D-Bus Specification

Havoc Pennington

Red Hat, Inc.

<hp@pobox.com>

Anders Carlsson

CodeFactory AB

<andersca@codefactory.se>

Alexander Larsson

Red Hat, Inc.

<alexl@redhat.com>

Sven Herzberg

Imendio AB

<sven@imendio.com>

Simon McVittie

Collabora Ltd.

<smcv@collabora.com>

David Zeuthen

<zeuthen@gmail.com>

Version 0.38

Revision History

Revision latest

See commit log

Revision 0.38 2022-02-23

- Add ActivatableServicesChanged signal and feature flag
- * is optionally-escaped in addresses

Revision 0.37 2021-12-17

- Update recommendations for interoperable DBUS_COOKIE_SHA1 timeouts
- Clarify padding requirements for arrays and variants
- Describe where the interoperable machine ID comes from
- Clarify use of dictionary (array of dict-entry) types

Revision 0.36 2020-04-21

- Fix a typo in an annotated hexdump of part of a message

Revision 0.35 2019-05-13

- Add UnixGroupIDs to GetConnectionCredentials
- Avoid redundancy in defining interface name syntax

Revision 0.34 2018-12-04 pwithnall

- Correct ObjectManager example AddMatch rule

Revision 0.33 2018-04-27 smcv

- Deprecate TCP on Unix
- Deprecate non-local TCP everywhere

Revision 0.32	2018-01-30	smcv

- Deprecate hyphen/minus in bus names, with underscore as the recommended replacement
- Document the convention for escaping leading digits in interface and bus names (org._7_zip)
- Recommend using SASL EXTERNAL where possible, or DBUS_COOKIE_SHA1 otherwise
- Message buses should not accept SASL ANONYMOUS
- Document the meaning of non-empty SASL authorization identity strings
- Document the optional argument to SASL ERROR
- Document who sends each SASL command, and the possible replies
- Document the authentication states used to negotiate Unix fd-passing
- Servers that relay messages should remove header fields they do not understand
- Clarify who controls each header field
- Document the HeaderFiltering message bus feature flag
- Non-message-bus servers may use the SENDER and DESTINATION fields

Revision 0.31	2017-06-29	smcv, TG

- Don't require implementation-specific search paths to be lowest priority
- Correct regex syntax for optionally-escaped bytes in addresses so it includes hyphen-minus, forward slash and underscore as intended
- Describe all message bus methods in the same section
- Clarify the correct object path for method calls to the message bus
- Document that the message bus implements Introspectable, Peer and Properties
- Add new Features and Interfaces properties for message bus feature-discovery
- Add unix:dir=..., which resembles unix:tmpdir=... but never uses abstract sockets
- Don't require eavesdrop='true' to be accepted from connections not sufficiently privileged to use it successfully
- Formally deprecate eavesdropping in favour of BecomeMonitor

Revision 0.30	2016-11-28	smcv, PW

Define the jargon terms service activation and auto-starting more clearly. Document the SystemdService key in service files. Document how AppArmor interacts with service activation, and the new AssumedAppArmorLabel key in service files (dbus-daemon 1.11.8). Clarify intended behaviour of Properties.GetAll. Use versioned interface and bus names in most examples.

Revision 0.29	2016-10-10	PW

Introspection arguments may contain annotations; recommend against using the object path '/'

Revision 0.28	2016-08-15	PW

Clarify serialization

Revision 0.27	2015-12-02	LU

Services should not send unwanted replies

Revision 0.26	2015-02-19	smcv, rh

GetConnectionCredentials can return LinuxSecurityLabel or WindowsSID; add privileged BecomeMonitor method

Revision 0.25	2014-11-10	smcv, lennart

ALLOW_INTERACTIVE_AUTHORIZATION flag, EmitsChangedSignal=const

Revision 0.24	2014-10-01	SMcV

non-method-calls never expect a reply even without NO_REPLY_EXPECTED; document how to quote match rules

Revision 0.23	2014-01-06	SMcV, CY

method call messages with no INTERFACE may be considered an error; document tcp:bind=... and nonce-tcp:bind=...; define listenable and connectable addresses

Revision 0.22	2013-10-09	

add GetConnectionCredentials, document GetAtdAuditSessionData, document GetConnectionSELinuxSecurityContext, document and correct .service file syntax and naming

Revision 0.21	2013-04-25	smcv

allow Unicode noncharacters in UTF-8 (Unicode Corrigendum #9)

Revision 0.20	22 February 2013	smcv, walters

reorganise for clarity, remove false claims about basic types, mention /o/fd/DBus

Revision 0.19	20 February 2012	smcv/lp

formally define unique connection names and well-known bus names; document best practices for interface, bus, member and error names, and object paths; document the search path for session and system services on Unix; document the systemd transport

Revision 0.18	29 July 2011	smcv

define eavesdropping, unicast, broadcast; add eavesdrop match keyword; promote type system to a top-level section

Revision 0.17	1 June 2011	smcv/davidz

define ObjectManager; reserve extra pseudo-type-codes used by GVariant

Revision 0.16	11 April 2011	

add path_namespace, arg0namespace; argNpath matches object paths

Revision 0.15	3 November 2010	
Revision 0.14	12 May 2010	
Revision 0.13	23 Dezember 2009	
Revision 0.12	7 November, 2006	
Revision 0.11	6 February 2005	
Revision 0.10	28 January 2005	
Revision 0.9	7 Januar 2005	
Revision 0.8	06 September 2003	

First released document.

Table of Contents

Introduction

D-Bus is a system for low-overhead, easy to use interprocess communication (IPC). In more detail:

- D-Bus is *low-overhead* because it uses a binary protocol, and does not have to convert to and from a text format such as XML. Because D-Bus is intended for potentially high-resolution same-machine IPC, not primarily for Internet IPC, this is an interesting optimization. D-Bus is also designed to avoid round trips and allow asynchronous operation, much like the X protocol.

- D-Bus is *easy to use* because it works in terms of *messages* rather than byte streams, and automatically handles a lot of the hard IPC issues. Also, the D-Bus library is designed to be wrapped in a way that lets developers use their framework's existing object/type system, rather than learning a new one specifically for IPC.

The base D-Bus protocol is a one-to-one (peer-to-peer or client-server) protocol, specified in the section called “Message Protocol”. That is, it is a system for one application to talk to a single other application. However, the primary intended application of the protocol is the D-Bus *message bus*, specified in the section called “Message Bus Specification”. The message bus is a special application that accepts connections from multiple other applications, and forwards messages among them.

Uses of D-Bus include notification of system changes (notification of when a camera is plugged in to a computer, or a new version of some software has been installed), or desktop interoperability, for example a file monitoring service or a configuration service.

D-Bus is designed for two specific use cases:

- A "system bus" for notifications from the system to user sessions, and to allow the system to request input from user sessions.

- A "session bus" used to implement desktop environments such as GNOME and KDE.

D-Bus is not intended to be a generic IPC system for any possible application, and intentionally omits many features found in other IPC systems for this reason.

At the same time, the bus daemons offer a number of features not found in other IPC systems, such as single-owner "bus names" (similar to X selections), on-demand startup of services, and security policies. In many ways, these features are the primary motivation for developing D-Bus; other systems would have sufficed if IPC were the only goal.

D-Bus may turn out to be useful in unanticipated applications, but future versions of this spec and the reference implementation probably will not incorporate features that interfere with the core use cases.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. However, the document could use a serious audit to be sure it makes sense to do so. Also, they are not capitalized.

Protocol and Specification Stability

The D-Bus protocol is frozen (only compatible extensions are allowed) as of November 8, 2006. However, this specification could still use a fair bit of work to make interoperable reimplementation possible without reference to the D-Bus reference implementation. Thus, this specification is not marked 1.0. To mark it 1.0, we'd like to see someone invest significant effort in clarifying the specification language, and growing the specification to cover more aspects of the reference implementation's behavior.

Until this work is complete, any attempt to reimplement D-Bus will probably require looking at the reference implementation and/or asking questions on the D-Bus mailing list about intended behavior. Questions on the list are very welcome.

Nonetheless, this document should be a useful starting point and is to our knowledge accurate, though incomplete.

Type System

D-Bus has a type system, in which values of various types can be serialized into a sequence of bytes referred to as the *wire format* in a standard way. Converting a value from some other representation into the wire format is called *marshaling* and converting it back from the wire format is *unmarshaling*.

The D-Bus protocol does not include type tags in the marshaled data; a block of marshaled values must have a known *type signature*. The type signature is made up of zero or more *single complete types*, each made up of one or more *type codes*.

A type code is an ASCII character representing the type of a value. Because ASCII characters are used, the type signature will always form a valid ASCII string. A simple string compare determines whether two type signatures are equivalent.

A single complete type is a sequence of type codes that fully describes one type: either a basic type, or a single fully-described container type. A single complete type is a basic type code, a variant type code, an array with its element type, or a struct with its fields (all of which are defined below). So the following signatures are not single complete types:

```
"aa"

"(ii"

"ii)"
```

And the following signatures contain multiple complete types:

```
"ii"

"aiai"

"(ii)(ii)"
```

Note however that a single complete type may *contain* multiple other single complete types, by containing a struct or dict entry.

Basic types

The simplest type codes are the *basic types*, which are the types whose structure is entirely defined by their 1-character type code. Basic types consist of fixed types and string-like types.

The *fixed types* are basic types whose values have a fixed length, namely BYTE, BOOLEAN, DOUBLE, UNIX_FD, and signed or unsigned integers of length 16, 32 or 64 bits.

As a simple example, the type code for 32-bit integer (`INT32`) is the ASCII character 'i'. So the signature for a block of values containing a single `INT32` would be:

```
"i"
```

A block of values containing two `INT32` would have this signature:

```
"ii"
```

Use this types to generate a message "signature", or description of what is inside. Ex: this **signature** is a struct of {int32_t, int32_t, uint32_t, int64_t, double}. Parenthesis indicate "struct" (see next pg): `"(iiuxd)"  // struct signature`

The characteristics of the fixed types are listed in this table.

Conventional name	ASCII type-code	Encoding
`BYTE`	`y` (121)	Unsigned 8-bit integer
`BOOLEAN`	`b` (98)	Boolean value: 0 is false, 1 is true, any other value allowed by the marshalling format is invalid
`INT16`	`n` (110)	Signed (two's complement) 16-bit integer
`UINT16`	`q` (113)	Unsigned 16-bit integer
`INT32`	`i` (105)	Signed (two's complement) 32-bit integer
`UINT32`	`u` (117)	Unsigned 32-bit integer
`INT64`	`x` (120)	Signed (two's complement) 64-bit integer (mnemonic: x and t are the first characters in "sixty" not already used for something more common)
`UINT64`	`t` (116)	Unsigned 64-bit integer
`DOUBLE`	`d` (100)	IEEE 754 double-precision floating point
`UNIX_FD`	`h` (104)	Unsigned 32-bit integer representing an index into an out-of-band array of file descriptors, transferred via some platform-specific mechanism (mnemonic: h for handle)

The *string-like types* are basic types with a variable length. The value of any string-like type is conceptually 0 or more Unicode codepoints encoded in UTF-8, none of which may be U+0000. The UTF-8 text must be validated strictly: in particular, it must not contain overlong sequences or codepoints above U+10FFFF.

Since D-Bus Specification version 0.21, in accordance with Unicode Corrigendum #9, the "noncharacters" U+FDD0..U+FDEF, U+nFFFE and U+nFFFF are allowed in UTF-8 strings (but note that older versions of D-Bus rejected these noncharacters).

The marshalling formats for the string-like types all end with a single zero (NUL) byte, but that byte is not considered to be part of the text.

The characteristics of the string-like types are listed in this table.

Conventional name	ASCII type-code	Validity constraints
`STRING`	`s` (115)	No extra constraints
`OBJECT_PATH`	`o` (111)	Must be a syntactically valid object path
`SIGNATURE`	`g` (103)	Zero or more *single complete types*

Valid Object Paths

An object path is a name used to refer to an object instance. Conceptually, each participant in a D-Bus message exchange may have any number of object instances (think of C++ or Java objects) and each such instance will have a path. Like a filesystem, the object instances in an application form a hierarchical tree.

Object paths are often namespaced by starting with a reversed domain name and containing an interface version number, in the same way as interface names and well-known bus names. This makes it possible to implement more than one service, or more than one version of a service, in the same process, even if the services share a connection but cannot otherwise co-operate (for instance, if they are implemented by different plugins).

Using an object path of `/` is allowed, but recommended against, as it makes versioning of interfaces hard. Any signals emitted from a D-Bus object have the service's unique bus name associated with them, rather than its well-known name. This means that receipients of the signals must rely entirely on the signal name and object path to work out which interface the signal originated from.

For instance, if the owner of `example.com` is developing a D-Bus API for a music player, they might use the hierarchy of object paths that start with `/com/example/MusicPlayer1` for its objects.

The following rules define a valid object path. Implementations must not send or accept messages with invalid object paths.

- The path may be of any length.
- The path must begin with an ASCII '/' (integer 47) character, and must consist of elements separated by slash characters.
- Each element must only contain the ASCII characters "[A-Z][a-z][0-9]_"
- No element may be the empty string.
- Multiple '/' characters cannot occur in sequence.
- A trailing '/' character is not allowed unless the path is the root path (a single '/' character).

Valid Signatures

Curly braces in a signature indicate a "dict" (key:value hash table dictionary map)

An implementation must not send or accept invalid signatures. Valid signatures will conform to the following rules:

- The signature is a list of single complete types. Arrays must have element types, and structs must have both open and close parentheses.
- Only type codes, open and close parentheses, and open and close curly brackets are allowed in the signature. The `STRUCT` type code is not allowed in signatures, because parentheses are used instead. Similarly, the `DICT_ENTRY` type code is not allowed in signatures, because curly brackets are used instead.
- The maximum depth of container type nesting is 32 array type codes and 32 open parentheses. This implies that the maximum total depth of recursion is 64, for an "array of array of array of ... struct of struct of struct of ..." where there are 32 array and 32 struct.
- The maximum length of a signature is 255.

When signatures appear in messages, the marshalling format guarantees that they will be followed by a nul byte (which can be interpreted as either C-style string termination or the INVALID type-code), but this is not conceptually part of the signature.

Container types

In addition to basic types, there are four *container* types: `STRUCT`, `ARRAY`, `VARIANT`, and `DICT_ENTRY`.

`STRUCT` has a type code, ASCII character 'r', but this type code does not appear in signatures. Instead, ASCII characters '(' and ')' are used to mark the beginning and end of the struct. So for example, a struct containing two integers would have this signature:

```
"(ii)"
```

Structs can be nested, so for example a struct containing an integer and another struct:

```
"(i(ii))"
```

The value block storing that struct would contain three integers; the type signature allows you to distinguish "(i(ii))" from "((ii)i)" or "(iii)" or "iii".

The `STRUCT` type code 'r' is not currently used in the D-Bus protocol, but is useful in code that implements the protocol. This type code is specified to allow such code to interoperate in non-protocol contexts.

Empty structures are not allowed; there must be at least one type code between the parentheses.

`ARRAY` has ASCII character 'a' as type code. The array type code must be followed by a *single complete type*. The single complete type following the array is the type of each array element. So the simple example is:

```
"ai"
```

which is an array of 32-bit integers. But an array can be of any type, such as this array-of-struct-with-two-int32-fields:

```
"a(ii)"
```

Or this array of array of integer:

```
"aai"
```

`VARIANT` has ASCII character 'v' as its type code. A marshaled value of type `VARIANT` will have the signature of a single complete type as part of the *value*. This signature will be followed by a marshaled value of that type. A variant is a `union` in C, I believe, or a `std::variant` in C++.

Unlike a message signature, the variant signature can contain only a single complete type. So "i", "ai" or "(ii)" is OK, but "ii" is not. Use of variants may not cause a total message depth to be larger than 64, including other container types such as structures.

A `DICT_ENTRY` works exactly like a struct, but rather than parentheses it uses curly braces, and it has more restrictions. The restrictions are: it occurs only as an array element type; it has exactly two single complete types inside the curly braces; the first single complete type (the "key") must be a basic type rather than a container type. Implementations must not accept dict entries outside of arrays, must not accept dict entries with zero, one, or more than two fields, and must not accept dict entries with non-basic-typed keys. A dict entry is always a key-value pair.

The first field in the `DICT_ENTRY` is always the key. A message is considered corrupt if the same key occurs twice in the same array of `DICT_ENTRY`. However, for performance reasons implementations are not required to reject dicts with duplicate keys.

In most languages, an array of dict entry would be represented as a map, hash table, or dict object.

Summary of types

The following table summarizes the D-Bus types.

Category	Conventional Name	Code	Description
reserved	`INVALID`	0 (ASCII NUL)	Not a valid type code, used to terminate signatures
fixed, basic	`BYTE`	121 (ASCII 'y')	8-bit unsigned integer
fixed, basic	`BOOLEAN`	98 (ASCII 'b')	Boolean value, 0 is `FALSE` and 1 is `TRUE`. Everything else is invalid.

Category	Conventional Name	Code	Description
fixed, basic	INT16	110 (ASCII 'n')	16-bit signed integer
fixed, basic	UINT16	113 (ASCII 'q')	16-bit unsigned integer
fixed, basic	INT32	105 (ASCII 'i')	32-bit signed integer
fixed, basic	UINT32	117 (ASCII 'u')	32-bit unsigned integer
fixed, basic	INT64	120 (ASCII 'x')	64-bit signed integer
fixed, basic	UINT64	116 (ASCII 't')	64-bit unsigned integer
fixed, basic	DOUBLE	100 (ASCII 'd')	IEEE 754 double
string-like, basic	STRING	115 (ASCII 's')	UTF-8 string (*must* be valid UTF-8). Must be nul terminated and contain no other nul bytes.
string-like, basic	OBJECT_PATH	111 (ASCII 'o')	Name of an object instance
string-like, basic	SIGNATURE	103 (ASCII 'g')	A type signature
container	ARRAY	97 (ASCII 'a')	Array
container	STRUCT	114 (ASCII 'r'), 40 (ASCII '('), 41 (ASCII ')')	Struct; type code 114 'r' is reserved for use in bindings and implementations to represent the general concept of a struct, and must not appear in signatures used on D-Bus.
container	VARIANT	118 (ASCII 'v')	Variant type (the type of the value is part of the value itself)
container	DICT_ENTRY	101 (ASCII 'e'), 123 (ASCII '{'), 125 (ASCII '}')	Entry in a dict or map (array of key-value pairs). Type code 101 'e' is reserved for use in bindings and implementations to represent the general concept of a dict or dict-entry, and must not appear in signatures used on D-Bus.
fixed, basic	UNIX_FD	104 (ASCII 'h')	Unix file descriptor
reserved	(reserved)	109 (ASCII 'm')	Reserved for a 'maybe' type compatible with the one in GVariant, and must not appear in signatures used on D-Bus until specified here
reserved	(reserved)	42 (ASCII '*')	Reserved for use in bindings/implementations to represent any *single complete type*, and must not appear in signatures used on D-Bus.
reserved	(reserved)	63 (ASCII '?')	Reserved for use in bindings/implementations to represent any *basic type*, and must not appear in signatures used on D-Bus.
reserved	(reserved)	64 (ASCII '@'), 38 (ASCII '&'), 94 (ASCII '^')	Reserved for internal use by bindings/implementations, and must not appear in signatures used on D-Bus. GVariant uses these type-codes to encode calling conventions.

