Pydbus3+ Documentation

Release 1

Harry G. Coin

Jul 31, 2017

Contents:

CHAPTER 1

Getting Started

Installation

1: Download PyDBus along with support files

PyDBus comes with source, testing and support files. The easiest way to load them is to install the 'git' package, then run:

```
cd ~
git clone https://github.com/hcoin/pydbus
```

To browse the code, perhaps choose an earlier release or learn other ways to get a copy, visit this link to the github website

> Note the python pip package manager may have a version of pydbus3. While the pip system does load necessary python support packages, it doesn't check necessary Linux libraries. Follow the instructions below to load the support packages your system may require before you attempt a pip based install.

2: Install required Linux support packages

All Linux distributions share some setup details, but each has its own special cases and available Python versions. While PyDbus is 'fully Pythonic', many popular Linux releases do not provide or by default install system libraries PyDBus needs. Some releases provide faulty versions of necessary libraries, PyDbus supports them using extra code. PyDbus does not change any standard system library.

So, installation details vary among Linux distributions. Names of the necessary packages often differ, and other nuances. The biggest difference is seen between Debian/Ubuntu based and Fedora/RedHat/Centos.

Some distributions are of such an age their dbus libraries do not natively support things like DBus publishing, or are simply missing altogether, or aren't compatible with the python version you may desire.

However, these distributions are very stable and widely used nevertheless. As such, Pydbus re-distributes many packages it requires to support your distribution needed for proper operation. The very latest versions of some distributions often have everything needed by default.

The .dockerfile for your setup will have details.

To find PyDbus installation instructions specific to your particular Linux distribution, distro version, and also python version:

- Read the file **'in this directory /tree/master/tests>'_** whose name most closely matches your installation and that ends with .dockerfile.

The .dockerfile contains all the instructions necessary to get your system ready for PyDBus.

- The .dockerfile instructions presume you have copied all the files into the /root directory on your system. Change those references to point to the PyDbus library on your system. If you followed the instructions above, that will be ~/pydbus.

Notice the source home for PyDbus on github includes a 'badge' linking to 'TravisCI', following that link shows the result of our testing the installation instructions against each supported Linux system type. If that does not show 'green' for your distribution (which it should always be), something has gone far wrong and should be investigated before putting more time into installing PyDbus.

The instructions for a few distributions use links to distribution specific files to find necessary packages. While the distribution support people rarely change or 'break' these, it has happened. If when installing a package the system reports it can't be found on your distribution, it may be because the directory the package was in when PyDbus was tested has since moved. We try to catch these things, but you may need to find the needed package elsewhere.

Note: If you want to use the version of pydbus loaded on pip, be sure to use the pip version that matches the python version you intend to use, (i.e. pip3 or pip3.4 etc). The pip system also requires all the packages named in the .dockerfile for your system to have been installed before it will succeed.

3: Setup and test PyDbus

The last step in every set of .dockerfile instructions uses setup.py to install pydbus. That step should complete without error.

As a check, all instruction files show how to run a 'unit testing' suite to confirm proper PyDbus operations on your system. If that unit test suite completes without errors, PyDbus is properly installed on your system and it is ready to go. If that test reports errors: verify the setup instructions used match your setup very closely, that the version of python you intend to use works, and that the pydbus release downloaded did not come from an experimental branch.

Credits and Bug Reporting

Author, Author!

PyDBus3 is based on the PyDBus package mostly written during 2016 and before by Linus Lewandowski, with some documentation by Collabra Ltd. The home page for the early version of PyDbus (occasionally updated) can be found here. At the time of this writing, that version supports Python version 2, but does not support DBus publishing on many popular Linux distributions, named argument positions, argument access via attribute or dictionary, the _setup capability and more.

Nearly all the code written 2017 and later (all the argument translation abilities, the ability reference arguments by name, providing testing and support code for various popular distributions, nearly all the documentation) was written by Harry G. Coin, of Quiet Fountain LLC and maintained by all those listed as providing patches on github and/or who commit changes to PyDbus.

The PyDBus3 team offers a 'tip of the hat' to Linus for a very creative Python approach to the DBus.

PyDBus3 is backward compatible with PyDBus.

Blame, Blame!

The maintainers follow the bug and comment reporting system on github's pydbus page here.

A 'Pythonic' way to understand Dbus object paths, interface names, etc.

"I found the DBus method or property or signal I want. What's with the dots, slashes, connections, object paths, interfaces, and other overlapping addressing machinery? Internet connections are easier than these local ones."

By Harry Coin

Documents explaining DBus programming take a 'top down' approach most of the time. That works when the reader already has an understanding of the issues involved when creating a dbus service. They proceed from a general to specific perspective. For that type of DBus specification read here,

As software of any complexity is more 'grown first' then 'specified later', I found it's much easier to understand both the how and the why of the DBus's seeming mishmash of addressing elements from the point of view of a team of people setting about creating a DBus service from the beginning. It's much more 'obvious' why the DBus addressing seems 'natural' when presented from this, more or less, 'bottom up' view.

This document explains terms like 'bus_name', 'object_path', 'interface member' and 'proxy'.

Waycool.org's Airtaxi Service

Suppose you and three other Python engineers at waycool.org decide to provide a DBus service so your product can be used by pretty much any program written in any language.

You decide to call the service 'airtaxi'.

It is a big project, so you divide it into four different conceptual areas, natural to the purpose of the project. You decide to call them, 'setup', 'status', 'operations' and 'reports'.

Method, Property and Signal Names

Each engineer is assigned one area, when done they each return with a directory full of many methods, properties and signals.

> A method is a routine that is capable of both accepting and returning information. Pretty much the same as a function combined with the process running it. Lots of processes may be using the same function with different 'global' variables. The DBus 'object path' is, more or less, like the process id. DBus here borrows from the language of 'Object oriented programming'.
>
> It's better if the writers are careful to use the same names for the named arguments methods if they convey the same meaning.
>
> A read-only property is a routine that takes no parameters/arguments that can only appear on the right side of =. A write-only property can appear only on the left side of an =. Readwrite properties (most of them) can appear on either side of the =, though they can take no parameters.
>
> A signal calls a user routine with details about an event of interest.

So, back to the example, very creatively, the first engineer calls her routines 'a1,b1,c1,version'. They add 'version' because of their long range thinking. The second engineer writes code, calling theirs 'a2,b2,c2, and version'. Similarly for engineer 3 and 4.

They notice some of their routines have the same name, but do different things.

What to do?

Interface Members

Engineer 1 preprends 'setup.' to their routines, 2 prepends 'status.' to theirs, 3 prepends 'operations.' and 4 prepends 'reports.'. Why? Because those names best describe what each person's work is for, what that basket of routines does, what they have in common.

If it happened any two of them used the same name for some routine, because they stuck the purpose name in front of it, and no two purpose names are the same, there would be no mistake about which was desired.

So, now we have 'setup.a1' and 'setup.version' and 'reports.a4' and 'reports.version' and so on. So it can be clear which area of the project the version function describes.

The name DBus gives such as 'setup' and 'reports' and 'status' and 'operations' as used above is 'interface members'. More or less, the names for baskets of like-purpose routines.

> Because most of the time the routines have names that differ across the whole project, and the name of the routine makes it pretty clear what role it has in the overall project, it seems an avoidable bother to have to always use the interface member name when there is no conflict.
>
> So, if there is no conflict, it is optional whether to use the interface member name. The system will puzzle out which to use. If there is a conflict, and there is no interface member name specified, the routine specified by the service publisher is the 'default' and as such the interface member name is used. So in this case, a call for 'version' gets the same as if, say, operations.version was used.

Project names

Our waycool.org company has got quite a few projects going, airtaxi is only one, and nobody wants to keep track whether any two of them will pick the same names for interface members.

So, for pretty much the same reason 'setup.' and 'operations.' was stuck on before the routine name, they stick 'airtaxi.' in front of all of them.

So, 'airtaxi.reports.a4' 'airtaxi.operations.b3' and so on.

Organization names

Our waycool.org folks know their airtaxi service could be one of a great many services written by who knows who running on a machine. And, there could be who knows how many different divisions of waycool might be cooking up their own version of airtaxi. So, for pretty much the same reason the interface member name got stuck on before the routine name, and the project name got stuck on before the interface name, DBus sticks on whatever else is needed to make all that clear. It is prepended in the order of most general to most specific. In our case 'org.waycool.' Could be as.long.as.it.needs.to.be.

If the company is a one trick pony, having but the one project, then it may leave the 'project name' noted above off. It's not a good idea to do that.

So, now, all the routines together have no possibility of name conflict:

```
<organization.thru.project> .<interface member like 'operations'> .<a routine of that␣
↪member>
```

DBus terms <organization.thru.project> above the 'interface', So Dbus naming is:

```
<interface.usually.with.dots>.<interface member no dots>.<method, signal or property␣
↪of that member no dots>
```

Usage of the above in other online documentation tends to use just the word 'interface' or 'interface spec' rather loosely, leaving to the reader to puzzle through from the context whether it's just the <interface> above they mean when using the word 'interface'. Or, the interface with the member name. Or, the whole thing. But Pydbus makes a clear distinction.

> Jumping ahead, pydbus infers the <interface> part from elements discussed later below. All of the interface members of an <interface> are loaded as keys to a dictionary. The method, signal and so forth above are either attributes or method names of the object returned as the value for that key. If the method, signal or property name is unique, then it can be accessed without the [interface mamber] bit.

From here, the work of the service designers is done, all their routines have names that are different enough to let out into the world without fear of conflict.

Keeping with the implementation story, now comes the time for the DBus engineering team to do all the connection plumbing needed to make it available.

First up, is it to be one service per system, or one service for each user's session?

System or Session Bus?

The engineers notice the airtaxi service is to be provided to any process launched by any user. Linux calls that the namespce the SystemBus. Had the service been to provide something to each user's GUI session, pretty much independently of other users that may or may not do the same thing, the SessionBus would be the namespace to use. DBus provides a rich set of other options, which 99% of Linux services will never use. i.e. dbus connections over TCP.

The PyDBus code returning the root object connecting a client program to either the systemwide or user-session-wide overall communications environment is:

```
import pydbus
mysysbus = SystemBus()
#mysessionbus = SessionBus()
```

Next up: The client knows where their end of the communications pipe is. But how and where to connect the other end to the service?

Bus Name: Connection Destination

Much as a website address connects a browser client to whatever it may be the server offers: A PyDbus destination address is the name on the DBus for connecting the client to whatever number of interfaces a dbus service publisher offers.

It may be the dbus service publisher has combined a several entirely different whole services, including airtaxi named above, from any number of organizations, into one service. Just as one website may offer connections to many things that perhaps have little to do with one another.

PyDbus calls this more-or-less dns style address the 'bus name'. The parameter name used for this later on is bus_name.

Nearly all the time, it is exactly the same as the <interface> element described above. In our example 'org.waycool.airtaxi' . As it is almost always the same as that, it uses the this.that.other string format.

Confusingly, the bus_name connecting the client to the published service software need have no terms in common whatsoever with the interface names offered there. com.quietfountain.permanet could offer the waycool.org.airtaxi

interface. Most of the time, to use some elements of an interface, the bus name is the same as the <interface> term above.

Next up: Finally, how to choose from among the interfaces one to access?

Object Path: Name the routine needed.

The designers have created a suite of interfaces described above, and connected the bunch of those to the appropriate session or system bus and given it a bus name (usually the <interface> above).

Here we make the big perspective shift from the people designing and publishing a service, to the programmer making use of it. The means whereby the client programmer tells the dbus system which one from among all the routines it wants to use is called the 'object path'. Not the same as the interface name above. The interface system is as the contents of filesystem or directory, the object path is about picking one of them.

Each routine the client wants to use is termed a dbus object and its name is the 'object path'. It nearly always leads with the name of the related <interface> and, technically (the pydbus syntax is different), ends with the specific routine name.

An example of a whole object path could be /org/waycool/airtaxi/reports/version though it won't look like that in the client code.

The object path format has three differences from interface names (which use a .) and bus names (which use a .):

1: It leads with a /

2: It replaces all the . in the (possibly) related interface name or bus name with a /

3: When used, it is broken into three parts.

The first part is nearly always the / version of the bus name (which is usually but not always the <interface> above, such as org.waycool.airtaxi becomes object path /org/waycool/taxi).

So often is the object path the / version of the bus_name, that the routines that set up the connection to the bus use the / version of the bus name and the object path argument can be omitted.

The second part is one from among the entire collection of <interface members> that lie 'under' the first part. In practice, The interface desired is given as the key in a dictionary having as values objects that stand for each of the interfaces the organization publishes.

The third part is the name of the specific routine the client wants the publisher to run. These are attributes of the interface value named above.

So, the object /org/waycool/airtaxi/setup/initialize

in python / pydbus it looks like:

```
result =  pydbus.SystemBus.get('org.waycool.airtaxi')['.setup'].initialize(parameter,␣
↪parameter)
          |               |                           |          \__________v
          comm structure  .get('the bus_name')        ['interface member'] . method␣
↪name
```

The first term above chooses the bus that has one instance of the service for the whole system.

Then it chooses the bus_name org.waycool.airtaxi as the general dns-like name for the particular server on the bus. It sets the start of the default <interface> above to be also org.waycool.airtaxi, As the object path is omitted, it automatically sets the object path to start with /org/waycool/airtaxi

Then it chooses from among all the interfaces 'setup'. As it starts with a '.', it implies to not overwrite the initial object path and interface name, but to append .setup as the name of the <interface member> desired, and append /setup to the object path desired.

Then it chooses from among all the methods, properties and objects in org.waycool.airtaxi.setup the method initialize. It appends .initialize to the interface specification, finalizing that, and it appends /interface to the object path, finalizing that.

Which explains a lot, but is a bit clunky to read and use. Read on.

Proxies: One Name to Rule them All

Great, we have bus types and bus connection names and object paths and interface names leading to the routines wanted, how to actually call them without repeating all the detail above each time? DBus / Pydbus calls the objects that retain all that detail in themselves 'proxies' or 'proxy objects'.

So, the code to access the airtaxi routines written by waycool's engineers, selected from whatever else may be in the entire collection marketing added to the object path /org/waycool/airtaxi provides that would be:

```
waycoolAirtaxi = mysysbus.get(bus_name='org.waycool.airtaxi',object_path='/org/
↪waycool/airtaxi', .. other parameters)
or
waycoolAirtaxi = mysysbus.get(bus_name='org.waycool.airtaxi', .. other parameters)
or
waycoolAirtaxi = mysysbus.get('org.waycool.airtaxi', .. other parameters)
```

Because so very many interface names begin with org.freedesktop, any bus name that begins with a . is presumed to include org.freedesktop before the particular name. So '.bus_name' is understood to mean 'org.freedesktop.bus_name'.

Finally: Using the service's capabilities

so, above, we have the proxy named waycoolAirtaxi all set, ready to go. We call call engineer 1's status routine a1 like this:

```
mystatus_a1 = waycoolsAirTaxi['.status'].a1(<arguments ...>)
```

and, engineer 2's reporting property b2 like this:

```
myreport_b2 = waycoolsAirTaxi['.reporting'].b2
```

We could have used the whole interface spec:

```
myreport_b2 = waycoolsAirTaxi['org.waycool.airtaxi.reporting'].b2
```

Being able to name the specific <interface member> is great because both the .status and .reporting collection (a.k.a interfaces) have routines called 'Version' and they are not the same.

To improve speed and readability, to get a particular interface basket of routines and guarantee no name collisions, there is a shorthand convenience for readability. Append the [.<inteface name>] above to the get call and omit it from the uses. Be careful to name the object instance appropriately as only the routines published under that particular interface will be available if the [.<interface name>] is appended in the .get call. So:

```
waycoolAirtaxi = mysysbus.get(bus_name='org.waycool.airtaxi', .. other parameters)
mystatus_a1 = waycoolsAirTaxi['.status'].a1(<arguments ...>)

#is delivers the same value as

waycoolAirtaxi_status = mysysbus.get(object_path='/org/waycool/airtaxi', .. other␣
↪parameters)['.status']
mystatus_a1 = waycoolsAirTaxi_status.a1(<arguments ...>)
```

As a further convenience, noticing that nearly all the routines in all the interfaces written by the waycool.org engineers have names that differ, if there is no method, signal or property name conflict it is allowed to omit the <interface member> part above:

```
waycoolAirtaxi = mysysbus.get(bus_name='org.waycool.airtaxi', .. other parameters)
mystatus_a1 = waycoolsAirTaxi['.status'].a1(<arguments ...>)

#is the same as

waycoolAirtaxi = mysysbus.get('org.waycool.airtaxi', .. other parameters)
mystatus_a1 = waycoolsAirTaxi.a1(<arguments ...>)
```

Wait? Where did the part of the interface name that comes before the specific routine but after 'airtaxi' go?

Nowhere, it can still be accessed explicitly as shown above. However this shorthand takes advantage of the fact routine names rarely conflict and so specifying the last bit of the interface name (.status,.reports,.setup,.operations) above doesn't really add anything other than length to the code.

