Spectra ORB
C Edition

Naming Service Guide

PRISMTECH

Spectra ORB
C Edition

Naming Service Guide

PrismTech

Part Number: EORBC-NAMG

Doc Issue 25, 04 Jun 13

Copyright Notice

Contents

Table of Contents

Preface

About the Naming Service Guide

The *Naming Service Guide* explains how to use the Spectra ORB Naming Service C Edition product.

Intended Audience

The *Naming Service Guide* is intended to be used by developers who wish to integrate the Spectra ORB Naming Service into products which comply with OMG standards for object services. Readers who use this guide should have a good understanding of the relevant programming languages (for example C, IDL) and of the relevant underlying technologies (such as CORBA).

Organisation

The *Naming Service Guide* provides:

- a high level description and list of main features
- explanations of the OMG Naming Service architecture and concepts
- descriptions of how to configure, run and deploy the Spectra ORB Naming Service C Edition
- descriptions of how to create applications which use the Spectra ORB Naming Service.

Conventions

The conventions listed below are used to guide and assist the reader in understanding the Naming Service Guide.


Item of special significance or where caution needs to be taken.

i
Item contains helpful hint or special information.

WIN
Information applies to Windows (*e.g.* XP, Vista, Windows 7) only.

UNIX
Information applies to Unix-based systems (*e.g.* Solaris) only.

C
C language specific.

C++
C++ language specific.

Java
Java language specific.

Hypertext links are shown as *blue italic underlined.*

On-Line (PDF) versions of this document: Items shown as cross references, *e.g.* *Contacts* on page viii, are hypertext links: click on the reference to go to the item.

```
% Commands or input which the user enters on the
  command line of their computer terminal
```

