

Orber Application

version 3.2

Typeset in L^AT_EX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

1	Orber User's Guide	1
1.1	The Orber Application	1
1.1.1	Content Overview	1
1.1.2	Brief Description of the User's Guide	1
1.2	Introduction to Orber	3
1.2.1	Overview	3
1.3	The Orber Application	5
1.3.1	ORB Kernel and IIOP	5
1.3.2	The Object Request Broker (ORB)	6
1.3.3	Internet Inter-Object Protocol (IIOP)	7
1.4	Interface Repository	8
1.4.1	Interface Repository(IFR)	8
1.5	Installing Orber	8
1.5.1	Installation Process	8
1.5.2	Configuration	11
1.6	OMG IDL Mapping	13
1.6.1	OMG IDL Mapping - Overview	13
1.6.2	OMG IDL Mapping Elements	13
1.6.3	Basic OMG IDL Types	14
1.6.4	Constructed OMG IDL Types	14
1.6.5	References to Constants	15
1.6.6	References to Objects Defined in OMG IDL	15
1.6.7	Invocations of Operations	16
1.6.8	Exceptions	17
1.6.9	Access to Attributes	17
1.6.10	Typecode, Identity and Name Access Functions	17
1.6.11	Type Code Representation	18
1.6.12	Scoped Names	19
1.7	CosNaming Service	22
1.7.1	Overview of the CosNaming Service	22

1.7.2	The Basic Use-cases of the Naming Service	24
1.7.3	Interoperable Naming Service	26
1.8	How to use security in Orber	29
1.8.1	Security in Orber	29
1.9	Orber Examples	31
1.9.1	A Tutorial on How to Create a SimpleService	31
1.9.2	A Tutorial on How to Start Orber as Lightweight	39
1.10	Orber Stubs/Skeletons	40
1.10.1	Orber Stubs and Skeletons Description	40
1.11	CORBA System and User Defined Exceptions	44
1.11.1	System Exceptions	44
1.11.2	User Defined Exceptions	47
1.11.3	Throwing Exceptions	47
1.11.4	Catching Exceptions	47
1.12	Orber Interceptors	48
1.12.1	Using Interceptors	48
1.12.2	Interceptor Example	50
1.13	OrberWeb	53
1.13.1	Using OrberWeb	53
1.13.2	Starting OrberWeb	62
1.14	Debugging	62
1.14.1	Tools and FAQ	62
1.15	Orber Release Notes	64
1.15.1	Orber 3.2.12, Release Notes	64
1.15.2	Orber 3.2.11, Release Notes	65
1.15.3	Orber 3.2.10, Release Notes	66
1.15.4	Orber 3.2.9, Release Notes	67
1.15.5	Orber 3.2.8, Release Notes	67
1.15.6	Orber 3.2.7, Release Notes	68
1.15.7	Orber 3.2.6, Release Notes	68
1.15.8	Orber 3.2.5, Release Notes	69
1.15.9	Orber 3.2.4, Release Notes	70
1.15.10	Orber 3.2.3, Release Notes	70
1.15.11	Orber 3.2.2, Release Notes	71
1.15.12	Orber 3.2.1, Release Notes	71
1.15.13	Orber 3.2, Release Notes	72
1.15.14	Orber 3.1.8, Release Notes	72
1.15.15	Orber 3.1.7, Release Notes	73
1.15.16	Orber 3.1.6, Release Notes	74
1.15.17	Orber 3.1.5, Release Notes	74

1.15.18	Orber 3.1.4, Release Notes	75
1.15.19	Orber 3.1.3, Release Notes	75
1.15.20	Orber 3.1.2, Release Notes	76
1.15.21	Orber 3.1.1, Release Notes	77
1.15.22	Orber 3.1, Release Notes	77
1.15.23	Orber 3.0.1, Release Notes	78
1.15.24	Orber 3.0, Release Notes	78
2	Orber Reference Manual	81
2.1	CosNaming	94
2.2	CosNaming_BindingIterator	97
2.3	CosNaming_NamingContext	99
2.4	CosNaming_NamingContextExt	102
2.5	Module_Interface	104
2.6	any	110
2.7	corba	112
2.8	corba_object	118
2.9	interceptors	120
2.10	lname	125
2.11	lname_component	127
2.12	orber	129
2.13	orber_ifr	136
2.14	orber_tc	150
	List of Figures	155
	List of Tables	157
	Glossary	159

Chapter 1

Orber User's Guide

The *Orber* application is an Erlang implementation of a CORBA Object Request Broker.

