```
package dbus
import (
	"bytes"
	"reflect"
	"strings"
	"sync"
)
func newIntrospectIntf(h *defaultHandler) *exportedIntf {
	methods := make(map[string]Method)
	methods["Introspect"] = exportedMethod{
		reflect.ValueOf(func(msg Message) (string, *Error)
{
			path := msg.Headers[FieldPath].value.
(ObjectPath)
			return h.introspectPath(path), nil
}), }
	return newExportedIntf(methods, true)
}
//NewDefaultHandler returns an instance of the default
//call handler. This is useful if you want to implement only
//one of the two handlers but not both.
//
// Deprecated: this is the default value, don't use it, it will
be unexported.
func NewDefaultHandler() *defaultHandler {
	h := &defaultHandler{
		objects:     make(map[ObjectPath]*exportedObj),
		defaultIntf: make(map[string]*exportedIntf),
}
	h.defaultIntf["org.freedesktop.DBus.Introspectable"] =
newIntrospectIntf(h)
return h }
type defaultHandler struct {
	sync.RWMutex
	objects     map[ObjectPath]*exportedObj
	defaultIntf map[string]*exportedIntf
}
func (h *defaultHandler) PathExists(path ObjectPath) bool {
	_, ok := h.objects[path]
	return ok
}
func (h *defaultHandler) introspectPath(path ObjectPath) string
{
	subpath := make(map[string]struct{})
	var xml bytes.Buffer
	xml.WriteString("<node>")
```

```
      for obj := range h.objects {
            p := string(path)
            if p != "/" {
                  p += "/"
            }
            if strings.HasPrefix(string(obj), p) {
                  node_name :=
strings.Split(string(obj[len(p):]), "/")[0]
                  subpath[node_name] = struct{}{}
            }
      }
      for s := range subpath {
            xml.WriteString("\n\t<node name=\"" + s + "\"/>")
      }
      xml.WriteString("\n</node>")

      return xml.String()
}
func (h *defaultHandler) LookupObject(path ObjectPath)
(ServerObject, bool) {
      h.RLock()
      defer h.RUnlock()
      object, ok := h.objects[path]
      if ok {
            return object, ok
      }
      // If an object wasn't found for this exact path,
      // look for a matching subtree registration
      subtreeObject := newExportedObject()
      path = path[:strings.LastIndex(string(path), "/")]
      for len(path) > 0 {
subtree
object, ok = h.objects[path]
if ok {
      for name, iface := range object.interfaces {
            // Only include this handler if it registered for
the
            if iface.isFallbackInterface() {
                  subtreeObject.interfaces[name] = iface
} }
break }
            path = path[:strings.LastIndex(string(path), "/")]
      }
      for name, intf := range h.defaultIntf {
            if _, exists := subtreeObject.interfaces[name];
exists {
```

```
continue }
            subtreeObject.interfaces[name] = intf
      }
      return subtreeObject, true
}
func (h *defaultHandler) AddObject(path ObjectPath, object
*exportedObj) {
      h.Lock()
      h.objects[path] = object
h.Unlock() }
func (h *defaultHandler) DeleteObject(path ObjectPath) {
      h.Lock()
      delete(h.objects, path)
h.Unlock() }
type exportedMethod struct {
      reflect.Value
}
func (m exportedMethod) Call(args ...interface{})
([]interface{}, error) {
      t := m.Type()
      params := make([]reflect.Value, len(args))
      for i := 0; i < len(args); i++ {
            params[i] = reflect.ValueOf(args[i]).Elem()
      }

      ret := m.Value.Call(params)
      var err error
      nilErr := false // The reflection will find almost-nils,
let's only pass
back clean ones!