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, “”);
```

Italics and ***Italic Bold*** indicate new terms, or emphasise an item.

Arial Bold indicates Graphical User Interface (GUI) elements and commands, for example, **File > Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters	**European Head Office**
PrismTech Corporation 400 TradeCenter Suite 5900 Woburn, MA 01801 USA	PrismTech Limited PrismTech House 5th Avenue Business Park Gateshead NE11 0NG UK
Tel: +1 781 569 5819	Tel: +44 (0)191 497 9900 Fax: +44 (0)191 497 9901

Web: *http://www.prismtech.com*
Technical questions: *crc@prismtech.com* (Customer Response Center)
Sales enquiries: *sales@prismtech.com*

Introduction

Description

The Naming Service provides a straightforward way of finding and using objects by associating meaningful, human-understandable names to those objects. The Naming Service is used like the white pages of a telephone directory to find an object and obtain its object reference, without the need to resort to complex programming or proprietary ORB mechanisms.

Product Description

The Spectra ORB Naming Service C Edition product is a full, CORBA-compliant Naming Service. The Spectra ORB Naming Service is fully compliant with the OMG's Naming Service Specification.

This implementation has been designed and implemented to be as small and as proficient as possible: it is designed for use in highly demanding environments where footprint size and resources are limited.

The Spectra ORB Naming Service C Edition does ***not*** support the `kind` property of the `CosNaming::NameComponent`. The Service will ignore this property when binding or resolving names. Developers should ensure that they use unique `id` values creating name components.

OMG Standard Features

The Spectra ORB Naming Service C Edition provides the following OMG specified features:

- give meaningful names to objects (*name bindings*)
- find names which have been bound to objects (*resolve*)
- group names in logical hierarchies (*naming contexts*)
- group distributed naming hierarchies (*federation*) (not supported)
- retrieve lists of names and step through them (*iteration*)

The Spectra ORB Naming Service

CHAPTER

1 Basic Concepts

This section describes the basic concepts and architecture of the standard Naming Service as defined in the OMG's Naming Service Specification. The features provided by this version of the service may vary from those described below. Refer to Section 2, Specific Features, on page 13 for those features provided with this version of the Naming Service.

1.1 OMG Standard Features

The Naming Service has the ability to:

- give meaningful names to objects (*name bindings*)
- allow objects, which have been bound to names, to be easily found (*resolve*)
- organise names in logical hierarchies (*naming contexts*)
- use *stringified names* to make it easier to identify and locate names
- retrieve lists of names and iterate through the list (*iteration*)

1.1.1 Names

The Naming Service associates meaningful names with objects. These names can be used to retrieve or reference the object, or in other words, obtain the object's IOR. An association between a name and an object is known as a *name binding*.

A *Naming Service name* is an object in its own right. A name contains a sequence of one or more *name components*. A name component has two string-type attributes, *id* and *kind*. The id and kind attributes identify the name.

A name contains a sequence of one or more name components. A name with a single name component is called a *simple name*. A name which contains a sequence of two or more name components is called a *compound name*. Compound names are used to access name bindings when they are in a hierarchy of *naming contexts*, described below.

A name binding is held in or otherwise associated with a *naming context*. A name binding cannot exist outside of a naming context. Names are bound to naming contexts, as well as to objects. It is possible to have orphaned contexts if the name binding is removed without keeping a reference to the context being unbound.

An object can be bound to one or more names, but a name can only be bound to one object. If an object is bound to more than one name, then any of those names can be used to locate the object.

Resolving is the process of locating an object or naming context by using its name.

Iteration is the process of iterating through a list of names with a binding iterator.

1.1.2 Naming Contexts

A naming context is an object which contains name bindings. Each name in a given naming context must be unique, in other words, the combination of a name's name component *id* and *kind* values must be unique. The name can be used in other naming contexts.

Naming hierarchies can be created by binding a naming context to another naming context. A simple naming context hierarchy is shown in *Figure 1*. Names in a naming context can refer to other naming contexts as well as to objects. A hierarchy of naming contexts is called a *naming graph*.

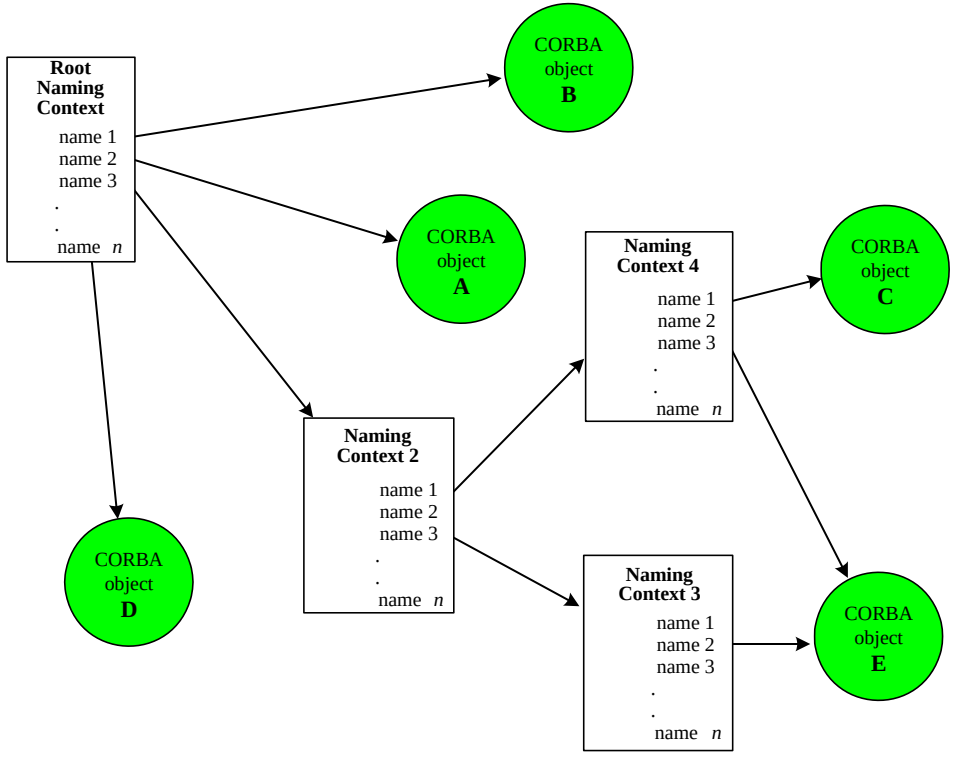

Figure 1 Example Naming Graph

The top level of the naming graph is called the *root context*. The root context is also the default context, that is, it does not need to be explicitly created whereas all other contexts, *child contexts* of the root context, must be explicitly created.

Objects, and contexts themselves, are referenced by following the hierarchy of naming contexts, starting from the root context and ending with the desired object or naming context. Technically, the object or context can be referenced in one of two ways:

1. Obtain an object reference to the context which contains the required object or context. The object's name can then be used to obtain the object's IOR.
2. Construct a compound name which contains a sequence of name components, where each name component identifies each successive naming context in the naming graph and where the last name component identifies the required object or context.

For example, in *Figure 1* objects *A*, *B* and *NamingContext2* are bound directly to the root context: they can be referenced using a simple name (containing the name component which identifies the objects themselves). Objects *C* and *E* are bound to child contexts lower down the hierarchy: in order to access objects *C* and *E* from the root context, the name component for each successive naming context must be provided as a compound name. For example, the compound name for referencing object C from the root context will contain a sequence which looks like this in pseudo-code:

```
name[0] = NamingContext2.nameComponent
name[1] = NamingContext4.nameComponent
name[2] = C.nameComponent
```

The root context is always implicit in a compound name; a special operation, `resolve_initial_references()`, is performed once to obtain the root context, and all subsequent `resolve` operations depend on that.

1.1.3 Stringified Names

A stringified name consists of the name components which are written in a user-convenient form where the name is written as a string with the name components of the naming graph (*i.e.* contexts and subcontexts) separated by a '/' character. For example, a name consisting of the components "a", "b", and "c" (in that order) is represented as *`a/b/c`*.

i The *`NamingContextExt`* interface, derived from `NamingContext`, is required when stringified names are used. Check that your version of the Naming Service supports the `NamingContextExt` interface before attempting to use stringified names in your application.

Stringified names use the '.' character to separate `id` and `kind` fields in the stringified representation. For example, the stringified name *`a.b/c.d/.`* represents the `CosNaming::Name`:

Index	id	kind
0	a	b
1	c	d
2	<empty>	<empty>

The single '.' character is the only representation of a name component with empty id and kind fields.

If a name component in a stringified name does not contain a '.' character, the entire component is interpreted as the id field, and the kind field is empty.

For example: *`a/./c.d/.e`* corresponds to the `CosNaming::Name`:

Index	id	kind
0	a	<empty>
1	<empty>	<empty>
2	c	d
3	<empty>	e

If a name component has a non-empty id field and an empty kind field, the stringified representation consists only of the id field. A trailing '.' character is not permitted.

1.1.3.1 Escape Mechanism

The backslash '\' character escapes the reserved meaning of '/', '.', and '\' in a stringified name. The meaning of any other character following a '\' is reserved for future use.

1.1.3.1.1 NameComponent Separators

If a name component contains a '/' slash character, the stringified representation uses the '\' character as an escape. For example, the stringified name *`a/x\/y\/z/b`* represents the name consisting of the name components "a", "x/y/z", and "b".

1.1.3.1.2 id and kind Fields

The backslash escape mechanism is also used for '.', so id and kind fields can contain a literal '.'. To illustrate, the stringified name `a\.b.c\.d/e.f`. represents the `CosNaming::Name`:

Index	id	kind
0	a.b	c.d
1	e	f

1.1.3.1.3 The Escape Character

The escape character '\' must be escaped if it appears in a name component.

For example, the stringified name `a/b\\/c` represents the name consisting of the components "a", "b\", and "c".

CHAPTER

2 *Specific Features*

The Spectra ORB Naming Service's features are listed here. As mentioned previously, the service conforms to the OMG's full Naming Service Specification.

The interfaces, datatypes, methods and exceptions supported by the Spectra ORB Naming Service are listed below.

Interfaces and Datatypes

- `Name` (datatype)
- `NameComponent` (datatype)
- `Binding` (datatype)
- `NamingContext` (interface)
- `BindingIterator` (interface)

NamingContext Methods

- `bind()`
- `rebind()`
- `bind_context()`
- `rebind_context()`
- `resolve()`
- `unbind()`
- `new_context()`
- `bind_new_context()`
- `destroy()`
- `list()`

NamingContext Exceptions

- `NotFoundReason`
- `NotFound`
- `CannotProceed`
- `InvalidName`
- `AlreadyBound`

- `NotEmpty`

BindingIterator

- `next_one()`
- `next_n()`
- `destroy()`

2.1 Naming Context Creation, and Destruction

The Naming Service supports three methods to create new contexts, *`new_context()`*, *`bind_new_context()`* and *`rebind_context()`*, and a single method to remove contexts, *`destroy()`*.

new_context()

```
NamingContext new_context()
```

The `new_context()` method returns a `NamingContext` object. The new context is not bound to any name. Contexts which are created with the `new_context()` method must use *`bind_context()`* to bind a name to the new context.

bind_new_context()

```
NamingContext bind_new_context(in Name n)
   raises (NotFound, AlreadyBound, CannotProceed,
      InvalidName);
```

The `bind_new_context()` method creates a `NamingContext` object and binds it to the supplied name. This method effectively combines the `new_context()` and `bind_context()` methods into a single operation.

rebind_context()

```
void rebind_context(in Name n, in NamingContext nc)
   raises (NotFound, CannotProceed, InvalidName)
```

The *`rebind_context()`* method binds a name to a `NamingContext` object such that:

- if the context is already bound to a name, then the name binding is replaced with the new name binding
- a new name binding is created if one does not already exist
- the object being bound to a name must be to a `NamingContext` object (binding type of *`ncontext`*) and ***not*** to a CORBA object (binding type of *`nobject`*).

destroy()

