Spectra ORB
C++ Edition

Naming Service Guide

PRISMTECH

Spectra ORB
C++ Edition

Naming Service Guide

Part Number: EORBCPP-NAMG

Doc Issue 31, 4 June 2013

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
- an explanation of the OMG Naming Service architecture and concepts
- descriptions of how to configure, run and deploy the Spectra ORB Naming Service
- descriptions of how to create applications which use the Spectra ORB Naming Service
- other information about the 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 to other parts of the document, *e.g. Contacts* on page viii, behave as hypertext links: users can jump to that section of the document by clicking on the cross reference.

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

`Courier`, **`Courier Bold`**, or *`Courier Italic`* fonts indicate programming code and files 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”, “”);

rootContext.bind (newName, demoObject);
```

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

Arial Bold indicates user related actions, *e.g.* **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 Spectra ORB Naming Service C++ Edition provides a straightforward way of finding and using objects by associating meaningful, human-understandable names to those objects. The Spectra ORB 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.

The Spectra ORB Naming Service C++ Edition product is a complete, CORBA-compliant Naming Service. The Spectra ORB Naming Service is fully compliant with the OMG's Naming Service Specification and satisfies the Naming Service requirements of the Software Communications Architecture (SCA).

This release of 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 when creating name component.s

OMG Standard Features

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

- enable objects to be located by using human-intelligible names by binding names to objects
- add and remove name bindings
- change the object which a name is bound to
- group names in logical hierarchies
- retrieval of and iteration through lists of names

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*.

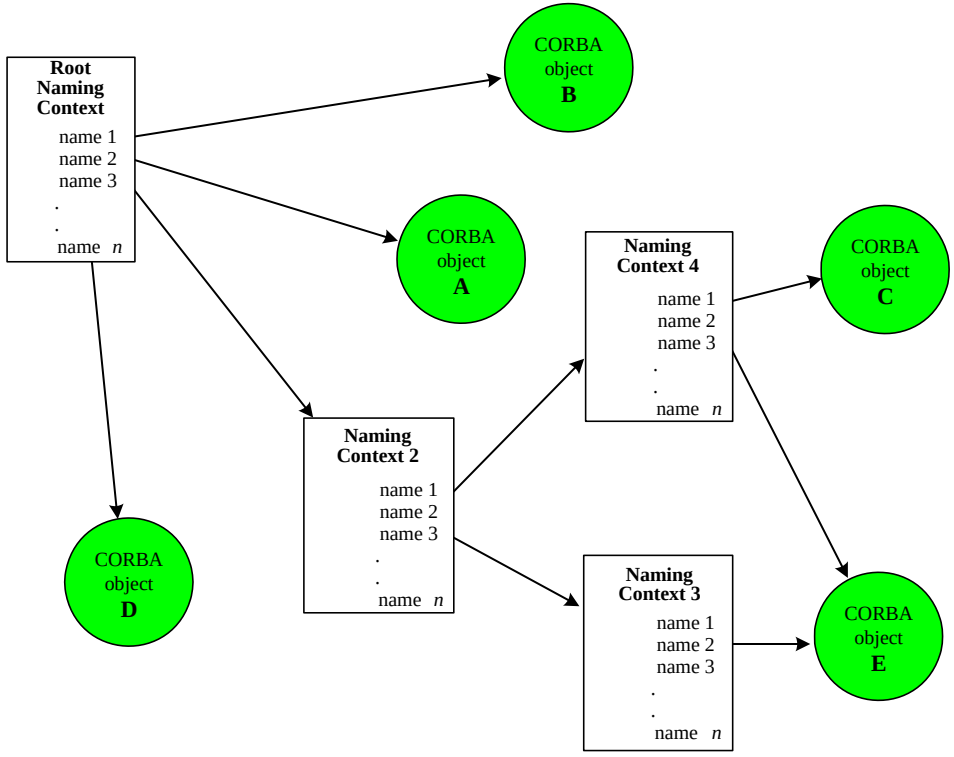

Figure 1 Example Naming Graph

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 described below. 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

2.1.1 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.

2.1.2 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`.

2.1.3 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 an `OBJECT_NOT_EXIST` system exception.

CHAPTER

3 Using the Service

*This section describes the specific procedures and requirements for creating and running CORBA-based applications with the 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 instances can be run from the command line or by embedding 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 23.

3.1.1 Embedding the Service

Refer to the *Platforms* section of the *User Guide* for specific compiler and linking details for your platform.

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

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

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

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

Step 2: When compiling and linking instruct the linker to link the *`e_naming_s.lib`* library file.

Step 3: 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 and are described later.

Step 4: Initialise a Naming Service instance by using the Naming Service's *`init()`* method:

