An Introduction to 'D-Bus' in Perl

Using Perl and the Net::D-Bus Bindings Module

Author:

Madison Kelly, mkelly@alteeve.com

Date:

August 28, 2008

Shameless Plug:

http://tle-bu.org

Table of Contents

1. Welcome!

First and foremost, thank you for taking the time you are taking and allowing me to introduce a (relatively) new resource available in the open source community.

D-Bus has been in development for years, but is now really starting to come into it's own as it rapidly matures and slips into the mainstream of open source software. I first became interested in D-Bus around 2004, but at the time it was still too young to be very useful to me. That said, I knew that one day it would become a critical part of my personal projects, and now that day has come.

I hope you find D-Bus to be as impressive and useful as I have. More, I hope to be able to bring it just this much further into the mainstream. I think it is one of the most important developments in the general open source community in many years.

2. So Then, What is D-Bus, Anyway?

D-Bus is, at it's most basic, a new way of handling inter-process communincation (IPC) and lifecycle tracking of other processes.

2.1 That's Nice, So Why Would I Use It?

D-Bus was designed to address a few issues. They are:

- D-Bus is designed to have very low latency.
- It can run synchronously or asynchronously.
- It uses a binary protocol and so doesn't need to translate between binary and text formats.
- It's easy to use because it uses 'messages' instead of byte streams.
- It's fully UTF-8 compliant.

2.2 Well, is it Perfect for Everything?

No, sorry.

2.2.1 Oh?

It is designed primarily for same-machine IPC functions. It's not designed for intermachine communication, though that is certainly possible to use D-Bus across multiple machines.

Though it's fast, it is not the fastest option with regard to raw throughput. Pushing data raw over a socket is faster (by 2.5 times or more, according to the D-Bus FAQ, question 7). It's focus on security, asynchronous communication, abstraction and such was decided to be more important than raw throughput ability.

3. Alright, Let's Get Started Then

D-Bus can be used in a few ways.

- You can use it to simply listen to signals or call methods already published on a message bus.
- You can publish new methods under a given interface on an existing D-Bus message bus.
- You can make your program act as a D-Bus daemon running it's very own message bus. You can then listen for connections from clients on a UNIX socket, a TCP socket and so on.

3.1 But What Do All Those Terms Mean?

For me at least, one of the biggest struggles I had when I was trying to learn about D-Bus was getting all the terms straightened out in my head. D-Bus is designed to be object oriented and until I set out to learn it, I had little experience with OO programming. To help others who may also be unfamiliar with OO concepts, I will approach this talk from the point of view of a non-OO programmer.

Let me begin with a few term comparisons that might help people, like me, who aren't traditionally fans of object oriented programming. Then I will talk about terms more specific to D-Bus.

- A '*method*' is like a function or subroutine. It takes zero or more arguments, does some work and returns zero or more arguments.
- A '*signal*' is simply a method that takes one input parameters (the message) and emits it onto the message bus. The D-Bus daemon worries about delivery and routing of the message to zero or more interested listeners.
- An '*Input Parameter*' is a declaration of what kind of data a method accepts as input arguments.
- An '*Output Parameter*' is a declaration of what kind of data is returned by a method.

Now, some more specific terms used by D-Bus.

- To '*publish*' a method is to make it available to callers or listeners on the message bus. When a method is published, the input and output parameters must be pre-defined as to what kind of data the method will accept or return. Each method or signal is registered with the D-Bus daemon under a given '*interface*' and available at a give '*object path*'.
- An '*interface*' is a collection of published methods and signals. It provides no function in and of itself, rather, it is simply a named collection. An interface will have one or more '*services*' under it.
- A '*service*' is a named collection of methods and signals with an associated '`.service`' file. D-Bus uses the service name to know what script or program to start when a given object is called.
- An '*object*' is like a module or library. It's a handle to a collection of methods and signals.
- A program uses a '*Name Space*' to publish methods and signals. The name space is usually the program's reversed domain name. When a given web domain, company or group provides several programs, the program name is often added to the mix. For example, I use the name space '`org.tle-bu`'. If I published two programs '`foo`' and '`bar`', their name spaces would be '*`org.tle-bu.foo`*' and '*`org.tle-bu.bar`*', respectively. These are general guides though, this is not strictly required.
- A common term used is '*Well Known Name*', which refers to nothing more than a name space that has been clearly published in the program's always excellent documentation. Say for example, my '`foo`' program published a method called '`doMath`', I might publish it at the object path with the well known name '`/org/tle-bu/foo/Math`'.
- The '*Bus*' is the virtual “wire” that messages are passed over. The message bus is managed by the D-Bus daemon. Think of the bus as the programmatic version of a network router.
- A D-Bus '*Binding*' is a programming language-specific library that handles the communication between the language's normal programming commands and the D-Bus API. For example, Perl's D-Bus binding is the '`Net::DBus`' module available from CPAN.

3.2 A Note on the Terms '*Server*' and '*Client*'

There is an important note about the terms '*Server*' and '*Client*' in D-Bus parlance.

So far as D-Bus is concerned, the concept of a '*Server*' and '*Client*' don't mean much of anything. The closest thing to a D-Bus '*Server*' would be a program that acts as a message bus daemon. Generally though, most programs use one of the '`Freedesktop.org`' message busses. This is because a given program can both publish methods and signal and access methods and signals published by other programs at the same time. In this way, any program can be both a server and a client in the traditional sense.

The closest thing to a “server” that you will find in D-Bus is a '*Service*'. A “service” is simple a program that can be called by the D-Bus daemon when a given interface is called by a third-party program. When and interface is called and it is not currently available, the daemon starts the associated program and waits for the interface to be registered by the program with the daemon, and then tells the calling program that the method under the interface is now ready. This said, once the service is started, it is no longer special in any real way.

