dbus-next Documentation

Release 0.2.3

Tony Crisci

Sep 11, 2022

REFERENCE:

CHAPTER

ONE

THE TYPE SYSTEM

1.1 Variant

class dbus_next.**Variant**(*signature: Union[str, dbus_next.signature.SignatureTree, dbus_next.signature.SignatureType]*, *value: Any*, *verify: bool = True*)

A class to represent a DBus variant (type "v").

This class is used in message bodies to represent variants. The user can expect a value in the body with type "v" to use this class and can construct this class directly for use in message bodies sent over the bus.

Variables

- **signature** (*str*) – The signature for this variant. Must be a single complete type.
- **signature_type** (*SignatureType*) – The parsed signature of this variant.
- **value** (*Any*) – The value of this variant. Must correspond to the signature.

Raises *InvalidSignatureError* if the signature is not valid. *SignatureBodyMismatchError* if the signature does not match the body.

1.2 SignatureTree

class dbus_next.**SignatureTree**(*signature: str = ''*)

A class that represents a signature as a tree structure for conveniently working with DBus signatures.

This class will not normally be used directly by the user.

Variables

- **types** (list(*SignatureType*)) – A list of parsed complete types.
- **signature** (*str*) – The signature of this signature tree.

Raises *InvalidSignatureError* if the given signature is not valid.

verify(*body: List[Any]*)

Verifies that the give body matches this signature tree

Parameters **body** (*list(Any)*) – the body to verify for this tree

Returns True if the signature matches the body or an exception if not.

Raises *SignatureBodyMismatchError* if the signature does not match the body.

1.3 SignatureType

class dbus_next.**SignatureType**(*token: str*)

A class that represents a single complete type within a signature.

This class is not meant to be constructed directly. Use the *SignatureTree* class to parse signatures.

Variables

- **signature** (*str*) – The signature of this complete type.
- **children** (list(*SignatureType*)) – A list of child types if this is a container type. Arrays have one child type, dict entries have two child types (key and value), and structs have child types equal to the number of struct members.

validators = {'(': <function SignatureType._verify_struct>, 'a': <function Signature'

verify(*body: Any*) → bool

Verify that the body matches this type.

Returns True if the body matches this type.

Raises *SignatureBodyMismatchError* if the body does not match this type.

Values that are sent or received over the message bus always have an associated signature that specifies the types of those values. For the high-level client and service, these signatures are specified in XML data which is advertised in a standard DBus interface. The high-level client dynamically creates classes based on this introspection data with methods and signals with arguments based on the type signature. The high-level service does the inverse by introspecting the class to create the introspection XML data which is advertised on the bus for clients.

Each token in the signature is mapped to a Python type as shown in the table below.

Name	Token	Python Type	Notes
BYTE	y	int	An integer 0-255. In an array, it has type `bytes`.
BOOLEAN	b	bool	
INT16	n	int	
UINT16	q	int	
INT32	i	int	
UINT32	u	int	
INT64	x	int	
UINT64	t	int	
DOUBLE	d	float	
STRING	s	str	
OBJECT_PATH	o	str	Must be a valid object path.
SIGNATURE	g	str	Must be a valid signature.
UNIX_FD	h	int	In the low-level interface, an index pointing to a file descriptor in the `unix_fds` member of the *Message*. In the high-level interface, it is the file descriptor itself.
ARRAY	a	list	Must be followed by a complete type which specifies the child type.
STRUCT	(	list	Types in the Python `list` must match the types between the parens.
VARIANT	v	*Variant*	This class is provided by the library.
DICT_ENTRY	{	dict	Must be included in an array to be a `dict`.

The types `a`, `(`, `v`, and `{` are container types that hold other values. Examples of container types and Python examples are in the table below.

Signature	Example	Notes
`(su)`	`[ 'foo', 5 ]`	Each element in the array must match the corresponding type of the struct member.
`as`	`[ 'foo', 'bar' ]`	The child type comes immediately after the `a`. The array can have any number of elements, but they all must match the child type.
`a{su}`	`{ 'foo': 5 }`	An "array of dict entries" is represented by a `dict`. The type after `{` is the key type and the type before the `}` is the value type.
`ay`	`b'\0x62\0x75\0x66'`	Special case: an array of bytes is represented by Python `bytes`.
`v`	`Variant('as', ['hello'])`	Signature must be a single type. A variant may hold a container type.
`(asv)`	`[ ['foo'], Variant('s', 'bar') ]`	Containers may be nested.

For more information on the DBus type system, see the specification.

CHAPTER

TWO

THE HIGH LEVEL CLIENT

2.1 BaseProxyObject

class dbus_next.proxy_object.**BaseProxyObject**(*bus_name: str*, *path: str*, *introspection: Union[*dbus_next.introspection.Node*, str, xml.etree.ElementTree.Element]*, *bus:* dbus_next.message_bus.BaseMessageBus, *ProxyInterface: Type[*dbus_next.proxy_object.BaseProxyInterface*]*)

An abstract class representing a proxy to an object exported on the bus by another client.

Implementations of this class are not meant to be constructed directly. Use *BaseMessageBus.get_proxy_object()* to get a proxy object. Each message bus implementation provides its own proxy object implementation that will be returned by that method.

The primary use of the proxy object is to select a proxy interface to act on. Information on what interfaces are available is provided by introspection data provided to this class. This introspection data can either be included in your project as an XML file (recommended) or retrieved from the `org.freedesktop.DBus.Introspectable` interface at runtime.

Variables

- **bus_name** (*str*) – The name of the bus this object is exported on.
- **path** (*str*) – The object path exported on the client that owns the bus name.
- **introspection** (*Node*) – Parsed introspection data for the proxy object.
- **bus** (*BaseMessageBus*) – The message bus this proxy object is connected to.
- **ProxyInterface** (Type[*BaseProxyInterface*]) – The proxy interface class this proxy object uses.
- **child_paths** (*list(str)*) – A list of absolute object paths of the children of this object.

Raises

- *InvalidBusNameError* - If the given bus name is not valid.
- *InvalidObjectPathError* - If the given object path is not valid.
- *InvalidIntrospectionError* - If the introspection data for the node is not valid.

get_children() → List[*dbus_next.proxy_object.BaseProxyObject*]

Get the child nodes of this proxy object according to the introspection data.

get_interface(*name: str*) → *dbus_next.proxy_object.BaseProxyInterface*

Get an interface exported on this proxy object and connect it to the bus.

Parameters **name** (*str*) – The name of the interface to retrieve.

Raises

- *InterfaceNotFoundError* - If there is no interface by this name exported on the bus.

2.2 BaseProxyInterface

class dbus_next.proxy_object.**BaseProxyInterface**(*bus_name*, *path*, *introspection*, *bus*)

An abstract class representing a proxy to an interface exported on the bus by another client.

Implementations of this class are not meant to be constructed directly by users. Use *BaseProxyObject.get_interface()* to get a proxy interface. Each message bus implementation provides its own proxy interface implementation that will be returned by that method.

Proxy interfaces can be used to call methods, get properties, and listen to signals on the interface. Proxy interfaces are created dynamically with a family of methods for each of these operations based on what members the interface exposes. Each proxy interface implementation exposes these members in a different way depending on the features of the backend. See the documentation of the proxy interface implementation you use for more details.

Variables

- **bus_name** (*str*) – The name of the bus this interface is exported on.
- **path** (*str*) – The object path exported on the client that owns the bus name.
- **introspection** (*Node*) – Parsed introspection data for the proxy interface.
- **bus** (*BaseMessageBus*) – The message bus this proxy interface is connected to.

2.3 aio.ProxyObject

class dbus_next.aio.**ProxyObject**(*bus_name: str*, *path: str*, *introspection: Union[dbus_next.introspection.Node, str, xml.etree.ElementTree.Element]*, *bus: dbus_next.message_bus.BaseMessageBus*)

Bases: *dbus_next.proxy_object.BaseProxyObject*

The proxy object implementation for the GLib *MessageBus*.

For more information, see the *BaseProxyObject*.

get_children() → List[dbus_next.aio.proxy_object.ProxyObject]

Get the child nodes of this proxy object according to the introspection data.

get_interface(*name: str*) → dbus_next.aio.proxy_object.ProxyInterface

Get an interface exported on this proxy object and connect it to the bus.

Parameters **name** (*str*) – The name of the interface to retrieve.

Raises

- *InterfaceNotFoundError* - If there is no interface by this name exported on the bus.

2.4 aio.ProxyInterface

class dbus_next.aio.**ProxyInterface**(*bus_name*, *path*, *introspection*, *bus*)

Bases: *dbus_next.proxy_object.BaseProxyInterface*

A class representing a proxy to an interface exported on the bus by another client for the asyncio *MessageBus* implementation.

This class is not meant to be constructed directly by the user. Use *ProxyObject.get_interface()* on a asyncio proxy object to get a proxy interface.

This class exposes methods to call DBus methods, listen to signals, and get and set properties on the interface that are created dynamically based on the introspection data passed to the proxy object that made this proxy interface.