Also, it's a bit of a convenience since the service publisher can specify which <interface member> has 'the latest' version of routine with a given name, and so provide that without forcing the client to specify one if it wants the latest, while still being able to specify a particular one if desired.

Gotcha – Routines with the same name, different interfaces

Notice if the 'Version' name is used, like waycoolsAirTaxi.Version, there are four Version routines, each provided by a different interface. The one that gets called is controlled by the service publisher. This scheme can help replace old versions of routines with new ones without changing any names for the client. Clients that want a specific routine provided by a particular interface even if there are others should specify in the interface name in the .get call or as a ['.<interface name>'] later on.

Moving deeper:

Often, a method in an interface will return a string or more of other interfaces of interest, all of which are related to the one called but not specified in advance except as a general template. For example, a service doc may specifically identify how to get a list of, say, usb devices. But the whole interface spec for each device may not begin with anything like the spec used to collect the names. So, the object path may not be the same as the interface spec.

In those cases, while the bus_name will still be 'org.waycool', the object name (which picks from among the specifications), could be /com/warranty/usb and the spec also com.warranty.usb... If the call to get the proxy fails, check whether the object_name (='/com/warranty') parameter must be stated explicitly in the .get call above.

Argument Names as object-wide variables and defaults

So many DBus services use the same argument position name in more than one method. What's more, usually with the same meaning across functions. So in some ways, each such name is an attribute that has a meaning within the bus.get(...) object.

In the example above, waycoolAirTaxi._state.<argument position name> is always updated whenever data is supplied for that parameter, whether by default or passed in to a method or named as a member of the result returned by the method.

PyDbus has an option *override_defaults_with_state*, which when true causes the latest value for an argument name in ._state.argname (also ._state._arg_argname) to be supplied to the method as a default value for an omitted parameter. With that, code like:

```
status = waycoolAirTaxi.GetStatus(condition='warm')
waycoolAirTaxi.report(status,condition)
```

becomes:

```
waycoolAirTaxi.GetStatus(condition='warm')
waycoolAirTaxi.report()
```

So, what's 'Introspection' about?

Routine Names

DBus provides a way to fetch the names of the routines in the basket of functions each interface provides.

Much of the time this is of no use as that detail is in the documentation for the service. But, this can be a useful when some of the routines list the names of further interfaces. The number of those may vary from time to time or machine to machine. For example the number of network adapters a machine has may control the number of sub interfaces each providing the same basket of routines but with the assumption all related activity refers to the particular interface.

This can also be a good way to check whether the particular version of a dbus service has the routines necessary by the caller's code.

Parameter Names

Also, Introspection provides, usually, the names of each of the parameters in a method call. This version of PyDbus allow you to use either positional or named parameters.

In something of a twist, introspection names the results of method calls and property calls as if parameters. Even the name for not only a property, but also for what the one argument would have been named had it been a one argument method call.

Types and order of data per parameter

Last, introspection defines almost everything to do with the order and data type of each parameter. Whether it should be a list, or an array of integers, or a tuple of some list of other data types.

The Python to Dbus Variant Conundrum

The only thing introspection does not provide is to do with a data type called 'Variant' when it is being used as a python parameter element to a dbus method. Variant data types are a shorthand for 'don't presume to know anything about what this has within it until the moment you get it'. All the details as to what's in it come along with it so it can be unpacked and used.

Which in some ways is just fine when information is coming from the dbus toward a python use. Pydbus can puzzle it all out and use the most relevant python type needed each time a Variant comes in.

But, it is often the case that a DBus service which specifies that a Variant data type should be supplied, in fact cannot cope with whatever combination of data types the python client may choose to send. Introspection is of no use, as there is no guidance from it as to what Variant content a method can cope with. The service doc just asks for 'variant'.

Pydbus Translation Specifications allow PyDbus to check for the various allowed data type patterns the service can actually manage.

Pydbus Pythonic "Full Introspection"

When integers are not amounts, but each 'stands for' some condition or state

The documentation for nearly all services provides not only all the paths and interfaces and routines and parameter names and (mostly) what data types to use. Those documents also define how to interpret what integer results 'mean' when used as flags or descriptions and not as amounts. Right in those same documents you'll read something like:

```
#define MYSERVICE_RUNNING_HOT 1
#define MYSERVICE_RUNNING_WARM 2
#define MYSERVICE_RUNNING_COLD 3
```

Then there will be property like 'running_status' that returns a number leaving the programmer to puzzle out what the publisher meant by it. That help is not provided in the introspection. So PyDbus offers it. Properties like 'running_status' mentioned earlier returns 'HOT' or 'WARM' or 'COLD'. And can be set the same way if the publisher allows it:

```
myservice.running_status='HOT'
#instead of
myservice.running_status=dict_of_meaning_to_value_for_running['HOT']
#or, worse,
myservice.running_status=1
```

Similarly for integers used as a collection of bit flags. flag&1 means on or off, flag&2 means high or low, etc. etc.

Someday:

You might suppose if:

```
waycoolsAirtaxi = mysysbus.get('org.waycool.airtaxi', .. other parameters)
```

and the routines available in the .status so called <interface member> term is

```
waycoolsAirtaxi_status_proxy = waycoolsAirtaxi['.status']
```

and then the .status interface has some further level of routines in 'sub' interface .status.substatus then you should be able to access those routines by:

```
waycoolsAirtaxi_substatus_proxy = waycoolsAirtaxi['.status']['.substatus']
```

I'd like to see something like:

```
myobject = bus.get('bus.name' , ...)
#then
myobject['some.interface.or.other'].foo(arg)
#and
mysubobject_method_yadda_result = myobject['some.interface.or.other']['.deeper.object
↪'].whatnot(yadda)
#and have interface .<whatnot> by default refer to the object with / changed from .
#so
mysubobject == myobject['.deeper.object'] == myobject[some.interface.or.other]['.
↪deeper.object']
```

The point is to avoid mostly redundant 'pydbus.SystemBus.get(...)' calls for one-of use of sub-interfaces.

Not so much yet.

Basic Operations

While there are many ways to 'fine tune' PyDBus operations (all documented elsewhere), the defaults are usually all that's needed.

These documents presume you understand the basics of what a DBus is and what it can do. The DBus system was created so that programs written in various languages all operating at any time could make use of capabilities provided by a 'service' no matter the language in which that service is written.

The means by which various Linux distributions launch services, whether the service is provided on the same host as the client/user (usually via systemd) , whether the user's authorizations include access to this or that part of this or that service (usually via polkit), and so on are distribution specific though largely similar.

As PyDBus aims to handle most of the details about DBus operations internally, it is likely all you will need to make use of a DBus service is:

1. The ability to import PyDbus services (see the installation section if this doesn't work for you). And, whether the service is provided via the system dbus (most of them), or the session dbus (services to do with the GUI usually). For example:

```
from pydbus import SystemBus # or SessionBus
```

2. The dbus name of the service. For example, to access one of the Network Manager's services:

```
sb = SystemBus()
NetworkManager = sb.get("org.freedesktop.NetworkManager",translation_spec=True)
```

   If you leave out 'translation_spec...' then there is no access of arguments by name, just position.

   If how to use 'object path' and 'interface' is unclear, visit here

3. The name(s) of the methods, properties or signals of interest. For example:

```
result = NetworkManager.ActivateConnection(....)
```

4. The name(s) and expected contents of the arguments to be set or fetched (or the position of that argument in the method call). For example, suppose ActivateConnection was implemented on the service side, maybe something like:

```
class NetworkManager(...):
         def ActivateConnection(device,connection=somedefault, specific_
↪object=anotherdefault):
             ....
```

   Then way that would be accessed via a PyDbus client could be:

```
    my_connection_info = ....
    my_device_info = ....
    my_netobject = ...

result = NetworkManager.ActivateConnection(my_device_info,my_connection_info,my_
↪netobject)
```

   or, if there are defaults for some arguments (or you don't want to have your code break when the service publisher changes the order or number of arguments):

```
result = NetworkManager.ActivateConnection(device=my_new_device_info)
```

5. Return values. Suppose the result above was a list. For discussion, suppose the names of the list entries are:

```
[condition, load, cost]
```

6. Let's say an argument, whether named in the call or the return, is a number, but, the number isn't used to represent a count of something. Instead each possible number 'stands for' some named state or condition.

   For example the dbus service documentation explains for argument 'load' above, '0 means off, 1 means low, 2 means middle and 3 means high.' Then if translation is active, the PyDbus value for load will not be an integer, but one of the strings 'off', 'low', 'middle' and 'high'. Both when reading and when setting.

   This is only the case when there is a 'translation specification' provided for that dbus service. See the folder 'translations' within the pydbus source folder to see what services have translation capabilities. Use of these is optional. So, if translation is active this works:

```
result[1] in ('off', 'low', 'middle', 'high) ==True

result[1] = 'middle'
```

   otherwise if translation is inactive, result[1] would be a number. The PyDbus service is presented only with the related number.

7. Let's say argument result.condition above is defined as so many are as flags, one per bit. So for instance if:

```
all defined bits 0 -> off.
bit 1 on -> 'wide' , off 'narrow'.
bit 2 on -> 'tall' , off 'short'.
bits 4 and 5 -> both off 'slow', ==1 'normal', ==2 'fast', ==3 'too fast'
bit 6 -> don't care, ignore it whether it's on or off.
```

   then, for example:

```
result.condition == result[0] == ('narrow','tall','slow')
```

   there are options to not include 0 values in the list, to have defaults for unset values and the like.

Notice the DBus service specifications give names to not only the arguments required by methods, but also the names of the results. Then, if translation is not active or you are using the older Pydbus, this is the way you can access that result:

```
result[1]
```

But, if translation services are active, these are the ways you can access the result variable:

```
result[1] == result.load \
          == result['load'] \
          == result._arg_load \
          == NetworkManager._state.load
```

The _arg_ above provides a means to access properties whose names conflict with all the names and properties defined by the list class.

Notice the yyy._state.xxx above. As it is the custom among DBus service writers that an argument name has the same meaning and format every time it appears in an argument list for all the functions, properties and signals that use it: PyDbus updates the ._state.argname every time it is changed whether set during a function call or set in a return.

The PDbus setting override_defaults_with_state, for example:

```
NetworkManager = sb.get("org.freedesktop.NetworkManager",translation_spec=True,
↪override_defaults_with_state=True)
```

provides the last value set for a given argument name (whether by call or named in a return value) as a default for any use of that argument within the particular bus.get / interface. Anytime an argument is omitted whether by name or position, instead of the default fixed once at setup time, the latest known value for that argument name is used.

This saves a great deal of keeping track of which argument goes in what position, and repeatedly entering arguments that are used often but not changed by client code.

It's also a great boon to code maintainability because: It's entirely up to pydbus to manage which arguments go in what position in method calls, and it gets that information directly from the service publisher using whatever version of it is provided by the version of the service that happens then to be running.

Should the service change the number or order of arguments, so long as all the argument names specified by the client still exist, the client side code need not change.

For example:

```
r = NetworkManager.somemethod('wide','middle',4)
...
r = NetworkManager.somemethod(cost=5) == NetworkManager.somemethod('wide','middle',5)
...
#suppose
NetworkManager.GetCost ==10  #reads pydbus property with argument name 'cost'
...
r = NetworkManager.somemethod() == NetworkManager.somemethod('wide','middle',10)
```

While some would call this capability 'dangerous', remember that many DBus services have just a few methods and properties, and have a namespace that is consistent within it, including argument names. While the ._state attribute is always updated, it is optional to use it in place of defaults for argument names or positions omitted in any particular method call.

Last, the scope of all settings, names, timeouts, etc is not global across all uses of pydbus, but limited to the object returned, in the case above:

```
instance1 = sb.get(<a set of items>)
instance2 = sb.get(<the same thing as instance1>)
instance3 = sb.get(<something else>)
```

Even if an argument position should have the same name, its contents are in no way connected among instances above. Changes to any one do not affect any of the others (unless the same dbus service is used by both and the service chooses to use one value for all callers).

To get more familiar with the concepts in general terms, see the examples section. To see runnable examples, look in the tests and examples folder within the pydbus source.

tests._unittest
tests.context
tests.identifer
tests.publish_properties
tests.publish_multiface
tests.publish
tests.unittest_server
examples.systemctl
examples.nminfo

tests._unittest

examples.systemctl

examples.nminfo

CHAPTER 2

Sourcecode Links

pydbus.argcarrier
pydbus.auto_names
pydbus.bus_names
pydbus.bus
pydbus.exitable
pydbus.generic
pydbus.identifier
pydbus._inspect3
pydbus.method_call_context
pydbus.proxy_method
pydbus.proxy_property
pydbus.proxy
pydbus.proxy_signal
pydbus.publication
pydbus.registration
pydbus.request_name
pydbus.subscription
pydbus.timeout
pydbus.translator
pydbus.translator_bitfield
pydbus.translator_introspection
pydbus.translator
pydbus.translator_setup
pydbus.translator_utilities

pydbus.argcarrier

pydbus.auto_names

pydbus.bus_names

pydbus.bus

pydbus.exitable

pydbus.generic

pydbus.identifier

pydbus._inspect3

pydbus.method_call_context

pydbus.proxy_method

pydbus.proxy_property

pydbus.proxy

pydbus.proxy_signal

pydbus.publication

pydbus.registration

pydbus.request_name

pydbus.subscription

pydbus.timeout

pydbus.translator

pydbus.translator_bitfield

pydbus.translator_introspection

pydbus.translator_setup

CHAPTER 3

Legacy/Deprecated Docs

Getting Started:

Those new to this package should start with [Goal, Rationale and Use Cases](https://github.com/hcoin/pydbus/wiki/Goal,-Rationale-&-Use-Cases-(First-visit%3F-Start-here.))

Experienced pydbus users: this version of pydbus is backward compatible if translation services are not enabled. [Jump to differences](#Differences)

The project of 'native python' access to a dbus service begins with reading through the pydbus translation specification in the file whose name is the D-Bus service path of interest in the PyDBus translations sub folder. How to reference that file and begin using

    that file in programs is described here.

Pydbus allows for a translation structure for each dbus service string, e.g. 'org.freedesktop.NetworkManager.Device'. Each translation structure is a dictionary describing everything necessary to express the dbus service as if it was written from the ground up in python. It may contain custom functions in complex cases or when a collection of arguments for a dbus method or signal, or even property, is best represented as an instance of a python class.

If no such file is found, then either the native pydbus untranslated system was enough to make the service appear as if it was written in python in the first place, or there is no translation spec as yet for that service. Writing a translation specification often can be a short effort, look a few of them over then read [Introduction To Translation Specifications](https://github.com/hcoin/pydbus/wiki/Introduction-to-Translation-Specifications). If you write one, and it is for a published dbus service, consider submitting it here for publication so everyone can use it.

The translation guidance specification file will either be documented and readable enough to be complete for that service. A minimally documented file will still contain everything necessary to understand the published d-bus specification from the python perspective. From there, read through this "Basic PyDBus Usage" and that should be enough to succeed.

If your need is met by the roster below, you might be able to skip everything to do with pythonic translation files. It's worth a check.

- D-Bus services that all accept/return a format fixed not at call time but when programming.
- Signals and methods would not benefit from named arguments, they use only positional arguments.

- Only make calls to D-Bus methods or publish signals that do not permit variant arguments (arguments whose format is specified at call time –whether or not it's known at programming time how it will be used.)
- Do not use integers for anything other than amounts. Not describing a state or condition forcing the user to maintain information about what each number 'means'.
- Do not use an integer to mean more than one thing. So, no 'bit position X means Y is active'. No 'these bits are a number from x to y, those bits are flags, these two are another number' etc.
- Do not use classes that represent within one instance more than one argument to or from a D-Bus server.
- Uses only arguments that can be expressed as basic python tuples, dicts, lists, string, ints, booleans and doubles.
- Accept writing 'one of' code necessary to convert D-Bus-formatted arguments to/from 'the right endian-ness', 'the right signed/unsigned', 'the right byte count', or merge them to/from python classes.

To activate translation services and begin programming, read on:

Summary: If the translation specification for the bus name is in the translation folder:

> **' from pydbus.bus import SystemBus # or SessionBus '** *bus=SystemBus() # or SessionBus() ' 'proxy_for_dbus_object=bus.get(<bus name>,<object path>,translation_spec=True) #any use of proxy_for_dbus_object will need only python standard classes and variables.*

If the translation specification will be passed at setup time:

> **' from pydbus.bus import SystemBus # or SessionBus'** *bus=SystemBus() # or SessionBus() ' 'proxy_for_dbus_object=bus.get(<bus name>,<object path>,translation_spec=<translation guidance dict>) #any use of proxy_for_dbus_object will need only python standard classes and variables.*

After that, dbus service properties, methods and signals are native python properties and methods. All the details necessary to succeed with pydbus and each service should be in the translation file spec for that service.