1.1 The Orber Application

1.1.1 Content Overview

The Orber documentation is divided into three sections:

- PART ONE - The User's Guide
Description of the Orber Application including IDL-to-Erlang language mapping, services and a small tutorial demonstrating the development of a simple service.
- PART TWO - Release Notes
A concise history of Orber.
- PART THREE - The Reference Manual
A quick reference guide, including a brief description, to all the functions available in Orber.

1.1.2 Brief Description of the User's Guide

The User's Guide contains the following parts:

- ORB kernel and IIOP support
- Interface Repository
- IDL to Erlang mapping
- CosNaming Service
- Resolving initial reference from Java or C++
- Tutorial - creating a simple service
- CORBA Exceptions
- Interceptors
- OrberWeb
- Debugging

ORB Kernel and IIOP Support

The ORB kernel which has IIOP support will allow the creation of persistent server objects in Erlang. These objects can also be accessed via Erlang and Java environments. For the moment a Java enabled ORB is needed to generate Java from IDL to use Java server objects (this has been tested using OrbixWeb).

Interface Repository

The IFR is an interface repository used for some type-checking when coding/decoding IIOP. The IFR is capable of storing all interfaces and declarations of OMG IDL.

IDL to Erlang Mapping

The OMG IDL mapping for Erlang, which is necessary to access the functionality of Orber, is described. The mapping structure is included as the basic and the constructed OMG IDL types references, invocations and Erlang characteristics. An example is also provided.

CosNaming Service

Orber contains a CosNaming compliant service.

Resolving Initial References from Java or C++

A couple of classes are added to Orber to simplify initial reference access from Java or C++.

Resolving initial reference from Java

A class with only one method which returns an *IOR* on the external string format to the INIT object (see "Interoperable Naming Service" specification).

Resolving initial reference from C++

A class (and header file) with only one method which returns an *IOR* on the external string format to the INIT object (see "Interoperable Naming Service" specification).

Orber Stub/Skeleton

An example which describes the API and behavior of Orber stubs and skeletons.

CORBA Exceptions

A listing of all system exceptions supported by Orber and how one should handle them. This chapter also describe how to generate user defined exceptions.

Interceptors

Describes how to implement and activate interceptors.

OrberWeb

Offers the possibility to administrate and supervise Orber via a GUI.

Debugging

Describes how to use different tools when debugging and/or developing new applications using Orber. Also includes a FAQ, which deal with the most common mistakes when using Orber.

1.2 Introduction to Orber

1.2.1 Overview

The Orber application is a CORBA compliant Object Request Brokers (ORB), which provides CORBA functionality in an Erlang environment. Essentially, the ORB channels communication or transactions between nodes in a heterogeneous environment.

CORBA (Common Object Request Broker Architecture) provides an interface definition language allowing efficient system integration and also supplies standard specifications for some services.

The Orber application contains the following parts:

- ORB kernel and IIOP support
- Interface Repository
- Interface Definition Language Mapping for Erlang
- CosNaming Service

Benefits

Orber provides CORBA functionality in an Erlang environment that enables:

- *Platform interoperability and transparency*
Orber enables communication between OTP applications or Erlang environment applications and other platforms; for example, Windows NT, Solaris etc, allowing platform transparency. This is especially helpful in situations where there are many users with different platforms. For example, booking airline tickets would require the airline database and hundreds of travel agents (who may not have the same platform) to book seats on flights.
- *Application level interoperability and transparency*
As Orber is a CORBA compliant application, its purpose is to provide interoperability and transparency on the application level. Orber simplifies the distributed system software by defining the environment as objects, which in effect, views everything as identical regardless of programming languages.
Previously, time-consuming programming was required to facilitate communication between different languages. However, with CORBA compliant Orber the Application Programmer is relieved of this task. This makes communication on an application level relatively transparent to the user.