4. Now That We Have a Common Vernacular...

A somewhat accurate analogue for D-Bus is that it's daemon works like a network router. It's responsible for making sure messages get passed properly between callers and methods, making sure signals reach interested listeners and enforcing security.

4.1 HAL and D-Bus, The First Example

Before we get into our own, somewhat artificial examples, let's look briefly at a real-world used of the D-Bus message bus and how it is used by HAL (Hardware Abstraction Layer) and the Gnome window manager. Specifically, let's look at how the HAL program provides near-real time information on hardware changes so that Gnome knows what to show to the user, like a CD icon when an optical disk is mounted.

The HAL program is responsible for monitoring hardware changes on a system. For example, when you plug in a USB storage device, insert or remove an optical disk or so forth, it detects these events. Beyond that though, it doesn't actually *do* anything. Instead, it uses D-Bus to inform interested programs about these changes.

When the HAL detects a change, it sends a signal to the D-Bus daemon which in turn emits it to any interested listeners. This “signal” is much like a network broadcast, nobody may care or be listening, but that is of no concern to HAL. If a program does cares, it will tell the D-Bus daemon that it is interested in HAL's signals. This is called subscribing to HAL's D-Bus signals. After this, when D-Bus emits a signal, the D-Bus daemon will route the signal to the interested program(s). How the listener reacts to these signals is entirely up to that program and is of no concern to HAL or D-Bus.

The listening program, in this case the Gnome window manager, will know about hardware changes from the point after it subscribes to the HAL signals.

What if it wants to ask HAL a question though? Gnome could go to the trouble of discovering information about existing hardware itself, but that would be duplicating the work HAL already has done. As all programmers know, laziness is one of a programmer's highest callings! So it makes much more sense for

Gnome to ask HAL about the underlying hardware.

Well, this is the second way that D-Bus allows for communication between clients and servers; published methods.

4.2 Talking to Other Programs Via Published Methods

When the HAL starts, it collects data about the hardware on a system. HAL then publishes a method called '`GetAllDevices`' which, when called, returns the list of hardware it found. So, when Gnome wants to know what hardware is on the system, it simply calls the HAL's '`GetAllDevices`' method on the message bus.

4.2.1 A Little More Detail

Before the HAL program can do anything on the D-Bus, it must first tell the D-Bus daemon what methods and signals it plans to publish. For each method or signal, it needs to specify what each method's input parameters are (if any) and what kind of data each of it's signals and methods return, if any. This is because the D-Bus message bus is a strongly typed system where all data types must be pre-declared before data can be passed or collected.

Using the '`GetAllDevices`' method as an example, the HAL must tell the D-Bus daemon that this method takes no arguments and returns an array of strings. Only then can other programs begin using this method.

4.3 Bindings

A program could, if the programmer wished, contact the D-Bus daemon directly using the libdbus C library. Of course, this is a lot of work and we've already covered the sin involved here. Thankfully, other people have already done the hard work by creating a '*Binding*', a glue layer between a programming language and the D-Bus C library.

The goal of a D-Bus binding is to make access to the D-Bus simple and in a format familiar to programmer's of the given language. In Perl, this is Daniel Belange's '`Net::DBus`' perl module, available on your local CPAN.

Now then, here is the promised first example; You can connect to the D-Bus '`system`' bus, call HAL's '`GetAllDevices`' method and gather the responces with just a few lines of code. Here is an example:

mk-lshal.pl
Download from: http://tle-bu.org/files/examples/mk-lshal.pl

```
#!/usr/bin/perl
# This is a slight variant on an example provided by Net::DBus.

use strict;
use warnings;

# Load the Net::DBus D-Bus binding.
use Net::DBus;

# Connect to the system bus.
my $bus = Net::DBus->system;

# Get a handle to the HAL service.
```

```
my $hal = $bus->get_service("org.freedesktop.Hal");

# Get a handle to the device manager.
my $manager = $hal->get_object("/org/freedesktop/Hal/Manager", "org.freedesktop.Hal.Manager");

# Get a reference to the array of hardware provided by HAL's 'GetAllDevices' method.
my $hal_list=$manager->GetAllDevices;

# List the devices now.
print "All the hardware that HAL knows about is below:\n";
foreach my $dev (@{$hal_list})
{
        print "Device: [$dev]\n";
}

exit(0);
```

To try to do the same this without the '`Net::DBus`' binding would be much more difficult and require far more code! If you are curious, take a look at the various modules provided by the '`Net::DBus`' module and you will see how much work goes in to making the magic happen!

5. Going a Little Deeper

For the rest of the talk, I would like to explore various ways of using D-Bus. Specially, I'd like to show working perl code actually doing each of the main D-Bus functions. We've already seen a simple example that connects to an existing D-Bus message bus, calls a published method and process the returned data. Now let's step it up a little.

5.1 Our first Service and Caller!

To get a simple example script working is a little tricky, as a few things need to be in place to get started. For that reason, this first example will be the most involved. All further sample code after this will be based on these scripts. Please be sure you understand then how these work before proceeding.

I will do my best to explain all parts in as much detail as I can. Also, I will do my best to keep the functionality of the code as simple as possible and to make the code as clear as possible. In the real world, there will be many steps that could be consolidated into “cleaner” code. Some people may find this a little basic, but please bear with me.

5.1.1 The Initial 'Service' Daemon

This is the '`service`' script that will run as a daemon. It will start, publish a method and a signal onto the '`session`' message bus and then wait for callers to call it.

I will start by publishing a simple method called '`doMath`' which will do simple math and return the results and, when needed, an error string.

