exchange ... 100
9.2.4 Example Put Exchange ... 100
9.3 The Capability Object ... 101
9.3.1 The Capability Container (Capability) ... 102
9.3.2 General Information Container (General) ... 102
9.3.3 Inbox Container (Inbox) ... 103
9.3.4 Service/Application List ... 104
9.3.5 Requesting the Capability Object ... 105
9.3.6 Capability Object DTD ... 105
9.3.7 Capability Object Example ... 106
9.4 The Object Profile Object ... 107
9.4.1 Creating an Object Profile ... 108
9.4.2 Object Profiles ... 108
9.4.3 Object Profile Example ... 109

10. SESSION CAPABILITIES IN OBEX ... 110

10.1 OBEX Session Overview ... 110
10.2 Reliable OBEX Session Overview ... 111
10.3 Session Context ... 113
10.4 Persistence ... 113
10.5 Multiple Transport Support Proposal ... 113

11. OBEX TESTING REQUIREMENTS ... 115

11.1 Symbols and Conventions ... 115
11.2 OBEX Roles ... 115
11.3 IrDA-Specific Features ... 115
11.4 OBEX Features ... 115
11.5 Required Behavior ... 115

12. APPENDICES ... 117

12.1 Minimum level of service ... 117

12.2 Extending OBEX 117
12.3 Proposed Additions to OBEX 117
12.4 Known Target Identifiers 117
12.5 MD5 Algorithm for Authentication 117

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 OBEX (for IrDA Object Exchange, OBEX for short). OBEX is a compact, efficient, binary protocol that enables a wide range of devices to 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 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 "Push" or "Pull" application, allowing rapid and ubiquitous communications among portable devices or in dynamic environments. For instance, a laptop user pushes a file to another laptop or PDA; an industrial computer pulls status and diagnostic information from a piece of factory floor machinery; a digital camera pushes its pictures into a film development kiosk, or if lost can be queried (pulled) 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, printers, cameras, auto-tellers, information kiosks, calculators, data collection devices, watches, home electronics, industrial machinery, medical instruments, automobiles, 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, and machinery. Finally, OBEX can be used to perform complex tasks such as database transactions and synchronization.

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 devices to be simpler to use via 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. Testable and Debuggable.

1.2 OBEX components

The OBEX specification consists of two major parts: a protocol and an application framework. The OBEX protocol is a session level protocol that specifies the structure for the conversation between devices. It

also contains a model for representing objects. The OBEX application framework is built on top of the OBEX protocol. Its main purpose is to facilitate interoperability between devices using the OBEX protocol. Both of these are discussed in more detail below.

1.2.1 OBEX Session Protocol

The OBEX protocol consists of two major elements: 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 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 IrDA Tiny TP [IRDATTP] (including IrDA Lite implementations), or TCP/IP 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 headers in HTTP.

- A session protocol, which structures the dialogue between two devices. The session protocol uses a binary packet-based client/server request-response model.

- An IAS definition and hint bits for the service.

1.2.2 OBEX Application Framework

The OBEX application framework is necessary to ensure interoperability between devices using OBEX. It puts a structure on top of the OBEX protocol. The application framework is the foundation for a set of standard OBEX services that satisfy many object exchange requirements. OBEX implementations are not required to follow the conventions specified by the application framework but doing so will ensure interoperability with other devices. The table below outlines the elements of the OBEX application framework.

Element	Description
OBEX Client	An OBEX Client is the entity that initiates the underlying transport connection to an OBEX server and initiates OBEX operations.
OBEX Server	An OBEX Server is the entity that responds to OBEX operations. The OBEX server waits for the OBEX client to initiate the underlying transport connection.
Default OBEX Server	The Default OBEX server is the server that resides at the LSAP-Sel specified in the OBEX IAS definition. Other OBEX servers can exist but the Default OBEX server is the "well known" server. This is analogous to the HTTP server located at TCP port number 80.
OBEX Transport Connection	The OBEX transport connection is the underlying transport connection, which carries the OBEX protocol.
OBEX Connection	An OBEX Connection is a virtual binding between two applications or services. An OBEX connection is initiated by sending an OBEX **CONNECT** packet. Once a connection is established all operations sent over the connection are interpreted in a continuous context.
Directed Connection	A directed connection is one where the OBEX **CONNECT** packet contains targeting information which the OBEX protocol uses to connect the client to its intended service or application.
The Inbox Connection	The inbox connection is the OBEX connection made to the default OBEX server, where the OBEX **CONNECT** packet does not contain targeting information. A number of services can be accessed via the inbox connection. These services are described later.
Inbox	The inbox is the intended recipient of a client push operation over the Inbox Connection. The inbox does not have to be an actual storage location. It is really a method for encapsulating the concept that the client pushes an object to a recipient without the need to understand the details of how the recipient stores the object.
Application	An OBEX application communicates using a proprietary method known only by the manufacturer. Such applications can only expect to be understood by exact peers. Alternatively, an application may be a service with proprietary extensions. In this case the application must know if it is communicating with a service or application peer.
Service	An OBEX service communicates using procedures specified in a publicly available standard. Such as in IrMC or this specification.
Capability Service	The capability service is used to find information about the OBEX server including device information, types of objects supported, object profiles and supported applications.

1.3 Relation to other IrDA protocols

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

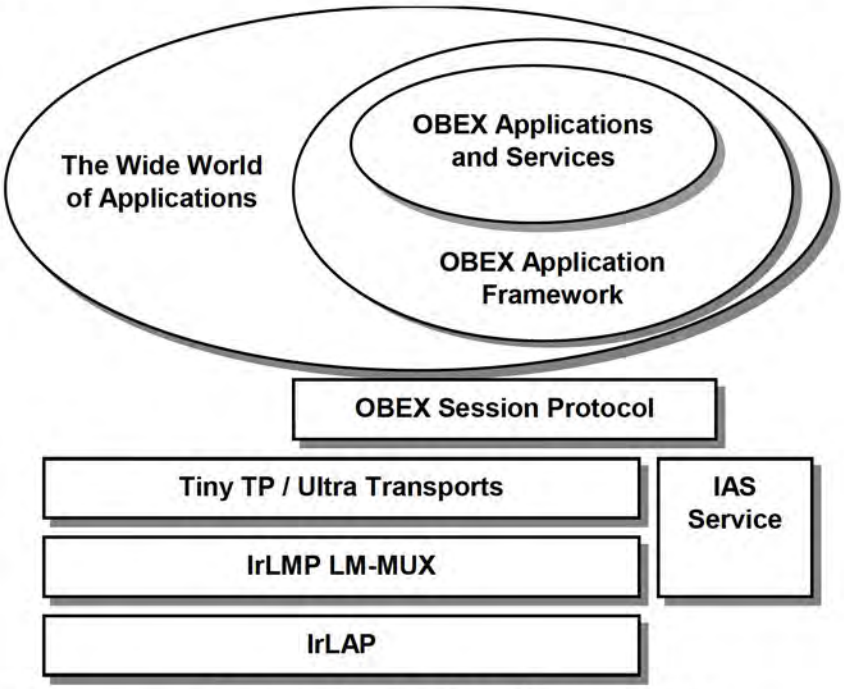


Figure 1. OBEX in the IrDA architecture

The above figure 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 Tiny TP flow control to allow for multiple logical connections, including simultaneous OBEX client/server sessions in both directions.

1.4 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 not specify APIs at either the top or bottom boundaries. The primary OBEX context assumes a lower protocol layer that supports reliable data interchange with other devices (as provided by [IRDATTP]).

1.4.1 Mapping OBEX packets to TinyTP/IrLMP packets

There is no requirement on the alignment of OBEX packets within TinyTP or IrLMP PDUs (packets). Except that all OBEX data shall be carried in data packets, not in TinyTP/IrLMP Connect or Disconnect packets.

OBEX specifically does not use Tiny TP segmentation and reassembly. There is no relationship between OBEX packets and TTP-SDUs. Therefore, the TinyTP Connect packet should not use the MaxSduSize parameter.

1.5 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 1.1, 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
IRMC	Infrared Data Association Specifications for Ir Mobile Communications (IrMC) Version 1.1
DATES	ISO 8601:1988, Data elements and interchange formats - Information interchange - Representation of dates and times
LANG	ISO 639:1988 (E/F), Code for the representation of names of languages

2. 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.

2.1 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 unrecognized headers to be skipped easily. Unrecognized headers should be skipped by the receiving device.

OBEX defines a set of headers that are used quite frequently and therefore benefit from a compact representation. It also 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. For Unicode text, the length field (immediately following the header ID) includes the 2 bytes of the null terminator (0x00, 0x00). Therefore the length of the string "Jumar" would be 12 bytes; 5 visible characters plus the null terminator, each two bytes in 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 sequence, 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 – 4 byte 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 OBEX application, used to tell if talking to a peer
0xCB	Connection Id	An identifier used for OBEX connection multiplexing
0x4C	App. Parameters	Extended application request & response information
0x4D	Auth. Challenge	Authentication digest-challenge
0x4E	Auth. Response	Authentication digest-response
0xCF	Creator ID	Indicates the creator of an object
0x50	WAN UUID	Uniquely identifies the network client (OBEX server)
0x51	Object Class	OBEX Object class of object
0x52	Session-Parameters	Parameters used in session commands/responses
0x93	Session-Sequence-Number	Sequence number used in each OBEX packet for reliability
0x94	Action Id	Specifies the action to be performed (used in ACTION operation)
0x15	DestName	The destination object name (used in certain ACTION operations)
0xD6	Permissions	4-byte bit mask for setting permissions
0x97	Single Response Mode	1-byte value to setup Single Response Mode (SRM)
0x98	Single Response Mode Parameters	1-byte value for setting parameters used during Single Response Mode (SRM)
0x19 to 0x2F	Reserved for future use	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. Note that the header identifiers are numbered in order, starting with zero. The high order bits which specify the encoding obscure this linear sequence of header numbering.

Certain headers like **Body** are expected to be present repeatedly, however headers like **Name** and **Time** do not necessarily make sense when sent multiple times. The behavior by the recipient of multiple non-**Body** headers is not defined by the protocol.

All Unicode headers are encoded using UTF-16 format with the bytes sent in network order.

2.2 Header descriptions

2.2.1 Count

Count is a four byte unsigned integer to indicate the number of objects involved in the operation.

2.2.2 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. In some cases an empty **Name** header is used to specify a particular behavior; see the **GET** and **SETPATH** Operations. An empty **Name** header is defined as a **Name** header of length 3 (one byte opcode + two byte length).

2.2.3 Type

Type is a byte sequence 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". OBEX follows RFC 1521 which defines the media type format and handles **Type** header values are case insensitive values. See the following URL for a list of MIME defined media types: http://www.isi.edu/in-notes/iana/assignments/media-types.

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.

2.2.4 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.

2.2.5 Time

Time is a byte sequence that gives the object's UTC date/time of last modification in ISO 8601 format. Local times should be represented in the format YYYYMMDDTHHMMSS and UTC time in the format YYYYMMDDTHHMMSSZ. The letter "T" delimits the date from the time. UTC time is identified by concatenating a "Z" to the end of the sequence. When possible UTC times should be used. This header is encoded as a US-ASCII string. 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.

2.2.6 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.

2.2.7 Target

Target is a byte sequence 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.

The **Target** headers most common use, is when sent in an OBEX **CONNECT** packet to initiate a directed connection to an OBEX server (see section 4.6). A list of well-known **Target** header values is contained in the appendix of this specification. The **Target** header is commonly used in conjunction with the **Who** and **Connection Id** headers when establishing a directed connection.

When used with the **PUT** operation, it allows for behavior analogous to the HTTP POST operation. Wherein the **Target** specifies the service that should process the object contained in the **PUT** request.

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. When used, the **Target** header must be the first header in the operation.

To work effectively in a broad application environment it is necessary that the **Target** header identify a universally unique service or client. It is recommended that 128-bit UUID's be used to fulfill this requirement. Since the **Target** header is a binary header type, values are compared in a case sensitive manner. Therefore, care must be taken to maintain the proper case when using ASCII values.

It is illegal to send a **Connection Id** and a **Target** header in the same request. It is legal to send both headers in a response, as discussed in section 3.4.1.6.

2.2.8 HTTP

HTTP is a byte sequence 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.

2.2.9 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.

2.2.10 Who

Who is a length prefixed byte sequence used 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.

The **Who** header is typically used in an OBEX **CONNECT** response packet to indicate the UUID of the service which has accepted the directed connection. The value of the **Who** header matches the value of the **Target** header sent in the **CONNECT** command.

To work effectively in a broad application environment it is necessary that the **Who** header identify a universally unique service or client. It is recommended that 128-bit UUID's be used to fulfill this requirement.

2.2.11 Connection Identifier

Connection Id is a 4-byte value that tells the recipient of the request which OBEX connection this request belongs to. The **Connection Id** header is optional. When in use, the **Connection Id** header *must* be the first header in the request.

When it is desirable to provide concurrent access to multiple OBEX services over the same TinyTP connection, the **Connection Id** is used to differentiate between multiple clients. This is often the case when multiple services are accessed via the default OBEX server. The server can identify the need for a connection to be assigned, by the presence of a **Target** header in the OBEX **CONNECT** packet. The connection identifier is returned to the client in the OBEX **CONNECT** response packet. This connection identifier must be unique so that services can be uniquely identified. Once a logical OBEX Connection has been established, all further client operations on that service must include the **Connection Id** header. Only the first request packet in the operation needs to contain the **Connection Id** header. As a result, only one operation can be processed at a time.

If a **Connection Id** header is received with an invalid connection identifier, it is recommended that the operation be rejected with the response code (0xD3) "Service Unavailable". Since not all servers will parse this header, there is no guarantee that this response code will be returned in all such cases. For convenience the connection identifier value 0xFFFFFFFF is reserved and is considered invalid.

It does not make sense to send a **Connection Id** header in an OBEX **CONNECT** operation and is therefore forbidden. It is optional to send a **Connection Id** header in an OBEX **ABORT** operation, as discussed in section 3.4.5. It is also illegal to send a **Connection Id** and a **Target** header in the same request.

2.2.12 Application Request-Response Parameters

The **Application Parameters** header is used by applications (and protocols) layered above OBEX to convey additional information in a request or response packet. In a request, this header conveys request parameters or modifiers. In a response, it is used in cases where the simple pass/fail status returned by OBEX is insufficient. In order to support an unbounded set of values, from integer values to whole structures (used in RPC style requests) the Application Parameter header is based on the OBEX Byte-Sequence Header format.

A Tag-Length-Value encoding scheme is used to support a variety of request/response types and levels. An application parameters header may contain more than one tag-length-value triplet. The header format is shown below:

Parameter Triplet 1			Parameter Triplet 2			Parameter Triplet . . .		
Tag1	Length	Value	Tag2	Length	Value	Tag	Length	Value

The tag and length fields are each one byte in length. The value field can be from zero to *n* bytes long. The value *n* is constrained by the maximum size of an OBEX header, the length field maximum of 255 bytes and the size of other TLV-triplets encoded in the header.

TAG values are defined by the applications or upper protocol layer that uses them and only need to be unique within the scope of the application or protocol layer.

2.2.13 Authenticate Challenge

The **Authenticate Challenge** header is used by both clients and servers to initiate the authentication of the remote device. This header carries the digest-challenge string. See the Authentication chapter for a detailed explanation of the authentication procedure.

A Tag-Length-Value encoding scheme is used to support the variety of options available for authentication. An **Authenticate Challenge** header may contain more than one tag-length-value triplet. The header format is shown below:

Authentication Triplet 1			Authentication Triplet 2			Authentication Triplet . . .		
Tag1	Length	Value	Tag2	Length	Value	Tag	Length	Value

The tag and length fields are each one byte in length. The value field can be from zero to *n* bytes long. The value *n* is constrained by the maximum size of an OBEX header, the length field maximum of 255 bytes and the size of other TLV-triplets encoded in the header.

2.2.14 Authenticate Response

The **Authenticate Response** header is used by both clients and servers to respond to an authentication request. This header carries the digest-response string. See the Authentication chapter for a detailed explanation of the authentication procedure.

A Tag-Length-Value encoding scheme is used to support the variety of options available for authentication. An **Authenticate Response** header may contain more than one tag-length-value triplet. The tag format is the same as that defined for the **Authenticate Challenge** header shown above.

2.2.15 Creator ID

A creator ID on the PalmOS is similar to the creator ID on a Macintosh. It is a 4-byte (32bit) value that indicates the creator of a file. It can be used to find the application that should receive the data in the same way as the extension or the MIME type.

2.2.16 WAN UUID

The **WAN UUID** header is for use when layering OBEX over stateless networks where the roles of network client and server are opposite to those of OBEX client and server (such as in WAP). In such an environment, the network client (the mobile device) is also the OBEX server. Here, the **WAN UUID** is used to provide state because the network client is only recognized by the server for one request/response pair at a time. The **WAN UUID** header is a 16 byte UUID that uniquely identifies the network client (OBEX server) to the network server. The network server provides the UUID to the OBEX server in an OBEX Connect request packet. The OBEX Server then sends the header to the network server in all future OBEX responses. The header type is byte-sequence (0x20).

2.2.17 Object Class

The **Object Class** header is used to reference the object class and properties. It is based on the byte sequence header type. At its lowest level the **Object Class** header works similarly to the OBEX **Type** header except that its namespace is that of OBEX not MIME. For sub-object level access, the header can be detailed enough to express specific fields or subsets of the objects' contents. The values used in **Object Class** headers are limited in scope to the application or services that define them. However, the sharing of Object Classes is encouraged.

2.2.18 Session-Parameters

Session-Parameters is a byte sequence that is required for the **CREATESESSION**, **CLOSESESSION**, **SUSPENDSESSION**, **RESUMESESSION** and **SETTIMEOUT** commands and the responses to these commands. It contains a set of parameters, which are specific to the command in which the header is used. If required, the **Session-Parameters** must be the first header in a **SESSION** command or response. In addition, for **SESSION** commands only, the Session Opcode must be the first tag-length-value triplet within the **Session-Parameters** header.

A Tag-Length-Value encoding scheme is used to support a variety of parameters. A **Session-Parameters** header may contain more than one tag-length-value triplet. The header format is shown below:

Parameter Triplet 1			Parameter Triplet 2			Parameter Triplet . . .		
Tag1	Length	Value	Tag2	Length	Value	Tag	Length	Value

The tag and length fields are each one byte in length. The value field can be from zero to *n* bytes long. The value *n* is constrained by the maximum size of an OBEX Packet, which at this point is 255 bytes, the length field maximum of 255 bytes and the size of other TLV-triplets encoded in the header.

The following TAG values are defined.

Tag Value	Name	Meaning
0x00	Device Address	The *device address* of the device sending the header. If running over IrDA this is the 32-bit device address. For Bluetooth this is the 48-bit Bluetooth address. If Running over TCP/IP this is the IP address.
0x01	Nonce	The nonce is a value provided by the device, which will be used in the creation of the session ID. This number must be unique for each session created by the device. One method for creating the nonce is to start with a random number then increment the value each time a new session is created. The Nonce should be at least 4 bytes and at most 16 bytes in size.
0x02	Session ID	Session ID. This is a 16-byte value, which is formed by taking the device address and nonce from the client and server and running the MD5 algorithm over the resulting string. The Session ID is created as follows: MD5("*Client Device Address*" "*Client Nonce*" "*Server Device Address*" "*Server Nonce*")
0x03	Next Sequence Number	This is a one-byte value sent by the server, which indicates the next sequence number expected when the session is resumed.
0x04	Timeout	This is a 4-byte value that contains the number of seconds a session can be in suspend mode before it is considered closed. The value of 0xffffffff indicates a timeout of infinity. This is the default timeout. If a device does not send a timeout field then it can be assumed that the desired timeout is infinity. The timeout in affect is the smallest timeout sent by the client or server.
0x05	Session Opcode	The session opcode is a 1-byte value sent by the client to indicate the Session command that is to be performed. This tag-length-value is only sent in **SESSION** commands and must be the first tag in the **Session-Parameters** header. The session opcode definitions are defined in the bulleted list below.
0x06	Object Offset	The object offset is an unsigned integer value containing the number of object data bytes (i.e. Body/End Of Body header data) that have been successfully **received** during a portion of object exchange. This object data pertains to Body/End Of Body header data received from the Client during a **PUT** operation or from the Server during a **GET** operation. This value shall be used to determine the position in the object where the data flow shall continue on the remote device when the session and its subsequent suspended operation are resumed. The Object Offset length may vary, but it is **recommended** to be at least 4-bytes in length. If **Single Response Mode (SRM)** is enabled, this value **shall** be maintained by the receiving device for the active **PUT/GET** operation and issued during a Resume Session should it occur (see Resume Session discussion in section 3.4.7.4). This value shall only track the progress of the object data for the active operation, with all outstanding queued operations being monitored separately based on the SSN issued.
0x07 - 0xff		Reserved

Session Opcode Definitions:

- 0x00 Create Session
- 0x01 Close Session
- 0x02 Suspend Session
- 0x03 Resume Session
- 0x04 Set Timeout
- 0x05 – 0xFF Reserved

2.2.19 Session-Sequence-Number

Session-Sequence-Number (SSN) is a 1-byte quantity containing the current sequence number. Even though OBEX is a command/response protocol and only one outstanding command can exist at any given time, the counting capacity of the sequence number is 256 using digits 0 – 255 (0xFF). At 255 the sequence number wraps around to 0. This adds another level of verification that the correct session is being resumed. The exception to this case is **SRM** which allows multiple outstanding commands and operations to exist. If the sequence number is invalid then the session should be aborted. The Client shall issue a CLOSESESSION in this case, while the Server shall issue a transport disconnection.. This procedure should be followed both when the session is being resumed and during an active session.

When the session is first created the **SSN** number is set to 0. The first command packet sent by the client after the session is established shall contain a **SSN** header with value of 0. The response to this packet from the server shall contain the next packet number expected. If the packet was received successfully then the value of the **SSN** header in the response would be 1. In the case of **SRM** the SSN shall increment for each request/response packet issued, even though the remote device may be unable to immediately acknowledge the outstanding packet.

In the case where **SRM** is used in combination with **Reliable Sessions two possible scenarios may exist:**

> 1) If SRM is enabled over an OBEX Connection it is possible for multiple outstanding packets (or operations) to exist, which could result in outstanding sequence numbers exceeding 256. In this case, the **Single Response Mode Parameters (SRMP)** header shall be issued at least every 255 packets to request an additional packet (0x00 parameter is used – see section 2.2.24) to ensure the SSN is acknowledged, in order to maintain reliability over the OBEX Connection.
>
> The **Object Offset** shall be tracked to guarantee reliability of the object data (i.e. Body/End Of Body header data) for the active operation. All non-Body headers for the active operation and all headers for any queued operations shall be tracked based on the **SSN** pertaining to these packets. If a Resume Session is required, the **Object Offset** and **Next Sequence Number** TAG values shall be used to restore the interrupted operation(s).
>
> 2) If **SRM** is enabled for a single OBEX operation the **Object Offset** shall be tracked to guarantee reliability of the object data (i.e. Body/End Of Body header data) for the single active operation. All non-Body headers for this single active operation shall be tracked based on the **SSN** pertaining to their packet. In the case where a large number of non-Body headers are transmitted, the use of the **SRMP Header** as mentioned above shall be used.

If all non-Body headers have been issued, it shall be allowed for the sending device to issue one final **SRMP Header** to trigger an acknowledgement of any outstanding non-Body headers, after which time the **SSN** shall be allowed to wrap-around without further use of the **SRMP Header**. During this time the **SSN** shall be ignored, as the **Object Offset** shall be tracked to maintain reliability during the sending of the object data. If a Resume Session is required, the **Object Offset** and **Next Sequence Number** TAG values shall be used to restore the interrupted operation.

The **Session-Sequence-Number** must be the first header. The OBEX specification indicates that the **Connection-ID** header must be the first header so there appears to be conflict. Since the new reliable session features sit below the standard OBEX parser, the reliable session could be considered a lower layer protocol. Thus, session information should be stripped off before being sent up to the OBEX parser. The standard OBEX parser will not see the **Session-Sequence-Number** header and the **Connection ID** header will appear to be the first header when it exists.

Session-Sequence-Number headers are not used in Session commands or responses. Thus, session numbers are not advanced when sending session commands/responses when a reliable session is active.

2.2.20 Action Identifier

Action Id is a 1-byte quantity for specifying the action of the ACTION operation. The Action Id header contains an Action Identifier, which defines what action is to be performed. The Action Id header is mandatory for the ACTION operation and restricted for any other OBEX operations (see Section 3.4.8 Action). When in use, the Action Id header must be the first header in the request unless the Connection Id header is in use, in which case the Action Id header must immediately follow the Connection Id header(see section 2.2.11 Connection Identifier).

Action Identifier	Action Name	Description
0x00	Copy Object	(See Section 3.4.8.1 Copy Object Action)
0x01	Move/Rename Object	(See Section 3.4.8.2 Move/Rename Object Action)
0x02	Set Object Permissions	(See Section 3.4.8.3 Set Object Permissions Action)
0x03 – 7F	Reserved for future use	Reserved for assignment by OBEX.
0x80 – FF	Reserved for use in vendor extensions	An area of the Action Identifier space, which is reserved for use in vendor specific applications.

