

```

1 package dbus
2 import (
3     "bytes"
4     "reflect"
5     "strings"
6     "sync"
7 )
8 func newIntrospectIntf(h *defaultHandler) *exportedIntf {
9     methods := make(map[string]Method)
10    methods["Introspect"] = exportedMethod{
11        reflect.ValueOf(func(msg Message) (string, *Error)
12    {
13        path := msg.Headers[FieldPath].value.
14    (ObjectPath)
15        return h.introspectPath(path), nil
16    })), }
17    return newExportedIntf(methods, true)
18 }
19 //NewDefaultHandler returns an instance of the default
20 //call handler. This is useful if you want to implement only
21 //one of the two handlers but not both.
22 //
23 // Deprecated: this is the default value, don't use it, it will
24 // be unexported.
25 func NewDefaultHandler() *defaultHandler {
26     h := &defaultHandler{
27         objects:      make(map[ObjectPath]*exportedObj),
28         defaultIntf: make(map[string]*exportedIntf),
29     }
30     h.defaultIntf["org.freedesktop.DBus.Introspectable"] =
31     newIntrospectIntf(h)
32     return h }
33 type defaultHandler struct {
34     sync.RWMutex
35     objects      map[ObjectPath]*exportedObj
36     defaultIntf map[string]*exportedIntf
37 }
38 func (h *defaultHandler) PathExists(path ObjectPath) bool {
39     _, ok := h.objects[path]
40     return ok
41 }
42 func (h *defaultHandler) introspectPath(path ObjectPath) string
43 {
44     subpath := make(map[string]struct{})
45     var xml bytes.Buffer
46     xml.WriteString("<node>")

```

```

43     for obj := range h.objects {
44         p := string(path)
45         if p != "/" {
46             p += "/"
47         }
48         if strings.HasPrefix(string(obj), p) {
49             node_name :=
strings.Split(string(obj[len(p):]), "/")[0]
50             subpath[node_name] = struct{}{}
51         }
52     }
53     for s := range subpath {
54         xml.WriteString("\n\t<node name=\"" + s + "\"/>")
55     }
56     xml.WriteString("\n</node>")
57
58     return xml.String()
59 }
60 func (h *defaultHandler) LookupObject(path ObjectPath)
(ServerObject, bool) {
61     h.RLock()
62     defer h.RUnlock()
63     object, ok := h.objects[path]
64     if ok {
65         return object, ok
66     }
67     // If an object wasn't found for this exact path,
68     // look for a matching subtree registration
69     subtreeObject := newExportedObject()
70     path = path[:strings.LastIndex(string(path), "/")]
71     for len(path) > 0 {
72         subtree
73         object, ok = h.objects[path]
74         if ok {
75             for name, iface := range object.interfaces {
76                 // Only include this handler if it registered for
the
77                 if iface.isFallbackInterface() {
78                     subtreeObject.interfaces[name] = iface
79                 }
80             }
81             path = path[:strings.LastIndex(string(path), "/")]
82         }
83         for name, intf := range h.defaultIntf {
84             if _, exists := subtreeObject.interfaces[name];
exists {

```

```

85  continue }
86      subtreeObject.interfaces[name] = intf
87  }
88  return subtreeObject, true
89  }
90  func (h *defaultHandler) AddObject(path ObjectPath, object
    *exportedObj) {
91      h.Lock()
92      h.objects[path] = object
93  h.Unlock() }
94  func (h *defaultHandler) DeleteObject(path ObjectPath) {
95      h.Lock()
96      delete(h.objects, path)
97  h.Unlock() }
98  type exportedMethod struct {
99      reflect.Value
100 }
101 func (m exportedMethod) Call(args ...interface{})
    ([]interface{}, error) {
102     t := m.Type()
103     params := make([]reflect.Value, len(args))
104     for i := 0; i < len(args); i++ {
105         params[i] = reflect.ValueOf(args[i]).Elem()
106     }
107
108     ret := m.Value.Call(params)
109     var err error
110     nilErr := false // The reflection will find almost-nils,
    let's only pass
111 back clean ones!
112     if t.NumOut() > 0 {
113 *Error
114 if e, ok := ret[t.NumOut()-1].Interface().(*Error); ok { //
    godbus
115     nilErr = ret[t.NumOut()-1].IsNil()
116     ret = ret[:t.NumOut()-1]
117     err = e
118 } else if ret[t.NumOut()-1].Type().Implements(errType) { // Go
    error
119     i := ret[t.NumOut()-1].Interface()
120     if i == nil {
121         nilErr = ret[t.NumOut()-1].IsNil()
122     } else {
123         err = i.(error)
124     }
125     ret = ret[:t.NumOut()-1]