I start the program in the normal way. Being a good little coder, I will enable '`strict`' and '`warnings`'.

```
1.  #!/usr/bin/perl
2.  use strict;
3.  use warnings;
```

Here I load the 'Net::DBus' core module. This provides all the basic binding functions.

```
4.  use Net::DBus;
```

This loads the 'Net::DBus::Service' module. It provides the functions needed to publish methods and signals onto the D-Bus message bus under our service name.

```
5.  use Net::DBus::Service;
```

This loads the 'Net::DBus::Reactor' event loop. This will keep this program running and will “react” to requests from callers.

```
6.  use Net::DBus::Reactor;
```

At this point, you can include more program functions and features. In our case though, I will get right to the D-Bus stuff. In the real world, you may wish to read in config files, print greetings or start logging or ... whatever.

From here on, this must be a 'package'. This is required as it will be used when I create our object further down. Also, this is often (though not need be) the same name used on the object path when the 'new' constructor method is called below.

A note on the position of this; You can define the package name further up, but it MUST be declared before the 'Net::DBus::Object' module is loaded.

```
7.  package DBusExampleService;
```

Now that I've defined our package name, I load the 'Net::DBus::Object' module. This provides the mechanism for preparing the actual methods and signals for use on the D-Bus. For info on the 'base' syntax, see: http://perldoc.perl.org/base.html.

```
8.  use base qw(Net::DBus::Object);
```

Now load the module that will handle exporting our prepared methods and signals onto the D-Bus.

Here, I decide to define a default interface that will be used when I don't explicitly define an interface when preparing a method or signal. This is an optional step, though. Without it, I would have to specify an interface at all times.

```
9.  use Net::DBus::Exporter qw(org.tle_bu.DBusExampleInterface);
```

Start the 'new' method which is our package's constructor.

```
10. sub new
11. {
12.         my $class=shift;
13.         my $service=shift;
```

Here, I create the service and make it available at the object path '/org/tle_bu/DBusExampleObject'. Usually this matches the name of the package, but I am keeping it different to make it a bit easier to see that these are different.

```
14.         my $self=$class->SUPER::new($service, "/org/tle_bu/DBusExampleObject");
15.         bless $self, $class;
16.         return $self;
17. }
```

This method will take two integers and a string and, depending on the string, does some math on the two integers. It then returns the resulting real number and a string.

```
18. sub doMath
19. {
20.         my ($self, $num1, $num2, $symbol)=@_;
21.         my $result=0;
22.         my $string="";
23.         print "The 'doMath' method was called in 'service_01.pl' asking what: [$num1 $symbol $num2] is.\n";
24.         if ( $symbol eq "+" )
25.         {
26.                 # Add the numbers.
27.                 $result=$num1+$num2;
28.         }
29.         elsif ( $symbol eq "-" )
30.         {
31.                 # Subtract the second number from the first.
32.                 $result=$num1-$num2;
33.         }
34.         elsif (( $symbol eq "/" ) || ( $symbol eq "÷" ))
35.         {
36.                 # Divide the first number by the second.
37.                 if ( $num2 == 0 )
38.                 {
39.                         $string="Divide by 0 error!";
40.                 }
41.                 else
42.                 {
43.                         $result=$num1/$num2;
44.                 }
45.         }
46.         elsif (( $symbol eq "*" ) || ( $symbol eq "x" ))
47.         {
48.                 # Multiple the numbers/
49.                 $result=$num1*$num2;
50.         }
51.         else
52.         {
53.                 $string="Unknown math symbol '$symbol'. Unable to process the numbers: '$num1' and '$num2'.
    Supported math commands are: '+', '-', '/', '÷', '*' and 'x'.\n";
54.         }
55.
56.         # Return the result.
57.         return ($result, $string);
58. }
```

This publishes the above method on the D-Bus. As mentioned, the D-Bus is strongly typed so I need to specify that this method will accept two '`int32`' signed integers and a '`string`' (the input parameters) and it will return a '`double`' IEEE-754 floating point number and a '`string`' (the output parameters).

```
59. dbus_method("doMath", ["int32", "int32", "string"], ["double", "string"]);
```

Now that I've got the methods and signals done for this package, I need to make them available on the D-Bus message bus. To do this, I will create a new package called '`main`', register a service with the '`session`' bus, publish the methods above under that service and then create a reactor loop to listen for calls to the exported methods above.

This creates a small package called '`main`' used to setup the exporting of any methods and signals created above.

```
60. package main;
```

Create a connection to the Freedesktop.org '`session`' bus, which is the bus accessible by the user invoking this module only.

```
61. my $bus=Net::DBus->session();
```

Export a service with the given service name. In this example, I will use the name '`org.tle-`

bu.org.SampleService'. Not that this does not need to relate to the Interface name or object path name, but it can if you like.

```
62. my $service=$bus->export_service("org.tle-bu.SampleService");
```

This exports the methods and signals made available in the 'DBusExampleService' package above onto the service I just created above.

```
63. my $object=DBusExampleService->new($service);
```

This starts the reactor loop for the 'main' package. This makes the service exported above, and the object registered with the service, available to callers. This loop will wait for calls until it catches a signal. For now, exit the loop by hitting '<ctrl>'+'c'.

```
64. Net::DBus::Reactor->main->run();
```

Finally, should the program ever run to the end, exit cleanly.

```
65. exit 0;
```

Done!