2.2.21 DestName

DestName is a null terminated Unicode text string describing the destination name of the object. The DestName header is used with different ACTION operations and its usage is action specific (see for more details sections 3.4.8.1 Copy Object Action and 3.4.8.2 Move/Rename Object Action). It MUST NOT be used with the PUT and GET operations.

The schema used in the DestName can be either the same as the standard FTP (File Transfer Protocol) uses or UNC (Universal Naming Convention).

In the "FTP style" the object names can be relative to the current folder or absolute pathnames. The backslash "\"or slash "/" character must be used to indicate the folder hierarchy within the DestName headers i.e. "\blah\blahsubdir\file.txt", when referring locations outside the current folder.

Examples of valid DestName values using "FTP style":

Text.txt (relative to the current folder)
MyStuff\Text.txt (relative to the current folder)
MyStuff/Text.txt (relative to the current folder)
c:\MyStuff\Texts\Text.txt (absolute)
c:/MyStuff/Texts/Text.txt (absolute)
\MyStuff\Texts\Text.txt (relative to the root folder)
/MyStuff/Texts/Text.txt (relative to the root folder)

If UNC path names are used a full path name must be given starting with "\\". Backslash "\" is used to delimit path.

Example of valid UNC path name:

\\server1\storage1\MyStuff\Text.txt

2.2.22 Permissions

Permissions is a 4-byte unsigned integer where the 4 bytes describe bit masks representing the various

permission values. It is used for setting "Read", "Write", "Delete" and "Modify Permissions" permissions for files and folders. The permissions are applied to three different permission levels, which are "User", "Group" and "Other". The **Permissions** header can be used with different **ACTION** operations or with the **PUT**, **GET** and **SETPATH** operations. In the case of a **PUT** and **SETPATH**, the object permissions should be set according to the **Permissions** header only when a new file/folder is created. When changing the permissions of an already existing file/folder the **ACTION** operation should be used (see section 3.4.8.3 Set Object Permissions Action). On a **GET**, the OBEX Server may return the permissions of the object in this header.

Byte 0 (MSB)	Byte 1	Byte 2	Byte 3 (LSB)
Reserved (Should be set to zero)	User permissions	Group permissions	Other permissions

Within the context of OBEX, a "User" refers to permissions for the OBEX Client's current authenticated user (see section 3.6 Authentication Procedure). If no user is authenticated, permissions are relative to a"guest" user. "Group" refers to the group or groups to which the OBEX Server has assigned the current authenticated user. "Other" refers to permissions for a user who is neither the current user nor the member of any of the user's groups.

Permissions flags

The bits in each permissions byte have the following meanings:

bit	Meaning
0	Read. When this bit is set to 1, reading permission is granted. For a file object, the OBEX Client may use ACTION/Copy or GET on the file. For a folder object, the OBEX Client may use ACTION/Copy or SETPATH on the folder. To do ACTION/Move client needs to have both "Read" and "Delete" permission to the source file/folder.
1	Write. When this bit is set to 1, writing permission is granted. For a file object, the OBEX Client may use PUT on the file. To do ACTION/Move client needs to have both "Read" and "Delete" permission to the source file/folder.
2	Delete. When this bit is set to 1, deletion permission is granted. For a file/folder object, the OBEX Client may use PUT-delete.
3	Reserved for future use
4	Reserved for future use
5	Reserved for future use
6	Reserved for future use
7	Modify Permissions. When this bit is set to 1 the file access permissions can be changed. For a file/folder object, the OBEX Client may use ACTION/Set Object Permissions

For devices that do not discriminate between permission groups, the default is "User" permissions. That is, the "Group" permissions byte should match the "User" permissions byte.

2.2.23 Single Response Mode (SRM)

Single Response Mode (SRM) is a 1-byte quantity containing a value to enable or disable SRM, as well as to indicate support for SRM. This header shall be used during **CONNECT**, **DISCONNECT**, **SETPATH**, **PUT** and **GET** operations only (see the table below).

When issued in conjunction with the **Single Response Mode (SRM) Parameters** header, this header MUST occur first. This is necessary for cases where SRM is being enabled and parameters are being set all in the same OBEX operation.

The possible 1-byte parameter values are listed in the table below

Parameter Value	Meaning
0x00	This value informs the remote device that the local device is requesting SRM to be disabled. Disabling SRM will occur when this value is issued by the Client and received by the Server. The Server may also request that SRM be disabled by issuing this value, but SRM will only be disabled when the Client responds by also issuing this value. (Valid during all operations) When issued by the Server, the Client **must** respond before the disable request will complete. However, when issued by the Client, the Server **shall not** respond, as the disable occurs upon receipt by the Server. Disabling SRM is **optional** and may be ignored if a device does not support this feature. If unsupported, SRM will be disabled through the **DISCONNECT** operation or when an OBEX transport disconnect occurs.
0x01	This value informs the remote device that the local device is requesting SRM to be enabled. Enabling SRM requires **both** devices to send this value; otherwise the feature will remain disabled. Either the Client or Server may initiate the request for SRM to be enabled, but this request **must** be followed by a response from the remote device using this same value. (Valid during all operations except DISCONNECT)
0x02	This value informs the remote device that the local device supports SRM. This is used by the Client to advertise SRM support, to prevent a Server from attempting to enable SRM on a Client that cannot support this mode. Similarly, a Server may have the opportunity to advertise SRM support, to handle the case where a Client does not immediately enable SRM as its first exchange with the Server. (Valid during all operations except DISCONNECT.)

2.2.24 Single Response Mode (SRM) Parameters

Single Response Mode (SRM) Parameters is a 1-byte quantity containing the value for the parameters used for SRM. This header shall be used during **PUT** and **GET** operations, with limited use allowed during all other operations except **DISCONNECT** (see the table below).

The possible 1-byte parameter values are listed in the table below.

Parameter Value	Meaning
0x00	The local device (client or server) is requesting an additional packet (response if the local device is a client; request if the local device is a server) from the remote device (client or server), which shall occur upon receipt of the packet containing this header. This packet is in addition to the expected Single Request/Response for the operation. The issuing device may choose to continue sending packets while anticipating the incoming additional packet from the remote device. (Only valid during **PUT** operations for Client devices and **GET** operations for Server devices)
0x01	The local device (client or server) is requesting that the remote device (client or server) wait, after sending its next request/response packet, until another packet is received from the local device. If another request/response is not expected by the remote device, this parameter value will additionally inform the remote device that an additional request/response shall occur from the local device. This behavior will be applied to valid SRM operations only (**PUT** and **GET**). If this wait is pending upon the completion of an operation, it will remain pending until another valid SRM operation occurs. (Only valid during **GET** operations for Client devices, and **all** operations except **DISCONNECT** for Server devices)
0x02	This value issues a combination of parameter values 0x00 and 0x01 as discussed above. This is useful in cases where the local device needs to request an additional packet and invoke a wait on the remote device. (Only valid during **PUT** operations for Client devices, and **GET** operations for Server devices)

2.2.24.1 Use Cases

The following use cases describe some of the possible uses of the Single Response Mode Parameter values described in the table above. This is not an exhaustive listing of all possible use cases, as other cases may exist and are up to the application layers to define.

Case 1: **Client requests approval of the PUT operation by the Server**

PUT →
0x98 – HI for **Single Response Mode Parameters**
0x00 – HV (Request an additional response packet)
← **CONTINUE** (Additional response packet)
PUT →
PUT (Final Bit) →
← **SUCCESS** (Final Bit)

Case 2: Server requests feedback from Client during a GET operation

GET →
← **CONTINUE**
0x98 – HI for **Single Response Mode Parameters**
0x00 – HV (Request an additional request packet)
GET →
(Additional request packet)
← **CONTINUE**
← **CONTINUE**
← **SUCCESS** (Final Bit)

Case 3: **Server enforces flow control for PUT operation**

CONNECT →
← **SUCCESS** (Final Bit)
0x98 – HI for **Single Response Mode Parameters**
0x01 – HV (Client must wait for next response)
PUT →
(WAIT)
← **CONTINUE**
PUT →
PUT →
PUT (Final Bit) →
← **SUCCESS** (Final Bit)

Case 4: Client enforces flow control for GET operation

GET →
0x98 – HI for **Single Response Mode Parameters**
0x01 – HV (Server must wait for next request)
← **CONTINUE**
(WAIT)
GET →
← **CONTINUE**
← **CONTINUE**
← **SUCCESS** (Final Bit)

Case 5: Server forces only one operation to be queued by the Client

CONNECT →
← **SUCCESS** (Final Bit)
0x98 – HI for **Single Response Mode Parameters**
0x01 – HV (Client must wait for next response)
GET →
(WAIT)
← **CONTINUE**
0x98 – HI for **Single Response Mode Parameters**
0x02 – HV (Request an additional request packet, and Client must wait for next response)
GET →
(Additional request packet)
(WAIT)
← **CONTINUE**
0x98 – HI for **Single Response Mode Parameters**
0x02 – HV (Request an additional request packet, and Client must wait for next response)
GET →
(Additional request packet)
(WAIT)
← **CONTINUE**
0x98 – HI for **Single Response Mode Parameters**
0x02 – HV (Request an additional request packet, and Client must wait for next response)
GET →
(Additional request packet)
(WAIT)

← **SUCCESS** (Final Bit)

Now the next operation can be initiated:

PUT →
PUT →
PUT (Final Bit) →
← **SUCCESS** (Final Bit)

2.2.25 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.

3. 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 IrLAP 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 two exceptions.

1) An **ABORT** operation may come in the middle of a request/response sequence. This cancels the current outstanding operation. In this case one response **shall** occur for the **ABORT** operation, however, one additional response **should** occur prior to the first for the outstanding **PUT** or **GET** request. Exceptions to this rule include legacy devices that are only capable to issue one response and during a **Single Response Mode (SRM)** connection where only one response is expected. In the case of two responses they **shall** be transmitted in the order in which the requests were received.

2) **Single Response Mode (SRM)** may be negotiated during a **CONNECT** which will remain enabled until a **DISCONNECT,** OBEX transport disconnect, or SRM is disabled using the SRM header. If this mode is enabled during a single operation (e.g. PUT or GET), SRM shall be automatically disabled when the operation that enabled this mode completes. SRM allows multiple outstanding request or response packets during the context of an OBEX connection (or OBEX operation). Since SRM allows multiple outstanding packets, this mode also makes available the ability for multiple outstanding OBEX operations to exist at a time. See the discussion on SRM in **section 3.3** for complete details on this procedure and **section 7** for examples on suspend session.

3) A **SUSPENDSESSION** operation may occur in the middle of a request/response sequence. This shall immediately suspend the session, including the outstanding operation. In this case two responses shall occur, one for the **SUSPENDSESSION** operation, and one for the outstanding **PUT** or **GET** request. The responses shall be transmitted in the same order in which the requests were received.

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**.

3.1 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.

If the operation requires multiple response packets to complete after the Final bit is set in the request (as in the case of a **GET** operation returning an object that is too big to fit in one response packet). The server will signal this with the "Continue" response code, telling the client that it should ask for more. The client (requestor) should send another request packet with the same opcode, Final bit set, and no headers if it wishes to continue receiving the object. If the client does not want to continue the operation, it should send an **ABORT** packet.

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.

3.2 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. In OBEX 1.0 it is always set for response packets. Its meaning is slightly different from the request packet final bit—the response packet final bit tells the other side (the client) that it is once again the client's turn to send a packet. If the response packet carries a success or failure response code, the client is free to begin a new operation. However, if the response code is "Continue" as is often the case in a lengthy **GET** operation, then the client next issues a continuation of the **GET** request. See the **GET** operation description below for examples.

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 before 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).

3.2.1 Response Code values

OBEX response code	HTTP status code	Definition
0x00 to 0x0F	None	Reserved
0x10 (0x90)	100	Continue
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
0x50 (0xD0)	500	Internal Server Error
0x51 (0xD1)	501	Not Implemented
0x52 (0xD2)	502	Bad Gateway
0x53 (0xD3)	503	Service Unavailable
0x54 (0xD4)	504	Gateway Timeout
0x55 (0xD5)	505	HTTP version not supported
0x60 (0xE0)	- - -	Database Full
0x61 (0xE1)	- - -	Database Locked

3.3 OBEX Single Response Mode (SRM)

With the introduction of faster transports for OBEX, a bottleneck has been reached due to the request/response behavior in OBEX. This behavior causes a significant turn around time delay and additional overhead, since every request packet must be followed by a response packet. In the case of large **PUT** and **GET** operations, this significantly slows down the exchange of objects.

OBEX Single Response Mode (SRM) addresses this problem by altering the existing request/response requirements, but only under specific circumstances. This mode provides the following new rules and behavior for normal SRM usage.

Normal SRM is defined as SRM without the use of the **Single Response Mode (SRM) Parameters** header. This header may significantly alter the normal behavior of SRM, and a description of these effects is described in **section 2.2.24**.

Normal SRM Rules and Behavior

- The underlying OBEX transport layer **must** provide a reliable flow control mechanism. This behavior will not be provided by the OBEX layer directly.

- The ability to send and receive the **Single Response Mode (SRM)** header is **required** for devices supporting SRM. Additionally, the ability to receive the **Single Response Mode (SRM) Parameters** header is also **required**, but **optional** to send during SRM.

- Client or Server devices supporting SRM **may** issue a **Single Response Mode (SRM)** header to advertise support for SRM to the remote device. This may occur during all operation requests/responses (with the exception of **DISCONNECT**) that occur outside of an active SRM session. For **PUT** and **GET** operations which may span many packets, only the first request/response shall include this header.

- Server devices may issue a request for SRM to be enabled, but this **must** occur after receiving a **Single Response Mode (SRM)** header from the remote Client indicating that it supports SRM. Since the Client must respond to this request prior to SRM being enabled, this will occur in the next operation initiated by the Client. Once the **Single Response Mode (SRM)** header is received by the Server confirming SRM, this mode is enabled and will pertain to all future operations, including the current operation.

- If a **Single Response Mode (SRM)** header requesting SRM mode is received, the remote device must issue a response in its next packet. Failure to do so nullifies the SRM enable attempt, and must be issued again by the requesting device.

- SRM shall be enabled by using the **Single Response Mode (SRM)** header. SRM **shall** be used for all multi-response operations (**PUT** and **GET**) for the duration of the SRM mode.

- OBEX **PUT** and **GET** operations shall use the **Single Response Mode (SRM) Parameters** header to alter the normal behavior of SRM. Other OBEX operations (**see table in section 2.2.24**) may use the **Single Response Mode (SRM) Parameters** to influence an upcoming **PUT/GET** operation, but no behavior will be impacted in an operation outside of a **PUT** or **GET**.

- SRM applies only to the OBEX connection it is enabled under. All other OBEX connections shall behave according to the normal OBEX request/response model. If no OBEX connection exists when SRM is enabled, SRM shall be automatically disabled upon completion of the operation which enabled it. "

- **Single Response Mode (SRM) Parameters** header values may be issued in any valid request/response packet, as described in **section 2.2.24.**

- SRM may be disabled by using the **Single Response Mode (SRM)** header. SRM will also be disabled if the OBEX connection using SRM is closed with the **DISCONNECT** command, or if the OBEX transport is disconnected. However, if Reliable Sessions are used SRM can be activated again, if the transport disconnection caused SRM to disable, by using the **RESUME SESSION** command.

- All operations issued during SRM can be sent immediately, without waiting for the Server to respond. The Server **must** queue all pending OBEX commands and process them in the order they arrive. As each command is processed, a single response will be indicated for each operation. The exception to this rule is **GET**, which will trigger multiple Server response packets until the object is finished being sent.

- In the case of **PUT** or **GET** commands, which may span many OBEX packets, multiple OBEX request packets (**PUT**) or OBEX response packets (**GET**) can be sent immediately, without waiting for the remote device.

- Each **PUT** operation will result in **exactly one** OBEX response (unless the SRM Parameters header is used), with this response indicating the final status of the operation once it is complete.

- Each **GET** operation will result in **exactly one** OBEX request including the Final bit (unless the SRM Parameters header is used), which will indicate that the object transfer can begin on the Server. No OBEX response packets including object data will occur until the OBEX request including the Final bit is received by the Server. Additional OBEX request packets may be sent by the Client prior to issuing the Final bit in the case where many OBEX headers are needed to complete the Request phase of the **GET** operation.

- The **ABORT** operation is handled in the order it arrives on the Server, exactly the same as all other operations. This means that any operations received prior to the Abort will complete, and only the operation that is in progress when the Abort is processed will be aborted. In the case of a **GET** operation, it is possible that this operation could complete prior to an issued **ABORT** being processed. This would result in an exception case, where two responses would be issued. The first response would indicate the **GET** completion and the second indicates the **ABORT** completion. This apparent exception case actually follows the rules as well, since no operation would be available to abort in this case. Therefore, each operation will end up completing separately, with one response being received for each.

- **SRM** can be used with Reliable Sessions, however, as more than one outstanding packet can occur, a Resume Session operation may result in a **Session Sequence Number** requiring many packets to be resent, if the remote device was unable to process all of the pending OBEX requests or responses prior to the suspension of the OBEX session. This adds additional requirements on the Client/Server to manage all outstanding OBEX packets until confirmation is received from the remote device, just in case an unexpected transport disconnection occurs. The management of the outstanding OBEX packets will require the device to either store the outstanding data or be able to recreate the data if requested.
- As one or more operations may be queued on the Server at the time an **ABORT** or **SUSPENDSESSION** is issued, it may take a significant portion of time for the Server to process either command. Therefore, it is recommended that a timeout be maintained on the Client, which will trigger a transport disconnection in the case that these commands remain outstanding upon timer expiration. This will facilitate a good user experience when canceling or suspending an object transfer. The exact timeout may vary, but is recommended to be less than 15 seconds.

- If SRM and Reliable Sessions are used concurrently, it is **recommended** that the OBEX Client issues only one outstanding operation at a time. In the case where a spurious transport

disconnection or immediate suspend session occurs, this will significantly simplify the requirements to resume such a session.

3.4 OBEX Operations and Opcode definitions

OBEX operations consist of the following:

Opcode (w/high bit set)	Definition	Meaning
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)	Reserved	Not to be used w/out extension to this specification
0x85 *high bit always set	SetPath	Modifies the current path on the receiving side
0x06 (0x86)	Action	Common actions to be performed by the target
0x87 *high bit always set	Session	Used for reliable session support
0xFF *high bit always set	Abort	Abort the current operation
0x08 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
Bits 5 and 6 are reserved and should be set to zero.		
Bit 7 of the opcode means Final packet of request.		

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. However, one notable exemption from this rule is the **ABORT** opcode, which currently sets bits 5 and 6.

If a server receives an unrecognized opcode, it should return 0xD1 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.

3.4.1 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
0x80	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.

3.4.1.1 OBEX version number

The version number is the version of the OBEX protocol encoded with the major number in the high order 4 bits, and the minor version in the low order 4 bits. The protocol version is not always the same as the specification version. The current protocol version is 1.0. See the example later in this section.

3.4.1.2 Connect flags

The flags have the following meanings:

bit	Meaning
0	Response: Indicates support for multiple IrLMP connections to the same LSAP-SEL. Request: reserved
1	Reserved
2	Reserved
3	Reserved
4	Reserved
5	Reserved
6	Reserved
7	Reserved

All request flags except bit 0 are currently reserved, and must be set to zero on the sending side and ignored by the receiving side. All reserved response bits must also be set to zero and ignored by the receiving side.

In a **CONNECT** Response packet bit 0 is used to indicate the ability of the OBEX server's transport to accept multiple IrLMP connections to the same LSAP-SEL. This capability, usually found in more robust IrDA stacks, allows the server to use the LM-MUX for multiplexing multiple simultaneous client requests. Conveying this information is necessary because the IrDA Lite specification does not support this type of multiplexing and attempts to connect to an already connected LSAP-SEL will result in an IrLAP disconnect.

Bit 0 should be used by the receiving client to decide how to multiplex operations to the server (should it desire to do so). If the bit is 0 the client should serialize the operations over a single TTP connection. If the bit is set the client is free to establish multiple TTP connections to the server and concurrently exchange its objects.

3.4.1.3 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.

This is one of the most important features of the **CONNECT** packet because it permits the application to increase the OBEX packet size used during an exchange. The default OBEX packet size of 255 bytes is inefficient for large object transfers and will impede transfer rates. Larger OBEX packet sizes such as 8k to 32k allow large amounts of data to be sent without acknowledgement. However, packet sizes should be intelligently limited on slower links to reduce abort request latency.

3.4.1.4 Minimum OBEX Packet Length

The minimum size of the OBEX Maximum packet length allowed for negotiation is 255 bytes. This is in order to provide a greater likelihood of meeting the requirement for single packet requests and responses in a broad range of cases. In addition, since an OBEX header must fit completely within one OBEX packet it is advantageous to mandate a minimum that allows for reasonable header sizes. Note that

OBEX does not exchange Minimum Packet Length values. This value is the minimum acceptable value for the OBEX Maximum Packet Length parameter exchanged in the **CONNECT** Operation.

3.4.1.5 Using Count and Length headers in Connect

The **Count** and **Length** headers, defined in the Object Model chapter, can be used in **CONNECT**. **Count** is used to indicate the number of objects that will be sent during this connection. **Length** is used to express the approximate total length of the bodies of all the objects in the transaction.

When used in the **CONNECT** Operation, the **Length** header contains the length in bytes of the bodies of all the objects that the sender plans to send. Note that this length cannot be guaranteed correct, so while the value may be useful for status indicators and resource reservations, the receiver should not die if the length is not exact. The receiver can depend on **Body** headers to accurately indicate the size of an object as it is delivered and the **End-of-Body** header to indicate when an object is complete.

3.4.1.6 Using Who and Target headers in Connect

Target and **Who** are 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 body headers.

On full-featured [PC] platforms, multiple OBEX applications may exist concurrently. This leads to the need for the client to be able to uniquely identify which server it wants to handle its request. The server is therefore identified with the OBEX **Target** header. If necessary, the client can also identify itself, using the OBEX **Who** header. The following text describes the exact uses of these headers.

To target a specific application with OBEX commands the client must set-up a connection to the application by using the OBEX **Target** header in a **CONNECT** request. This type of connection is called a directed connection and provides a virtual binding between the client and server. The **Target** header should specify the UUID of the desired application. The **Who** header can also be used when it is necessary to identify the client initiating the exchange. The **Who** header should be used in cases where the target server application supports different client applications and may care which one it is connecting to. It is unnecessary to send a **Who** header in the request if its only logical value is the same as the **Target** header.

The response to the targeted connect operation should contain a **Who** with the same UUID as sent in the request's matching **Target** header. If the **Who** header was present in the request, a **Target** header identifying the same client should be sent in the response. In addition, a unique connection identifier must be sent in a **Connection Id** header. This connection identifier is used by the client in all future operations. If the response does not contain the correct headers then it should be assumed that the connection has not been made to the specific application but to the inbox service. This will be the response when sending a directed connect to a system that does not parse these (Target) headers. In the event that a connection is made to the inbox service, it is the responsibility of the client application to determine whether to continue the exchange or disconnect.

3.4.1.7 Using Description headers in Connect

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.

3.4.1.8 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.

3.4.1.9 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	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:		
response code	0xA0	SUCCESS, Final bit set
	0x0007	packet length of 7
	0x10	version 1.0 of OBEX
	0x00	Flags
	0x0400	1K max packet size

3.4.1.10 OBEX Operations without Connect

It is highly recommended that implementations assume default values for connection parameters (currently just a minimum OBEX packet size of 255 bytes) and accept operations such as **PUT** and **GET** without first requiring a **CONNECT** operation.

3.4.2 Disconnect

This opcode signals the end of the OBEX session. It may include a **Description** header for additional user readable information. It **shall** include a **Connection Id** header if a directed connection is being disconnected, otherwise the default OBEX connection will be assumed. 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. However, if the disconnect packet contains invalid information, such as an invalid **Connection Id** header the response code may be “Service Unavailable” (0xD3). Server side handling of this case is not required.

Byte 0	Bytes 1, 2	Bytes 3 to n
0xA0 or 0xD3	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. Currently, this is common practice and **DISCONNECT** is infrequently used. However, it is good practice to send an OBEX **DISCONNECT** for each OBEX **CONNECT** sent but few applications track or care about such details.

3.4.3 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 that 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 initial packet, or wait until the response to a subsequent 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

3.4.3.1 Headers used in Put

Any of the headers defined in the Object model chapter can be used with **PUT**. These might include **Name**, **Type**, **Description**, **Length**, **Connection Id, HTTP** or other 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 **Name** or **Type** headers are used, they must precede all object **Body** headers.

3.4.3.2 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:

Server Response:	**Bytes**	**Meaning**
response code	0xA0	SUCCESS, Final bit set
	0x0003	length of response packet

3.4.3.3 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	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	Length of **Body** header (1K) plus HI and header length
	0x.....	1K bytes of body
Server Response:		
response code	0x90	CONTINUE, Final bit set
	0x0003	length of response packet
Client Request:		
opcode	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:		
response code	0x90	CONTINUE, Final bit set
	0x0003	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	0x82	**PUT**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	last 1K bytes of body
Server Response:		
response code	0xA0	SUCCESS, Final bit sent
	0x0003	length of response packet

3.4.3.4 Server side handling of Put objects

Servers may do whatever they wish with an incoming object - the OBEX 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 folder 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. It is important that clients, who have a particular purpose in mind when transferring an object, connect to a specific service that it knows can perform the desired behavior.