Purpose and Dependencies

The system architecture and OTP dependencies of Orber are illustrated in figure 1 below:

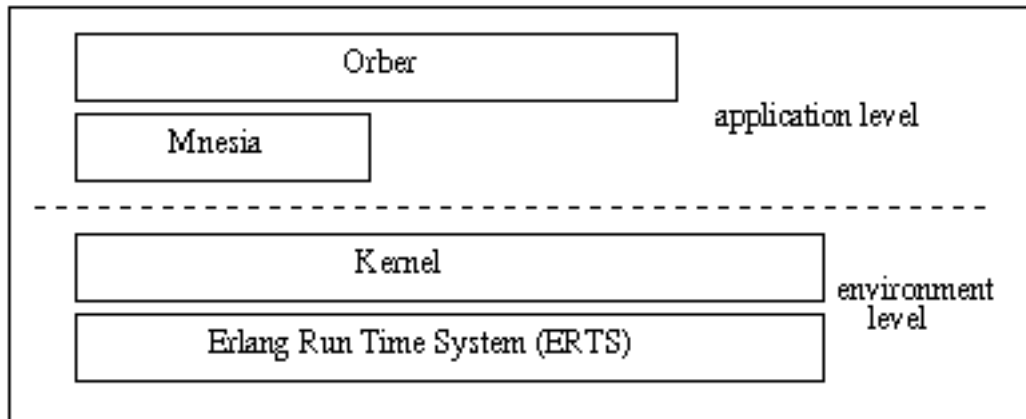


Figure 1.1: Figure 1: Orber Dependencies and Structure.

Orber is dependent on Mnesia (see the Mnesia documentation) - an Erlang database management application used to store object information.

Note:

Although Orber does not have a run-time application dependency to IC (an *IDL* compiler for Erlang), it is necessary when building services and applications. See the IC documentation for further details.

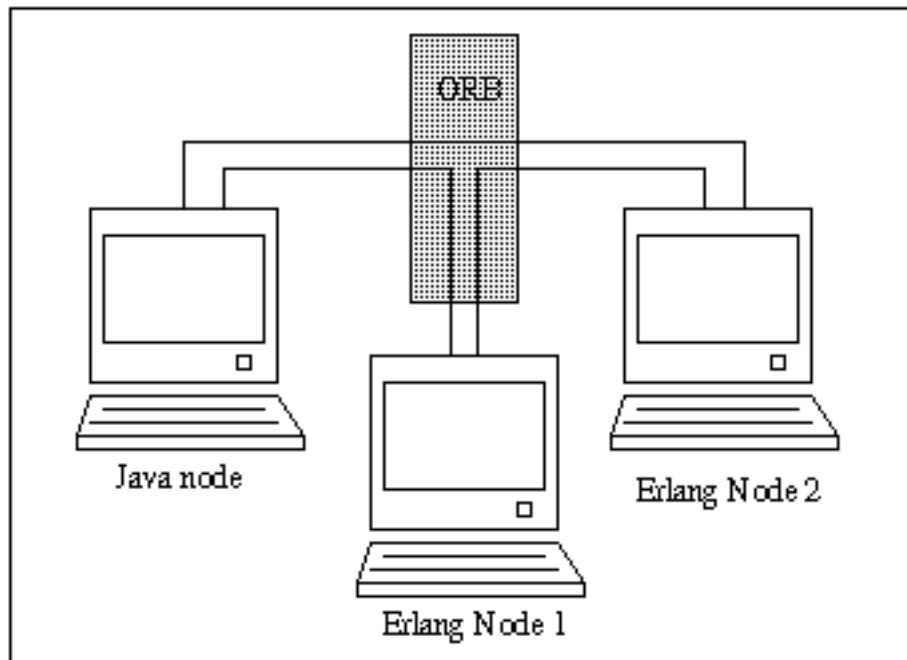


Figure 1.2: Figure 2: ORB interface between Java and Erlang Environment Nodes.

This simplified illustration in figure 2 demonstrates how Orber can facilitate communication in a heterogeneous environment. The Erlang Nodes running OTP and the other Node running applications written in Java can communicate via an *ORB* (Object Request Broker). Using Orber means that CORBA functions can be used to achieve this communication.

For example, if one of the above nodes requests an object, it does not need to know if that object is located on the same, or different, Erlang or Java nodes. The ORB will channel the information creating platform and application transparency for the user.

Prerequisites

To fully understand the concepts presented in the documentation, it is recommended that the user is familiar with distributed programming and CORBA (Common Object Request Broker Architecture).

Recommended reading includes *Open Telecom Platform Documentation Set* and *Concurrent Programming in Erlang*.

1.3 The Orber Application

1.3.1 ORB Kernel and IIOP

This chapter gives a brief overview of the ORB and its relation to objects in a distributed environment and the usage of Domains in Orber. Also Internet-Inter ORB Protocol (*IIOP*) is discussed and how this protocol facilitates communication between ORBs to allow the accessory of persistent server objects in Erlang.