Marshaling (Wire Format)

D-Bus defines a marshalling format for its type system, which is used in D-Bus messages. This is not the only possible marshalling format for the type system: for instance, GVariant (part of GLib) re-uses the D-Bus type system but implements an alternative marshalling format.

Byte order and alignment

Given a type signature, a block of bytes can be converted into typed values. This section describes the format of the block of bytes. Byte order and alignment issues are handled uniformly for all D-Bus types.

A block of bytes has an associated byte order. The byte order has to be discovered in some way; for D-Bus messages, the byte order is part of the message header as described in the section called “Message Format”. For now, assume that the byte order is known to be either little endian or big endian.

Each value in a block of bytes is aligned "naturally," for example 4-byte values are aligned to a 4-byte boundary, and 8-byte values to an 8-byte boundary. Boundaries are calculated globally, with respect to the first byte in the message. To properly align a value, *alignment padding* may be necessary before the value. The alignment padding must always be the minimum required padding to properly align the following value; and it must always be made up of nul bytes. The alignment padding must not be left uninitialized (it can't contain garbage), and more padding than required must not be used.

As an exception to natural alignment, STRUCT and DICT_ENTRY values are always aligned to an 8-byte boundary, regardless of the alignments of their contents.

Marshalling basic types

To marshal and unmarshal fixed types, you simply read one value from the data block corresponding to each type code in the signature. All signed integer values are encoded in two's complement, DOUBLE values are IEEE 754 double-precision floating-point, and BOOLEAN values are encoded in 32 bits (of which only the least significant bit is used). Boolean makes sense as 32-bits, I suppose, because it would otherwise just be 1-byte with 3 bytes of padding to enforce 4-byte alignment anyway!

The string-like types (STRING, OBJECT_PATH and SIGNATURE) are all marshalled as a fixed-length unsigned integer n giving the length of the variable part, followed by n nonzero bytes of UTF-8 text, followed by a single zero (nul) byte which is not considered to be part of the text. The alignment of the string-like type is the same as

the alignment of n: any padding required for n appears immediately before n itself. There is never any alignment padding between n and the string text, or between the string text and the trailing nul. The alignment padding for the next value in the message (if there is one) starts after the trailing nul.

For the STRING and OBJECT_PATH types, n is encoded in 4 bytes (a UINT32), leading to 4-byte alignment. For the SIGNATURE type, n is encoded as a single byte (a UINT8). As a result, alignment padding is never required before a SIGNATURE.

For example, if the current position is a multiple of 8 bytes from the beginning of a little-endian message, strings 'foo', '+' and 'bar' would be serialized in sequence as follows:

```
                                    no padding required, we are already at a multiple of 4
0x03 0x00 0x00 0x00                 length of 'foo' = 3
                   0x66 0x6f 0x6f   'foo'
                               0x00 trailing nul

                                    no padding required, we are already at a multiple of 4
0x01 0x00 0x00 0x00                 length of '+' = 1
                   0x2b             '+'
                        0x00        trailing nul

                              0x00 0x00  2 bytes of padding to reach next multiple of 4
0x03 0x00 0x00 0x00                 length of 'bar' = 3
                   0x62 0x61 0x72   'bar'
                               0x00 trailing nul
```

Marshalling containers

Arrays are marshalled as a UINT32 n giving the length of the array data in bytes, followed by alignment padding to the alignment boundary of the array element type, followed by the n bytes of the array elements marshalled in sequence. n does not include the padding after the length, or any padding after the last element. i.e. n should be divisible by the number of elements in the array. Note that the alignment padding for the first element is required even if there is no first element (an empty array, where n is zero).

For instance, if the current position in the message is a multiple of 8 bytes and the byte-order is big-endian, an array containing only the 64-bit integer 5 would be marshalled as:

```
00 00 00 08               n = 8 bytes of data
00 00 00 00               padding to 8-byte boundary
00 00 00 00  00 00 00 05  first element = 5
```

Arrays have a maximum length defined to be 2 to the 26th power or 67108864 (64 MiB). Implementations must not send or accept arrays exceeding this length.

Structs and dict entries are marshalled in the same way as their contents, but their alignment is always to an 8-byte boundary, even if their contents would normally be less strictly aligned.

Variants are marshalled as the SIGNATURE of the contents (which must be a single complete type), followed by a marshalled value with the type given by that signature. The variant has the same 1-byte alignment as the signature, which means that alignment padding before a variant is never needed. Use of variants must not cause a total message depth to be larger than 64, including other container types such as structures. (See Valid Signatures.)

It should be noted that while a variant itself does not require any alignment padding, the contained value does need to be padded according to the alignment rules of its type.

For instance, if the current position in the message is at a multiple of 8 bytes and the byte-order is big-endian, a variant containing a 64-bit integer 5 would be marshalled as:

```
0x01 0x74 0x00                      signature bytes (length = 1, signature = 't' and trailing nul)
             0x00 0x00 0x00 0x00 0x00 padding to 8-byte boundary
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x05 8 bytes of contained value
```

Summary of D-Bus marshalling

marshaling = (means) "serialization"--see p4

Given all this, the types are marshaled on the wire as follows:

Conventional Name	Encoding	Alignment
INVALID	Not applicable; cannot be marshaled.	N/A
BYTE	A single 8-bit byte.	1
BOOLEAN	As for UINT32, but only 0 and 1 are valid values.	4
INT16	16-bit signed integer in the message's byte order.	2
UINT16	16-bit unsigned integer in the message's byte order.	2
INT32	32-bit signed integer in the message's byte order.	4
UINT32	32-bit unsigned integer in the message's byte order.	4
INT64	64-bit signed integer in the message's byte order.	8
UINT64	64-bit unsigned integer in the message's byte order.	8
DOUBLE	64-bit IEEE 754 double in the message's byte order.	8
STRING	A UINT32 indicating the string's length in bytes excluding its terminating nul, followed by non-nul string data of the given length, followed by a terminating nul byte.	4 (for the length)
OBJECT_PATH	Exactly the same as STRING except the content must be a valid object path (see above).	4 (for the length)

Conventional Name	Encoding	Alignment
SIGNATURE	The same as STRING except the length is a single byte (thus signatures have a maximum length of 255) and the content must be a valid signature (see above).	1
ARRAY	A UINT32 giving the length of the array data in bytes, followed by alignment padding to the alignment boundary of the array element type, followed by each array element.	4 (for the length)
STRUCT	A struct must start on an 8-byte boundary regardless of the type of the struct fields. The struct value consists of each field marshaled in sequence starting from that 8-byte alignment boundary.	8
VARIANT	The marshaled SIGNATURE of a single complete type, followed by a marshaled value with the type given in the signature.	1 (alignment of the signature)
DICT_ENTRY	Identical to STRUCT.	8
UNIX_FD	32-bit unsigned integer in the message's byte order. The actual file descriptors need to be transferred out-of-band via some platform specific mechanism. On the wire, values of this type store the index to the file descriptor in the array of file descriptors that accompany the message.	4

Message Protocol

A *message* consists of a *header* and a *body*. If you think of a message as a package, the header is the address, and the body contains the package contents. The message delivery system uses the header information to figure out where to send the message and how to interpret it; the recipient interprets the body of the message.

The body of the message is made up of zero or more *arguments*, which are typed values, such as an integer or a byte array.

Both header and body use the D-Bus type system and format for serializing data.

Message Format

A message consists of a header and a body. The header is a block of values with a fixed signature and meaning. The body is a separate block of values, with a signature specified in the header.

The length of the header must be a multiple of 8, allowing the body to begin on an 8-byte boundary when storing the entire message in a single buffer. If the header does not naturally end on an 8-byte boundary up to 7 bytes of nul-initialized alignment padding must be added.

The message body need not end on an 8-byte boundary.

The maximum length of a message, including header, header alignment padding, and body is 2 to the 27th power or 134217728 (128 MiB). Implementations must not send or accept messages exceeding this size.

The signature of the header is:

```
"yyyyuua(yv)"
```

Written out more readably, this is:

```
BYTE, BYTE, BYTE, BYTE, UINT32, UINT32, ARRAY of STRUCT of (BYTE,VARIANT)
```

These values have the following meanings: Header definition:

Value	Description
1st BYTE	Endianness flag; ASCII 'l' for little-endian or ASCII 'B' for big-endian. Both header and body are in this endianness.
2nd BYTE	*Message type*. Unknown types must be ignored. Currently-defined types are described below.
3rd BYTE	Bitwise OR of flags. Unknown flags must be ignored. Currently-defined flags are described below.
4th BYTE	Major protocol version of the sending application. If the major protocol version of the receiving application does not match, the applications will not be able to communicate and the D-Bus connection must be disconnected. The major protocol version for this version of the specification is 1.
1st UINT32	Length in bytes of the message body, starting from the end of the header. The header ends after its alignment padding to an 8-boundary.
2nd UINT32	The serial of this message, used as a cookie by the sender to identify the reply corresponding to this request. This must not be zero.
ARRAY of STRUCT of (BYTE,VARIANT)	An array of zero or more *header fields* where the byte is the field code, and the variant is the field value. The message type determines which fields are required.

Message types that can appear in the second byte of the header are:

Conventional name	Decimal value	Description
INVALID	0	This is an invalid type.
METHOD_CALL	1	Method call. This message type may prompt a reply.
METHOD_RETURN	2	Method reply with returned data.
ERROR	3	Error reply. If the first argument exists and is a string, it is an error message.
SIGNAL	4	Signal emission.

Flags that can appear in the third byte of the header:

Conventional name	Hex value	Description
NO_REPLY_EXPECTED	0x1	This message does not expect method return replies or error replies, even if it is of a type that can have a reply; the reply should be omitted. Note that METHOD_CALL is the only message type currently defined in this specification that can expect a reply, so the presence or absence of this flag in the other three message types that are currently documented is meaningless: replies to those message types should not be sent, whether this flag is present or not.
NO_AUTO_START	0x2	The bus must not launch an owner for the destination name in response to this message.
ALLOW_INTERACTIVE_AUTHORIZATION	0x4	This flag may be set on a method call message to inform the receiving side that the caller is prepared to wait for interactive authorization, which might take a considerable time to complete. For instance, if this flag is set, it would be appropriate to query the user for passwords or confirmation via Polkit or a similar framework. This flag is only useful when unprivileged code calls a more privileged method call, and an authorization framework is deployed that allows possibly interactive authorization. If no such framework is deployed it has no effect. This flag should not be set by default by client implementations. If it is set, the caller should also set a suitably long timeout on the method call to make sure the user interaction may complete. This flag is only valid for method call messages, and shall be ignored otherwise. Interaction that takes place as a part of the effect of the method being called is outside the scope of this flag, even if it could also be characterized as authentication or authorization. For instance, in a method call that directs a network management service to attempt to connect to a virtual private network, this flag should control how the network management service makes the decision "is this user allowed to change system network configuration?", but it should not affect how or whether the network management service interacts with the user to obtain the credentials that are required for access to the VPN. If a this flag is not set on a method call, and a service determines that the requested operation is not allowed without interactive authorization, but could be allowed after successful interactive authorization, it may return the `org.freedesktop.DBus.Error.InteractiveAuthorizationRequired` error. The absence of this flag does not guarantee that interactive authorization will not be applied, since existing services that pre-date this flag might already use interactive authorization. However, existing D-Bus APIs that will use interactive authorization should document that the call may take longer than usual, and new D-Bus APIs should avoid interactive authorization in the absence of this flag.

Header Fields

The array at the end of the header contains *header fields*, where each field is a 1-byte field code followed by a field value. A header must contain the required header fields for its message type, and zero or more of any optional header fields. Future versions of this protocol specification may add new fields. Implementations must not invent their own header fields; only changes to this specification may introduce new header fields.

If an implementation sees a header field code that it does not expect, it must accept and ignore that field, as it will be part of a new (but compatible) version of this specification. This also applies to known header fields appearing in unexpected messages, for example: if a signal has a reply serial it must be ignored even though it has no meaning as of this version of the spec.

However, implementations must not send or accept known header fields with the wrong type stored in the field value. So for example a message with an `INTERFACE` field of type `UINT32` would be considered corrupt.

Server implementations that might relay messages from one mutually-distrustful client to another, such as the message bus, should remove header fields that the server does not recognise. However, a client must assume that the server has not done so, unless it has evidence to the contrary, such as having checked for the `HeaderFiltering` message bus feature.

New header fields controlled by the message bus (similar to `SENDER`) might be added to this specification in future. Such message fields should normally only be added to messages that are going to be delivered to a client that specifically requested them (for example by calling some method), and the message bus should remove those header fields from all other messages that it relays. This design principle serves two main purposes. One is to avoid unnecessary memory and throughput overhead when delivering messages to clients that are not interested in the new header fields. The other is to give clients a reason to call the method that requests those messages (otherwise, the clients would not work). This is desirable because looking at the reply to that method call is a natural way to check that the message bus guarantees to filter out faked header fields that might have been sent by malicious peers.

Here are the currently-defined header fields:

Conventional Name	Decimal Code	Type	Required In	Description
INVALID	0	N/A	not allowed	Not a valid field name (error if it appears in a message)
PATH	1	OBJECT_PATH	METHOD_CALL, SIGNAL	The object to send a call to, or the object a signal is emitted from. The special path `/org/freedesktop/DBus/Local` is reserved; implementations should not send messages with this path, and the reference implementation of the bus daemon will disconnect any application that attempts to do so. This header field is controlled by the message sender.
INTERFACE	2	STRING	SIGNAL	The interface to invoke a method call on, or that a signal is emitted from. Optional for method calls, required for signals. The special interface `org.freedesktop.DBus.Local` is reserved; implementations should not send messages with this interface, and the reference implementation of the bus daemon will disconnect any application that attempts to do so. This header field is controlled by the message sender.
MEMBER	3	STRING	METHOD_CALL, SIGNAL	The member, either the method name or signal name. This header field is controlled by the message sender.
ERROR_NAME	4	STRING	ERROR	The name of the error that occurred, for errors
REPLY_SERIAL	5	UINT32	ERROR, METHOD_RETURN	The serial number of the message this message is a reply to. (The serial number is the second `UINT32` in the header.) This header field is controlled by the message sender.

Conventional Name	Decimal Code	Type	Required In	Description
DESTINATION	6	STRING	optional	The name of the connection this message is intended for. This field is usually only meaningful in combination with the message bus (see the section called “Message Bus Specification”), but other servers may define their own meanings for it. This header field is controlled by the message sender.
SENDER	7	STRING	optional	Unique name of the sending connection. This field is usually only meaningful in combination with the message bus, but other servers may define their own meanings for it. On a message bus, this header field is controlled by the message bus, so it is as reliable and trustworthy as the message bus itself. Otherwise, this header field is controlled by the message sender, unless there is out-of-band information that indicates otherwise.
SIGNATURE	8	SIGNATURE	optional	The signature of the message body. If omitted, it is assumed to be the empty signature "" (i.e. the body must be 0-length). This header field is controlled by the message sender.
UNIX_FDS	9	UINT32	optional	The number of Unix file descriptors that accompany the message. If omitted, it is assumed that no Unix file descriptors accompany the message. The actual file descriptors need to be transferred via platform specific mechanism out-of-band. They must be sent at the same time as part of the message itself. They may not be sent before the first byte of the message itself is transferred or after the last byte of the message itself. This header field is controlled by the message sender.

Valid Names

The various names in D-Bus messages have some restrictions.

There is a *maximum name length* of 255 which applies to bus names, interfaces, and members.

Interface names

Interfaces have names with type `STRING`, meaning that they must be valid UTF-8. However, there are also some additional restrictions that apply to interface names specifically:

- Interface names are composed of 2 or more elements separated by a period ('.') character. All elements must contain at least one character.
- Each element must only contain the ASCII characters "[A-Z][a-z][0-9]_" and must not begin with a digit.
- Interface names must not exceed the maximum name length.

Interface names should start with the reversed DNS domain name of the author of the interface (in lower-case), like interface names in Java. It is conventional for the rest of the interface name to consist of words run together, with initial capital letters on all words ("CamelCase"). Several levels of hierarchy can be used. It is also a good idea to include the major version of the interface in the name, and increment it if incompatible changes are made; this way, a single object can implement several versions of an interface in parallel, if necessary.

For instance, if the owner of `example.com` is developing a D-Bus API for a music player, they might define interfaces called `com.example.MusicPlayer1`, `com.example.MusicPlayer1.Track` and `com.example.MusicPlayer1.Seekable`.

If the author's DNS domain name contains hyphen/minus characters ('-'), which are not allowed in D-Bus interface names, they should be replaced by underscores. If the DNS domain name contains a digit immediately following a period ('.'), which is also not allowed in interface names), the interface name should add an underscore before that digit. For example, if the owner of 7-zip.org defined an interface for out-of-process plugins, it might be named `org._7_zip.Plugin`.

D-Bus does not distinguish between the concepts that would be called classes and interfaces in Java: either can be identified on D-Bus by an interface name.

Bus names

Connections have one or more bus names associated with them. A connection has exactly one bus name that is a *unique connection name*. The unique connection name remains with the connection for its entire lifetime. A bus name is of type `STRING`, meaning that it must be valid UTF-8. However, there are also some additional restrictions that apply to bus names specifically:

- Bus names that start with a colon (':') character are unique connection names. Other bus names are called *well-known bus names*.
- Bus names are composed of 1 or more elements separated by a period ('.') character. All elements must contain at least one character.
- Each element must only contain the ASCII characters "[A-Z][a-z][0-9]_-", with "-" discouraged in new bus names. Only elements that are part of a unique connection name may begin with a digit, elements in other bus names must not begin with a digit.
- Bus names must contain at least one '.' (period) character (and thus at least two elements).
- Bus names must not begin with a '.' (period) character.
- Bus names must not exceed the maximum name length.