Here is the above script in a cut-and-paste friendly format.

service_01.pl
Download from: http://tle-bu.org/files/examples/service_01.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Net::DBus;
use Net::DBus::Service;
use Net::DBus::Reactor;

package DBusExampleService;
use base qw(Net::DBus::Object);
use Net::DBus::Exporter qw(org.tle_bu.DBusExampleInterface);

# This is the constructor mothod.
sub new
{
	my $class = shift;
	my $service = shift;
	my $self = $class->SUPER::new($service, "/org/tle_bu/DBusExampleObject");
	bless $self, $class;

	return $self;
}

# This method will take two integers and a string and, depending on the string,
# do some math on the two integers and return the resulting real number.
sub doMath
{
	my ($self, $num1, $num2, $symbol)=@_;
	my $result=0;
	my $string="";

	print "The 'doMath' method was called in 'service_01.pl' asking what: [$num1 $symbol $num2] is.\n";
	if ( $symbol eq "+" )
	{
		# Add the numbers.
		$result=$num1+$num2;
	}
	elsif ( $symbol eq "-" )
	{
```

```
		# Subtract the second number from the first.
		$result=$num1-$num2;
	}
	elsif (( $symbol eq "/" ) || ( $symbol eq "÷" ))
	{
		# Divide the first number by the second.
		if ( $num2 == 0 )
		{
			$string="Divide by 0 error!";
		}
		else
		{
			$result=$num1/$num2;
		}
	}
	elsif (( $symbol eq "*" ) || ( $symbol eq "x" ))
	{
		# Multiple the numbers/
		$result=$num1*$num2;
	}
	else
	{
		$string="Unknown math symbol '$symbol'. Unable to process the numbers: '$num1' and '$num2'.
Supported math commands are: '+', '-', '/', '÷', '*' and 'x'.\n";
	}

	# Return the result.
	return ($result, $string);
}
dbus_method("doMath", ["int32", "int32", "string"], ["double", "string"]);

package main;
my $bus=Net::DBus->session();
my $service=$bus->export_service("org.tle-bu.SampleService");
my $object=DBusExampleService->new($service);
Net::DBus::Reactor->main->run();

exit 0;
```

Done!

5.1.2 The Initial 'Caller' Program

It's all nice and dandy to have the service script above, but by itself it won't *do* anything. To actually make something happen, I need a program that calls the published '`doMath`' method. That is what this caller script will do.

As above, I start the program in the normal way.

```
1.  #!/usr/bin/perl
2.  use strict;
3.  use warnings;
4.  use Net::DBus;
5.  my $bus=Net::DBus->session();
```

Get a connection to the service with the well known name '`org.tle-bu.SampleService`'. I know to use this service name because the author of the service made it clearly known in their always-excellent documentation. In our case, this is the service name used in the '`main`' package in the '`service_01.pl`' file's '`export_service`' call.

```
6.  my $service=$bus->get_service("org.tle-bu.SampleService");
```

Now that I have a connection to the service, I need to connect to the methods exported under the object path '`/org/tle_bu/DBusExampleObject`' published under the interface '`org.tle_bu.DBusExampleInterface`'. Technically, I could leave off the interface decleration *IF* I can be sure the '`DBusExampleObject`' object is

uniquly named within this service *AND* the remote object supports introspection. However, it is always safest to be specific.

The object path is specified in the '`service_01.pl`' file's '`new`' method where '`$self`' was first invoked. The interface is specified in the same file when '`Net::DBus::Exporter`' was called. Alternatively, if a different interface was defined when the '`dbus_method`' call was made for the exported method I am interested in, I would use that here instead.

```
7.  my $object=$service->get_object("/org/tle_bu/DBusExampleObject", "org.tle_bu.DBusExampleInterface");
```

Now let's do some math using the '`doMath`' method!

```
8.  my $num1=10;
9.  my $num2=4;
10. my $symbol="+";
11. my ($result, $string)=$object->doMath($num1, $num2, $symbol);
12. if ($string)
13. {
14.         print "I asked what: [$num1 $symbol $num2] was, but I got an error: [$string]\n";
15. }
16. else
17. {
18.         print "I asked what: [$num1 $symbol $num2] is and I was told: [$result].\n";
19. }
20. exit 0;
```

Here is the above script in a cut-and-paste friendly format.

caller_01.pl
Download from: http://tle-bu.org/files/examples/caller_01.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Net::DBus;

my $bus=Net::DBus->session();
my $service=$bus->get_service("org.tle-bu.SampleService");
my $object=$service->get_object("/org/tle_bu/DBusExampleObject", "org.tle_bu.DBusExampleInterface");

# Let's do some math using the 'doMath' method.
my $num1=10;
my $num2=4;
my $symbol="+";
my ($result, $string)=$object->doMath($num1, $num2, $symbol);
if ($string)
{
        print "I asked what: [$num1 $symbol $num2] was, but I got an error: [$string]\n";
}
else
{
        print "I asked what: [$num1 $symbol $num2] is and I was told: [$result].\n";
}

exit 0;
```

That's it!

At this point you now have a fully functional D-Bus service running with an exported method and a caller program that calls it. All things considered, that wasn't too painful, eh?

To test this, start two terminals; On the first, start the '`service_01.pl`' script. It will not return but instead wait for connections. On the second terminal and start the '`caller_01.pl`' script. It will finish very quickly, showing the results of the math equation. If you go back to the first terminal when you started the

service program, you will see that that a message was printed saying that the 'doMath' method was called (line 23 in service_01.pl).

Terminal 1
$ ./service_01.pl _

Terminal 2
$ ./caller_01.pl I asked what: [10 + 4] is and I was told: [14]. $ _

Terminal 1 (after 'caller_01.pl' was called)
$ ./service_01.pl The 'doMath' method was called in 'service_01.pl' asking what: [10 + 4] is. _

You can change up the integers and the symbol in the 'caller_01.pl' file to play with what the program returns.