A *method call* takes this form:

```
result = await interface.call_[METHOD](*args)
```

Where `METHOD` is the name of the method converted to snake case.

DBus methods are exposed as coroutines that take arguments that correpond to the *in args* of the interface method definition and return a `result` that corresponds to the *out arg*. If the method has more than one out arg, they are returned within a `list`.

To *listen to a signal* use this form:

```
interface.on_[SIGNAL](callback)
```

To *stop listening to a signal* use this form:

```
interface.off_[SIGNAL](callback)
```

Where `SIGNAL` is the name of the signal converted to snake case.

DBus signals are exposed with an event-callback interface. The provided `callback` will be called when the signal is emitted with arguments that correspond to the *out args* of the interface signal definition.

To *get or set a property* use this form:

```
value = await interface.get_[PROPERTY]()
await interface.set_[PROPERTY](value)
```

Where `PROPERTY` is the name of the property converted to snake case.

DBus property getters and setters are exposed as coroutines. The `value` must correspond to the type of the property in the interface definition.

If the service returns an error for a DBus call, a *DBusError* will be raised with information about the error.

2.5 glib.ProxyObject

class dbus_next.glib.**ProxyObject**(*bus_name: str*, *path: str*, *introspection: Union[dbus_next.introspection.Node, str, xml.etree.ElementTree.Element]*, *bus: dbus_next.message_bus.BaseMessageBus*)

Bases: *dbus_next.proxy_object.BaseProxyObject*

The proxy object implementation for the asyncio *MessageBus*.

For more information, see the *BaseProxyObject*.

get_children() → List[dbus_next.glib.proxy_object.ProxyObject]
Get the child nodes of this proxy object according to the introspection data.

get_interface(*name: str*) → dbus_next.glib.proxy_object.ProxyInterface
Get an interface exported on this proxy object and connect it to the bus.

Parameters **name** (*str*) – The name of the interface to retrieve.

Raises

- *InterfaceNotFoundError* - If there is no interface by this name exported on the bus.

2.6 glib.ProxyInterface

class dbus_next.glib.**ProxyInterface**(*bus_name*, *path*, *introspection*, *bus*)

Bases: *dbus_next.proxy_object.BaseProxyInterface*

A class representing a proxy to an interface exported on the bus by another client for the GLib *MessageBus* implementation.

This class is not meant to be constructed directly by the user. Use *ProxyObject.get_interface()* on a GLib proxy object to get a proxy interface.

This class exposes methods to call DBus methods, listen to signals, and get and set properties on the interface that are created dynamically based on the introspection data passed to the proxy object that made this proxy interface.

A *method call* takes this form:

```
def callback(error: Exception, result: list(Any)):
    pass

interface.call_[METHOD](*args, callback)
result = interface.call_[METHOD]_sync(*args)
```

Where `METHOD` is the name of the method converted to snake case.

To call a method, provide `*args` that correspond to the *in args* of the introspection method definition.

To *asynchronously* call a method, provide a callback that takes an error as the first argument and a list as the second argument. If the call completed successfully, `error` will be `None`. If the service returns an error, it will be a *DBusError* with information about the error returned from the bus. The result will be a list of values that correspond to the *out args* of the introspection method definition.

To *synchronously* call a method, use the `call_[METHOD]_sync()` form. The `result` corresponds to the *out arg* of the introspection method definition. If the method has more than one otu arg, they are returned within a `list`.

To *listen to a signal* use this form:

```
interface.on_[SIGNAL](callback)
```

To *stop listening to a signal* use this form:

```
interface.off_[SIGNAL](callback)
```

Where `SIGNAL` is the name of the signal converted to snake case.

DBus signals are exposed with an event-callback interface. The provided `callback` will be called when the signal is emitted with arguments that correspond to the *out args* of the interface signal definition.

To *get or set a property* use this form:

```
def get_callback(error: Exception, value: Any):
    pass

def set_callback(error: Exception)
    pass

interface.get_[PROPERTY](get_callback)
value: Any = interface.get_[PROPERTY]_sync()

interface.set_[PROPERTY](set_callback)
interface.set_[PROPERTY]_sync(value)
```

Where `PROPERTY` is the name of the property converted to snake case.

The `value` must correspond to the type of the property in the interface definition.

To asynchronously get or set a property, provide a callback that takes an `Exception` as the first argument. If the call completed successfully, `error` will be `None`. If the service returns an error, it will be a *DBusError* with information about the error returned from the bus.

If the service returns an error for a synchronous DBus call, a *DBusError* will be raised with information about the error.

DBus interfaces are defined with an XML-based introspection data format which is exposed over the standard org.freedesktop.DBus.Introspectable interface. Calling the `Introspect` at a particular object path may return XML data similar to this:

```
<!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"/>
    </method>
    <method name="Bazify">
      <arg name="bar" type="(iiu)" direction="in"/>
      <arg name="bar" type="v" direction="out"/>
    </method>
    <method name="Mogrify">
```

(continues on next page)

(continued from previous page)

```
      <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>
```

The object at this path (a `node`) may contain interfaces and child nodes. Nodes like this are represented in the library by a *ProxyObject*. The interfaces contained in the nodes are represented by a *ProxyInterface*. The proxy interface exposes the methods, signals, and properties specified by the interface definition.

The proxy object is obtained by the *MessageBus* through the *get_proxy_object()* method. This method takes the name of the client to send messages to, the path exported by that client that is expected to export the node, and the XML introspection data. If you can, it is recommended to include the XML in your project and pass it to that method as a string. But you may also use the *introspect()* method of the message bus to get this data dynamically at runtime.

Once you have a proxy object, use the *get_proxy_interface()* method to create an interface passing the name of the interface to get. Each message bus has its own implementation of the proxy interface which behaves slightly differently. This is an example of how to use a proxy interface for the asyncio *MessageBus*.

If any file descriptors are sent or received (DBus type `h`), the variable refers to the file descriptor itself. You are responsible for closing any file descriptors sent or received by the bus. You must set the `negotiate_unix_fd` flag to `True` in the `MessageBus` constructor to use unix file descriptors.

example

```
from dbus_next.aio import MessageBus
from dbus_next import Variant

bus = await MessageBus().connect()

with open('introspection.xml', 'r') as f:
    introspection = f.read()

# alternatively, get the data dynamically:
# introspection = await bus.introspect('com.example.name',
#                                      '/com/example/sample_object0')

proxy_object = bus.get_proxy_object('com.example.name',
                                    '/com/example/sample_object0',
                                    introspection)

interface = proxy_object.get_interface('com.example.SampleInterface0')

# Use call_[METHOD] in snake case to call methods, passing the
# in args and receiving the out args. The `baz` returned will
# be type 'a{us}' which translates to a Python dict with `int`
# keys and `str` values.
baz = await interface.call_frobate(5, 'hello')

# `bar` will be a Variant.
bar = await interface.call_bazify([-5, 5, 5])
```

(continues on next page)

(continued from previous page)

```
await interface.call_mogrify([5, 5, [ Variant('s', 'foo') ])

# Listen to signals by defining a callback that takes the args
# specified by the signal definition and registering it on the
# interface with on_[SIGNAL] in snake case.

def changed_notify(new_value):
    print(f'The new value is: {new_value}')

interface.on_changed(changed_notify)

# Use get_[PROPERTY] and set_[PROPERTY] with the property in
# snake case to get and set the property.

bar_value = await interface.get_bar()

await interface.set_bar(105)

await bus.wait_for_disconnect()
```

CHAPTER

THREE

THE HIGH LEVEL SERVICE

3.1 ServiceInterface

class dbus_next.service.**ServiceInterface**(*name: str*)

An abstract class that can be extended by the user to define DBus services.

Instances of *ServiceInterface* can be exported on a path of the bus with the *export* method of a *MessageBus*.

Use the *@method*, *@dbus_property*, and *@signal* decorators to mark class methods as DBus methods, properties, and signals respectively.

Variables **name** (*str*) – The name of this interface as it appears to clients. Must be a valid interface name.

emit_properties_changed(*changed_properties: Dict[str, Any]*, *invalidated_properties: List[str] = []*)

Emit the org.freedesktop.DBus.Properties.PropertiesChanged signal.

This signal is intended to be used to alert clients when a property of the interface has changed.

Parameters

- **changed_properties** (*dict(str, Any)*) – The keys must be the names of properties exposed by this bus. The values must be valid for the signature of those properties.
- **invalidated_properties** (*list(str)*) – A list of names of properties that are now invalid (presumably for clients who cache the value).

introspect() → *dbus_next.introspection.Interface*

Get introspection information for this interface.

This might be useful for creating clients for the interface or examining the introspection output of an interface.

Returns The introspection data for the interface.

Return type *dbus_next.introspection.Interface*

@dbus_next.service.**dbus_property**(*access: dbus_next.constants.PropertyAccess = <PropertyAccess.READWRITE: 'readwrite'>*, *name: Optional[str] = None*, *disabled: bool = False*)

A decorator to mark a class method of a *ServiceInterface* to be a DBus property.