```
void destroy() raises (NotEmpty);
```

The `destroy()` method requests the destruction of a `NamingContext` object. The naming context must be empty. After `destroy` is invoked, no further operations can be invoked on the object reference of the naming context object.

Bindings to a destroyed context are not removed. To do so would require a context to know about all of its parents as well as its children. An attempt to resolve a binding to a destroyed context will throw the `CORBA.INV_OBJREF` exception. Accordingly, bindings to a naming context should be removed before it is destroyed.

When a hierarchical name is used to create a new context, all the contexts that constitute the path to the new context must already exist or the *`NotFound`* exception will be raised

2.2 Object Binding and Unbinding

The `NamingContext` interface provides the object binding and unbinding operations described below.

bind()

```
void bind (in Name n, in Object obj)
   raises (NotFound, CannotProceed, InvalidName,
   AlreadyBound)
```

The `bind()` method binds a name, which is a `NameComponent` object, to a CORBA object or a naming context.

rebind()

```
void rebind (in Name n, in Object obj)
   raises (NotFound, CannotProceed, InvalidName)
```

The `rebind()` method binds a name, which is a `NameComponent` object, to a CORBA object such that:

- if an object is already bound to a name, then the name binding is replaced with the new name binding
- a new name binding is created if one a does not already exist
- the object being bound to a name must be to a CORBA object (binding type of *`nobject`*) and ***not*** to a naming context (binding type of *`ncontext`*).

unbind()

```
void unbind (in Name n)
   raises (NotFound, CannotProceed, InvalidName)
```

The `unbind()` method removes a name binding from a context. This operation does not destroy or otherwise affect the object that was bound to the name.

2.3 Accessing Objects and Naming Contexts

Object references to objects and naming contexts are obtained with the *resolve()* method. Lists of name bindings objects can be obtained with the `list()` method.

resolve()

```
Object resolve (in Name n)
   raises (NotFound, CannotProceed, InvalidName)
```

The method takes a name, a `NameComponent` object, and returns the object that the name is bound to. If the name binding does not exist or is invalid, then the `NotFound` or `InvalidName` exceptions are raised.

list()

```
void list (in unsigned long how_many,
   out BindingList bl, out BindingIterator bi)
```

`list()` returns the bindings contained in a context as in as the *bl* parameter. The `bl` parameter is a `BindingList` (a sequence where each element is a Binding containing a Name of length 1 representing a single NameComponent).

The `how_many` parameter sets the maximum number of bindings to return in the `bl` parameter; any remaining bindings are passed in the returned `BindingIterator bi` parameter.

- A non-zero value of `how_many` guarantees that `bl` contains at most `how_many` elements. The number of bindings returned may be fewer than as requested by `how_many`. If `how_many` is non-zero, then it may not return a `bl` sequence with zero elements unless the context contains no bindings.
- If `how_many` is set to zero (0), then the client is requesting to use only the `BindingIterator bi` to access the bindings and list returns a zero length sequence in `bl`.
- The `bi` parameter returns a reference to a `BindingIterator` object.
- If the `bi` parameter returns a non-nil reference, then this indicates that the call to list may not have returned all of the bindings in the context and that the remaining bindings, if any, must be retrieved using the iterator. This applies for all values of `how_many`.
- If the `bi` parameter returns a nil reference, then this indicates that the `bl` parameter contains all of the bindings in the context. This applies for all values of `how_many`.

2.4 BindingIterator

The `BindingIterator` enables a client to iterate through the bindings using the *`next_one()`* or *`next_n()`* methods. A *`destroy()`* method is provided which destroys the iterator and frees its associated memory.

next_one()

```
boolean next_one (out Binding b)
```

The `next_one()` operation sets the next binding as an *`out`* parameter and returns true if successfully returning a binding. The method returns false if there are no more bindings to retrieve. If `next_one()` returns false, then any binding which might be returned will be indeterminate. Calls to `next_one()` after it has returned false have undefined behaviour.

next_n()

```
boolean next_n (in unsigned long how_many,
   out BindingList bl);
```

The `next_n()` method returns bindings, as a `BindingList`, which have not yet previously been retrieved from the BindingList bl by the either the `list()`, `next_one()` or `next_n()` methods. The `how_many` parameter sets the maximum number of bindings which will be returned.

For example, if `how_many` is set to 50 and there are 100 bindings remaining, then only 50 bindings will be returned; if there are only 25 bindings remaining, then only 25 binding will be returned.

i Setting `how_many` to 0 (zero) is not allowed and raises a BAD_PARAM system exception if done so.

If all bindings have been retrieved, the `how_many` returns false and sets the `BindingList bl` parameter to zero length.

destroy()