      if t.NumOut() > 0 {
*Error
if e, ok := ret[t.NumOut()-1].Interface().(*Error); ok { //
godbus
      nilErr = ret[t.NumOut()-1].IsNil()
      ret = ret[:t.NumOut()-1]
      err = e
} else if ret[t.NumOut()-1].Type().Implements(errType) { // Go
error
      i := ret[t.NumOut()-1].Interface()
      if i == nil {
            nilErr = ret[t.NumOut()-1].IsNil()
      } else {
            err = i.(error)
      }
      ret = ret[:t.NumOut()-1]
```

```
}
	}
	out := make([]interface{}, len(ret))
	for i, val := range ret {
		out[i] = val.Interface()
	}
	if nilErr || err == nil {
		//concrete type to interface nil is a special case
		return out, nil
}
	return out, err
}
func (m exportedMethod) NumArguments() int {
	return m.Value.Type().NumIn()
}
func (m exportedMethod) ArgumentValue(i int) interface{} {
	return reflect.Zero(m.Type().In(i)).Interface()
}
func (m exportedMethod) NumReturns() int {
	return m.Value.Type().NumOut()
}
func (m exportedMethod) ReturnValue(i int) interface{} {
	return reflect.Zero(m.Type().Out(i)).Interface()
}
func newExportedObject() *exportedObj {
	return &exportedObj{
		interfaces: make(map[string]*exportedIntf),
	}
}
type exportedObj struct {
	mu         sync.RWMutex
	interfaces map[string]*exportedIntf
}
func (obj *exportedObj) LookupInterface(name string)
(Interface, bool) {
	if name == "" {
		return obj, true
	}
	obj.mu.RLock()

	defer obj.mu.RUnlock()
	intf, exists := obj.interfaces[name]
	return intf, exists
}
func (obj *exportedObj) AddInterface(name string, iface
*exportedIntf) {
```

```
	obj.mu.Lock()
	defer obj.mu.Unlock()
	obj.interfaces[name] = iface
}
func (obj *exportedObj) DeleteInterface(name string) {
	obj.mu.Lock()
	defer obj.mu.Unlock()
	delete(obj.interfaces, name)
}
func (obj *exportedObj) LookupMethod(name string) (Method,
bool) {
	obj.mu.RLock()
	defer obj.mu.RUnlock()
	for _, intf := range obj.interfaces {
		method, exists := intf.LookupMethod(name)
		if exists {
			return method, exists
		}
}
	return nil, false
}
func (obj *exportedObj) isFallbackInterface() bool {
	return false
}
func newExportedIntf(methods map[string]Method, includeSubtree
bool)
*exportedIntf {
	return &exportedIntf{
		methods:        methods,
		includeSubtree: includeSubtree,
} }
type exportedIntf struct {
	methods map[string]Method
	// Whether or not this export is for the entire subtree
	includeSubtree bool
}
func (obj *exportedIntf) LookupMethod(name string) (Method,
bool) {
	out, exists := obj.methods[name]
	return out, exists
}
func (obj *exportedIntf) isFallbackInterface() bool {
	return obj.includeSubtree
}
//NewDefaultSignalHandler returns an instance of the default
//signal handler. This is useful if you want to implement only
```

```
//one of the two handlers but not both.
//
// Deprecated: this is the default value, don't use it, it will be unexported.
func NewDefaultSignalHandler() *defaultSignalHandler {
return &defaultSignalHandler{}

}
type defaultSignalHandler struct {
      mu      sync.RWMutex
closed bool
      signals []*signalChannelData
}
func (sh *defaultSignalHandler) DeliverSignal(intf, name string, signal *Signal)
{
      sh.mu.RLock()
      defer sh.mu.RUnlock()
      if sh.closed {
return }
      for _, scd := range sh.signals {
            scd.deliver(signal)
} }
func (sh *defaultSignalHandler) Terminate() {
      sh.mu.Lock()
      defer sh.mu.Unlock()
      if sh.closed {
return }
      for _, scd := range sh.signals {
            scd.close()
            close(scd.ch)
      }
      sh.closed = true
      sh.signals = nil
}
func (sh *defaultSignalHandler) AddSignal(ch chan<- *Signal) {
      sh.mu.Lock()
      defer sh.mu.Unlock()
      if sh.closed {
return }
      sh.signals = append(sh.signals, &signalChannelData{
            ch:   ch,
            done: make(chan struct{}),
      })
}
func (sh *defaultSignalHandler) RemoveSignal(ch chan<- *Signal)
```

```
{
	sh.mu.Lock()
	defer sh.mu.Unlock()
	if sh.closed {
return }
	for i := len(sh.signals) - 1; i >= 0; i-- {
		if ch == sh.signals[i].ch {
			sh.signals[i].close()
			copy(sh.signals[i:], sh.signals[i+1:])
			sh.signals[len(sh.signals)-1] = nil
			sh.signals = sh.signals[:len(sh.signals)-1]
} }
}
type signalChannelData struct {

	wg   sync.WaitGroup
	ch   chan<- *Signal
	done chan struct{}
}
func (scd *signalChannelData) deliver(signal *Signal) {
	select {
	case scd.ch <- signal:
	case <-scd.done:
		return
	default:
		scd.wg.Add(1)
		go scd.deferredDeliver(signal)
	}
}
func (scd *signalChannelData) deferredDeliver(signal *Signal) {
	select {
	case scd.ch <- signal:
	case <-scd.done:
	}
	scd.wg.Done()
}
func (scd *signalChannelData) close() {
	close(scd.done)
	scd.wg.Wait() // wait until all spawned goroutines return
}
```