Generally speaking:

- If a dbus service publishes MyProperty, continuing the example above

'>>>proxy_for_dbus_object.MyProperty = 'Red' #set the property. ' '>>>print(proxy_for_dbus_object.MyProperty) #read the property. ' 'Red '

- If a dbus service publishes a method (function) called MyFunction continuing the example above and the translation spec supports named arguments, there are three options: passing one object instance and both extracting arguments and setting return values from object attributes whose names are in the translation specification, or using typical python named arguments

'>>>object_instance = ClassICareAbout() ' '#... ' '>>>proxy_for_dbus_object.MyFunction(object_instance) '

#If it does not support named arguments (boo!):

>>> returnarg1, returnarg2 = proxy_for_dbus_object.MyFunction(object_instance.var1,object_instance.var2)

- If a dbus service offers to send a signal, and the translation spec supports named arguments:

'def MySignalHandler(signal_object_instance): ' *pass*

'def ILikeToKeepTrackOfArgumentPositions(a,b,c): ' *useful_object= MyClass(a,b,c)* 'pass '

'from gi.repository import GLib ' 'proxy_for_dbus_object.onStateChanged = MySignalHandler ' '#replaces ' '#proxy_for_dbus_object.onStateChanged = ILikeToKeepTrackOfArgumentPositions ' 'loop = GLib.MainLoop() '

- Publishing A New Service:

from gi.repository import GLib ' 'proxy_for_dbus_object.onStateChanged = MySignalHandler ' 'bus.publish("path.to.my.service",

MyFunction(), ("Subfunction1", MyFunction()), ("Subfunction2", MyFunction()), ("Subfunction2/NextLevel", MyFunction())

) ‘loop = GLib.MainLoop() ‘

- Accessing Path SubFolders:

‘ from pydbus import SystemBus ‘ ‘ from tests.nmdefines import PydbusNetworkManagerSpec,NM_DBUS_INTERFACE,NM_DBUS_INTERFACE_DEVICE ‘

‘bus=SystemBus() ‘

old C way nm=bus.get("org.freedesktop.NetworkManager",'Devices/0')["org.freedesktop.NetworkManager.Device"] ‘ ‘print(str(nm.Capabilities) + ", "+str(nm.DeviceType)) ‘ ‘7, 14

*#pythonic way ‘ ‘nm_trans=bus.get(NM_DBUS_INTERFACE,'Devices/0',translation_spec=PydbusNetworkManagerSpec)[NM_DBUS_*
‘print(str(nm_trans.Capabilities) + ", "+str(nm_trans.DeviceType)) ‘ ‘#(‘NM_SUPPORTED’, ‘CARRIER_DETECT’, ‘IS_SOFTWARE’), GENERIC ‘

Other Pydbus differences when translation services are active ### The ‘timeout’ named argument in pydbus’s method and signal functions is deprecated, replaced by "_pydbus_timeout" The translation services enabled named arguments in dbus operations. The named argument ‘timeout’ is a popular name for a named argument for dbus services, and as such should not be claimed by pydbus. The facility to wait for dbus services to reply only for a fixed amount of time is now "_pydbus_timeout". The new phrase is recognized whether or not translation operations are chosen. For backward compatibility, ‘timeout’ retains its old meaning only when translation service are not engaged.

- As ever:

Help(object)

is useful.

This document assumes you are familiar with [’Basic Python D-Bus Usage’](https://github.com/hcoin/pydbus/wiki/Basic-PyDbus-Usage), and that you’ve looked over some translation specifications just in an overview way before proceeding here.

The Translation Guidance Specification

Each dbus path in need of translation guidance to appear as though the service was written in python to begin with associates the dbus path name to a translation structure. Whether that structure is passed as a parameter when access to the dbus path is first requested, or fetched from the translations pydbus subfolder, the structure specification is the same.

Advantages of file-based translation guidance specs. If the dbus service in question has components that require the use of the "_match_function" facility (usually calling user or library functions to contain several arguments in one python class instance), using a file based system encourages a good practice: including the functions the particular specification needs with the spec itself. It’s a good idea whenever possible if only for maintainability. **Beyond the python code in the spec, that file is understood by all pydbus users to be the one place everything there is to know about understanding that service in the python context may be found.**

Whether that includes exhaustive details about the ‘pythonic way’ to access every key the service supports, or just how to understand the related general-use service publication document in the python context documenting only the python enhancements: A python user of the dbus spec knows to start there. Don’t let them down.

Argument based translation specifications have value during development, and for dbus services that are entirely ‘in house’, with no intent to publish for other developers to access.

Overview Before reading further here, get a feeling for how all this goes by giving a little attention to a few existing specifications in the translations subfolder of pydbus. Not for detail, but a general idea about the structure and organization.

Note: All of the pydbus code related to these features is called internally, only by pydbus routines, not the users. All access is via the dbus proxy's published properties, methods and signals. As such, the documentation in the code itself is meant for the maintainers.

With that, what follows is the specification for the translation structure understood by this package, described 'organicly' from general to detail:

Describing whether and how to translate by levels: 1. File organization / dbus paths 1. The signal, method or property keys and dataflow direction in use per path 1. For each key, call type and dataflow direction: the number of arguments and whether or not to translate some, none or all of them. 1. For each specific argument to be translated: whether to name it, what part of the introspection string (data format definition) dbus thinks it should match, whether to translate some or only part of it, whether to call user defined translation functions or built-in translation capabilities. 1. Guidance sections describing different built-in often used specific translation choices. 1. Special Cases

Fully implemented translation specs for well known dbus interfaces renders the dbus 'fully pythonic'– meaning anything that's a class is aggregated into an instance of the class, all variables have names and are numbers only if used as an offset from 0, and it's possible to just pass an instance of a class that has the necessary variables as members to any method, property or signal and the system will figure out how to pack it up and send it off.

Overview

Pydbus with translation enabled:

>‘ dbuspath_object.method_name(myClassObject)‘ ‘ print(myClassObject.y)‘ ‘ >>> ‘Updated_y’‘

Pydbus without translation:

>‘ myclass.x, myclass.y = dbuspath_object.method_name(helper(myClassObject.x),lookup_number[myClassObject.y].byteswa
print(str(myClassObject.y))

‘ >>> 14‘

When using PyDbus, if you enable a translation dictionary for a dbus path that describes anything special about it, all the dbus-specific issues are invisibly managed so well it's very hard to tell the difference between a python method, property or signal written from the ground up in python vs. one handled by a partner over the dbus. Given a well written translation spec, hopefully distributed freely in the pydbus translations directory for widely used packages, the user need know nothing further about how the dbus delivers capability.

PyDBus without translation enabled does expose DBus properties, methods and signals as python class attributes and methods, but requires comprehension of DBus 'machinery' and cumbersome one-of glue functions to mutate arguments from native python usages to and from DBus needs, except for the most basic DBus services.

First time users, start with [Goal, Rationale and Use Cases](https://github.com/hcoin/pydbus/wiki/Goal,-Rationale-&-Use-Cases-(First-visit%3F-Start-here.))

To use a ready-to-go translation to access a pydbus partner, read [Basic PyDBus Usage](https://github.com/hcoin/pydbus/wiki/Basic-PyDbus-Usage)

To write a translation specification for a dbus path, start with [Introduction To Translation Specifications](https://github.com/hcoin/pydbus/wiki/Introduction-to-Translation-Specifications)

See [status.txt](https://github.com/hcoin/pydbus/blob/master/status.txt) for development plans and status information.

Rationale

The original PyDBus author, Linus Lewandowski, wrote this package's aim as a sentiment: _PyDBus aims to make [D-Bus](https://www.freedesktop.org/wiki/Software/D-Bus/) ['fully pythonic'](https://www.python.org/dev/peps/pep-0020)_.

Such a package for the D-Bus system, (which passes structured messages including the message's format between entirely independent processes to provide well described services), would include the ability to:

- **name arguments in methods and signals as well as 'C style' position sensitive argument lists. The first pydbus allowed:**
  a,b = dbuspath.method(b,c,False)

  to that, we add: *a,b = dbuspath.method(this=c,that=b) #third arg has a default.*

  and *myarg = { 'this' : c , 'that' : b } dbuspath.method(myarg) print(myarg['new_key_was_a'] + ' updated: ' + myarg['that'] + ' ' + myarg['this'])*

  and *myarg.this = c myarg.that = b dbuspath.method(myarg) print(myarg.new_attrib_was_a + ' updated: ' + myarg.that + ' ' + myarg.this)*

- exchange typical standard library and other class instances as such, that is: not forcing each user to write custom code to pick out and reformat attributes into the data format, argument position order the messaging system understands. i.e for an object 'myobject' that has attributes this, that and whatnot:

  YourDBusPath.dbus_key_update_this_that_using_whatnot(myobject)

  not

  return_args = YourDBusPath.dbus_key_update_this_that_using_whatnot(helper(myobject.this),
  myobject.that.byteswap(),str(myobject.whatnot))

  myobject.this = str(return_args[0]) myobject.that = return_args[1].byteswap()

' * create a means for D-Bus service publishers / python porters to specify once then distribute to everyone everything necessary to bring D-Bus published services to 'pythonic' standards, * offer default values for missing named arguments, * build in 'future proofing' to existing code: so only changes to the translation spec are necessary to allow code for future or older versions of either the python or D-Bus side to work with available python/D-Bus code, * provide a means to add version information to methods/signals so python users can write more maintainable code, * remove the need for python users to manage distracting, non-pythonic, D-Bus-specific machinery like:

1. whether an integer is packed big or little end first, signed or not, number of bits, 1. whether a floating point number is double or single precision, 1. whether a D-Bus server is expecting a single character string to be sent as a char or a byte or a string, 1. managing the formatting of arguments to those D-Bus methods that allow argument variability from call to call (D-Bus 'variants' in python-to-D-Bus introspection strings), 1. D-Bus services using integers not to keep track of amounts but to represent a state or condition instead of the name for it (e.g. 1 instead of 'red', 2 instead of 'blue'.) 1. D-Bus services using individual bits in one integer as a collection of related boolean 'flags' (e.g. & 1 fan on, & 2 heat on, & 4 light on, &7==0 all off) 1. D-Bus services that use a few bits here and there in a larger integer as little integers representing list indexes, while using other bits as boolean flags, and others as mini-integers. (e.g.: 0x1 fan up, 0x2 fan middle, 0x3 fan down, 0x4 light on, bits 0x8,0x10,0x20 as vent angle 0 to 7) These formats are common in hardware device registers.

While this version of PyDBus does do those things, earlier versions did not. The original package gave a hint it was possible: to do all the 'plumbing' necessary to make remote service provided processing seem no different than using a typical python package. The original package did connect python attributes and methods to D-Bus properties and methods/signals, and hid the details of control and data flow on to, and off of the D-Bus. The original package pretty much stopped at the boundaries of D-Bus capabilities common to all server specifications, including D-Bus design gaps which forced 'low level' very 'un-pythonic' D-Bus byte-fiddling into higher level code (e.g. calling methods that expect variant formatted arguments).

By now it should be plain the key to this capability for each D-Bus server is the python translation guidance used by PyDBus. Summarizing this: The translation spec aims to provide enough expressive power so that no add-on functions are needed in most cases to describe how to deliver a fully pythonic D-Bus interface. When the D-Bus information is too complex, the spec provides a concise means to pass to user functions included directly in the per-service spec everything required to turn D-Bus traffic into native python results.

I offer these extensions are the natural 'next steps' needed.

***

The three levels of PyDBus users

1. Most users: Those who want to make use of already finished, fully pythonic, D-Bus capabilities. There are two basic steps: Reading through the translation guidance file for the desired D-Bus service to see examples, then get started. From here, go to [Basic PyDbus Usage](https://github.com/hcoin/pydbus/wiki/Basic-PyDbus-Usage)

1. Translation spec writers: If there is no translation specification available for the D-Bus path/service of interest, in most cases it is straightforward to write one. First read through a few translation specifications and use examples to get familiar with the look and feel. Then kindly read through the documentation in this wiki, starting with [Introduction to Translation Specifications](https://github.com/hcoin/pydbus/wiki/Introduction-to-Translation-Specifications). Your spec can be tested by passing it directly at bus access setup time. When finished, put a copy of it in the translations folder under the D-Bus path name (with _ instead of .). If the D-Bus service your spec completes (from a python perspective) is widely offered, consider sending your spec for general distribution in the project's translations folder.

1. Maintainers: Translation spec writers would find little benefit from reading the source code. The PyDBus source code operates so 'down in the engine room' there is little in it of interest to anyone but maintainers. Although I think most readers would find taking the time to understand it would generate deep insight into the capabilities that make python so popular.

***

Personal Note: This software and documentation amounts to my 'thank you' to the open source community after years of use and a few other contributions.

By [Harry G. Coin](https://www.linkedin.com/in/harrygcoin), Initial version complete: March 29, 2017.

Guidance:

Packing many meanings into a single integer evolved from concerns of memory use, communications efficiency and electrical engineering. This relieves the python user from having to keep track of 'what a bit position means', instead dealing only with importance: the meaning.

When mapping whole dbus integers to strings, at most one pair in an argument's guidance dictionary matches. However, when the pair:

{ '_is_bitfield' : True }

is included, each bit in the dbus value is, potentially, an independent flag with its own name, or part of a smaller integer used as an amount, or part of a smaller integer where each value 'stands for' or 'means' a named condition, or some combination. So, all the pairs in the guidance dictionary are evaluated, they are meant to be taken in combination. There are many allowed ways to format the argument(s) to be translated on the way to the dbus, and several options for formatting the results for python of the dbus translations. When building a dbus argument, the value starts at zero, then the matches are applied.

This documentation has four sections: Argument formats going from Python to Dbus, guidance dictionary key/value entries defining bits/labels/mini-integers of interest, special commands to do with dbus-to-python data formats and other matching controls, finally discussion and examples.

Python to DBus Argument Format: No matter the format requested when processing from dbus to python, all of these formats from python to dbus are always valid. Any string matching in the python to dbus direction is not case sensitive. If a dbus client, this direction applies to arguments in method calls (not returns), and setting properties.

- None: A zero will be sent over the dbus.
- True/False: Looks for the labels 'True' and 'False' (case insensitive) in the guidance. If there are none, it is sent as is. If there's a match, the related value is set.
- A single int where the guidance defines only one thing: a single int bit-shifting definition. The result bits will be computed per the definition and those sent. If no such guidance is in the definition, the int itself will be sent as-is. If more than one match is found in the guidance, an exception is thrown.
- A simple string that is mapped to a true/match value in the guidance. The associated value will be sent over the dbus. All other bits will be zero. An exception is thrown if the string matches nothing. Not case sensitive.
- A dictionary: The keys are the labels defined in the guidance, the values processed through the matching guidance to build the dbus result using the values. Labels specified in the guidance but not included in the dictionary have no affect on the result. Not case sensitive.
- A list or tuple: Any string list member is processed as if the tuple ('string', True). The only other allowed members are tuples or lists with at least one element, usually two. It is processed as if a dictionary where (key , value) is { key , value }. If there is only one value, True/1 is used for the second. Not case sensitive.

The DBus value sent is the result of 'oring' together the results of matching the arguments above to the guidance defined below.

Labeling bit(s) and smaller int(s) within an int argument

Below are the various ways to associate the state of one or more bits with python strings or as if independent, smaller, integers. They can be used in any combination unless stated otherwise.

Convenience Note: These guidance pairs defined below obeys the

```
{ \_from_python_to_dbus :  \<True/False\> }
```

Specification setting. If False, the default, the format is as below:

```
{ \<dbus value\> , \<python value\> },
```

If True, the order is reversed in the spec,

```
{ \<python value\> , \<dbus value\> },
```

either one can be used whether going to or from dbus. This the same as the system described elsewhere that matches a whole integer to a label. Sometimes it makes for easier reading to put the python value on the left, and the dbus equivalent on the right.

Specifics:

```
{ \<number\> , \'label\' }
```

From dbus: if (number & dbusvalue) == number, add 'label' to the list of matches. From python: if label (case insensitive) is among list to match, set to_dbus_value |= number. Typical use: 0x1 means "this is on", 0x2 means "that is so", 0x4 means "something else is on".

{ (<number off>, <number on> or None), 'label' }

or

```
{ \(\<any_on_bits\>, \<number off\>, \<number on\> or None\), 'label' }
```

- From dbus: if ((any_on_bits | dbusvalue)!=0) or ((number_off & dbusvalue)==0) and ((number_on==None) or ((number_on & dbusvalue)==number_on)), then add 'label' to the list of matches.

- From python: if label (case insensitive) is among list to match, set to_dbus_value |= number_on. If number_on ==None, use 0.
- Typical use: When 'this is off' needs its own name, beyond just the absence of 'this is on' in the match list, i.e. *{ (0,1), 'this is on' }, { (1,0), 'this is off' }*
- Also used when the presence of a zero bit voids the meaning of another, i.e.: { (0x0,0x3) , "fan high" }, { (0x1, 0x2) , "fan low" }, { (0x2, None) , "fan off" } , { (0x3, 0, 0, "fan on") } so when 0x2 is off, 0x1 is a 'don't care', it can be 0 or 1 from dbus.
- Note: If when going to dbus the 'don't care' bit needs to be a 1, then the 'todbus' and 'fromdbus' versions of the argument guidance need to be different: same labels, but with { (0x2,1), 'power off' } for the todbus side.
- Common Mistakes: A bit test line appears among the true results when the result of the test is true, which may or may not be the same as the bits named being on. So:

  (0,0) is always true, it means 'don't care what's off, and don't care what's on'. (1,0) means 'Report True/match only when bit position zero is off, don't care about the others' (-1,0) means 'Report True/match when all bits are zero' 1 appears in the 'True' results when bit 1 is on, don't care about the others. If the list is { (1,0) : 'bit zero off", 1 : 'bit zero on' } then one or the other will always be among the results, and if showing all the true/false values of results, the will both be among them.