Note that the hyphen ('-') character is allowed in bus names but not in interface names. It is also problematic or not allowed in various specifications and APIs that refer to D-Bus, such as Flatpak application IDs, the `DBusActivatable` interface in the Desktop Entry Specification, and the convention that an application's "main" interface and object path resemble its bus name. To avoid situations that require special-case handling, it is recommended that new D-Bus names consistently replace hyphens with underscores.

Like interface names, well-known bus names should start with the reversed DNS domain name of the author of the interface (in lower-case), and it is conventional for the rest of the well-known bus name to consist of words run together, with initial capital letters. As with interface names, including a version number in well-known bus names is a good idea; it's possible to have the well-known bus name for more than one version simultaneously if backwards compatibility is required.

As with interface names, if the author's DNS domain name contains hyphen/minus characters they should be replaced by underscores, and if it contains leading digits they should be escaped by prepending an underscore. For example, if the owner of 7-zip.org used a D-Bus name for an archiving application, it might be named `org._7_zip.Archiver`.

If a well-known bus name implies the presence of a "main" interface, that "main" interface is often given the same name as the well-known bus name, and situated at the corresponding object path. For instance, if the owner of `example.com` is developing a D-Bus API for a music player, they might define that any application that takes the well-known name `com.example.MusicPlayer1` should have an object at the object path `/com/example/MusicPlayer1` which implements the interface `com.example.MusicPlayer1`.

Member names

Member (i.e. method or signal) names:

- Must only contain the ASCII characters "[A-Z][a-z][0-9]_" and may not begin with a digit.
- Must not contain the '.' (period) character.
- Must not exceed the maximum name length.
- Must be at least 1 byte in length.

It is conventional for member names on D-Bus to consist of capitalized words with no punctuation ("camel-case"). Method names should usually be verbs, such as `GetItems`, and signal names should usually be a description of an event, such as `ItemsChanged`.

Error names

Error names have the same restrictions as interface names.

Error names have the same naming conventions as interface names, and often contain `.Error.`; for instance, the owner of `example.com` might define the errors `com.example.MusicPlayer1.Error.FileNotFound` and `com.example.MusicPlayer1.Error.OutOfMemory`. The errors defined by D-Bus itself, such as `org.freedesktop.DBus.Error.Failed`, follow a similar pattern.

Message Types

Each of the message types (`METHOD_CALL`, `METHOD_RETURN`, `ERROR`, and `SIGNAL`) has its own expected usage conventions and header fields. This section describes these conventions.

Method Calls

Some messages invoke an operation on a remote object. These are called method call messages and have the type tag `METHOD_CALL`. Such messages map naturally to methods on objects in a typical program.

A method call message is required to have a `MEMBER` header field indicating the name of the method. Optionally, the message has an `INTERFACE` field giving the interface the method is a part of. Including the `INTERFACE` in all method call messages is strongly recommended.

In the absence of an `INTERFACE` field, if two or more interfaces on the same object have a method with the same name, it is undefined which of those methods will be invoked. Implementations may choose to either return an error, or deliver the message as though it had an arbitrary one of those interfaces.

In some situations (such as the well-known system bus), messages are filtered through an access-control list external to the remote object implementation. If that filter rejects certain messages by matching their interface, or accepts only messages to specific interfaces, it must also reject messages that have no `INTERFACE`: otherwise, malicious applications could use this to bypass the filter.

Method call messages also include a `PATH` field indicating the object to invoke the method on. If the call is passing through a message bus, the message will also have a `DESTINATION` field giving the name of the connection to receive the message.

When an application handles a method call message, it is required to return a reply. The reply is identified by a `REPLY_SERIAL` header field indicating the serial number of the `METHOD_CALL` being replied to. The reply can have one of two types; either `METHOD_RETURN` or `ERROR`.

If the reply has type `METHOD_RETURN`, the arguments to the reply message are the return value(s) or "out parameters" of the method call. If the reply has type `ERROR`, then an "exception" has been thrown, and the call fails; no return value will be provided. It makes no sense to send multiple replies to the same method call.

Even if a method call has no return values, a `METHOD_RETURN` reply is required, so the caller will know the method was successfully processed.

The `METHOD_RETURN` or `ERROR` reply message must have the `REPLY_SERIAL` header field.

If a `METHOD_CALL` message has the flag `NO_REPLY_EXPECTED`, then the application receiving the method should not send the reply message (regardless of whether the reply would have been `METHOD_RETURN` or `ERROR`).

Unless a message has the flag `NO_AUTO_START`, if the destination name does not exist then a program to own the destination name will be started (activated) before the message is delivered. See the section called “Message Bus Starting Services (Activation)”. The message will be held until the new program is successfully started or has failed to start; in case of failure, an error will be returned. This flag is only relevant in the context of a message bus, it is ignored during one-to-one communication with no intermediate bus.

Mapping method calls to native APIs

APIs for D-Bus may map method calls to a method call in a specific programming language, such as C++, or may map a method call written in an IDL to a D-Bus message.

In APIs of this nature, arguments to a method are often termed "in" (which implies sent in the `METHOD_CALL`), or "out" (which implies returned in the `METHOD_RETURN`). Some APIs such as CORBA also have "inout" arguments, which are both sent and received, i.e. the caller passes in a value which is modified. Mapped to D-Bus, an "inout" argument is equivalent to an "in" argument, followed by an "out" argument. You can't pass things "by reference" over the wire, so "inout" is purely an illusion of the in-process API.

Given a method with zero or one return values, followed by zero or more arguments, where each argument may be "in", "out", or "inout", the caller constructs a message by appending each "in" or "inout" argument, in order. "out" arguments are not represented in the caller's message.

The recipient constructs a reply by appending first the return value if any, then each "out" or "inout" argument, in order. "in" arguments are not represented in the reply message.

Error replies are normally mapped to exceptions in languages that have exceptions.

In converting from native APIs to D-Bus, it is perhaps nice to map D-Bus naming conventions ("FooBar") to native conventions such as "fooBar" or "foo_bar" automatically. This is OK as long as you can say that the native API is one that was specifically written for D-Bus. It makes the most sense when writing object implementations that will be exported over the bus. Object proxies used to invoke remote D-Bus objects probably need the ability to call any D-Bus method, and thus a magic name mapping like this could be a problem.

This specification doesn't require anything of native API bindings; the preceding is only a suggested convention for consistency among bindings.

Signal Emission

Unlike method calls, signal emissions have no replies. A signal emission is simply a single message of type `SIGNAL`. It must have three header fields: `PATH` giving the object the signal was emitted from, plus `INTERFACE` and `MEMBER` giving the fully-qualified name of the signal. The `INTERFACE` header is required for signals, though it is optional for method calls.

Errors

Messages of type `ERROR` are most commonly replies to a `METHOD_CALL`, but may be returned in reply to any kind of message. The message bus for example will return an `ERROR` in reply to a signal emission if the bus does not have enough memory to send the signal.

An `ERROR` may have any arguments, but if the first argument is a `STRING`, it must be an error message. The error message may be logged or shown to the user in some way.

Notation in this document

This document uses a simple pseudo-IDL to describe particular method calls and signals. Here is an example of a method call:

```
org.freedesktop.DBus.StartServiceByName (in STRING name, in UINT32 flags,
                                         out UINT32 resultcode)
```

This means `INTERFACE` = org.freedesktop.DBus, `MEMBER` = StartServiceByName, `METHOD_CALL` arguments are `STRING` and `UINT32`, `METHOD_RETURN` argument is `UINT32`. Remember that the `MEMBER` field can't contain any '.' (period) characters so it's known that the last part of the name in the "IDL" is the member name.

In C++ that might end up looking like this:

```
unsigned int org::freedesktop::DBus::StartServiceByName (const char  *name,
                                                          unsigned int flags);
```

or equally valid, the return value could be done as an argument:

```
void org::freedesktop::DBus::StartServiceByName (const char   *name,
                                                 unsigned int  flags,
                                                 unsigned int *resultcode);
```

It's really up to the API designer how they want to make this look. You could design an API where the namespace wasn't used in C++, using STL or Qt, using varargs, or whatever you wanted.

Signals are written as follows:

```
org.freedesktop.DBus.NameLost (STRING name)
```

Signals don't specify "in" vs. "out" because only a single direction is possible.

It isn't especially encouraged to use this lame pseudo-IDL in actual API implementations; you might use the native notation for the language you're using, or you might use COM or CORBA IDL, for example.

Invalid Protocol and Spec Extensions

For security reasons, the D-Bus protocol should be strictly parsed and validated, with the exception of defined extension points. Any invalid protocol or spec violations should result in immediately dropping the connection without notice to the other end. Exceptions should be carefully considered, e.g. an exception may be warranted for a well-understood idiosyncrasy of a widely-deployed implementation. In cases where the other end of a connection is 100% trusted and known to be friendly, skipping validation for performance reasons could also make sense in certain cases.

Generally speaking violations of the "must" requirements in this spec should be considered possible attempts to exploit security, and violations of the "should" suggestions should be considered legitimate (though perhaps they should generate an error in some cases).

The following extension points are built in to D-Bus on purpose and must not be treated as invalid protocol. The extension points are intended for use by future versions of this spec, they are not intended for third parties. At the moment, the only way a third party could extend D-Bus without breaking interoperability would be to introduce a way to negotiate new feature support as part of the auth protocol, using EXTENSION_-prefixed commands. There is not yet a standard way to negotiate features.

- In the authentication protocol (see the section called “Authentication Protocol”) unknown commands result in an ERROR rather than a disconnect. This enables future extensions to the protocol. Commands starting with EXTENSION_ are reserved for third parties.
- The authentication protocol supports pluggable auth mechanisms.
- The address format (see the section called “Server Addresses”) supports new kinds of transport.

- Messages with an unknown type (something other than `METHOD_CALL`, `METHOD_RETURN`, `ERROR`, `SIGNAL`) are ignored. Unknown-type messages must still be well-formed in the same way as the known messages, however. They still have the normal header and body.
- Header fields with an unknown or unexpected field code must be ignored, though again they must still be well-formed.
- New standard interfaces (with new methods and signals) can of course be added.

Authentication Protocol

Before the flow of messages begins, two applications must authenticate. A simple plain-text protocol is used for authentication; this protocol is a SASL profile, and maps fairly directly from the SASL specification. The message encoding is NOT used here, only plain text messages.

Using SASL in D-Bus requires that we define the meaning of non-empty authorization identity strings. When D-Bus is used on Unix platforms, a non-empty SASL authorization identity represents a Unix user. An authorization identity consisting entirely of ASCII decimal digits represents a numeric user ID as defined by POSIX, for example `0` for the root user or `1000` for the first user created on many systems. Non-numeric authorization identities are not required to be accepted or supported, but if used, they must be interpreted as a login name as found in the `pw_name` field of POSIX `struct passwd`, for example `root`, and normalized to the corresponding numeric user ID. For best interoperability, clients and servers should use numeric user IDs.

When D-Bus is used on Windows platforms, a non-empty SASL authorization identity represents a Windows security identifier (SID) in its string form, for example `S-1-5-21-3623811015-3361044348-30300820-1013` for a domain or local computer user or `S-1-5-18` for the LOCAL_SYSTEM user. The user-facing usernames such as `Administrator` or `LOCAL_SYSTEM` are not used in the D-Bus protocol.

In examples, "C:" and "S:" indicate lines sent by the client and server respectively. The client sends the first line, and the server must respond to each line from the client with a single-line reply, with one exception: there is no reply to the BEGIN command.

Protocol Overview

The protocol is a line-based protocol, where each line ends with \r\n. Each line begins with an all-caps ASCII command name containing only the character range [A-Z_], a space, then any arguments for the command, then the \r\n ending the line. The protocol is case-sensitive. All bytes must be in the ASCII character set. Commands from the client to the server are as follows:

- AUTH [mechanism] [initial-response]
- CANCEL
- BEGIN
- DATA <data in hex encoding>
- ERROR [human-readable error explanation]
- NEGOTIATE_UNIX_FD

From server to client are as follows:

- REJECTED <space-separated list of mechanism names>
- OK <GUID in hex>
- DATA <data in hex encoding>
- ERROR [human-readable error explanation]
- AGREE_UNIX_FD

Unofficial extensions to the command set must begin with the letters "EXTENSION_", to avoid conflicts with future official commands. For example, "EXTENSION_COM_MYDOMAIN_DO_STUFF".

Special credentials-passing nul byte

Immediately after connecting to the server, the client must send a single nul byte. This byte may be accompanied by credentials information on some operating systems that use sendmsg() with SCM_CREDS or SCM_CREDENTIALS to pass credentials over UNIX domain sockets. However, the nul byte must be sent even on other kinds of socket, and even on operating systems that do not require a byte to be sent in order to transmit credentials. The text protocol described in this document begins after the single nul byte. If the first byte received from the client is not a nul byte, the server may disconnect that client.

A nul byte in any context other than the initial byte is an error; the protocol is ASCII-only.

The credentials sent along with the nul byte may be used with the SASL mechanism EXTERNAL.

AUTH command

The AUTH command is sent by the client to the server. The server replies with DATA, OK or REJECTED.

If an AUTH command has no arguments, it is a request to list available mechanisms. The server must respond with a REJECTED command listing the mechanisms it understands, or with an error.

If an AUTH command specifies a mechanism, and the server supports said mechanism, the server should begin exchanging SASL challenge-response data with the client using DATA commands.

If the server does not support the mechanism given in the AUTH command, it must send either a REJECTED command listing the mechanisms it does support, or an error.

If the [initial-response] argument is provided, it is intended for use with mechanisms that have no initial challenge (or an empty initial challenge), as if it were the argument to an initial DATA command. If the selected mechanism has an initial challenge and [initial-response] was provided, the server should reject authentication by sending REJECTED.

If authentication succeeds after exchanging DATA commands, an OK command must be sent to the client.

CANCEL Command

The CANCEL command is sent by the client to the server. The server replies with REJECTED.

At any time up to sending the BEGIN command, the client may send a CANCEL command. On receiving the CANCEL command, the server must send a REJECTED command and abort the current authentication exchange.

DATA Command

The DATA command may come from either client or server, and simply contains a hex-encoded block of data to be interpreted according to the SASL mechanism in use. If sent by the client, the server replies with DATA, OK or REJECTED.

Some SASL mechanisms support sending an "empty string"; FIXME we need some way to do this.

BEGIN Command

The BEGIN command is sent by the client to the server. The server does not reply.

The BEGIN command acknowledges that the client has received an OK command from the server and completed any feature negotiation that it wishes to do, and declares that the stream of messages is about to begin.

The first octet received by the server after the \r\n of the BEGIN command from the client must be the first octet of the authenticated/encrypted stream of D-Bus messages.

Unlike all other commands, the server does not reply to the BEGIN command with an authentication command of its own. After the \r\n of the reply to the command before BEGIN, the next octet received by the client must be the first octet of the authenticated/encrypted stream of D-Bus messages.

REJECTED Command

The REJECTED command is sent by the server to the client.

The REJECTED command indicates that the current authentication exchange has failed, and further exchange of DATA is inappropriate. The client would normally try another mechanism, or try providing different responses to challenges.

Optionally, the REJECTED command has a space-separated list of available auth mechanisms as arguments. If a server ever provides a list of supported mechanisms, it must provide the same list each time it sends a REJECTED message. Clients are free to ignore all lists received after the first.

OK Command

The OK command is sent by the server to the client.

The OK command indicates that the client has been authenticated. The client may now proceed with negotiating Unix file descriptor passing. To do that it shall send NEGOTIATE_UNIX_FD to the server.

Otherwise, the client must respond to the OK command by sending a BEGIN command, followed by its stream of messages, or by disconnecting. The server must not accept additional commands using this protocol after the BEGIN command has been received. Further communication will be a stream of D-Bus messages (optionally encrypted, as negotiated) rather than this protocol.

If there is no negotiation, the first octet received by the client after the \r\n of the OK command must be the first octet of the authenticated/encrypted stream of D-Bus messages. If the client negotiates Unix file descriptor passing, the first octet received by the client after the \r\n of the AGREE_UNIX_FD or ERROR reply must be the first octet of the authenticated/encrypted stream.

The OK command has one argument, which is the GUID of the server. See the section called “Server Addresses” for more on server GUIDs.

ERROR Command

The ERROR command can be sent in either direction. If sent by the client, the server replies with REJECTED.

The ERROR command indicates that either server or client did not know a command, does not accept the given command in the current context, or did not understand the arguments to the command. This allows the protocol to be extended; a client or server can send a command present or permitted only in new protocol versions, and if an ERROR is received instead of an appropriate response, fall back to using some other technique.

If an ERROR is sent, the server or client that sent the error must continue as if the command causing the ERROR had never been received. However, the the server or client receiving the error should try something other than whatever caused the error; if only canceling/rejecting the authentication.

If the D-Bus protocol changes incompatibly at some future time, applications implementing the new protocol would probably be able to check for support of the new protocol by sending a new command and receiving an ERROR from applications that don't understand it. Thus the ERROR feature of the auth protocol is an escape hatch that lets us negotiate extensions or changes to the D-Bus protocol in the future.

NEGOTIATE_UNIX_FD Command

The NEGOTIATE_UNIX_FD command is sent by the client to the server. The server replies with AGREE_UNIX_FD or ERROR.

The NEGOTIATE_UNIX_FD command indicates that the client supports Unix file descriptor passing. This command may only be sent after the connection is authenticated, i.e. after OK was received by the client. This command may only be sent on transports that support Unix file descriptor passing.

On receiving NEGOTIATE_UNIX_FD the server must respond with either AGREE_UNIX_FD or ERROR. It shall respond the former if the transport chosen supports Unix file descriptor passing and the server supports this feature. It shall respond the latter if the transport does not support Unix file descriptor passing, the server does not support this feature, or the server decides not to enable file descriptor passing due to security or other reasons.

AGREE_UNIX_FD Command

The AGREE_UNIX_FD command is sent by the server to the client.

The AGREE_UNIX_FD command indicates that the server supports Unix file descriptor passing. This command may only be sent after the connection is authenticated, and the client sent NEGOTIATE_UNIX_FD to enable Unix file descriptor passing. This command may only be sent on transports that support Unix file descriptor passing.

On receiving AGREE_UNIX_FD the client must respond with BEGIN, followed by its stream of messages, or by disconnecting. The server must not accept additional commands using this protocol after the BEGIN command has been received. Further communication will be a stream of D-Bus messages (optionally encrypted, as negotiated) rather than this protocol.