```
root_ctx = EORB::NamingService::init(orb, poa, config)
```

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)

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 *PortableServe::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 3:* above defines the property fields described below in 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. *`qosContextLocking`* should normally only be set to *`false`*, unlocked, when the service is used in read-only mode. The default is *`true`*, locked.
`qosMaxContexts`	The number of naming contexts that this service instance is expected to create. The default value is 1024.
`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.qosMaxContexts = 500;
```

3.1.1.3 Example Server Module

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

```
#include "eOrb/EORB/Plugin/Current.h"
#include "eorb_name_i.h"
#include "eOrb/EORB/NamingService.h"

EORB_MAIN (server)
{
   EORB_DECLARE_ENV;

   eorb_name_impl * servant = 0;
   EORB::NamingService::Config config;
   config.qosMaxContexts = 11;
   config.qosPersistent = 0;

   EORB::Plugin::Current::add ();
   EORB::Plugin::IIOP::add ();
   EORB::Plugin::POA::add ();

   EORB_Stdio_plugin ();
   EORB_File_plugin ();

   EORB_TRY
   {
      printf ("Naming server starting\n");
```

```
      // Naming Service setup

      CosNaming::NamingContext_var ctx;
      PortableServer::POA_var poa;
      PortableServer::POA_var child_poa;
      PortableServer::ObjectId_var oid;
      CORBA::Object_var obj;
      CORBA::ORB_var orb;

      orb = CORBA::ORB_init (argc, argv EORB_ENV_VARN);
      EORB_CHECK_ENV;

      // Get the RootPOA

      obj = orb->resolve_initial_references ("RootPOA"
EORB_ENV_VARN);
      EORB_CHECK_ENV;

      poa = PortableServer::POA::_narrow (obj EORB_ENV_VARN);
      EORB_CHECK_ENV;

      // Create servant and activate it

      servant = new eorb_name_impl (orb EORB_ENV_VARN);
      EORB_CHECK_ENV;

      oid = poa->activate_object (servant EORB_ENV_VARN);
      EORB_CHECK_ENV;

      EORBTest::name_var server = servant->_this (EORB_ENV_VAR1);
      EORB_CHECK_ENV;

      // Initialise a naming service

      ctx = EORB::NamingService::init
      (
         orb,
         NULL,
         config
         EORB_ENV_VARN
      );
      EORB_CHECK_ENV;

      orb->register_initial_reference ("NameService", ctx
EORB_ENV_VARN);
      EORB_CHECK_ENV;

      orb->register_initial_reference ("server", server
EORB_ENV_VARN);
      EORB_CHECK_ENV;

      printf ("Name Service started...\n");

      orb->run (EORB_ENV_VAR1);
      EORB_CHECK_ENV;

      orb->destroy (EORB_ENV_VAR1);
      EORB_CHECK_ENV;
   }
```

```
   EORB_CATCH (CORBA::Exception, exc)
   {
      printf ("Exception %s\n", exc._rep_id ());
   }
   EORB_END_TRY

   delete servant;

   printf ("Naming server complete\n");

   return 0;
}
```

3.1.2 Running from the Command Line

An alternative to writing a module which creates and runs a Naming Service server is to run the Spectra ORB Naming Service executable, *namingcpp*, from the command line. *namingcpp* is located in the `$EORBHOME/bin/$EORBENV` directory. The `naming` executable can be run with zero or more of the command line options listed below in *Table 2*.