1.3.2 The Object Request Broker (ORB)

An ORB kernel can be best described as the middle-ware, which creates relationships between clients and servers, but is defined by its interfaces. This allows transparency for the user, as they do not have to be aware of where the requested object is located. Thus, the programmer can work with any other platform provided that an IDL mapping and interfaces exist.

The IDL mapping which is described in a later chapter is the translator between other platforms, and languages. However, it is the ORB, which provides objects with a structure by which they can communicate with other objects.

ORBs intercept and direct messages from one object, pass this message using IIOP to another ORB, which then directs the message to the indicated object.

An ORB is the base on which interfaces, communication stubs and mapping can be built to enable communication between objects. Orber uses *domains* to group objects of different nodes

How the ORB provides communication is shown very simply in figure 1 below:

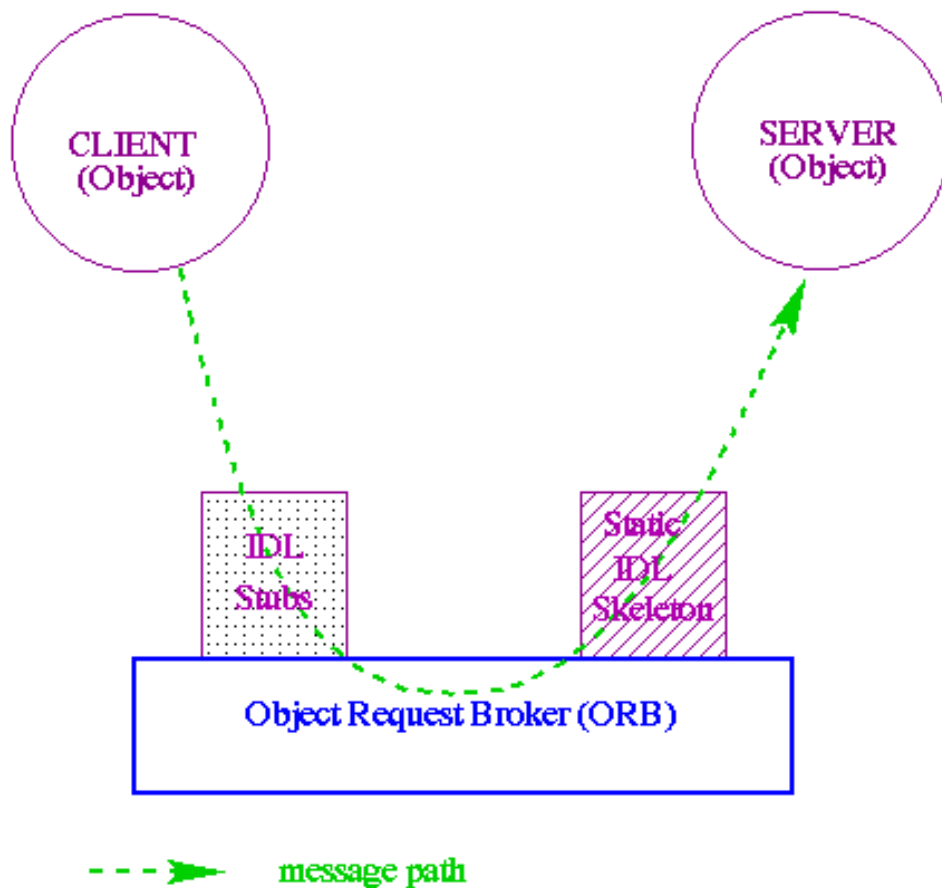


Figure 1.3: Figure 1: How the Object Request Broker works.

The domain in Orber gives an extra aspect to the distributed object environment as each domain has one ORB, but it is distributed over a number of object in different nodes. The domain binds objects on

nodes more closely than distributed objects in different domains. The advantage of a domain is that a faster communication exists between nodes and objects of the same domain. An internal communication protocol (other than IIOP) allows a more efficient communication between these objects.

Note:

Unlike objects, domains can only have one name so that no communication ambiguities exist between domains.

1.3.3 Internet Inter-Object Protocol (IIOP)

IIOP is a communication protocol developed by the OMG to facilitate communication in a distributed object-oriented environment.