5.2 Adding a 'Listener' Script

There is one more key function of D-Bus that now needs to be explored; Signals.

The idea of signals is to allow a program to make annoucements that other programs may be interested in. A good example is how HAL announces the removal of hardware by emitting signals it calls 'DeviceRemoved' under it's interface 'org.freedesktop.Hal.Manager'. Interested programs subscribed to this signal will then be able to react to hardware being removed. If you pull the cable on a mounted drive, for example, Gnome will react by warning you that the drive was unsafely removed.

Before I write the listener script though, I need to add a signal to the service script and export it onto the D-Bus message bus. Then I need to modify my caller script to make use of this new signal. So let's start by making copies of the service and caller files called 'service_02.pl' and 'caller_02.pl' respectively. To keep things consistent, let's call our first listener 'listener_02.pl' to go with it.

5.2.1 The Updated Service Script

This creates a signal called 'signalMessage' that emits 'strings'. Note that this does *not* have a coresponding subroutine! Signals have no room for logic themselves. Think of them as glorified 'print's. In this case, we're telling D-Bus that this string will take (and then emit) a 'string' type message.

```
1.  dbus_signal("signalMessage", ["string"]);
```

This is a method that will call the 'signalMessage' signal and then return the message that it sent to the signal. This is *not* needed for a signal to work! This is here to simply wrap some functionality around the signal call. You will see in the listener how the signal can be called directly.

```
2.  sub emitMessage
3.  {
4.          # This is a method, so 'self' is also passed in first.
5.          my $self=shift;
6.          my $string=shift;
7.
```

```
8.        # This will print to the terminal that this service was started on that
9.        # this method has been called.
10.       print "Method 'emitMessage' called in 'service_02.pl'.\n";
11.
12.       # For clarity sake, I will put my string into a variable. I could just
13.       # pass this directly, too.
14.       $string="Hello, World!" unless defined $string;
15.
16.       # This calls the 'Net::DBus' 'emit_signal' method
17.       $self->emit_signal("signalMessage", "$string");
18.
19.       return "Emitted signal string: $string";
20. }
21. dbus_method("emitMessage", ["string"], ["string"]);
```

The new, complete '`signal_02.pl`' file is below in copy-and-paste a friendly format.

service_02.pl
Download from: http://tle-bu.org/files/examples/service_02.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Net::DBus;
use Net::DBus::Service;
use Net::DBus::Reactor;

package DBusExampleService;
use base qw(Net::DBus::Object);
use Net::DBus::Exporter qw(org.tle_bu.DBusExampleInterface);

# This is the constructor mothod.
sub new
{
	my $class = shift;
	my $service = shift;
	my $self = $class->SUPER::new($service, "/org/tle_bu/DBusExampleObject");
	bless $self, $class;

	return $self;
}

# This method will take two integers and a string and, depending on the string,
# do some math on the two integers and return the resulting real number.
sub doMath
{
	my ($self, $num1, $num2, $symbol)=@_;
	my $result=0;
	my $string="";

	print "The 'doMath' method was called in 'service_01.pl' asking what: [$num1 $symbol $num2] is.\n";
	if ( $symbol eq "+" )
	{
		# Add the numbers.
		$result=$num1+$num2;
	}
	elsif ( $symbol eq "-" )
	{
		# Subtract the second number from the first.
		$result=$num1-$num2;
	}
	elsif (( $symbol eq "/" ) || ( $symbol eq "÷" ))
	{
		# Divide the first number by the second.
		if ( $num2 == 0 )
		{
			$string="Divide by 0 error!";
		}
		else
		{
			$result=$num1/$num2;
```

```
		}
	}
	elsif (( $symbol eq "*" ) || ( $symbol eq "x" ))
	{
		# Multiple the numbers/
		$result=$num1*$num2;
	}
	else
	{
		$string="Unknown math symbol '$symbol'. Unable to process the numbers: '$num1' and '$num2'.
Supported math commands are: '+', '-', '/', '÷', '*' and 'x'.\n";
	}

	# Return the result.
	return ($result, $string);
}
dbus_method("doMath", ["int32", "int32", "string"], ["double", "string"]);

# This creates a signal called 'signalMessage' that emits 'strings'. Note that
# this does not have a coresponding subroutine!
dbus_signal("signalMessage", ["string"]);

# Now create the method called 'emitMessage".
sub emitMessage
{
	# This is a method, so 'self' is also passed in first.
	my $self=shift;
	my $string=shift;

	# This will print to the terminal that this service was started on that
	# this method has been called.
	print "Method 'emitMessage' called in 'service_02.pl'.\n";

	# For clarity sake, I will put my string into a variable. I could just
	# pass this directly, too.
	$string="Hello, World!" unless defined $string;

	# This calls the 'Net::DBus' 'emit_signal' method
	$self->emit_signal("signalMessage", "$string");

	return "Emitted signal string: $string";
}

# And now publish the 'emitMessage' method on the message bus.
dbus_method("emitMessage", ["string"], ["string"]);

package main;
my $bus=Net::DBus->session();
my $service=$bus->export_service("org.tle-bu.SampleService");
my $object=DBusExampleService->new($service);
Net::DBus::Reactor->main->run();

exit 0;
```

Done!

5.2.2 The Updated Caller Script

Next up, I will modify the caller script to call the '`signalMessage`' signal to indicate when the script starts and finishes. I was also emit a signal indirectly by calling the new '`emitMessage`' method, take the returned string and print it locally after the '`doMath`' method returns. The modifications are next.

This emits a signal indicating that the caller has started. Add this after the '`$object`' has been created.