Table 2 Command Line Options

Option	Description
`-ORBNameServiceContextLocking <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 *100*

```
%  namingcpp -NameServiceMaxContexts 100
```

3.1.3 Running on a Fixed Endpoint

The server can be run on a fixed endpoint by running with the `-ORBPOAEndpoints` argument. The Name 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 client-server applications which use the Spectra ORB Naming Service. Topics covered include:

- *clients and servers that use the Naming Service for object resolution*
- *creating and destroying naming contexts*
- *associating objects with name bindings and adding them to naming contexts*
- *resolving objects by using name bindings.*

This release of the Spectra ORB Naming Service C++ Edition does ***not*** support the `kind` property of the `CosNaming::NameComponent`. Although the examples shown here use the `kind` property, it is nonetheless ignored by the Service when binding or resolving names.

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 (see *Embedding the Service* on page 19)
  - link with the `e_naming_cpp.lib` library file
- Applications or modules which ***create and run*** Spectra ORB Naming Service instances must also:
  - include the `eOrb/EORB/NamingService.h` header file in the source code file
  - link with the `e_naming_s.lib` library file
  - be implemented and configured as described in Chapter 3, *Using the Service*.

4.1 A Basic Application

A basic client-server application which uses the Spectra ORB Naming Service must:

- enable its server and client components to obtain an object reference to a running Spectra ORB Naming Service server instance (referred to here as the *naming service* for brevity)

- register objects with the naming service
- retrieve or resolve the objects registered with the naming service

An example application demonstrates how these tasks can be performed. The application:

- has a server which
  - creates and activates a Spectra ORB Naming Service servant
- has a client which
  - obtains a reference to the servant
  - creates and binds contexts and objects with the naming service
  - retrieves bindings using list and iterator methods
  - unbinds objects and contexts
  - destroys contexts

The complete example application is supplied as source files in *`examples/cpp/services/naming`* below the installation directory.

4.1.1 Registering Objects

The server must make its servants available to clients, in other words, it must provide some mechanism which enables clients to resolve objects. It can do this using various methods, such as saving the objects' IORs to a file or by using the naming service (see the Spectra ORB's *User Guide* for other alternatives): the example naming server outputs its object reference as a string.

4.1.2 Obtaining the Root Context

The first task for a client is to obtain the *root context* of a naming service.

The root context is a normal CORBA object and can be obtained using the standard CORBA methods for object resolution. For example, a separate module can create a naming service instance and publish its stringified IOR in a file called *`NamingService.ior`*.[1] The server can use this naming service by retrieving its stringified IOR and converting it to an object reference.

The following example server code initialises the ORB then creates and activates a servant.

4.1.2.1 Example server source code

The complete source code for the supplied example server is shown below:

```
#include "eOrb/EORB/Plugin/Current.h"
```

1. Another example module which creates a Naming Service server is shown under *Example Server Module* on page 17.

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

EORB_MAIN (server)
{
   EORB_DECLARE_ENV;

   eorb_name_impl * servant = 0;
   EORB::NamingService::Config config;
   config.qosMaxContexts = 11;
   config.qosPersistent = 0;

   EORB::Plugin::Current::add ();
   EORB::Plugin::IIOP::add ();
   EORB::Plugin::POA::add ();

   EORB_Stdio_plugin ();
   EORB_File_plugin ();

   EORB_TRY
   {
      printf ("Naming server starting\n");

      // Naming Service setup

      CosNaming::NamingContext_var ctx;
      PortableServer::POA_var poa;
      PortableServer::POA_var child_poa;
      PortableServer::ObjectId_var oid;
      CORBA::Object_var obj;
      CORBA::ORB_var orb;

      orb = CORBA::ORB_init (argc, argv EORB_ENV_VARN);
      EORB_CHECK_ENV;

      // Get the RootPOA

      obj = orb->resolve_initial_references ("RootPOA"
EORB_ENV_VARN);
      EORB_CHECK_ENV;

      poa = PortableServer::POA::_narrow (obj EORB_ENV_VARN);
      EORB_CHECK_ENV;