Figure 2 below demonstrates how IIOP works between objects:

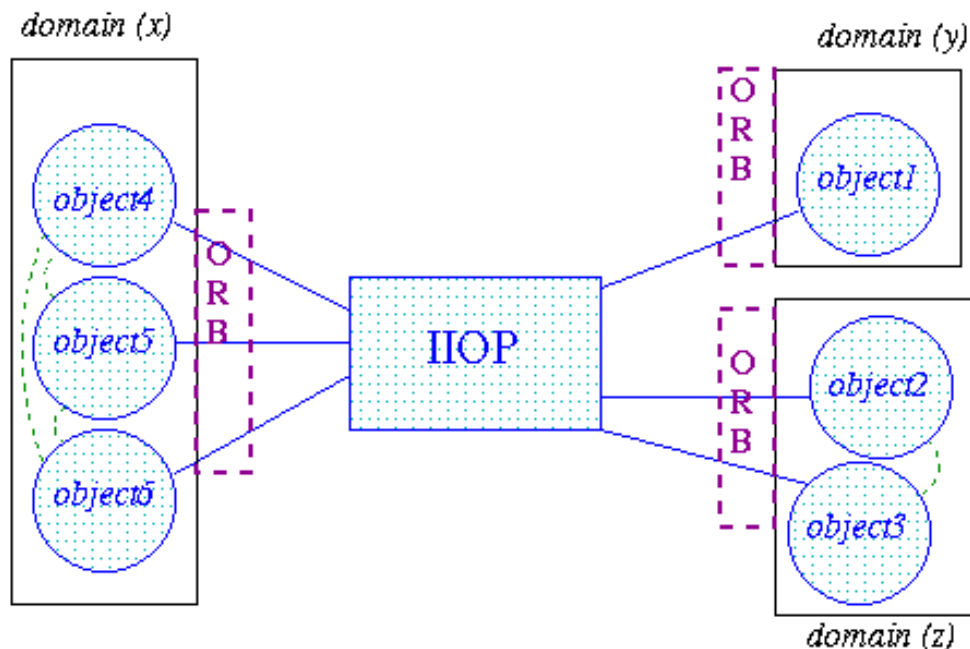


Figure 1.4: Figure 2: IIOP communication between domains and objects.

Note:

Within the Orber domains the objects communicate without using the IIOP. However, the user is unaware of the difference in protocols, as this difference is not visible.

1.4 Interface Repository

1.4.1 Interface Repository(IFR)

The IFR is an interface repository built on the Mnesia application. Orber uses the IFR for some type-checking when coding/decoding IIOP. The IFR is capable of storing all interfaces and declarations of OMG IDL.

The interface repository is mainly used for dynamical interfaces, and as none are currently supported this function is only really used for retrieving information about interfaces.

Functions relating to the manipulation of the IFR including, initialization of the IFR, as well as, locating, creating and destroying initial references are detailed further in the Manual Pages.

1.5 Installing Orber

1.5.1 Installation Process

This chapter describes how to install Orber in an Erlang Environment.

Preparation

To begin with, you must decide if you want to run Orber as a:

- *Single node (non-distributed)* - all communication with other Orber instances and ORB's supplied by other vendors use the OMG GIOP protocol.
- *Multi node (distributed)* - all Orber nodes, within the same domain, communicate via the Erlang distribution protocol. For all other Orber instances, i.e. not part of the same domain, and ORB's supplied by other vendors, the OMG GIOP protocol is used.

Which approach to use is highly implementation specific, but a few things you should consider:

- All nodes within an Orber domain should have the same security level.
- If the capacity is greater than load (volume of traffic) a single-node Orber might be a good solution.
- In some cases the distributed system architecture requires a single-node *Orber installation*.
- A multi-node Orber makes it possible to load balance and create a more fault tolerant system. The Objects can also have a uniform view if you use distributed Mnesia tables.
- Since the GIOP protocol creates a larger overhead than the Erlang distribution protocol, the performance will be better when communicating with Objects within the same Orber domain compared with inter ORB communication (GIOP).

You also have to decide if you want Orber to store internal data using `disc_copies` and/or `ram_copies`. Which storage type you should depends if/how you intend to use Mnesia in your application. If you intend to use `disc_copies` you must start with creating a Mnesia schema, which contain information about the location of the Erlang nodes where Orber is planned to be run. For more background information, see the Mnesia documentation.