3.4.3.5 Sending objects that may have read errors

When sending the last portion of an object in an **End-of-Body** header, ambiguity arises if there is any chance that there are read errors in this last portion. This is because the **End-of-Body** normally triggers the receiving side to close the received object and indicate successful completion. If an **ABORT** packet subsequently arrives, it is "too late".

The recommended approach when sending objects which may have such errors is to send the **End-of-Body** header only when the sender knows that the entire object has been safely read, even if this means sending an empty **End-of-Body** header at the end of the object. This applies to both **GET** and **PUT** operations.

3.4.3.6 Put-Delete and Create-Empty Methods

A **PUT** operation with NO **Body** or **End-of-Body** headers whatsoever should be treated as a *delete* request. Similarly, a **PUT** operation with an empty **End-of-Body** header requests the recipient to create an *empty* object. This definition may not make sense or apply to every implementation (in other words devices are not required to support delete operations or empty objects),

A folder can be deleted using the same procedure used to delete a file. Deleting a non-empty folder will delete all its contents, including other folders. Some servers may not allow this operation and will return the "Precondition Failed" (0xCC) response code, indicating that the folder is not empty. In this case, the client will need to delete the contents before deleting the folder.

3.4.4 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	Packet length	sequence of headers

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 the request phase of the **GET** is complete. The server device must not begin sending the object body chunks until the request phase is complete. Once a **GET** is sent with the final bit, all subsequent **GET** request packets must set the final bit until the operation is complete.

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.

3.4.4.1 The default GET object

Refer to chapter 8.4 OBEX Get Default Object for more information on providing default objects. This chapter defines a mechanism by which the client can request a specific type of default object. Such as the default business card or web page.

3.4.5 Abort

The **ABORT** request is used when the client decides to terminate a multi-packet operation (such as **PUT**) before it would normally end. 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. Since there can only be one OBEX operation in progress at a time, the use of a **Connection Id** header to abort a directed multi-packet operation is optional.

Byte 0	Bytes 1, 2	Bytes 3 to n
0xFF	Packet length	optional headers

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

3.4.6 SetPath

The **SETPATH** operation is used to set the "current folder" on the receiving side in order to enable transfers that need additional path information. For instance, when a nested set of folders is sent between two machines, **SETPATH** is used to create the folder structure on the receiving side. The Path name is contained in a **Name** header. The **SETPATH** request and response each always fit in one OBEX packet and have the Final bit set.

Byte 0	Bytes 1, 2	Byte 3	Byte 4	Byte 5 to n
0x85	Packet length	flags	constants	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.

When a new OBEX or TinyTP connection is established, the OBEX server's current folder should be its root folder. In this manner a device may retrieve (or serve) objects from the root folder without requiring the client to perform a **SETPATH** to the root first.

3.4.6.1 Flags

The flags have the following meanings:

bit	Meaning
0	backup a level before applying name (equivalent to ../ on many systems)
1	Don't create folder if it does not exist, return an error instead.
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. Refer to section 3.3.3.6 for a description of how to delete folders.

3.4.6.2 Constants

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

3.4.6.3 Name header

Allowable values for the **Name** header are:

- *<name>* - go down one level into this folder name, relative to the current folder
- *<empty>* - reset to the default folder

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 starts out in a root folder. When sending a folder, the client will start by sending a **SETPATH** request containing the name of the folder. For example, if the full path for the folder on a PC is C:\notes, the client will send "notes" as the **Name** header of a **SETPATH** operation. A PC server would create or switch to the "notes" folder relative to the root folder (or perhaps create a mangled name if "notes" already exists and that was the preferred behavior). As subfolders of "notes" are encountered by the sending routine, additional **SETPATH** requests with the subfolder names will be sent. As folders are completed (all objects in the folder sent), the client will send a **SETPATH** (generally with no **Name** header) and the "back up a level" flag set.

3.4.7 Session

The **SESSION** request is formatted as follows:

Byte 0	Bytes 1, 2	Bytes 3 to n
0x87 (Final bit set)	Packet length	Sequence of headers. **Session-Parameters** must be the first header in the sequence.

The positive response to a **SESSION** request is 0xA0 (Success, with the high bit set. Any failure response code can be used to indicate a negative response. Certain types of failures and the

corresponding failure codes are described in the sections corresponding to each session command. The format of the response is as follows:

Byte 0	Bytes 1, 2	Bytes 3 to n
response code	Response packet length	Optional sequence of response headers. If the **Session-Parameters** is used it must be the first header in the sequence.

3.4.7.1 CREATESESSION

The **CREATESESSION** command is sent by the Client to create a new session. This command must include a **Session-Parameters** header containing the *Session Opcode, Device Address* and *Nonce* fields. Optionally, a *Timeout* field can be included. The successful response to **CREATESESSION** is 0xA0 (Success, with the high bit set) in the response code followed by a **Session-Parameters** header containing *Device Address*, *Nonce*, and *Session ID* fields. Optionally, a *Timeout* field can be included. The client should verify that it creates the same session ID as sent by the server. If the Session ID does not match then the client should close the session by using the **CLOSESESSION** command otherwise the session is considered active.

The **CREATESESSION** command and response must fit in a single OBEX packet of size 255 bytes (default maximum size) and have their Final bits set.

Only one session can be in the active state per transport connection. If a reliable session is already active a second request should be rejected using the 0xC3 (Forbidden) response. To switch sessions the current session must first be suspended. A server is not required to maintain multiple sessions at the same time. All servers will have some maximum number of suspended sessions they can maintain. At some point a server will receive a **CREATESESSION** request that will exceed this maximum number. The server can either reject the new request or close an existing session and accept the new request. The correct response depends on how long the existing sessions have been suspended. The server may want to prompt the user if it cannot decide on its own. A proposal for an algorithm is given in the next paragraph. If a **CREATESESSION** request is rejected because the maximum number of sessions already exists then the proper response is 0xD3 (Service Unavailable).

When a server receives a **CREATESESSION** request that exceeds the maximum number of sessions supported the following algorithm can be used.

1. Keep all suspended sessions with infinite timeouts on a sorted list by time suspended. The session with the longest time is at the front of the list.
2. Sessions with timeouts less then infinite should not be closed until the timeout occurs.
3. When a **CREATESESSION** request occurs exceeding the maximum number of sessions, the server should check the suspended session list. If the list is not empty the first session on the list is removed. This session is closed and its resources are used for the new session. If the list is empty the incoming **CREATESESSION** request is rejected.

3.4.7.2 CLOSESESSION

CLOSESESSION is used to gracefully close an existing session. This command must include a **Session-Parameters** header containing the *Session Opcode and Session ID* fields. The value of the *Session ID* field is the one sent by the server in the response to the **CREATESESSION** command corresponding to the session being closed. The successful response to **CLOSESESSION** is 0xA0 (Success, with the high bit set) in the response code. The **CLOSESESSION** command can be used to close the active session or any suspended sessions.

The **CLOSESESSION** command and response must fit in a single OBEX packet and have their Final bits set.

The CLOSESESSION command cannot be rejected if the *Session ID* corresponds to a valid session. If the *Session ID* does not correspond to a valid session then the proper response is 0xC3 (Forbidden). If the active session is closed the unreliable session becomes the active session.

3.4.7.3 SUSPENDSESSION

SUSPENDSESSION is used to suspend the active session. This suspension can occur gracefully (when no operation is active) or immediately (during an active operation). This command must include a **Session-Parameters** header containing the *Session Opcode* field; a *Timeout* field may be included. The successful response to **SUSPENDESESSION** is 0xA0 (Success, with the high bit set) in the response code; a *Timeout* field may be included.

The **SUSPENDSESSION** command and response must fit in a single OBEX packet and have their Final bits set.

The **SUSPENDSESSION** command cannot be rejected if a reliable session is active. If the unreliable session is the current active session then the proper response is 0xC3 (Forbidden). The unreliable session becomes the active session when a reliable session is suspended.

3.4.7.4 RESUMESESSION

RESUMESESSION is used to resume a session that has been suspended. A session is suspended by using the **SUSPENDSESSION** command or when the underlying transport is broken. This command shall include a **Session-Parameters** header containing the **Session Opcode**, **Device Address**, **Nonce**, and **Session ID** fields; a **Timeout** field may be included. Additionally, if SRM is enabled in the session being resumed, the **Next Sequence Number** and **Object Offset** fields shall be issued. The exception to this case occurs when no **GET** operation is being resumed or when no offset exists, in which case the **Object Offset** value is zero, and may be thereby omitted.

The **Session ID** corresponds to the session being resumed and is the value returned by the server in its response to the **CREATESESSION** command. The **Device Address** and **Nonce** are the values originally used in the **CREATESESSION** command. This is true even if the device address has changed or the session is being resumed over a different transport. Matching a session requires that all values match not just the **Session ID** (**Device Address**, **Nonce** and **Session ID** must match). The **Next Sequence Number** is the value of the sequence number that will be issued in the next request packet. This keeps the server aware of the client's sequence number, as it is possible for the server to immediately resume an operation, without waiting for the client to issue another request, during SRM. The **Object Offset** specifies the offset into the object being resumed.

The successful response to **RESUMESESSION** is 0xA0 (Success, with the high bit set) in the response code. A **Session-Parameters** header containing **Device Address**, **Nonce**, **Session ID**, and **Next Sequence Number** fields shall be sent; a **Timeout** field is optional. Additionally, if SRM was enabled in the suspended session, the **Object Offset** field shall be included. The exception to this case occurs when no **PUT** operation is being resumed or when no offset exists, in which case the **Object Offset** value is zero, and may be thereby omitted. The **Object Offset** specifies the offset into the object being resumed. If the *Device Address*, *Nonce* and *Session ID* do not correspond to a valid session then 0xD3 (Service Unavailable) should be returned. If the information returned by the server does not match then the client should disconnect from the server. **CLOSESESSION** should not be used since this could be a valid session for another device.

The **RESUMESESSION** command and response must fit in a single OBEX packet and have their Final bits set.

If the session is resumed successfully then the client and server shall determine if packet(s) should be retransmitted. As a session may be resumed over an alternate transport or while other services are active, retransmissions are not required to use the same OBEX packet size or issue the same portion of

header contents per packet. However, the contents shall start from the proper resumed location and shall send all of the missing content to the remote device in one or more retransmitted packets. If the session was suspended gracefully then no packets need to be retransmitted, otherwise the client/server shall follow one of the algorithms below:

1) Normal OBEX Command/Response Scenario (no SRM used) – As the OBEX command/response behavior is being used, the client shall issue a request packet, after the session is resumed, prior to any responses from the Server.

Client Algorithm:

```
If the Next Sequence Number field equals the sequence number of the last
packet sent then {
     Re-Send last packet
} else if the Next Sequence Number field equals the sequence number of the
next packet to send then {
     If the response to the last packet was not received then {
          Re-Send last packet minus all headers except
          Session-Sequence-Number;
     } else {
```

```
        Send next packet;
    }
} else {
    /* Sequence number is invalid */
    Disconnect underlying transport connection to OBEX server;
    Remove this session from the list of Suspended Sessions;
}
```

Server Algorithm:

The server sets the Next-Sequence-Number field in the RESUMESESSION response to the sequence number of the next OBEX packet it expects to receive.

```
If the sequence number of the request is equal to the (Next Sequence Number
field minus one) mod 256 then {
    /* The client did not receive the previous response */
    Re-send the previous response;
    Ignore the contents of the request since it was received properly
    before the link was suspended;
} else if the sequence number of the request is equal to the Next Sequence
Number field then {
    /* This is the next packet */
    Process the packet;
    Send the proper response;
} else {
    /* Sequence number is invalid */
    Disconnect underlying transport connection to OBEX client;
    Remove this session from the list of Suspended Sessions;
}
```

2) SRM Scenario – As SRM is being used, the Client shall expect the Server to respond immediately after issuing the RESUME response for a GET operation that is resumed, without waiting for an additional GET request from the Client. All other operations shall be resumed with the Client issuing the first request, prior to any response from the Server.

Client Algorithm:

The client sets the Next-Sequence-Number field in the RESUMESESSION request to the sequence number of the next packet it expects to send. After receiving the **RESUMESESSION** response from the Server do the following:

```
If the Next Sequence Number field equals the sequence number of one of the
outstanding packets sent then {
      Re-Send these packets beginning with the Next Sequence Number;
            The object data (Body/End of Body headers) of the active
            operation shall resume based on the Object Offset TAG from the
            Server; all other headers and queued operations shall resume
            based on the Next Sequence Number;
} else if the Next Sequence Number field equals the sequence number of the
next packet to send then {
    If a response to the last packet was not received, but expected then {
        Re-Send last packet minus all headers except
        Session-Sequence-Number and
            Single Response Mode Parameters;
    } else {
```

```
Send next packet;
```

```
                        The object data (Body/End of Body headers) of the
                        active operation shall resume based on the Object
                        Offset TAG from the Server;
     }
} else {
     /* Sequence number is invalid */
     Disconnect underlying transport connection to OBEX server;
     Remove this session from the list of Suspended Sessions;
}
```

Server Algorithm:

The server sets the **Next-Sequence-Number** field in the **RESUMESESSION** response to the sequence number of the next packet it expects to receive. After receiving the **RESUMESESSION** request packet from the client with the **Next-Sequence-Number** and issuing the **RESUMESESSION** response do the following:

```
If the sequence number of the request equals the sequence number of one of
the outstanding packets sent then {
     /* The client did not receive one (or more) of the
      * previous responses, so immediately respond.
      */
     Re-send the previous responses starting with the received
      sequence number;
            The object data (Body/End Of Body headers) of the active
            operation shall resume based on the Object Offset TAG from the
            Client; all other headers shall resume based on the SSN of the
            request;
} else if the sequence number of the request is equal to the Next Sequence
Number field then {
     If a GET operation has been resumed then {
          /* Immediately issue the next response packet */
          Send the proper GET response packet to continue the
            Operation;
                 The object data (Body/End of Body headers) shall resume
                 based on the Object Offset TAG from the Client;
     } else {
          /* Wait for the next request packet */
          Process the next packet which will use the Next Sequence
     Number from the RESUME request;
          Send the proper response;
     }
} else {
     /* Sequence number is invalid */
     Disconnect underlying transport connection to OBEX client;
     Remove this session from the list of Suspended Sessions;
}
```

3.4.7.5 SETTIMEOUT

The **SETTIMEOUT** command is used to negotiate a new timeout specifying the number of seconds a session should be maintained while the session is suspended. This command must include a **Session-Parameters** header containing the *Session Opcode* field. Optionally, a *Timeout* field can be included. If the **Session-Parameters** header does not contain the *Timeout* field, the request is for an infinite timeout. The successful response to **SETTIMEOUT** is 0xA0 (Success, with the high bit set) in the response code. A **Session-Parameters** field may be sent with a *Timeout* field. The lowest value for the timeout is the value used by both sides. This command can only be used to change the timeout of the active

connection. If the active connection is the unreliable connection then 0xC3 (Forbidden) is the proper response code.

The **SETTIMEOUT** command and response must fit in a single OBEX packet and have their Final bits set.

3.4.7.6 Generating a Session ID

The purpose of the *Session ID* is to uniquely identify the session. The *Session ID* is used when closing and resuming sessions. Both the client and the server want to be certain that the remote device is referring to the same session when issuing commands, especially when resuming a suspended session. Both the client and the server provide information used to generate the *Session ID*. This helps to make certain that the *Session ID* is unique. Each device provides two pieces of information. The first is its *Device Address*. In the case of IrDA this is the 32-bit device address, which is not guaranteed to be unique. The second is a *Nonce*, which must be unique for the given device. Given that the *Nonce* is unique as far as each device is concerned and that the device addresses are reasonably unique (even in the case of IrDA) the resulting *Session ID* has a high probability of being unique.

The Session ID is generated by applying the MD5 algorithm to the string created by concatenating the *Client Device Address*, *Client Nonce*, *Server Device Address* and *Server Nonce*. The reason for applying an algorithm such as MD5 is so that the size of *Session IDs* is fixed (16 bytes). Device Addresses and Nonces can very in size so the MD5 algorithm is used to produce a 16-byte quantity. The MD5 algorithm was designed to produce a unique number given a string or file. MD5 was chosen since it is used in the OBEX authentication procedure and was very likely to already be present in an OBEX implementation.

3.4.7.7 Negotiating Timeouts

All commands except **CLOSESESSION** involve timeout negotiation. Timeout negotiation is simple. Each time a Session Command exchange occurs (except **CLOSESESSION**) both sides take the smallest timeout values sent and use that as the new timeout for the session. If a timeout value is not explicitly sent it is assumed that the device is requesting an infinite timeout.

3.4.8 Action

The ACTION operation is defined to cover the needs of common actions. The **Action Id** header is used in the **ACTION** operation and contains an action identifier, which defines what action is to be performed. All actions are optional and devices may chose to implement all, some or none of the actions.

The **Action Id** header is described in section 2.2.20 Action Identifier.

Byte 0	Bytes 1, 2	Bytes 3,4	Bytes 5 to n
0x06 (0x86)	Packet length	Action Identifier Header	Sequence of headers

The positive responses to an **ACTION** request are either SUCCESS or CONTINUE depending on the presence of the FINAL bit in the request. Any failure response code can be used to indicate a negative response.

3.3.8.1 Copy Object Action

This action copies an object from one location to another. The **Name** header specifies the source file name and the **DestName** header specifies the destination file name. These two headers are mandatory for this action.

The Action Identifier for the Copy Object Action is specified in section 2.2.20 Action Identifier.

Byte 0	Bytes 1, 2	Bytes 3, 4	Bytes 5 to n		Bytes n to m
Opcode	Packet	Action Identifier	Name Header	DestName	Optional

	length	Header for Copy		(Source Filename)	Header (Destination Filename)		headers
0x06 (0x86)		0x94	0x00		0x15		

Response Codes:

- Success 0x20 (0xA0)
- Not Found 0x44 (0xC4) – source object or destination folder does not exist
- Forbidden 0x43 (0xC3) – cannot read source object or create object in destination folder, permission denied
- Database Full 0x60 (0xE0) - cannot create object in destination folder, out of memory
- Conflict 0x49 (0xC9) - cannot create object in destination folder, sharing violation, object or Copy Object Action reserved/busy
- Not Implemented 0x51 (0xD1) – Copy Object Action not supported
- Not modified 0x34 (0xB4) - cannot create folder/file, destination folder/file already exits

If the **Permissions** header is used with the Copy Object Action, then the permissions of the destination object should be set as specified in the **Permissions** header. If the header is not used, the permissions should be set to be the same as the source.

3.3.8.2 Move/Rename Object Action

This action moves an object from one location to another. It can also be used to rename an object. The **Name** header specifies the source file name and the **DestName** header specifies the destination file name. These two headers are mandatory for this action.

The Action Identifier for the Move/Rename Object Action is specified in section 2.2.20 Action Identifier.

Byte 0	Bytes 1, 2	Bytes 3, 4		Bytes 5 to n			Bytes n to m
Opcode	Packet length	Action Identifier header for Move/Rename		Name header (Source filename)	DestName header (Destination filename)		Optional headers
0x06 (0x86)		0x94	0x01		0x15		

Response Codes:

- Success 0x20 (0xA0)
- Not Found 0x44 (0xC4) – source object or destination folder does not exist
- Forbidden 0x43 (0xC3) – cannot read source object or create object in destination folder, permission denied
- Database Full 0x60 (0xE0) - cannot create object in destination folder, out of memory
- Conflict 0x49 (0xC9) - cannot create object in destination folder, sharing violation, object or Move/Rename Object Action busy
- Not Implemented 0x51 (0xD1) - Move/Rename Object Action not supported
- Not modified 0x34 (0xB4) - cannot create folder/file, destination folder/file already exits

If the **Permissions** header is used with the Move/Rename Object Action, then the permissions of the destination object should be set as specified in the **Permissions** header. If the header is not used, the permissions should be set to be the same as the source.

3.3.8.3 Set Object Permissions Action

This action sets the access permissions of an object or folder. The **Name** header specifies the object and the **Permissions** header specifies the new permissions for this object. These two headers are

mandatory for this action.. When using the Set Object Permissions Action for folders, it will set the permissions only for the folder; it doesn't affect the permissions of the contents of the folder.

The Action Identifier for the Set Object Permissions Action is specified in section 2.2.20 Action Identifier.

Byte 0	Bytes 1, 2	Bytes 3, 4		Bytes 5 to n			Bytes n to m
Opcode	Packet length	Action Identifier header for Set Object Permissions		Name header	Permissions header		Optional headers
0x06 (0x86)		0x94	0x02		0xD6		

Response Codes:

- Success 0x20 (0xA0)
- Not Found 0x44 (0xC4) – source object or destination folder does not exist
- Forbidden 0x43 (0xC3) – cannot modify the permissions of the destination object/folder, permission denied
- Not Implemented 0x51 (0xD1) – Set Object Permissions Action not supported
- Conflict 0x49 (0xC9) - cannot modify the permissions, sharing violation, object or Set Object Permissions Action busy

3.3.8.4 Common Issues For Move/Rename, Copy and Set Object Permissions Actions

The operations and actions on files within a file structure are complicated when some of the files are protected. Ideally the command should succeed or fail completely. Unfortunately, some devices may not be able to restore deleted files if an error occurs in a multi-file delete operation. The following defines what should happen in the event that the entire operation cannot be completed or failed as a single operation.

When a **PUT** operation is used to delete a folder, all of the subfolders beneath it should be deleted. If a file or folder within the structure is protected (deleting forbidden), then the file/folder itself and the folders in the tree above the file/folder should not be deleted. All of the other files and folders in the subfolders that are not protected should be deleted. The error response FORBIDDEN should be returned in this case.

A Move Action of a folder should also move the contents of all the subfolders beneath it. If a file or folder within the structure is protected (deleting forbidden), then the file/folder itself and the folders in the tree above the file/folder should not be deleted. Instead, new versions of the protected files/folders should be created in the destination folder. All of the other files in the folders that are not protected should be moved. The error response FORBIDDEN should be returned in this case.

A Copy Action of a folder should also copy the contents of all the subfolders beneath it. If a file or folder within the structure is protected (reading forbidden), then the folder itself and the files/folders beneath it will not be copied. The same principle applies to Move action when reading of the source file/folder is forbidden. The error response FORBIDDEN should be returned.

When a folder is moved or copied, the current folder is not changed, even if this folder is no longer valid. For example if you are in the folder "\blah" and you move the folder "\blah" to "\blah2", the current folder is still "\blah".

Files and folders at the same folder level MUST not have the same name, i.e. it must be obvious by the name that they are different, and not implicitly known by the operating system of the file store.

The client must preserve case between the folder listings returned and the names sent in the Name and **DestName** headers, as the remote file system may be case sensitive.

3.5 Packet Timeouts

OBEX does not impose any timeouts between OBEX packets. Deciding whether to continue or cancel a session is normally left to user control, since waiting for user action is often what creates long periods between packets. In devices with reasonable user interface capabilities, timeouts are not recommended. However this specification does not prohibit the use of timeouts. Some devices may find timeouts useful or even necessary, particularly when sufficient user interface to understand and/or control the reason for delay is not available.

3.6 Authentication Procedure

The OBEX authentication procedure is based on two OBEX headers, the **Authenticate Challenge** and **Authenticate Response** headers. It is assumed that the client and the server both share a secret such as a password or pin number. This value is not sent over OBEX during the authentication process.

The client authenticates the server by sending an OBEX command with an **Authenticate Challenge** header. This header will contain the digest-challenge, which is described later. To be authenticated, the server's response packet must contain the SUCCESS response code and an **Authenticate Response** header. The content of the **Authenticate Response** header contains the digest-response string, which is described later. The client verifies the server by generating its own request-digest string and comparing it to the one sent as part of the digest-response. If the server is not authenticated the client can simply disconnect from the server.

The server authenticates the client using the same basic algorithm. When a client attempts an operation to the server which requires authentication, the server sends an **Authenticate Challenge** header along with an UNAUTHORIZED response code in a response packet. When the client receives this response, it must re-send the command packet with an **Authenticate Response** header. The server verifies the client by generating its own request-digest and comparing it to the one sent in the digest-response by the client. If they are the same, the client is authenticated. If the client is not authenticated, the server will simply respond to all operations, including the current one, with the UNAUTHORIZED response code.

The algorithm used by the client to authenticate a server is straightforward. The client will normally authenticate the server as part of the OBEX **CONNECT** procedure. The server can authenticate the client for any operation including the **CONNECT** operation and individual **PUT** and **GET** operations. When the server authenticates the client for an OBEX connection the server will send the **Authenticate Challenge** header in the response to the **CONNECT** command with a response code of UNAUTHORIZED. When the client receives a response to the **CONNECT** command containing an **Authenticate Challenge** header and the UNAUTHORIZED response code, it should repeat the **CONNECT** command and include an **Authenticate Response** header. If the original **CONNECT** command also contained an **Authenticate Challenge** and/or **Target** header, these headers must also be present in the second **CONNECT** command.

When the server wants to authenticate an individual operation, it will reject the first attempt by the client with an OBEX response containing the UNAUTHORIZED response code and an **Authenticate Challenge** header. The client must retry the operation this time including an **Authenticate Response** header. If every operation requires authentication then it probably makes sense to authenticate the **CONNECT** operation and not perform individual operation authentication.