The class method must be a Python getter method with a return annotation that is a signature string of a single complete DBus type. When a client gets the property through the org.freedesktop.DBus.Properties interface, the getter will be called and the resulting value will be returned to the client.

If the property is writable, it must have a setter method that takes a single parameter that is annotated with the same signature. When a client sets the property through the `org.freedesktop.DBus.Properties` interface, the setter will be called with the value from the calling client.

The parameters of the getter and the setter must conform to the dbus-next type system. The getter or the setter may raise a *DBusError* to return an error to the client.

Parameters

- **name** (*str*) – The name that DBus clients will use to interact with this property on the bus.
- **disabled** (*bool*) – If set to true, the property will not be visible to clients.

Example

```
@dbus_property()
def string_prop(self) -> 's':
    return self._string_prop

@string_prop.setter
def string_prop(self, val: 's'):
    self._string_prop = val
```

`@dbus_next.service.`**method**(*name: Optional[str] = None*, *disabled: bool = False*)

A decorator to mark a class method of a *ServiceInterface* to be a DBus service method.

The parameters and return value must each be annotated with a signature string of a single complete DBus type.

This class method will be called when a client calls the method on the DBus interface. The parameters given to the function come from the calling client and will conform to the dbus-next type system. The parameters returned will be returned to the calling client and must conform to the dbus-next type system. If multiple parameters are returned, they must be contained within a `list`.

The decorated method may raise a *DBusError* to return an error to the client.

Parameters

- **name** (*str*) – The member name that DBus clients will use to call this method. Defaults to the name of the class method.
- **disabled** (*bool*) – If set to true, the method will not be visible to clients.

Example

```
@method()
def echo(self, val: 's') -> 's':
    return val

@method()
def echo_two(self, val1: 's', val2: 'u') -> 'su':
    return [val1, val2]
```

`@dbus_next.service.`**signal**(*name: Optional[str] = None*, *disabled: bool = False*)

A decorator to mark a class method of a *ServiceInterface* to be a DBus signal.

The signal is broadcast on the bus when the decorated class method is called by the user.

If the signal has an out argument, the class method must have a return type annotation with a signature string of a single complete DBus type and the return value of the class method must conform to the dbus-next type system. If the signal has multiple out arguments, they must be returned within a `list`.

Parameters

- **name** (*str*) – The member name that will be used for this signal. Defaults to the name of the class method.
- **disabled** (*bool*) – If set to true, the signal will not be visible to clients.

Example

```
@signal()
def string_signal(self, val) -> 's':
    return val

@signal()
def two_strings_signal(self, val1, val2) -> 'ss':
    return [val1, val2]
```

The high level service interface provides everything you need to export interfaces on the bus. When you export an interface on your *MessageBus*, clients can send you messages to call methods, get and set properties, and listen to your signals.

If you're exposing a service for general use, you can request a well-known name for your connection with *MessageBus.request_name()* so users have a predictable name to use to send messages your client.

Services are defined by subclassing *ServiceInterface* and defininining members as methods on the class with the decorator methods *@method()*, *@dbus_property()*, and *@signal()*. The parameters of the decorated class methods must be annotated with DBus type strings to indicate the types of values they expect. See the documentation on the type system for more information on how DBus types are mapped to Python values with signature strings. The decorator methods themselves take arguments that affect how the member is exported on the bus, such as the name of the member or the access permissions of a property.

A class method decorated with `@method()` will be called when a client calls the method over DBus. The parameters given to the class method will be provided by the calling client and will conform to the parameter type annotations. The value returned by the class method will be returned to the client and must conform to the return type annotation specified by the user. If the return annotation specifies more than one type, the values must be returned in a `list`. When `aio.MessageBus` is used, methods can be coroutines.

A class method decorated with `@dbus_property()` will be exposed as a DBus property getter. This decoration works the same as a standard Python `@property`. The getter will be called when a client gets the property through the standard properties interface with `org.freedesktop.DBus.Properties.Get`. Define a property setter with `@method_name.setter` taking the new value as a parameter. The setter will be called when the client sets the property through `org.freedesktop.DBus.Properties.Set`. When `aio.MessageBus` is used, property getters and setters can be coroutines, although this will cause some functionality of the Python `@property` annotation to be lost.

A class method decorated with `@signal()` will be exposed as a DBus signal. The value returned by the class method will be emitted as a signal and broadcast to clients who are listening to the signal. The returned value must conform to the return annotation of the class method as a DBus signature string. If the signal has more than one argument, they must be returned within a `list`.

A class method decorated with `@method()` or `@dbus_property()` may throw a *DBusError* to return a detailed error to the client if something goes wrong.

After the service interface is defined, call *MessageBus.export()* on a connected message bus and the service will be made available on the given object path.

If any file descriptors are sent or received (DBus type `h`), the variable refers to the file descriptor itself. You are responsible for closing any file descriptors sent or received by the bus. You must set the `negotiate_unix_fd` flag to `True` in the `MessageBus` constructor to use unix file descriptors.

example

```
from dbus_next.aio import MessageBus
from dbus_next.service import (ServiceInterface,
                               method, dbus_property, signal)
from dbus_next import Variant, DBusError

import asyncio

class ExampleInterface(ServiceInterface):
    def __init__(self):
        super().__init__('com.example.SampleInterface0')
        self._bar = 105

    @method()
    def Frobate(self, foo: 'i', bar: 's') -> 'a{us}':
        print(f'called Frobate with foo={foo} and bar={bar}')

        return {
            1: 'one',
            2: 'two'
        }

    @method()
    async def Bazify(self, bar: '(iiu)') -> 'vv':
        print(f'called Bazify with bar={bar}')

        return [Variant('s', 'example'), Variant('s', 'bazify')]

    @method()
    def Mogrify(self, bar: '(iiav)'):
        raise DBusError('com.example.error.CannotMogrify',
                        'it is not possible to mogrify')

    @signal()
    def Changed(self) -> 'b':
        return True

    @dbus_property()
    def Bar(self) -> 'y':
        return self._bar

    @Bar.setter
    def Bar(self, val: 'y'):
        if self._bar == val:
            return

        self._bar = val

        self.emit_properties_changed({'Bar': self._bar})

async def main():
    bus = await MessageBus().connect()
    interface = ExampleInterface()
    bus.export('/com/example/sample0', interface)
    await bus.request_name('com.example.name')

    # emit the changed signal after two seconds.
    await asyncio.sleep(2)
```

(continues on next page)

(continued from previous page)

```
    interface.Changed()

    await bus.wait_for_disconnect()

asyncio.get_event_loop().run_until_complete(main())
```

CHAPTER

FOUR

THE LOW LEVEL INTERFACE

4.1 Message

class dbus_next.**Message**(*destination: Optional[str] = None*, *path: Optional[str] = None*, *interface: Optional[str] = None*, *member: Optional[str] = None*, *message_type: dbus_next.constants.MessageType = <MessageType.METHOD_CALL: 1>*, *flags: dbus_next.constants.MessageFlag = <MessageFlag.NONE: 0>*, *error_name: Optional[str] = None*, *reply_serial: Optional[int] = None*, *sender: Optional[str] = None*, *unix_fds: List[int] = []*, *signature: str = ''*, *body: List[Any] = []*, *serial: int = 0*)

A class for sending and receiving messages through the *MessageBus* with the low-level api.

A `Message` can be constructed by the user to send over the message bus. When messages are received, such as from method calls or signal emissions, they will use this class as well.

Variables

- **destination** (*str*) – The address of the client for which this message is intended.
- **path** (*str*) – The intended object path exported on the destination bus.
- **interface** (*str*) – The intended interface on the object path.
- **member** (*str*) – The intended member on the interface.
- **message_type** (*MessageType*) – The type of this message. A method call, signal, method return, or error.
- **flags** (*MessageFlag*) – Flags that affect the behavior of this message.
- **error_name** (*str*) – If this message is an error, the name of this error. Must be a valid interface name.
- **reply_serial** (*int*) – If this is a return type, the serial this message is in reply to.
- **sender** (*str*) – The address of the sender of this message. Will be a unique name.
- **unix_fds** (*list (int)*) – A list of unix fds that were sent in the header of this message.
- **signature** (*str*) – The signature of the body of this message.
- **signature_tree** (*SignatureTree*) – The signature parsed as a signature tree.
- **body** (*list (Any)*) – The body of this message. Must match the signature.
- **serial** (*int*) – The serial of the message. Will be automatically set during message sending if not present. Use the `new_serial()` method of the bus to generate a serial.

Raises

- *InvalidMessageError* - If the message is malformed or missing fields for the message type.
- *InvalidSignatureError* - If the given signature is not valid.
- *InvalidObjectPathError* - If `path` is not a valid object path.
- *InvalidBusNameError* - If `destination` is not a valid bus name.
- *InvalidMemberNameError* - If `member` is not a valid member name.
- *InvalidInterfaceNameError* - If `error_name` or `interface` is not a valid interface name.

static new_error(*msg: dbus_next.message.Message*, *error_name: str*, *error_text: str*) → dbus_next.message.Message

A convenience constructor to create an error message in reply to the given message.