```
1.  $object->emit_signal("signalMessage", "The 'caller_02.pl' script has started.");
```

This calls the '`emitMessage`' method which returns the message emitted on the signal, which I print. Then I send a final signal indicating that the caller is finished.

```
1.  my $message=$object->emitMessage("I can do math, me! I think '$num1 $symbol $num2 = $result $string'.");
2.  print "Called 'emitMessage' and got string: [$message]\n";
3.  $object->emit_signal("signalMessage", "The 'caller_02.pl' script is done.");
```

The new, complete '`caller_02.pl`' file is below in copy-and-paste a friendly format.

caller_02.pl
Download from: http://tle-bu.org/files/examples/caller_02.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Net::DBus;

my $bus=Net::DBus->session();
my $service=$bus->get_service("org.tle-bu.SampleService");
my $object=$service->get_object("/org/tle_bu/DBusExampleObject", "org.tle_bu.DBusExampleInterface");

$object->emit_signal("signalMessage", "The 'caller_02.pl' script has started.");

# Let's do some math using the 'doMath' method.
my $num1=10;
my $num2=8;
my $symbol="/";
my ($result, $string)=$object->doMath($num1, $num2, $symbol);
if ($string)
{
	print "I asked what: [$num1 $symbol $num2] was, but I got an error: [$string]\n";
}
else
{
	print "I asked what: [$num1 $symbol $num2] is and I was told: [$result].\n";
}

my $message=$object->emitMessage("I can do math, me! I think '$num1 $symbol $num2 = $result $string'.");
print "Called 'emitMessage' and got string: [$message]\n";

$object->emit_signal("signalMessage", "The 'caller_02.pl' script is done.");

exit 0;
```

Done!

At this point, you can run the new '`service_02.pl`' and then '`caller_02.pl`' and it will appear to run just as their '`01`' counterparts did. That's pretty boring though, because I won't see what all my hard work has added to my program!

5.2.3 Finally, The New Listener Script

The listener script runs as a daemon, like the service script. Once started, it waits for signals named '`signalMessage`' to arrive from the D-Bus message bus. Whenever it sees a signal, it calls an associated subroutine to process the contents of the signal. In this case, it merely prints the message locally.

Start things off in the usual way, connecting to the '`session`' bus, then to my service and finally to my object at the given path under the given interface.

```
1.  #!/usr/bin/perl
2.  use strict;
3.  use warnings;
```

```
4.  use Net::DBus;
5.  use Net::DBus::Reactor;
6.  my $bus=Net::DBus->session();
7.  my $service=$bus->get_service("org.tle-bu.SampleService");
8.  my $object=$service->get_object("/org/tle_bu/DBusExampleObject", "org.tle_bu.DBusExampleInterface");
```

This is where the listener takes shape. This is the function that I will call when a signal is received. The only data passed into this function is the contents of the signal itself.

```
9.  sub handleMessage
10. {
11.         # Pick up the message emitted by the signal.
12.         my $message=shift;
13.
14.         # Print the message to the terminal showing the message we received
15.         # from the signal.
16.         print "Received signal: [$message].\n";
17.
18.         return;
19. }
```

Here is where the magic happens. This tells the D-Bus daemon that I am interested in messages sent by the signal named '`signalMessage`' and tells the binding that when I receive a message, to call the referenced function called '`handleMessage`'.

```
20. $object->connect_to_signal("signalMessage", \&handleMessage);
```

This simply shows the user that the listener is now running.

```
21. print "Now listening to 'signalMessage' signals.\n";
```

Last, I create a new '`reactor`' object and start it's event loop. This will keep the script running until it is killed by a signal.

```
22. my $reactor=Net::DBus::Reactor->main();
23. $reactor->run();
24. exit 0;
```

Here is the completed '`listener_02.pl`' script in a cut-and-paste friendly format.

listener_02.pl
Download from: http://tle-bu.org/files/examples/listener_02.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Net::DBus;
use Net::DBus::Reactor;

my $bus=Net::DBus->session();
my $service=$bus->get_service("org.tle-bu.SampleService");
my $object=$service->get_object("/org/tle_bu/DBusExampleObject", "org.tle_bu.DBusExampleInterface");

# This will be a simple subroutine that will be called when the signal we care
# about is received. The message sent by the signal will be passed into this.
sub handleMessage
{
        # Pick up the message emitted by the signal.
        my $message=shift;

        # Print the message to the terminal showing the message we received
        # from the signal.
        print "Received signal: [$message].\n";

        return;
}
```

```
# I need to know the name of the signal being emitted. In this case, I know it
# is called "signalMessage" and is emitted under the object I created above.
$object->connect_to_signal("signalMessage", \&handleMessage);
print "Now listening to 'signalMessage' signals.\n";

# Now I need to create my event reactor which will keep the program alive and
# react to signals coming from the D-Bus message bus.
my $reactor=Net::DBus::Reactor->main();

# Now start the reactor loop. This will keep this program alive until it is
# killed by a signal.
$reactor->run();

# Exit cleanly when the reactor loop completes.
exit 0;
```

Done!

To test this, start three terminals; On the first and second terminals, start the '`service_02.pl`' and '`listener_02.pl`' scripts in that order. On the third terminal start the '`caller_02.pl`' script. It will finish very quickly, showing the results of the math equation. If you go back to the first terminal when you started the service program, you will see that that a message was printed saying that the '`doMath`' method was called. If you go to the second terminal you will see the three signals that were emitted by the caller.

Terminal 1

```
$ ./service_02.pl
_
```

Terminal 2

```
$ ./listener_02.pl
_
```

Terminal 3

```
$ ./caller_02.pl
I asked what: [10 / 8] is and I was told: [1.25].
Called 'emitMessage' and got string: [Emitted signal string: I can do math, me! I think '10 / 8 = 1.25 '.]
$ _
```

Terminal 1 (after 'caller_02.pl' was called)