In some cases it is absolutely necessary to change the default configuration of Orber. For example, if two Orber-ORB's shall be able to communicate via GIOP, they must have a unique domain. Consult the configuration settings [page 11] section. If you encounter any problems; see the chapter about *Debugging* in this User's Guide.

Jump Start Orber

The easiest way to start Orber is to use `orber:jump_start(Port)`, which start a single-node ORB with (most likely) a unique domain (i.e. "IP-number:Port"). This function may only be used during development and testing. For any other situation, install and start Orber as described in the following sections. The listen port, i.e. `iiop_port` configuration parameter, is set to the supplied Port.

Warning:

How Orber is configured when using `orber:jump_start(Port)` may change at any time without warning. Hence, this operation must not be used in systems delivered to a customer.

Install Single Node Orber

Since a single node Orber communicate via the OMG GIOP protocol it is not necessary to start the Erlang distribution (i.e. using `-name/-sname`).

If we use `ram_copies` there is no need for creating a disc based schema. Simply use:

```
erl> mnesia:start().
erl> corba:orb_init([domain, "MyRAMSingleNodeORB"]).
erl> orber:install([node()], [{ifr_storage_type, ram_copies}]).
erl> orber:start().
```

If you installation requires `disc_copies` you must begin with creating a Mnesia schema. Otherwise, the installation is similar to a RAM installation:

```
erl> mnesia:create_schema([node()]).
erl> mnesia:start().
erl> corba:orb_init([domain, "MyDiscSingleNodeORB"]).
erl> orber:install([node()], [{ifr_storage_type, disc_copies},
                             {nameservice_storage_type, disc_copies}]).
erl> orber:start().
```

You can still choose to store the IFR data as `ram_copies`, but then the data must be re-installed (i.e. invoke `orber:install/2`) if the node is restarted. Hence, since the IFR data is rather static you should use `disc_copies`. For more information see the `orber` section in the reference manual.

If you do not need to change Orber's configuration you can skip `orb_init/1` [page 112]. But, you *should* at least set the IIOP timeout parameters.

Note:

When starting Orber as lightweight, `mnesia` and `orber:install/*` are not required. You must, however, use the configuration [page 11] parameter `lightweight`.

Install RAM Based Multi Node Orber

Within a domain Orber uses the Erlang distribution protocol. Hence, you *must* start it first by, for example, using:

```
hostA> erl -sname nodeA
```

In this example, we assume that we want to use two nodes; nodeA and nodeB. Since Mnesia must know which other nodes should be a part of the distribution we either need to add the Mnesia configuration parameter `extra_db_nodes` or use `mnesia:change_config/2`. To begin with, Mnesia must be started on all nodes before we can install Orber:

```
nodeA@hostA> mnesia:start().
nodeA@hostA> mnesia:change_config(extra_db_nodes,
                                   [nodeA@hostA, nodeB@hostB]).
```

After that the above have been repeated on nodeB we must first make sure that both nodes will use the same domain name, then we can install Orber:

```
nodeA@hostA> corba:orb_init([domain, "MyRAMMultiNodeORB"]).
nodeA@hostA> orber:install([nodeA@hostA, nodeB@hostB],
                           [{ifr_storage_type, ram_copies}]).
nodeA@hostA> orber:start().
```

Note that you can only invoke `orber:install/1/2` on one of the nodes. Now we can start Orber on the other node:

```
nodeB@hostB> corba:orb_init([domain, "MyRAMMultiNodeORB"]).
nodeB@hostB> orber:start().
```

Install Disc Based Multi Node Orber

As for RAM based multi-node Orber installations, the Erlang distribution must be started (e.g. `erl -sname nodeA`). The major difference is that when it is disc based a Mnesia schema must be created:

```
nodeA@hostA> mnesia:create_schema([nodeA@hostA, nodeB@hostB]).
nodeA@hostA> mnesia:start().
```

In this example, we assume that we want to use two nodes; nodeA and nodeB. Since it is not possible to create a schema on more than one node. Hence, all we have to do is to start Mnesia (i.e. invoke `mnesia:start()`) on nodeB.

After Mnesia have been started on all nodes, you must confirm that all nodes have the same domain name, then Orber is ready to be installed:

```
nodeA@hostA> corba:orb_init([domain, "MyDiscMultiNodeORB"]).
nodeA@hostA> orber:install([nodeA@hostA, nodeB@hostB],
                           [{ifr_storage_type, disc_copies}]).
nodeA@hostA> orber:start().
```

Note that you can only invoke `orber:install/1/2` on one of the nodes. Now we can start Orber on the other node:

```
nodeB@hostB> corba:orb_init([domain, "MyDiscMultiNodeORB"]).
nodeB@hostB> orber:start().
```


Running Java clients against Orber.