{ <number>, ('labelfor0','labelfor1', ..) } From dbus: In short: make a 'mini integer'. Use the 1 bits in number to define the desired bit positions in the mini-int. Collect those bits in order from the dbus value, use the smaller int result as an index to the label list, add the matching value to the list of matches. Also, the non-zero bits in number need not be consecutive. * To dbus: Look up the index that matches the label (case insensitive), reverse the to-dbus math to determine the on and off status of the correct bits in number, set just those bits in the dbus value to match. * Typical use, unpacking smaller integers from a larger one:

{ 0x30, ('heat off', 'heat low', 'heat medium', 'heat high') } , ' '{ 0x3, ('fan off','fan low','fan medium','fan high') } , ' '{ (0x33,0) , 'power off' } , ' '{ (0x33,0,0) , 'unit active' }

{ (<number off>, <number>) , ('labelfor0','labelfor1', ..) }

- If dbusval&<number off> != 0, continue. Otherwise as above.

```
{ \<number\> , '\#label_for_mini_int' }
```

- As above, but instead of using the 'mini-int' as an index into a list of labels, use the value itself.
- The leading # is stripped from the label name before any use.
- If to Dbus: if the tuple or list ('label_for_mini_int',value_of_mini_int) appears, use the mini-int to reverse the math above, set those bits on the to-bus larger int.
- If from Dbus: the tuple ('label_for_mini_int',value_of_mini_int) will appear in the match list in any case, with the value ranging from 0 to the maximum allowed by the number of non-zero bits in the number.
- Typical use: when some subset of the bits in an integer represent an amount, not names for states or conditions. This is the easiest way to have a label appear no matter its state , i.e. 1 bit, on or off.

```
{ (\<number off\>, \<number\>) , '\#label_for_mini_int' }
```
* From Dbus: Same as above, but only include the match tuple at all if dbus&<number off>==0. * To Dbus: <number off> is ignored. * Note: There is an option below which, if set, ignores the <number off> value.

{ '#everything_else', '#the name of your catch-all variable' }

- Dbus to python: Treat all bit(s) not referenced in any way elsewhere as an 'and' mask for the dbus value, call the value x, assign it as the value of the tuple: ('the name of your catch-all variable', x)
- Python to Dbus: If the tuple ('the name of your catch-all variable', x) appears, dbusvalue |= x.
- NOTE: From Dbus to python: ONLY IF an #everything_else pair appears THEN: if the computed mini-int 'index' is out of range, has no label in the list, do not throw an exception but include those bits here.

Note: Everything above this line *does* obey the

`{ '_from_python_to_dbus' : \<True/False\> }`

setting, if False,

`{ \<dbus value\> , \<python value\> }`, if True, `{ \<python value\> , \<dbus value\> }`,

just as does the system that matches a whole integer to a label. However, the directives below are always as written.

DBus to Python result formats and Special Options:

Below are several options below like { 'special name', value } which if non False, changes the behavior described above.

{ (-1,0) , 'your name for all_bits_off' }

- This special pair does what it seems to do, with one extra fact, as follows:
- To dbus: no changes to the dbus outbound value.
- From Dbus: Adds the label 'name for all_bits_off' to the match set when the dbus value is 0.
- Typical Use: There is a habit that gives its own name and meaning to a zero dbus value when many of the bits are used as on/off condition flags for a collection of related parts. This is a meaning beyond each of the bits individually used as flags to mean 'this is on'. Consider a frequently found case where, each of the bits, if on, stands for some sub-component being active. When all of them are off, the recognition of the group of things sharing the off state is important. There is is a need to give that its own name, to not return an empty match list.
- SPECIAL PROCESSING: Use of this pair does not affect anything to do with '_everything_else' pair.

Dbus to Python Format and Detail Control:

These have meaning for dbus to python traffic only, usually return values from methods, signal arguments and reading properties:

{ '_show_all_names': True } * Dbus to python: consider the truth value of every pair defined, return a value for all of them, using False for those not matched, not just the true/ matched ones, according to the format specified below. By default, any pair that does not match does not have a label/value result included. So, if there are 15 single bit 'report this if that bit is on' tests, and only one bit is on, the result will have one entry. With this pair included, that example will have 15 return entries, 14 false and 1 true. If True, the return value format must be either a dictionary or a list.

{ "_arg_format" : 'dict' } * The default. Dbus results are returned as a single dictionary to python. Keys are labels, True or mini-int result is the value. True/match labels and mini-ints are always included. See '_show_all_names' above for more control over what is and isn't included.

{ "_arg_format" : 'list' } * Same as above in all respects, except a list of two value lists for what would have been a { key : value } pair. The first element the key, the second the value.The python results from dbus are, by default, returned as a dictionary[label]=dbusvalue pairs.

{ "_arg_format" : 'shortlist' } * As above for a list, but every return value that would have looked like (<entry>,<entry>, (label:True), <entry>) becomes becomes (<entry>,<entry>, label, <entry>)

{ "_arg_format" : 'single' } * Require the result to be just one label/value pair or throw an exception. Return the value.

{ "_arg_format" : 'prettydict' } * For pretty-printing. Try 'single' above. If that throws an exception, behave as 'dict'

{ "_arg_format" : 'prettylist' } * For pretty-printing. Try 'single' above. If that throws an exception, behave as 'shortlist'

Further Discussion, Rationale and Examples: **_With more detail than above, the description below is a restatement,_ _with examples and discussion._**

There is no new capability described below, if there is any conflict detected in what follows, what is above should be preferred.

The usual use of this facility is to avoid forcing the python user from having to keep track of details having nothing whatever to do with the important decisions but all about writing yet another bit-fiddling routine. So we have the usual case of '1's bit means this', '2s bit means that', '4s bit means something else'.

However, the waters can get a lot deeper than that.

Reading this, those that are new might wonder what software engineer in their right mind would pack booleans and smaller integers interspersed in the bits of a single integer? The answer is a requirement for projects 'close to the hardware' as seldom do hardware engineers use different addresses for each flag for reasons that are really obvious given some electrical engineering or data transmission background. These artifacts, totally unrelated to the meanings of the inforation involved, tend to carry over into 'C' code / device drivers, which habit bleeds onto the dbus.

Probably as far from 'pythonic' sensibility as might be.

However, there have been 'real world' setups where the state of a bit in a position several away from the first of a few materially changes the understanding of the first few. Consider a aircraft related setup where a bit near the most significant end is 'true-> display full range, false-> display idle range', and the bottom two bits 'speed, 0..3'. A reasonable map would be

'off','ground idle','flight idle', 'approach idle', 'max climb thrust', 'max continuous thrust','military thrust'

skipping over lots of bits in between to do with related topics or often just marked 'reserved'. This facility aims to let the translation structure writer take what amounts to distraction from the point of it all away from the python user's to-do list.

General Concepts:

Whereas in the last section we had:

{ <a number> : 'what the number means string' }

and, if

{ '_from_python_to_dbus' : True }

is there, then the same thing looks like:

{ 'what the number means string' : <a number> }

But if

{ '_isbitfield' : True }

is present, each number is treated as an 'and mask' so that only the bits that are a 1 in the number must also be set in the dbus value to match if from dbus, and will be set if going from python to dbus– either way, they are associated with the 'what the number means string'.

All the entries in the dictionary for this argument are considered as being part of just one composite int.

Simple 'this 1 bit on <–> this variable name':

{ 0x4 , 'what the 3rd bit position on means' }

So, including two strings in a tuple for this argument would result in the 'or' of the integer masks associated with it. Likewise on the way 'back' to python from dbus, every 'what the number means string' is included in a tuple when the variable from dbus has a non-zero value when it is 'anded' with each <a number> value.

So far, so good. But what if the number has more than one bit on? For example if the 5th and 6th bit of an integer was meant to be understood as its own number between 0 and 3? Or, the 6th bit when the 10th bit is 'on' means this or that, and when the 10th bit is off means some other two things.

Slightly more complex: '2 or more bits on <–> this variable name'

{ 0x5 , 'what to call the 0 and 2 bit when both are on' }

The 'what the number means' string now has a changed meaning. if it is a simple string, as before, then using it in an argument to dbus will turn 'on' all the associated bits. On the way back from the dbus to python, if all those bits are 'on', the string will be included in the 'answer tuple'.

It is not an error to include two entries with bitmasks that share a bit position in common. There's little point if only one bit is selected in the mask, but there are occasions when a certain 'pattern' of two or more on bits with one in common 'means' something different depending on the 'common' bit being on and the other this or that.

What if we also need certain bits to be off, and other on?

{ (0x2,0x5) , 'what to call the 0 and 2 bit on, only when the 1 bit is off'}

So far, we've discussed bitmasks that act when certain bits are on, without regard to whether other bits are off or on.

If the bitmask position is a two member tuple, an argument is recognized as matching when going from dbus to python if the [0] element, when 'anded' with the argument, must be 0 and the [1] element when anded with the argument must equal the element.

When going from python to dbus the [0] offset is ignored because the base value upon which the final dbus value is built is all 0 to begin with.

What about if all 0 means something altogether different?

{ (-1,0) , 'what all bits off means' }

Note: To include a string that appears only when all the values are 0, that is, to capture a special meaning that 'all off' has, use the pair for the bits (-1,0) which will require no bits to be on but also all bits to be 0.

What if some bits taken together is a number where each has it's own name?

{ 5 , ('what 0 means', 'what 1 means', 'what 4 means', 'what 5 means') }

How about a way create two names, one for off and another for on, do I need two lines like { (4,0) , "4 is off" } and { 4, "4 is on" }? No.

{ 4 , ('what 4 off means', 'what 4 on means')}

If the 'what the bits mean' string is a not a string, but a tuple or list of strings? Then the first string will be equivalent to 'all 1 bits off', the second 'the least significant bit on', the third 'the bit next left of the last one in the mask on', and so forth. So if the two bits in the mask are consecutive, it would seem like an entry for a typical integer with a name, number pair for 0, 1, and 2. But it would be the same if the mask was for the 3 and 9 bit position as well– a four value tuple for both off, 3 on, 9 on, 3 and 9 on.

Now, most of the time, there's just going to be one bit on in the mask, so the first entry in the tuple is 'off' or 'false', and the next 'on' or 'true'. When used in arguments, these string entries will appear as any other members of the tuple which together makes up all the bits of the integer.

This 'string tuple' option is the only way to cause a tuple to be in the argument list when an 'off' condition exists.

Q: Can I have two entries that refer one way or another to the same bit(s)?

A:Yes, but be really sure there's no other way. And test the code with all possibilities because the results are often not obvious.

Q: What if when going from python to dbus, one guidance entry requires a certain bit to be both on, and another off?

A: On wins.

Q: Suppose one entry uses a bit in a number, but another uses the same bit in an and match test? It works, but take care if providing names for each number there are enough names to cover all the cases.

Q: I want one variable name to hold more than 'this matches' like a small number, or 0/False, 1/True. A 'mini-integer' out of certain bits? I don't want a name for off and another for on. Or four different names to express two bits. One only.

A:

{ <some bit or bits> : "#Result as 0 based integer" }

We see often in hardware device i/o control registers, some of the bits 'stand for' this or that condition, but a couple here or there are actually to be understood as simple unsigned integers but with a much smaller maximum? Even just one bit to be understood not as on/off but used as the integer range(0,2)

In these cases, instead of 'what the number stands for' as a string which will only deal with bits as such described above, or a tuple of strings wherein the first one 'means' 0, the second entry 'means' 1, etc. begin the string with a #. so

{ 0x6 , '#number between 0 and 3' }

The name of the variable when used does not include the leading #.

Q: Is there a 'catch all' way to collect in one place any bits not otherwise defined?

A:

{ '_everything_else', '#the name of your catch-all variable' }

If instead of a bit mask or tuple for (off bits, on bits), the string '_everything_else' is used, then it expects a single #nameforthatvariable as the name.

When going from python to dbus, if in a list or simple string the value will be 0 (it will be ignored). In a dictionary or tuple, the value will be 'ored', as is, with the final integer computed above.

When going from dbus to python, any bit that is not mentioned anywhere else in the dictionary will mask the to python variable and the result returned as the value for this key.

Note: a *{ (<these must be 0>, <these must be 1>) , "Here's the name for that" }* entry removes all both the 0 and 1 bits mentioned above from the 'everything else' bin: EXCEPT { (-1,0), "all off name" } which does not affect the 'everything else' variable content.

Why? This feature exists because sometimes dbus values are mapped to hardware registers meant to be read, changed, then written, set up with bits that are to be 'unchanged' upon writing but not defined by the manufacturer.

Just including this key in the bitfield guidance and then using the same argument dictionary on the way in and out takes care of this situation.

Q: Is there a way to alert if there are bits set that aren't described in the list?

A: Yes. Test for the _everything_else variable to be non-zero when going from dbus to python. So this can be used as a check to alert there are non-zero bits that weren't otherwise defined in the guidance.

When going to the dbus, any bit not explicitly set to non-zero will be zero. When going to the dbus, variable names are not case sensitive.

Bitfield Argument Formats:

There are several ways to structure the argument when traversing from python to dbus.

Python to Dbus:

All these formats are always recognized.

- One string: If passed a string, only the 'on'

bits for that string are set. If a #variety variable representing an integer appears, 0 will be the value used for it.

- A list or tuple containing only strings: If passed a list or tuple, all the on bits associated with each member

are 'ored' together. If a string that represents a mini integer appears, 0 is used as the value.

- A list containing any mix of strings or (name,value) tuples: If in a string list or tuple, a list a sub-list or sub-tuple appears as a member, the [0] entry is taken to be the name of the variable, and the [1] entry the

value to use.

- A dictionary: If passed a dictionary from python, the names for the variables are the keys

and the values are the state of that key. 0/False/None, 1/True, 2, 3, 4...

What if I leave a key out that has a definition? 0/False will be used.

What if I put in two values that affect the same bits? We assume you meant it. Be sure if such is coming from dbus to python there is at least one entry in the guidance that will match the combined ored result.

When going from python, unrecognized strings throw exceptions.

When going from dbus to python for 'mini integers' wherein each value 'stands for' a string, and there is no entry in the list for the dbus value returned, none of the string names will appear, and the #variablename associated with '_everything_else' if such exists, will include those bits. If there is no _everything_else variable, and when there is no entry in a list for a dbus variable included in a bit mash: an exception is thrown.

There are a few special cases left to define.

Dbus to Python:

When passing arguments to dbus from python, almost all formats are accepted. But, with so many possibilities, what can I expect the dbus to return in these situations?

The default to-python argument format is a dictionary that:

{ "_arg_format" : 'dict' }

'Just the True things':

> will always have a { 'variable_name' : True } entry for on or (off, on) variables that match.
>
> will always have a { 'variable_name' : <integer value> } entries for variables that have a #variable_name specification in the dictionary, even if the value is 0.
>
> will always have a {'variable_name' : True } entry for exactly one of the names in a list given to coorrespond 1 <–> 1 to a mini-integer.
>
> That's the default. It is possible to change that.
>
> 'Everything, either way':
>
> It in the guidance the pair { '_show_all_names': True } appears, then the dictionary returned will list every variable named in the guidance, with value False or True, or if a mini-integer that value.
>
> 'I want a list with possible tuple pairs for mini-integers':

{ "_arg_format" : 'list' }

If the above appears, a tuple or list, even if there is only one thing in it, or possibly nothing in it will be returned. Mini-integers appear as tuples within the list (variablename , mini-int value)

'I know there's only ever going to be exactly one match, either exactly one mini int, or zero or one variable matches. I don't want a container. I just want to be able to set and read a property of this sort using the variable name and let this thing figure out what bit to fiddle it in.

{ "_arg_format" : 'single' }

If the above appears, the list above is computed, and if it has length 1 the [0] value is returned, if length 0 -> None, otherwise an exception is thrown. Very useful for those situations where it is certain that at most one bit in the field is

to be on. In this case, if no bits are on, 'None' is returned to python from dbus. In this case the name associated with a mini-ints used as a number will be included iff the number is !0.

If *{'_arg_format' : 'shortest' }*

Then if _return_as_list : 'single' throws an exception, return_as_list = True is used.