Future Extensions

Future extensions to the authentication and negotiation protocol are possible. For that new commands may be introduced. If a client or server receives an unknown command it shall respond with ERROR and not consider this fatal. New commands may be introduced both before, and after authentication, i.e. both before and after the OK command.

Authentication examples

Figure 1. Example of successful EXTERNAL authentication

```
31303030 is ASCII decimal "1000" represented in hex, so
the client is authenticating as Unix uid 1000 in this example.

C: AUTH EXTERNAL 31303030
S: OK 1234deadbeef
C: BEGIN
```

Figure 2. Example of finding out mechanisms then picking one

```
C: AUTH
S: REJECTED KERBEROS_V4 SKEY
C: AUTH SKEY 7ab83f32ee
S: DATA 8799cabb2ea93e
C: DATA 8ac876e8f68ee9809bfa876e6f9876g8fa8e76e98f
S: OK 1234deadbeef
C: BEGIN
```

Figure 3. Example of client sends unknown command then falls back to regular auth

```
532d312d352d3138 is the Windows SID "S-1-5-18" in hex,
so the client is authenticating as Windows SID S-1-5-18
in this example.

C: FOOBAR
S: ERROR
C: AUTH EXTERNAL 532d312d352d3138
S: OK 1234deadbeef
C: BEGIN
```

Figure 4. Example of server doesn't support initial auth mechanism

```
C: AUTH EXTERNAL
S: REJECTED KERBEROS_V4 SKEY
C: AUTH SKEY 7ab83f32ee
S: DATA 8799cabb2ea93e
C: DATA 8ac876e8f68ee9809bfa876e6f9876g8fa8e76e98f
S: OK 1234deadbeef
C: BEGIN
```

Figure 5. Example of wrong password or the like followed by successful retry

```
C: AUTH EXTERNAL 736d6376
S: REJECTED KERBEROS_V4 SKEY
C: AUTH SKEY 7ab83f32ee
S: DATA 8799cabb2ea93e
C: DATA 8ac876e8f68ee9809bfa876e6f9876g8fa8e76e98f
S: REJECTED
C: AUTH SKEY 7ab83f32ee
S: DATA 8799cabb2ea93e
C: DATA 8ac876e8f68ee9809bfa876e6f9876g8fa8e76e98f
S: OK 1234deadbeef
```

```
C: BEGIN
```

Figure 6. Example of skey cancelled and restarted

```
C: AUTH EXTERNAL 32303438
S: REJECTED KERBEROS_V4 SKEY
C: AUTH SKEY 7ab83f32ee
S: DATA 8799cabb2ea93e
C: CANCEL
S: REJECTED
C: AUTH SKEY 7ab83f32ee
S: DATA 8799cabb2ea93e
C: DATA 8ac876e8f68ee9809bfa876e6f9876g8fa8e76e98f
S: OK 1234deadbeef
C: BEGIN
```

Figure 7. Example of successful EXTERNAL authentication with successful negotiation of Unix FD passing

```
C: AUTH EXTERNAL 31303030
S: OK 1234deadbeef
C: NEGOTIATE_UNIX_FD
S: AGREE_UNIX_FD
C: BEGIN
```

Figure 8. Example of successful EXTERNAL authentication with unsuccessful negotiation of Unix FD passing

```
C: AUTH EXTERNAL 31303030
S: OK 1234deadbeef
C: NEGOTIATE_UNIX_FD
S: ERROR Not supported on this OS
C: BEGIN
```

Authentication state diagrams

This section documents the auth protocol in terms of a state machine for the client and the server. This is probably the most robust way to implement the protocol.

Client states

To more precisely describe the interaction between the protocol state machine and the authentication mechanisms the following notation is used: MECH(CHALL) means that the server challenge CHALL was fed to the mechanism MECH, which returns one of

- CONTINUE(RESP) means continue the auth conversation and send RESP as the response to the server;
- OK(RESP) means that after sending RESP to the server the client side of the auth conversation is finished and the server should return "OK";
- ERROR means that CHALL was invalid and could not be processed.

Both RESP and CHALL may be empty.

The Client starts by getting an initial response from the default mechanism and sends AUTH MECH RESP, or AUTH MECH if the mechanism did not provide an initial response. If the mechanism returns CONTINUE, the client starts in state *WaitingForData*, if the mechanism returns OK the client starts in state *WaitingForOK*.

The client should keep track of available mechanisms and which it mechanisms it has already attempted. This list is used to decide which AUTH command to send. When the list is exhausted, the client should give up and close the connection.

***WaitingForData*.**

- Receive DATA CHALL

  MECH(CHALL) returns CONTINUE(RESP) → send DATA RESP, goto *WaitingForData*
  MECH(CHALL) returns OK(RESP) → send DATA RESP, goto *WaitingForOK*
  MECH(CHALL) returns ERROR → send ERROR [msg], goto *WaitingForData*

- Receive REJECTED [mechs] → send AUTH [next mech], goto WaitingForData or *WaitingForOK*
- Receive ERROR → send CANCEL, goto *WaitingForReject*
- Receive OK → *authenticated*, choose one:

  send NEGOTIATE_UNIX_FD, goto *WaitingForAgreeUnixFD*
  send BEGIN, terminate auth conversation (successfully)

- Receive anything else → send ERROR, goto *WaitingForData*

WaitingForOK.

- Receive OK → *authenticated*, choose one:

  send NEGOTIATE_UNIX_FD, goto *WaitingForAgreeUnixFD*
  send BEGIN, terminate auth conversation (successfully)

- Receive REJECTED [mechs] → send AUTH [next mech], goto *WaitingForData* or *WaitingForOK*
- Receive DATA → send CANCEL, goto *WaitingForReject*
- Receive ERROR → send CANCEL, goto *WaitingForReject*
- Receive anything else → send ERROR, goto *WaitingForOK*

WaitingForReject.

- Receive REJECTED [mechs] → send AUTH [next mech], goto *WaitingForData* or *WaitingForOK*
- Receive anything else → terminate auth conversation, disconnect

WaitingForAgreeUnixFD. By the time this state is reached, the client has already been authenticated.

- Receive AGREE_UNIX_FD → enable Unix fd passing, send BEGIN, terminate auth conversation (successfully)
- Receive ERROR → disable Unix fd passing, send BEGIN, terminate auth conversation (successfully)
- Receive anything else → terminate auth conversation, disconnect

Server states

For the server MECH(RESP) means that the client response RESP was fed to the the mechanism MECH, which returns one of

- CONTINUE(CHALL) means continue the auth conversation and send CHALL as the challenge to the client;
- OK means that the client has been successfully authenticated;
- REJECTED means that the client failed to authenticate or there was an error in RESP.

The server starts out in state *WaitingForAuth*. If the client is rejected too many times the server must disconnect the client.

WaitingForAuth.

- Receive AUTH → send REJECTED [mechs], goto *WaitingForAuth*
- Receive AUTH MECH RESP

  MECH not valid mechanism → send REJECTED [mechs], goto *WaitingForAuth*
  MECH(RESP) returns CONTINUE(CHALL) → send DATA CHALL, goto *WaitingForData*
  MECH(RESP) returns OK → send OK, goto *WaitingForBegin*
  MECH(RESP) returns REJECTED → send REJECTED [mechs], goto *WaitingForAuth*

- Receive BEGIN → terminate auth conversation, disconnect
- Receive ERROR → send REJECTED [mechs], goto *WaitingForAuth*
- Receive anything else → send ERROR, goto *WaitingForAuth*

WaitingForData.

- Receive DATA RESP

  MECH(RESP) returns CONTINUE(CHALL) → send DATA CHALL, goto *WaitingForData*
  MECH(RESP) returns OK → send OK, goto *WaitingForBegin*
  MECH(RESP) returns REJECTED → send REJECTED [mechs], goto *WaitingForAuth*

- Receive BEGIN → terminate auth conversation, disconnect
- Receive CANCEL → send REJECTED [mechs], goto *WaitingForAuth*
- Receive ERROR → send REJECTED [mechs], goto *WaitingForAuth*
- Receive anything else → send ERROR, goto *WaitingForData*

WaitingForBegin.

- Receive BEGIN → terminate auth conversation, client authenticated
- Receive NEGOTIATE_UNIX_FD → send AGREE_UNIX_FD or ERROR, goto *WaitingForBegin*
- Receive CANCEL → send REJECTED [mechs], goto *WaitingForAuth*

- Receive ERROR → send REJECTED [mechs], goto *WaitingForAuth*
- Receive anything else → send ERROR, goto *WaitingForBegin*

Authentication mechanisms

This section describes some authentication mechanisms that are often supported by practical D-Bus implementations. The D-Bus protocol also allows any other standard SASL mechanism, although implementations of D-Bus often do not.

EXTERNAL

The EXTERNAL mechanism is defined in RFC 4422 "Simple Authentication and Security Layer (SASL)", appendix A "The SASL EXTERNAL Mechanism". This is the recommended authentication mechanism on platforms where credentials can be transferred out-of-band, in particular Unix platforms that can perform credentials-passing over the unix: transport.

On Unix platforms, interoperable clients should prefer to send the ASCII decimal string form of the integer Unix user ID as the authorization identity, for example 1000. When encoded in hex by the authentication protocol, this will typically result in a line like `AUTH EXTERNAL 31303030` followed by \r\n.

On Windows platforms, clients that use the EXTERNAL mechanism should use the Windows security identifier in its string form as the authorization identity, for example `S-1-5-21-3623811015-3361044348-30300820-1013` for a domain or local computer user or `S-1-5-18` for the LOCAL_SYSTEM user. When encoded in hex by the authentication protocol, this will typically result in a line like `AUTH EXTERNAL 532d312d352d3138` followed by \r\n.

DBUS_COOKIE_SHA1

DBUS_COOKIE_SHA1 is a D-Bus-specific SASL mechanism. Its reference implementation is part of the reference implementation of D-Bus.

This mechanism is designed to establish that a client has the ability to read a private file owned by the user being authenticated. If the client can prove that it has access to a secret cookie stored in this file, then the client is authenticated. Thus the security of DBUS_COOKIE_SHA1 depends on a secure home directory. This is the recommended authentication mechanism for platforms and configurations where EXTERNAL cannot be used.

Throughout this description, "hex encoding" must output the digits from a to f in lower-case; the digits A to F must not be used in the DBUS_COOKIE_SHA1 mechanism.

Authentication proceeds as follows:

- The client sends the username it would like to authenticate as, hex-encoded.
- The server sends the name of its "cookie context" (see below); a space character; the integer ID of the secret cookie the client must demonstrate knowledge of; a space character; then a randomly-generated challenge string, all of this hex-encoded into one, single string.
- The client locates the cookie and generates its own randomly-generated challenge string. The client then concatenates the server's decoded challenge, a ":" character, its own challenge, another ":" character, and the cookie. It computes the SHA-1 hash of this composite string as a hex digest. It concatenates the client's challenge string, a space character, and the SHA-1 hex digest, hex-encodes the result and sends it back to the server.
- The server generates the same concatenated string used by the client and computes its SHA-1 hash. It compares the hash with the hash received from the client; if the two hashes match, the client is authenticated.

Each server has a "cookie context," which is a name that identifies a set of cookies that apply to that server. A sample context might be "org_freedesktop_session_bus". Context names must be valid ASCII, nonzero length, and may not contain the characters slash ("/"), backslash ("\"), space (" "), newline ("\n"), carriage return ("\r"), tab ("\t"), or period (".") . There is a default context, "org_freedesktop_general" that's used by servers that do not specify otherwise.

Cookies are stored in a user's home directory, in the directory `~/.dbus-keyrings/`. This directory must not be readable or writable by other users. If it is, clients and servers must ignore it. The directory contains cookie files named after the cookie context.

A cookie file contains one cookie per line. Each line has three space-separated fields:

- The cookie ID number, which must be a non-negative integer and may not be used twice in the same file.
- The cookie's creation time, in UNIX seconds-since-the-epoch format.
- The cookie itself, a hex-encoded random block of bytes. The cookie may be of any length, though obviously security increases as the length increases.

Only server processes modify the cookie file. They must do so with this procedure: How to open and create a lockfile!

- Create a lockfile name by appending ".lock" to the name of the cookie file. The server should attempt to create this file using `O_CREAT | O_EXCL`. If file creation fails, the lock fails. Servers should retry for a reasonable period of time, then they may choose to delete an existing lock to keep users from having to manually delete a stale lock. [1]
- Once the lockfile has been created, the server loads the cookie file. It should then delete any cookies that are old (the timeout can be fairly short), or more than a reasonable time in the future (so that cookies never accidentally become permanent, if the clock was set far into the future at some point). The reference implementation deletes cookies that are more than 5 minutes into the future, or more than 7 minutes in the past. For interoperability, using the same arbitrary times in other implementations is suggested.
- If no sufficiently recent cookies remain, the server generates a new cookie. To avoid spurious authentication failures, cookies that are close to their deletion time should not be used for new authentication operations. For example, this avoids a client starting to use a cookie whose age is 6m59s, and having authentication subsequently fail because it takes 2 seconds, during which time the cookie's age became 7m01s, greater than 7 minutes, causing the server to delete it. The reference implementation generates a new cookie whenever the most recent cookie is older than 5 minutes, giving clients at least 2 minutes to finish authentication. For interoperability, using the same arbitrary time in other implementations is suggested.
- The pruned and possibly added-to cookie file must be resaved atomically (using a temporary file which is rename()'d).
- The lock must be dropped by deleting the lockfile.

Clients need not lock the file in order to load it, because servers are required to save the file atomically.

ANONYMOUS

The ANONYMOUS mechanism is defined in RFC 4505 "Anonymous Simple Authentication and Security Layer (SASL) Mechanism". It does not perform any authentication at all, and should not be accepted by message buses. However, it might sometimes be useful for non-message-bus uses of D-Bus.

Server Addresses

Server addresses consist of a transport name followed by a colon, and then an optional, comma-separated list of keys and values in the form key=value. Each value is escaped.

For example:

```
unix:path=/tmp/dbus-test
```

Which is the address to a unix socket with the path /tmp/dbus-test.

Value escaping is similar to URI escaping but simpler.

- The set of optionally-escaped bytes is: `[-0-9A-Za-z_/.\*]`. To escape, each *byte* (note, not character) which is not in the set of optionally-escaped bytes must be replaced with an ASCII percent (`%`) and the value of the byte in hex. The hex value must always be two digits, even if the first digit is zero. The optionally-escaped bytes may be escaped if desired.
- To unescape, append each byte in the value; if a byte is an ASCII percent (`%`) character then append the following hex value instead. It is an error if a `%` byte does not have two hex digits following. It is an error if a non-optionally-escaped byte is seen unescaped.

The set of optionally-escaped bytes is intended to preserve address readability and convenience.

A server may specify a key-value pair with the key `guid` and the value a hex-encoded 16-byte sequence. the section called “UUIDs” describes the format of the `guid` field. If present, this UUID may be used to distinguish one server address from another. A server should use a different UUID for each address it listens on. For example, if a message bus daemon offers both UNIX domain socket and TCP connections, but treats clients the same regardless of how they connect, those two connections are equivalent post-connection but should have distinct UUIDs to distinguish the kinds of connection.

The intent of the address UUID feature is to allow a client to avoid opening multiple identical connections to the same server, by allowing the client to check whether an address corresponds to an already-existing connection. Comparing two addresses is insufficient, because addresses can be recycled by distinct servers, and equivalent addresses may look different if simply compared as strings (for example, the host in a TCP address can be given as an IP address or as a hostname).

Note that the address key is `guid` even though the rest of the API and documentation says "UUID," for historical reasons.

[FIXME clarify if attempting to connect to each is a requirement or just a suggestion] When connecting to a server, multiple server addresses can be separated by a semi-colon. The library will then try to connect to the first address and if that fails, it'll try to connect to the next one specified, and so forth. For example

```
unix:path=/tmp/dbus-test;unix:path=/tmp/dbus-test2
```

Some addresses are *connectable*. A connectable address is one containing enough information for a client to connect to it. For instance, `tcp:host=127.0.0.1,port=4242` is a connectable address. It is not necessarily possible to listen on every connectable address: for instance, it is not possible to listen on a `unixexec:` address.

Some addresses are *listenable*. A listenable address is one containing enough information for a server to listen on it, producing a connectable address (which may differ from the original address). Many listenable addresses are not connectable: for instance, `tcp:host=127.0.0.1` is listenable, but not connectable (because it does not specify a port number).

Listening on an address that is not connectable will result in a connectable address that is not the same as the listenable address. For instance, listening on `tcp:host=127.0.0.1` might result in the connectable address `tcp:host=127.0.0.1,port=30958`, listening on `unix:tmpdir=/tmp` might result in the connectable address `unix:abstract=/tmp/dbus-U8OSdmf7`, or listening on `unix:runtime=yes` might result in the connectable address `unix:path=/run/user/1234/bus`.

Transports

[FIXME we need to specify in detail each transport and its possible arguments] Current transports include: unix domain sockets (including abstract namespace on linux), launchd, systemd, TCP/IP, an executed subprocess and a debug/testing transport using in-process pipes. Future possible transports include one that tunnels over X11 protocol.

Unix Domain Sockets

Unix domain sockets can be either paths in the file system or on Linux kernels, they can be abstract which are similar to paths but do not show up in the file system.

When a socket is opened by the D-Bus library it truncates the path name right before the first trailing Nul byte. This is true for both normal paths and abstract paths. Note that this is a departure from previous versions of D-Bus that would create sockets with a fixed length path name. Names which were shorter than the fixed length would be padded by Nul bytes.

Unix domain sockets are not available on Windows. On all other platforms, they are the recommended transport for D-Bus, either used alone or in conjunction with systemd or launchd addresses.

Unix addresses that specify `path` or `abstract` are both listenable and connectable. Unix addresses that specify `tmpdir` or `dir` are only listenable: the corresponding connectable address will specify either `path` or `abstract`. Similarly, Unix addresses that specify `runtime` are only listenable, and the corresponding connectable address will specify `path`.

