NAME

Net::DBus::Exporter − Export object methods and signals to the bus

SYNOPSIS

```
# Define a new package for the object we're going
# to export
package Demo::HelloWorld;

# Specify the main interface provided by our object
use Net::DBus::Exporter qw(org.example.demo.Greeter);

# We're going to be a DBus object
use base qw(Net::DBus::Object);

# Ensure only explicitly exported methods can be invoked
dbus_strict_exports;

# Export a 'Greeting' signal taking a stringl string parameter
dbus_signal("Greeting", ["string"]);

# Export 'Hello' as a method accepting a single string
# parameter, and returning a single string value
dbus_method("Hello", ["string"], ["string"]);

# Export 'Goodbye' as a method accepting a single string
# parameter, and returning a single string, but put it
# in the 'org.exaple.demo.Farewell' interface
dbus_method("Goodbye", ["string"], ["string"], "org.example.demo.Farewell");
```

DESCRIPTION

The `Net::DBus::Exporter` module is used to export methods and signals defined in an object to the message bus. Since Perl is a loosely typed language it is not possible to automatically determine correct type information for methods to be exported. Thus when sub-classing Net::DBus::Object, this package will provide the type information for methods and signals.

When importing this package, an optional argument can be supplied to specify the default interface name to associate with methods and signals, for which an explicit interface is not specified. Thus in the common case of objects only providing a single interface, this removes the need to repeat the interface name against each method exported.

SCALAR TYPES

When specifying scalar data types for parameters and return values, the following string constants must be used to denote the data type. When values corresponding to these types are (un)marshalled they are represented as the Perl SCALAR data type (see perldata).

‘‘string’’
    A UTF−8 string of characters

‘‘int16’’
    A 16−bit signed integer

‘‘uint16’’
    A 16−bit unsigned integer

‘‘int32’’
    A 32−bit signed integer

‘‘uint32’’
    A 32−bit unsigned integer

“int64”
: A 64−bit signed integer. NB, this type is not supported by many builds of Perl on 32−bit platforms, so if used, your data is liable to be truncated at 32−bits.

“uint64”
: A 64−bit unsigned integer. NB, this type is not supported by many builds of Perl on 32−bit platforms, so if used, your data is liable to be truncated at 32−bits.

“byte”
: A single 8−bit byte

“bool”
: A boolean value

“double”
: An IEEE double-precision floating point

COMPOUND TYPES

When specifying compound data types for parameters and return values, an array reference must be used, with the first element being the name of the compound type.

[“array”, ARRAY−TYPE]
: An array of values, whose type os `ARRAY−TYPE`. The `ARRAY−TYPE` can be either a scalar type name, or a nested compound type. When values corresponding to the array type are (un)marshalled, they are represented as the Perl ARRAY data type (see perldata). If, for example, a method was declared to have a single parameter with the type, [“array”, “string”], then when calling the method one would provide a array reference of strings:

```
    $object−>hello(["John", "Doe"])
```

[“dict”, KEY-TYPE, VALUE−TYPE]
: A dictionary of values, more commonly known as a hash table. The `KEY−TYPE` is the name of the scalar data type used for the dictionary keys. The `VALUE−TYPE` is the name of the scalar, or compound data type used for the dictionary values. When values corresponding to the dict type are (un)marshalled, they are represented as the Perl HASH data type (see perldata). If, for example, a method was declared to have a single parameter with the type [“dict”, “string”, “string”], then when calling the method one would provide a hash reference of strings,

```
   $object−>hello({forename => "John", surname => "Doe"});
```

[“struct”, VALUE−TYPE−1, VALUE−TYPE−2]
: A structure of values, best thought of as a variation on the array type where the elements can vary. Many languages have an explicit name associated with each value, but since Perl does not have a native representation of structures, they are represented by the LIST data type. If, for exaple, a method was declared to have a single parameter with the type [“struct”, “string”, “string”], corresponding to the C structure

```
    struct {
      char *forename;
      char *surname;
    } name;
```

then, when calling the method one would provide an array reference with the values orded to match the structure

```
   $object−>hello(["John", "Doe"]);
```

MAGIC TYPES

When specifying introspection data for an exported service, there are a couple of so called `magic` types. Parameters declared as magic types are not visible to clients, but instead their values are provided automatically by the server side bindings. One use of magic types is to get an extra parameter passed with the unique name of the caller invoking the method.

''caller''
: The value passed in is the unique name of the caller of the method. Unique names are strings automatically assigned to client connections by the bus daemon, for example ':1.15'

''serial''
: The value passed in is an integer within the scope of a caller, which increments on every method call.