{ "_arg_format" : 'shortlist' }

Then as above for a list, but every return value that would have looked like (<entry>,<entry>, (label:True), <entry>) becomes becomes (<entry>,<entry>, label, <entry>)

'I want the shortest possible non-dictionary result, it's all for pretty printing and I don't want to show containers of one thing':

{ "_arg_format" : 'prettydict' }

For pretty-printing. Try 'single' above. If that throws an exception, behave as 'dict'

{ "_arg_format" : 'prettylist' }

For pretty-printing. Try 'single' above. If that throws an exception, behave as 'shortlist'

Argspec Format

Previous sections covered: loading translation specs per dbus path, identifying which keys in that path need translating, then isolating per key the whether the data flow is to or from the dbus and whether for a method, property or signal. For each of those last elements, this section describes how to manage the argument list for that purpose.

Managing the argument list, the argspec format:

Note: when an argument to/from dbus is described as 'unchanged' below, take **that** _to mean 'what pydbus would have done had no translation been requested', **not_** _'nothing'_

Specifically: When flowing from dbus to python, what the GLib **function** _variant.unpack() creates is either passed unchanged through the translation process, or passed to these **routines_** _for translation then passing that on. When flowing from python to dbus, 'unchanged' means whatever the Glib'**s_** _Variant parser does with the to-dbus direction introspection string and the caller's arguments (all ints are introspection 'i', not 'u' or byteswapped, or..). If translation is active, it accepts those values as input. What the **glib_** _library does in these functions is often correct, and on other occasions needs the attention of **these_** _routines)._

The format for managing the argument list is same for all argument directions and for methods, properties and signals.

The Argspec Dictionary

The argument specification dictionary has key / value pairs as follows:

{ <argument position number (leftmost is 0)> : <single argument translation guidance> , ... }

Like:

{ 0 : translation_guidance_for_arg0 , 2 : translation_guidance_for_arg2, ...}

The translation guidance value may be None, or as described in the next section.

So,

{ 1 : single_argument_translation_guidance }

would leave arguments 0, 2 and up unchanged, and would use 'single_argument_translation_guidance' defined in the next section, to manage the translation for argument 1.

Alternatives for readability and convenience:

- If the argument processing specification is not a dictionary, but instead a list or tuple, it is processed as if each member is a dictionary pair above:

{ list_index : list_content_at_index }

So if the argument specification is

(argspec_0, argspec_1, None, argspec_3)

The leftmost argument and the one next to it would be translated further. The one after that skipped, and the next translated, and any more skipped. An equivalent dictionary as described above is:

{ 0 : argspec_0 , 1 : argspec_1, 3 : argspec_3 }

- Anything other than a dictionary, None, list or tuple:

So many argument lists have just one argument, for example, all properties. For readability, it is possible to pass a single argument argspec itself: not in a list or dictionary or tuple. It is the same as if passed

{ 0 : argspec }

A dataflow dictionary entry for this sort of argspec (previous document) might look like this:

{ "property", argspec }

Avoids syntax clutter, improving spec readability.

Adding arguments on the Python side:

The above specification calls for an integer corresponding to the position of the argument in the list to be the key, with the value being the argspec translation guidance for what to do with it. If the value for the key is -1, then the argument is added to the python side (only when using non-default named arguments, see the next section). But the value of these added arguments never traverses the dbus. The next section describes a way to specify default values for arguments. The combination of those two features makes it possible to send to upper level code details such as the version number of the translation specification, dummy values for dbus services specified but unimplemented, and more.

Pydbus Int Autotranslation

by Harry G. Coin, Quiet Fountain LLC, 2/2017

dbus/'C' ints as states or flags <–> python strings or tuples

tl;dr:

::

Many flags packed in an integer:

```
>>> print(dbus.interface.property_boolean_packed_int)
```

was >>> 0x1001

becomes >>> ('what1000means','what1means') was >>> dbus.interface.property_flag = 0x1002

becomes >>> dbus.interface.property_flag = ('what1000means','what2means')

Int value not used as a number but representing a state or condition:

```
>>> print(dbus.interface.property)
```

was >>> 1234

becomes >>> 'whatcondition1234means' was >>> dbus.interface.property=1234

becomes >>> dbus.interface.property='whatcondition1234means'

Same for arguments to/from methods and from signals.

No changes to anything else in the pydbus tutorial except: bus.get(... ,translation_spec=your_spec_here)

See_tests/nmdefines.py for details.

Aim:

The autotranslator extension to pydbus takes the package further in its aim to make dbus entirely and natively 'pythonic'. It makes integer variables passed in dbus that 'mean something other than a number' into python variables that 'mean as they appear': strings or tuples.

Secondary Benefits:

1. Collect into exactly one readable place all obscure relationships layered on integers that 'mean other things'.
2. Avoid requiring all source files that use dbus to include all the definition myvar_isblue = 1, myvar_isred = 2, etc. files.
3. Avoid repetitive kludgey code that 'looks up' printable strings from variables and interprets strings into 'the int the dbus partner needs'
4. Create a standard for translating 'c include headers' into python.
5. Avoids implementing partial variable interpretations in multiple files.

Use Case Details:

dbus integers better represented as one member of a tuple

Many dbus interfaces represent variables as integers such that each integer value has a specific meaning. There are thousands of 'include file'.h or similar files written in C and its variants that have section after section that read something like:

:: int myvariable; #define myvariable_isred 1 #define myvariable_isgreen 2 #define myvariable_isblue 3

or

::

typedef enum { myvariabletype_isred=1, myvariabletype_isgreen=2, myvariabletype_isblue=3 } myvariabletype;

myvariabletype instance1, instance2

A quick look at gnome's various dbus interface specifications has many such examples.

When used as dbus properties, or arguments to or results from dbus methods, or passed as information to signal callbacks, there is always extra work to render the information in printing or to translate the information often in xml

text or similar to the right number. What this extension does is to burn a dictionary of sorts right into pydbus so that all the conversions from integers-as-state-info to/from 'the shortest string that defines the state' happens before the user has to deal with it.

so:

```
from what_was_a_c_dot_h_file import mycondition_isX, mycondition_isY, mycondition_isZ
    ...
    bus=SessionBus()
    interface=bus.get("what.not")
    mycondition = interface.someproperty
    if mycondition==mycondition_isX: do_this()
    elif mycondition==mycondition_isY: do_that()
    ...
    choice = get_from_user("desired condition?")
    if choice == 'string that means X': num_condition = mycondition_isX
    elif choice == 'string that means Y': num_conditon = mycondition_isY
    else: print("Not found...")
    interface.someproperty=num_condition
    >>> print(lookup_what_mycondition_means(mycondition))
    >>> 'myconditon_isZ'
```

:: with this extension becomes

```
bus=SessionBus()
interface=bus.get("what.not",translation_spec=your_spec_here)
work = { 'X': do_this, 'Y': do_that }
work[interface.someproperty](args)
...
choice = get_from_user("desired condition?")
try:
   interface.property=choice #not case sensitive
except:
   print("Not found...")
>>>  print(interface.someproperty)
>>> 'Z'
```

:: Once the interface is defined using the translation spec, it can be referenced elsewhere without importing all the int-to-meaning files.

v2.0 will allow the values to be arbitrary objects that support ==, not just strings.

Dbus integer as (different power of 2 for each item)*(item true/false value) –> tuple

Dbus's C legacy has many instances where the individual bits in one integer stand for a collection of boolean variables that are related to the state of some activity. Often termed 'flags'. We see many examples like:

```
#define myvariable_isdead 0x0
#define myvariable_iswarm 0x1
#define myvariable_isrunning 0x2
#define myvariable_isready_for_whatnot 0x4
int myvariable /* see above */
myvariable = myvariable_iswarm | myvariable_isrunning
```

While that's more 'readable' than myvariable = 3, it can't be rendered as a string without code that's been written and rewritten in almost every program and that has to maintained separately from the definitions. And, even when 'automated' still has a preprocessor that 'has to be run' that 'looks for changes in the code' and 'writes them to an output module' etc. etc.

This extension to pydbus allows for a natively python, and logically cleaner, way to relate to dbus with these variables. The examples above become

```
pydbus_interface.myproperty = ('ready_for_whatnot','Warm') #not case sensitive
```

:: which this extension would translate invisibly as 5 when being passed into dbus, and ('ready_for_whatnot','warm') when being received from dbus.

The Translation Specficication

Check the nmdefines.py file in the tests directory for full details.

In summary, the translation spec is a python dictonary with a key that is the dbus.interface.spec and whose value is a dictionary. That dictionary has keys which are the names of properties, methods or signals that have integers used as flag bitfields, or integers that represent one condition for each value, and have as values a tuple that describes what to do when that key is used as a method's arguments, a method's return value, a signal or a property.

The 'what to do' item is usually just a dictionary that looks up the number of the key as the name of a condition. If the item is used for a signal or method that takes many arguments, then it is a tuple with 'none' if the argument is to be passed through unchanged, or the name of a simple 'name':'number that goes with it' dictionary. There's a special entry in the dictionary if the integer is to be used as a collection of on/off,true/false flags.

That's it!

See the __main__ entry at the end of pydbus.translator.py for tests and examples. See the file test/nmdefines.py for a completely finished definition for Gnome's NetworkManager, with extensive stage-by-stage instructional comments.

License

translation_spec = { 'keyname' : <dataflow spec> ,...}

The DataFlow Specification

This level resolves two general conflict areas. The first is the possible use of one key name for any combination of signal, property or method.

The second is the possibly different translation needs of arguments flowing from python code to the dbus such as when a service client calls a method or sets a property, or

from the translation needs of arguments flowing back from the dbus to the caller. In this case return arguments to the function call, the value of a property being read, or the arguments to a user provided signal function.

When publishing a service, the dbus to python are arguments TO the method. python to dbus are the return values FROM the method. So, the data flow is reversed.

The allowed key : value pairs in the dataflow level dictionary are:

{ 'method_py_to_dbus': to_dbus_method_argspec_described_below, *#used when calling a method or publishing a reply*

'method_dbus_to_py': from_dbus_method_argspec_described_below, #used to process method return values or accept published method arguments

'signal_dbus_to_py': from_dbus_signal_argspec_described_below, #used to accept information when receiving a signal

'signal_py_to_dbus': to_dbus_signal_argspec_described_below, ' '#used when sending a published signal

'property': property_dbus_argspec_described_below #used both for setting and reading a property

'property_dbus_to_py': from_dbus_property_argspec_described_below 'property_py_to_dbus': to_dbus_property_argspec_described_below

#used in rare cases when a property needs a different set and get translation

}

Note: In the rare case there is a need for a different spec for when getting and another for setting a property, the keys: property_dbus_to_py and property_py_to_dbus can be specified, and if so, they override any value set in property.

All the ...argspec... objects mentioned above as values follow the same specification, described in the next section.

It is often the case that a dbus service does not offer properties or signals or methods. There need only be entries for offered services.

It is often the case there is data flow from dbus to python for a signal, property or method, but not from python to dbus for the same method. There need only be entries for cases where there is data flow. However, if there is nothing in a particular data flow that needs translation services, (For example, a property key that accepts and sets only a string), that key-value pair may be omitted.

When there is no translation service needed for an entire direction for a signal, key or method, while it is usual to leave out the related name, it may be included with a value of None.

Any other key/value pairs are ignored, but may be used in the future.

A sample program fragment to access the Connectivity method of the NetworkManager, jumping ahead a bit:

' from pydbus.bus import SystemBus' *bus=SystemBus() ' 'nm=bus.get('org.freedesktop.NetworkManager',translation_spec={ 'Connectivity' : { 'method_py_to_dbus' : <argspec for the call arguments, defined next section> })*

' 'method_dbus_to_py' [\<argspec for the return arguments, defined next section\> })'] *print(str(nm.Connectivity))*

would print a string describing the connectivity (only if the <argspec...> is fully defined). But

from pydbus.bus import SystemBus nm=bus.get('org.freedesktop.NetworkManager') print(str(nm.Connectivity))

would print an integer.

The next section describes whether to translate none, some or all of the arguments flowing across the dbus in a particular direction for a particular key : the argspec section.

Python to DBus Variants

Guidance Item: { '_variant_expansion' : <vinfo> [,<vinfo>]* } *Where* **<vinfo> :: <guidance>[/<guidance>]*[.<repeated tail>] and <guidance>:: 'an introspection string with no variants' <repeated tail>:: 'a convenience, appends 'repeated tail' to each guidance'**

Typical Use: Specify all possible argument formats to accept for a 'v' in an introspection string when going from Python to DBus

Discussion: There exists a bothersome meaning asymmetry in the dbus architecture.

It arises when going from python to dbus when the dbus introspection string contains a 'v' (meaning GLib Variant). Variant is pretty close to the python idea of object, the actual content could be anything the DBus service can parse for that key.

This is not a problem when going from dbus to python, as each variant sent over the D-Bus has within code that describes what it is about in precise format detail, and so has the information necessary to convert it to a native python type. This is done automatically by GLib unpack routines before the translation functions are called. In fact this list is what pydbus returns when no translation is asked for, and the one supplied as input to the translation routines.

So, from the Dbus, it is baked in to the message how to set it all ready for the next level to deal with.

Not so going from python to dbus if the published dbus specification has a 'v' / variant. While technically the DBus will accept nearly anything it can express within a 'v' and ship it over to the DBus service: Each service can parse only specific patterns, which patterns are known only to the service.

Without using this feature at all, pydbus will parse the arguments and take reasonable guesses to build then send a correct DBus message. Should you be very confident the other side can handle any legal dbus message matching the overall spec, you don't need to read any further.

However:

1. What if the remote service accepts only 32 bit unsigned integers? This is not the default format for a python int. The service will throw an error message when it gets 'ii' instead of 'uu'.

1. The introspection string is a 'v', the argument is an int. Is the recipient expecting a signed or unsigned value? An 8, 16, 32, 64 or other sized integer? Big endian or little endian?

1. The introspection string is a 'v', the argument is a floating point number. Is the recipient expecting a double or a float?

1. The introspection string a{sv} or a{sa{sv}} is often seen in dbus server specs even when there is but one legal way for the types the v's can contain that the recipient is capable to understand in the message.

This feature provides pydbus the guidance necessary to transform the python arguments into the specific variant forms expected by the remote DBus partner. As there can be many possible legal argument formats, the first specification that matches the provided python argument structure is chosen. If none match, an error is thrown.

Implementation Details:

1. To dbus 'v' values MUST be translated by the full dictionary formatted per argument guidance entities. The shorthand guidance forms mentioned before that in this document will throw an exception.

1. The very first top level dictionary translating an argument that includes an introspection string with a 'v' SHOULD have something like a { "_variant_expansion" : 'this/that/theother','what to do with next v', etc }. Before argument processing begins, the overall expansion is built by extending the list. So, It's up to you whether to include everything in the _variant_expansion key/value pair in arg[0], or spread it out over arguments. For code re-use it's probably best to include a string in the associated argument with a v.

However it gets built, the whole "_variant_expansion" directive to understand any v in the passed in introspection string is applied against the entire string before any argument processing occurs.

Detail: By 'applied against the passed introspection string', is meant the translation guidance for an argument is no longer passed just a 'v', but (say for two legal ways to parse a v):

'v:<guidance>/<guidance>:'

If there is no _variant_expansion key, or not enough to match all the v's demanded in the introspection string for the call, the v will be replaced in the translation specification v:: and pydbus will take it's best reasonable guess how to fill the variant from the arguments provided.

Note that v:...: is treated as a single introspection string unit, like u or s. The unmodified introspection string is retained for all uses outside these routines.)

Note: If the arguments are to be managed by function calls using { '_match_to_function' : True } then it is up to that function to manage all the parsing possibilities and details. Nothing further in this section applies, however the introspection string passed to the external function will be the modified v:: one for its use.

Further examples:

The explanation below uses phrases like 'a{sv}' is parsed as as a{su}'. While that is so, the actual data passed to dbus does not pass a 'u', it passes a variant that contains a 'u'

All this is so because without guidance, a python list with two integers would be by default packed as ai, when it is completely possible the recipient is expected au and rejects the ai (e.g. the network manager parsing a network address/prefix combo).

So the string {'_variant_expansion' : "u/s,s/(i)"} applied against the introspection provided string "ava{sv}' would be broken into two argument calls: av and then a{sv}. This would attempt to parse the first argument with information av:u/s: first as an au, and if that failed, an as. In either case the result would be an array of variants with only the content differing.

The second would be parsed first as a{ss} and if that failed a{s(i)}, again with the ..s or ..(i) would actually be packed as a variant containing an s or a variant containing a tuple holding ints.

If there is the '_variant_expansion' key, and the value is anything other than a string, or is v:: the system will attempt a default parsing. Lists become arrays, tuples are tuples, dictionaries are arrays of dictionary entries, strings of any length, including single characters, are strings, floating points are doubles and ints are unsigned 32 bit values, boolean is bool, and None argument elements are parsed as 0, false, '' etc.

If this parsing is acceptable, just include {'_variant_expansion' : None }

If a <vinfo> is empty, e.g. ',i' the first variant would get the default parsing above (functions would see v::), the second would parse a signed integer and convert it to variants containing an int.

Last, just as a convenience, instead of writing axyz/bxyx the same can be written a/b.xyz

Forced replacement of a variable with a constant

Guidance Item: { '_forced_replacement' : <whatever> }

Typical Use: Ignore the argument supplied, return <whatever>.