Multiple **Authenticate Challenge** headers can be sent in one OBEX packet. This is done for a number of reasons:

- In the future, there may be different types of challenge algorithms or hashing functions used. Each challenge will represent a possible algorithm to use.
- There may be different passwords for full access versus read only access. Two headers are used, one for each access method.

When more than one **Authenticate Challenge** header exists in an OBEX packet, the nonce must be different for each one. Devices are not required to deal with multiple **Authenticate Challenge** headers so some devices may only process the first one. Therefore, it is important that the most general or common challenge be sent in the first header. When multiple **Authenticate Challenge** headers are sent, the sender will use the nonce in the digest-response to determine which nonce to use when generating the request-digest. If a nonce does not exist in the digest-response then the nonce sent in the first **Authenticate Challenge** header is used.

3.6.1 Digest Challenge

The OBEX digest-challenge is a simplified version of the HTTP digest-challenge. One of the limitations of OBEX is that the response to a **PUT** or **CONNECT** command must fit in a single OBEX packet. Some devices only support a maximum size OBEX packet of 255 bytes so this limits the amount of data that can fit in a response and therefore, limits the size of the digest-challenge and digest-response.

The HTTP challenge contains a number of fields including a string that identifies the challenge as basic or digest. Within a digest challenge the digest-challenge can specify the hashing algorithm to use, the realm, the domain, a nonce, and option fields. The main element is the nonce. The nonce is the random string, which is combined with the secret to produce the main portion of the digest-response. The OBEX authentication algorithm needs to be simple yet flexible. The minimal challenge should only contain a nonce but there may be cases where a userid/username is needed to determine which password to use. Some OBEX servers may support multiple different users each with their own passwords. Therefore, the server must be able tell the client that a userid is needed.

The default hashing function in HTTP is MD5 and OBEX will use this algorithm. In the future, other algorithms may be introduced but at this time, only MD5 is supported.

The OBEX digest-challenge string can contain multiple fields. The HTTP digest-challenge is based on ASCII strings. To keep the size small, the OBEX digest-challenge will use a tag-length-value encoding scheme. Both the tag and value length fields, are one byte in length. The table below shows the OBEX digest-challenge fields.

Tag	Name	Value Len	Value Description	Default Value
0x00	Nonce	16	String of bytes representing the nonce.	
0x01	Options	1	Optional Challenge Information	0
0x02	Realm	*n*	A displayable string indicating which userid and/or password to use. The first byte of the string is the character set to use. The character set uses the same values as those defined in IrLMP for the nickname.	

Each field is described in more detail below.

3.6.1.1 Nonce

The nonce is required. It is important that the nonce be different each time it is sent. An example nonce, based on the example given in the HTTP authentication document, is as follows:

```
H(time-stamp ":" password)
```

The time-stamp is a sender-generated time or other non-repeating value and the password is the secret known by both the client and server. The password is not null-terminated and shall be encoded according to the **UTF-8** standard. The MD5 hashing algorithm is applied to the time-stamp and **UTF-8** encoded password string to produce a 128 bit (16 byte) binary string. This resulting string is the nonce.

3.6.1.2 Options

The option field is optional. If an option field is not sent the assumed value for it is 0. Two options are defined. The first controls the sending of a userid. If this option is set then the entity receiving the challenge must respond with a userid in the digest response string. The second option indicates the access mode being offered by the sender of the challenge. The two access modes are full access (both read and write) and read only.

bit	Meaning
0	When set, the User Id must be sent in the authenticate response.
1	Access mode: Read Only when set, otherwise Full access is permitted.
2	Reserved
3	Reserved
4	Reserved
5	Reserved
6	Reserved
7	Reserved

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

3.6.1.3 Realm

The realm field is optional. The realm is intended to be displayed to users so they know which userid and password to use. The first byte of the string is the character set of the string. The table below shows the different values for character set.

Char Set Code	Meaning
0	ASCII
1	ISO-8859-1
2	ISO-8859-2
3	ISO-8859-3
4	ISO-8859-4
5	ISO-8859-5
6	ISO-8859-6
7	ISO-8859-7
8	ISO-8859-8
9	ISO-8859-9
0xFF = 255	UNICODE (UTF-16)

3.6.2 Digest Response

The digest-response string can contain multiple fields and uses a tag-length-value encoding scheme. The digest-response fields are shown below.

Tag	Name	Value Len	Value Description	Default Value
0x00	Request -digest	16	String of bytes representing the request digest.	

0x01	User Id	*N*	User ID string of length n. Max size is 20 bytes.	
0x02	Nonce	16	The nonce sent in the digest challenge string.	

Each field is described in more detail below.

3.6.2.1 Request-digest

The request-digest is required and is calculated as follows:

```
H(nonce ":" password)
```

The nonce is the string sent in the digest-challenge. The password is the secret known by both the client and server. The password is not null-terminated and shall be encoded according to the **UTF-8** standard. The MD5 hashing function is applied to the nonce and **UTF-8** encoded password string to produce a 128-bit (16 byte) string. This resulting string is the request-digest.

3.6.2.2 Userid

The userid is required if the digest-challenge contains an options field with the userid bit set to 1. The userid is used to identify the proper password. The userid is not null-terminated and shall be encoded according to the **UTF-8** standard.

3.6.2.3 Nonce

The nonce is optional. When multiple **Authenticate Challenge** headers are sent, the nonce is used to indicate to which header the request-digest is responding. The nonce is the same value as sent in the digest-challenge. If a nonce is not sent in the digest-response then it is assumed that the digest-response is a response to the first **Authenticate Challenge** header. This allows devices that do not parse multiple **Authenticate Challenge** headers to work with devices that send multiple headers.

3.6.3 Hashing Function

The default hashing function used in OBEX is MD5. The following paragraph briefly describes the MD5 algorithm. Refer to RFC 1321 at http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1321.txt for more detailed information, including example source code. Additional MD5 source code can be found in Appendix 11.5 MD5 Algorithm for Authentication at the end of this specification.

The MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

The MD5 algorithm is designed to be quite fast on 32-bit machines. In addition, the MD5 algorithm does not require any large substitution tables; the algorithm can be coded quite compactly.

3.6.4 Authentication Examples

Example 1: Client and Server Connection Authentication (no userid)

	Opcode or Response plus Headers	Final bit	Header Data	Header Length
Request →	**CONNECT**			
	Authenticate Challenge		0x00, 0x10, <nonce>, 0x01, 0x01, 0x00	24
	Target		<target-UUID>	19
Response ←	UNAUTHORIZED	4		
	Authenticate Challenge		0x00, 0x10, <nonce>, 0x01, 0x01, 0x00	24
Request →	**CONNECT**			
	Authenticate Challenge		0x00, 0x10, <nonce>, 0x01, 0x01, 0x00	24
	Authenticate Response		0x00, 0x10, <request-digest>	21
	Target		<target-UUID>	19
Response ←	SUCCESS	4		
	Connection Id		0x00000001	5
	Who		<target-UUID>	19
	Authenticate Response		0x00, 0x10, <request-digest>	21

Example 2: Server Authenticates Get operation (userid required)

	Opcode or Response plus Headers	Final bit	Header Data	Header Length
Request →	**GET** **Name**	4	 "Test File Object"	 37
Response ←	UNAUTHORIZED **Authenticate Challenge**	4	 0x00, 0x10, <nonce>, 0x01, 0x01, 0x01	 24
Request ←	**GET** **Name** **Authenticate Response**	4	 "Test File Object" 0x00, 0x10, <request-digest>, 0x01, 0x08, <userid>	 37 31
Response ←	CONTINUE **Length** **Time** **Body**	4	 "2000" "0x41a50016" "… start of file data…."	 499

3.7 Multiplexing with OBEX

It is sometimes necessary for different clients to perform operations in a manner that appears to the user to be simultaneous. In addition, some connections may be very long lived while others come and go quickly. This creates the need for a mechanism that allows multiple clients to access OBEX services concurrently. The following sections cover two methods recommended for multiplexing commands over OBEX. The first method is geared toward use with applications that run under the umbrella of the default OBEX server.

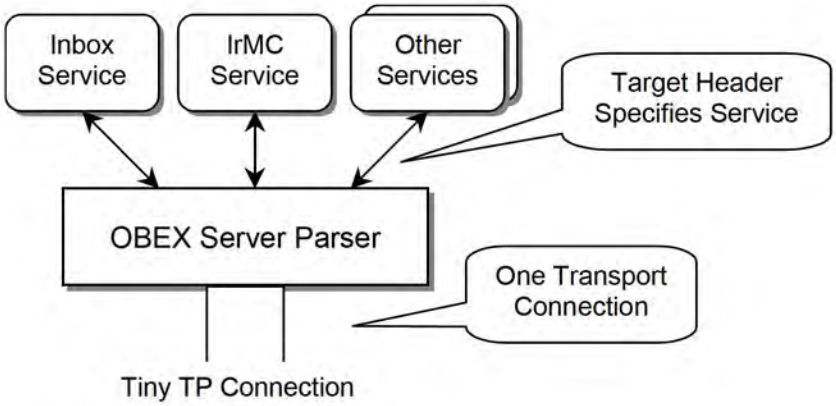


Figure 3.6.1: OBEX Command Level Multiplexing

3.7.1 Connections multiplexed at the OBEX Command level

The OBEX protocol layer provides this form of multiplexing. Multiple OBEX connections (using the **Connection Id** header) are allowed at one time but only one command can be executed at a given time. This type of multiplexing only requires one OBEX parser because the multiplexing occurs above the parser. This type of multiplexing is necessary because most services are accessed through the default OBEX server and hence over a single transport connection.

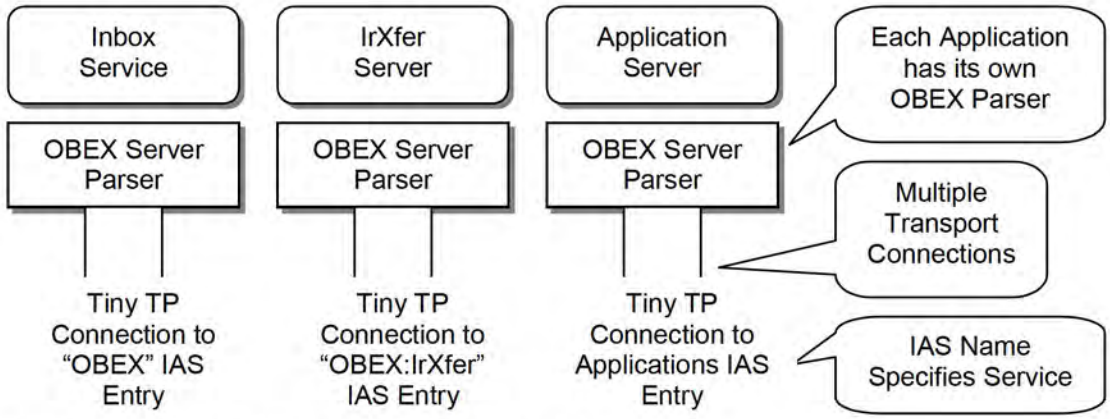


Figure 3.6.2: OBEX Transport Layer Multiplexing

3.7.2 Connections multiplexed at the OBEX Transport layer

This type of multiplexing is external to OBEX and relies on the multiplexing capabilities of the underlying transport layer. Multiple OBEX servers exist, one for each different application. In turn each of these OBEX servers may themselves offer the connection and multiplexing types described above. In IrDA for example, the LM-MUX provides this form of multiplexing. Multiplexing at the transport layer is most useful when neither side of the application are constrained or limited by resources. It also allows for a finer grained multiplexing which may be important for some applications.

4. OBEX Application Framework

The OBEX application framework is a set of conventions and services designed for the purpose of creating interoperable devices. OBEX is a very flexible protocol and the application framework tightens up the usage of OBEX while creating a set of standard services that satisfy many object exchange needs. The foundation of the OBEX application framework is the Default OBEX Server.

4.1 The Default OBEX Server

The Default OBEX Server is the server that resides at the LSAP-Sel specified by the IAS entry with class name "OBEX" (see section 6.1 IAS entry for details on this IAS entry). The Default OBEX server is the "Well Known" OBEX server. All the standard services along with many applications are accessed via this server including IrMC applications. For general device to device interoperability the default OBEX server is used. There can only be one entity registered to be the default OBEX server. If more than one exists it will be impossible for another device to distinguish between the two.

The default OBEX server can be used to access file systems, databases, services and applications. OBEX provides conventions for distinguishing the types of accesses. The **Name**, **Type**, **Target** and **Connect Id** headers can be used for this purpose. The most basic service accessed via the default OBEX server is the inbox service.

4.2 The Inbox Service

The most basic form of object exchange is pushing a file/object to another device. The sending device (client) is only responsible for making sure the object is transferred correctly. The client does not know or care what happens to the object after it has been received. The receiver (server) is responsible for placing the object in the correct location. This basic from of file transfer is accomplished with the OBEX **PUT** operation. The client does not need knowledge of the server's object store or folder hierarchy. The client initiates the operation by sending the object to the default OBEX server. When pushing an object, a **Name** header is used to name the object and a **Type** header may be used to identify the type. Other headers such as **Length** and **Time** can be also sent.

In this simple form of object transfer, the object is pushed into the server's "inbox". Inboxes come in many forms. A PC based application could have a folder in the PC's file system, which represents the inbox. But, an inbox does not necessarily have to be a fixed folder or location. It is also possible to dispatch the object based on the **Type** header or the extension in the **Name** header to a more appropriate location such as an application or database. Products, which do not have a file system, can operate in this way. The structure of the inbox is an issue strictly for the server. The client simply pushes objects to the "inbox". What happens after that is up to the server. The push model works best for single objects. It can also be used for moving whole folder/directory hierarchies.

It is important to note that what may be an application specific object for one device is just a file in another device. Thus, it is important that all devices standardize on using the default OBEX server for performing this simple operation. In this way, interoperability is more likely to occur. The capability service is also used to help increase the interoperability for simple object pushing. Refer to chapter 8.2 Simple OBEX Put file transfer (+ SetPath) for more information on the Inbox model.

The Inbox service also enables a device to **GET** certain objects from another device without understanding the semantics of the service or database that contains the object. This is called the default **GET** and is intended for use with a narrow band of object types that have a default object. A vCard is a good example of an object type that has a default object.

4.3 Inbox Connection

The inbox is accessed via an OBEX connection called the "inbox" connection. This connection is made to the default OBEX server by sending a **CONNECT** command with no **Target** headers. Objects sent using Ultra are placed in the inbox but in this case the **CONNECT** command is not used. Other services are also accessed via the inbox connection. These services all have specific characteristics that allow an implementation to distinguish between the different service requests. The table below gives the complete list of services available via the inbox connection along with the distinguishing characteristics.

Service type	Service description
Capability Service	The capability object is accessed by using a **GET** command with a **Type** header containing the MIME type of the capability object, "x-obex/capability".
IrMC Level 2 and 3	Level 2 and 3 services of the IrMC application are accessed using **PUT** and **GET** commands with a **Name** header in which the first part the name contains "telecom/".
Pushing objects into the inbox	Objects are pushed into the inbox by using the **PUT** command with a **Name** header. The string in the **Name** header should not contain any path characters such as ':', '/' or '\'. Objects with improperly formed names should be rejected.
Pulling objects from the inbox	Default objects are pulled from the inbox by using the **GET** command with a **Type** header containing the MIME type of the desired object. **Name** headers are not allowed.

4.4 Capability Service

The OBEX capability service is designed to provide a general-purpose method for accessing service information with OBEX. The capability service lists the type of objects supported and details about the fields or formats of specific types. A client may be able to render an object into multiple formats. By reading the server's capability object the client can determine the best format to send the object. The client can also determine if it makes sense to even send an object at all.

The capability service is based upon two OBEX objects, the capability object and the object profile object. The capability object contains general-purpose information about the device, including the services and applications that are supported. The object profile object contains information specific to the objects that the device supports. The details of the capability service and its objects can be found in sections 8.5 Capability Service, 9.3 The Capability Object and 9.4 The Object Profile Object.

4.5 Custom OBEX applications

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 the default OBEX server. 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 server inbox, 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.

4.6 Directed Operations and Connections

A directed operation is an operation that is sent toward a specific service. An OBEX **Target** header is used to identify the designated service. If the receiving side has a matching service, all packets in the request are forwarded (or re-directed) to that service. In addition, a **Who** header is sent in the response to indicate that the request is being serviced by the requested service. If a matching service cannot be located, the server is encouraged to accept the request if at all possible. This allows the client to make the final decision on whether to continue the operation or abort. In this case a **Who** header is not returned in the response. When the operation is complete, the association is dissolved. If the client wishes to direct another operation to the service it must again include the **Target** header. This is where directed connections come in handy.

Directed connections enable a client and server to establish a virtual connection based on the principles of directed operations. They exist for clients that need to perform multiple operations to a specific service. Once a directed connection is established, the client can direct operations to the same service by simply including the **Connection Id** header in each request. A directed connection is initiated when a client sends an OBEX **CONNECT** packet with a **Target** header containing the UUID of the service it wishes to connect to. A successful match is indicated with a **Who** header in the response packet as well as a **Connection Id** header. The **Connection Id** header contains the identifier of the connection that the client will use in future requests as shorthand for a **Target** header.

4.6.1 Directed Connection Mechanics

Directed connections do not follow the usual paradigms one would associate with a "connection". It is recommended that directed connections always be "open". Therefore, it is unnecessary to close a connection with an OBEX **DISCONNECT** operation. In essence, a directed connection provides a shortcut so the client doesn't have to specify a **Target** header in every operation. As well as providing insurance that the designated service will receive the clients requests.

It is not necessary to track the open or closed status of directed connections. Therefore OBEX **CONNECT** operations should not be reference counted and matched with OBEX **DISCONNECT** operations.

It may or may not be necessary for a server to differentiate an operation that arrives with a **Connection Id** from one that arrives with a **Target** header for the same service (other than making sure to respond with the **Who** header). This is unlikely to be an issue in normal implementations but it should be noted that the acceptance of a directed connection by a service, does not guarantee that another operation sent with a **Target** header to the same service will be accepted by that service.

It is anticipated that **CONNECT** operations that are intended to establish directed connections will contain the same flags, OBEX packet size and protocol versions as did all previous **CONNECT** operations. Handling of **CONNECT** options that differ from original or previous values is implementation dependent.

4.6.2 Target Header Processing

In some cases, more than one **Target** header may be present in a request. The server should match the headers in the order received and ignore any **Target** headers encountered after a match is made. The matching header is then sent in the response **Who** header.

4.7 OBEX Access Methods

Since not all objects are the same, there arises the need to define methods for accessing different types of objects. This section defines sets of standard procedures or templates for each type of access. Three basic forms of access have been identified: File, Database and Process. Interoperability will become

easier when, for example, accessing a database all clients follow the same basic procedures. Of course, the values in the headers and such will differ for each database but the basic procedure should be the same.

4.7.1 File Access

File level access is the lowest level of object access provided by OBEX. Using file level access an object can be retrieved and stored only as a whole. It relies on addressing an object by providing a **Name** and optionally a **Type** header. The **Name** header identifies the instance of the object. This is often sufficient when sending an object. The **Type** header is used to refine the objects identification when multiple objects may exist with the same name.

4.7.2 Database Access

A database is different from a file system. The objects manipulated in a file system tend to be course blobs. While the items in a database are more refined. Often, the individual fields and records can be manipulated in a database. Whereas many different types of objects are stored in a file system, databases tend to store or organize objects of a similar nature.

OBEX database level access allows for finer granularity access to objects. Objects accessed using database level headers can be manipulated by part. In other words pieces of a database object can be requested and stored instead of having to exchange the entire object.

Database access relies on a **Name** header to identify the instance of the database object. In addition, OBEX has an **Object Class** header, which is designed for use with database objects.

4.7.3 Process / RPC

In interacting with a process, the actual information transferred is generated and consumed on the fly. The best example is the Synchronization command interpreter in the IrMC specification. The items being **PUT** are often commands and not objects.

5. 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, some devices support only connectionless operations, such as those described by the IrDA Ultra proposal.

Ultra refers to communications carried out over connectionless data services of IrLAP, as described in "Guidelines for *Ultra* Protocols". The Ultra protocol is not reliable and does not guarantee ordered delivery, so the request/response model of a full OBEX implementation is not appropriate. However, for delivery of small objects, the simple structure of OBEX requests and the header based object wrapper are still useful. A number of issues arise and are discussed in the following paragraphs.

Ultra does not assure reliable ordered delivery. However, it does use IrLAP framing, which has a CRC, so it is possible to tell if an individual packet arrived intact. It also uses a segmentation-reassembly (SAR) counter in each packet header, and a maximum time constraint between consecutive packets using SAR. The count allows up to 15 packets to be reassembled. The CRC and SAR acting together put a maximum upper bound on the size of an Ultra OBEX packet of 15 times the max packet size permitted by the link speed of either 2400 or 9600 bps. Connectionless IrLAP packets are limited to 64 bytes, of which 2 bytes are used by IrLMP and 2 bytes are used by the PID and SAR fields. Thus leaving 15 packets of 60 bytes (900 bytes total) for use by Ultra OBEX. In practice, transfers of 15 packets will take too long, and Ultra OBEX is best suited for very small objects that will fit in one (or a few) packets. A typical well-suited operation would be beaming a single name and phone number between two devices.

Ultra OBEX uses only the OBEX **PUT** command and all object data and headers must be sent in one OBEX packet. The packet must have the final-bit set and all object data is contained in a single **End-of-Body** header. Minimum implementations of Ultra OBEX need only support the default OBEX packet size of 255 bytes. Legal implementations can however, support up to the maximum Ultra OBEX packet size of 900 bytes. For compatibility, it is recommended that devices do not send more than 255 bytes over Ultra.

To guarantee a successful transfer of data over Ultra, the exchange should be kept to 255 bytes or less . This includes all headers. The data should be contained in one OBEX packet. Meaning the first and only frame contains the **PUT**+FINAL opcode.

NOTE: Implementations are not required to receive OBEX packets larger than 255 bytes. Possible ways that implementations can deal with Ultra OBEX packets larger than 255 bytes is as follows:

1. Ignore the packet completely
2. Only accept the first 255 bytes
3. Accept the whole packet.

The OBEX **CONNECT**, **DISCONNECT, ABORT** and **SETPATH** operations are not used in Ultra OBEX, because Ultra OBEX does not have any notion of a session with multiple operations. Each operation is completely distinct from every other, and each operation must fit within the SAR constraint described above. Therefore, any version or capability information must be included as headers in the operation being performed.

Ultra OBEX does not use any response packets. All feedback as to the success or failure of the operation is performed by some out-of-band means such as a beep or visual indication to the user that the operation was successful. For this reason, the OBEX **GET** operation is also not used in Ultra OBEX.

6. IrDA OBEX IAS entries, service hint bit and TCP Port number

6.1 IAS entry

The 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 class name should be used only by implementations that are intended as default OBEX servers, and in particular have some provision for dealing with multiple object types and services. Special purpose programs (custom applications) that use this protocol should define unique IAS class names that they alone will use. Alternatively, special purpose applications can be accessed via the default OBEX server by utilizing a directed connection as discussed in this specification. There is one very common subset of object exchange application - applications that are strictly for file exchange. They should use a class name 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 class name the peer normally uses, and if not found should look for the default OBEX server to accept the transfer.

6.1.1 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 class name is.

6.2 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. Setting the OBEX hint bit is required. Many OBEX clients will not even attempt a connection unless the OBEX hint bit is set.

6.3 TCP port number

IANA has assigned the port number 650 to the OBEX protocol. This port number should be used when transporting OBEX protocol data over a TCP network. More specifically, it represents the location of the default OBEX server in the TCP network. As with TinyTP, there is no requirement on the alignment of OBEX packets with TCP PDUs. Except that all OBEX data shall be carried in data packets, not in TCP Connect or Disconnect packets.

7. OBEX Examples

The following examples are provided to round out the readers understanding of the procedures used in OBEX.

7.1 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	0x80	**CONNECT**, Final bit set
	0x0007	7 bytes is length of packet
	0x10	version 1.0 of OBEX
	0x00	no connect flags
	0x2000	8K max packet size
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x0007	packet length of 7
	0x10	version 1.0 of OBEX
	0x00	no connect flags
	0x0800	2K max packet size
Client Request:	**bytes**	**Meaning**
opcode	0x02	**PUT**, Final bit not set
	0x0422	1058 bytes is length of packet
	0x01	HI for **Name**
	0x0017	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:		
response code	0x90	CONTINUE, Final bit set
	0x0003	length of response packet
Client Request:		
opcode	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:		
response 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	0x82	**PUT**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	next 1K bytes of body
Server Response:		
response code	0xA0	SUCCESS, Final bit sent
	0x0003	length of response packet

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 the server assumes a binary file and stores it exactly as is.

7.2 Simple Get - field data collection

This example illustrates a **GET** operation with user defined headers, which, in this case, are for application specific access identifier 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	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 access header
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x000B	packet length of 11
	0x10	version 1.0 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	0x83	**GET**, Final bit set
	0x0003	length of **GET** packet

Server Response:		
Response code	0xA0	SUCCESS, Final bit set
	0x0038	length of response packet
	0x49	HI for **End-of-Body** chunk
	0x0035	Length of 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.