ANNOTATIONS

When exporting methods, signals & properties, in addition to the core data typing information, a number of metadata annotations are possible. These are specified by passing a hash reference with the desired keys as the last parameter when defining the export. The following annotations are currently supported

no_return
: Indicate that this method does not return any value, and thus no reply message should be sent over the wire, likewise informing the clients not to expect / wait for a reply message

deprecated
: Indicate that use of this method/signal/property is discouraged, and it may disappear altogether in a future release. Clients will typically print out a warning message when a deprecated method/signal/property is used.

param_names
: An array of strings specifying names for the input parameters of the method or signal. If omitted, no names will be assigned.

return_names
: An array of strings specifying names for the return parameters of the method. If omitted, no names will be assigned.

strict_exceptions
: Exceptions thrown by this method which are not of type Net::DBus::Error will not be caught and converted to D−Bus errors. They will be rethrown and continue up the stack until something else catches them (or the process dies).

METHODS

dbus_method($name, `$params`, `$returns`, [\%annotations]);
dbus_method($name, `$params`, `$returns`, `$interface`, [\%annotations]);
: Exports a method called `$name`, having parameters whose types are defined by `$params`, and returning values whose types are defined by `$returns`. If the `$interface` parameter is provided, then the method is associated with that interface, otherwise the default interface for the calling package is used. The value for the `$params` parameter should be an array reference with each element defining the data type of a parameter to the method. Likewise, the `$returns` parameter should be an array reference with each element defining the data type of a return value. If it not possible to export a method which accepts a variable number of parameters, or returns a variable number of values.

dbus_no_strict_exports();
: If a object is using the Exporter to generate DBus introspection data, the default behaviour is to only allow invocation of methods which have been explicitly exported.

: To allow clients to access methods which have not been explicitly exported, call `dbus_no_strict_exports`. NB, doing this may be a security risk if you have methods considered to be ''private'' for internal use only. As such this method should not normally be used. It is here only to allow switching export behaviour to match earlier releases.

dbus_property($name, `$type`, `$access`, [\%attributes]);
dbus_property($name, `$type`, `$access`, `$interface`, [\%attributes]);
: Exports a property called `$name`, whose data type is `$type`. If the`$interface` parameter is provided, then the property is associated with that interface, otherwise the default interface for the calling package is used.

dbus_signal($name, `$params`, [\%attributes]);
dbus_signal($name, `$params`, `$interface`, [\%attributes]);

Exports a signal called `$name`, having parameters whose types are defined by `$params`. If the `$interface` parameter is provided, then the signal is associated with that interface, otherwise the default interface for the calling package is used. The value for the `$params` parameter should be an array reference with each element defining the data type of a parameter to the signal. Signals do not have return values. It not possible to export a signal which has a variable number of parameters.

EXAMPLES

No parameters, no return values

A method which simply prints ''Hello World'' each time its called

```
sub Hello {
    my $self = shift;
    print "Hello World\n";
}

dbus_method("Hello", [], []);
```

One string parameter, returning an boolean value

A method which accepts a process name, issues the killall command on it, and returns a boolean value to indicate whether it was successful.

```
sub KillAll {
    my $self = shift;
    my $processname = shift;
    my $ret  = system("killall $processname");
    return $ret == 0 ? 1 : 0;
}

dbus_method("KillAll", ["string"], ["bool"]);
```

One list of strings parameter, returning a dictionary

A method which accepts a list of files names, stats them, and returns a dictionary containing the last modification times.

```
 sub LastModified {
    my $self = shift;
    my $files = shift;

    my %mods;
    foreach my $file (@{$files}) {
       $mods{$file} = (stat $file)[9];
    }
    return \%mods;
 }

 dbus_method("LastModified", ["array", "string"], ["dict", "string", "int32
```

Annotating methods with metdata

A method which is targeted for removal, and also does not return any value

```
 sub PlayMP3 {
     my $self = shift;
     my $track = shift;

     system "mpg123 $track &";
 }
```

```
    dbus_method("PlayMP3", ["string"], [], { deprecated => 1, no_return => 1 }
```

Or giving names to input parameters:

```
    sub PlayMP3 {
        my $self = shift;
        my $track = shift;

        system "mpg123 $track &";
    }

    dbus_method("PlayMP3", ["string"], [], { param_names => ["track"] });
```

AUTHOR

Daniel P. Berrange <dan@berrange.com>

COPYRIGHT

SEE ALSO

Net::DBus::Object, Net::DBus::Binding::Introspector