```
void destroy()
```

The `destroy()` operation destroys its iterator. If a client invokes any operation on an iterator after calling destroy, the operation raises `OBJECT_NOT_EXIST`.

CHAPTER

3 Using the Service

*This section describes the specific procedures and requirements for creating and running CORBA-based applications with Spectra ORB Naming Service C Edition. (Please note that this section is **not** intended as a tutorial of how to write CORBA-based applications with the Naming Service.)*

3.1 Running the Service

The Naming Service can be run from the command line or by embedding it into application or module code. Instructions for running the service using these methods is described in the following sections, *Embedding the Service*, below, and *Running from the Command Line* on page 22.

3.1.1 Embedding the Service

The following basic tasks must be performed in order to embed a Naming Service instance into an executable or code module:

Step 1: Include the following *`#include`* statements in your code:

```
#include "CosNaming.h"
#include "eOrbC/EORB/NamingService.h"
```

i Ensure your build system has *`$(EORBHOME)/include/services/full`* on its include path.

Step 2: Configure the service instance by setting the property fields in the naming service's configuration structure: these properties are used to determine specific aspects of your instance's behaviour.

Step 3: Initialise a Naming Service instance by using the Naming Service's *`init()`* method:

```
root_ctx = EORB_NamingService_init(orb, poa, config, ev)
```

where:

`orb` is the orb which the service is to be run on
`poa` is the POA which the service is to run in (see *POA Argument Choices* below)
`config` is the service instance's configuration (see *Configuration Structure* below)
`ev` is the CORBA Environment

Step 4: Link with the *ec_naming_s* library.

3.1.1.1 POA Argument Choices

The *poa* agrument allows the user to create their own poa for the Naming Service that has properties configured to meet the demands of their system or implementation.

There are two choices for the poa argument: `NULL` or User Defined POA

1. NULL

   In this case the `EORB_NamingService` will create a POA where:

   id assignment policy is set to *PortableServer_USER_ID*
   id_uniqueness_policy is set to *PortableServer_MULTIPLE_ID*

   lifespan_policy is set to *PortableServer_PERSISTENT* if the `config` option *qosPersistent* is set to `TRUE`, otherwise it will be set to *PortableServer_TRANSIENT*

2. User Defined POA

   The User Defined POA ***MUST*** have the following policies set:

   id assignment policy set to *PortableServer_USER_ID*
   id_uniqueness_policy set to *PortableServer_MULTIPLE_ID*

   The `config` option *qosPersistent* will be ignored when a user defined POA is provided. If persistence is required, then the appropriate policies must be set on the user defined POA.

   All other policies can be set as needed.

3.1.1.2 Configuration Structure

The structure mentioned in *Step 2:* above defines the property fields described below under Table 1, *Configuration Property Descriptions*. An example of setting the configuration structure fields is shown in Example 1, *Setting the Configuration Structure Fields*, on page 21.

Table 1 Configuration Property Descriptions

Property	Description
`qosContextLocking`	Controls the read and write locking protection for concurrent access to naming contexts. `ctx_rwLock` should normally only be set to `false`, unlocked, when the service is used in read-only mode.
`qosMaxContexts`	The number of naming contexts that this service instance is expected to create.
`qosPersistent`	Creates a persistent IOR for the Naming Service and causes it to listen on a fixed port. This enables the Naming Service to always be resolved using the same IOR or CORBALOC, even when the service has been shutdown and restarted.

Example 1 Setting the Configuration Structure Fields

```
EORB_NamingService_Config config;

config.qosContextLocking = TRUE;
config.qosMaxContexts    = 100;
config.qosPersistent     = TRUE;
```

3.1.1.3 Simple Example

The following ***C*** code example shows how a Naming Service instance can be created.

C

```
int main (int argc, char ** argv)
{
   CORBA_Environment env;
   CosNaming_NamingContext context;

   /* Initialize Naming Service config */

   EORB_NamingService_Config config;

   config.qosPersistent     = TRUE;
   config.qosContextLocking = TRUE;
   config.qosMaxContexts    = 100;

   /* Hook in Minimum POA and IIOP profile */

   EORB_plugin (EORB_POA);
   EORB_plugin (EORB_IIOP);

   memset (&env, 0, sizeof (CORBA_Environment));

   orb = CORBA_ORB_init (&argc, argv, "eorb-ce", &env);

   context = EORB_NamingService_init (orb, NULL, &config, &env);
```

```
   CORBA_Object_release (context, &env);