7.3 Example Get of the Capability Object

This example shows a client requesting the capability object from a device. The OBEX packet size is 512 bytes.

	Opcode or Response plus Headers	Final bit	Header Data	Header Length	Running Total of Data Exchanged
Request →	**GET** **Type**	4	"x-obex/capability	20	
Response ←	CONTINUE **Length** **Body**	4	"2000" "start of capability object"	499	496 bytes
Request →	**GET** no headers	4			
Response ←	CONTINUE **Body**	4	"..continuation of object.."	509	1002 bytes
Request →	**GET** no headers	4			
Response ←	CONTINUE **Body**	4	"..continuation of object.."	509	1508 bytes
Request →	**GET** no headers	4			
Response ←	SUCCESS **End-Of-Body**	4	"..final segment of object"	495	2000 bytes

7.4 Connect using Target, Who and Connection Id headers

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. This illustrates the format of a directed Connection, using the **Target**, **Who** and **Connection Id** headers. The connection was accepted by the Targeted application.

Client Request:	**bytes**	**Meaning**
opcode	0x80	**CONNECT**, Final bit set
	0x0023	packet length = 35
	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
	0x46	**Target** HI
	0x0013	Length of **Target** Header
	0x382D2BD03C39	UUID for desired service/application (binary
	11D1AADC0040F6	representation)
	14953A	{382D2BD0-3C39-11d1-AADC-0040F614953A}
	0xC3	HI for **Length** header (optional header)
	0x0000F483	total length of hex F483 bytes
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x0023	packet length of 35
	0x10	version 1.0 of OBEX
	0x00	flags (LSAP-SEL multiplexing not supported)
	0x0800	2K max packet size
	0xCB	HI for **Connection Id** header
	0x00000001	ConnId = 1
	0x4A	**Who** HI
	0x0013	Length of **Who** Header
	0x382D2BD03C39	UUID of the responding application (same value as
	11D1AADC0040F6	**Target** header in request)
	14953A	{382D2BD0-3C39-11d1-AADC-0040F614953A}
Client Request:		
opcode	0x02	**PUT**, Final bit not set
	0x0427	1063 bytes is length of packet
	0xCB	HI for **Connection Id** header
	0x00000001	ConnId = 1
	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	Length of **Body** header (1K) plus HI and header length
	0x.....	1K bytes of body
Server Response:		
response code	0x90	CONTINUE, Final bit set
	0x0003	length of response packet
	The PUT operation	…proceeds to completion…

Client Request:	**bytes**	**Meaning**
opcode	0x81	**DISCONNECT**, Final bit set
	0x0008	packet length = 8
	0xCB	HI for **Connection Id** header
	0x00000001	ConnId = 1
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x0003	packet length = 3

7.5 Combined Get and Put - paying for the groceries

Now that you have the idea of what the byte sequence 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 briquettes. 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...

7.6 Create Session Followed by a Directed Connection

Client Request:	**bytes**	**Meaning**
opcode	0x87	**SESSION**, Final bit set
	0x0015	packet length = 21
	0x52	**Session-Parameters** HI
	0x0012	Length of **Session-Parameters** header (18 bytes)
	0x05, 0x01,	Client *Session Opcode* field (1-byte)
	0x00	Create Session opcode
	0x00, 0x04, 0xXXXXXXXX	Client *Device Address* field (IrDA size)
	0x01, 0x04, 0xYYYYYYYY	Client *Nonce* field (4 bytes)
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x0024	packet length of 36
	0x52	**Session-Parameters** HI
	0x0021	Length of **Session-Parameters** header (33 bytes)
	0x00, 0x04, 0xXXXXXXXX	Server *Device Address* field (IrDA size)
	0x01, 0x04, 0xYYYYYYYY	Server *Nonce* field (4 bytes)
	0x02, 0x10, 0xZZZZZZZZZZZZZZZZ ZZZZZZZZZZZZZZZZ	*Session ID* field (16 bytes)
Client Request:		
opcode	0x80	**CONNECT**, Final bit set
	0x001C	packet length = 28
	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
	0x93	**Session-Sequence-Number** HI
	0x00	First packet sent by Client has sequence number of 0
	0x46	**Target** HI
	0x0013	Length of **Target** Header
	0x382D2BD03C3911D1 AADC0040F614953A	UUID for desired service/application {382D2BD0-3C39-11d1-AADC-0040F614953A}
Server Response:		

response code	0xA0	SUCCESS, Final bit set
	0x0021	packet length of 33
	0x10	version 1.0 of OBEX
	0x00	flags (LSAP-SEL multiplexing not supported)
	0x0800	2K max packet size
	0x93	**Session-Sequence-Number** HI
	0x01	Next OBEX packet expected by Server is 1
	0xCB	**Connection Id** HI
	0x00000001	ConnId = 1
	0x4A	**Who** HI
	0x0013	Length of **Who** Header
	0x382D2BD03C3911D1 AADC0040F614953A	UUID of responding application (same value as **Target** header in request){382D2BD0-3C39-11d1-AADC-0040F614953A}

7.7 Suspend a Session

Note that **Session-Sequence-Number** headers are not used when sending Session commands in an active session.

Client Request:	**Bytes**	**Meaning**
opcode	0x87	**SESSION**, Final bit set
	0x0009	packet length = 9
	0x52	**Session-Parameters** HI
	0x0006	Length of **Session-Parameters** header (6 bytes)
	0x05, 0x01,	Client *Session Opcode* field (1-byte)
	0x02	Suspend Session opcode
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x000C	packet length of 12
	0x52	**Session-Parameters** HI
	0x0009	Length of **Session-Parameters** header (9 bytes)
	0x04, 0x04,	*Timeout* Field
	0x000004B0	Requesting timeout of 20 minutes (1200 seconds)

7.8 Immediate Suspend Session during an Operation

It is assumed that a Reliable Session has already been setup using the Create Session operation (see section 7.6). Note that **Session-Sequence-Number** headers are not used when sending Session commands in an active session.

Client Request:	**Bytes**	**Meaning**
opcode	0x02	**PUT**, Final bit not set
	0x0424	1060 bytes is length of packet
	0x93	HI for Session-Sequence-Number
	0x00	SSN 0
	0x01	HI for **Name**
	0x0017	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:		
Response code	0x90 0x93 0x01 0x0005	**CONTINUE**, Final bit set HI for Session-Sequence-Number SSN 1 length of response packet
Client Request:		
opcode	0x02 0x0408 0x93 0x01 0x48 0x0403 0x.....	**PUT**, Final bit not set 1032 bytes is length of packet HI for Session-Sequence-Number SSN 1 HI for Object **Body** chunk Length of **Body** header (1K) plus HI and header length next 1K bytes of body

Two outstanding requests are allowed with an Immediate Suspend

Client Request:	**Bytes**	**Meaning**
opcode	0x87 0x0009 0x52 0x0006 0x05, 0x01, 0x02	**SESSION**, Final bit set packet length = 9 **Session-Parameters** HI Length of **Session-Parameters** header (6 bytes) Client *Session Opcode* field (1-byte) **Suspend Session** opcode
Server Response:		Response to the outstanding Put request
Response code	0x90 0x0005 0x93 0x02	**CONTINUE, Final bit set** length of response packet HI for Session-Sequence-Number SSN 2
Server Response:		Response to the outstanding Suspend Session request
Response code	0xA0 0x000C 0x52 0x0009 0x04, 0x04, 0x000004B0	**SUCCESS**, Final bit set packet length of 12 **Session-Parameters** HI Length of **Session-Parameters** header (9 bytes) *Timeout* Field Requesting timeout of 20 minutes (1200 seconds)

7.9 Resume a Session

If a session was gracefully suspended using the **SUSPENDSESSION** command then resuming the session does not require retransmission of OBEX packets. If the session was suspended because the underlying transport was lost or by issuing the **SUSPENDSESSION** command during an active operation it might be necessary to retransmit packets.Three cases exist. The first case is when the last client request is lost. The second case is when the last server response is lost. The third case is when the client receives the last response but the server does not know if the client has received it (packets do not need to be retransmitted in this case). In the first case the client knows the server did not receive the request because the *Next-Sequence-Number* field in the response to the RESUMESESSION command indicates this, so the client will retransmit the command. In the second and third cases the server knows what to do based on the value of the *Session-Sequence-Number* field in the first packet sent by the client.

The example below shows the sequence needed to resume the session. Retransmission of packets is not shown.

Client Request:	**Bytes**	**Meaning**
Opcode	0x87	**SESSION**, Final bit set
	0x0027	packet length = 39
	0x52	**Session-Parameters** HI
	0x0024	Length of **Session-Parameters** header (36 bytes)
	0x05, 0x01,	Client *Session Opcode* field (1-byte)
	0x03	Resume Session opcode
	0x00, 0x04, 0xXXXXXXXX	Client *Device Address* field (IrDA size)
	0x01, 0x04, 0xYYYYYYYY	Client *Nonce* field (4 bytes)
	0x02, 0x10, 0xZZZZZZZZZZZZZZZZ ZZZZZZZZZZZZZZZZ	*Session ID* field (16 bytes)
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x0027	packet length of 39
	0x52	**Session-Parameters** HI
	0x0024	Length of **Session-Parameters** header (36 bytes)
	0x00, 0x04, 0xXXXXXXXX	Server *Device Address* field (IrDA size)
	0x01, 0x04, 0xYYYYYYYY	Server *Nonce* field (4 bytes)
	0x02, 0x10, 0xZZZZZZZZZZZZZZZZ ZZZZZZZZZZZZZZZZ	*Session ID* field (16 bytes)
	0x03, 0x01, 0x05	*Next Sequence Number* field (1-byte). As an example, the server expects 0x05

7.10 Resume Session for an interrupted PUT operation using SRM

It is assumed that a Reliable Session exists, Single Response mode has already been setup using the Connect operation (see section 7.11), and an ongoing PUT operation has been interrupted by a spurious transport disconnection.

Client Request:	**Bytes**	**Meaning**
Opcode	0x87	**SESSION**, Final bit set
	0x0027	packet length = 39
	0x52	**Session-Parameters** HI
	0x0024	Length of **Session-Parameters** header (36 bytes)
	0x05, 0x01,	Client *Session Opcode* field (1-byte)
	0x03	Resume Session opcode
	0x00,0x04, 0xXXXXXXXX	Client *Device Address* field (IrDA size)
	0x01,0x04, 0xYYYYYYYY	Client *Nonce* field (4 bytes)
	0x02,0x10, 0xZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ	*Session ID* field (16 bytes)
Server Response:		
response code	0xA0	**SUCCESS**, Final bit set
	0x002D	packet length of 45
	0x52	**Session-Parameters** HI
	0x002A	Length of **Session-Parameters** header (42 bytes)
	0x00, 0x04, 0xXXXXXXXX	Server *Device Address* field (IrDA size)
	0x01,0x04, 0xYYYYYYYY	Server *Nonce* field (4 bytes)
	0x02,0x10, 0xZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ	*Session ID* field (16 bytes)
	0x03, 0x01, 0x05	*Next Sequence Number* field (1-byte). As an example, the server expects 0x05
	0x06, 0x04, 0x00004000	*Offset* (4 bytes) into the interrupted PUT operation's object was 16K. All transmissions from the Client should start from an offset of 16K into the object data.

7.11 Resume Session for an interrupted GET operation using SRM

It is assumed that a Reliable Session exists, Single Response mode has already been setup using the Connect operation (see section 7.11), and an ongoing GET operation has been interrupted by a spurious transport disconnection.

Client Request:	**Bytes**	**Meaning**
Opcode	0x87	**SESSION**, Final bit set
	0x002D	packet length = 45
	0x52	**Session-Parameters** HI
	0x002A	Length of **Session-Parameters** header (42 bytes)
	0x05, 0x01,	Client *Session Opcode* field (1-byte)
	0x03	Resume Session opcode
	0x00,0x04, 0xXXXXXXXX	Client *Device Address* field (IrDA size)
	0x01,0x04, 0xYYYYYYYY	Client *Nonce* field (4 bytes)
	0x02,0x10, 0xZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ	*Session ID* field (16 bytes)
	0x06, 0x04, 0x00004000	*Offset* (4 bytes) into the interrupted GET operation's object was 16K. All transmissions from the Server should start from an offset of 16K into the object data.
Server Response:		
response code	0xA0	**SUCCESS**, Final bit set
	0x0027	packet length of 39
	0x52	**Session-Parameters** HI
	0x0024	Length of **Session-Parameters** header (36 bytes)
	0x00, 0x04, 0xXXXXXXXX	Server *Device Address* field (IrDA size)
	0x01,0x04, 0xYYYYYYYY	Server *Nonce* field (4 bytes)
	0x02,0x10, 0xZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ	*Session ID* field (16 bytes)
	0x03, 0x01, 0x05	*Next Sequence Number* field (1-byte). As an example, the server expects 0x05

7.12 Close a Session

Client Request:	**bytes**	**Meaning**
opcode	0x87	**SESSION**, Final bit set
	0x001B	packet length = 27
	0x52	**Session-Parameters** HI
	0x0018	Length of **Session-Parameters** header (24 bytes)
	0x05, 0x01,	Client *Session Opcode* field (1-byte)
	0x01	Close Session opcode
	0x02, 0x10, 0xZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ	*Session ID* field (16 bytes)
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x0003	packet length of 3

7.13 Negotiate a Timeout

Client Request:	**Bytes**	**Meaning**
opcode	0x87	**SESSION**, Final bit set
	0x000F	packet length = 15
	0x52	**Session-Parameters** HI
	0x000C	Length of **Session-Parameters** header (12 bytes)
	0x05, 0x01,	Client *Session Opcode* field
	0x04	Set Timeout opcode
	0x04, 0x04,	*Timeout* field
	0x000004B0	Requesting timeout of 20 minutes (1200 seconds)
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x000C	packet length of 12
	0x52	**Session-Parameters** HI
	0x0009	Length of **Session-Parameters** header (9 bytes)
	0x04, 0x04,	*Timeout* field
	0x00000384	Requesting timeout of 15 minutes (900 seconds)

7.14 SRM during a Single PUT Operation

Client Request:	**Bytes**	**Meaning**
Opcode	0x02	**PUT**, Final bit not set
	0x0424	1060 bytes is length of packet
	0x97	HI for Single Response Mode (SRM)
	0x01	Request to enable SRM mode
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	THING.DOC	name of object, null terminated
	0xC3	HI for **Length**
	0x00001800	Length of object is 6K 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:		
Response code	0x90	**CONTINUE, Final bit set**
	0x0005	Packet length of 5
	0x97	HI for Single Response Mode (SRM)
	0x01	Confirmation to enable SRM mode
Client Request:		
Opcode	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

A number of packets containing chunks of file body are sent, and finally we arrive at the last packet, which has the Final bit set. This signals to the Server that it will issue a response for the overall status of the **PUT** operation

Client Request:	**Bytes**	**Meaning**
Opcode	0x82	**PUT**, Final bit set
	0x0406	1030 bytes is length of packet

	0x49 0x0403 0x.....	HI for **End-of-Body** chunk Length of header (1K) plus HI and header length Last 1K bytes of body
Server Response:		
Response code	0xA0 0x0003	**SUCCESS**, Final bit sent Length of response packet

At this point Single Response Mode is automatically disabled as the **PUT** operation in which it was enabled is now complete. All future operations will resume use of the standard OBEX request/response behavior.

7.15 CONNECT for Single Response Mode (SRM)

7.15.1 Client Initiates Single Response Mode

Client Request:	**Bytes**	**Meaning**
Opcode	0x80	**CONNECT**, Final bit set
	0x0009	Packet length = 9
	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
	0x97	HI for **Single Response Mode (SRM)**
	0x01	Request to enable SRM mode
Server Response:		
Response code	0xA0	SUCCESS, Final bit set
	0x0009	Packet length of 9
	0x10	Version 1.0 of OBEX
	0x00	Flags, all zero for this version of OBEX
	0x0400	1K max packet size
	0x97	HI for **Single Response Mode (SRM)**
	0x01	Confirmation to enable SRM mode

7.15.2 Server Initiates Single Response Mode

Client Request:	**Bytes**	**Meaning**
Opcode	0x80	**CONNECT**, Final bit set
	0x0009	Packet length = 9
	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
	0x97	HI for **Single Response Mode (SRM)**
	0x02	Indicates support for SRM mode
Server Response:		
Response code	0xA0	SUCCESS, Final bit set
	0x0009	Packet length of 9
	0x10	Version 1.0 of OBEX
	0x00	Flags, all zero for this version of OBEX
	0x0400	1K max packet size
	0x97	HI for **Single Response Mode (SRM)**
	0x01	Request to enable SRM mode

Client Request:	**Bytes**	**Meaning**
Opcode	0x02	**PUT**, Final bit not set
	0x0424	1060 bytes is length of packet
	0x97	HI for **Single Response Mode (SRM)**
	0x01	Confirmation to enable SRM
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	THING.DOC	name of object, null terminated
	0xC3	HI for **Length**
	0x00001800	Length of object is 6K bytes
	0x48	HI for Object **Body** chunk
	0x0403	Length of **Body** header (1K) plus HI and header length
	0x.....	1K bytes of body

SRM is enabled, including the processing of this PUT operation

Client Request:		
Opcode	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

A number of packets containing chunks of file body are sent, and finally we arrive at the last packet, which has the Final bit set. This signals to the Server that it will issue a response for the overall status of the **PUT** operation

Client Request:		
Opcode	0x82	**PUT**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Last 1K bytes of body
Server Response:		
Response code	0xA0	**SUCCESS**, Final bit sent
	0x0003	Length of response packet

7.16 PUT during Single Response Mode (SRM)

It is assumed that Single Response Mode has already been setup using the Connect operation (see **section 7.15**).

Client Request:	**Bytes**	**Meaning**
Opcode	0x02	**PUT**, Final bit not set
	0x0422	1058 bytes is length of packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	THING.DOC	name of object, null terminated
	0xC3	HI for **Length**
	0x00001800	Length of object is 6K bytes
	0x48	HI for Object **Body** chunk
	0x0403	Length of **Body** header (1K) plus HI and header length
	0x.....	1K bytes of body

Client Request:		
Opcode	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

A number of packets containing chunks of file body are sent, and finally we arrive at the last packet, which has the Final bit set. This signals to the Server that it will issue a response for the overall status of the **PUT** operation

Client Request:		
Opcode	0x82	**PUT**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Last 1K bytes of body
Server Response:		
Response code	0xA0	**SUCCESS**, Final bit sent
	0x0003	Length of response packet

7.17 GET during Single Response Mode (SRM)

It is assumed that Single Response Mode has already been setup using the Connect operation (see **section 7.15**).

Client Request:	**Bytes**	**Meaning**
Opcode	0x83	**GET**, Final bit set
	0x001A	Length of **GET** packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	ITEMS.DOC	Name of object, null terminated
Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	1K bytes of body
Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Next 1K bytes of body

A number of packets containing chunks of file body are sent, and finally we arrive at the last packet, which has the Final bit set. This signals to the Client that the **GET** operation is complete.

Server Response:		
Response code	0xA0	**SUCCESS**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-Of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length

0x.....	Last 1K bytes of body

7.18 PUT and GET during Single Response Mode (SRM)

It is assumed that Single Response Mode has already been setup using the Connect operation (see **section 7.15**).

Client Request:	**Bytes**	**Meaning**
Opcode	0x02	**PUT**, Final bit not set
	0x0422	1058 bytes is length of packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	THING.DOC	name of object, null terminated
	0xC3	HI for **Length**
	0x00000C00	Length of object is 3K bytes
	0x48	HI for Object **Body** chunk
	0x0403	Length of **Body** header (1K) plus HI and header length
	0x.....	1K bytes of body
Client Request:		
Opcode	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
Client Request:		
Opcode	0x82	**PUT**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Last 1K bytes of body

Start **GET** operation without waiting for the Server to respond to the outstanding **PUT** operation. Since only one response occurs per operation, it is known that the first Server response packet will pertain to the **PUT** operation.

Client Request:	**Bytes**	**Meaning**
Opcode	0x83	**GET**, Final bit set
	0x001A	Length of **GET** packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	ITEMS.DOC	Name of object, null terminated

Server response to **PUT** operation

Server Response:	**Bytes**	**Meaning**
Opcode	0xA0	**SUCCESS**, Final bit set
	0x0003	Length of response packet

Now the Server response to the **GET** operation

Server Response:	**Bytes**	**Meaning**
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet

	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	1K bytes of body
Server Response:		
Response code	0xA0	**SUCCESS**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-Of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Last 1K bytes of body

7.19 Abort a PUT operation during Single Response Mode (SRM)

It is assumed that Single Response Mode has already been setup using the Connect operation (see **section 7.15**).

Client Request:	**Bytes**	**Meaning**
Opcode	0x02	**PUT**, Final bit not set
	0x0422	1058 bytes is length of packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	THING.DOC	name of object, null terminated
	0xC3	HI for **Length**
	0x00001800	Length of object is 6K bytes
	0x48	HI for Object **Body** chunk
	0x0403	Length of **Body** header (1K) plus HI and header length
	0x.....	1K bytes of body
Client Request:		
Opcode	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

Issue **ABORT** during the **PUT** operation.

Client Request:		
Opcode	0xFF	**ABORT**, Final bit set
	0x0003	Length of the request packet

Server responds to **ABORT** successfully. **PUT** operation is aborted and **ABORT** is complete.

Server Response:		
Response code	0xA0	**SUCCESS**, Final bit sent
	0x0003	Length of response packet

7.20 Abort a GET operation during Single Response Mode (SRM)

It is assumed that Single Response Mode has already been setup using the Connect operation (see **section 7.15**).

Case 1: Normal Abort case for GET operation

Client Request:	**Bytes**	**Meaning**
Opcode	0x83	**GET**, Final bit set
	0x001A	Length of **GET** packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	ITEMS.DOC	Name of object, null terminated
Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	1K bytes of body
Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Next 1K bytes of body

Issue **ABORT** during the **GET** operation.

Client Request:		
Opcode	0xFF	**ABORT**, Final bit set
	0x0003	Length of the request packet

Server responds to **ABORT** successfully. **GET** operation is aborted and **ABORT** is complete.

Server Response:		
Response code	0xA0	**SUCCESS**, Final bit sent
	0x0003	Length of response packet

Case 2: Exception case when aborting GET operation.

In this case the **ABORT** is issued, but the last **GET** Response is sent prior to the Server receiving and processing the **ABORT**. In this case the first Success response is known to be for the **GET** operation, since it contains the **End-Of-Body** header. In this case, we must expect a second response, as the **ABORT** operation must also signify completion, and since no operation was available to abort, it will complete with its own response code.

Client Request:	**bytes**	**Meaning**
Opcode	0x83	**GET**, Final bit set
	0x001A	Length of **GET** packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	ITEMS.DOC	Name of object, null terminated
Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length

	0x.....	1K bytes of body
Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Next 1K bytes of body

Issue **ABORT** during the **GET** operation.

Client Request:		
Opcode	0xFF	**ABORT**, Final bit set
	0x0003	Length of the request packet

Server responds to **GET** successfully. **GET** operation is completed prior to the Server processing the **ABORT**.

Server Response:		
Response code	0xA0	**SUCCESS**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-Of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Last 1K bytes of body

Server responds to **ABORT** successfully. No operation was aborted.

Server Response:		
Response code	0xA0	**SUCCESS**, Final bit sent
	0x0003	Length of response packet

7.21 Abort of PUT operation while GET outstanding during Single Response Mode (SRM)

It is assumed that Single Response Mode has already been setup using the Connect operation (see **section 7.15**).

Issue **GET** operation

Client Request:	**Bytes**	**Meaning**
Opcode	0x83	**GET**, Final bit set
	0x001A	Length of **GET** packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	THING.DOC	Name of object, null terminated

Issue **PUT** operation with **GET** operation outstanding

Client Request:	**Bytes**	**Meaning**
Opcode	0x02	**PUT**, Final bit not set
	0x0422	1058 bytes is length of packet
	0x01	HI for **Name**
	0x0017	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
Client Request:		
Opcode	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

Issue **ABORT** during the **PUT** operation.

NOTE: At this point we cannot expect to abort the **GET** operation as the subsequent **PUT** operation has already been started. In this case we expect the Server to process the operations in sequence, so the **PUT** will be aborted only after the **GET** operation completes and the portion of **PUT** packets have been processed.

Client Request:		
Opcode	0xFF	**ABORT**, Final bit set
	0x0003	Length of the request packet

Server responds to the **GET** operation. **GET** operation completes.

Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	1K bytes of body
Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Next 1K bytes of body
Server Response:		
Response code	0xA0	**SUCCESS**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for End-Of-Body chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Last 1K bytes of body

Server responds to **ABORT** successfully. **PUT** operation is aborted and **ABORT** is complete.

Server Response:		
Response code	0xA0	**SUCCESS**, Final bit sent
	0x0003	Length of response packet

7.22 PUT during Single Response Mode (SRM) using SRM Parameters Header

It is assumed that Single Response Mode has already been setup using the Connect operation (see **section 7.15**).

Client Request:	**Bytes**	**Meaning**
Opcode	0x02	**PUT**, Final bit not set
	0x0021	33 bytes is length of packet
	0x01	HI for **Name**
	0x0017	Length of **Name** header
	THING.DOC	name of object, null terminated
	0xC3	HI for **Length**
	0x00001800	Length of object is 6K bytes
	0x98	HI for **SRM Parameters**
	0x00	1-byte value (Client requests a Server Response)

The Client can pause and wait for the Server response (as shown here), or it may continue sending PUT requests, expecting the Server to respond as soon as possible. It is left up to the application to decide which method is used.