```
$ ./service_01.pl
The 'doMath' method was called in 'service_02.pl' asking what: [10 / 8] is.
Method 'emitMessage' called in 'service_02.pl'.
_
```

Terminal 2 (after 'caller_02.pl' was called)

```
$ ./listener_02.pl
Now listening to 'signalMessage' signals.
Received signal: [The 'caller_02.pl' script has started.].
Received signal: [I can do math, me! I think '10 / 8 = 1.25 '.].
Received signal: [The 'caller_02.pl' script is done.].
$ _
```

Pretty cool, eh!

5.3 Autostart Services via '.service' Files

Up to now, I've started the '`service_0#.pl`' scripts manually. D-Bus has the ability to start services on an as-needed basis using "`.service`" files. These are very simple plain text, UTF-8 encoded files in the '`/usr/share/dbus-1/services`' directory that tell the D-Bus daemon what script or program to start when a given service or services are called.

Before I proceed, let me declare ***mea culpa***.

In D-Bus parclance, it is adviced that a '*service*' can only be called such when the program has an associated '`.service`' file so that D-Bus knows how to start it. Until now, this has not been the case.

5.3.1 The First '.service' File; 'tle-bu.org.service'

This little file tells the D-Bus daemon what program to start when a caller wants to call a method provided by my service. Here is the service file I would create for the '02' series of sample scripts created above:

/usr/share/dbus-1/services/tle-bu.org.example_02.service
Download from: http://tle-bu.org/files/examples/tle-bu.org.example_02.service