   /* Run ORB */

   CORBA_ORB_run (orb, &env);
   return 0;
}
```

3.1.2 Running from the Command Line

The Spectra ORB Naming Service executable, *namingc*, is located in the `$EORBHOME/bin/$EORBENV` directory. The `naming` executable can be run with zero or more of the command line options listed in *Table 2*.

Table 2 Command Line Options

Option	Description
`-NameServiceContextLocking <on|off>`	Sets context locking state (see above). The default is *on*.
`-NameServiceMaxContexts <num>`	Sets expected number of contexts to be created. The default is *100*.
`-NameServicePersistent <yes|no>`	Sets whether the lifespan policy is set to `PortableServer_PERSISTENT` or `PortableServer_TRANSIENT` The default is *yes* (= *_PERSISTENT*).
`-NameServiceUIOP`	Runs the service on a UIOP endpoint. Only available on systems where UIOP is a supported transport. The default is `no`.

3.1.2.1 Example

The following command line example demonstrates how to start a Naming Service instance, on a UNIX operating system, where:

- the expected number of contexts to be created is *250*

```
%  namingc -NameServiceMaxContexts 250
```

3.1.3 Running on a Fixed Endpoint

The server can be run on a fixed endpoint by running with the `-ORBPOAEndpoints` argument. The Naming Service servant is created within a child POA *NameService* so for example can be run on a fixed IIOP endpoint with:

```
-ORBPOAEndpoints NameService:iiop:<host>:<port>
```

and resolved as an initial reference by a client using:

```
-ORBInitRef
   NameService=corbaloc:iiop:<host>:<port>/NameService
```

CHAPTER

4 Creating Applications

This section describes how to create applications by using specific features of the Spectra ORB Naming Service, including:

- *creating and destroying naming contexts and name bindings*
- *retrieving the contents of a naming context*
- *how to resolve a binding to an object.*

Examples are provided which demonstrate how these features can be implemented.

This release of the Spectra ORB Naming Service C Edition does not support the `kind` property of the *`CosNaming_NameComponent`*.

Note

- For the sake of clarity and brevity, the examples shown here do not have all of the error or exception trapping code normally used. Exceptions and errors must, naturally, be caught and properly handled in a production system.
- Applications which ***use*** the Spectra ORB Naming Service must
  - have a *`#include "CosNaming.h"`* line in the source code file and ensure your build system has *`$(EORBHOME)/include/services/full`* on its include path (see *Embedding the Service* on page 19)
  - have a *`#include “eOrbC/EORB/NamingService.h”`* line in the source code file
  - link the *`ec_naming_c.lib`* library file
- Applications or modules which ***create and run*** Spectra ORB Naming Service instances must also:
  - include the *`NamingService.h`* header file in the source code file
  - be implemented and configured as described in Chapter 3, *Using the Service*.

4.1 Obtaining the Root Context

Before any objects or naming contexts can be added to (bound) or located (resolved) in the Spectra ORB Naming Service, the root or initial context must be resolved. This is achieved by first obtaining the root context’s IOR, then passing it to the *`CORBA_ORB_string_to_object`* function. This IOR is located, as a string, in a

file created by the Spectra ORB Naming Service server. The following example function, *obtainRootContext*, shows how the IOR can be obtained and used to resolve the root context.

```
static void obtainRootContext (CORBA_ORB orb, CORBA_Environment *env)
{
   FILE *iorfile;
   char *filename = "naming.ior";
   char ior_str[256];

   iorfile = fopen (filename, "r");
   fgets (ior_str, 256, iorfile);

   root_ctx = CORBA_ORB_string_to_object (orb, ior_str, env);
   assert (root_ctx);
   fclose (iorfile);
}
```

4.2 Naming Context Creation and Destruction

The `NamingContext` interface provides two `NamingContext` creation operations and a single destroy operation, defined in IDL as:

```
NamingContext new_context ();

NamingContext bind_new_context (in Name n)
   raises (NotFound, CannotProceed, InvalidName, AlreadyBound);

void destroy () raises (NotEmpty);
```

The *new_context()* operation creates a new `NamingContext` object which is not bound to any other Naming Context:

The *bind_new_context()* operation creates a new Naming Context and binds it using the supplied name.

The *destroy* operation requests the destruction of a `NamingContext` object. The naming context must be empty. After `destroy` is invoked, no further operations can be invoked on the object reference of the naming context object.

Bindings to a destroyed context are not removed. To do so would require a context to know about all of its parents as well as its children. An attempt to resolve a binding to a destroyed context will throw the `CORBA.INV_OBJREF` exception. Accordingly, bindings to a naming context should be removed before it is destroyed.

4.3 Binding and Unbinding Operations

The `NamingContext` interface provides five bind operations and a single unbind operation, defined in IDL as:

```
void bind (in Name n, in Object obj)
   raises (NotFound, CannotProceed, InvalidName, AlreadyBound);

void rebind (in Name n, in Object obj)
   raises (NotFound, CannotProceed, InvalidName);

void bind_context (in Name n, in NamingContext nc)
   raises (NotFound, CannotProceed, InvalidName, AlreadyBound);

void rebind_context (in Name n, in NamingContext nc)
   raises (NotFound, CannotProceed, InvalidName);

NamingContext bind_new_context (in Name n)
   raises (NotFound, CannotProceed, InvalidName, AlreadyBound);

void unbind (in Name n)
   raises (NotFound, CannotProceed, InvalidName);
```

The bind operations allow binding to occur between a name and either a generic CORBA object or a Naming Context. In order to bind a CORBA object, the name to bind against must be correctly constructed. Given a name with *n* components, the first *n - 1* components must resolve to a bound `NamingContext`.

i

The implementation for this version of the Spectra ORB Naming Service does not support the *NamingContextExt* interface. However, in order to facilitate the creation of *CosNaming_Name* (a `NameComponent`), a utility function, *EORB_NamingService_to_name()*, is provided and can be used to create name components.

The *EORB_NamingService_to_name()* function signature is:

```
CosNaming_Name *EORB_NamingService_to_name(char *name,
   CORBA_Environment *env)
```

The `NameComponent` *kind* element is not supported: name components can only be specified using the *id* element, where each `id` value is unique within a naming context.

Stringified names are supported, together with the use of the "\" escape character, as in accordance with the OMG specification.

The *rebind* operation is identical to the `bind` operation except that the `AlreadyBound` exception is not thrown; an existing binding with the same name is replaced by the new binding.

The *bind_context* operation adds a NamingContext object so that it becomes part of the graph of Naming Contexts used for resolving compound names. Note that a NamingContext can be also be added using the bind operation but that the NamingContext will not become part of the graph of Naming Contexts and will not be used for resolving compound names.

The *rebind_context* operation is identical to the bind_context operation except that the AlreadyBound exception is not thrown; an existing binding with the same name is replaced by the new binding.

The *bind_new_context* operation is equivalent to creating a new NamingContext and then adding it using bind_context:

The following example uses a function called *example* to create ten naming contexts, assigns a unique name to each, then binds them to the root context using *CosNaming_NamingContext_bind_new_context*. (The names created in the example, *context_N* where *N* is a sequential value, are generated by the example function, *create_name_string*.

```
#include "CosNaming.h"

static CosNaming_NamingContext root_ctx;

/* utility */
static void create_name_string (int i, char * name, char ** result)
{
   char num [8];

   sprintf (num, "%d", i);
   strcpy (*result, name);
   strcat (*result, num);
}
static void example (CORBA_Environment *env)
{
   CosNaming_Name *cos_name;
   CosNaming_BindingList *list;
   CosNaming_BindingIterator iterator;
   CosNaming_BindingType bindingtype;
   CosNaming_Binding *cos_binding;
   CosNaming_NamingContext ctx, recovered;
   CORBA_Object object;
   char *name;
   int i;

   /* bind 10 contexts into root context */
   for (i = 0; i < 10; i++)
   {
      name = malloc (32);
      create_name_string (i + 1, "context_", &name);

     cos_name = (CosNaming_Name*) EORB_NamingService_to_name (name,
env);
      ctx = CosNaming_NamingContext_bind_new_context (root_ctx,
cos_name, env);
```

```
      free (name);
      CORBA_free (cos_name);
      CORBA_Object_release (ctx, &env);
   }
```

The unbind operation, *CosNamingContext_unbind(CosNamingContext, CORBA_SequenceNameComponent, CORBA_Environment)*, removes a name binding. It does not matter which of the bind operations was used to create the binding. The following example destroys bindings created with the previous example:

```
static void bind_unbind_example (CORBA_Environment *env)
{
   CosNaming_Name *cos_name;
   CosNaming_NamingContext object;
   char *name = "entry_1";

   cos_name = (CosNaming_Name*) EORB_NamingService_to_name (name,
env);
   object = CosNaming_NamingContext_bind_new_context (root_ctx,
cos_name, env);

   CosNaming_NamingContext_unbind (root_ctx, cos_name, env);