If you intend to run Java clients and your Java ORB does not support the Interoperable Naming Service (INS), a specific

```
<OTP_INSTALLPATH>/lib/orber-<current-version>/priv
```

must be added to your *CLASSPATH* variable to allow Orber support for the initial references. For more information about INS, see the Name Service chapter in this User's Guide.

1.5.2 Configuration

The following configuration parameters exist:

- *domain* - default is "ORBER". The value is a string. Since Orber domains must have unique names, communication will fail if two domains have the same name. The domain name *MAY NOT* contain ^G (i.e. \007).
- *iiop_port* - default 4001. The value is an integer.
Note: On a UNIX system it is preferable to have a IIOP port higher than 1023, since it is not recommended to run Erlang as a root user.
- *iiop_out_ports* - default 0 (i.e. use any available port). If set, Orber will only use the local ports within the interval when trying to connect to another ORB (Orber acts as a client ORB). If all ports are in use communication will fail. Hence, it is *absolutely necessary* to set *iiop_connection_timeout* as well. Otherwise, connections no longer in use will block further communication. If one use, for example, `erl -orber iiop_out_ports "{5000,5020}"`, Orber will only use port 5000 to 5020 when connecting. SSL do not support this feature. Hence, one *MAY NOT* combine these settings.
- *bootstrap_port* - It is used for fetching initial service references and has the IIOP port as the default setting. The value is an integer.
- *ip_address* - default is all interfaces. This option is used if orber only should listen on a specific ip interface on a multi-interface host. The value is the ip address as a string or a tuple of four integers.
- *objectkeys_gc.time* - default is infinity. This option should be set if objects are started using the option `{persistent, true}`. The value is `integer()` seconds.
- *giop_version* - default is IIOP 1.1. It is possible to IIOP Versions 1.0 or 1.2 as well, e.g., `erl -orber giop_version "{1,2}"`
- *iiop_setup_connection_timeout* - default is infinity. The value is an integer (seconds) or the atom infinity. This option is only valid for client-side connections. If this option is set, attempts to connect to other ORB's will timeout after the given time limit. Note, if the time limit is large the TCP protocol may timeout before the supplied value.
- *iiop_connection_timeout* - default is infinity. The value is an integer (timeout in seconds between 0 and 1000000) or the atom infinity. This option is only valid for client object connections, i.e., will have no effect on server connections. Setting this option will cause client connections to be terminated, if and only if, there are no pending requests. If there are a client still waiting for a reply, Orber will try again after the given seconds have passed. The main purpose for this option is to reduce the number of open connections; it is, for example, not necessary to keep a connection, only used once a day, open at all time.

- *iiop_timeout* - default is infinity. The value is an integer (timeout in seconds between 0 and 1000000) or the atom infinity. This option is only valid on the client side. Setting this option, cause all intra-ORB requests to timeout and raise the `COMM.FAILURE` system exception if no replies are delivered within the given time limit.
- *interceptors* - if one set this parameter, e.g., `erl -orber interceptors "{native, ['myInterceptor']}"`, Orber will use the supplied interceptor(s) for all inter-ORB communication. 'myInterceptor' is the module name of the interceptor. For more information, see the interceptor chapter in the User's Guide and the Reference Manual.
- *lightweight* - default is false. This option must be set if Orber is supposed to be started as lightweight. The value is a list of RemoteModifiers, equal to the `orber:resolve_initial_references_remote/2` argument. The list must contain Orber nodes addresses, to which we have access and are not started as lightweight.
- *orbInitRef* - default is undefined. Setting this option, e.g., `erl -orber orbInitRef ["NameService=corbaloc::host.com/NameService\""],` will alter the location from where `corba:resolve_initial_references(Key)` tries to find an object matching the given Key. The keys will also appear when invoking `corba:list_initial_services()`. This variable overrides `orbDefaultInitRef`
- *orbDefaultInitRef* - default is undefined. If a matching Key for `orbInitRef` is not found, and this variable is set, it determines the location from where `orber:resolve_initial_references(Key)` tries to find an object matching the given Key. Usage: `erl -orber orbDefaultInitRef \"corbaloc::host.com\"`
- *orber_debug_level* - default is 0 and the range is 0 to 10. Using level 10 is the most verbose configuration. This option will generate reports, using the `error_logger`, for abnormal situations. It is not recommended to use this option for delivered systems since some of the reports is not to be considered as errors. The main purpose is to assist during development.