Server Address Format

Unix domain socket addresses are identified by the "unix:" prefix and support the following key/value pairs:

Name	Values	Description
path	(path)	Path of the unix domain socket.
dir	(path)	Directory in which a socket file with a random file name starting with 'dbus-' will be created by the server. This key can only be used in server addresses, not in client addresses; the resulting client address will have the "path" key instead. be set.
tmpdir	(path)	The same as "dir", except that on platforms with abstract sockets, the server may attempt to create an abstract socket whose name starts with this directory instead of a path-based socket. This key can only be used in server addresses, not in client addresses; the resulting client address will have the "abstract" or "path" key instead.
abstract	(string)	Unique string in the abstract namespace, often syntactically resembling a path but unconnected to the filesystem namespace. This key is only supported on platforms with abstract Unix sockets, of which Linux is the only known example.
runtime	`yes`	If given, This key can only be used in server addresses, not in client addresses. If set, its value must be `yes`. This is typically used in an address string like `unix:runtime=yes;unix:tmpdir=/tmp` so that there can be a fallback if `XDG_RUNTIME_DIR` is not set.

Exactly one of the keys `path`, `abstract`, `runtime`, `dir` or `tmpdir` must be provided.

launchd

launchd is an open-source server management system that replaces init, inetd and cron on Apple Mac OS X versions 10.4 and above. It provides a common session bus address for each user and deprecates the X11-enabled D-Bus launcher on OSX.

launchd allocates a socket and provides it with the unix path through the DBUS_LAUNCHD_SESSION_BUS_SOCKET variable in launchd's environment. Every process spawned by launchd (or dbus-daemon, if it was started by launchd) can access it through its environment. Other processes can query for the launchd socket by executing: $ launchctl getenv DBUS_LAUNCHD_SESSION_BUS_SOCKET This is normally done by the D-Bus client library so doesn't have to be done manually.

launchd is not available on Microsoft Windows.

launchd addresses are listenable and connectable.

Server Address Format

launchd addresses are identified by the "launchd:" prefix and support the following key/value pairs:

Name	Values	Description
env	(environment variable)	path of the unix domain socket for the launchd created dbus-daemon.

The `env` key is required.

systemd

systemd is an open-source server management system that replaces init and inetd on newer Linux systems. It supports socket activation. The D-Bus systemd transport is used to acquire socket activation file descriptors from systemd and use them as D-Bus transport when the current process is spawned by socket activation from it.

The systemd transport accepts only one or more Unix domain or TCP streams sockets passed in via socket activation. Using Unix domain sockets is strongly recommended.

The systemd transport is not available on non-Linux operating systems.

The systemd transport defines no parameter keys.

systemd addresses are listenable, but not connectable. The corresponding connectable address is the `unix` or `tcp` address of the socket.

TCP Sockets

The tcp transport provides TCP/IP based connections between clients located on the same or different hosts.

Similar to remote X11, the TCP transport has no integrity or confidentiality protection, so it should normally only be used across the local loopback interface, for example using an address like `tcp:host=127.0.0.1` or `tcp:host=localhost`. In particular, configuring the well-known system bus or the well-known session bus to listen on a non-loopback TCP address is insecure.

On Windows and most Unix platforms, the TCP stack is unable to transfer credentials over a TCP connection, so the EXTERNAL authentication mechanism does not normally work for this transport (although the reference implementation of D-Bus is able to identify loopback TCPv4 connections on Windows by their port number, partially enabling the EXTERNAL mechanism). The DBUS_COOKIE_SHA1 mechanism is normally used instead.

Developers are sometimes tempted to use remote TCP as a debugging tool. However, if this functionality is left enabled in finished products, the result will be dangerously insecure. Instead of using remote TCP, developers should relay connections via Secure Shell or a similar protocol.

Remote TCP connections were historically sometimes used to share a single session bus between login sessions of the same user on different machines within a trusted local area network, in conjunction with unencrypted remote X11, a NFS-shared home directory and NIS (YP) authentication. This is insecure against an attacker on the same LAN and should be considered strongly deprecated; more specifically, it is insecure in the same ways and for the same reasons as unencrypted remote X11 and NFSv2/NFSv3. The D-Bus maintainers recommend using a separate session bus per (user, machine) pair, only accessible from within that machine.

All `tcp` addresses are listenable. `tcp` addresses in which both `host` and `port` are specified, and `port` is non-zero, are also connectable.

Server Address Format

TCP/IP socket addresses are identified by the "tcp:" prefix and support the following key/value pairs:

Name	Values	Description
host	(string)	DNS name or IP address

Name	Values	Description
bind	(string)	Used in a listenable address to configure the interface on which the server will listen: either the IP address of one of the local machine's interfaces (most commonly `127.0.0.1`), or a DNS name that resolves to one of those IP addresses, or '*' to listen on all interfaces simultaneously. If not specified, the default is the same value as "host".
port	(number)	The tcp port the server will open. A zero value let the server choose a free port provided from the underlaying operating system. libdbus is able to retrieve the real used port from the server.
family	(string)	If set, provide the type of socket family either "ipv4" or "ipv6". If unset, the family is unspecified.

Nonce-authenticated TCP Sockets

The nonce-tcp transport provides a modified TCP transport using a simple authentication mechanism, to ensure that only clients with read access to a certain location in the filesystem can connect to the server. The server writes a secret, the nonce, to a file and an incoming client connection is only accepted if the client sends the nonce right after the connect. The nonce mechanism requires no setup and is orthogonal to the higher-level authentication mechanisms described in the Authentication section.

The nonce-tcp transport is conceptually similar to a combination of the DBUS_COOKIE_SHA1 authentication mechanism and the tcp transport, and appears to have originally been implemented as a result of a misunderstanding of the SASL authentication mechanisms.

Like the ordinary tcp transport, the nonce-tcp transport has no integrity or confidentiality protection, so it should normally only be used across the local loopback interface, for example using an address like `tcp:host=127.0.0.1` or `tcp:host=localhost`. Other uses are insecure. See the section called “TCP Sockets” for more information on situations where these transports have been used, and alternatives to these transports.

Implementations of D-Bus on Windows operating systems normally use a nonce-tcp transport via the local loopback interface. This is because the unix transport, which would otherwise be recommended, is not available on these operating systems.

On start, the server generates a random 16 byte nonce and writes it to a file in the user's temporary directory. The nonce file location is published as part of the server's D-Bus address using the "noncefile" key-value pair. After an accept, the server reads 16 bytes from the socket. If the read bytes do not match the nonce stored in the nonce file, the server MUST immediately drop the connection. If the nonce match the received byte sequence, the client is accepted and the transport behaves like an ordinary tcp transport.

After a successful connect to the server socket, the client MUST read the nonce from the file published by the server via the noncefile= key-value pair and send it over the socket. After that, the transport behaves like an ordinary tcp transport.

All nonce-tcp addresses are listenable. nonce-tcp addresses in which `host`, `port` and `noncefile` are all specified, and `port` is nonzero, are also connectable.

Server Address Format

Nonce TCP/IP socket addresses uses the "nonce-tcp:" prefix and support the following key/value pairs:

Name	Values	Description
host	(string)	DNS name or IP address
bind	(string)	The same as for tcp: addresses
port	(number)	The tcp port the server will open. A zero value let the server choose a free port provided from the underlaying operating system. libdbus is able to retrieve the real used port from the server.
family	(string)	If set, provide the type of socket family either "ipv4" or "ipv6". If unset, the family is unspecified.
noncefile	(path)	File location containing the secret. This is only meaningful in connectable addresses: a listening D-Bus server that offers this transport will always create a new nonce file.

Executed Subprocesses on Unix

This transport forks off a process and connects its standard input and standard output with an anonymous Unix domain socket. This socket is then used for communication by the transport. This transport may be used to use out-of-process forwarder programs as basis for the D-Bus protocol.

The forked process will inherit the standard error output and process group from the parent process.

Executed subprocesses are not available on Windows.

`unixexec` addresses are connectable, but are not listenable.

Server Address Format

Executed subprocess addresses are identified by the "unixexec:" prefix and support the following key/value pairs:

Name	Values	Description
path	(path)	Path of the binary to execute, either an absolute path or a binary name that is searched for in the default search path of the OS. This corresponds to the first argument of execlp(). This key is mandatory.
argv0	(string)	The program name to use when executing the binary. If omitted the same value as specified for path= will be used. This corresponds to the second argument of execlp().
argv1, argv2, ...	(string)	Arguments to pass to the binary. This corresponds to the third and later arguments of execlp(). If a specific argvX is not specified no further argvY for Y > X are taken into account.

Meta Transports

Meta transports are a kind of transport with special enhancements or behavior. Currently available meta transports include: autolaunch

Autolaunch

The autolaunch transport provides a way for dbus clients to autodetect a running dbus session bus and to autolaunch a session bus if not present.

On Unix, `autolaunch` addresses are connectable, but not listenable.

On Windows, `autolaunch` addresses are both connectable and listenable.

Server Address Format

Autolaunch addresses uses the "autolaunch:" prefix and support the following key/value pairs:

Name	Values	Description

scope	(string)	scope of autolaunch (Windows only)
		• "*install-path" - limit session bus to dbus installation path. The dbus installation path is determined from the location of the shared dbus library. If the library is located in a 'bin' subdirectory the installation root is the directory above, otherwise the directory where the library lives is taken as installation root.
		`<install-root>/bin/[lib]dbus-1.dll`
		`<install-root>/[lib]dbus-1.dll`
		• "*user" - limit session bus to the recent user.
		• other values - specify dedicated session bus like "release", "debug" or other

Windows implementation

On start, the server opens a platform specific transport, creates a mutex and a shared memory section containing the related session bus address. This mutex will be inspected by the dbus client library to detect a running dbus session bus. The access to the mutex and the shared memory section are protected by global locks.

In the recent implementation the autolaunch transport uses a tcp transport on localhost with a port choosen from the operating system. This detail may change in the future.

Disclaimer: The recent implementation is in an early state and may not work in all cirumstances and/or may have security issues. Because of this the implementation is not documentated yet.

UUIDs

A working D-Bus implementation uses universally-unique IDs in two places. First, each server address has a UUID identifying the address, as described in the section called “Server Addresses”. Second, each operating system kernel instance running a D-Bus client or server has a UUID identifying that kernel, retrieved by invoking the method org.freedesktop.DBus.Peer.GetMachineId() (see the section called “`org.freedesktop.DBus.Peer`”).

The term "UUID" in this document is intended literally, i.e. an identifier that is universally unique. It is not intended to refer to RFC4122, and in fact the D-Bus UUID is not compatible with that RFC.

16 bytes

The UUID must contain 128 bits of data and be hex-encoded. The hex-encoded string may not contain hyphens or other non-hex-digit characters, and it must be exactly 32 characters long. To generate a UUID, the current reference implementation concatenates 96 bits of random data followed by the 32-bit time in seconds since the UNIX epoch (in big endian byte order).

It would also be acceptable and probably better to simply generate 128 bits of random data, as long as the random number generator is of high quality. The timestamp could conceivably help if the random bits are not very random. With a quality random number generator, collisions are extremely unlikely even with only 96 bits, so it's somewhat academic.

Implementations should, however, stick to random data for the first 96 bits of the UUID.

Standard Interfaces

See the section called “Notation in this document” for details on the notation used in this section. There are some standard interfaces that may be useful across various D-Bus applications.

`org.freedesktop.DBus.Peer`

The `org.freedesktop.DBus.Peer` interface has two methods:

```
org.freedesktop.DBus.Peer.Ping ()
org.freedesktop.DBus.Peer.GetMachineId (out STRING machine_uuid)
```

On receipt of the `METHOD_CALL` message `org.freedesktop.DBus.Peer.Ping`, an application should do nothing other than reply with a `METHOD_RETURN` as usual. It does not matter which object path a ping is sent to. The reference implementation handles this method automatically.

On receipt of the `METHOD_CALL` message `org.freedesktop.DBus.Peer.GetMachineId`, an application should reply with a `METHOD_RETURN` containing a hex-encoded UUID representing the identity of the machine the process is running on. This UUID must be the same for all processes on a single system at least until that system next reboots. It should be the same across reboots if possible, but this is not always possible to implement and is not guaranteed. It does not matter which object path a GetMachineId is sent to. The reference implementation handles this method automatically.

On Unix, implementations should try to read the machine ID from `/var/lib/dbus/machine-id` and `/etc/machine-id`. The latter is defined by systemd, but systems not using systemd may provide an equivalent file. If both exist, they are expected to have the same contents, and if they differ, the spec does not define which takes precedence (the reference implementation prefers `/var/lib/dbus/machine-id`, but sd-bus does not).

On Windows, the hardware profile GUID is used as the machine ID, with the punctuation removed. This can be obtained with the `GetCurrentHwProfile` function.

The UUID is intended to be per-instance-of-the-operating-system, so may represent a virtual machine running on a hypervisor, rather than a physical machine. Basically if two processes see the same UUID, they should also see the same shared memory, UNIX domain sockets, process IDs, and other features that require a running OS kernel in common between the processes.

The UUID is often used where other programs might use a hostname. Hostnames can change without rebooting, however, or just be "localhost" - so the UUID is more robust.

the section called “UUIDs” explains the format of the UUID.

`org.freedesktop.DBus.Introspectable`

This interface has one method:

```
org.freedesktop.DBus.Introspectable.Introspect (out STRING xml_data)
```

Objects instances may implement `Introspect` which returns an XML description of the object, including its interfaces (with signals and methods), objects below it in the object path tree, and its properties.

the section called “Introspection Data Format” describes the format of this XML string.

`org.freedesktop.DBus.Properties`

Many native APIs will have a concept of object *properties* or *attributes*. These can be exposed via the `org.freedesktop.DBus.Properties` interface.

```
org.freedesktop.DBus.Properties.Get (in STRING interface_name,
                                     in STRING property_name,
                                     out VARIANT value);
org.freedesktop.DBus.Properties.Set (in STRING interface_name,
                                     in STRING property_name,
                                     in VARIANT value);
org.freedesktop.DBus.Properties.GetAll (in STRING interface_name,
                                        out ARRAY of DICT_ENTRY<STRING,VARIANT> props);
```

It is conventional to give D-Bus properties names consisting of capitalized words without punctuation ("CamelCase"), like member names. For instance, the GObject property `connection-status` or the Qt property `connectionStatus` could be represented on D-Bus as `ConnectionStatus`.

Strictly speaking, D-Bus property names are not required to follow the same naming restrictions as member names, but D-Bus property names that would not be valid member names (in particular, GObject-style dash-separated property names) can cause interoperability problems and should be avoided.

The available properties and whether they are writable can be determined by calling `org.freedesktop.DBus.Introspectable.Introspect`, see the section called “`org.freedesktop.DBus.Introspectable`”.

An empty string may be provided for the interface name; in this case, if there are multiple properties on an object with the same name, the results are undefined (picking one by according to an arbitrary deterministic rule, or returning an error, are the reasonable possibilities).

If `org.freedesktop.DBus.Properties.GetAll` is called with a valid interface name which contains no properties, an empty array should be returned. If it is called with a valid interface name for which some properties are not accessible to the caller (for example, due to per-property access control implemented in the service), those properties should be silently omitted from the result array. If `org.freedesktop.DBus.Properties.Get` is called for any such properties, an appropriate access control error should be returned.

If one or more properties change on an object, the `org.freedesktop.DBus.Properties.PropertiesChanged` signal may be emitted (this signal was added in 0.14):

```
org.freedesktop.DBus.Properties.PropertiesChanged (STRING interface_name,
                                                   ARRAY of DICT_ENTRY<STRING,VARIANT> changed_properties,
                                                   ARRAY<STRING> invalidated_properties);
```

where `changed_properties` is a dictionary containing the changed properties with the new values and `invalidated_properties` is an array of properties that changed but the value is not conveyed.

Whether the `PropertiesChanged` signal is supported can be determined by calling `org.freedesktop.DBus.Introspectable.Introspect`. Note that the signal may be supported for an object but it may differ how whether and how it is used on a per-property basis (for e.g. performance or security reasons). Each property (or the parent interface) must be annotated with the `org.freedesktop.DBus.Property.EmitsChangedSignal` annotation to convey this (usually the default value `true` is sufficient meaning that the annotation does not need to be used). See the section called “Introspection Data Format” for details on this annotation.

`org.freedesktop.DBus.ObjectManager`

An API can optionally make use of this interface for one or more sub-trees of objects. The root of each sub-tree implements this interface so other applications can get all objects, interfaces and properties in a single method call. It is appropriate to use this interface if users of the tree of objects are expected to be interested in all interfaces of all objects in the tree; a more granular API should be used if users of the objects are expected to be interested in a small subset of the objects, a small subset of their interfaces, or both.

The method that applications can use to get all objects and properties is `GetManagedObjects`:

```
org.freedesktop.DBus.ObjectManager.GetManagedObjects (out ARRAY of DICT_ENTRY<OBJPATH,ARRAY of DICT_ENTRY<STRING,ARRAY of DICT_ENTR
```

The return value of this method is a dict whose keys are object paths. All returned object paths are children of the object path implementing this interface, i.e. their object paths start with the ObjectManager's object path plus '/'.

Each value is a dict whose keys are interfaces names. Each value in this inner dict is the same dict that would be returned by the org.freedesktop.DBus.Properties.GetAll() method for that combination of object path and interface. If an interface has no properties, the empty dict is returned.

Changes are emitted using the following two signals:

```
org.freedesktop.DBus.ObjectManager.InterfacesAdded (OBJPATH object_path,
                                                    ARRAY of DICT_ENTRY<STRING,ARRAY of DICT_ENTRY<STRING,VARIANT>> interfaces_and_
org.freedesktop.DBus.ObjectManager.InterfacesRemoved (OBJPATH object_path,
                                                      ARRAY<STRING> interfaces);
```

The `InterfacesAdded` signal is emitted when either a new object is added or when an existing object gains one or more interfaces. The `InterfacesRemoved` signal is emitted whenever an object is removed or it loses one or more interfaces. The second parameter of the `InterfacesAdded` signal contains a dict with the interfaces and properties (if any) that have been added to the given object path. Similarly, the second parameter of the `InterfacesRemoved` signal contains an array of the interfaces that were removed. Note that changes on properties on existing interfaces are not reported using this interface - an application should also monitor the existing PropertiesChanged signal on each object.

Applications SHOULD NOT export objects that are children of an object (directly or otherwise) implementing this interface but which are not returned in the reply from the `GetManagedObjects()` method of this interface on the given object.

The intent of the `ObjectManager` interface is to make it easy to write a robust client implementation. The trivial client implementation only needs to make two method calls:

```
org.freedesktop.DBus.AddMatch (bus_proxy,
                               "type='signal',sender='org.example.App2',path_namespace='/org/example/App2'");
objects = org.freedesktop.DBus.ObjectManager.GetManagedObjects (app_proxy);
```

on the message bus and the remote application's `ObjectManager`, respectively. Whenever a new remote object is created (or an existing object gains a new interface), the `InterfacesAdded` signal is emitted, and since this signal contains all properties for the interfaces, no calls to the `org.freedesktop.Properties` interface on the remote object are needed. Additionally, since the initial `AddMatch()` rule already includes signal messages from the newly created child object, no new `AddMatch()` call is needed.