Parameters

- **msg** (*Message*) – The message this error is in reply to.
- **error_name** (*str*) – The name of this error. Must be a valid interface name.
- **error_text** – Human-readable text for the error.

Returns The error message.

Return type *Message*

Raises

- *InvalidInterfaceNameError* - If the error_name is not a valid interface name.

static new_method_return(*msg: dbus_next.message.Message*, *signature: str = ''*, *body: List[Any] = []*, *unix_fds: List[int] = []*) → dbus_next.message.Message

A convenience constructor to create a method return to the given method call message.

Parameters

- **msg** (*Message*) – The method call message this is a reply to.
- **signature** (*str*) – The signature for the message body.
- **body** (*list(int)*) – The body of this message. Must match the signature.
- **unix_fds** – List integer file descriptors to send with this message.

Returns The method return message

Return type *Message*

Raises

- *InvalidSignatureError* - If the signature is not a valid signature.

static new_signal(*path: str*, *interface: str*, *member: str*, *signature: str = ''*, *body: Optional[List[Any]] = None*, *unix_fds: Optional[List[int]] = None*) → dbus_next.message.Message

A convenience constructor to create a new signal message.

Parameters

- **path** (*str*) – The path of this signal.
- **interface** (*str*) – The interface of this signal.
- **member** (*str*) – The member name of this signal.

- **signature** (*str*) – The signature of the signal body.
- **body** (*list(int)*) – The body of this signal message.
- **unix_fds** – List integer file descriptors to send with this message.

Returns The signal message.

Return type *Message*

Raises

- *InvalidSignatureError* - If the signature is not a valid signature.
- *InvalidObjectPathError* - If `path` is not a valid object path.
- *InvalidInterfaceNameError* - If `interface` is not a valid interface name.
- *InvalidMemberNameError* - If `member` is not a valid member name.

The low-level interface allows you to work with messages directly through the *MessageBus* with the *Message* class. This might be useful in the following cases:

- Implementing an application that works with DBus directly like `dbus-send(1)` or `dbus-monitor(1)`.
- Creating a new implementation of the *BaseMessageBus*.
- Creating clients or services that use an alternative to the standard DBus interfaces.

The primary methods and classes of the low-level interface are:

- *Message*
- *MessageBus.send()*
- *MessageBus.add_message_handler()*
- *MessageBus.remove_message_handler()*
- *MessageBus.next_serial()*
- *aio.MessageBus.call()*
- *glib.MessageBus.call()*
- *glib.MessageBus.call_sync()*

Mixed use of the low and high level interfaces on the same bus connection is not recommended.

example Call a standard interface

```
bus = await MessageBus().connect()

msg = Message(destination='org.freedesktop.DBus',
              path='/org/freedesktop/DBus',
              interface='org.freedesktop.DBus',
              member='ListNames',
              serial=bus.next_serial())

reply = await bus.call(msg)

assert reply.message_type == MessageType.METHOD_RETURN

print(reply.body[0])
```

example A custom method handler. Note that to receive these messages, you must add a match rule for the types of messages you want to receive.

```
bus = await MessageBus().connect()

reply = await bus.call(
    Message(destination='org.freedesktop.DBus',
            path='/org/freedesktop/DBus',
            member='AddMatch',
            signature='s',
            body=["member='MyMember', interface='com.test.interface'"]))

assert reply.message_type == MessageType.METHOD_RETURN

def message_handler(msg):
    if msg.interface == 'com.test.interface' and msg.member == 'MyMember':
        return Message.new_method_return(msg, 's', ['got it'])

bus.add_message_handler(message_handler)

await bus.wait_for_disconnect()
```

example Emit a signal

```
bus = await MessageBus().connect()

await bus.send(Message.new_signal('/com/test/path',
                                  'com.test.interface',
                                  'SomeSignal',
                                  's', ['a signal']))
```

example Send a file descriptor. The message format will be the same when received on the client side. You are responsible for closing any file descriptor that is sent or received by the bus. You must set the `negotiate_unix_fd` flag to `True` in the `MessageBus` constructor to use unix file descriptors.

```
bus = await MessageBus().connect(negotiate_unix_fd=True)

fd = os.open('/dev/null', os.O_RDONLY)

msg = Message(destination='org.test.destination',
              path='/org/test/destination',
              interface='org.test.interface',
              member='TestMember',
              signature='h',
              body=[0],
              unix_fds=[fd])

await bus.send(msg)
```

CHAPTER

FIVE

THE MESSAGE BUS

5.1 BaseMessageBus

class dbus_next.message_bus.**BaseMessageBus**(*bus_address: Optional[str] = None*, *bus_type: dbus_next.constants.BusType = <BusType.SESSION: 1>*, *ProxyObject: Optional[Type[dbus_next.proxy_object.BaseProxyObject]] = None*)

An abstract class to manage a connection to a DBus message bus.

The message bus class is the entry point into all the features of the library. It sets up a connection to the DBus daemon and exposes an interface to send and receive messages and expose services.

This class is not meant to be used directly by users. For more information, see the documentation for the implementation of the message bus you plan to use.

Parameters

- **bus_type** (*BusType*) – The type of bus to connect to. Affects the search path for the bus address.
- **bus_address** (*str*) – A specific bus address to connect to. Should not be used under normal circumstances.
- **ProxyObject** (Type[*BaseProxyObject*]) – The proxy object implementation for this message bus. Must be passed in by an implementation that supports the high-level client.

Variables

- **unique_name** (*str*) – The unique name of the message bus connection. It will be `None` until the message bus connects.
- **connected** (*bool*) – True if this message bus is expected to be able to send and receive messages.

add_message_handler(*handler: Callable[[dbus_next.message.Message], Optional[Union[dbus_next.message.Message, bool]]]*)

Add a custom message handler for incoming messages.

The handler should be a callable that takes a *Message*. If the message is a method call, you may return another Message as a reply and it will be marked as handled. You may also return `True` to mark the message as handled without sending a reply.

Parameters **handler** (`Callable` or None) – A handler that will be run for every message the bus connection received.

disconnect()

Disconnect the message bus by closing the underlying connection asynchronously.

All pending and future calls will error with a connection error.

export (*path: str*, *interface:* dbus_next.service.ServiceInterface)
Export the service interface on this message bus to make it available to other clients.

Parameters

- **path** (*str*) – The object path to export this interface on.
- **interface** (*ServiceInterface*) – The service interface to export.

Raises

- *InvalidObjectPathError* - If the given object path is not valid.
- ValueError - If an interface with this name is already exported on the message bus at this path

get_proxy_object (*bus_name: str*, *path: str*, *introspection: Union[*dbus_next.introspection.Node*, str, xml.etree.ElementTree.Element]*) → *dbus_next.proxy_object.BaseProxyObject*
Get a proxy object for the path exported on the bus that owns the name. The object is expected to export the interfaces and nodes specified in the introspection data.

This is the entry point into the high-level client.

Parameters

- **bus_name** (*str*) – The name on the bus to get the proxy object for.
- **path** (*str*) – The path on the client for the proxy object.
- **introspection** (*Node* or str or ElementTree) – XML introspection data used to build the interfaces on the proxy object.

Returns A proxy object for the given path on the given name.

Return type *BaseProxyObject*

Raises

- *InvalidBusNameError* - If the given bus name is not valid.
- *InvalidObjectPathError* - If the given object path is not valid.
- *InvalidIntrospectionError* - If the introspection data for the node is not valid.

introspect (*bus_name: str*, *path: str*, *callback: Callable[[Optional[*dbus_next.introspection.Node*], Optional[Exception]], None]*)
Get introspection data for the node at the given path from the given bus name.

Calls the standard org.freedesktop.DBus.Introspectable.Introspect on the bus for the path.

Parameters

- **bus_name** (*str*) – The name to introspect.
- **path** (*str*) – The path to introspect.
- **callback** (Callable) – A callback that will be called with the introspection data as a *Node*.

Raises

- *InvalidObjectPathError* - If the given object path is not valid.
- *InvalidBusNameError* - If the given bus name is not valid.

next_serial() → int
Get the next serial for this bus. This can be used as the `serial` attribute of a *Message* to manually handle the serial of messages.

Returns The next serial for the bus.

Return type int

release_name(*name: str*, *callback: Optional[Callable[[Optional[dbus_next.constants.ReleaseNameReply], Optional[Exception]], None]] = None*)
Request that this message bus release the given name.

Parameters

- **name** (*str*) – The name to release.
- **callback** (`Callable`) – A callback that will be called with the reply of the release request as a *ReleaseNameReply*.

Raises

- *InvalidBusNameError* - If the given bus name is not valid.

remove_message_handler(*handler: Callable[[dbus_next.message.Message], Optional[Union[dbus_next.message.Message, bool]]]*)
Remove a message handler that was previously added by *add_message_handler()*.

Parameters **handler** (`Callable`) – A message handler.

request_name(*name: str*, *flags: dbus_next.constants.NameFlag = <NameFlag.NONE: 0>*, *callback: Optional[Callable[[Optional[dbus_next.constants.RequestNameReply], Optional[Exception]], None]] = None*)
Request that this message bus owns the given name.