This is a convenience. Instead of having to write match_to_function methods that do nothing but 'zero out' values such as for which upper level code are not capable to understand, use this. Just a space saver.

Detail:

From python to dbus, <whatever> replaces the value supplied at call time. If more sophistication is required, use the _match_to_function facility.

From dbus to python, ignore the dbus value returned and replace it with <whatever> to the python caller.

Discussion:

Handy for occasions argument positions hold per-run constants. Version numbers, hostname strings, system property or capability, etc. With this, the user only need bother with actually possibly varying arguments.

The system will correctly handle None or False as an argument. If present the variable will be replaced with None or False.

Argument Specification Guidance Dictionaries

The previous sections documented argument specification capabilities available when the argument specification is a string or list or tuple, just convenience readability shorthand for the full specification here. Provided by the shorthand when the specification is:

1. A string: names the variable position, so that *myclass.x = 234 myclass.y = 567 myclass.z = 890 myclass.x, myclass.y, myclass.z = path_object.method(myclass.x,myclass.y,myclass.z)*

   becomes:

   myclass.data['x']=234 myclass.data['y']=456 myclass.data['z']=890 path_object.method(myclass.data)

   Or, almost equivalently:

   path_object.method(x=234,y=456,z=890)

   The presence of any keyword argument changes the argument style from a single dictionary to the python keyword style argument list.

   However: Whereas in the single dictionary argument case, if a translation spec gives an argument position the same name on the call and return side, the entry in the dictionary passed will be updated. That is not available when using the keyword style.

   The point of these two possibilities (and the attribute based possibility below) is:

   1. To offer the typical 'python' suite of argument passing styles, but also: 1. Give the translation writer the ability to provide a python class that tightly integrates one or more related dbus services. By carefully choosing matching names for method call and return arguments, a 'call by reference' capability is simulated.

1. A list or tuple: for integers that 'stand for' something else so that *myclass.property = lookup_number_for['all good'] ' 'print(myclass.property2) ' '14*

   becomes:

   myclass.property = 'all good' print(myclass.property2) 'what 14 means to me'

The full capability of the per argument translation specification is delivered when using a dictionary as the translation guidance argspec follows.

(Recap: An 'argspec' describes what to do with a particular argument position when used specifically as an argument or as a return value for a particular named method, signal or property of a particular key for a specific dbus path e.g. y in bdus_path_object.key_that_object_publishes(x,y,z).

The following dictionary specification outlines all available translation options. There are many further convenience and readability features, not least using the same argspec for setting and reading properties, and for method arguments which return values of the same sort. The details follow.

The general form of the dictionary argspec is:

dictionary_argspec = { <directive name> : <directive setting> , ... *#Then one of the following two styles: <dbus value match info> : <how python should convert it>, ... #or <python value match info> : <how dbus wants to see it>, ... }*

1. Argument format: default positional, named via dictionary, and class attributes.

To offer full python capability, Dbus translations must be able to offer users more than the default positional arguments. Pydbus offers two other ways to define method and signal arguments (and add optional context to properties as well): dictionary style and attribute style.

1. Default: 'traditional' argument meaning by position in argument list.

By default, the methods and signals expect the arguments on the 'python side' of any dbus operation to to be the usual list with the position of the argument in the list defining its purpose. The DBus side always uses positional arguments as defined by the dbus service spec. If this is desired, nothing further need be added to the argspec translation guidance.

1. Dictionary Style:

   If the pair:

   { '_dictkey' : 'name for what the argument in this position is for' }

   is included, it is expected a single dictionary argument will be passed and returned on the python side no matter the number of arguments defined by the dbus service. That all the arguments for this key have no conflicting settings.

   In that single dictionary argument, the keys are the names given in the above _dictkey pairs for each argspec position, the values are what would have been in the respective argument position had the default list style been used.

   There are three benefits: Giving the same name in a function call and function return argspec will cause a data item to be passed to the method then updated automatically upon return. The other benefit is changes can be made to the dbus service or the user's code regarding the number and purpose of the arguments to the method without forcing refactoring of the user's code. Last, by using a dictionary instead of object attributes to name arguments, related groups of arguments can be kept in one place at arm's length from the method, akin to a 'structure' in other programming languages. This can ease some database and other disk read/write issues.

Note: If even one argument specification sets the dictionary style, the entire **argument** _list changes from the default list to the dictionary style._

1. Attribute Style:

   If the argspec dictionary has the pair:

{ '_attributename' : "attribute for this argument"}

   Only a single class instance is passed on the python side. The arguments to the dbus partner will be read from the object using the attribute name above to identify the appropriate dbus argument position.

   So:

myclassinstance.x = dbus_path_object.method(myclassinstance.a,myclassinstance.b,myclassinstance.c)

   becomes:

dbus_path_object.method(myclassinstance)

   Note: If even one argument specification sets the '_attributename' style, the entire argument list changes from the default list to the attribute style.

   It is an error for _attributename and _dictkey both to appear anywhere in the specifications for one argument list, or within the specification of one argument.

   While, technically, this could be done for a property, the occasions to do so seem few as there ever is only one argument.

The argument naming feature, combined with passing class instances to methods and signals (or dictionaries instead of argument lists), makes it possible for the python programmer to not have to remember argument order. It's also a boost to maintainability, since service spec changes can be expressed in the translation guidance and translation spec without disturbing established code, or code able to run on different versions of the service.

Remember, all attributes of objects used in these calls that are not mentioned in the guidance are ignored. In that way, complex objects upon which many different dbus methods might offer services can use the same instance so long as the attributes names don't conflict.

Special Note for translating Methods

Only when using the "_attributename" style calling procedure in the guidance on BOTH the call side and the

return side of a method call, will the same object will be populated by the return call as was passed to the method. This is the default.

The idea being if the same attribute name appears in both guidance structures associating (probably different) argument positions to the same name, the return call value will replace the value used on the way in. If the names differ, the return side of a method call will add/update other attributes to that object.

Only when using the "_dictkey" style calling procedure the default is that a NEW dictionary will be returned to hold the responses to a method call.

HOWEVER: If the pair

‘ {"_new_return_instance" : False }‘

appears in the guidance for a method call on both the call side specification AND the return side, the dictionary or object passed as an argument on the call side will be populated by the return values.

If the pair

‘ {"_new_return_instance" : True }‘

appears in the guidance on either the calling or returning argument spec, the response to a method call will be a new dictionary or a new class with only the return values as keys / attributes respectively.

Of course, if using the traditional list of arguments on either side of a method call, or not matching attribute / dict style arguments then the _new_return_instance will be treated as False, even if it is otherwise specified.

This feature has no meaning for signals or properties, since these are ever only sending information one way per call.

Default Values for Named Arguments

In order to duplicate the python capability of defaults in the case named variables do not appear in a dictionary or as an attribute in the information passed in on the python side, the argument dictionary pair

‘ { "_default" : <whatever> }‘

Will cause the missing argument to be added by pydbus with the default named in the pair as the value. Otherwise, arguments specified in the translation but missing when used are defaulted to None (which is usually an error on the dbus).

This is useful for maintaining the functioning of legacy code when new arguments are added to dbus keys that older code does not supply.

Specific Translation Capabilities

So far, almost everything has been described except the actual business of changing the python side argument to be compatible with the dbus side, and back. Calling conventions, default values, and a few convenience features one of which translates an integer that ‘stands for’ something into the thing it "stands for" and back.

Described now is the actual business of deciding whether to change a specific argument, and if so, to what and how.

The Default Translations:

Each argument presents to the translator with an initial value, and the portion of an ‘introspection string’ the dbus service indicates should direct the format and nature of the argument. [Click here for introspection string usage information](https://dbus.freedesktop.org/doc/dbus-specification.html#type-system)

**Unless specifically stated otherwise below, the translation routines pass the data as presented along without change.**

When the argument is a string, or a boolean, or otherwise have but one representation on both the dbus side and the python side, there is no need to provide any further detail (unless condensing several arguments into or from a new object instance, see below). However, there are the following capabilities for the other situations:

- When the argument is a list or tuple

Absent any further direction, when a list ('array' in dbus parlance) appears as an argument, or a tuple appears, the default is to pass the list or tuple with all its contents unexamined along unchanged. However, if the pair

' { "_container" : < another entire per_arg_translation_spec > }'

appears, this will cause the given whole new argspec translation specification to be applied to every element of the container. The introspection string passed along will reflect only the container contents. The secondary argument specification is formatted exactly as is this, but any directives to do with argument format are ignored. This can be nested to any necessary depth.

The resulting argument will be a container of the same sort the argument was, in the same order.

- When the argument is a dictionary

Absent any further direction, when a dictionary appears as an argument ('array of key value pairs' in dbus parlance), the dictionary is passed along unchanged. However, if the pair

' { "_container" : < another entire per_arg_translation_spec > }'

appears, the new translation specification will be applied **ONLY TO THE VALUES** of the keys in the dictionary.

If there is a translation activity to be applied to the keys of a dictionary argument, the entry

' { "_container_keys" : < another entire per_arg_translation_spec > }'

Will apply the named guidance to the keys of the dictionary, while applying the argspec named in "_container" to the values. If it is not desired to translate the values but only the keys, omit the "_container" entry.

The introspection string passed along to the key translation argspec will be just that of the key value in the dictionary, likewise the value/container argspec will be passed the string covering just the values.

Some may notice there is no facility to operate on the keys and values as a pair. There is, in the form of user supplied functions, see below. Any other approach seemed, 'unpythonic' as it were.

Convenience and Readability Feature:

At a level the translation writer expects a container argument (tuple, list, dict) to appear for translation, and the argument specification at that point expects an integer to 'stand for' some other meaning (usually a string), the translator will act as though:

{ "_container" : <whatever was there that wasn't a dictionary> }

was written. That is, the shorthand will be applied to each member of a list or tuple, and the values if a dictionary. So, if the pla
in a dictionary, just include the spec for the values. Note: the integer to label mapping applies

only one level, to the members of the immediate container which the dbus expects to be an integer and the python side an equivalent label (usually string), not further containers. If the same mapping is used in more than one place in a specification, then define it outside the translation structure then use that variable name as the value of the related container arguments.

Wrapping it up

There is, of course, much more. Everything left has to do with particular translation situations and the capabilities available for acting on them. Those are documented online. Check the links in the documentation for such as "TS Guidance ..." and "TS Special Case".

Documented there are new specific purpose translation capabilities, including the specific argspec guidance necessary to engage them. For example, how to give names to flags as individual bit positions in an integer, how to connect integers with values that 'stand for' something else, and more.

Remember: any argument given to the translator is passed through unchanged if no translation specification has been written for it **EXCEPT**: When going from python to DBus and the introspection string has even one 'v' in it.

If that's the case, read [TS Special Case: Define Python To DBus Variants](https://github.com/hcoin/pydbus/wiki/TS-Special-Case:-Define-Python-To-DBus-Variants). Whether by inattention to detail by dbus service publishers or a gap in the dbus specification: translation routines must be engaged in these situations, there is no other way to create the necessary dbus message. That is, without forcing upon the pydbus user the need to comprehend GLib packing details and other code to do purely with below the water line machinery to the meanings involved).

In these cases it is almost always best if the translation writer supplies information as to what formats the dbus service can handle without error when the argument is marked as 'variant' (determined at run time). The above document describes that process.

Per Argument Position Specification

This document begins the seemingly 'main work' of the translation module: What to do with a particular argument in a call with possibly other arguments, to a method, signal, or a property, knowing whether the direction of the data is to or from python re: dbus traffic. These are the values of the dictionary keys <argspec> in the previous section, they provide translation guidance, just called 'guidance' below.

Everything in this document is a convenience readability shorthand that accomplishes a subset of what a full translation dictionary argument specification (described in the next document) can do.

NOTE: If the direction of data is from python to dbus, and the introspection _string contains a 'v', be sure to read "Special Cases 2" below after_ _reading the main definition._

Each of the following sections is a convenience and readability shorthand for a more capable but harder to read dictionary format described in the next section. At the top of each section is summary statement, how the capability would be expressed if using the full dictionary spec (presented later).

- Argspec entry: None

  None - Do nothing, as if the request to process this argument was omitted.

- Argspec entry: string or equivalent. Names the argument position. Changes the argument format to dictionary.

  Summary: any non container argument, almost always a string, is equivalent to { '_dictkey' : str(non_container_arg) }

  If guidance is a non-container type (int,string..), name the argument position. Also causes the arguments flowing to python to be in the form of a dictionary with keys using these names, values translated from the dbus, this in place of a positional argument list.

  Rationale: Here we define an optional facility to free the python programmer from retaining the distracting knowledge of which argument goes in what dbus argument position, and also allows naming each of the dbus arguments used in a method or signal (or property, but as there is only ever one, it's redundant to the property name).

  When calling methods, the dictionary used to create the argument list on the way to the dbus server is the same as used on the way out. If the purpose of the method is to update some argument, give that argument the same name in both the translation specifications.

  Restated: If the return value of a method is the updated version of a value passed as an argument, give it the same name. Then the user will not have to write busy-work code assigning output variables.

One might imagine a class with many well named variables then passing vars(instance) as the argument to any number of dbus methods without further ado, trusting the translation to ignore whatever it doesn't need and arranging what it does need in the order the dbus routine wants and in the format it's looking for.

Not so useful for single valued signals, methods and properties, but very useful for more complex calls.

Detail: Let's call the passed in non-container (almost always string) guidance value 'argument_name'.

When going from python to dbus, instead of the usual arguments expected by the dbus recipient, expect exactly one dictionary with members:

{argument_name : <the variable that would have been in that argument position> ,...}

The translation routine will call the dbus proxy with an argument list built with the values in the dictionary ordered in the variable position associated with 'argument_name' in the translation spec.

Note: If even ONE translation argument_name position is specified, any missing specifications for other arguments will be replaced by an integer equal to the 0 based index of that argument position as the dictionary key and the translated dbus value as the argument when going to python, and

Any omitted argument position/name when going from python to dbus is called as if { <argument position> : None } was included.

Example:

So, a dbus method that takes/returns two arguments, with a translation spec:

' { 'MyDbusKeyNameLikeAddConnection', {'method_py_to_dbus': { 1 ['foo' }}}, '] *#'foo' is not a container, so it is the name used for argument position 1 {'method_dbus_to_py': { 0 : 'bar' } } } ' '#'bar' is the name used when argument position 0 is meant.*

would expect as an argument from python to dbus not the arguments themselves in a fixed order, but instead one dictionary with

' { 0 : <whatever arg 0 is expected> , ' ' 'foo' : <what would have been next after the first argument> }'

n.b. if the 0: .. key value pair was omitted, the original pydbus function would be called with arguments

(None , <what would have been next after the first argument>)

and return from dbus to python a dictionary (not a tuple or list):

' { 1 [<whatever the response in arg position 1 was> , ']

'bar': <whatever the response in arg position 0 was}

n.b. the spec did not include a name for argument position one, so the translator uses the argument position number as the default name.

Note: when defined in this way, the variable content itself is as pydbus would have done with no translation.

Moving on:

1. Argspec entry: List or Tuple – When number 'means' a string value

   Here is a 'convenience feature' that does in a shorter form what the full dictionary structure could do with more typing:

   Summary: Guidance (a,b,c) or [a,b,c] is shorthand for

   { 0 : a, 1:b, 2:c, "_from_python_to_dbus" : False }

If the guidance is a tuple or a list for a argument position, and not a full dictionary spec, when going from dbus to python, replace the dbus argument with guidance[dbusargument]. If evaluating guidance[argument] results in an exception, None is passed to python.

When going from python to dbus, replace the python argument with dbarg such that argument = guidance[dbarg]. In cases where more than one argument results in the same guidance[argument] the result will be the highest argument. If the string from python has no member in the tuple, an exception is thrown.

n.b. The 'inverse' map (in which python arguments result in dbus integers used when going from python to dbus is computed once when the translation structure is first passed. It is this way:

inverse_map = { guidance[arg] : arg for arg in argument}

Feature: If an element in the list or tuple is a string, when going from python to dbus the string is not case sensitive. When going from dbus to python, the result is capitalized as in the list or tuple.

Remember: This specification is ALWAYS given as a tuple or list with members being the python representation, and offset being the dbus equivalent — even if the only time it's used is when going from dbus to python.

The list/tuple facility above is the easiest way to specify short more or less one to one maps between a reasonable python object and what dbus is looking for that 'stands for' or 'means' that object. Most often it's just ('what 0 means','what 1 means', ...) when the list is short with no gaps, (or None is used to fill a gap and you know what you are in for)

Note however, there is no way give the argument position described in this shorthand a name, so doing this shorthand blocks the use of passing arguments as dictionaries.

Recap:

So far, we have a way to give a name to an argument and otherwise leave it unchanged. This changes the python side of all routines from passing a list of arguments to passing a single dictionary. It has as keys the names given and lets these routines keep track of which one goes in what dbus argument position.

We have a way to swap integer arguments that aren't used for arithmetic with strings that describe the situation (instead of a number that means a string the user has to keep track of that describes a situation.)