The `org.freedesktop.DBus.ObjectManager` interface was added in version 0.17 of the D-Bus specification.

Introspection Data Format

As described in the section called “`org.freedesktop.DBus.Introspectable`”, objects may be introspected at runtime, returning an XML string that describes the object. The same XML format may be used in other contexts as well, for example as an "IDL" for generating static language bindings.

Here is an example of introspection data:

```
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node name="/com/example/sample_object0">
  <interface name="com.example.SampleInterface0">
    <method name="Frobate">
      <arg name="foo" type="i" direction="in"/>
      <arg name="bar" type="s" direction="out"/>
      <arg name="baz" type="a{us}" direction="out"/>
      <annotation name="org.freedesktop.DBus.Deprecated" value="true"/>
    </method>
    <method name="Bazify">
      <arg name="bar" type="(iiu)" direction="in"/>
      <arg name="bar" type="v" direction="out"/>
    </method>
    <method name="Mogrify">
      <arg name="bar" type="(iiav)" direction="in"/>
    </method>
    <signal name="Changed">
      <arg name="new_value" type="b"/>
    </signal>
    <property name="Bar" type="y" access="readwrite"/>
  </interface>
  <node name="child_of_sample_object"/>
  <node name="another_child_of_sample_object"/>
</node>
```

A more formal DTD and spec needs writing, but here are some quick notes.

- Only the root <node> element can omit the node name, as it's known to be the object that was introspected. If the root <node> does have a name attribute, it must be an absolute object path. If child <node> have object paths, they must be relative.
- If a child <node> has any sub-elements, then they must represent a complete introspection of the child. If a child <node> is empty, then it may or may not have sub-elements; the child must be introspected in order to find out. The intent is that if an object knows that its children are "fast" to introspect it can go ahead and return their information, but otherwise it can omit it.
- The direction element on <arg> may be omitted, in which case it defaults to "in" for method calls and "out" for signals. Signals only allow "out" so while direction may be specified, it's pointless.
- The possible directions are "in" and "out", unlike CORBA there is no "inout"
- The possible property access flags are "readwrite", "read", and "write"
- Multiple interfaces can of course be listed for one <node>.
- The "name" attribute on arguments is optional.

Method, interface, property, signal, and argument elements may have "annotations", which are generic key/value pairs of metadata. They are similar conceptually to Java's annotations and C# attributes. Well-known annotations:

Name	Values (separated by ,)	Description
org.freedesktop.DBus.Deprecated	true,false	Whether or not the entity is deprecated; defaults to false
org.freedesktop.DBus.GLib.CSymbol	(string)	The C symbol; may be used for methods and interfaces
org.freedesktop.DBus.Method.NoReply	true,false	If set, don't expect a reply to the method call; defaults to false.
org.freedesktop.DBus.Property.EmitsChangedSignal	true,invalidates,const,false	If set to `false`, the `org.freedesktop.DBus.Properties.PropertiesChanged` signal, see the section called "`org.freedesktop.DBus.Properties`" is not guaranteed to be emitted if the property changes. If set to `const` the property never changes value during the lifetime of the object it belongs to, and hence the signal is never emitted for it. If set to `invalidates` the signal is emitted but the value is not included in the signal. If set to `true` the signal is emitted with the value included. The value for the annotation defaults to `true` if the enclosing interface element does not specify the annotation. Otherwise it defaults to the value specified in the enclosing interface element. This annotation is intended to be used by code generators to implement client-side caching of property values. For all properties for which the annotation is set to `const`, `invalidates` or `true` the client may unconditionally cache the values as the properties don't change or notifications are generated for them if they do.

Message Bus Specification

Message Bus Overview

The message bus accepts connections from one or more applications. Once connected, applications can exchange messages with other applications that are also connected to the bus.

In order to route messages among connections, the message bus keeps a mapping from names to connections. Each connection has one unique-for-the-lifetime-of-the-bus name automatically assigned. Applications may request additional names for a connection. Additional names are usually "well-known names" such as "com.example.TextEditor1". When a name is bound to a connection, that connection is said to *own* the name.

The bus itself owns a special name, `org.freedesktop.DBus`, with an object located at `/org/freedesktop/DBus` that implements the `org.freedesktop.DBus` interface. This service allows applications to make administrative requests of the bus itself. For example, applications can ask the bus to assign a name to a connection.

Each name may have *queued owners*. When an application requests a name for a connection and the name is already in use, the bus will optionally add the connection to a queue waiting for the name. If the current owner of the name disconnects or releases the name, the next connection in the queue will become the new owner.

This feature causes the right thing to happen if you start two text editors for example; the first one may request "com.example.TextEditor1", and the second will be queued as a possible owner of that name. When the first exits, the second will take over.

Applications may send *unicast messages* to a specific recipient or to the message bus itself, or *broadcast messages* to all interested recipients. See the section called "Message Bus Message Routing" for details.

Message Bus Names

Each connection has at least one name, assigned at connection time and returned in response to the `org.freedesktop.DBus.Hello` method call. This automatically-assigned name is called the connection's *unique name*. Unique names are never reused for two different connections to the same bus.

Ownership of a unique name is a prerequisite for interaction with the message bus. It logically follows that the unique name is always the first name that an application comes to own, and the last one that it loses ownership of.

Unique connection names must begin with the character ':' (ASCII colon character); bus names that are not unique names must not begin with this character. (The bus must reject any attempt by an application to manually request a name beginning with ':'.) This restriction categorically prevents "spoofing"; messages sent to a unique name will always go to the expected connection.

When a connection is closed, all the names that it owns are deleted (or transferred to the next connection in the queue if any).

A connection can request additional names to be associated with it using the `org.freedesktop.DBus.RequestName` message. the section called "Bus names" describes the format of a valid name. These names can be released again using the `org.freedesktop.DBus.ReleaseName` message.

Message Bus Message Routing

Messages may have a `DESTINATION` field (see the section called "Header Fields"), resulting in a *unicast message*. If the `DESTINATION` field is present, it specifies a message recipient by name. Method calls and replies normally specify this field. The message bus must send messages (of any type) with the `DESTINATION` field set to the specified recipient, regardless of whether the recipient has set up a match rule matching the message.

When the message bus receives a signal, if the `DESTINATION` field is absent, it is considered to be a *broadcast signal*, and is sent to all applications with *message matching rules* that match the message. Most signal messages are broadcasts, and no other message types currently defined in this specification may be broadcast.

Unicast signal messages (those with a `DESTINATION` field) are not commonly used, but they are treated like any unicast message: they are delivered to the specified receipient, regardless of its match rules. One use for unicast signals is to avoid a race condition in which a signal is emitted before the intended recipient can call the section called "`org.freedesktop.DBus.AddMatch`" to receive that signal: if the signal is sent directly to that recipient using a unicast message, it does not need to add a match rule at all, and there is no race condition. Another use for unicast signals, on message buses whose security policy prevents eavesdropping, is to send sensitive information which should only be visible to one recipient.

When the message bus receives a method call, if the `DESTINATION` field is absent, the call is taken to be a standard one-to-one message and interpreted by the message bus itself. For example, sending an `org.freedesktop.DBus.Peer.Ping` message with no `DESTINATION` will cause the message bus itself to reply to the ping immediately; the message bus will not make this message visible to other applications.

Continuing the `org.freedesktop.DBus.Peer.Ping` example, if the ping message were sent with a `DESTINATION` name of `com.yoyodyne.Screensaver`, then the ping would be forwarded, and the Yoyodyne Corporation screensaver application would be expected to reply to the ping.

Message bus implementations may impose a security policy which prevents certain messages from being sent or received. When a method call message cannot be sent or received due to a security policy, the message bus should send an error reply, unless the original message had the `NO_REPLY` flag.

Eavesdropping

Receiving a unicast message whose `DESTINATION` indicates a different recipient is called *eavesdropping*. On a message bus which acts as a security boundary (like the standard system bus), the security policy should usually prevent eavesdropping, since unicast messages are normally kept private and may contain security-sensitive information.

Eavesdropping interacts poorly with buses with non-trivial access control restrictions, and is deprecated. The `BecomeMonitor` method (see the section called “`org.freedesktop.DBus.Monitoring.BecomeMonitor`”) provides a preferable way to monitor buses.

Eavesdropping is mainly useful for debugging tools, such as the `dbus-monitor` tool in the reference implementation of D-Bus. Tools which eavesdrop on the message bus should be careful to avoid sending a reply or error in response to messages intended for a different client.

Clients may attempt to eavesdrop by adding match rules (see the section called “Match Rules”) containing the `eavesdrop='true'` match. For compatibility with older message bus implementations, if adding such a match rule results in an error reply, the client may fall back to adding the same rule with the `eavesdrop` match omitted.

Match Rules

An important part of the message bus routing protocol is match rules. Match rules describe the messages that should be sent to a client, based on the contents of the message. Broadcast signals are only sent to clients which have a suitable match rule: this avoids waking up client processes to deal with signals that are not relevant to that client.

Messages that list a client as their `DESTINATION` do not need to match the client's match rules, and are sent to that client regardless. As a result, match rules are mainly used to receive a subset of broadcast signals.

Match rules can also be used for eavesdropping (see the section called “Eavesdropping”), if the security policy of the message bus allows it, but this usage is deprecated in favour of the `BecomeMonitor` method (see the section called “`org.freedesktop.DBus.Monitoring.BecomeMonitor`”).

Match rules are added using the AddMatch bus method (see the section called “`org.freedesktop.DBus.AddMatch`”). Rules are specified as a string of comma separated key/value pairs. Excluding a key from the rule indicates a wildcard match. For instance excluding the the member from a match rule but adding a sender would let all messages from that sender through. An example of a complete rule would be "type='signal',sender='org.freedesktop.DBus',interface='org.freedesktop.DBus',member='Foo',path='/bar/foo',destination=':452345.34',arg2='bar'"

Within single quotes (ASCII apostrophe, U+0027), a backslash (U+005C) represents itself, and an apostrophe ends the quoted section. Outside single quotes, \' (backslash, apostrophe) represents an apostrophe, and any backslash not followed by an apostrophe represents itself. For instance, the match rules `arg0=''\'',arg1='\',arg2=',',arg3='\\'` and `arg0=\',arg1=\,arg2=',',arg3=\\` both match messages where the arguments are a 1-character string containing an apostrophe, a 1-character string containing a backslash, a 1-character string containing a comma, and a 2-character string containing two backslashes[2].

The following table describes the keys that can be used to create a match rule.

Key	Possible Values	Description
`type`	'signal', 'method_call', 'method_return', 'error'	Match on the message type. An example of a type match is type='signal'
`sender`	A bus or unique name (see Bus Name and Unique Connection Name respectively)	Match messages sent by a particular sender. An example of a sender match is sender='org.freedesktop.Hal'
`interface`	An interface name (see the section called “Interface names”)	Match messages sent over or to a particular interface. An example of an interface match is interface='org.freedesktop.Hal.Manager'. If a message omits the interface header, it must not match any rule that specifies this key.
`member`	Any valid method or signal name	Matches messages which have the give method or signal name. An example of a member match is member='NameOwnerChanged'
`path`	An object path (see the section called “Valid Object Paths”)	Matches messages which are sent from or to the given object. An example of a path match is path='/org/freedesktop/Hal/Manager'

Key	Possible Values	Description
`path_namespace`	An object path	Matches messages which are sent from or to an object for which the object path is either the given value, or that value followed by one or more path components. For example, `path_namespace='/com/example/foo'` would match signals sent by `/com/example/foo` or by `/com/example/foo/bar`, but not by `/com/example/foobar`. Using both `path` and `path_namespace` in the same match rule is not allowed. *This match key was added in version 0.16 of the D-Bus specification and implemented by the bus daemon in dbus 1.5.0 and later.*
`destination`	A unique name (see Unique Connection Name)	Matches messages which are being sent to the given unique name. An example of a destination match is destination=':1.0'
`arg[0, 1, 2, 3, ...]`	Any string	Arg matches are special and are used for further restricting the match based on the arguments in the body of a message. Only arguments of type STRING can be matched in this way. An example of an argument match would be arg3='Foo'. Only argument indexes from 0 to 63 should be accepted.
`arg[0, 1, 2, 3, ...]path`	Any string	Argument path matches provide a specialised form of wildcard matching for path-like namespaces. They can match arguments whose type is either STRING or OBJECT_PATH. As with normal argument matches, if the argument is exactly equal to the string given in the match rule then the rule is satisfied. Additionally, there is also a match when either the string given in the match rule or the appropriate message argument ends with '/' and is a prefix of the other. An example argument path match is arg0path='/aa/bb/'. This would match messages with first arguments of '/', '/aa/', '/aa/bb/', '/aa/bb/cc/' and '/aa/bb/cc'. It would not match messages with first arguments of '/aa/b', '/aa' or even '/aa/bb'. This is intended for monitoring “directories” in file system-like hierarchies, as used in the *dconf* configuration system. An application interested in all nodes in a particular hierarchy would monitor `arg0path='/ca/example/foo/'`. Then the service could emit a signal with zeroth argument `"/ca/example/foo/bar"` to represent a modification to the “bar” property, or a signal with zeroth argument `"/ca/example/"` to represent atomic modification of many properties within that directory, and the interested application would be notified in both cases. *This match key was added in version 0.12 of the D-Bus specification, implemented for STRING arguments by the bus daemon in dbus 1.2.0 and later, and implemented for OBJECT_PATH arguments in dbus 1.5.0 and later.*
`arg0namespace`	Like a bus name, except that the string is not required to contain a '.' (period)	Match messages whose first argument is of type STRING, and is a bus name or interface name within the specified namespace. This is primarily intended for watching name owner changes for a group of related bus names, rather than for a single name or all name changes. Because every valid interface name is also a valid bus name, this can also be used for messages whose first argument is an interface name. For example, the match rule `member='NameOwnerChanged',arg0namespace='com.example.backend1'` matches name owner changes for bus names such as `com.example.backend1.foo`, `com.example.backend1.foo.bar`, and `com.example.backend1` itself. See also the section called “`org.freedesktop.DBus.NameOwnerChanged`”. *This match key was added in version 0.16 of the D-Bus specification and implemented by the bus daemon in dbus 1.5.0 and later.*
`eavesdrop`	`'true'`, `'false'`	Since D-Bus 1.5.6, match rules do not match messages which have a `DESTINATION` field unless the match rule specifically requests this (see the section called “Eavesdropping”) by specifying `eavesdrop='true'` in the match rule. `eavesdrop='false'` restores the default behaviour. Messages are delivered to their `DESTINATION` regardless of match rules, so this match does not affect normal delivery of unicast messages. In older versions of D-Bus, this match was not allowed in match rules, and all match rules behaved as if `eavesdrop='true'` had been used. Use of `eavesdrop='true'` is deprecated. Monitors should prefer to use the `BecomeMonitor` method (see the section called “`org.freedesktop.DBus.Monitoring.BecomeMonitor`”), which was introduced in version 0.26 of the D-Bus specification and version 1.9.10 of the reference dbus-daemon. Message bus implementations may restrict match rules with `eavesdrop='true'` so that they can only be added by privileged connections. *This match key was added in version 0.18 of the D-Bus specification and implemented by the bus daemon in dbus 1.5.6 and later.*

Message Bus Starting Services (Activation)

The message bus can start applications on behalf of other applications. This is referred to as *service activation* or *activation*. An application that can be started in this way is called a *service* or an *activatable service*.

Starting a service should be read as synonymous with service activation.

In D-Bus, service activation is normally done by *auto-starting*. In auto-starting, applications send a message to a particular well-known name, such as `com.example.TextEditor1`, without specifying the `NO_AUTO_START` flag in the message header. If no application on the bus owns the requested name, but the bus daemon does know how to start an activatable service for that name, then the bus daemon will start that service, wait for it to request that name, and deliver the message to it.

It is also possible for applications to send an explicit request to start a service: this is another form of activation, distinct from auto-starting. See the section called “`org.freedesktop.DBus.StartServiceByName`” for details.

In either case, this implies a contract documented along with the name `com.example.TextEditor1` for which object the owner of that name will provide, and what interfaces those objects will have.

To find an executable corresponding to a particular name, the bus daemon looks for *service description files*. Service description files define a mapping from names to executables. Different kinds of message bus will look for these files in different places, see the section called “Well-known Message Bus Instances”.

Service description files have the ".service" file extension. The message bus will only load service description files ending with .service; all other files will be ignored. The file format is similar to that of desktop entries. All service description files must be in UTF-8 encoding. To ensure that there will be no name collisions, service files must be namespaced using the same mechanism as messages and service names.

On the well-known system bus, the name of a service description file must be its well-known name plus `.service`, for instance `com.example.ConfigurationDatabase1.service`.

On the well-known session bus, services should follow the same service description file naming convention as on the system bus, but for backwards compatibility they are not required to do so.

[FIXME the file format should be much better specified than "similar to .desktop entries" esp. since desktop entries are already badly-specified. ;-)] These sections from the specification apply to service files as well:

- General syntax
- Comment format

Service description files must contain a `D-BUS Service` group with at least the keys `Name` (the well-known name of the service) and `Exec` (the command to be executed).

Figure 9. Example service description file

```
# Sample service description file
[D-BUS Service]
Name=com.example.ConfigurationDatabase1
Exec=/usr/bin/sample-configd
```

Additionally, service description files for the well-known system bus on Unix must contain a `User` key, whose value is the name of a user account (e.g. `root`). The system service will be run as that user.

When an application asks to start a service by name, the bus daemon tries to find a service that will own that name. It then tries to spawn the executable associated with it. If this fails, it will report an error.

On the well-known system bus, it is not possible for two .service files in the same directory to offer the same service, because they are constrained to have names that match the service name.

On the well-known session bus, if two .service files in the same directory offer the same service name, the result is undefined. Distributors should avoid this situation, for instance by naming session services' .service files according to their service name.

If two .service files in different directories offer the same service name, the one in the higher-priority directory is used: for instance, on the system bus, .service files in /usr/local/share/dbus-1/system-services take precedence over those in /usr/share/dbus-1/system-services.

The executable launched will have the environment variable `DBUS_STARTER_ADDRESS` set to the address of the message bus so it can connect and request the appropriate names.