Parameters

- **name** (*str*) – The name to request.
- **flags** (*NameFlag*) – Name flags that affect the behavior of the name request.
- **callback** (`Callable`) – A callback that will be called with the reply of the request as a *RequestNameReply*.

Raises

- *InvalidBusNameError* - If the given bus name is not valid.

send(*msg: dbus_next.message.Message*) → None
Asynchronously send a message on the message bus.

Parameters **msg** (*Message*) – The message to send.

unexport(*path: str*, *interface: Optional[Union[*dbus_next.service.ServiceInterface*, str]] = None*)
Unexport the path or service interface to make it no longer available to clients.

Parameters

- **path** (*str*) – The object path to unexport.
- **interface** (*ServiceInterface* or str or None) – The interface instance or the name of the interface to unexport. If `None`, unexport every interface on the path.

Raises

- *InvalidObjectPathError* - If the given object path is not valid.

5.2 aio.MessageBus

class dbus_next.aio.**MessageBus**(*bus_address: Optional[str] = None, bus_type: dbus_next.constants.BusType = <BusType.SESSION: 1>, auth: Optional[dbus_next.auth.Authenticator] = None, negotiate_unix_fd=False*)

Bases: *dbus_next.message_bus.BaseMessageBus*

The message bus implementation for use with asyncio.

The message bus class is the entry point into all the features of the library. It sets up a connection to the DBus daemon and exposes an interface to send and receive messages and expose services.

You must call *connect()* before using this message bus.

Parameters

- **bus_type** (*BusType*) – The type of bus to connect to. Affects the search path for the bus address.
- **bus_address** – A specific bus address to connect to. Should not be used under normal circumstances.
- **auth** (*Authenticator*) – The authenticator to use, defaults to an instance of *AuthExternal*.
- **negotiate_unix_fd** (*bool*) – Allow the bus to send and receive Unix file descriptors (DBus type 'h'). This must be supported by the transport.

Variables

- **unique_name** (*str*) – The unique name of the message bus connection. It will be `None` until the message bus connects.
- **connected** (*bool*) – True if this message bus is expected to be able to send and receive messages.

add_message_handler(*handler: Callable[[dbus_next.message.Message], Optional[Union[dbus_next.message.Message, bool]]]*)

Add a custom message handler for incoming messages.

The handler should be a callable that takes a *Message*. If the message is a method call, you may return another Message as a reply and it will be marked as handled. You may also return `True` to mark the message as handled without sending a reply.

Parameters **handler** (`Callable` or None) – A handler that will be run for every message the bus connection received.

coroutine call(*msg: dbus_next.message.Message*) → Optional[dbus_next.message.Message]

Send a method call and wait for a reply from the DBus daemon.

Parameters **msg** (*Message*) – The method call message to send.

Returns A message in reply to the message sent. If the message does not expect a reply based on the message flags or type, returns `None` after the message is sent.

Return type *Message* or `None` if no reply is expected.

Raises

- `Exception` - If a connection error occurred.

coroutine connect() → dbus_next.aio.message_bus.MessageBus

Connect this message bus to the DBus daemon.

This method must be called before the message bus can be used.

Returns This message bus for convenience.

Return type *MessageBus*

Raises

- *AuthError* - If authorization to the DBus daemon failed.
- `Exception` - If there was a connection error.

disconnect()
Disconnect the message bus by closing the underlying connection asynchronously.

All pending and future calls will error with a connection error.

export(*path: str*, *interface:* dbus_next.service.ServiceInterface)
Export the service interface on this message bus to make it available to other clients.

Parameters

- **path** (*str*) – The object path to export this interface on.
- **interface** (*ServiceInterface*) – The service interface to export.

Raises

- *InvalidObjectPathError* - If the given object path is not valid.
- `ValueError` - If an interface with this name is already exported on the message bus at this path

get_proxy_object(*bus_name: str*, *path: str*, *introspection:* dbus_next.introspection.Node) → dbus_next.aio.proxy_object.ProxyObject
Get a proxy object for the path exported on the bus that owns the name. The object is expected to export the interfaces and nodes specified in the introspection data.

This is the entry point into the high-level client.

Parameters

- **bus_name** (*str*) – The name on the bus to get the proxy object for.
- **path** (*str*) – The path on the client for the proxy object.
- **introspection** (*Node* or str or `ElementTree`) – XML introspection data used to build the interfaces on the proxy object.

Returns A proxy object for the given path on the given name.

Return type *BaseProxyObject*

Raises

- *InvalidBusNameError* - If the given bus name is not valid.
- *InvalidObjectPathError* - If the given object path is not valid.
- *InvalidIntrospectionError* - If the introspection data for the node is not valid.

coroutine introspect(*bus_name: str*, *path: str*, *timeout: float = 30.0*) → *dbus_next.introspection.Node*
Get introspection data for the node at the given path from the given bus name.

Calls the standard `org.freedesktop.DBus.Introspectable.Introspect` on the bus for the path.

Parameters

- **bus_name** (*str*) – The name to introspect.
- **path** (*str*) – The path to introspect.
- **timeout** (*float*) – The timeout to introspect.

Returns The introspection data for the name at the path.

Return type *Node*

Raises

- *InvalidObjectPathError* - If the given object path is not valid.
- *InvalidBusNameError* - If the given bus name is not valid.
- *DBusError* - If the service threw an error for the method call or returned an invalid result.
- `Exception` - If a connection error occurred.
- `asyncio.TimeoutError` - Waited for future but time run out.

next_serial() → int

Get the next serial for this bus. This can be used as the `serial` attribute of a *Message* to manually handle the serial of messages.

Returns The next serial for the bus.

Return type int

coroutine release_name(*name: str*) → dbus_next.constants.ReleaseNameReply

Request that this message bus release the given name.

Parameters **name** (*str*) – The name to release.

Returns The reply to the release request.

Return type *ReleaseNameReply*

Raises

- *InvalidBusNameError* - If the given bus name is not valid.
- *DBusError* - If the service threw an error for the method call or returned an invalid result.
- `Exception` - If a connection error occurred.

remove_message_handler(*handler: Callable[[dbus_next.message.Message], Optional[Union[dbus_next.message.Message, bool]]]*)

Remove a message handler that was previously added by *add_message_handler()*.

Parameters **handler** (`Callable`) – A message handler.

coroutine request_name(*name: str, flags: dbus_next.constants.NameFlag = <NameFlag.NONE: 0>*) → dbus_next.constants.RequestNameReply

Request that this message bus owns the given name.

Parameters

- **name** (*str*) – The name to request.
- **flags** (*NameFlag*) – Name flags that affect the behavior of the name request.

Returns The reply to the name request.

Return type *RequestNameReply*

Raises

- *InvalidBusNameError* - If the given bus name is not valid.
- *DBusError* - If the service threw an error for the method call or returned an invalid result.
- `Exception` - If a connection error occurred.

send(*msg: dbus_next.message.Message*)
Asynchronously send a message on the message bus.

Note: This method may change to a couroutine function in the 1.0 release of the library.

Parameters **msg** (*Message*) – The message to send.

Returns A future that resolves when the message is sent or a connection error occurs.

Return type `Future`

unexport(*path: str*, *interface: Optional[Union[*dbus_next.service.ServiceInterface*, str]] = None*)
Unexport the path or service interface to make it no longer available to clients.

Parameters

- **path** (*str*) – The object path to unexport.
- **interface** (*ServiceInterface* or str or None) – The interface instance or the name of the interface to unexport. If `None`, unexport every interface on the path.

Raises

- *InvalidObjectPathError* - If the given object path is not valid.

coroutine wait_for_disconnect()
Wait for the message bus to disconnect.

Returns `None` when the message bus has disconnected.

Return type `None`

Raises

- `Exception` - If connection was terminated unexpectedly or an internal error occurred in the library.

5.3 glib.MessageBus

class dbus_next.glib.**MessageBus**(*bus_address: Optional[str] = None*, *bus_type: dbus_next.constants.BusType = <BusType.SESSION: 1>*, *auth: Optional[dbus_next.auth.Authenticator] = None*)
Bases: *dbus_next.message_bus.BaseMessageBus*

The message bus implementation for use with the GLib main loop.

The message bus class is the entry point into all the features of the library. It sets up a connection to the DBus daemon and exposes an interface to send and receive messages and expose services.

You must call *connect()* or *connect_sync()* before using this message bus.

Parameters

- **bus_type** (*BusType*) – The type of bus to connect to. Affects the search path for the bus address.
- **bus_address** – A specific bus address to connect to. Should not be used under normal circumstances.
- **auth** (*Authenticator*) – The authenticator to use, defaults to an instance of *AuthExternal*.

Variables

- **connected** (*bool*) – True if this message bus is expected to be able to send and receive messages.
- **unique_name** (*str*) – The unique name of the message bus connection. It will be `None` until the message bus connects.

add_message_handler(*handler: Callable[[dbus_next.message.Message], Optional[Union[dbus_next.message.Message, bool]]]*)

Add a custom message handler for incoming messages.