```

```

126 }
127     }
128     out := make([]interface{}, len(ret))
129     for i, val := range ret {
130         out[i] = val.Interface()
131     }
132     if nilErr || err == nil {
133         //concrete type to interface nil is a special case
134         return out, nil
135     }
136     return out, err
137 }
138 func (m exportedMethod) NumArguments() int {
139     return m.Value.Type().NumIn()
140 }
141 func (m exportedMethod) ArgumentValue(i int) interface{} {
142     return reflect.Zero(m.Type().In(i)).Interface()
143 }
144 func (m exportedMethod) NumReturns() int {
145     return m.Value.Type().NumOut()
146 }
147 func (m exportedMethod) ReturnValue(i int) interface{} {
148     return reflect.Zero(m.Type().Out(i)).Interface()
149 }
150 func newExportedObject() *exportedObj {
151     return &exportedObj{
152         interfaces: make(map[string]*exportedIntf),
153     }
154 }
155 type exportedObj struct {
156     mu          sync.RWMutex
157     interfaces map[string]*exportedIntf
158 }
159 func (obj *exportedObj) LookupInterface(name string)
160 (Interface, bool) {
161     if name == "" {
162         return obj, true
163     }
164     obj.mu.RLock()
165     defer obj.mu.RUnlock()
166     intf, exists := obj.interfaces[name]
167     return intf, exists
168 }
169 func (obj *exportedObj) AddInterface(name string, iface
170 *exportedIntf) {

```

```

170     obj.mu.Lock()
171     defer obj.mu.Unlock()
172     obj.interfaces[name] = iface
173 }
174 func (obj *exportedObj) DeleteInterface(name string) {
175     obj.mu.Lock()
176     defer obj.mu.Unlock()
177     delete(obj.interfaces, name)
178 }
179 func (obj *exportedObj) LookupMethod(name string) (Method,
180 bool) {
181     obj.mu.RLock()
182     defer obj.mu.RUnlock()
183     for _, intf := range obj.interfaces {
184         method, exists := intf.LookupMethod(name)
185         if exists {
186             return method, exists
187         }
188     }
189     return nil, false
190 }
191 func (obj *exportedObj) isFallbackInterface() bool {
192     return false
193 }
194 func newExportedIntf(methods map[string]Method, includeSubtree
195 bool)
196 *exportedIntf {
197     return &exportedIntf{
198         methods:      methods,
199         includeSubtree: includeSubtree,
200     } }
201 type exportedIntf struct {
202     methods map[string]Method
203     // Whether or not this export is for the entire subtree
204     includeSubtree bool
205 }
206 func (obj *exportedIntf) LookupMethod(name string) (Method,
207 bool) {
208     out, exists := obj.methods[name]
209     return out, exists
210 }
211 func (obj *exportedIntf) isFallbackInterface() bool {
212     return obj.includeSubtree
213 }
214 //NewDefaultSignalHandler returns an instance of the default
215 //signal handler. This is useful if you want to implement only

```

```

213 //one of the two handlers but not both.
214 //
215 // Deprecated: this is the default value, don't use it, it will
    be unexported.
216 func NewDefaultSignalHandler() *defaultSignalHandler {
217     return &defaultSignalHandler{}
218 }
219 }
220 type defaultSignalHandler struct {
221     mu      sync.RWMutex
222     closed bool
223     signals []*signalChannelData
224 }
225 func (sh *defaultSignalHandler) DeliverSignal(intf, name
    string, signal *Signal)
226 {
227     sh.mu.RLock()
228     defer sh.mu.RUnlock()
229     if sh.closed {
230     return }
231     for _, scd := range sh.signals {
232         scd.deliver(signal)
233     }
234 func (sh *defaultSignalHandler) Terminate() {
235     sh.mu.Lock()
236     defer sh.mu.Unlock()
237     if sh.closed {
238     return }
239     for _, scd := range sh.signals {
240         scd.close()
241         close(scd.ch)
242     }
243     sh.closed = true
244     sh.signals = nil
245 }
246 func (sh *defaultSignalHandler) AddSignal(ch chan<- *Signal) {
247     sh.mu.Lock()
248     defer sh.mu.Unlock()
249     if sh.closed {
250     return }
251     sh.signals = append(sh.signals, &signalChannelData{
252         ch:    ch,
253         done: make(chan struct{}),
254     })
255 }
256 func (sh *defaultSignalHandler) RemoveSignal(ch chan<- *Signal)

```

```

257 {
258     sh.mu.Lock()
259     defer sh.mu.Unlock()
260     if sh.closed {
261         return }
262     for i := len(sh.signals) - 1; i >= 0; i-- {
263         if ch == sh.signals[i].ch {
264             sh.signals[i].close()
265             copy(sh.signals[i:], sh.signals[i+1:])
266             sh.signals[len(sh.signals)-1] = nil
267             sh.signals = sh.signals[:len(sh.signals)-1]
268         } }
269     }
270     type signalChannelData struct {
271
272         wg    sync.WaitGroup
273         ch    chan<- *Signal
274         done chan struct{}
275     }
276     func (scd *signalChannelData) deliver(signal *Signal) {
277         select {
278             case scd.ch <- signal:
279             case <-scd.done:
280                 return
281             default:
282                 scd.wg.Add(1)
283                 go scd.deferredDeliver(signal)
284         }
285     }
286     func (scd *signalChannelData) deferredDeliver(signal *Signal) {
287         select {
288             case scd.ch <- signal:
289             case <-scd.done:
290             }
291         scd.wg.Done()
292     }
293     func (scd *signalChannelData) close() {
294         close(scd.done)
295         scd.wg.Wait() // wait until all spawned goroutines return
296     }

```