Server Response:		
Response code	0x90	**CONTINUE**, Final bit set
	0x0003	Length of response packet
Client Request:		
Opcode	0x02	**PUT**, Final bit not set
	0x0406	1030 bytes is length of packet
	0x48	HI for **Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Next 1K bytes of body

A number of packets containing chunks of file body are sent, and finally we arrive at the last packet, which has the Final bit set. This signals to the Server that it will issue a response for the overall status of the **PUT** operation

Client Request:		
Opcode	0x82	**PUT**, Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for **End-of-Body** chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.....	Last 1K bytes of body
Server Response:		
Response code	0xA0	**SUCCESS**, Final bit sent
	0x0003	Length of response packet

8. OBEX Services and Procedures

The following section covers the services currently defined for use over OBEX. This list in no way represents the entire set of services eligible for use over OBEX. It is intended to provide an understanding of what services currently use the OBEX protocol and how they use it. In addition, common processes or methods, such as the "Simple OBEX **PUT**" are also covered.

8.1 Folder Browsing Service

A folder/file system is a hierarchy of folders that contain objects/files and folders. A client can browse folders and **GET** the contents. The client can also traverse the folders and **PUT** and **GET** objects into and out of the folders. Life in a folder is more permanent than life in an inbox. It is very likely that after a client has **PUT** an object into a folder it can **GET** it at a later time.

The Folder Browsing service provides access to a device's object store via OBEX. The application that provides the service is called the *folder browsing server*, the client application is called the *folder browser*. The service is built using OBEX **PUT**, **GET** and **SETPATH** commands as well as the OBEX Folder-Listing object. It uses file access methods to **PUT** and **GET** Objects to the remote object store over a directed OBEX connection. To connect to a folder browsing service the client issues an OBEX **CONNECT** with a **Target** header containing the UUID (in binary) for the Folder Browsing service to the Default OBEX server. This UUID is F9EC7BC4-953C-11d2-984E-525400DC9E09.

The availability of the folder browsing service can be determined by inspecting the OBEX capability Object for the folder browsing service. Alternatively, a connection attempt to the folder browsing UUID can be performed. There should only be one instance of a folder browsing service on the default OBEX server. You could have folder browsing using a different OBEX server but it would not be a standard service.

8.1.1 Exchanging Folder Listings

Often, an application will request a listing of the remote devices folder system, in-order to locate an object it wants or to position an object for storage. When an application requests a listing of a folder's contents, it receives a folder-listing object. The specific format of this object is discussed in chapter 9.1 The Folder Listing Object. It should be noted that any OBEX headers sent with the folder-listing object refer to the object itself and not the folder for which the object is a listing of.

The folder-listing object is exchanged as octet-sequence data in one or more OBEX **Body** headers. The boundaries of the **Body** headers have no relevance to the internal structure of the folder listing. Individual **Body** headers should be concatenated to form the complete folder listing.

8.1.1.1 Requesting A Folder Listing

To retrieve a folder listing an OBEX **GET** request is sent to the folder-browsing server. Within the **GET** request, a **Name** header is used to convey the name of the folder for which a listing is desired. A **Type** header is used to indicate to the server the type of object requested. This is necessary because the client could be requesting a folder listing, or an object located within the folder. A connection must have already been established to the folder browsing service before requests can be sent.

The method used when requesting a folder is based on the relative position of the current folder. The following is an overview describing the four methods used when requesting a folder.

1. To retrieve the ***current*** folder: Send a **GET** Request with an empty **Name** header and a **Type** header that specifies the folder object type.
2. To retrieve a ***child*** folder: Send a **GET** Request with the **Name** of the child folder in a **Name** header and a **Type** header that specifies the folder object type.

3. To retrieve the ***parent*** folder: Send a **SETPATH** with the Backup flag set, to change to the parent folder. Then send a **GET** Request with an empty **Name** header and a **Type** header that specifies the folder object type.
4. To retrieve the ***root*** folder: Send a **SETPATH** with an empty **Name** header, to change to the root folder. Then send a **GET** Request with an empty **Name** header and a **Type** header that specifies the folder object type.

A recap of OBEX headers used when requesting a folder object:

- **Name**: *(Required)* The **Name** header specifies the name of the folder for which the listing is requested.
- **Type**: *(Required)* The **Type** header conveys the type of object requested. The value of this header must be "x-obex/folder-listing" when requesting a folder listing.
- **Connection ID**: (Required) The **Connection Id** of the folder browsing service.

8.1.1.2 Responding with a Folder Listing Object

A successful response to the **GET** operation should either indicate a **SUCCESS** or **CONTINUE** response code followed by one or more **Body** headers. The following headers may also be present in the response: **Name**, **Size** and **Time**. When present these headers describe the folder-listing object that is being retrieved.

- **Name**: *(Optional)* The **Name** header specifies the name of the folder for which the object is a listing. This should match the name present in the request, but is not required to.
- **Size**: *(Optional)* The **Size** header represents the total size of the folder-listing object in bytes.
- **Time**: *(Optional)* The **Time** header is used to convey the ISO **Time** that the folder-listing object was last updated.
- **Body / End-Of-Body** *(Required if successful)* The **Body** and **End-of-Body** headers are used to return the folder-listing object.

8.1.2 Navigating Folders

This section reviews the methods used for navigating a remote object store. The concepts are the same as those presented elsewhere in the OBEX specification. They are discussed here simply to make the discussion more complete.

8.1.2.1 Changing Folders

The OBEX **SETPATH** operation is used to change folders. A **Name** header is used for downward navigation and the Backup flag is used for upward navigation. To change to the root folder a **SETPATH** operation with an empty **Name** header is sent.

8.1.2.2 Creating Folders

The OBEX **SETPATH** operation is used to create folders in the same manner as it is used to traverse them. Implementations usually support a "create if non-existent" behavior for the **SETPATH** operation. In this way, entire trees of objects can be moved from one device to another.

8.1.3 Security

Anytime a device exposes its internal object store to a remote user, security is of some concern. This specification does not address any specific security measures because such behavior is highly device and operating system dependent. It is recommended that any application that provides access to its object store take necessary precautions to protect the privacy and consistency of the local object store.

8.2 Simple OBEX Put file transfer (+ SetPath)

This process transfers an object or a set of objects to the receiving device's inbox. The sending device has no understanding of what was done with the object after it was received. The **PUT** command is used with **Name** and optionally **Type** headers. An OBEX connection is required for compatibility with existing applications and is also used to increase transfer performance. This service supports file level access.

The QuickBeam and IrXfer applications are both based on this model of simple file transfer. These applications do not use directed OBEX connections, all objects go to the "inbox". The QuickBeam application provides a basic implementation of an inbox server, storing received objects on the file system in an inbox folder. The QuickBeam and IrXfer clients push generic file objects, using the headers and methods discussed in section 9.2 Generic File Object.

Applications exist which use the simple push file transfer model in conjunction with the **SETPATH** command to transfer an entire folder to the receiving device's inbox. This process is more complex than the simple push model because it implies that the sender has some knowledge of how the receiver will process or organize the objects. This creates difficulties for the receiving application and future applications should avoid sending the **SETPATH** command to the Inbox. On the receiving side, server applications should make an effort to permit the **SETPATH** operation to the extent possible.

Note that this process is not defined as a service because it is just a use of the inbox service. It is simply a description of existing implementations of an inbox server.

8.3 Telecom/IrMC Synchronization Service

This service allows a client to communicate with a server in a predefined manner so that the client can provide synchronization and backup services for the server. This service uses the OBEX **PUT** and **GET** commands as well as IMC vCard, vCalendar objects and other IrMC specifically defined objects. Refer to the IrMC specification for a complete object description. This service uses both File and Database access methods.

The IrMC server is targeted by appending the string, "telecom/", in a **Name** header before all object names. This service is organized by 4 levels of operation, with each successive level increasing in functionality. The levels 1, 2 & 3 do not use a dedicated OBEX connection and all requests are sent to the default OBEX server's inbox. Devices that support the IrMC service must handle this special case by redirecting all such requests to the IrMC service. Level 4 uses a directed connection to target the service (UUID, "IRMC-SYNC") via the default OBEX server.

Both simple **PUT** file transfer and IrMC sync use the default OBEX server's inbox service and they both use **PUT** with the **Name** header. They are able to distinguish the target because IrMC uses names with "telecom/" while simple **PUT** file transfer names do not contain paths.

8.4 OBEX Get Default Object

This process enables a device to request an object from a device's inbox. Typically, the objects available for request using this method will be few. Currently only business cards and the capability object have been identified for use with this method. The purpose of providing this capability is to enable a device to request basic information from a second device, instead of requiring that the second device initiate the exchange.

The procedure for requesting a default object is to send a **GET** command with a **Type** header specifying the MIME type of the desired object and an empty **Name** header. In this situation an empty **Name** header should be treated the same as no **Name** header. If a default object of the **Type** requested is available, it should be returned in one or more **Body/End-of-Body** headers in a successful response. If the object is not available, the response code NOT FOUND should be returned. If the request contained a value in the

Name header then the request is illegal (unless it is for the capability service) and FORBIDDEN should be returned.

8.4.1 Get default vCard example

In this example, the client connects to the default OBEX server and requests the devices default vCard. The device contains a default vCard and returns it successfully.

Client Request:	**bytes**	**Meaning**
opcode	0x80	**CONNECT**, Final bit set
	0x0007	packet length
	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
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x0007	packet length
	0x10	Version 1.0 of OBEX
	0x00	no flags
	0x0800	2K max packet size
Client Request:		
opcode	0x83	**GET**, Single packet request, so final bit set
	0x0018	packet length
	0x01	HI for **Name** header
	0x0003	length of empty **Name** header
	0x42	HI for **Type** header
	0x0010	Length of **Type** header
	text/x-vCard	Type of object
Server Response:		
response Code	0xA0	SUCCESS, Final bit set
	0x0210	length of response packet
	0x49	HI for **End-of-Body** header
	0x020D	522 byte vCard length
	0x……..	default vCard

8.5 Capability Service

The OBEX capability service is designed to provide a general-purpose method for accessing service information with OBEX. Traditionally, OBEX services and applications have derived private methods for handling the device and service information that is of interest to them. However a broader scope of services could interoperate if there were a more common way to access this information. For example, an IrMC device represents the vCard fields it can handle, in a file called “telecom/pb/info.log”. This is fine for IrMC devices because they see eye-to-eye, but a general-purpose application has no way of knowing how to retrieve this information. Additionally, different applications that want to support vCards may choose the same representation for acceptable vCard fields but will unlikely provide it in a file called “telecom/pb/info.log”. The capability service aims to bridge this gap by providing a simple access methods for retrieving commonly used information from a device.

8.5.1 The Capability Object/Database

The capability object is designed to provide a single place where general-purpose device, service and application information can be stored. Collecting this information in one place allows an application to easily retrieve a host of interesting device specific information. The capability object contains object information such as what objects the device supports and which ones can be **PUT** to the inbox. It contains general-purpose information such as the devices serial number and manufacturer. It also

contains a specification of service connection procedures for the various services available on a device. Such as, **CONNECT Target** Header values and Alternate LSAPs where the service is available.

The standardization of the capability object and its access methods allows a variety of services to retrieve common information in a common manner. New services will know how to advertise their object and service information and will not need to derive a private method. In addition, existing general purpose services will be able to locate and retrieve information about new services in the same manner that they retrieve existing service information. This will not only allow a more diverse set of OBEX applications to exchange objects but it will also allow more applications to be installed on a device because they'll use fewer resources.

There are three main sections in the Capability Object they are; General Information, Supported Objects and Service/Application-Info. The General section contains information about the device, which is likely to be of general interest. Information such as serial number and manufacturer are contained in this section. The Supported Objects section is separated into two sub-sections. The first section, the Inbox, lists the objects that are recognized by the device's inbox. This allows a connected device to determine if the recipient will accept the object it wishes to send before it initiates the transmission. The Supported Objects section provides information about other objects that are used in the device but are not permissible in the inbox. The Service Info section is designed for use by applications that need to convey static configuration information. Information such as application version and supported options is recorded here

8.5.2 The Object Profile Database

The object profile database works in conjunction with the capability object to provide information about individual objects. It is designed to allow a client to retrieve specific information about the level of support that is provided for a specific object. The server builds an *object profile* for each object that the device wants to support in the object database. An object profile contains the same information as the IrMC protocol provides in the info.log file, object fields section. Under IrMC, this vital information is hidden from most services because of its obscure location. By supplying this information in a service independent manner, the information is more useful.

Each *object profile* must have a specified format that is self-supporting. No knowledge of the supporting service can be required to understand the object profile. The method used by IrMC to describe vCards is valid under this criterion. In order to be useful to a variety of applications, the object profile must be generally available. This document is the repository for all object profiles. In addition, a method for defining the access method for each object profile must be available.

Object profiles are accessed in a very specific manner. By providing a common method, new objects can easily be added to the database. The capability service is responsible for providing the object profiles. Requests are made of the capability service, by sending the MIME type of the object for which the profile is requested. Since the capability service is designed to support many different types of objects, the type of object requested is also provided. In this case, it is an object profile object, which has the type "x-obex/object-profile".

8.5.3 Locating the Capability Service

Before a request for the capability object or an object profile can be made the application must connect to the capability service. Because the capability service is integral to the OBEX protocol it was designed to be located in the default OBEX server, along side the Inbox service. Therefore the capability service is connected to by sending a **CONNECT** packet with no targeting information. This is the same method used to connect to the inbox. The OBEX server can differentiate between requests sent to the capability service from that of the inbox because the Type header of requests to the Capability service will contain either "x-obex/capability" or "x-obex/object-profile". In addition, GET requests to the Inbox service are not allowed to contain any Name header.

Object profiles are a part of the capability service. Therefore, the same connection that is used to retrieve the capability object is used for retrieving object profiles. The server can differentiate between the requests by the **Type** header sent with the request.

Since IrMC also uses the default OBEX server with no targeting information, it is necessary to discuss how capability service transactions are differentiated from IrMC transactions. There are two significant differences that IrMC devices can use to differentiate IrMC from Capability Service requests. All Capability Service requests will contain a **Type** header with the value "x-obex/object-profile" or "x-obex/capability". Neither of these values are valid in IrMC. However, if the **Type** header is ignored by the server, the **Name** header also provides uniqueness. All **Name** headers used by IrMC start with "telecom/". Since "telecom/" is not a valid root MIME type, the Capability Service naming convention will not conflict with the IrMC namespace.

8.5.3.1 Requesting the Capability Object

The capability object is requested by sending a **GET** request with the MIME type of the capability object provided in an OBEX **Type** header. The MIME type of the capability object is "x-obex/capability". A successful response will contain one or more OBEX **Body** headers with the full capability object as their contents.

8.5.3.2 Requesting an Object Profile

The format for an object profile object request is based on the MIME type of the object for which the profile is requested. A **GET** request containing the MIME type of the object whose profile is desired in a **Name** header, with a **Type** header containing the value "x-obex/object-profile" is interpreted by the capability service as an object profile request. A successful response will contain one or more **Body** headers with the object profile object contents.

If a client does not know the MIME type of an object it can look-up the object in the capability object using the name extension. This record also contains the MIME type of the object, which can then be used to make an object profile object query. Below is an example **GET** sequence for the vCard object profile.

Client Request:		
Opcode	0x83	**GET**, Single packet request, so final bit set
	0x003B	packet length
	0x42	HI for **Type** header
	0x0019	length of **Type** header
	x-obex/object-profile	MIME type of an object profile object
	0x01	HI for **Name** header
	0x001F	length of **Name** header (Unicode)
	text/x-vCard	MIME type of the object for which a profile is requested
Server Response:		
response code	0xA0	SUCCESS, Final bit set
	0x012C	length of response packet
	0x49	HI for **End-of-Body** header
	0x0129	294 byte object length
	0x…..	vCard object profile

8.5.3.3 Sending an Object Profile (unilaterally)

In some application environments, it may be necessary for a client to inform the server of its object profile. This is the reverse of the service offered by the capability service. This can be the case when the client is requesting objects from the server and wishes for them to adhere to the object profile that it supports.

To fulfill this requirement, an object profile can be **PUT** to an application or service (NOT the capability service). The service receiving the object profile can ignore and discard the object if it does not support customized object responses. In this case, the **PUT** request should be failed by responding with a non-success OBEX response code. If the service supports customized object responses then it should process the object profile and apply its rules to any request for that objects type. The rules should be applied for the duration of the connection or until another object profile is received. To **PUT** an object profile the client sends a **PUT** request with the same headers as a **GET** request, a **Type** header of "x-obex/object-profile" and a **Name** header with the object profile's MIME type.

Client Request:		
Opcode	0x82	**PUT**, Single packet request, so final bit set
	0x0164	packet length
	0x01	HI for **Name** header
	0x001F	length of **Name** header
	text/x-vCard	name of object profile object (Unicode, null-terminated)
	0x42	HI for **Type** header
	0x0019	length of **Type** header
	x-obex/object-profile	MIME type of an object profile object
	0x49	HI for **End-of-Body** Header
	0x0129	294 byte object length
	0x……	vCard object profile

One instance where this is useful is when accessing the folder browsing service. In this service, the client **GET**s folder-listing objects from the server. In order to convey its desire to receive only limited information in the returned object the client sends a folder-listing object profile to the server before the first folder-listing request. In this manner, the server can avoid overwhelming the client and wasting bandwidth by eliminating information that the client is not interested in receiving.

9. OBEX Objects

This section contains definitions for objects, which have been created or adopted for use within the OBEX protocol and its applications. Whenever possible industry standard objects are used, such as vCards, etc. However occasionally the need arises to create an object specific to OBEX. Such objects are the topic of this chapter. The following is a list of the objects covered:

- The Folder Listing Object
- The Generic File Object
- The Capability Object
- The Object Profile Object

Often when creating a new OBEX object, the problem of assigning a MIME type for use with the OBEX **Type** header arises. It is recommended that the following approach be used to determine the MIME type for an object.

- If the object's domain is outside of OBEX it should be given an experimental MIME type under one of the existing MIME root types. A vCard is an example of such an object, it is rooted in the "text" type. These types must follow the rules for creating an experimental MIME type, specifically they must start with "x-". The MIME type for vCards is "text/x-vCard".
- If the object was created for use within OBEX than the type should be rooted in the OBEX experimental root type "x-obex". The capability and folder-listing objects, which have MIME types "x-obex/capability" and "x-obex/folder-listing", are examples.

9.1 The Folder Listing Object

A folder-listing object is a detailed itemization of the contents of a particular folder. Each object in a listing can contain a variable amount of information, conveyed as attributes of the object. A folder can be a folder or other similar type of container object. It is expressed as an application of the Extensible Markup Language (XML) specified by the W3C. Folder object data is exchanged in one or more OBEX **Body** headers. An example of a folder listing conforming to the OBEX folder object format is shown below.

```
<folder-listing>
    <folder name = "System" created = "19961103T141500Z"/>
    <file name = "Jumar.txt" created = "19971209T090300Z" size = "6672"/>
    <file name = "Obex.doc" created = "19970122T102300Z" size = "41042"/>
</folder-listing>
```

Folder-listing objects are provided by a Folder Browsing service. This service is designed to provide access to the folder/file system of the serving device. The folder system is represented as a hierarchy of folders that contain objects/files and sub-folders. A client can browse folders and **GET** their contents. The folder browsing service defines the methods used to traverse folders, as well as for storing and retrieving objects.

9.1.1 Element Specification

9.1.1.1 Document Definition

The syntax for describing OBEX folder objects is based on the XML specification. A folder description is written using the syntax of XML because it provides for flexible data tagging that fits the needs for an open, flexible and general-purpose way of describing objects. An OBEX folder listing is a set of various XML elements. Collectively these elements form an XML document, referred to as the folder listing.

A folder-listing document may contains zero or more of the following elements:

- **folder** – A description of a folder type object.

- **parent-folder** – Indicates the existence of a parent folder.
- **file** – A description of a file type object.

These elements may also contain attributes that describe the file or folder element.

9.1.1.2 Elements of a Folder Listing Document

Each object in a listing is described by an element with attributes. This section details the syntax and use of the defined elements. For more detailed information on the XML syntax and encoding rules, refer to the XML specification available at http://www.w3.org/TR/REC-xml.

9.1.1.2.1 Elements

Element Name	Meaning
file	Indicates the existence of a file contained within the folder.
folder	Indicates the existence of a folder contained within the folder.
parent-folder	Represents the existence of a parent to the folder being presented.
folder-listing	The document type.

9.1.1.2.2 Attributes used in File and Folder Elements

Attribute Name	Meaning
name	The name of the folder or file. It does not include any path or similar information. The value of this attribute is used in a **NAME** header to retrieve the object if desired. This value must be unique in all similar elements in the document. The current folder may be represented by a folder element with the name ".". For example: "<folder name=".">".
size	The size of the folder or file object in bytes. This size is an estimate and is not required to be exact. It is expressed as an unsigned base 10 integer.
modified	This attribute represents the last modified time for the object. It is expressed in the same format used by the OBEX ISO **Time** header. That format is YYYYMMDDTHHMMSS, where the capital letter 'T' is expressly inserted between the day and the hour fields. It is recommended that whenever possible UTC time be used. When expressing UTC time, the letter "Z" is appended to the end (for example: 19670110T153410Z).
created	This attribute represents the creation time for the object. It is expressed in the same format used by the OBEX ISO **Time** header. It is recommended that whenever possible UTC time be used.
accessed	This attribute represents the last accessed time for the object. It is expressed in the same format used by the OBEX ISO **Time** header. It is recommended that whenever possible UTC time be used.

user-perm **group-perm** **other-perm**	These attributes convey the access permissions for the object. The permissions are encoded based on the access currently available to this user over the object listed. The following alphabetic characters are used to describe access: **"R", "W", "D"** The value of the permissions type is an unordered sequence of these alphabetic characters. The permissions indicators are case independent. R: The READ permission applies to all object types. It indicates that an attempt to **GET** the named object should successfully retrieve its contents. D: The DELETE permission applies to file types. It indicates that the file may be removed by sending a PUT-DELETE command. W: The WRITE permission applies to all object types. It indicates that an attempt to modify the contents of the file by **PUT**'ing to the file should succeed. For folder objects it indicates that attempts to create a folder or other object within that folder should succeed. There are three levels of permissions. These are for systems that distinguish between access for the user, the group and other. The default permissions attribute is user. Note: A permission indicator does not imply that the appropriate command is guaranteed to work – just that it might. Other system specific limitations, such as limitations on available space for storing objects, may cause an operation to fail, where the permission flags may have indicated that it was likely to succeed. The permissions are a guide only. Some systems may have more specific permissions than those listed here, such systems should map those to the flags defined as best they are able.
owner	The Owner attribute is used to convey the user associated with ownership or responsibility for this object.
group	Some file systems have the notion of group ownership. This attribute is used to convey that information when present
type	This attribute works similarly to the OBEX **Type** header and expresses the MIME type of the file object. It can be used to interpret the files' internal format or an application association.
xml:lang	The XML defined language attribute may be used to specify the language of both content and attribute values. The default language of English (us-EN) need not be specified. This attribute affects the interpretation of the following attributes: **name**, **owner** and **group**. As well as any element content.

9.1.1.2.3 Attributes used with the Folder Listing Element

Attribute Name	Meaning
version	The version attribute is used in the folder-listing element to convey the version of the folder listing DTD that the document conforms to. The current version is 1.0.

9.1.1.2.4 File and Folder Element Content

Both file and folder elements may contain element content. When present, the content expresses the recommended display name of the file or folder. This differs from the name attribute in that it does not need to be unique or be a valid file/folder name. In the absence of this content, the name attribute should be used when displaying the element.

9.1.2 Folder Listing Details

9.1.2.1 The Folder Object Type

When requesting a folder listing, the **Type** header must contain the value "x-obex/folder-listing". By specifying this type, ambiguities about the type of document requested can be avoided. This value is not case sensitive. It is not necessary to send the **Type** header in the response.

9.1.2.2 Empty Folder Listing Objects

When a request for a folder listing results in an empty listing, the response should follow the behavior of any successful request. The folder-listing object contained in the response **Body** header should be a valid XML empty object. Here is an example of such an object:

```
<?xml version="1.0"?>
<!DOCTYPE folder-listing SYSTEM "obex-folder-listing.dtd">
<folder-listing version="1.0"/>
```

9.1.2.3 Customizing Folder Listings

A folder listing object profile can be used to provide information about the elements and attributes supported by the folder browsing server or browser. If available, the object profile should be registered with the capability service under the name "x-obex/folder-listing". The object profile is an empty XML folder object. It lists all the supported elements and attributes in their regular format but with no values. If a client browser wishes to inform the server of its object profile it should **PUT** the object profile to the server before it performs the first **GET** of a folder listing.

Adherence to a received object profile is not required. However, it is recommended that devices, which, by default provide detailed responses, support this feature. It is most useful in the case where a simple device is querying a more robust device and it doesn't want to get flooded with information. For example, if a PDA chooses to browse a desktop device it might want only a list of file and folder *names* and *sizes*. Below is an example **PUT** of an object profile for an application that only recognizes name and size attributes. Note that the syntax can be completely parsed by an XML processor.