      // Create servant and activate it

      servant = new eorb_name_impl (orb EORB_ENV_VARN);
      EORB_CHECK_ENV;

      oid = poa->activate_object (servant EORB_ENV_VARN);
      EORB_CHECK_ENV;

      EORBTest::name_var server = servant->_this (EORB_ENV_VAR1);
      EORB_CHECK_ENV;

      // Initialise a naming service

      ctx = EORB::NamingService::init
      (
         orb,
```

```
         NULL,
         config
         EORB_ENV_VARN
      );
      EORB_CHECK_ENV;

      orb->register_initial_reference ("NameService", ctx
EORB_ENV_VARN);
      EORB_CHECK_ENV;

      orb->register_initial_reference ("server", server
EORB_ENV_VARN);
      EORB_CHECK_ENV;

      printf ("Name Service started...\n");

      orb->run (EORB_ENV_VAR1);
      EORB_CHECK_ENV;

      orb->destroy (EORB_ENV_VAR1);
      EORB_CHECK_ENV;
   }
   EORB_CATCH (CORBA::Exception, exc)
   {
      printf ("Exception %s\n", exc._rep_id ());
   }
   EORB_END_TRY

   delete servant;

   printf ("Naming server complete\n");

   return 0;
}
```

4.1.2.2 Example client source code

The complete source code for the supplied example client is shown below:

```
#include "CosNaming.h"
#include "eorb_name.h"

void full_naming_examples
(
   CosNaming::NamingContext_ptr root_ctx
   EORB_ENV_ARGN
)
{
   EORB_TRY
   {
      printf ("Binding 10 names into a naming context\n");

      CORBA::Object_var obj = new CORBA::Object;
      CosNaming::NamingContext_var ctx;
      CosNaming::Binding_var binding;
      CosNaming::BindingList_var blist;
      CosNaming::BindingIterator_var biter;
      CosNaming::NameComponent comp;
```

```
CosNaming::Name name (1);
name.length (1);
unsigned long i = 0;
const char * type;
char* id;

/* Bind 10 objects into root context, 5 contexts, 5 objects */

for (i = 0; i < 10; i++)
{
   char idbuf[32] = "";

   if (i < 5)
   {
      sprintf (idbuf, "context binding #%ld", i);
   }
   else
   {
      sprintf (idbuf, "object binding #%ld", i);
   }

   comp.id = (const char*) idbuf;
   name[0] = comp;

   if (i < 5)  /* bind context */
   {
      ctx = root_ctx->bind_new_context (name EORB_ENV_VARN);
      EORB_CHECK_ENV;

      printf ("Bind new context to name: %s\n", idbuf);
   }
   else  /* bind object */
   {
      root_ctx->bind (name, obj.in () EORB_ENV_VARN);
      EORB_CHECK_ENV;

      printf ("Bind object to name: %s\n", idbuf);
   }
}

/* Recover 5 Bindings with list */

root_ctx->list (5, blist, biter EORB_ENV_VARN);
EORB_CHECK_ENV;

for (i = 0; i < blist->length (); i++)
{
   type = ((*blist)[i].binding_type == CosNaming::ncontext)
             ? "Context" : "Object ";
   id = ((*blist)[i].binding_name[0].id);

   printf ("List - %s : %s\n", type, id);
}

/* Recover 2  bindings with BindingIterator_next_n */

biter->next_n (2, blist EORB_ENV_VARN);
EORB_CHECK_ENV;

for (i = 0; i < blist->length (); i++)
{
```

```
         type = ((*blist)[i].binding_type == CosNaming::ncontext)
                     ? "Context" : "Object";
         id = ((*blist)[i].binding_name[0].id);

         printf ("BindingIterator_next_n - %s : %s\n", type, id);
      }

      /* Recover 3 bindings with BindingIterator_next_one */
      while (biter->next_one (binding EORB_ENV_VARN))
      {
         type = (binding->binding_type == CosNaming::ncontext)
                     ? "Context" : "Object ";
         id = (binding->binding_name[0].id);