The executable being launched may want to know whether the message bus starting it is one of the well-known message buses (see the section called “Well-known Message Bus Instances”). To facilitate this, the bus must also set the `DBUS_STARTER_BUS_TYPE` environment variable if it is one of the well-known buses. The currently-defined values for this variable are `system` for the systemwide message bus, and `session` for the per-login-session message bus. The new executable must still connect to the address given in `DBUS_STARTER_ADDRESS`, but may assume that the resulting connection is to the well-known bus.

[FIXME there should be a timeout somewhere, either specified in the .service file, by the client, or just a global value and if the client being activated fails to connect within that timeout, an error should be sent back.]

Message Bus Service Scope

The "scope" of a service is its "per-", such as per-session, per-machine, per-home-directory, or per-display. The reference implementation doesn't yet support starting services in a different scope from the message bus itself. So e.g. if you start a service on the session bus its scope is per-session.

We could add an optional scope to a bus name. For example, for per-(display,session pair), we could have a unique ID for each display generated automatically at login and set on screen 0 by executing a special "set display ID" binary. The ID would be stored in a `_DBUS_DISPLAY_ID` property and would be a string of random bytes. This ID would then be used to scope names. Starting/locating a service could be done by ID-name pair rather than only by name.

Contrast this with a per-display scope. To achieve that, we would want a single bus spanning all sessions using a given display. So we might set a `_DBUS_DISPLAY_BUS_ADDRESS` property on screen 0 of the display, pointing to this bus.

systemd Activation

Service description files may contain a `SystemdService` key. Its value is the name of a systemd service, for example `dbus-com.example.MyDaemon.service`.

If this key is present, the bus daemon may carry out activation for this D-Bus service by sending a request to systemd asking it to start the systemd service whose name is the value of `SystemdService`. For example, the reference `dbus-daemon` has a `--systemd-activation` option that enables this feature, and that option is given when it is started by systemd.

On the well-known system bus, it is a common practice to set `SystemdService` to `dbus-`, followed by the well-known bus name, followed by `.service`, then register that name as an alias for the real systemd service. This allows D-Bus activation of a service to be enabled or disabled independently of whether the service is started by systemd during boot.

Mediating Activation with AppArmor

Please refer to AppArmor documentation for general information on AppArmor, and how it mediates D-Bus messages when used in conjunction with a kernel and `dbus-daemon` that support this.

In recent versions of the reference `dbus-daemon`, AppArmor policy rules of type `dbus send` are also used to control auto-starting: if a message is sent to the well-known name of an activatable service, the `dbus-daemon` will attempt to determine whether it would deliver the message to that service *before* auto-starting it, by making some assumptions about the resulting process's credentials.

If it does proceed with auto-starting, when the service appears, the `dbus-daemon` repeats the policy check (with the service's true credentials, which might not be identical) before delivering the message. In practice, this second check will usually be more strict than the first; the first check would only be more strict if there are "blacklist"-style rules like `deny dbus send peer=(label=/usr/bin/protected)` that match on the peer's specific credentials, but AppArmor is normally used in a "whitelist" style where this does not apply.

To support this process, service description files may contain a `AssumedAppArmorLabel` key. Its value is the name of an AppArmor label, for example `/usr/sbin/mydaemon`. If present, AppArmor mediation of messages that auto-start a service will decide whether to allow auto-starting to occur based on the assumption that the activated service will be confined under the specified label; in particular, rules of the form `dbus send peer=(label=/usr/sbin/mydaemon)` or `deny dbus send peer=(label=/usr/sbin/mydaemon)` will match it, allowing or denying as appropriate (even if there is in fact no profile of that name loaded).

Otherwise, AppArmor mediation of messages that auto-start a service will decide whether to allow auto-starting to occur without specifying any particular label. In particular, any rule of the form `dbus send peer=(label=X)` or `deny dbus send peer=(label=X)` (for any value of X, including the special label `unconfined`) will not influence whether the auto-start is allowed.

Rules of type `dbus receive` are not checked when deciding whether to allow auto-starting; they are only checked against the service's profile after the service has started, when deciding whether to deliver the message that caused the auto-starting operation.

Explicit activation via the section called “`org.freedesktop.DBus.StartServiceByName`” is not currently affected by this mediation: if a confined process is to be prevented from starting arbitrary services, then it must not be allowed to call that method.

Well-known Message Bus Instances

Two standard message bus instances are defined here, along with how to locate them and where their service files live.

Login session message bus

Each time a user logs in, a *login session message bus* may be started. All applications in the user's login session may interact with one another using this message bus.

The address of the login session message bus is given in the `DBUS_SESSION_BUS_ADDRESS` environment variable. If that variable is not set, applications may also try to read the address from the X Window System root window property `_DBUS_SESSION_BUS_ADDRESS`. The root window property must have type `STRING`. The environment variable should have precedence over the root window property.

The address of the login session message bus is given in the `DBUS_SESSION_BUS_ADDRESS` environment variable. If DBUS_SESSION_BUS_ADDRESS is not set, or if it's set to the string "autolaunch:", the system should use platform-specific methods of locating a running D-Bus session server, or starting one if a running instance cannot be found. Note that this mechanism is not recommended for attempting to determine if a daemon is running. It is inherently racy to attempt to make this determination, since the bus daemon may be started just before or just after the determination is made. Therefore, it is recommended that applications do not try to make this determination for their functionality purposes, and instead they should attempt to start the server.

X Windowing System

For the X Windowing System, the application must locate the window owner of the selection represented by the atom formed by concatenating:

- the literal string "_DBUS_SESSION_BUS_SELECTION_"
- the current user's username
- the literal character '_' (underscore)
- the machine's ID

The following properties are defined for the window that owns this X selection:

Atom	meaning
_DBUS_SESSION_BUS_ADDRESS	the actual address of the server socket
_DBUS_SESSION_BUS_PID	the PID of the server process

At least the _DBUS_SESSION_BUS_ADDRESS property MUST be present in this window.

If the X selection cannot be located or if reading the properties from the window fails, the implementation MUST conclude that there is no D-Bus server running and proceed to start a new server. (See below on concurrency issues)

Failure to connect to the D-Bus server address thus obtained MUST be treated as a fatal connection error and should be reported to the application.

As an alternative, an implementation MAY find the information in the following file located in the current user's home directory, in subdirectory .dbus/session-bus/:

- the machine's ID
- the literal character '-' (dash)
- the X display without the screen number, with the following prefixes removed, if present: ":", "localhost:" ."localhost.localdomain:". That is, a display of "localhost:10.0" produces just the number "10"

The contents of this file NAME=value assignment pairs and lines starting with # are comments (no comments are allowed otherwise). The following variable names are defined:

Variable	meaning
DBUS_SESSION_BUS_ADDRESS	the actual address of the server socket

DBUS_SESSION_BUS_PID	the PID of the server process
DBUS_SESSION_BUS_WINDOWID	the window ID

At least the DBUS_SESSION_BUS_ADDRESS variable MUST be present in this file.

Failure to open this file MUST be interpreted as absence of a running server. Therefore, the implementation MUST proceed to attempting to launch a new bus server if the file cannot be opened.

However, success in opening this file MUST NOT lead to the conclusion that the server is running. Thus, a failure to connect to the bus address obtained by the alternative method MUST NOT be considered a fatal error. If the connection cannot be established, the implementation MUST proceed to check the X selection settings or to start the server on its own.

If the implementation concludes that the D-Bus server is not running it MUST attempt to start a new server and it MUST also ensure that the daemon started as an effect of the "autolaunch" mechanism provides the lookup mechanisms described above, so subsequent calls can locate the newly started server. The implementation MUST also ensure that if two or more concurrent initiations happen, only one server remains running and all other initiations are able to obtain the address of this server and connect to it. In other words, the implementation MUST ensure that the X selection is not present when it attempts to set it, without allowing another process to set the selection between the verification and the setting (e.g., by using XGrabServer / XungrabServer).

Finding session services

On Unix systems, the session bus should search for .service files in `$XDG_DATA_DIRS/dbus-1/services` as defined by the XDG Base Directory Specification. Implementations may also search additional locations, with a higher or lower priority than the XDG directories.

As described in the XDG Base Directory Specification, software packages should install their session .service files to their configured `${datadir}/dbus-1/services`, where `${datadir}` is as defined by the GNU coding standards. System administrators or users can arrange for these service files to be read by setting XDG_DATA_DIRS or by symlinking them into the default locations.

System message bus

A computer may have a *system message bus*, accessible to all applications on the system. This message bus may be used to broadcast system events, such as adding new hardware devices, changes in the printer queue, and so forth.

The address of the system message bus is given in the `DBUS_SYSTEM_BUS_ADDRESS` environment variable. If that variable is not set, applications should try to connect to the well-known address `unix:path=/var/run/dbus/system_bus_socket`. [3]

On Unix systems, the system bus should default to searching for .service files in `/usr/local/share/dbus-1/system-services`, `/usr/share/dbus-1/system-services` and `/lib/dbus-1/system-services`, with that order of precedence. It may also search other implementation-specific locations, but should not vary these locations based on environment variables. [4]

Software packages should install their system .service files to their configured `${datadir}/dbus-1/system-services`, where `${datadir}` is as defined by the GNU coding standards. System administrators can arrange for these service files to be read by editing the system bus' configuration file or by symlinking them into the default locations.

Message Bus Messages

The special message bus name `org.freedesktop.DBus` responds to a number of additional messages at the object path `/org/freedesktop/DBus`. That object path is also used when emitting the the section called “`org.freedesktop.DBus.NameOwnerChanged`” signal.

For historical reasons, some of the methods in the `org.freedesktop.DBus` interface are available on multiple object paths. Message bus implementations should accept method calls that were added before specification version 0.26 on any object path. Message bus implementations should not accept newer method calls on unexpected object paths, and as a security hardening measure, older method calls that are security-sensitive may be rejected with the error `org.freedesktop.DBus.Error.AccessDenied` when called on an unexpected object path. Client software should send all method calls to `/org/freedesktop/DBus` instead of relying on this.

In addition to the method calls listed below, the message bus should implement the standard Introspectable, Properties and Peer interfaces (see the section called “Standard Interfaces”). Support for the Properties and Peer interfaces was added in version 1.11.x of the reference implementation of the message bus.

`org.freedesktop.DBus.Hello`

As a method:

```
STRING Hello ()
```

Reply arguments:

Argument	Type	Description
0	STRING	Unique name assigned to the connection

Before an application is able to send messages to other applications it must send the `org.freedesktop.DBus.Hello` message to the message bus to obtain a unique name. If an application without a unique name tries to send a message to another application, or a message to the message bus itself that isn't the `org.freedesktop.DBus.Hello` message, it will be disconnected from the bus.

There is no corresponding "disconnect" request; if a client wishes to disconnect from the bus, it simply closes the socket (or other communication channel).

`org.freedesktop.DBus.RequestName`

As a method:

```
UINT32 RequestName (in STRING name, in UINT32 flags)
```

Message arguments:

Argument	Type	Description
0	STRING	Name to request
1	UINT32	Flags

Reply arguments:

Argument	Type	Description
0	UINT32	Return value

Ask the message bus to assign the given name to the method caller. Each name maintains a queue of possible owners, where the head of the queue is the primary or current owner of the name. Each potential owner in the queue maintains the DBUS_NAME_FLAG_ALLOW_REPLACEMENT and DBUS_NAME_FLAG_DO_NOT_QUEUE settings from its latest RequestName call. When RequestName is invoked the following occurs:

- If the method caller is currently the primary owner of the name, the DBUS_NAME_FLAG_ALLOW_REPLACEMENT and DBUS_NAME_FLAG_DO_NOT_QUEUE values are updated with the values from the new RequestName call, and nothing further happens.
- If the current primary owner (head of the queue) has DBUS_NAME_FLAG_ALLOW_REPLACEMENT set, and the RequestName invocation has the DBUS_NAME_FLAG_REPLACE_EXISTING flag, then the caller of RequestName replaces the current primary owner at the head of the queue and the current primary owner moves to the second position in the queue. If the caller of RequestName was in the queue previously its flags are updated with the values from the new RequestName in addition to moving it to the head of the queue.
- If replacement is not possible, and the method caller is currently in the queue but not the primary owner, its flags are updated with the values from the new RequestName call.
- If replacement is not possible, and the method caller is currently not in the queue, the method caller is appended to the queue.
- If any connection in the queue has DBUS_NAME_FLAG_DO_NOT_QUEUE set and is not the primary owner, it is removed from the queue. This can apply to the previous primary owner (if it was replaced) or the method caller (if it updated the DBUS_NAME_FLAG_DO_NOT_QUEUE flag while still stuck in the queue, or if it was just added to the queue with that flag set).

Note that DBUS_NAME_FLAG_REPLACE_EXISTING results in "jumping the queue," even if another application already in the queue had specified DBUS_NAME_FLAG_REPLACE_EXISTING. This comes up if a primary owner that does not allow replacement goes away, and the next primary owner does allow replacement. In this case, queued items that specified DBUS_NAME_FLAG_REPLACE_EXISTING *do not* automatically replace the new primary owner. In other words, DBUS_NAME_FLAG_REPLACE_EXISTING is not saved, it is only used at the time RequestName is called. This is deliberate to avoid an infinite loop anytime two applications are both DBUS_NAME_FLAG_ALLOW_REPLACEMENT and DBUS_NAME_FLAG_REPLACE_EXISTING.

The flags argument contains any of the following values logically ORed together:

Conventional Name	Value	Description
DBUS_NAME_FLAG_ALLOW_REPLACEMENT	0x1	If an application A specifies this flag and succeeds in becoming the owner of the name, and another application B later calls RequestName with the DBUS_NAME_FLAG_REPLACE_EXISTING flag, then application A will lose ownership and receive a `org.freedesktop.DBus.NameLost` signal, and application B will become the new owner. If DBUS_NAME_FLAG_ALLOW_REPLACEMENT is not specified by application A, or DBUS_NAME_FLAG_REPLACE_EXISTING is not specified by application B, then application B will not replace application A as the owner.
DBUS_NAME_FLAG_REPLACE_EXISTING	0x2	Try to replace the current owner if there is one. If this flag is not set the application will only become the owner of the name if there is no current owner. If this flag is set, the application will replace the current owner if the current owner specified DBUS_NAME_FLAG_ALLOW_REPLACEMENT.
DBUS_NAME_FLAG_DO_NOT_QUEUE	0x4	Without this flag, if an application requests a name that is already owned, the application will be placed in a queue to own the name when the current owner gives it up. If this flag is given, the application will not be placed in the queue, the request for the name will simply fail. This flag also affects behavior when an application is replaced as name owner; by default the application moves back into the waiting queue, unless this flag was provided when the application became the name owner.

The return code can be one of the following values:

Conventional Name	Value	Description
DBUS_REQUEST_NAME_REPLY_PRIMARY_OWNER	1	The caller is now the primary owner of the name, replacing any previous owner. Either the name had no owner before, or the caller specified DBUS_NAME_FLAG_REPLACE_EXISTING and the current owner specified DBUS_NAME_FLAG_ALLOW_REPLACEMENT.
DBUS_REQUEST_NAME_REPLY_IN_QUEUE	2	The name already had an owner, DBUS_NAME_FLAG_DO_NOT_QUEUE was not specified, and either the current owner did not specify DBUS_NAME_FLAG_ALLOW_REPLACEMENT or the requesting application did not specify DBUS_NAME_FLAG_REPLACE_EXISTING.
DBUS_REQUEST_NAME_REPLY_EXISTS	3	The name already has an owner, DBUS_NAME_FLAG_DO_NOT_QUEUE was specified, and either DBUS_NAME_FLAG_ALLOW_REPLACEMENT was not specified by the current owner, or DBUS_NAME_FLAG_REPLACE_EXISTING was not specified by the requesting application.
DBUS_REQUEST_NAME_REPLY_ALREADY_OWNER	4	The application trying to request ownership of a name is already the owner of it.

`org.freedesktop.DBus.ReleaseName`

As a method:

```
UINT32 ReleaseName (in STRING name)
```

Message arguments:

Argument	Type	Description
0	STRING	Name to release

Reply arguments:

Argument	Type	Description
0	UINT32	Return value

Ask the message bus to release the method caller's claim to the given name. If the caller is the primary owner, a new primary owner will be selected from the queue if any other owners are waiting. If the caller is waiting in the queue for the name, the caller will removed from the queue and will not be made an owner of the name if it later becomes available. If there are no other owners in the queue for the name, it will be removed from the bus entirely. The return code can be one of the following values:

Conventional Name	Value	Description
DBUS_RELEASE_NAME_REPLY_RELEASED	1	The caller has released his claim on the given name. Either the caller was the primary owner of the name, and the name is now unused or taken by somebody waiting in the queue for the name, or the caller was waiting in the queue for the name and has now been removed from the queue.
DBUS_RELEASE_NAME_REPLY_NON_EXISTENT	2	The given name does not exist on this bus.
DBUS_RELEASE_NAME_REPLY_NOT_OWNER	3	The caller was not the primary owner of this name, and was also not waiting in the queue to own this name.

`org.freedesktop.DBus.ListQueuedOwners`

As a method:

```
ARRAY of STRING ListQueuedOwners (in STRING name)
```

Message arguments:

Argument	Type	Description
0	STRING	The well-known bus name to query, such as `com.example.cappuccino`

Reply arguments:

Argument	Type	Description
0	ARRAY of STRING	The unique bus names of connections currently queued for the name

List the connections currently queued for a bus name (see Queued Name Owner).

`org.freedesktop.DBus.ListNames`

As a method:

```
ARRAY of STRING ListNames ()
```

Reply arguments:

Argument	Type	Description
0	ARRAY of STRING	Array of strings where each string is a bus name

Returns a list of all currently-owned names on the bus.

`org.freedesktop.DBus.ListActivatableNames`

As a method:

```
ARRAY of STRING ListActivatableNames ()
```

Reply arguments:

Argument	Type	Description
0	ARRAY of STRING	Array of strings where each string is a bus name

Returns a list of all names that can be activated on the bus.

`org.freedesktop.DBus.NameHasOwner`

As a method:

```
BOOLEAN NameHasOwner (in STRING name)
```

Message arguments:

Argument	Type	Description
0	STRING	Name to check

Reply arguments:

Argument	Type	Description
0	BOOLEAN	Return value, true if the name exists

Checks if the specified name exists (currently has an owner).

`org.freedesktop.DBus.NameOwnerChanged`

This is a signal:

```
NameOwnerChanged (STRING name, STRING old_owner, STRING new_owner)
```

Message arguments:

Argument	Type	Description
0	STRING	Name with a new owner
1	STRING	Old owner or empty string if none
2	STRING	New owner or empty string if none

This signal indicates that the owner of a name has changed. It's also the signal to use to detect the appearance of new names on the bus.