Operation	Header	Content
PUT	**Name**	x-obex/folder-listing
	Type	x-obex/object-profile
	Body	<?xml version="1.0"?> <!DOCTYPE folder-listing SYSTEM "obex-folder-listing.dtd"> <folder-listing version="1.0"> <parent-folder /> <folder name="" size="" /> <file name="" size="" /> </folder-listing>

9.1.3 Encoding Folder Listing Objects

9.1.3.1 XML Basics

The definition of a folder-listing object is based on the W3C Specification of XML. XML is used because it provides the structure and syntax for the folder-listing object. For a folder-listing object to be correct, it must follow the syntax rules specified by the XML specification. In XML, the content and organization of a particular object type is expressed by its Document Type Definition (DTD). In order for folder-object to be correct, it must also adhere to the obex-folder-listing DTD. This DTD is provided in section 9.1.4.1 of this document.

XML allows the DTD to be either internal or external to the document that relies on it. The folder object DTD is always external and is never sent as part of the folder object. It is assumed that the receiving entity will be able to handle the object without an internal DTD.

The content of an XML document is contained in its elements and attributes. This document defines the elements and attributes used in folder-listing objects. The XML specification states that element and attribute names are case sensitive. Therefore, the element names “Parent-Folder”, “parent-folder” and “PARENT-FOLDER” all refer to different elements.

9.1.3.2 Character Encoding Format

An XML document allows for the specification of the character encoding used in the document. The encoding declaration is positioned in the very beginning of the document to enable quick determination. The default encoding for XML documents is UTF-8 and need not be specified. If an alternate encoding is used it must be specified in the encoding declaration.

It is recommended that folder-listing objects which contain Japanese characters be encoded using the SHIFT_JIS encoding. This encoding has the benefit of ensuring that the characters of ASCII have their normal values, which makes the processing of the encoding declaration straightforward. The following example illustrates an XML declaration for a SHIFT_JIS encoded document.

```
<?xml encoding=”SHIFT_JIS”?>
```

9.1.3.3 Folder-Listing Object Examples

The following listing is used to illustrate the encoding of a folder into a folder-listing object.

Name	Size	Creation Time (UTC)		Last Modified (UTC)		Type
	--					parent folder
System	--	Nov 3,1996	2:15p	Nov 3, 1996	2:15p	folder
IR Inbox	--	Mar 30, 1995	10:50a	Mar 30, 1995	10:50a	folder
Jumar.txt	6,672	Dec 9, 1997	9:03a	Dec 22, 1997	4:41p	text/plain
Obex.doc	41,042	Jan 22, 1997	10:23a	Jan 22, 1997	10:23a	application/msword

9.1.3.3.1 Detailed encoding of Example Data

```
<?xml version="1.0"?>
<!DOCTYPE folder-listing SYSTEM "obex-folder-listing.dtd">
<folder-listing version=”1.0”>
    <parent-folder />
    <folder name = “System” created=”19961103T141500Z”/>
    <folder name =” IR Inbox” created=”19950330T105000Z”/>
    <file name = “Jumar.txt” created=”19971209T090300Z” size=“6672”
        modified=“19971222T164100Z” user-perm=”RW”/>
    <file name =”Obex.doc” created=“19970122T102300Z” size = “41042”
        type=”application/msword” modified=“19970122T102300Z”/>
```

```
</folder-listing>
```

9.1.3.3.2 A Simpler encoding illustrating display names

```
<?xml version="1.0"?>
<!DOCTYPE folder-listing SYSTEM "obex-folder-listing.dtd">
<folder-listing version="1.0">
    <parent-folder />
    <folder name="System"/>
    <folder name="IR Inbox"/>
    <file name="Jumar.txt" size="6672">Jumar Handling Guide</file>
    <file name ="Obex.doc" type = "application/msword">OBEX Specification v1.0</file>
</folder-listing>
```

9.1.3.4 Simple Encoding Methods

In the simplest case, devices that want to provide a listing of the objects available on the device can construct a static representation of the folders. If the object list can dynamically change, the application providing the listing may need to be more adaptive. In general, the construction of an XML folder-object is simple and straightforward. The first three lines of the document are generally static text. The following encoding of the attributes for the various files and folders will generally be a consistent straightforward format.

9.1.3.5 Room for Future Extensions

The folder-listing element contains an attribute that specifies the version of the DTD that was used to construct the object. If the future necessitates changes to the folder-listing DTD, the version information can then be used to indicate what revision of the DTD the object was based on. Since it is intended that folder-listings be exchanged without an internal DTD, it is essential that different versions of the DTD be readily discernable.

9.1.4 XML Document Definition

9.1.4.1 DTD Specification

```
<!-- DTD for the OBEX Folder-Listing Object -->

<!ELEMENT folder-listing ( folder | file | parent-folder )* >
<!ATTLIST folder-listing version CDATA "1.0">

<!ELEMENT file (#PCDATA)>
<!ATTLIST file name CDATA #REQUIRED>
<!ATTLIST file size CDATA #IMPLIED>
<!ATTLIST file type CDATA #IMPLIED>
<!ATTLIST file modified CDATA #IMPLIED>
<!ATTLIST file created CDATA #IMPLIED>
<!ATTLIST file accessed CDATA #IMPLIED>
<!ATTLIST file user-perm NMTOKEN #IMPLIED>
<!ATTLIST file group-perm NMTOKEN #IMPLIED>
<!ATTLIST file other-perm NMTOKEN #IMPLIED>
<!ATTLIST file group CDATA #IMPLIED>
<!ATTLIST file owner CDATA #IMPLIED>
<!ATTLIST file xml:lang NMTOKEN #IMPLIED>

<!ELEMENT folder (#PCDATA) >
<!ATTLIST folder name CDATA #REQUIRED>
<!ATTLIST folder size CDATA #IMPLIED>
<!ATTLIST folder modified CDATA #IMPLIED>
<!ATTLIST folder created CDATA #IMPLIED>
<!ATTLIST folder accessed CDATA #IMPLIED>
```

```
<!ATTLIST folder user-perm NMTOKEN #IMPLIED>
<!ATTLIST folder group-perm NMTOKEN #IMPLIED>
<!ATTLIST folder other-perm NMTOKEN #IMPLIED>
<!ATTLIST folder group CDATA #IMPLIED>
<!ATTLIST folder owner CDATA #IMPLIED>
<!ATTLIST folder xml:lang NMTOKEN #IMPLIED>

<!ELEMENT parent-folder EMPTY>
```

9.2 Generic File Object

9.2.1 Introduction

The purpose of this chapter is to outline a method for the exchange of a basic file object [blob] from one device to another. While the OBEX protocol provides many useful headers and capabilities for exchanging objects. This section is designed to provide the reader with an overview of the basic structure used when exchanging generic file data. The file object is assumed to be sent by a generic client to the receiving device's inbox. In a manner consistent to the way QuickBeam and IrXfer applications work. Based on these assumptions, the file exchange should follow the guidelines shown here for headers used and response codes expected.

9.2.2 Commonly Used Headers

- **Name:** *(Required)* This header is used to convey the full name of the object to exchange. The **Name** header should not contain any path information. If it is necessary to specify a path, it should be done with the **SETPATH** command. All information in this header is interpreted as the objects name.
- **Length:** *(Recommended)* This header is used to convey the size of the file object in bytes. This can be used for verification of storage space requirements. The sum of the bytes passed in **Body** headers should be this many bytes (but is not guaranteed).
- **Body/End-of-Body:** *(Required)* **Body** headers are used to transfer the file data itself. They are sent repeatedly as long as file data exists to be exchanged. The **End-of-Body** header is used to indicate to the receiving application that this is the last piece of the file data. Applications frequently close the file and consider the transfer complete upon reception of this header. The data contained in the **Body/End-of-Body** headers should be concatenated to form the file.
- **Time:** *(Recommended)* This header is used to exchange the time that the file was last modified. Current file transfer applications use both the simple 4-byte **Time** header and the ISO format. It is recommended that all applications use the more robust ISO **Time** header. For backward compatibility it should be noted that the "IrXfer" application provided by Microsoft only accepts the 4-byte **Time** header. Attempts to send the ISO header will result in the rejection of the operation.
- **Type:** (O*ptional)* The MIME type of the file object. This is not often used but can be helpful in some cases to identify the application to associate with the file object.

9.2.3 Response Codes Commonly Used in File Exchange

The following is a list of the known OBEX Response codes that are used during File Object Exchange.

- CONTINUE
- SUCCESS
- BAD REQUEST
- UNAUTHORIZED
- NOT FOUND
- INTERNAL SERVER ERROR

9.2.4 Example Put Exchange

An application wishes to **PUT** a 2000 byte file named "**Test File Object**" with the time stamp 0x41a50016 (Jan 9, 1992, 11:02:00 UTC). The OBEX packet size is 512 bytes.

	Opcode or Response plus Headers	Final bit	Header Data	Header Length	Running Total of Data Exchanged
Request →	**PUT** **Name** **Length** **Time**		 "Test File Object" "2000" "19920109T110200Z"	 37 19	
Response ←	CONTINUE no headers	4			
Request →	**PUT** **Body**		"start of file data.."	509	506 bytes
Response ←	CONTINUE no headers	4			
Request →	**PUT** **Body**		"..continuation of file data.."	509	1012 bytes
Response ←	CONTINUE no headers	4			
Request →	**PUT** **Body**		"..continuation of file data.."	509	1518 bytes
Response ←	CONTINUE no headers	4			
Request →	**PUT** **Body**		"..final segment of file data"	485	2000 bytes
Response ←	CONTINUE no headers	4			
Request →	**PUT** **End-of-Body**	4	Empty end of body header.	3	2000 bytes
Response ←	SUCCESS no headers	4			

9.3 The Capability Object

The syntax for the capability object is based on XML. XML was chosen because of its flexibility and ease with which elements can be categorized and described. The DTD of the Capability Object can be found in section 9.3.6 Capability Object DTD and an example Capability Object is shown in the section 9.3.7 Capability Object Example. The Capability Object is flexible enough that individual manufacturers can include objects, services and extension elements that are of interest to them. The extension element (see 9.3.2.10 Extension (Ext)) permits the addition of user-defined parameters for specific application or manufacturer needs.

9.3.1 The Capability Container (Capability)

This element is the root element containing all other elements of the Capability Object. It has one attribute called **Version**. When the XML document conforms to this revision of the Capability Object specification, the Version attribute MUST have the value "1.0".

The Capability container contains a General Information Container (see 9.3.2 General Information Container (General)), an optional Inbox Container (see 9.3.3 Inbox Container (Inbox)) and a list of services and applications (see 9.3.4 Service/Application List), which are supported by the device.

9.3.2 General Information Container (General)

The General Information Container is used to hold information that is specific to the device, which is hosting the Capability Object. This information may be used by a variety of services and applications and is therefore collected in the general section to make it easily available. The following is a description of the items that may be present in the general section.

9.3.2.1 Manufacturer (Manufacturer)

The manufacturer element is used to identify the vendor that builds the hardware device. This information can be handy when identifying the device to the user. It is a string value that shows the full name of the manufacturer. This value is **required**.

9.3.2.2 Model Name (Model)

The model name element is used to identify the model name and/or number assigned to the device by the manufacturer. Again, the information is handy when identifying the device to the user. This string value conveys the full model name of the device. This value is **required**.

9.3.2.3 Serial Number (SN)

The serial number is used to uniquely identify the device. This number should be assigned by the manufacturer in a globally unique method. The serial number does not have to be a UUID. If the manufacturer and model number are combined with the serial number you can get a universal unique ID. The format of the serial number is specified by the manufacturer, but it must be unique within a specific device model. This value is **optional**.

9.3.2.4 Original Equipment Manufacturer (OEM)

The OEM is used to specify the Original Equipment Manufacturer of the device. This value is **optional**.

9.3.2.5 Software Version (SW)

The software version specifies the software version of the device. The **Version** attribute specifies the version identifier and the **Date** attribute specifies the date of the SW version of the device. The date MUST be formatted as a complete representation, in the basic format of a [DATES] date or date and UTC time of day, for example, 19980114 or 19990714T133000Z. Only hours, minutes and seconds MUST be specified in the time component. Either **Version** or **Date** attribute should be present. This value is **optional**.

9.3.2.6 Firmware Version (FW)

Specifies the firmware version of the device. See 9.3.2.5 Software Version (SW) for more details. This value is **optional**.

9.3.2.7 Hardware Version (HW)

Specifies the hardware version of the device. See 9.3.2.5 Software Version (SW) for more details. This value is **optional**.

9.3.2.8 Currently Selected Language (Language)

Specifies the currently used language of the device.. The language MUST be represented by the two-letter lower-case language symbols defined by ISO [LANG], for example English would be represented as "en". This value is **optional**.

9.3.2.9 Memory Record (Memory)

Describes the available memory types and sizes. Also may contain device specific memory limitations like maximum file name length. This element is **optional**.

MemType	Describes the memory type. Values are manufacturer specific. This value is **optional**.
Location	Describes the logical memory location (for example, drive "c:\" or path name). This value is **optional**.
Free	Defines the free user memory in bytes. This value is **optional**.
Used	Defines the used user memory in bytes. This value is **optional**.
Shared	Boolean value indicating whether the user memory is shared between applications. A missing element indicates the memory is not shared. This value is **optional**.
FileSize	Specifies the maximum accepted file size in bytes. This value is **optional**.
FolderSize	Specifies the maximum accepted folder size in bytes. This value is **optional**.
FileNLen	Specifies the maximum accepted file name length in characters. If the maximum folder name length element (FolderNLen) is not present the maximum file name length indicates the combined length (file name and path together). This value is **optional**.
FolderNLen	Specifies the maximum accepted folder name length in characters. This value is **optional**.
CaseSenN	Specifies whether the device supports case insensitivity or case sensitivity of file and folder names. A missing element indicates case insensitive names. This value is **optional**.
Ext	See 9.3.2.10 Extension (Ext). This value is **optional**.

9.3.2.10 Extension (Ext)

Specifies a non-standard, experimental or manufacturer specific extension supported by the device. The extension is specified in terms of the XML element type name and a value. Extension elements are **optional**.

XNam	Specifies the name of an extension element. This value is **required**.
XVal	Specifies the value of an extension element. This value is **optional**.

9.3.3 Inbox Container (Inbox)

The Inbox container contains all the types of objects that the device will accept in its inbox. An object type is identified by its MIME type or name extension listed in the Inbox object record. This list of objects can

always include the object of type "ANY" to indicate that any object can be **PUT** to the Inbox. The Inbox can contain additional extension elements if needed, and is an **optional** element.

9.3.3.1 Object Record (Object)

An object record indicates the support for a particular object type. The Inbox container may contain zero or more objects. Each Inbox object must contain at least a **Type** or **Name-Ext** value. It is highly recommended that both values be present when available. The **Ext** element can be used to create additional attributes to the object. This element may contain one or more of the following attributes.

- **Type -** The MIME type of the object. This value is **conditionally required**.

- **Name-Ext -** The generally accepted filename extension used for this object format. This value is **conditionally required**. An object can contain zero or more Name-Ext elements.

- **Size -** The maximum accepted object size in bytes. This value is **optional**.

- **Ext -** Generic extension containing a name-value pair (see 9.3.2.10 Extension (Ext)). An object can contain zero or more Ext elements. This value is **optional.**

9.3.4 Service/Application List

The services and applications supported by the device are listed in the capability object. The service/application specific information of each service/application is presented inside of a service record. The service record contains all the service objects supported by the service/application.
Additionally, special information regarding application access methods is placed here. Applications are not required to have representation in the service/application list. The service/application list may contain zero or more service records.

9.3.4.1 Service/Application Record (Service)

One service record exists for each service that wishes to provide service specific information in the capability object. The service records can be identified by the **Name** of the service or its **UUID** but at least one of them has to be present. The information contained in each service record is governed by the service itself. This allows for the specification of service specific information. The following information may be present in any Service record.

- **Name -** The name of the service/application. This value is **conditionally required**.

- **UUID -** The UUID of the service/application. This value is **conditionally required**.

- **Version -** The version number of the service/application. This value is **optional**.

- **Object -** The format for a service object is the same as that of an inbox object in chapter 9.3.3.1 Object Record (Object). A service record may contain zero or more service objects. This value is **optional**.

- **Access -** The access method for the service. The attributes for this element are shown below. If not present, it can be assumed that the service is accessible via the standard targeted connection process used in OBEX. This value is **optional**.

Protocol	The transport protocol used to communicate with this service. Possible values are "TCP" and "IrDA". This value is **required**.
Endpoint	Contains the endpoint to which the client can connect to communicate with this service. In IrDA, the endpoint is synonymous with a TinyTP LSAP-SEL. In

	TCP, it is a TCP Port number. This value is **required**.
Target	The **Target** header value used to establish a connection with this service. This value is **required**.
Ext	See (see 9.3.2.10 Extension (Ext)). This value is **optional.**

• **Ext** - Generic extension containing a name-value pair (see 9.3.2.10 Extension (Ext)). This value is **optional**.

9.3.5 Requesting the Capability Object

The capability object is requested by sending a **GET** request with the MIME type of the capability object provided in an OBEX **Type** header. The MIME type of the capability object is "x-obex/capability". A successful response will contain one or more OBEX **Body** headers with the full capability object as their contents.

9.3.6 Capability Object DTD

```
<!ELEMENT Capability (General, Inbox?, Service*)>
        <!ATTLIST Capability Version CDATA "1.0">
<!ELEMENT General (Manufacturer, Model, SN?, OEM?, SW?, FW?, HW?, Language?, Memory*, Ext*)>
<!ELEMENT Manufacturer (#PCDATA)>
<!ELEMENT Model (#PCDATA)>
<!ELEMENT SN (#PCDATA)>
<!ELEMENT OEM (#PCDATA)>
<!ELEMENT SW EMPTY>
<!ATTLIST SW
        Version CDATA #IMPLIED
        Date CDATA #IMPLIED>
<!ELEMENT FW EMPTY>
<!ATTLIST FW
        Version CDATA #IMPLIED
        Date CDATA #IMPLIED>
<!ELEMENT HW EMPTY>
<!ATTLIST HW
        Version CDATA #IMPLIED
        Date CDATA #IMPLIED>
<!ELEMENT Language (#PCDATA)>
<!ELEMENT Memory (MemType?, Location?, Free?, Used?, Shared?, FileSize?, FolderSize?,
FileNLen?, FolderNLen?, CaseSenN?, Ext*)>
<!ELEMENT MemType (#PCDATA)>
<!ELEMENT Location (#PCDATA)>
<!ELEMENT Free (#PCDATA)>
<!ELEMENT Used (#PCDATA)>
<!ELEMENT Shared EMPTY>
<!ELEMENT FileSize (#PCDATA)>
<!ELEMENT FolderSize (#PCDATA)>
<!ELEMENT FileNLen (#PCDATA)>
<!ELEMENT FolderNLen (#PCDATA)>
<!ELEMENT CaseSenN EMPTY>
<!ELEMENT Ext (XNam, XVal*)>
<!ELEMENT XNam (#PCDATA)>
<!ELEMENT XVal (#PCDATA)>
<!ELEMENT Object (((Type, Name-Ext*)|(Type?, Name-Ext+)), Size?, Ext*)>
<!ELEMENT Inbox (Object*, Ext*)>
```

```
<!ELEMENT Type (#PCDATA)>
<!ELEMENT Name-Ext (#PCDATA)>
<!ELEMENT Size (#PCDATA)>
<!ELEMENT Service (((Name?, UUID)|(Name, UUID?)), Version?, Object*, Access*, Ext*)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT UUID (#PCDATA)>
<!ELEMENT Version (#PCDATA)>
<!ELEMENT Access (Protocol? ,Endpoint?, Target?, Ext*)>
<!ELEMENT Protocol (#PCDATA)>
<!ELEMENT Endpoint (#PCDATA)>
<!ELEMENT Target (#PCDATA)>
```

9.3.7 Capability Object Example

```
<?xml version="1.0" ?>
<!-- OBEX Capability Object -->

<!DOCTYPE Capability SYSTEM "obex-capability.dtd">

<Capability Version="1.0">

        <!-- General Purpose information -->
        <General>

                <Manufacturer>Big Factory, Ltd.</Manufacturer>
                <Model>Mighty 4119</Model>
                <SN>1234567890</SN>
                <OEM>Jane´s Phones</OEM>
                <SW Version="2.0"/>
                <FW Version="2.0e" Date="19971031T231210"/>
        </General>

        <!-- Inbox Object Definitions -->
        <Inbox>
                <Object>
                        <Type>text/x-vMsg</Type>
                        <Name-Ext>vmg</Name-Ext>
            </Object>
                <Object>
                        <Type>text/x-vCard</Type>
                        <Name-Ext>vcf</Name-Ext>
            </Object>
                <Object>
                        <Type>image/jpeg</Type>
                        <Name-Ext>jpg</Name-Ext>
            </Object>
        </Inbox>

        <!-- Service Object Definitions -->
        <Service>
                <Name>IrMC</Name>
                <UUID>IRMC-SYNC</UUID>
                <Object>
                        <Type>x-irmc/info.log</Type>
                        <Name-Ext>log</Name-Ext>
                </Object>
```

```
                <Access>
                        <Protocol>IrDA</Protocol>
                        <Target></Target>
                </Access>
            <Ext>
                <XNam>PhoneBook/Support</XNam>
                <XVal>4</XVal>
            </Ext>
            <Ext>
                <XNam>PhoneBook/Optional</XNam>
                <XVal>7</XVal>
            </Ext>
                <Ext>
                <XNam>PhoneBook/Version</XNam>
                <XVal>1.0</XVal>
            </Ext>
                <Ext>
                <XNam>Messaging/Support</XNam>
                <XVal>4</XVal>
            </Ext>
                <Ext>
                <XNam>Messaging/Version</XNam>
                <XVal>1.0</XVal>
            </Ext>
        </Service>
    <Service>
                <Name>Folder-Browsing</Name>
                <UUID>F9EC7BC4-953C-11d2-984E-525400DC9E09</UUID>
                <Object>
                        <Type>x-obex/folder-listing</Type>
                </Object>
                <Access>
                        <Protocol>IrDA</Protocol>
                        <Target>F9EC7BC4-953C-11d2-984E-525400DC9E09</Target>
                </Access>
    </Service>
    <Service>
                <Name>Image-X</Name>
                <UUID>F9EC7BC7-953C-11d2-984E-525400DC9E09</UUID>
                <Access>
                        <Protocol>IrDA</Protocol>
                        <Endpoint>7</Endpoint>
                        <Target>F9EC7BC7-953C-11d2-984E-525400DC9E09</Target>
                </Access>
            <Access>
                        <Protocol>TCP</Protocol>
                        <Target>F9EC7BC7-953C-11d2-984E-525400DC9E09</Target>
            </Access>
        </Service>
</Capability>
```

9.4 The Object Profile Object

This section gives an overview on how to create object profile objects for your objects. Object profiles do not have to be in XML or any other common format. Each object profile developer is free to choose the

most applicable format for describing the object. When complete, this format must be described in section 9 OBEX Objects.

9.4.1 Creating an Object Profile

Although all object profile objects have the same MIME type, they are not necessarily in the same format. Each object profile is defined in a format that best represents the object. For example, the vCard object profile syntax is very similar to the vCard syntax. This makes it easier for vCard oriented parsers to process the vCard object profile. The same holds true for the folder-listing object profile. This object profile is represented in XML format to match the folder-listing object format.

Whenever feasible the object profile syntax chosen should mirror the format and expressive capabilities of the object itself. In addition, a self-describing object format such as the format used by vCards and folder-objects is preferred. Whatever format is chosen, the author must add the specification of this format to this document. The MIME type of the object must also be specified to avoid possible ambiguity. The end of this document contains a section entitled Object Profiles. This is where all the object profiles available in the capability database are described.

9.4.2 Object Profiles

This section holds the definitions for each object profile supported by the capability object. Each definition must contain the MIME type and the name extension of the object. As well as, the format used by the profile object. The definitions are sorted alphabetically by object name.

9.4.2.1 Folder-Listing

MIME type	x-obex/folder-listing
Name extension	*None*
Profile format	XML, Refer to Folder-Listing Object Proposal.

9.4.2.2 vCalendar

MIME Type	text/x-vcalendar
Name extension	vcs
Profile format	TBD, Refer to the IrMC specification of X-IRMC-FIELDS definition.

9.4.2.3 vCard

MIME type	text/x-vcard
Name extension	vcf
Profile format	TBD, Refer to the IrMC specification of X-IRMC-FIELDS definition.

9.4.2.4 vMessage

MIME type	text/x-vmsg
Name extension	vmg
Profile format	TBD, Refer to the IrMC specification of X-IRMC-FIELDS definition.

9.4.2.5 UPF

MIME type	image/x-UPF
Name extension	upf
Profile format	TBD

9.4.3 Object Profile Example

This is an example of a vCard object profile. The vCard object profile is derived from the work of the IrMC group. This definition is similar to the X-IRMC-FIELDS extension property.

```
Begin: vCard-Profile
Version:
N:=20
UID:=4
ADR[1=20;2;6;7]
TEL;TYPE=HOME;WORK;
End: vCard-Profile
```

10. Session Capabilities in OBEX

A number of applications and services built on OBEX™ desire the ability to recover an OBEX session that has been broken due to loss of the underlying transport (e.g. IrLAP link loss). Examples include restarting a file transfer and resuming a sync session. The Infrared Financial Messaging (IrFM) group also has determined the need to maintain an OBEX session when the IrLAP link is lost. This section contains a proposal for providing reliable session support in OBEX. This proposal is aimed at meeting the needs of IrFM but should provide enough capabilities to meet the needs of other services and applications. A list of desired features for an OBEX session layer is given below.