         printf ("BindingIterator_next_one - %s : %s\n", type, id);
      }

      /* Resolve bindings and unbind objects and contexts, destroy
contexts */

      for (i = 0; i < 10; i++)
      {
         char idbuf[32] = "";

         if (i < 5)
         {
            sprintf (idbuf, "context binding #%ld", i);
         }
         else
         {
            sprintf (idbuf, "object binding #%ld", i);
         }

         comp.id = (const char*) idbuf;
         name[0] = comp;

         obj = root_ctx->resolve (name EORB_ENV_VARN);
         EORB_CHECK_ENV;

         printf ("Resolve binding : %s\n", idbuf);

         root_ctx->unbind (name EORB_ENV_VARN);
         EORB_CHECK_ENV;

         printf ("  Unbind from name : %s\n", idbuf);

         if (i < 5)
         {
            ctx = CosNaming::NamingContext::_narrow (obj.in ()
EORB_ENV_VARN);
            EORB_CHECK_ENV;

            printf ("  Narrow object to context\n");

            ctx->destroy (EORB_ENV_VAR1);
            EORB_CHECK_ENV;

            printf ("  Destroy context\n");
         }
      }
```

```
      biter->destroy (EORB_ENV_VAR1);
      EORB_CHECK_ENV;

   }
   EORB_CATCH (CORBA::Exception, exc)
   {
      printf ("Exception: %s\n", exc._rep_id ());
   }
   EORB_END_TRY
}


EORB_MAIN (client)
{
   EORB_DECLARE_ENV;

   EORB::Plugin::IIOP::add ();

   printf ("Naming client starting\n");

   EORB_TRY
   {
      CORBA::ORB_var orb;
      EORBTest::name_var example;
      CORBA::Object_var obj;
      CosNaming::NamingContext_var root;

      orb = CORBA::ORB_init (argc, argv EORB_ENV_VARN);
      EORB_CHECK_ENV;

      obj = orb->resolve_initial_references ("NameService"
EORB_ENV_VARN);
      EORB_CHECK_ENV;
      root = CosNaming::NamingContext::_narrow (obj EORB_ENV_VARN);
      EORB_CHECK_ENV;

     obj = orb->resolve_initial_references ("server" EORB_ENV_VARN);
      EORB_CHECK_ENV;
      example = EORBTest::name::_narrow (obj EORB_ENV_VARN);
      EORB_CHECK_ENV;

      full_naming_examples (root EORB_ENV_VARN);
      EORB_CHECK_ENV;

      root->destroy (EORB_ENV_VAR1);
      EORB_CHECK_ENV;

      example->shutdown (EORB_ENV_VAR1);
      EORB_CHECK_ENV;

      // Clean up

      orb->destroy (EORB_ENV_VAR1);
      EORB_CHECK_ENV;
   }
   EORB_CATCH (CORBA::Exception, exc)
   {
      printf ("Exception: %s\n", exc._rep_id ());
   }
```

```
    EORB_END_TRY

    printf ("Naming client complete\n");

    return 0;
}
```

The example client first obtains the root context of the naming service, resolves the initial reference to the example server, and binds in five new contexts and five objects. It then uses different methods to resolve the bindings and retrieve the objects, finally unbinding and destroying them.

CHAPTER

5 Supplemental Information

5.1 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.

Appendices

Appendix

A Examples' Source Code

Code examples shown in this Guide are taken from an example application which is supplied with the product. The example application is supplied as source files in *`examples/cpp/services/naming`* below the installation directory.

This release of the Spectra ORB Naming Service does ***not*** support the *`kind`* property of the *`CosNaming::NameComponent`*. Although the supplied examples use the `kind` property, it is nonetheless ignored by the Service when binding or resolving names.

This supplied code is for demonstration and educational purposes ***only***: users should ascertain for themselves the code's suitability and usability. It is expected that users will need to make changes to the source code to suit their particular platform and configuration.

The examples do not have all of the error or exception trapping code normally used, for the sake of clarity and brevity. Exceptions and errors must, naturally, be caught and properly handled in a production system.

Index

Index

O

R

S

U