   CORBA_free (cos_name);
   CORBA_Object_release (object, &env);
}
```

4.4 Accessing Naming Context Contents

Two operations are available for accessing the contents of naming contexts, defined in IDL as:

```
Object resolve (in Name n)
    raises (NotFound, CannotProceed, InvalidName);
void list (in unsigned long how_many, out BindingList bl,
    out BindingIterator bi);
```

The `resolve` operation returns the object, if any, bound to the name passed to it.

The `list` operation provides a means of accessing the entire content of a Naming Context. The `list` operation is the only means of determining the name bindings held by an arbitrary context. This operation returns results using two mechanisms: a `BindingList`, which is a sequence of bindings, and a `BindingIterator` which provides an iterator object to access the bindings.

The following example (a continuation of the example code shown above) lists the first five naming contexts and resolves each of them.

```
   /* obtain a list of first 5 */
  CosNaming_NamingContext_list (root_ctx, 5, &list, &iterator, env);
```

```
   /* process list */
   printf ("\nRecovering 5 contexts using list\n");
   for (i = 0; i < list->_length; i++)
   {
      bindingtype = list->_buffer[i].binding_type;
      cos_name = list->_buffer[i].binding_name;
      printf ("\tRecovered %s\n", cos_name->_buffer[0].id);
      object = CosNaming_NamingContext_resolve (root_ctx, cos_name,
env);

      if (bindingtype == CosNaming_ncontext)
      {
         recovered = (CosNaming_NamingContext) object;

         /* do something with context */

         CORBA_Object_release (recovered, &env);
      }
      else /* is CosNaming_nobject */
      {
         /* cast object to correct object type */
         /* do something with object */
      }
      CORBA_free (cos_name);
   }
   CORBA_free (list);
```

4.5 BindingIterator Operations

The `BindingIterator` interface provides two operations to access bindings and one destroy operation, defined in IDL as:

```
boolean next_one (out Binding b);
boolean next_n (in unsigned long how_many, out BindingList bl);
void destroy ();
```

The `next_one` operation returns `true` when passed a valid binding.

The `next_n` operation returns the number of bindings specified by the `how_many` variable in a `BindingList` sequence. The sequence is then accessed in the same way as the `BindingList` returned from a `NamingContext list` operation.

The following code fragment repeats the example of the `list` operation using the `next_one` operation to iterate through the contents:

```
   /* process iterator */
   printf ("\nRecovering 5 contexts using iterator\n");
  while (CosNaming_BindingIterator_next_one (iterator, &cos_binding,
env))
   {
      bindingtype = cos_binding->binding_type;
      cos_name = cos_binding->binding_name;
      printf ("\tRecovered %s\n", cos_name->_buffer[0].id);
      object = CosNaming_NamingContext_resolve (root_ctx, cos_name,
env);
```

```
      if (bindingtype == CosNaming_ncontext)
      {
         recovered = (CosNaming_NamingContext) object;

         /* do something with context */

         CORBA_Object_release (recovered, &env);
      }
      else /* is CosNaming_nobject */
      {
         /* cast object to correct object type */
         /* do something with object */
      }
      CORBA_free (cos_name);
   }

   CORBA_free (cos_binding);
   CosNaming_BindingIterator_destroy (iterator, env);
   CORBA_Object_release (iterator, &env);
}
```

CHAPTER

5 Supplemental Information

5.1 XML Export

i

XML Export is supported only by the ***C***-version of the service.

The Spectra ORB Naming Service can export XML files containing a representation of a naming hierarchy. This is performed at the command line; a specific naming hierarchy of a single Spectra ORB Naming Service instance is handled with a single command.

To export a naming hierarchy to an XML file, the following utility function is provided:

C

```
#include "EORB/NamingService.h"

void export (
   char* filename,
   CosNaming_NamingContext context,
   CORBA_ORB orb,
   CORBA_Environment * env)
```

5.2 Exceptions

The exceptions raised by the Spectra ORB Naming Service are listed in *Table 3*.

Table 3 Spectra ORB Naming Service Exceptions

Name	Purpose
AlreadyBound	Indicates an object is already bound to the specified name. Only one object can be bound to a particular name in a context.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context. One possible reason for this exception is that a Name Server holding one or more of the name bindings within a compound name is currently unavailable.
InvalidName	Indicates that the name is invalid. This implementation disallows zero length names only.
NotEmpty	Indicates that a naming context has bindings.
NotFound	Indicates that the name does not identify a binding or that the binding is not of the type required for the requested operations.

Index

Index

O

P

R

S

U

X