- Allow an OBEX session to be resumed without loss of data when the underlying transport layer is lost.
- Allow OBEX applications and services to suspend a session and later resume it.
- Provide indications to applications and services about loss of the underlying transport connection.
- Provide the ability to automatically resume the session including restarting the lower level transports without burdening the applications or services.
- Allow existing services such as the Inbox, File transfer, IrMC, etc. to have session support.
- Allow multiple directed connections to be started and stopped within a single session.

A lower priority feature is listed below.

- Ability to start a session on one transport and move it to another transport (e.g. start the OBEX session on IrDA and move it to Bluetooth).

10.1 OBEX Session Overview

The OBEX specification divides OBEX into two parts: a model for representing objects and a session protocol. The session protocol uses a binary packet-based client/server request-response model. The session protocol defines the basic structure of an OBEX conversation and includes a set of commands to perform specific actions such as **PUT** and **GET**. The OBEX conversation occurs within the context of an OBEX connection. The OBEX session protocol runs on top of a transport layer such as TinyTP in IrDA, RFCOMM in Bluetooth or TCP. Currently, when the underlying transport is disconnected the OBEX session is lost. This section describes a solution, which allows OBEX to maintain the session when the underlying transport is lost. Before describing this solution it is necessary to review the current OBEX session protocol.

When the OBEX client creates an OBEX Transport Connection to the OBEX Server, an implicit OBEX session is created. A connection-oriented session can be created by sending an OBEX **CONNECT** command. An implicit OBEX session exists because it is possible to send directed commands to a service without creating a directed connection to it first. Connection-oriented sessions created using the OBEX **CONNECT** command are also called OBEX Connections. In the case of the Default OBEX Server, sending an OBEX **CONNECT** command without a **Target** header creates a connection to the Inbox service. This connection is called the Inbox connection and there are a number of services, which utilize this connection. A Directed Connection is made by sending an OBEX **CONNECT** command with a **Target** header containing the ID of the desired service or application.

The Default OBEX Server resides at a well-known location. The method for finding the Default OBEX Server is dependent on the underlying transport being used. OBEX applications can use Directed Connections on the Default OBEX Server or they can use there own OBEX server which runs over a different OBEX transport connection from the one used by the Default OBEX Server.

Currently, the implicit OBEX session created when an OBEX Client initiates an OBEX Transport connection is only reliable as long as the OBEX Transport connection exists. When the underlying transport connection is disconnected the OBEX session fails. It is desired to have reliable OBEX

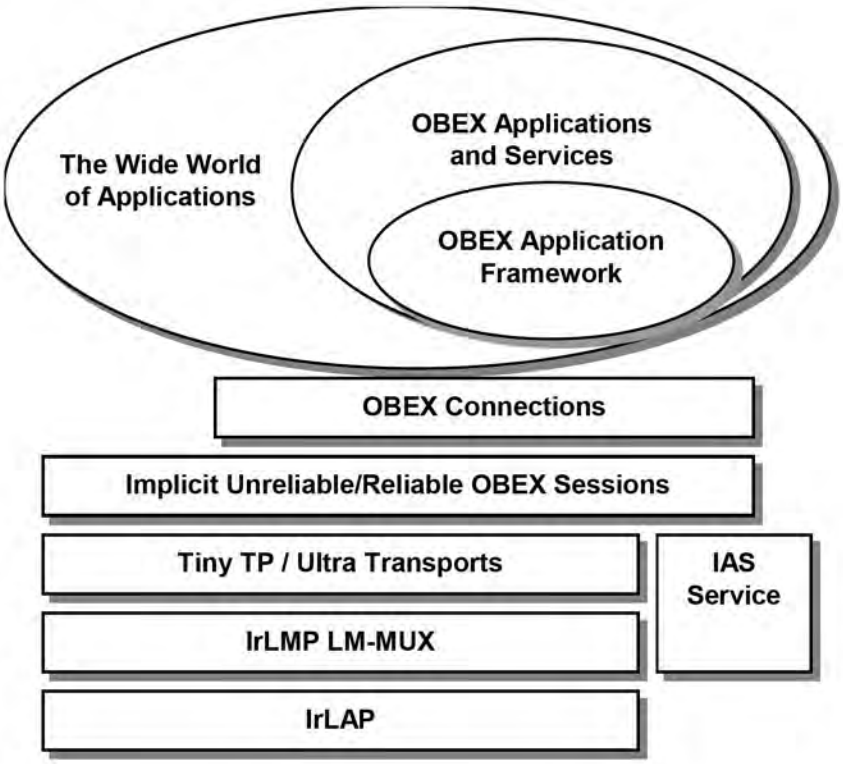
sessions, which can be resumed when the underlying transport connection is broken. Given this context (broken OBEX Transport Connection), the implicit OBEX session is considered unreliable.

10.2 Reliable OBEX Session Overview

When the OBEX Client initiates the underlying transport connection to the OBEX Server it creates an unreliable implicit OBEX session. Any connection-oriented sessions made on top of this implicit session are also unreliable. This document proposes that reliable OBEX connections can be built by creating a reliable underlying session. The OBEX client can create a reliable session by sending a new OBEX command called **CREATESESSION**. In order to be able to reestablish the reliable session when the underlying transport breaks it is necessary for the session to have a Session ID that is known by both the OBEX Client and Server. This Session ID must be unique enough such that both the Client and the Server know they are resuming the same session. It is also necessary that the Client and Server know enough information about each other so that the underlying transport can be reestablished. In the case of IrDA this would be the 32-bit device address. Until a reliable session is established the unreliable implicit session is active. The list below outlines the concepts of a reliable OBEX Session.

- New OBEX commands are created for managing reliable sessions. These are **CREATESESSION**, **CLOSESESSION**, **SUSPENDSESSION, RESUMESESSION** and **SETTIMEOUT**.
- New OBEX headers are created to be used with the new commands and features. These are **Session-Parameters** and **Session-Sequence-Number**.
- When a reliable session is established a session ID is created that is known by both the Client and the Server. This ID must be unique enough such that the reliable session can be reestablished. When the underlying transport is broken under a reliable session, the session is considered suspended.
- The session context must be saved in order to resume a session when the underlying transport is broken.
- Applications/services must also maintain context in order to resume a session.
- Only one session can be active at a time per OBEX transport connection. When the OBEX transport connection is first established the unreliable implicit session is active. Multiple sessions can exist between devices but only one can be active at a time. The other sessions are considered suspended including the unreliable session. If the underlying transport is broken the unreliable session is broken as well.
- Any number of connection-oriented sessions (OBEX connections) can be made within a reliable session.
- The OBEX Capability Object can be accessed via any underlying session including the unreliable session.
- To facilitate reliability, the session layer must ensure reliable transfer of OBEX packets. To achieve this a header containing sequence numbers is added to each OBEX packet. If an OBEX packet is lost it will be retransmitted. This will occur only when a session is resumed since OBEX packets can only be lost when the underlying transport is lost (OBEX requires a reliable transport).
- Since OBEX packets are large, retransmission should only be used if the packet was actually lost. It is possible that the server received a command but the client did not receive the response. A method exists to allow the server to indicate the last packet received during session resume. The client can then send an empty packet (command without headers) to facilitate retransmission of the response.

The picture below shows how a reliable sessions fit into the architecture.


The Wide World of Applications
OBEX Applications and Services
OBEX Application Framework
OBEX Connections
Implicit Unreliable/Reliable OBEX Sessions
Tiny TP / Ultra Transports
IrLMP LM-MUX
IrLAP
IAS Service

10.3 Session Context

In order to resume a suspended session a context must be saved by both the client and the server. This context includes both session and OBEX information. The context contains the following items.

Item	Description
Session ID	This is the session ID created for this session. The session ID is generated from the information sent by both the client and server when the session was created so it does not actually have to be saved since it can be re-calculated.
Client Device Address	The device address sent by the client in the **CREATESESSION** command.
Client Nonce	The nonce sent by the client in the **CREATESESSION** command.
Server Device Address	The device address sent by the server in its response to the **CREATESESSION** command.
Server Nonce	The nonce sent by the server in its response to the **CREATESESSION** command.
Session Sequence Number	For the client this is the sequence number of the last packet sent. For the server this is the next sequence number expected.
Last OBEX packet	If the session was suspended unexpectedly because of loss of the underlying transport the last packet sent must be saved in case it needs to be transmitted. Both the client and the server must save the last packet sent. Also if the OBEX implementation buffers packets for the application then these must also be saved. It is possible to create an API between the OBEX layer and application such that the OBEX layer does not actually need to save the data. The API can allow the OBEX layer to ask the application to provide the data to be retransmitted.
Target Strings and Connection Ids	For all directed connections, which were created within the session, the target string and the corresponding Connection ID must be saved. Along with this is the need to save enough information to map the target back to the corresponding application. Also if the inbox connection has been created then this knowledge must also be saved.
OBEX Packet sizes	For each OBEX connection the maximum OBEX packet size must be saved.

Applications and services must also save context in order to resume a session. The OBEX layer does not know how commands such as PUT, GET, SETPATH, etc are used by the application/service so it cannot save context for them.

10.4 Persistence

If the OBEX layer and/or applications are shut down while a session is suspended then the context must be saved in a persistent data store if resuming the session later is desired.

10.5 Multiple Transport Support Proposal

This section describes a proposal for supporting sessions over different transports. This is not part of the errata but really just a method for capturing the idea. The actual method will be proposed as errata at a later date.

A possible method for supporting a session over different transports utilizes the OBEX Capability Object. The OBEX capability object must be accessible from any session via a connection to the Inbox service or the service that is accessed via the non-targeted OBEX connection. OBEX implementations that support sessions running over multiple transports will list the transport information in the Capability Object. This

includes the name of the transport and the device address used for that transport (e.g. IrDA is the 32-bit device address, Bluetooth is the 48-bit address, IP is the IP address, etc).

The proposal is to add a new section to the Capability Object listing the transports in which OBEX currently runs over. These are transports that allow an OBEX reliable session to be resumed independent of the transport the session was originally created or suspended. The section contains a list of transports where each transport contains the following attributes:

Protocols	The pertinent list of protocols on which OBEX resides. Items in the list are separated by commas. The first item in the list should be the main physical/link layer in the protocol stack (e.g. IrDA, Bluetooth, USB, Ethernet, Serial, etc). Typically this is the only protocol required. Other protocols are listed if needed.
Addresses	The list of addresses or port numbers needed to access the OBEX layer. Items in the list are separated by commas. There should be an item in the **Address List** corresponding to each item in the **Protocol List**. The last address in **the Address List** is the value used as the *Device Address* field in Session commands.

Both attributes are required. Below is an example Transports section with two entries.

```
<!—Transports Section -->
<Transports>
    <Transport Protocols ="IrDA" Addresses="442356AE" />
    <Transport Protocols="Bluetooth" Addresses="00F349125E01" />
</Transports>
```

The method for moving a session to a different transport is to first suspend the session then resume the session over the new transport.

11. OBEX Testing Requirements

11.1 Symbols and Conventions

M	Mandatory support. Refers to capabilities that must be used in the OBEX test specification.
O	Optional support. Refers to capabilities that can be used in the OBEX test specification, but are not required.
X	Excluded. Refers to capabilities that may be supported by the device but shall not be exercised in this test specification.

11.2 OBEX Roles

OBEX Roles	Role
OBEX Client	O
OBEX Server	O

11.3 IrDA-Specific Features

IrDA-Specific Features	OBEX Client	OBEX Server
TinyTP connection	M	M
IAS Query procedure	M	M
IrLMP connection	M	M
IrLAP connection	M	M
Physical Layer	M	M
Physical Layer short-range option	O	O

11.4 OBEX Features

OBEX Features	OBEX Client	OBEX Server
Put operation	M	M
Get operation	O	O
SetPath operation	O	O
Connect operation	O	M
Disconnect operation	O	M
Session operation	O	O
Abort operation	O	O
Authentication	O	O
Directed Connections	O	M
Header support	M	M
User Defined operations	X	X

11.5 Required Behavior

Some behaviors are required of any device with an implementation of the OBEX protocol that wishes to be certified as an OBEX compliant device. The required behaviors are the subset of behaviors that are necessary to promote interoperability between various OBEX applications and devices.

Most of the required behaviors are intended to insure that any device that wishes to exchange an object in a protocol-compliant manner can do so. Therefore, some of the requirements affect only server behavior. Support for the **PUT** operation is required.

12. Appendices

12.1 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 have been illustrated throughout this document. The only protocol 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 OBEX server application would support **CONNECT** and either **PUT** or **GET**.

12.2 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.

12.3 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 Offer operation - enables receiving side to preview and approve proposed objects
4. Add Negotiate operation - in-band negotiation
5. Add content negotiation - all sides to propose multiple options and have other side select
6. Add encryption
7. Add compression hooks

12.4 Known Target Identifiers

This section summarizes the UUID's known at the time when this document was last updated.

- **IrMC Service**
  - Unique identifier value "IRMC-SYNC".
  - Encoded as 9 bytes of ASCII text in a **Target** header.
- **Folder Browsing Service**
  - Unique Identifier value "F9EC7BC4-953C-11d2-984E-525400DC9E09".
  - Encoded as 16 bytes of hexadecimal in a **Target** header.
- SyncML Service
  - Unique identifier value "SYNCML-SYNC".
  - Encoded as 11 bytes of ASCII text in a **Target** header.

12.5 MD5 Algorithm for Authentication

This appendix contains code for a copyright and royalty free implementation of the MD5 algorithm. The code was copied from http://www.pbm.com/dice/rnd.txt.

```
/*
 * This code implements the MD5 message-digest algorithm.
 * The algorithm is due to Ron Rivest.  This code was
 * written by Colin Plumb in 1993, no copyright is claimed.
 * This code is in the public domain; do with it what you wish.
 *
 * Equivalent code is available from RSA Data Security, Inc.
 * This code has been tested against that, and is equivalent,
 * except that you don't need to include two pages of legalese
 * with every copy.
 *
 * To compute the message digest of a chunk of bytes, declare an
 * MD5Context structure, pass it to MD5Init, call MD5Update as
 * needed on buffers full of bytes, and then call MD5Final, which
 * will fill a supplied 16-byte array with the digest.
 */

typedef unsigned long word32;
typedef unsigned char byte;

struct xMD5Context {
        word32 buf[4];
        word32 bytes[2];
        word32 in[16];
};

void xMD5Init(struct xMD5Context *context);
void xMD5Update(struct xMD5Context *context, byte const *buf, int len);
void xMD5Final(byte digest[16], struct xMD5Context *context);
void xMD5Transform(word32 buf[4], word32 const in[16]);


/*
 * Shuffle the bytes into little-endian order within words, as per the
 * MD5 spec.  Note: this code works regardless of the byte order.
 */
void
byteSwap(word32 *buf, unsigned words)
{
        byte *p = (byte *)buf;

        do {
                *buf++ = (word32)((unsigned)p[3] << 8 | p[2]) << 16 |
                        ((unsigned)p[1] << 8 | p[0]);
                p += 4;
        } while (--words);
}

/*
 * Start MD5 accumulation.  Set bit count to 0 and buffer to mysterious
 * initialization constants.
 */
void
xMD5Init(struct xMD5Context *ctx)
{
        ctx->buf[0] = 0x67452301;
```

```
        ctx->buf[1] = 0xefcdab89;
        ctx->buf[2] = 0x98badcfe;
        ctx->buf[3] = 0x10325476;

        ctx->bytes[0] = 0;
        ctx->bytes[1] = 0;
}

/*
 * Update context to reflect the concatenation of another buffer full
 * of bytes.
 */
void
xMD5Update(struct xMD5Context *ctx, byte const *buf, int len)
{
        word32 t;

        /* Update byte count */

        t = ctx->bytes[0];
        if ((ctx->bytes[0] = t + len) < t)
                ctx->bytes[1]++;        /* Carry from low to high */

        t = 64 - (t & 0x3f); /* Space avail in ctx->in (at least 1) */
        if ((unsigned)t > len) {
                bcopy(buf, (byte *)ctx->in + 64 - (unsigned)t, len);
                return;
        }
        /* First chunk is an odd size */
        bcopy(buf,(byte *)ctx->in + 64 - (unsigned)t, (unsigned)t);
        byteSwap(ctx->in, 16);
        xMD5Transform(ctx->buf, ctx->in);
        buf += (unsigned)t;
        len -= (unsigned)t;

        /* Process data in 64-byte chunks */
        while (len >= 64) {
                bcopy(buf, ctx->in, 64);
                byteSwap(ctx->in, 16);
                xMD5Transform(ctx->buf, ctx->in);
                buf += 64;
                len -= 64;
        }

        /* Handle any remaining bytes of data. */
        bcopy(buf, ctx->in, len);
}

/*
 * Final wrapup - pad to 64-byte boundary with the bit pattern
 * 1 0* (64-bit count of bits processed, MSB-first)
 */
void
xMD5Final(byte digest[16], struct xMD5Context *ctx)
{
        int count = (int)(ctx->bytes[0] & 0x3f); /* Bytes in ctx->in */
        byte *p = (byte *)ctx->in + count;      /* First unused byte */
```

```
        /* Set the first char of padding to 0x80. There is always room.*/
        *p++ = 0x80;

        /* Bytes of padding needed to make 56 bytes (-8..55) */
        count = 56 - 1 - count;

        if (count < 0) {        /* Padding forces an extra block */
                bzero(p, count+8);
                byteSwap(ctx->in, 16);
                xMD5Transform(ctx->buf, ctx->in);
                p = (byte *)ctx->in;
                count = 56;
        }
        bzero(p, count+8);
        byteSwap(ctx->in, 14);

        /* Append length in bits and transform */
        ctx->in[14] = ctx->bytes[0] << 3;
        ctx->in[15] = ctx->bytes[1] << 3 | ctx->bytes[0] >> 29;
        xMD5Transform(ctx->buf, ctx->in);

        byteSwap(ctx->buf, 4);
        bcopy(ctx->buf, digest, 16);
        bzero(ctx,sizeof(ctx));
}


/* The four core functions - F1 is optimized somewhat */

/* #define F1(x, y, z) (x & y | ~x & z) */
#define F1(x, y, z) (z ^ (x & (y ^ z)))
#define F2(x, y, z) F1(z, x, y)
#define F3(x, y, z) (x ^ y ^ z)
#define F4(x, y, z) (y ^ (x | ~z))

/* This is the central step in the MD5 algorithm. */
#define MD5STEP(f,w,x,y,z,in,s) \
         (w += f(x,y,z) + in, w = (w<<s | w>>(32-s)) + x)

/*
 * The core of the MD5 algorithm, this alters an existing MD5 hash to
 * reflect the addition of 16 longwords of new data.  MD5Update blocks
 * the data and converts bytes into longwords for this routine.
 */
void
xMD5Transform(word32 buf[4], word32 const in[16])
{
        register word32 a, b, c, d;

        a = buf[0];
        b = buf[1];
        c = buf[2];
        d = buf[3];

        MD5STEP(F1, a, b, c, d, in[0] + 0xd76aa478, 7);
        MD5STEP(F1, d, a, b, c, in[1] + 0xe8c7b756, 12);
```

```
MD5STEP(F1, c, d, a, b, in[2] + 0x242070db, 17);
MD5STEP(F1, b, c, d, a, in[3] + 0xc1bdceee, 22);
MD5STEP(F1, a, b, c, d, in[4] + 0xf57c0faf, 7);
MD5STEP(F1, d, a, b, c, in[5] + 0x4787c62a, 12);
MD5STEP(F1, c, d, a, b, in[6] + 0xa8304613, 17);
MD5STEP(F1, b, c, d, a, in[7] + 0xfd469501, 22);
MD5STEP(F1, a, b, c, d, in[8] + 0x698098d8, 7);
MD5STEP(F1, d, a, b, c, in[9] + 0x8b44f7af, 12);
MD5STEP(F1, c, d, a, b, in[10] + 0xffff5bb1, 17);
MD5STEP(F1, b, c, d, a, in[11] + 0x895cd7be, 22);
MD5STEP(F1, a, b, c, d, in[12] + 0x6b901122, 7);
MD5STEP(F1, d, a, b, c, in[13] + 0xfd987193, 12);
MD5STEP(F1, c, d, a, b, in[14] + 0xa679438e, 17);
MD5STEP(F1, b, c, d, a, in[15] + 0x49b40821, 22);

MD5STEP(F2, a, b, c, d, in[1] + 0xf61e2562, 5);
MD5STEP(F2, d, a, b, c, in[6] + 0xc040b340, 9);
MD5STEP(F2, c, d, a, b, in[11] + 0x265e5a51, 14);
MD5STEP(F2, b, c, d, a, in[0] + 0xe9b6c7aa, 20);
MD5STEP(F2, a, b, c, d, in[5] + 0xd62f105d, 5);
MD5STEP(F2, d, a, b, c, in[10] + 0x02441453, 9);
MD5STEP(F2, c, d, a, b, in[15] + 0xd8a1e681, 14);
MD5STEP(F2, b, c, d, a, in[4] + 0xe7d3fbc8, 20);
MD5STEP(F2, a, b, c, d, in[9] + 0x21e1cde6, 5);
MD5STEP(F2, d, a, b, c, in[14] + 0xc33707d6, 9);
MD5STEP(F2, c, d, a, b, in[3] + 0xf4d50d87, 14);
MD5STEP(F2, b, c, d, a, in[8] + 0x455a14ed, 20);
MD5STEP(F2, a, b, c, d, in[13] + 0xa9e3e905, 5);
MD5STEP(F2, d, a, b, c, in[2] + 0xfcefa3f8, 9);
MD5STEP(F2, c, d, a, b, in[7] + 0x676f02d9, 14);
MD5STEP(F2, b, c, d, a, in[12] + 0x8d2a4c8a, 20);

MD5STEP(F3, a, b, c, d, in[5] + 0xfffa3942, 4);
MD5STEP(F3, d, a, b, c, in[8] + 0x8771f681, 11);
MD5STEP(F3, c, d, a, b, in[11] + 0x6d9d6122, 16);
MD5STEP(F3, b, c, d, a, in[14] + 0xfde5380c, 23);
MD5STEP(F3, a, b, c, d, in[1] + 0xa4beea44, 4);
MD5STEP(F3, d, a, b, c, in[4] + 0x4bdecfa9, 11);
MD5STEP(F3, c, d, a, b, in[7] + 0xf6bb4b60, 16);
MD5STEP(F3, b, c, d, a, in[10] + 0xbebfbc70, 23);
MD5STEP(F3, a, b, c, d, in[13] + 0x289b7ec6, 4);
MD5STEP(F3, d, a, b, c, in[0] + 0xeaa127fa, 11);
MD5STEP(F3, c, d, a, b, in[3] + 0xd4ef3085, 16);
MD5STEP(F3, b, c, d, a, in[6] + 0x04881d05, 23);
MD5STEP(F3, a, b, c, d, in[9] + 0xd9d4d039, 4);
MD5STEP(F3, d, a, b, c, in[12] + 0xe6db99e5, 11);
MD5STEP(F3, c, d, a, b, in[15] + 0x1fa27cf8, 16);
MD5STEP(F3, b, c, d, a, in[2] + 0xc4ac5665, 23);

MD5STEP(F4, a, b, c, d, in[0] + 0xf4292244, 6);
MD5STEP(F4, d, a, b, c, in[7] + 0x432aff97, 10);
MD5STEP(F4, c, d, a, b, in[14] + 0xab9423a7, 15);
MD5STEP(F4, b, c, d, a, in[5] + 0xfc93a039, 21);
MD5STEP(F4, a, b, c, d, in[12] + 0x655b59c3, 6);
MD5STEP(F4, d, a, b, c, in[3] + 0x8f0ccc92, 10);
MD5STEP(F4, c, d, a, b, in[10] + 0xffeff47d, 15);
MD5STEP(F4, b, c, d, a, in[1] + 0x85845dd1, 21);
```

```
        MD5STEP(F4, a, b, c, d, in[8] + 0x6fa87e4f, 6);
        MD5STEP(F4, d, a, b, c, in[15] + 0xfe2ce6e0, 10);
        MD5STEP(F4, c, d, a, b, in[6] + 0xa3014314, 15);
        MD5STEP(F4, b, c, d, a, in[13] + 0x4e0811a1, 21);
        MD5STEP(F4, a, b, c, d, in[4] + 0xf7537e82, 6);
        MD5STEP(F4, d, a, b, c, in[11] + 0xbd3af235, 10);
        MD5STEP(F4, c, d, a, b, in[2] + 0x2ad7d2bb, 15);
        MD5STEP(F4, b, c, d, a, in[9] + 0xeb86d391, 21);

        buf[0] += a;
        buf[1] += b;
        buf[2] += c;
        buf[3] += d;
}


void MD5(void *dest, void *orig, int len)
{
        struct xMD5Context context;

        xMD5Init(&context);
        xMD5Update(&context, orig, len);
        xMD5Final(dest, &context);
}
```