The handler should be a callable that takes a *Message*. If the message is a method call, you may return another Message as a reply and it will be marked as handled. You may also return `True` to mark the message as handled without sending a reply.

Parameters **handler** (`Callable` or None) – A handler that will be run for every message the bus connection received.

call(*msg: dbus_next.message.Message*, *reply_notify: Optional[Callable[[Optional[dbus_next.message.Message], Optional[Exception]], None]] = None*)

Send a method call and asynchronously wait for a reply from the DBus daemon.

Parameters

- **msg** (*Message*) – The method call message to send.
- **reply_notify** (*Callable*) – A callback that will be called with the reply to this message. May return an `Exception` on connection errors.

call_sync(*msg: dbus_next.message.Message*) → Optional[dbus_next.message.Message]

Send a method call and synchronously wait for a reply from the DBus daemon.

Parameters **msg** (*Message*) – The method call message to send.

Returns A message in reply to the message sent. If the message does not expect a reply based on the message flags or type, returns `None` immediately.

Return type *Message*

Raises

- *DBusError* - If the service threw an error for the method call or returned an invalid result.
- `Exception` - If a connection error occurred.

connect(*connect_notify: Optional[Callable[[dbus_next.glib.message_bus.MessageBus, Optional[Exception]], None]] = None*)

Connect this message bus to the DBus daemon.

This method or the synchronous version must be called before the message bus can be used.

Parameters **connect_notify** – A callback that will be called with this message bus. May return an `Exception` on connection errors or *AuthError* on authorization errors.

connect_sync() → dbus_next.glib.message_bus.MessageBus
Connect this message bus to the DBus daemon.

This method or the asynchronous version must be called before the message bus can be used.

Returns This message bus for convenience.

Return type *MessageBus*

Raises

- *AuthError* - If authorization to the DBus daemon failed.
- Exception - If there was a connection error.

disconnect()
Disconnect the message bus by closing the underlying connection asynchronously.

All pending and future calls will error with a connection error.

export(*path: str*, *interface:* dbus_next.service.ServiceInterface)
Export the service interface on this message bus to make it available to other clients.

Parameters

- **path** (*str*) – The object path to export this interface on.
- **interface** (*ServiceInterface*) – The service interface to export.

Raises

- *InvalidObjectPathError* - If the given object path is not valid.
- ValueError - If an interface with this name is already exported on the message bus at this path

get_proxy_object(*bus_name: str*, *path: str*, *introspection:* dbus_next.introspection.Node) → dbus_next.glib.proxy_object.ProxyObject
Get a proxy object for the path exported on the bus that owns the name. The object is expected to export the interfaces and nodes specified in the introspection data.

This is the entry point into the high-level client.

Parameters

- **bus_name** (*str*) – The name on the bus to get the proxy object for.
- **path** (*str*) – The path on the client for the proxy object.
- **introspection** (*Node* or str or ElementTree) – XML introspection data used to build the interfaces on the proxy object.

Returns A proxy object for the given path on the given name.

Return type *BaseProxyObject*

Raises

- *InvalidBusNameError* - If the given bus name is not valid.
- *InvalidObjectPathError* - If the given object path is not valid.
- *InvalidIntrospectionError* - If the introspection data for the node is not valid.

introspect(*bus_name: str*, *path: str*, *callback: Callable[[Optional[*dbus_next.introspection.Node*], Optional[Exception]], None]*)
Get introspection data for the node at the given path from the given bus name.

Calls the standard `org.freedesktop.DBus.Introspectable.Introspect` on the bus for the path.

Parameters

- **bus_name** (*str*) – The name to introspect.
- **path** (*str*) – The path to introspect.
- **callback** (`Callable`) – A callback that will be called with the introspection data as a *Node*.

Raises

- *InvalidObjectPathError* - If the given object path is not valid.
- *InvalidBusNameError* - If the given bus name is not valid.

introspect_sync(*bus_name: str*, *path: str*) → *dbus_next.introspection.Node*

Get introspection data for the node at the given path from the given bus name.

Calls the standard `org.freedesktop.DBus.Introspectable.Introspect` on the bus for the path.

Parameters

- **bus_name** (*str*) – The name to introspect.
- **path** (*str*) – The path to introspect.

Returns The introspection data for the name at the path.

Return type *Node*

Raises

- *InvalidObjectPathError* - If the given object path is not valid.
- *InvalidBusNameError* - If the given bus name is not valid.
- *DBusError* - If the service threw an error for the method call or returned an invalid result.
- `Exception` - If a connection error occurred.

next_serial() → int

Get the next serial for this bus. This can be used as the `serial` attribute of a *Message* to manually handle the serial of messages.

Returns The next serial for the bus.

Return type int

release_name(*name: str*, *callback: Optional[Callable[[Optional[dbus_next.constants.ReleaseNameReply], Optional[Exception]], None]] = None*)

Request that this message bus release the given name.

Parameters

- **name** (*str*) – The name to release.
- **callback** (`Callable`) – A callback that will be called with the reply of the release request as a *ReleaseNameReply*.

Raises

- *InvalidBusNameError* - If the given bus name is not valid.

release_name_sync(*name: str*) → dbus_next.constants.ReleaseNameReply
Request that this message bus release the given name.

Parameters **name** (*str*) – The name to release.

Returns The reply to the release request.

Return type *ReleaseNameReply*

Raises

- *InvalidBusNameError* - If the given bus name is not valid.
- *DBusError* - If the service threw an error for the method call or returned an invalid result.
- `Exception` - If a connection error occurred.

remove_message_handler(*handler: Callable[[dbus_next.message.Message], Optional[Union[dbus_next.message.Message, bool]]]*)
Remove a message handler that was previously added by *add_message_handler()*.

Parameters **handler** (`Callable`) – A message handler.

request_name(*name: str, flags: dbus_next.constants.NameFlag = <NameFlag.NONE: 0>, callback: Optional[Callable[[Optional[dbus_next.constants.RequestNameReply], Optional[Exception]], None]] = None*)
Request that this message bus owns the given name.

Parameters

- **name** (*str*) – The name to request.
- **flags** (*NameFlag*) – Name flags that affect the behavior of the name request.
- **callback** (`Callable`) – A callback that will be called with the reply of the request as a *RequestNameReply*.

Raises

- *InvalidBusNameError* - If the given bus name is not valid.

request_name_sync(*name: str, flags: dbus_next.constants.NameFlag = <NameFlag.NONE: 0>*) → dbus_next.constants.RequestNameReply
Request that this message bus owns the given name.

Parameters

- **name** (*str*) – The name to request.
- **flags** (*NameFlag*) – Name flags that affect the behavior of the name request.

Returns The reply to the name request.

Return type *RequestNameReply*

Raises

- *InvalidBusNameError* - If the given bus name is not valid.
- *DBusError* - If the service threw an error for the method call or returned an invalid result.
- `Exception` - If a connection error occurred.

send(*msg: dbus_next.message.Message*)
Asynchronously send a message on the message bus.

Parameters **msg** (*Message*) – The message to send.

unexport(*path: str*, *interface: Optional[Union[*dbus_next.service.ServiceInterface*, str]] = None*)
Unexport the path or service interface to make it no longer available to clients.

Parameters

- **path** (*str*) – The object path to unexport.
- **interface** (*ServiceInterface* or str or None) – The interface instance or the name of the interface to unexport. If `None`, unexport every interface on the path.

Raises

- *InvalidObjectPathError* - If the given object path is not valid.

The message bus manages a connection to the DBus daemon. It's capable of sending and receiving messages and wiring up the classes of the high level interfaces.

There are currently two implementations of the message bus depending on what main loop implementation you want to use. Use *aio.MessageBus* if you are using an asyncio main loop. Use *glib.MessageBus* if you are using a GLib main loop.

For standalone applications, the asyncio message bus is preferable because it has a nice async/await api in place of the callback/synchronous interface of the GLib message bus. If your application is using other libraries that use the GLib main loop, such as a GTK application, the GLib implementation will be needed. However neither library is a requirement.

For more information on how to use the message bus, see the documentation for the specific interfaces you plan to use.

CHAPTER

SIX

INTROSPECTION

class dbus_next.introspection.**Node**(*name: Optional[str] = None*, *interfaces: Optional[List[*dbus_next.introspection.Interface*]] = None*, *is_root: bool = True*)

A class that represents a node in an object path in introspection data.

A node contains information about interfaces exported on this path and child nodes. A node can be converted to and from introspection XML exposed through the `org.freedesktop.DBus.Introspectable` standard DBus interface.

This class is an essential building block for a high-level DBus interface. This is the underlying data structure for the *ProxyObject*. A *ServiceInterface* definition is converted to this class to expose XML on the introspectable interface.

Variables

- **interfaces** (list(*Interface*)) – A list of interfaces exposed on this node.
- **nodes** (list(*Node*)) – A list of child nodes.
- **name** (*str*) – The object path of this node.
- **is_root** (*bool*) – Whether this is the root node. False if it is a child node.