Coming up:

A way to spare the python user having to figure out how to pack and unpack which bit in an integer 'means what'. We can also give these argument positions names to enable passing methods and signals a single dictionary instead of keeping track of which argument goes in what spot.

What if the argument is something we want to do translation work on, but it is a container type? List, Tuple or dict? What if it is a container that has other containers inside it?

Guidance

- **Implemented by pair { <dbus int-like item> : <what that item means in python> }**

  Unless optional pair: { '_from_python_to_dbus' : True } Then: { <what that item means in python> : <dbus int-like item> }

- **Optional: { '_replace_unknowns' : (<string if int missing>,<number if string missing >) }**

Conveniences: * Can use the same guidance dict whether data flow is from or to DBus. * Python to dbus only: If a python arg is an int, pass it as is, no lookup. * If no _replace_unknowns key: missed dbus lookup yields 'UNKNOWN_0x<hex number>' * String "UNKNOWN_0x<hex number>" always sends <hex number> to dbus.

Typical Use: Integers when not used as amounts, but to 'stand for a state' or 'express a condition'.

Also when an int is used as a collection of related booleans packed into an integer when at most one can be set, each a different power of 2 to avoid conflict.

Translation Dictionary Example:

'**SimplePath = { 'KeyNameForLocation'** [{ 'property'][' ' { 0 : "at a store" },' ' { 17 : "in my office" },' ' { 102 : "at the airport" },' ' { 23 : "at home" } }}'

from pydbus.bus import SystemBus 'bus = SystemBus() '

#New Way mydbuspathobject = bus.get('my.dbus.path', translation_spec = SimplePath) ' 'mydbuspathobject.KeyNameForLocation = 'in my office' assertEqual(mydbuspathobject.KeyNameForLocation, 'in my office') mydbuspathobject.KeyNameForLocation = 102 assertEqual(mydbuspathobject.KeyNameForLocation, 'at the airport')

#Old Way mydbuspathobject = bus.get('my.dbus.path') assertEqual(mydbuspathobject.KeyNameForLocation, 102) mydbuspathobject.KeyNameForLocation = <any illegal int>

Discussion:

This feature is most often used, and most often used this way:

In the case each expected value of integer 'stands for' a meaning described in a string, create the dictionary this way if specifying what should happen when going FROM dbus and TO python:

> { <dbus returned value key> : <pythonic interpretation of returned value> , ...}

so

> { 1234 : 'whatever1234means' }

With that, if one does pydbusitem.property='whatever1234means', it will result in a dbus call replacing that argument with the integer 1234.

pydbusitem.property == 'whatever1234means'

Migrating away from usage imposed by C

Style suggestion: Many 'C include files' have entries like

> typedef { PROPERTYNAME_MEANING1 = VALUE1, PROPERTYNAME_MEANING2 = VALUE2 } ITEMNAME;

Consider a translation entry like PROPERTYNAME = { "MEANING1" : VALUE1, "MEANING2" : VALUE2 } stripping off the repeated parts of the definition. Python is much better about namespaces. This avoids repetitive string content and eases 'pretty printing'.

Likewise typical 'include' file content like *#define THE_BLUE_FAN_OFF 1 #define THE_BLUE_FAN_LOW 2 #define THE_BLUE_FAN_HIGH 3 #define THE RED_FAN_OFF 4 #define THE_RED_FAN_LOW 5 #define THE_RED_FAN_HIGH 6*

assigned to such as: *int RedFanStatus; int BlueFanStatus;*

consider renaming to the least redundant possible form:

RedFanStatus = { 4 : "Off" , 5 : "Low", 6 : "High" } BlueFanStatus = { 1 : "Off" , 2 : "Low", 3 : "High" }

Convenience, writing { string : number } instead of { number : string }

If the dictionary includes the key value pair

{ '_from_python_to_dbus' : True }

then the entries are understood as being not {1234, 'whatever1234means'} but rather {'whatever1234means' : 1234 }

For what little it is worth, If {'_from_python_to_dbus' : False } is included, the default first understanding is used.

NOTE: When used as a key, 'whatever1234means' is case insensitive. This is to make pretty printing not interfere with whatever capitalization the dbus partner is using. So write key names that will appear in the preferred display capitalization format.

Handling End Conditions:

If when coming from dbus a value has no matching key entry relating it to a python string, "UNKNOWN_0x" where X is the hex rep of the dbusvalue is returned.

When going to dbus from python, the value "UNKNOWN_0x..." is parsed always and the hexidecimal number after the x used as the value.

HOWEVER: if the dictionary has the key, value pair

{ '_replace_unknowns' : (<string>,<number>) }

though UNKNOWN_0x.. will be parsed when going TO the dbus as described above, strings with no matching guidance values will not throw an exception but instead send the number above. FROM dbus will return the string argument above (Which can be anything, including 'None') and not "UNKNOWN_0x..." when a number comes from dbus that is out of range of the name tuple as the field.

Without a { 'replace_unknowns'...} as above, if a pythonic string value has no mapping as a dbus integer, an exception is thrown.

Feature: TO DBUS: If there is reason not to define every possible value as a string, just pass the integer itself and that will be transmitted as is to the dbus partner.

WARNING: If providing the same dictionary to both the to and from dbus routines and letting this module compute the inverse for the one not in the format required: If two different keys result in the same value, it is NOT DEFINED which of the keys the value will return when (if ever) that dictionary is used in the 'other' direction.

Guidance Items

Guidance Item: { "_all_arguments" : True }

Typical Use: express one python object as multiple D-Bus arguments.

With this omitted or False (the default): if a 'from D-Bus' message came in providing introspection string 'uu', translation guidance spec [0] would be given the first u integer to work on, translation guidance spec [1], the second u, and each translation result returned in the same position to the python routine.

If { "_all_arguments" : True } appears, translation guidance spec [0] would be passed [arg0, arg1] and introspection string (uu). The result returned is always a list, whatever the length, it is unpacked and sent on to Python. Similarly to D-Bus. See below.

Discussion:

Here we provide a means to manage occasions when the information necessary to transform D-Bus traffic into a class object isn't specified by the D-Bus publisher as a single container class with all the necessary in one argument, but requires information spread across more than one D-Bus argument.

For example, consider a python standard library network address and prefix class instance. These often traverse the D-Bus as an unsigned integer for the address packed often in the reverse byte order than most machines out there, followed by another integer expressing the number of bits that are part of the network part of the address.

If the D-Bus service publisher chose to express that using format 'au', (array of unsigned), then a single translation specifying a function to manage converting both numbers into a class instance would be simplest. One argument to the function, one result from the function. Typical.

However, what if the D-Bus service publisher chose to express that using format 'uu'. One argument position for the ip address, and the next argument position for the network prefix?

Ordinarily, with the default, or guidance { "_all_arguments" : False } there would be two translation specifications, one for each 'u' argument position, and then some manner of quite hack-ish glue to associate the processing of one function to the other in order to create the one class instance that represents both correctly.

Instead of that shortcoming, in these occasions use the guidance pair

{ "_all_arguments" : True }

which changes argument processing as follows:

Case - Information flowing from D-Bus, to Python (arguments returned by methods, sent by signals for D-Bus service clients and reading properties):

The very first translation routine will not deal with the first argument position, the second the next and so on as usual. Instead, the very first translation routine will be passed all the arguments from the D-Bus packaged in a single in-order list, and with the entire introspection string.

Translation processing commences, which always results in a list (though the length may differ) passed on to the python code. The length will only differ in the case the user specified a match function as well in this guidance (the usual case, combining D-Bus arguments into fewer python object instances).

If no match function is specified, all this will accomplish is allowing ordinary translation guidance to cause the python result to fill what would have been the first return argument instead with a list of all of them. Useful if one desires to express many D-Bus values as a single property being read by a python caller.

Case - Information flowing from Python, to D-Bus (arguments to methods, sent to properties for D-Bus service client):

All the arguments coming from Python are combined into one list passed as one argument to the first translation specification. If there is no match function there to separate class instances into (usually) more arguments, the list coming from the translation will be unpacked in order then sent as the requested D-Bus arguments. If a match function is used to accept the list, however long the list is that it returns are unpacked in order as the dbus arguments.

Note that even if this is used for single valued methods/etc, the input will be a one member tuple and the output is expected to be a tuple.

Shorthand Specs

This document begins the seemingly 'main work' of the translation module: What to do with a particular argument in a call with possibly other arguments, to a method, signal, or a property, knowing whether the direction of the data is to or from python re: dbus traffic. These are the values of the dictionary keys <argspec> in the previous section, they provide translation guidance, just called 'guidance' below.

Everything in this document is a convenience readability shorthand that accomplishes a subset of what a full translation dictionary argument specification (described in the next document) can do.

NOTE: If the direction of data is from python to dbus, and the introspection _string contains a 'v', be sure to read "Special Cases 2" below after_ _reading the main definition._

Each of the following sections is a convenience and readability shorthand for a more capable but harder to read dictionary format described in the next section. At the top of each section is summary statement, how the capability would be expressed if using the full dictionary spec (presented later).

- Argspec entry: None

  None - Do nothing, as if the request to process this argument was omitted.

- Argspec entry: string or equivalent. Names the argument position. Changes the argument format to dictionary.

  Summary: any non container argument, almost always a string, is equivalent to { '_dictkey' : str(non_container_arg) }

  If guidance is a non-container type (int,string..), name the argument position. Also causes the arguments flowing to python to be in the form of a dictionary with keys using these names, values translated from the dbus, this in place of a positional argument list.

  Rationale: Here we define an optional facility to free the python programmer from retaining the distracting knowledge of which argument goes in what dbus argument position, and also allows naming each of the dbus arguments used in a method or signal (or property, but as there is only ever one, it's redundant to the property name).

  When calling methods, the dictionary used to create the argument list on the way to the dbus server is the same as used on the way out. If the purpose of the method is to update some argument, give that argument the same name in both the translation specifications.

  Restated: If the return value of a method is the updated version of a value passed as an argument, give it the same name. Then the user will not have to write busy-work code assigning output variables.

  One might imagine a class with many well named variables then passing vars(instance) as the argument to any number of dbus methods without further ado, trusting the translation to ignore whatever it doesn't need and arranging what it does need in the order the dbus routine wants and in the format it's looking for.

  Not so useful for single valued signals, methods and properties, but very useful for more complex calls.

  Detail: Let's call the passed in non-container (almost always string) guidance value 'argument_name'.

  When going from python to dbus, instead of the usual arguments expected by the dbus recipient, expect exactly one dictionary with members:

  {argument_name : <the variable that would have been in that argument position> ,...}

  The translation routine will call the dbus proxy with an argument list built with the values in the dictionary ordered in the variable position associated with 'argument_name' in the translation spec.

  Note: If even ONE translation argument_name position is specified, any missing specifications for other arguments will be replaced by an integer equal to the 0 based index of that argument position as the dictionary key and the translated dbus value as the argument when going to python, and

  Any omitted argument position/name when going from python to dbus is called as if { <argument position> : None } was included.

  Example:

  So, a dbus method that takes/returns two arguments, with a translation spec:

' { 'MyDbusKeyNameLikeAddConnection', {'method_py_to_dbus': { 1 ['foo' }}}, '] *#'foo' is not a container, so it is the name used for argument position 1 {'method_dbus_to_py': { 0 : 'bar' } } } ' '#'bar' is the name used when argument position 0 is meant.*

  would expect as an argument from python to dbus not the arguments themselves in a fixed order, but instead one dictionary with

' { 0 : <whatever arg 0 is expected> , ' ' 'foo' : <what would have been next after the first argument> }'

n.b. if the 0: .. key value pair was omitted, the original pydbus function would be called with arguments

(None , <what would have been next after the first argument>)

and return from dbus to python a dictionary (not a tuple or list):

‘ { 1 [<whatever the response in arg position 1 was> , ‘]

‘bar’: <whatever the response in arg position 0 was}

n.b. the spec did not include a name for argument position one, so the translator uses the argument position number as the default name.

Note: when defined in this way, the variable content itself is as pydbus would have done with no translation.

Moving on:

1. Argspec entry: List or Tuple – When number ‘means’ a string value

   Here is a ‘convenience feature’ that does in a shorter form what the full dictionary structure could do with more typing:

   Summary: Guidance (a,b,c) or [a,b,c] is shorthand for

   { 0 : a, 1:b, 2:c, “_from_python_to_dbus” : False }

   If the guidance is a tuple or a list for a argument position, and not a full dictionary spec, when going from dbus to python, replace the dbus argument with guidance[dbusargument]. If evaluating guidance[argument] results in an exception, None is passed to python.

   When going from python to dbus, replace the python argument with dbarg such that argument = guidance[dbarg]. In cases where more than one argument results in the same guidance[argument] the result will be the highest argument. If the string from python has no member in the tuple, an exception is thrown.

   n.b. The ‘inverse’ map (in which python arguments result in dbus integers used when going from python to dbus is computed once when the translation structure is first passed. It is this way:

   inverse_map = { guidance[arg] : arg for arg in argument}

   Feature: If an element in the list or tuple is a string, when going from python to dbus the string is not case sensitive. When going from dbus to python, the result is capitalized as in the list or tuple.

   Remember: This specification is ALWAYS given as a tuple or list with members being the python representation, and offset being the dbus equivalent — even if the only time it’s used is when going from dbus to python.

   The list/tuple facility above is the easiest way to specify short more or less one to one maps between a reasonable python object and what dbus is looking for that ‘stands for’ or ‘means’ that object. Most often it’s just (‘what 0 means’,’what 1 means’, ...) when the list is short with no gaps, (or None is used to fill a gap and you know what you are in for)

   Note however, there is no way give the argument position described in this shorthand a name, so doing this shorthand blocks the use of passing arguments as dictionaries.

   Recap:

   So far, we have a way to give a name to an argument and otherwise leave it unchanged. This changes the python side of all routines from passing a list of arguments to passing a single dictionary. It has as keys the names given and lets these routines keep track of which one goes in what dbus argument position.

We have a way to swap integer arguments that aren't used for arithmetic with strings that describe the situation (instead of a number that means a string the user has to keep track of that describes a situation.)

Coming up:

A way to spare the python user having to figure out how to pack and unpack which bit in an integer 'means what'. We can also give these argument positions names to enable passing methods and signals a single dictionary instead of keeping track of which argument goes in what spot.

What if the argument is something we want to do translation work on, but it is a container type? List, Tuple or dict? What if it is a container that has other containers inside it?

Short Examples

Useful background documentation:

PyGI: https://wiki.gnome.org/Projects/PyGObject

GLib: https://developer.gnome.org/glib/

girepository: https://wiki.gnome.org/Projects/GObjectIntrospection

Examples

Send a desktop notification

```
from pydbus import SessionBus

bus = SessionBus()
notifications = bus.get('.Notifications')

notifications.Notify('test', 0, 'dialog-information', "Hello World!", "pydbus works :)
↪", [], {}, 5000)
```

List systemd units

```
from pydbus import SystemBus

bus = SystemBus()
systemd = bus.get(".systemd1")

for unit in systemd.ListUnits():
    print(unit)
```

Handle flags and states natively

```
    from pydbus import SystemBus
    from tests.nmdefines import PydbusNetworkManagerSpec,NM_DBUS_INTERFACE,NM_DBUS_
↪INTERFACE_DEVICE
```

```
bus=SystemBus()
#old C way
nm=bus.get("org.freedesktop.NetworkManager",'Devices/0')["org.freedesktop.
↪NetworkManager.Device"]
print(str(nm.Capabilities) + ", "+str(nm.DeviceType))
#7, 14

#pythonic way
nm_trans=bus.get(NM_DBUS_INTERFACE,'Devices/0',translation_
↪spec=PydbusNetworkManagerSpec)[NM_DBUS_INTERFACE_DEVICE]
print(str(nm_trans.Capabilities) + ", "+str(nm_trans.DeviceType))
#('NM_SUPPORTED', 'CARRIER_DETECT', 'IS_SOFTWARE'), GENERIC
```

Start or stop systemd unit

```
from pydbus import SystemBus

bus = SystemBus()
systemd = bus.get(".systemd1")

job1 = systemd.StopUnit("ssh.service", "fail")
job2 = systemd.StartUnit("ssh.service", "fail")
```

Watch for new systemd jobs

```
from pydbus import SystemBus
from gi.repository import GLib

bus = SystemBus()
systemd = bus.get(".systemd1")

systemd.JobNew.connect(print)
GLib.MainLoop().run()

# or

systemd.onJobNew = print
GLib.MainLoop().run()
```

View object's API

```
from pydbus import SessionBus

bus = SessionBus()
notifications = bus.get('.Notifications')

help(notifications)
```

More examples & documentation

The Tutorial contains more examples and docs.