`org.freedesktop.DBus.NameLost`

This is a signal:

```
NameLost (STRING name)
```

Message arguments:

Argument	Type	Description
0	STRING	Name which was lost

This signal is sent to a specific application when it loses ownership of a name.

`org.freedesktop.DBus.NameAcquired`

This is a signal:

```
NameAcquired (STRING name)
```

Message arguments:

Argument	Type	Description
0	STRING	Name which was acquired

This signal is sent to a specific application when it gains ownership of a name.

`org.freedesktop.DBus.ActivatableServicesChanged`

This is a signal:

```
ActivatableServicesChanged ()
```

This signal is sent when the list of activatable services, as returned by ListActivatableNames(), might have changed (see the section called "`org.freedesktop.DBus.ListActivatableNames`"). Clients that have cached information about the activatable services should call ListActivatableNames() again to update their cache.

The presence of this signal is indicated by a bus feature property (for details see the section called "`org.freedesktop.DBus.Features`"). In older implementations that do not have this feature, there is no way to be informed when the list of activatable names has changed.

`org.freedesktop.DBus.StartServiceByName`

As a method:

```
UINT32 StartServiceByName (in STRING name, in UINT32 flags)
```

Message arguments:

Argument	Type	Description
0	STRING	Name of the service to start
1	UINT32	Flags (currently not used)

Reply arguments:

Argument	Type	Description
0	UINT32	Return value

Tries to launch the executable associated with a name (service activation), as an explicit request. This is an alternative to relying on auto-starting. For more information on how services are activated and the difference between auto-starting and explicit activation, see the section called “Message Bus Starting Services (Activation)”.

It is often preferable to carry out auto-starting instead of calling this method. This is because calling this method is subject to a time-of-check/time-of-use issue: if a caller asks the message bus to start a service so that the same caller can make follow-up method calls to that service, the fact that the message bus was able to start the required service is no guarantee that it will not have crashed or otherwise exited by the time the caller makes those follow-up method calls. As a result, calling this method does not remove the need for the caller to handle errors from method calls. Given that fact, it is usually simpler to rely on auto-starting, in which the required service starts as a side-effect of the first method call.

The return value can be one of the following values:

Identifier	Value	Description
DBUS_START_REPLY_SUCCESS	1	The service was successfully started.
DBUS_START_REPLY_ALREADY_RUNNING	2	A connection already owns the given name.

`org.freedesktop.DBus.UpdateActivationEnvironment`

As a method:

```
UpdateActivationEnvironment (in ARRAY of DICT_ENTRY<STRING,STRING> environment)
```

Message arguments:

Argument	Type	Description
0	ARRAY of DICT_ENTRY<STRING,STRING>	Environment to add or update

Normally, session bus activated services inherit the environment of the bus daemon. This method adds to or modifies that environment when activating services.

Some bus instances, such as the standard system bus, may disable access to this method for some or all callers.

Note, both the environment variable names and values must be valid UTF-8. There's no way to update the activation environment with data that is invalid UTF-8.

`org.freedesktop.DBus.GetNameOwner`

As a method:

```
STRING GetNameOwner (in STRING name)
```

Message arguments:

Argument	Type	Description
0	STRING	Name to get the owner of

Reply arguments:

Argument	Type	Description
0	STRING	Return value, a unique connection name

Returns the unique connection name of the primary owner of the name given. If the requested name doesn't have an owner, returns a `org.freedesktop.DBus.Error.NameHasNoOwner` error.

`org.freedesktop.DBus.GetConnectionUnixUser`

As a method:

```
UINT32 GetConnectionUnixUser (in STRING bus_name)
```

Message arguments:

Argument	Type	Description
0	STRING	Unique or well-known bus name of the connection to query, such as `:12.34` or `com.example.tea`

Reply arguments:

Argument	Type	Description
0	UINT32	Unix user ID

Returns the Unix user ID of the process connected to the server. If unable to determine it (for instance, because the process is not on the same machine as the bus daemon), an error is returned.

`org.freedesktop.DBus.GetConnectionUnixProcessID`

As a method:

```
UINT32 GetConnectionUnixProcessID (in STRING bus_name)
```

Message arguments:

Argument	Type	Description
0	STRING	Unique or well-known bus name of the connection to query, such as `:12.34` or `com.example.tea`

Reply arguments:

Argument	Type	Description
0	UINT32	Unix process id

Returns the Unix process ID of the process connected to the server. If unable to determine it (for instance, because the process is not on the same machine as the bus daemon), an error is returned.

`org.freedesktop.DBus.GetConnectionCredentials`

As a method:

```
ARRAY of DICT_ENTRY<STRING,VARIANT> GetConnectionCredentials (in STRING bus_name)
```

Message arguments:

Argument	Type	Description
0	STRING	Unique or well-known bus name of the connection to query, such as `:12.34` or `com.example.tea`

Reply arguments:

Argument	Type	Description
0	ARRAY of DICT_ENTRY<STRING,VARIANT>	Credentials

Returns as many credentials as possible for the process connected to the server. If unable to determine certain credentials (for instance, because the process is not on the same machine as the bus daemon, or because this version of the bus daemon does not support a particular security framework), or if the values of those credentials cannot be represented as documented here, then those credentials are omitted.

Keys in the returned dictionary not containing "." are defined by this specification. Bus daemon implementors supporting credentials frameworks not mentioned in this document should either contribute patches to this specification, or use keys containing "." and starting with a reversed domain name.

Key	Value type	Value
UnixUserID	UINT32	The numeric Unix user ID, as defined by POSIX
UnixGroupIDs	ARRAY of UINT32	The numeric Unix group IDs (including both the primary group and the supplementary groups), as defined by POSIX, in numerically sorted order. This array is either complete or absent: if the message bus is able to determine some but not all of the caller's groups, or if one of the groups is not representable in a UINT32, it must not add this credential to the dictionary.
ProcessID	UINT32	The numeric process ID, on platforms that have this concept. On Unix, this is the process ID defined by POSIX.
WindowsSID	STRING	The Windows security identifier in its string form, e.g. "S-1-5-21-3623811015-3361044348-30300820-1013" for a domain or local computer user or "S-1-5-18" for the LOCAL_SYSTEM user
LinuxSecurityLabel	ARRAY of BYTE	On Linux systems, the security label that would result from the SO_PEERSEC getsockopt call. The array contains the non-zero bytes of the security label in an unspecified ASCII-compatible encoding[a], followed by a single zero byte. For example, the SELinux context `system_u:system_r:init_t:s0` (a string of length 27) would be encoded as 28 bytes ending with ':', 's', '0', '\x00'.[b] On SELinux systems this is the SELinux context, as output by `ps -Z` or `ls -Z`. Typical values might include `system_u:system_r:init_t:s0`, `unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023`, or `unconfined_u:unconfined_r:chrome_sandbox_t:s0-s0:c0.c1023`. On Smack systems, this is the Smack label. Typical values might include `_`, `*`, `User`, `System` or `System::Shared`. On AppArmor systems, this is the AppArmor context, a composite string encoding the AppArmor label (one or more profiles) and the enforcement mode. Typical values might include `unconfined`, `/usr/bin/firefox (enforce)` or `user1 (complain)`.

Key	Value type	Value
[a] It could be ASCII or UTF-8, but could also be ISO Latin-1 or any other encoding. [b] Note that this is not the same as the older GetConnectionSELinuxContext method, which does not append the zero byte. Always appending the zero byte allows callers to read the string from the message payload without copying.		

This method was added in D-Bus 1.7 to reduce the round-trips required to list a process's credentials. In older versions, calling this method will fail: applications should recover by using the separate methods such as the section called “`org.freedesktop.DBus.GetConnectionUnixUser`” instead.

`org.freedesktop.DBus.GetAdtAuditSessionData`

As a method:

```
ARRAY of BYTE GetAdtAuditSessionData (in STRING bus_name)
```

Message arguments:

Argument	Type	Description
0	STRING	Unique or well-known bus name of the connection to query, such as `:12.34` or `com.example.tea`

Reply arguments:

Argument	Type	Description
0	ARRAY of BYTE	auditing data as returned by adt_export_session_data()

Returns auditing data used by Solaris ADT, in an unspecified binary format. If you know what this means, please contribute documentation via the D-Bus bug tracking system. This method is on the core DBus interface for historical reasons; the same information should be made available via the section called “`org.freedesktop.DBus.GetConnectionCredentials`” in future.

`org.freedesktop.DBus.GetConnectionSELinuxSecurityContext`

As a method:

```
ARRAY of BYTE GetConnectionSELinuxSecurityContext (in STRING bus_name)
```

Message arguments:

Argument	Type	Description
0	STRING	Unique or well-known bus name of the connection to query, such as `:12.34` or `com.example.tea`

Reply arguments:

Argument	Type	Description
0	ARRAY of BYTE	some sort of string of bytes, not necessarily UTF-8, not including '\0'

Returns the security context used by SELinux, in an unspecified format. If you know what this means, please contribute documentation via the D-Bus bug tracking system. This method is on the core DBus interface for historical reasons; the same information should be made available via the section called “`org.freedesktop.DBus.GetConnectionCredentials`” in future.

`org.freedesktop.DBus.AddMatch`

As a method:

```
AddMatch (in STRING rule)
```

Message arguments:

Argument	Type	Description
0	STRING	Match rule to add to the connection

Adds a match rule to match messages going through the message bus (see the section called “Match Rules”). If the bus does not have enough resources the `org.freedesktop.DBus.Error.OOM` error is returned.

`org.freedesktop.DBus.RemoveMatch`

As a method:

```
RemoveMatch (in STRING rule)
```

Message arguments:

Argument	Type	Description
0	STRING	Match rule to remove from the connection

Removes the first rule that matches (see the section called “Match Rules”). If the rule is not found the `org.freedesktop.DBus.Error.MatchRuleNotFound` error is returned.

`org.freedesktop.DBus.GetId`

As a method:

```
GetId (out STRING id)
```

Reply arguments:

Argument	Type	Description
0	STRING	Unique ID identifying the bus daemon

Gets the unique ID of the bus. The unique ID here is shared among all addresses the bus daemon is listening on (TCP, UNIX domain socket, etc.) and its format is described in the section called “UUIDs”. Each address the bus is listening on also has its own unique ID, as described in the section called “Server Addresses”. The per-bus and per-address IDs are not related. There is also a per-machine ID, described in the section called “`org.freedesktop.DBus.Peer`” and returned by org.freedesktop.DBus.Peer.GetMachineId(). For a desktop session bus, the bus ID can be used as a way to uniquely identify a user's session.

`org.freedesktop.DBus.Monitoring.BecomeMonitor`

As a method:

```
BecomeMonitor (in ARRAY of STRING rule, in UINT32 flags)
```

Message arguments:

Argument	Type	Description
0	ARRAY of STRING	Match rules to add to the connection
1	UINT32	Not used, must be 0

Converts the connection into a *monitor connection* which can be used as a debugging/monitoring tool. Only a user who is privileged on this bus (by some implementation-specific definition) may create monitor connections[5].

Monitor connections lose all their bus names, including the unique connection name, and all their match rules. Sending messages on a monitor connection is not allowed: applications should use a private connection for monitoring.

Monitor connections may receive all messages, even messages that should only have gone to some other connection ("eavesdropping"). The first argument is a list of match rules, which replace any match rules that were previously active for this connection. These match rules are always treated as if they contained the special `eavesdrop='true'` member.

As a special case, an empty list of match rules (which would otherwise match nothing, making the monitor useless) is treated as a shorthand for matching all messages.

The second argument might be used for flags to influence the behaviour of the monitor connection in future D-Bus versions.

Message bus implementations should attempt to minimize the side-effects of monitoring — in particular, unlike ordinary eavesdropping, monitoring the system bus does not require the access control rules to be relaxed, which would change the set of messages that can be delivered to their (non-monitor) destinations. However, it is unavoidable that monitoring will increase the message bus's resource consumption. In edge cases where there was barely enough time or memory without monitoring, this might result in message deliveries failing when they would otherwise have succeeded.

Message Bus Properties

The special message bus name `org.freedesktop.DBus` exports several properties (see the section called “`org.freedesktop.DBus.Properties`”) on the object path `/org/freedesktop/DBus`.

`org.freedesktop.DBus.Features`

As a property:

```
Read-only constant ARRAY of STRING Features
```

This property lists abstract “features” provided by the message bus, and can be used by clients to detect the capabilities of the message bus with which they are communicating. This property was added in version 1.11.x of the reference implementation of the message bus.

Items in the returned array not containing “.” are defined by this specification. Bus daemon implementors wishing to advertise features not mentioned in this document should either contribute patches to this specification, or use keys containing “.” and starting with their own reversed domain name, for example `com.example.MyBus.SubliminalMessages`.

The features currently defined in this specification are as follows:

`ActivatableServicesChanged`

> This message bus emits the `ActivatableServicesChanged` signal whenever its list of activatable services might have changed (for details see the section called “`org.freedesktop.DBus.ActivatableServicesChanged`”).

`AppArmor`

This message bus filters messages via the AppArmor security framework. This feature should only be advertised if AppArmor mediation is enabled and active at runtime; merely compiling in support for AppArmor should not result in this feature being advertised on message bus instances where it is disabled by message bus or operating system configuration.

`HeaderFiltering`

This message bus guarantees that it will remove header fields that it does not understand when it relays messages, so that a client receiving a recently-defined header field that is specified to be controlled by the message bus can safely assume that it was in fact set by the message bus. This check is needed because older message bus implementations did not guarantee to filter headers in this way, so a malicious client could send any recently-defined header field with a crafted value of its choice through an older message bus that did not understand that header field.

`SELinux`

This message bus filters messages via the SELinux security framework. Similar to `apparmor`, this feature should only be advertised if SELinux mediation is enabled and active at runtime (if SELinux is placed in permissive mode, that is still considered to be active).

`SystemdActivation`

When asked to activate a service that has the `SystemdService` field in its `.service` file, this message bus will carry out systemd activation (for details see the section called “systemd Activation”).

`org.freedesktop.DBus.Interfaces`

As a property:

```
Read-only constant ARRAY of STRING Interfaces
```

This property lists interfaces provided by the `/org/freedesktop/DBus` object, and can be used by clients to detect the capabilities of the message bus with which they are communicating. Unlike the standard Introspectable interface, querying this property does not require parsing XML. This property was added in version 1.11.x of the reference implementation of the message bus.

The standard `org.freedesktop.DBus` and `org.freedesktop.DBus.Properties` interfaces are not included in the value of this property, because their presence can be inferred from the fact that a method call on `org.freedesktop.DBus.Properties` asking for properties of `org.freedesktop.DBus` was successful. The standard `org.freedesktop.DBus.Peer` and `org.freedesktop.DBus.Introspectable` interfaces are not included in the value of this property either, because they do not indicate features of the message bus implementation.

Glossary

This glossary defines some of the terms used in this specification.

Bus Name

The message bus maintains an association between names and connections. (Normally, there's one connection per application.) A bus name is simply an identifier used to locate connections. For example, the hypothetical `com.yoyodyne.Screensaver` name might be used to send a message to a screensaver from Yoyodyne Corporation. An application is said to *own* a name if the message bus has associated the application's connection with the name. Names may also have *queued owners* (see Queued Name Owner). The bus assigns a unique name to each connection, see Unique Connection Name. Other names can be thought of as "well-known names" and are used to find applications that offer specific functionality.

See the section called “Bus names” for details of the syntax and naming conventions for bus names.

Message

A message is the atomic unit of communication via the D-Bus protocol. It consists of a *header* and a *body*; the body is made up of *arguments*.

Message Bus

The message bus is a special application that forwards or routes messages between a group of applications connected to the message bus. It also manages *names* used for routing messages.

Name

See Bus Name. "Name" may also be used to refer to some of the other names in D-Bus, such as interface names.

Namespace

ex: com.google.myApp

Used to prevent collisions when defining new interfaces, bus names etc. The convention used is the same one Java uses for defining classes: a reversed domain name. See the section called “Bus names”, the section called “Interface names”, the section called “Error names”, the section called “Valid Object Paths”.

Object

Each application contains *objects*, which have *interfaces* and *methods*. Objects are referred to by a name, called a *path*.

One-to-One

An application talking directly to another application, without going through a message bus. One-to-one connections may be "peer to peer" or "client to server." The D-Bus protocol has no concept of client vs. server after a connection has authenticated; the flow of messages is symmetrical (full duplex).

Path

Object references (object names) in D-Bus are organized into a filesystem-style hierarchy, so each object is named by a path. As in LDAP, there's no difference between "files" and "directories"; a path can refer to an object, while still having child objects below it.

Queued Name Owner

Each bus name has a primary owner; messages sent to the name go to the primary owner. However, certain names also maintain a queue of secondary owners "waiting in the wings." If the primary owner releases the name, then the first secondary owner in the queue automatically becomes the new owner of the name.

Service

A service is an executable that can be launched by the bus daemon. Services normally guarantee some particular features, for example they may guarantee that they will request a specific name such as "com.example.Screensaver1", have a singleton object "/com/example/Screensaver1", and that object will implement the interface "com.example.Screensaver1.Control".

Service Description Files

".service files" tell the bus about service applications that can be launched (see Service). Most importantly they provide a mapping from bus names to services that will request those names when they start up.

Unique Connection Name

The special name automatically assigned to each connection by the message bus. This name will never change owner, and will be unique (never reused during the lifetime of the message bus). It will begin with a ':' character.

---

[1] Lockfiles are used instead of real file locking `fcntl()` because real locking implementations are still flaky on network filesystems.

[2] This idiosyncratic quoting style is based on the rules for escaping items to appear inside single-quoted strings in POSIX `/bin/sh`, but please note that backslashes that are not inside single quotes have different behaviour. This syntax does not offer any way to represent an apostrophe inside single quotes (it is necessary to leave the single-quoted section, backslash-escape the apostrophe and re-enter single quotes), or to represent a comma outside single quotes (it is necessary to wrap it in a single-quoted section).

[3] The D-Bus reference implementation actually honors the `$(localstatedir)` configure option for this address, on both client and server side.

[4] The system bus is security-sensitive and is typically executed by an init system with a clean environment. Its launch helper process is particularly security-sensitive, and specifically clears its own environment.

[5] In the reference implementation, the default configuration is that each user (identified by numeric user ID) may monitor their own session bus, and the root user (user ID zero) may monitor the system bus.