```
# The service file for TLE-BU example set '02'.
[D-BUS Service]
Name=org.tle-bu.SampleService
Exec=/home/digimer/projects/dbus_tutorial/examples/service_02.pl
```

Now, if you call '`listener_02.pl`' or '`caller_02.pl`', the D-Bus daemon will look for the service name '`org.tle-bu.SampleService` ' amoung all the files in the '`/usr/share/dbus-1/services/`' with the '`.service`' suffix. If it finds one, it will call the associated program defined in the '`Exec`' line.

A couple notes on service files;
- If your program supports multiple service names, you can specify them using the '`Names=org.tle-bu.SampleService;org.tle-bu.OtherService;`' syntax. Note that '`Names`' is plural and the different service names are semi-colon seperated.
- The name of the '`.service`' file doesn't need to match the service name in any way, but it must end in '`.service`'.
- The service file needs to be UTF-8 encoded.
- Only programs with a corresponding '`.service`' file can be properly called "Services".
- If you copy the file above, be sure to adjust the path to the service script to match your directory structure.

With this service file in place, try repeating the steps in '**5.2.3**' without first starting the '`service_02.pl`' script. Once you call either the listener or the caller, check your running processes. You should now see the '`service_02.pl`' script running automatically.

5.4 Further Reading

There are many more ways that D-Bus can be used that are not covered here. Here is an incomplete list of other capabilities of the D-Bus:

1. Calls to the D-Bus can be run asyncronously. For example, you can call a method and get back to work. When the reply comes, your program can be alerted and a registered function or subroutine

will be called to handle the reply. When a call is made to a method, a serial number is returned. When the reply finally comes, or an error or timeout relating to that call returns, it will bear the same serial number, allowing your program to easily track what replies belong to what calls.
2. A service can run independant of the Freedesktop.org daemon. Connections can be made via socket files, TCP sockets and memory addresses.

6. D-Bus Elements and Tidbits

Here I would like to cover some of the internal elements of D-Bus that you may come across. I would like to cover a few more essoteric topics, like licensing. A lot of this section is based, in some cases nearly verbatim, from the D-Bus FAQ.

6.1 The Legal Bits

As always, you should always consult your own lawyers for legal advice.

D-Bus is licenses under two licenses, the GPL and the AFL. The former requires that your program be licensed under the GPL as well. The latter is more of an 'X' or 'BSD' style license which allows D-Bus to be used in closed source programs. However, the main condition of the AFL license is that, should you decide to sue Freedesktop.org for some infringement relating to D-Bus, you must first stop using D-Bus.

For more specifics, please look for the '`COPYING`' file in the D-Bus source files.

6.2 A Call For Help

The original draft of this document was more ambitious. The future of this document is now far more ambitious than when I began.

In the original draft, I had planned to show how to create a dedicated daemon that would listen for callers on UNIX socket files and TCP sockets. I have also been working on a full reference of all aspects of the '`Net::DBus`' binding and all the methods it provides with working examples of all the ways D-Bus could be used in Perl.

It wasn't to be, though.

D-Bus has reached “critical mass” by any measurable standards, but it's future growth is hindered by the lack of accessible documentation and tutorials. What is available is somewhat incomplete, at best, and is aimed primarily at people who already have a good understanding of D-Bus and are looking on the nitty-gritty details of how to implement it.

I had original sought help from developers in writting this tutorial and reference website, and was offered the help, but it has yet to materialize. I believe this is simply because the developers are doing what they do best; making great programs, and have trouble setting aside the time to assist with the tedious task of documenting their work.

So now I turn to you to help me continue to grow this tutorial. If you as excited about the possibilities of D-Bus as I am, please let me know. I am happy to conitnue spear-heading the tedious “pen to paper” work, though I'd love help there, too! What I really need is help at some of the deeper technical aspects. I also need help in debugging and expanding my sample programs provided here.

When the Perl version is done, I would then like help in porting this tutorial and the web reference to other languages as well.

If you would like to help, please contact me through my project website: http://tle-bu.org

6.3 D-Bus Glossay

Binding

- A D-Bus binding is a set of libraries, modules or other programming language specific tools that make interaction with the D-Bus libdbus C library simpler. Generally the goal of a D-Bus binding is to make the various features of the D-Bus usable in a manner familiar to the given programming language.

Bus

- A D-Bus “bus” is the virtual wire that messages travel along. If you visualize a network router, the message bus would be the links between various network devices with the D-Bus daemon being the “brains”, routing traffic on the bus and enforcing security policy.

Bus Name

- A bus name is like a machine's host name. It identifies the location of your application within D-Bus.
- For example, '`org.freedesktop.TextEditor`' could be the bus name for a text editor released by the same folks who wrote D-Bus. Alternatively, '`org.tle-bu`' is the bus name for my personal, never quite finished backup program. Notice that in the first case, the bus name contains the name of the specific program as there are several Freedesktop.org program where TLE-BU will only have one program so it uses the root name of it's domain name.
- Bus Names are actually references to 'Unique Names'. This relationship can be compared to Internet domain names where the name of the domain (bus name) references a unique IP address (unique name).
- Bus naming conventions and restrictions:
  - Bus names must never exceed 255 characters.
  - Valid interface names must contain at least two elements.
  - Elements are separated by a period ('`.`'), no double-period.
  - Periods must not be at the beginning or end of the interface name.
  - Interface elements must be UTF-8 encoded.
  - Elements must only use the ASCII alphanumeric or underscore characters ('`[a-z][A-Z][0-9][_]`').
  - Hyphens are *not* allowed.
  - '`org.freedesktop.DBus.Local`' is reserved.

Daemon

- A “daemon” is simply a normal program that stays alive until explicitely shut down or killed. These programs generally provide certain functions that could be needed from time to time. In D-Bus, the process that manages connections, message passing and security on the D-Bus message bus runs as a daemon.

Interface

- An interface is a named collection of objects.
- Interface naming conventions and restriction:

- Interface names must never exceed 255 characters.
- Valid interface names must contain at least two elements.
- Elements are separated by a period ('.'), no double-period.
- Periods must not be at the beginning or end of the interface name.
- Interface elements must be UTF-8 encoded.
- Elements must only use the ASCII alphanumeric or underscore characters ('[a-z][A-Z][0-9][_]').
- Hyphens are *not* allowed.
- 'org.freedesktop.DBus.Local' is reserved.

IPC (Inter Process Communication)

- As it's name suggests, “IPC” simply refers to any system that allows communication between two seperate processes. These processes could be between a child process spawned by a parent process or between two entirely seperate processes.

Marshalling

- The term “marshalling” comes from the process of putting an electrical signal onto a physical wire. Conversely, reading the signal off of a wire is called “unmarshalling”. In D-Bus terms, “marshalling” is used to refer to the act of putting a message onto the D-Bus message bus (virtual wire).

Member Name

- A member name is simple the name of a method or signal published on the D-Bus message bus.
- Member Name conventions and restriction:
  - Must only contain the ASCII characters "[A-Z][a-z][0-9]_" and may not begin with a digit.
  - Must not contain the '.' (period) character.
  - Must not exceed the maximum name length of 255 characters.
  - Must be at least 1 byte in length.

Object

- An object is a container holding pointers to methods, signals and values. An interface may have several objects, and each object could have several methods, signals and values.

Object Path

- An object path is the pointer to a given “object”; a pointer to a given element. This path could point to a method, a value or other elements of your program. For example, an object called 'Manager' published by HAL under the well known name 'org.freedesktop.Hal' is accessible as a method at the object path '/org/freedesktop/Hal/Manager'.
- Object Path convention and restriction:
  - The path may be of any length.
  - The path must begin with an ASCII forward slash ('/') character, and must consist of elements separated by forward slash characters.
  - Each element must only contain the ASCII characters "[A-Z][a-z][0-9]_"
  - No element may be the empty string.
  - Multiple '/' characters cannot occur in sequence.
  - A trailing '/' character is not allowed.

Service

- The term “service” is often used in reference to bus names. The D-Bus specification urges avoiding the use of “service” for any reason other than refering to a program that the D-Bus

server knows how to start.

Service (Program)

- A service is a program that can be started by the D-Bus server on an as-needed basis to provide some functions.

Unique Name

- A unique name is a name automatically assigned to a program when it first joins the D-Bus and is generally associated with a 'Bus Name'. This is how the D-Bus daemon finds and communicates with the program throughout it's life. Likewise, this unique name is the last thing released by a program when it disconnects from the D-Bus.
- The unique name is never re-used. If the same program later reconnects to the D-Bus, even if under the same 'Bus Name', it will get a new unique name. If the parent server is restarted, only then could a bus name potentially be used again.
- Multiple Bus Names can be associated with a given unique name in a way similar to how multiple domain names can point to a single IP address.
- Unique Name conventions and restrictions:
  - Unique connection names must begin with the character ':' (ASCII colon character);
  - Same further restrictions as Bus Names.

References:

- Freedesktop.org Website: http://www.freedesktop.org
- Official D-Bus FAQ: http://D-Bus.freedesktop.org/doc/D-Bus-faq.html
- Official D-Bus Specifications: http://D-Bus.freedesktop.org/doc/D-Bus-specification.html
- TLE-BU D-Bus Tutorial: http://wiki.tle-bu.org/index.php/Net::D-Bus_Binding_Tutorial

Addendum

While writting this, I was listening to an interview with Oliver Schroer who was about to give his “Last tour on this planet”. His calmness in the face of his own end and his dedication to his art, music, is nothing short of inspirational.

http://www.oliverschroer.com – His “train came in” July 3rd, 2008. RIP.