Top Level Specification

The top level translation specification is a dictionary. It can be passed directly within the bus.get(...,translation_spec=myspec), or in a translation file for dbus path my.dbus.path:

pydbus/translations/my_dbus_path.py

Which contains:

''Everything the user needs to be successful with this translation.''

my_dbus_path = { #translation spec for my.dbus.path *'key name of signal, property or method 1': { <a dataflow spec> }, 'key name of signal, property or method 2': { <another dataflow spec> }, 'key name of signal, property or method 3': { <yet another dataflow spec> },*

#... }

The argument to translation_spec MUST be a dictionary or 'None' or omitted (same as 'None'). The keys of the dictionary are the names of the methods, properties or signals for which translations are provided, one entry no matter the same name be used for any combination of signal, method or property.

If you are creating a dbus service, think twice about using the same name for these things.

The next level of spec is the dataflow dictionary.

Translation File Specfication

Restating some of the 'getting started' document, but more formally, a typical pydbus startup pattern is:

' from pydbus.bus import SystemBus' *bus=SystemBus() # or SessionBus() ' 'xx=bus.get(...usual arguments...,translation_spec=<TranslationDictOptions>,method_return_format=<byname or argpos>,method_inarg_format=<byname or argpos>)*

Where <TranslationDictOptions> is one of:

- True: Look in pydbus' built in translations subdirectory for a module

whose name matches the dbus path with _ replacing . and from that module, use the dictionary in it with the same name as the module. e.g. org_freedesktop_NetworkManager

- 'module_name': import the module 'module_name', use the

dictionary with name matching the dbus path with _ replacing . So 'my_module' containing dictionary org_freedesktop_NetworkManager = { ... }

- translation_dictionary: A dictionary object in the format described in the next section.
- **'method_return_format' and 'method_inarg_format' is one of:** per_pydbus_spec_only dict_if_service_provides_argnames dict_if_service_provides_2_or_more_argnames (default)

  These are essentially about whether and how to use the argument names provided directly by the service.

  method_return_format controls whether method calls are returned as a dictionary of name:value pairs, or a single value, or a tuple. method_inarg_format controls whether method calls are to expect a single value, a tuple, or dictionary of name:value pairs.

per_pydbus_spec_only will ignore argument position names provided by the server using only what's in the translation spec.

dict_if_service_provides_argnames will expect/return a dictionary of name:value pairs if the service gives names for argument positions. Otherwise it will use what's in the translation dictionary.

dict_if_service_provides_argnames will expect/return a dictionary of name:value pairs if the service gives names for argument positions. Otherwise it will use what's in the translation dictionary.

dict_if_service_provides_2_or_more_argnames is as above, but will expect as the translation spec provides if there is but one argument. It makes little sense to use argument names when there is only one either in or out. An argument name:value pair dictionary (of length 1) will still be used if there is a translation spec that provides for it.

pydbus tutorial

Author Linus Lewandowski

Based on python-dbus tutorial by Simon McVittie, Collabora Ltd. (2006-06-14)

Date 2016-09-26

This tutorial requires Python 2.7 or up, and `pydbus` 0.6 or up.

Contents

Connecting to the Bus

Applications that use D-Bus typically connect to a *bus daemon*, which forwards messages between the applications. To use D-Bus, you need to create a `Bus` object representing the connection to the bus daemon.

There are generally two bus daemons you may be interested in. Each user login session should have a *session bus*, which is local to that session. It's used to communicate between desktop applications. Connect to the session bus by creating a `SessionBus` object:

```
from pydbus import SessionBus

session_bus = SessionBus()
```

The *system bus* is global and usually started during boot; it's used to communicate with system services like systemd, udev and NetworkManager. To connect to the system bus, create a `SystemBus` object:

```
from pydbus import SystemBus

system_bus = SystemBus()
```

Of course, you can connect to both in the same application.

For special purposes, you might use a non-default Bus using the Bus class.

D-Bus objects

D-Bus applications can export objects for other applications' use. To start working with an object in another application, you need to know:

- The *bus name*. This identifies which application you want to communicate with. You'll usually identify applications by a *well-known name*, which is a dot-separated string starting with a reversed domain name, such as `org.freedesktop.NetworkManager` or `com.example.WordProcessor`.
- The *object path*. Applications can export many objects - for instance, example.com's word processor might provide an object representing the word processor application itself and an object for each document window opened, or it might also provide an object for each paragraph within a document.

  To identify which one you want to interact with, you use an object path, a slash-separated string resembling a filename. For instance, example.com's word processor might provide an object at `/` representing the word processor itself, and objects at `/documents/123` and `/documents/345` representing opened document windows.

All objects have methods, properties and signals.

Setting up an event loop

To handle signals emitted by exported objects, or to export your own objects, you need to setup an event loop.

The only main loop supported by `pydbus` is GLib.MainLoop.

GLib.MainLoop

To create the loop object use:

```
from gi.repository import GLib

loop = GLib.MainLoop()
```

To execute the loop use:

```
loop.run()
```

While `loop.run()` is executing, GLib will watch for signals you're subscribed to, or accesses to objects you exported, and execute correct callbacks when appropriate. To stop, call `loop.quit()`.

Accessing exported objects

To interact with a remote object, you use a *proxy object*. This is a Python object which acts as a proxy or "stand-in" for the remote object - when you call a method on a proxy object, this causes dbus-python to make a method call on the remote object, passing back any return values from the remote object's method as the return values of the proxy method call.

Obtaining proxy objects

To obtain a proxy object, call the `get` method on the `Bus`. For example, NetworkManager has the well-known name `org.freedesktop.NetworkManager` and exports an object whose object path is `/org/freedesktop/NetworkManager`, plus an object per network interface at object paths like `/org/freedesktop/NetworkManager/Devices/eth0`. You can get a proxy for the object representing eth0 like this:

```
from pydbus import SystemBus
bus = SystemBus()
proxy = bus.get('org.freedesktop.NetworkManager',
                        '/org/freedesktop/NetworkManager/Devices/0')
```

pydbus has implemented shortcuts for the most common cases. If you start the bus name with "." (".NetworkManager"), "org.freedesktop" will become automatically prepended. If you specify a relative object path (without the leading "/"), the bus name transformed to a path format will get prepended ("/org/freedesktop/NetworkManager/"). If you don't specify the object path at all, the transformed bus name will be used automatically ("/org/freedesktop/NetworkManager"). Therefore, you may rewrite the above code as:

```
from pydbus import SystemBus
bus = SystemBus()
dev = bus.get('.NetworkManager', 'Devices/0')
```

Object API

To see the API of a specific proxy object, use help():

```
help(dev)
```

To call a method:

```
dev.Disconnect()
```

To read a property:

```
print(dev.Autoconnect)
```

To set a property:

```
dev.Autoconnect = True
```

To subscribe to a signal:

```
dev.StateChanged.connect(print)
loop.run()
```

connect() returns a Subscription object with a disconnect() method, that can be used to stop watching the signal. Also, it can be used as a context manager (with the ‘’with” statement), to automatically disconnect at the end of the scope. Alternatively, you can set the on‘Signal‘ property:

```
dev.onStateChanged = print
loop.run()
```

This way, you can unsubscribe from the signal by setting the property to None.

However, don’t mix subscriptions in one of those ways with unsubscribtions in another, it won’t work.

See also

See the examples in `examples/systemctl.py` and `tests/gnome_music.py`.

Interfaces

D-Bus uses *interfaces* to provide a namespacing mechanism for methods, signals and properties. An interface is a group of related methods, signals and properties, identified by a name which is a series of dot-separated components starting with a reversed domain name. For instance, each NetworkManager object representing a network interface implements the interface `org.freedesktop.NetworkManager.Device`, which has methods like `Disconnect`.

An object may have multiple interfaces. They may be incompatible, for example when using some sort of API versioning. By default, pydbus merges all the interfaces to offer a single proxy object’s API, but it’s possible to obtain a view providing only a single interface:

```
dev = bus.get('.NetworkManager', 'Devices/0')
dev_api = dev['org.freedesktop.NetworkManager.Device']
```

You may use all of the proxy object members described in the previous chapter on the dev_api too.

Signal matching

You may also match the signals using a pattern. See `help(bus.subscribe)` for more details.

Exporting own objects

Objects made available to other applications over D-Bus are said to be *exported*.

To export objects, the Bus needs to be connected to an event loop - see section *Setting up an event loop*. Exported methods will only be called, and queued signals will only be sent, while the event loop is running.

Class preparation

To prepare a class for exporting on the Bus, provide the dbus introspection XML in a ''dbus" class property or in its ''docstring''. For example:

```
from pydbus.generic import signal

class Example(object):
  """
    <node>
      <interface name='net.lew21.pydbus.TutorialExample'>
        <method name='EchoString'>
          <arg type='s' name='a' direction='in'/>
          <arg type='s' name='response' direction='out'/>
        </method>
        <property name="SomeProperty" type="s" access="readwrite">
          <annotation name="org.freedesktop.DBus.Property.EmitsChangedSignal" value=
↪"true"/>
        </property>
      </interface>
    </node>
  """

  def EchoString(self, s):
    """returns whatever is passed to it"""
    return s

  def __init__(self):
    self._someProperty = "initial value"

  @property
  def SomeProperty(self):
    return self._someProperty

  @SomeProperty.setter
  def SomeProperty(self, value):
    self._someProperty = value
    self.PropertiesChanged("net.lew21.pydbus.TutorialExample", {"SomeProperty": self.
↪SomeProperty}, [])

  PropertiesChanged = signal()
```

If you don't want to put XML in a Python file, you can add XML files to your Python package and use them this way:

```
import pkg_resources

ifaces = ["org.mpris.MediaPlayer2", "org.mpris.MediaPlayer2.Player", "org.mpris.
↪MediaPlayer2.Playlists", "org.mpris.MediaPlayer2.TrackList"]
MediaPlayer2.dbus = [pkg_resources.resource_string(__name__, "mpris/" + iface + ".xml
↪").decode("utf-8") for iface in ifaces]
```

Object publication

To publish an object, use the `bus.publish` method:

```
bus.publish("net.lew21.pydbus.TutorialExample", Example())
loop.run()
```

Here, publish() both binds the service to the net.lew21.pydbus.TutorialExample bus name, and exports the object as /net/lew21/pydbus/TutorialExample.

Note, that you can use the publish() method only once per a bus name that you want to bind. However, you can use it to export multiple objects - by passing them in additional parameters to the method:

```
bus.publish("net.lew21.pydbus.TutorialExample",
  Example(),
  ("Subdir1", Example()),
  ("Subdir2", Example()),
  ("Subdir2/Whatever", Example())
)
loop.run()
```

The 2nd, 3rd, ... arguments can be objects or tuples of a path and a object. `bus.publish()` uses the same path-deducing (and bus-name-deducing) logic that's used in `bus.get()`, so you may use relative paths or absolute paths, depending on your needs.

Like `signal.connect()`, `bus.publish()` returns an object with an `unpublish()` method, that can be used as a context manager.

See also

See the example in `examples/clientserver/server.py`.

Lower level API

Sometimes, you can't just publish everything in one call, you need more control over the process of binding a name and exporting single objects.

In this case, you can use `bus.request_name()` and `bus.register_object()` yourself. See `help(bus.request_name)` and `help(bus.register_object)` for details.

Data types

Unlike Python, D-Bus is statically typed. Each method and signal takes arguments of predefined types; each method returns value(s) of predefined types; and each property has a predefined type. You can't dynamically change those types.

D-Bus has an introspection mechanism, which `pydbus` uses to discover the correct argument types. Python types are converted into the right D-Bus data types automatically, if possible; `TypeError` is raised if the type is inappropriate.

Container types

D-Bus supports four container types: array (a variable-length sequence of the same type), struct (a fixed-length sequence whose members may have different types), dictionary (a mapping from values of the same basic type to values of the same type), and variant (a container which may hold any D-Bus type, including another variant).

Arrays are represented by Python lists. The signature of an array is 'ax' where 'x' represents the signature of one item. For instance, you could also have 'as' (array of strings) or 'a(ii)' (array of structs each containing two 32-bit integers).

Structs are represented by Python tuples. The signature of a struct consists of the signatures of the contents, in parentheses - for instance '(is)' is the signature of a struct containing a 32-bit integer and a string.

Dictionaries are represented by Python dictionaries. The signature of a dictionary is 'a{xy}' where 'x' represents the signature of the keys (which may not be a container type) and 'y' represents the signature of the values. For instance, 'a{s(ii)}' is a dictionary where the keys are strings and the values are structs containing two 32-bit integers.

Return values

If a D-Bus method returns no value, the Python proxy method will return `None`.

If a D-Bus method returns a single value, it will be returned directly.

Otherwise, Python proxy method will return a tuple containing all the values.

License for this document

User Supplied Translation Functions

So far, there's a way to pass an argument through without change, or optionally, do translation work on it if it is an integer that means something else, or optionally, work on the contents of a list or tuple dictionary values and/or dictionary keys.

What if the translation requirement is more complex for an argument? For example, an integer that needs to be packed or unpacked as an internet address? Or some other translation function that can do all it needs to do using one argument?

For those cases, the translation specification for each dbus path can include its own translation functions.

Note: If a user supplied translation function requires access to more than one argument to work, such as making a class instance out of several arguments, or vice versa, read [TS Special case: Convert one object instance to from many DBus arguments](https://github.com/hcoin/pydbus/wiki/TS-Special-case:-Convert-one-object-instance-to-%5C-from-many-DBus-arguments) then return to this document ### Calling User or Standard Translation Functions

If key,value pair:

‘ { ‘_match_to_function’ : True } ‘

exists in the translation dictionary for an argument, any other routines that could have changed the argument are ignored, and the following processing replaces it:

This is the facility whereby dbus arguments are converted into python class instances more complex than the above, and vice versa. also any special argument filtering or custom translations can occur.

If this pair is in the dictionary, processing proceeds as follows:

First: The argspec keys are searched for an exact match to the dbus/Glib format substring that is meant to describe the argument the translation is to deal with. So if the format string that is an exact match for the argument (Such as ‘au’ as ‘array of unsigned’), treat the value string as either a directly callable function, or: if not callable, then a tuple of strings like (‘the module to import’,’the callable function in it’)

‘ So: { “au”, (‘import_module_argument’,’function to deal with au’)}‘

and if directly callable:

‘ (name)(argument,argument_index,dbus_format,to_dbus)‘

using the function’s return value as the argument value returned to the caller.

So if the user supplied a function

‘def foo(arg,idx,fmt,direction): ‘ *return str(arg)+” is special”*

an an argspec dictionary key value pair (along with the _match_function : True pair above):

{ ‘s’ , foo } ‘ ‘# or { ‘s’, (‘module_with_foo’,’foo’}

function foo would be called as

foo(‘string argument’,1,’s’,True)

This, assuming we are dealing with the argument next right from the first. and the dbus variable replaced with whatever foo returns.

So in this case, every simple string would have ‘ is special’ added to it both on the way to and on the way from dbus.

Note: The dbus format for a particular argument is just that portion of the whole dbus introspection string that is to match that argument. So ‘us’ would apply ‘u’ to the first argument, and ‘s’ to the second argument.

If the function can deal with a container, then it can be passed containers and match such things as ‘au’. Howvever, it’s perfectly legal to use the description above to manage containers and if this function call feature is specified within a container, each element will be a separate call and the format string would be just for that element.

So in the ‘au’ case, if called to process the elements of the a (list) container, the function would be called for each element in the list and with the format guidance ‘u’.

Exactly likewise if going from python to dbus. The argument to foo will be the python supplied one, the return value passed to dbus, except the to_dbus variable will be false. However, it is good maintainability to be able to write one function that deals with going to or from a python class to keep like semantics together.

Second:

If there is no key in the dictionary matching the dbus introspection string, all the argspec keys are treated as regular expressions, and the introspection string is modified to have the argument index as a prefix:

str(argument_index)+’:’+dbus_introspection_arg_format_substring

The system will match the keys against the string as formatted above, stop at the first match and call respective function as above.

So: ‘translation_spec = { ‘dbus_path_key_name’ : { method_py_to_dbus : ‘

'simplematchfunction' :

{ 'method_dbus_to_py' :

{ 0 : { "_match_to_function" : True, "." : ('my.module.to.import.with.foo','foo') }}}}*

would convert every return value to the string version in the example above.

If no regular expression matches, an exception is thrown.

For performance, if there is a choice, it's always preferable to use specific introspection strings the related functions are able to handle.

Some functions are included and available by default. Only the value below is relevant, the match keys used are possible suggestions only:

'{ '2:u', pydbus.to_ipv4_address } '

converts unsigned 32 bit byte-swapped integers to the python ipaddress.IPv4Address class, only if this is argument index 2 (third from left).

'{ 'au', pydbus.to_ipv4_network } '

converts the first unsigned value in the array as the network address of an IPv4Network class with prefix length = the second value without regard to which argument index this is.

Other functions are: * pydbus.to_ipv6_address – converts a byte-swapped byte array to the ipv6 address class. * pydbus.to_ipv6_network – converts the tuple (byte-swapped byte array address,prefixlength) * pydbus.to_ipv6_address_list – converts [{'prefix':num},{'address':byteswappeduint32},.. to [IPv6Address, IPV6Address, ...]

etc. To see the full list, look in the translations directory in the pybus folder.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search
- Sourcecode