Raises

- *InvalidIntrospectionError* - If the name is not a valid node name.

static default(*name: Optional[str] = None*) → *dbus_next.introspection.Node*

Create a *Node* with the default interfaces supported by this library.

The default interfaces include:

- `org.freedesktop.DBus.Introspectable`
- `org.freedesktop.DBus.Peer`
- `org.freedesktop.DBus.Properties`
- `org.freedesktop.DBus.ObjectManager`

static from_xml(*element: xml.etree.ElementTree.Element*, *is_root: bool = False*)

Convert an `xml.etree.ElementTree.Element` to a *Node*.

The element must be valid DBus introspection XML for a `node`.

Parameters

- **element** (`xml.etree.ElementTree.Element`) – The parsed XML element.
- **is_root** (*bool*) – Whether this is the root node

Raises

- *InvalidIntrospectionError* - If the XML tree is not valid introspection data.

static parse(*data: str*) → *dbus_next.introspection.Node*
Parse XML data as a string into a *Node*.

The string must be valid DBus introspection XML.

Parameters **data** (*str*) – The XMl string.

Raises

- *InvalidIntrospectionError* - If the string is not valid introspection data.

to_xml() → xml.etree.ElementTree.Element
Convert this *Node* into an `xml.etree.ElementTree.Element`.

tostring() → str
Convert this *Node* into a DBus introspection XML string.

class dbus_next.introspection.**Interface**(*name: str*, *methods: Optional[List[dbus_next.introspection.Method]] = None*, *signals: Optional[List[dbus_next.introspection.Signal]] = None*, *properties: Optional[List[dbus_next.introspection.Property]] = None*)
A class that represents a DBus interface exported on on object path.

Contains information about the methods, signals, and properties exposed on this interface.

Variables

- **name** (*str*) – The name of this interface.
- **methods** (list(*Method*)) – A list of methods exposed on this interface.
- **signals** (list(*Signal*)) – A list of signals exposed on this interface.
- **properties** (list(*Property*)) – A list of properties exposed on this interface.

Raises

- *InvalidInterfaceNameError* - If the name is not a valid interface name.

static from_xml(*element: xml.etree.ElementTree.Element*) → *dbus_next.introspection.Interface*
Convert a `xml.etree.ElementTree.Element` into a *Interface*.

The element must be valid DBus introspection XML for an `interface`.

Parameters **element** (`xml.etree.ElementTree.Element`) – The parsed XML element.

Raises

- *InvalidIntrospectionError* - If the XML tree is not valid introspection data.

to_xml() → xml.etree.ElementTree.Element
Convert this *Interface* into an `xml.etree.ElementTree.Element`.

class dbus_next.introspection.**Property**(*name: str*, *signature: str*, *access: dbus_next.constants.PropertyAccess = <PropertyAccess.READWRITE: 'readwrite'>*)
A class that represents a DBus property exposed on an *Interface*.

Variables

- **name** (*str*) – The name of this property.

- **signature** (*str*) – The signature string for this property. Must be a single complete type.
- **access** (*PropertyAccess*) – Whether this property is readable and writable.
- **type** (*SignatureType*) – The parsed type of this property.

Raises

- *InvalidIntrospectionError* - If the property is not a single complete type.
- :class *InvalidSignatureError <dbus_next.InvalidSignatureError>* - If the given signature is not valid.
- **class** *InvalidMemberNameError <dbus_next.InvalidMemberNameError>* - If the member name is not valid.

from_xml()

Convert an `xml.etree.ElementTree.Element` to a *Property*.

The element must be valid DBus introspection XML for a `property`.

Parameters **element** (`xml.etree.ElementTree.Element`) – The parsed XML element.

Raises

- *InvalidIntrospectionError* - If the XML tree is not valid introspection data.

to_xml() → xml.etree.ElementTree.Element

Convert this *Property* into an `xml.etree.ElementTree.Element`.

class dbus_next.introspection.**Method**(*name: str*, *in_args: List[*dbus_next.introspection.Arg*] = []*, *out_args: List[*dbus_next.introspection.Arg*] = []*)

A class that represents a method exposed on an *Interface*.

Variables

- **name** (*str*) – The name of this method.
- **in_args** (*list (Arg)*) – A list of input arguments to this method.
- **out_args** (*list (Arg)*) – A list of output arguments to this method.
- **in_signature** (*str*) – The collected signature string of the input arguments.
- **out_signature** (*str*) – The collected signature string of the output arguments.

Raises

- *InvalidMemberNameError* - If the name of this method is not valid.

from_xml() → *dbus_next.introspection.Method*

Convert an `xml.etree.ElementTree.Element` to a *Method*.

The element must be valid DBus introspection XML for a `method`.

Parameters

- **element** (`xml.etree.ElementTree.Element`) – The parsed XML element.
- **is_root** (*bool*) – Whether this is the root node

Raises

- *InvalidIntrospectionError* - If the XML tree is not valid introspection data.

to_xml() → xml.etree.ElementTree.Element

Convert this *Method* into an `xml.etree.ElementTree.Element`.

class dbus_next.introspection.**Signal**(*name: str*, *args: Optional[List[dbus_next.introspection.Arg]] = None*)

A class that represents a signal exposed on an interface.

Variables

- **name** (*str*) – The name of this signal
- **args** (*list(Arg)*) – A list of output arguments for this signal.
- **signature** (*str*) – The collected signature of the output arguments.

Raises

- *InvalidMemberNameError* - If the name of the signal is not a valid member name.

from_xml()

Convert an `xml.etree.ElementTree.Element` to a *Signal*.

The element must be valid DBus introspection XML for a `signal`.

Parameters

- **element** (`xml.etree.ElementTree.Element`) – The parsed XML element.
- **is_root** (*bool*) – Whether this is the root node

Raises

- *InvalidIntrospectionError* - If the XML tree is not valid introspection data.

to_xml() → xml.etree.ElementTree.Element

Convert this *Signal* into an `xml.etree.ElementTree.Element`.

class dbus_next.introspection.**Arg**(*signature: Union[dbus_next.signature.SignatureType, str]*, *direction: Optional[List[dbus_next.constants.ArgDirection]] = None*, *name: Optional[str] = None*)

A class that represents an input or output argument to a signal or a method.

Variables

- **name** (*str*) – The name of this arg.
- **direction** (*ArgDirection*) – Whether this is an input or an output argument.
- **type** (*SignatureType*) – The parsed signature type of this argument.
- **signature** (*str*) – The signature string of this argument.

Raises

- *InvalidMemberNameError* - If the name of the arg is not valid.
- *InvalidSignatureError* - If the signature is not valid.
- *InvalidIntrospectionError* - If the signature is not a single complete type.

from_xml(*direction: dbus_next.constants.ArgDirection*) → *dbus_next.introspection.Arg*

Convert a `xml.etree.ElementTree.Element` into a *Arg*.

The element must be valid DBus introspection XML for an `arg`.

Parameters

- **element** (`xml.etree.ElementTree.Element`) – The parsed XML element.
- **direction** (*ArgDirection*) – The direction of this arg. Must be specified because it can default to different values depending on if it's in a method or signal.

Raises

- *InvalidIntrospectionError* - If the XML tree is not valid introspection data.

to_xml() → xml.etree.ElementTree.Element
Convert this *Arg* into an `xml.etree.ElementTree.Element`.

CHAPTER

SEVEN

VALIDATORS

dbus_next.**is_bus_name_valid**(*name: str*) → bool
Whether this is a valid bus name.

See also:

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-bus

Parameters **name** (*str*) – The bus name to validate.

Returns Whether the name is a valid bus name.

Return type bool

dbus_next.**is_member_name_valid**(*member: str*) → bool
Whether this is a valid member name.

See also:

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-member

Parameters **member** (*str*) – The member name to validate.

Returns Whether the name is a valid member name.

Return type bool

dbus_next.**is_object_path_valid**(*path: str*) → bool
Whether this is a valid object path.

See also:

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-marshaling-object-path

Parameters **path** (*str*) – The object path to validate.

Returns Whether the object path is valid.

Return type bool

dbus_next.**is_interface_name_valid**(*name: str*) → bool
Whether this is a valid interface name.

See also:

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-interface

Parameters **name** (*str*) – The interface name to validate.

Returns Whether the name is a valid interface name.

Return type bool

dbus_next.**assert_bus_name_valid**(*name: str*)
Raise an error if this is not a valid bus name.

See also:

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-bus

Parameters **name** (*str*) – The bus name to validate.

Raises

- *InvalidBusNameError* - If this is not a valid bus name.

dbus_next.**assert_member_name_valid**(*member*)
Raise an error if this is not a valid member name.

See also:

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-member

Parameters **member** (*str*) – The member name to validate.

Raises

- *InvalidMemberNameError* - If this is not a valid object path.

dbus_next.**assert_object_path_valid**(*path: str*)
Raise an error if this is not a valid object path.

See also:

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-marshaling-object-path

Parameters **path** (*str*) – The object path to validate.

Raises

- *InvalidObjectPathError* - If this is not a valid object path.

dbus_next.**assert_interface_name_valid**(*name: str*)
Raise an error if this is not a valid interface name.

See also:

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-interface

Parameters **name** (*str*) – The interface name to validate.

Raises

- *InvalidInterfaceNameError* - If this is not a valid object path.

CHAPTER

EIGHT

CONSTANTS

class dbus_next.**BusType**(*value*)
An enum that indicates a type of bus. On most systems, there are normally two different kinds of buses running.

SESSION = 1
A bus for the current graphical user session.

SYSTEM = 2
A persistent bus for the whole machine.

class dbus_next.**MessageType**(*value*)
An enum that indicates a type of message.

ERROR = 3
A return to a method call that has failed

METHOD_CALL = 1
An outgoing method call.

METHOD_RETURN = 2
A return to a previously sent method call

SIGNAL = 4
A broadcast signal to subscribed connections

class dbus_next.**MessageFlag**(*value*)
Flags that affect the behavior of sent and received messages

ALLOW_INTERACTIVE_AUTHORIZATION = 4

NONE = 0

NO_AUTOSTART = 2

NO_REPLY_EXPECTED = 1
The method call does not expect a method return.

class dbus_next.**NameFlag**(*value*)
A flag that affects the behavior of a name request.

ALLOW_REPLACEMENT = 1
If another client requests this name, let them have it.

DO_NOT_QUEUE = 4
Name requests normally queue and wait for the owner to release the name. Do not enter this queue.

NONE = 0

REPLACE_EXISTING = 2
If another client owns this name, try to take it.

class dbus_next.**RequestNameReply**(*value*)
An enum that describes the result of a name request.

ALREADY_OWNER = 4
The bus already owns the name.

EXISTS = 3
The name has an owner and NameFlag.DO_NOT_QUEUE was given.

IN_QUEUE = 2
The bus is in a queue and may receive the name after it is relased by the primary owner.

PRIMARY_OWNER = 1
The bus owns the name.

class dbus_next.**ReleaseNameReply**(*value*)
An enum that describes the result of a name release request

NON_EXISTENT = 2

NOT_OWNER = 3

RELEASED = 1

class dbus_next.**PropertyAccess**(*value*)
An enum that describes whether a DBus property can be gotten or set with the `org.freedesktop.DBus.Properties` interface.

READ = 'read'
The property is readonly.

READWRITE = 'readwrite'
The property can be read or written to.

WRITE = 'write'
The property is writeonly.

readable()
Get whether the property can be read.

writable()
Get whether the property can be written to.

class dbus_next.**ArgDirection**(*value*)
For an introspected argument, indicates whether it is an input parameter or a return value.

IN = 'in'

OUT = 'out'

class dbus_next.**ErrorType**(*value*)
An enum for the type of an error for a message reply.

Seealso http://man7.org/linux/man-pages/man3/sd-bus-errors.3.html

ACCESS_DENIED = 'org.freedesktop.DBus.Error.AccessDenied'

ADDRESS_IN_USE = 'org.freedesktop.DBus.Error.AddressInUse'

AUTH_FAILED = 'org.freedesktop.DBus.Error.AuthFailed'

BAD_ADDRESS = 'org.freedesktop.DBus.Error.BadAddress'

CLIENT_ERROR = 'com.dubstepdish.dbus.next.ClientError'
A custom error to indicate something went wrong with the client.

DISCONNECTED = 'org.freedesktop.DBus.Error.Disconnected'

FAILED = 'org.freedesktop.DBus.Error.Failed'

FILE_EXISTS = 'org.freedesktop.DBus.Error.FileExists'

FILE_NOT_FOUND = 'org.freedesktop.DBus.Error.FileNotFound'

INCONSISTENT_MESSAGE = 'org.freedesktop.DBus.Error.InconsistentMessage'

INTERACTIVE_AUTHORIZATION_REQUIRED = 'org.freedesktop.DBus.Error.InteractiveAuthorizat

INTERNAL_ERROR = 'com.dubstepdish.dbus.next.InternalError'
A custom error to indicate something went wrong with the library.

INVALID_ARGS = 'org.freedesktop.DBus.Error.InvalidArgs'

INVALID_SIGNATURE = 'org.freedesktop.DBus.Error.InvalidSignature'

IO_ERROR = 'org.freedesktop.DBus.Error.IOError'

LIMITS_EXCEEDED = 'org.freedesktop.DBus.Error.LimitsExceeded'

MATCH_RULE_INVALID = 'org.freedesktop.DBus.Error.MatchRuleInvalid'

MATCH_RULE_NOT_FOUND = 'org.freedesktop.DBus.Error.MatchRuleNotFound'

NAME_HAS_NO_OWNER = 'org.freedesktop.DBus.Error.NameHasNoOwner'

NOT_SUPPORTED = 'org.freedesktop.DBus.Error.NotSupported'

NO_MEMORY = 'org.freedesktop.DBus.Error.NoMemory'

NO_NETWORK = 'org.freedesktop.DBus.Error.NoNetwork'

NO_REPLY = 'org.freedesktop.DBus.Error.NoReply'

NO_SERVER = 'org.freedesktop.DBus.Error.NoServer'

PROPERTY_READ_ONLY = 'org.freedesktop.DBus.Error.PropertyReadOnly'

SERVICE_ERROR = 'com.dubstepdish.dbus.next.ServiceError'
A custom error to indicate an exported service threw an exception.

SERVICE_UNKNOWN = 'org.freedesktop.DBus.Error.ServiceUnknown'

TIMEOUT = 'org.freedesktop.DBus.Error.Timeout'

UNIX_PROCESS_ID_UNKNOWN = 'org.freedesktop.DBus.Error.UnixProcessIdUnknown'

UNKNOWN_INTERFACE = 'org.freedesktop.DBus.Error.UnknownInterface'

UNKNOWN_METHOD = 'org.freedesktop.DBus.Error.UnknownMethod'

UNKNOWN_OBJECT = 'org.freedesktop.DBus.Error.UnknownObject'

UNKNOWN_PROPERTY = 'org.freedesktop.DBus.Error.UnknownProperty'

CHAPTER

NINE

ERRORS

class dbus_next.**DBusError**(*type_*, *text*, *reply=None*)

class dbus_next.**SignatureBodyMismatchError**

class dbus_next.**InvalidSignatureError**

class dbus_next.**InvalidAddressError**

class dbus_next.**AuthError**

class dbus_next.**InvalidMessageError**

class dbus_next.**InvalidIntrospectionError**

class dbus_next.**InterfaceNotFoundError**

class dbus_next.**SignalDisabledError**

class dbus_next.**InvalidBusNameError**(*name*)

class dbus_next.**InvalidObjectPathError**(*path*)

class dbus_next.**InvalidInterfaceNameError**(*name*)

class dbus_next.**InvalidMemberNameError**(*member*)

CHAPTER

TEN

AUTHENTICATION

Classes for the DBus authentication protocol for us with *MessageBus* implementations.

class dbus_next.auth.**Authenticator**
The base class for authenticators for *MessageBus* authentication.

In the future, the library may allow extending this class for custom authentication protocols.

Seealso https://dbus.freedesktop.org/doc/dbus-specification.html#auth-protocol

class dbus_next.auth.**AuthExternal**
An authenticator class for the external auth protocol for use with the *MessageBus*.

Sealso https://dbus.freedesktop.org/doc/dbus-specification.html#auth-protocol

class dbus_next.auth.**AuthAnnonymous**
An authenticator class for the annonymous auth protocol for use with the *MessageBus*.

Sealso https://dbus.freedesktop.org/doc/dbus-specification.html#auth-protocol

CHAPTER

ELEVEN

OVERVIEW

Python DBus-Next is a library for the DBus message bus system for interprocess communcation in a Linux desktop or mobile environment.

Desktop application developers can use this library for integrating their applications into desktop environments by implementing common DBus standard interfaces or creating custom plugin interfaces.

Desktop users can use this library to create their own scripts and utilities to interact with those interfaces for customization of their desktop environment.

While other libraries for DBus exist for Python, this library offers the following improvements:

- Zero dependencies and pure Python 3.
- Support for multiple main loop backends including asyncio and the GLib main loop.
- Nonblocking IO suitable for GUI development.
- Target the latest language features of Python for beautiful services and clients.
- Complete implementation of the DBus type system without ever guessing types.
- Integration tests for all features of the library.
- Completely documented public API.

The library offers three core interfaces:

- The High Level Client - Communicate with an existing interface exported on the bus by another client through a proxy object.
- The High Level Service - Export a service interface for your application other clients can connect to for interaction with your application at runtime.
- The Low Level Interface - Work with DBus messages directly for applications that work with the DBus daemon directly or to build your own high level abstractions.

CHAPTER

TWELVE

INSTALLATION

This library is available on PyPi as dbus-next.

```
pip3 install dbus-next
```

CHAPTER

THIRTEEN

CONTRIBUTING

Development for this library happens on Github. Report bugs or request features there. Contributions are welcome.

CHAPTER

FOURTEEN

LICENSE

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

G

I

L

M

N

O

P

R

S

T

U

V

W