IC supply the compile option `ic:gen(IdlFile, [{timeout, "module::interface"}])`, which allow the user to add an extra timeout parameter, e.g., `module_interface(ObjRef, Timeout, ... Arguments ...)`, instead of `module_interface(ObjRef, ... Arguments ...)`. If a stub is compiled with the timeout option, the extra `Timeout` argument will override the configuration parameter `iiop_timeout`. It is, however, not possible to use infinity to override the `Timeout` parameter. The `Timeout` option is also valid for objects which resides within the same *Orber domain*.

The `iiop_setup_connection_timeout`, `iiop_timeout` and `iiop_connection_timeout` variables should be used. The specified values is implementation specific, i.e., WAN or LAN, but they should range from `iiop_setup_connection_timeout` to `iiop_connection_timeout`.

IIOP communication only occurs between different Orber domains and therefore, if IIOP communication is required between two Orber domains, their domain names must be set to different values.

The following options are the possible configurations when using Orber with secure IIOP. Orber currently only supports security with the help of SSL and not SECIOp. To get more information about the SSL read the SSL application manual. The security chapter later in this manual describes how to get security in Orber and how the options are used.

- *secure* - default is no security. The values are currently just the atoms *ssl* and *no*.
- *ssl_server_certfile* - The value is a file path to a server side certificate.
- *ssl_server_verify* - The value is an integer less or equal than two.
- *ssl_server_depth* - The value is an integer.
- *ssl_client_certfile* - The value is a file path to a client side certificate.
- *ssl_client_verify* - The value is an integer less or equal than two.

- *ssl_client_depth* - The value is an integer.

To change these settings in the configuration file, the `-config` flag must be added to the `erl` command. See the Reference Manual *config(4)* for further information. The values can also be sent separately as options to the Erlang node when it is started, see the Reference Manual *erl(1)* for further information.

1.6 OMG IDL Mapping

1.6.1 OMG IDL Mapping - Overview

The purpose of OMG IDL mapping is to act as translator between platforms and languages.

CORBA is independent of the programming language used to construct clients or implementations. In order to use the ORB, it is necessary for programmers to know how to access ORB functionality from their programming languages. It translates different IDL constructs to a specific programming language. This chapter describes the mapping of OMG IDL constructs to the Erlang programming language.

1.6.2 OMG IDL Mapping Elements

A complete language mapping will allow the programmer to have access to all ORB functionality in a way that is convenient for a specified programming language.

All mapping must define the following elements:

- All OMG IDL basic and constructed types
- References to constants defined in OMG IDL
- References to objects defined in OMG IDL
- Invocations of operations, including passing of parameters and receiving of results
- Exceptions, including what happens when an operation raises an exception and how the exception parameters are accessed
- Access to attributes
- Signatures for operations defined by the ORB, such as dynamic invocation interface, the object adapters etc.
- Scopes; OMG IDL has several levels of scopes, which are mapped to Erlang's two scopes. The scopes, and the files they produce, are described.

ReservedCcompiler Names

The use of some names is strongly discouraged due to ambiguities. However, the use of some names is prohibited when using the Erlang mapping, as they are strictly reserved for IC.

IC reserves all identifiers starting with `0E_` and `oe_` for internal use.

Note also, that an identifier in IDL can contain alphabetic, digits and underscore characters, but the first character *must* be alphabetic.

Using underscores in IDL names can lead to ambiguities due to the name mapping described above. It is advisable to avoid the use of underscores in identifiers.

Refer to the IC documentation for further details.

1.6.3 Basic OMG IDL Types

The OMG IDL mapping is strongly typed and (even if you have a good knowledge of CORBA types), it is essential to read carefully the following mapping to Erlang types.

The mapping of basic types is straightforward. Note that the OMG IDL double type is mapped to an Erlang float which does not support the full double value range.

OMG IDL type	Erlang type	Note
float	Erlang float	
double	Erlang float	value range not supported
short	Erlang integer	
unsigned short	Erlang integer	
long	Erlang integer	
long long	Erlang integer	
unsigned long	Erlang integer	
unsigned long long	Erlang integer	
char	Erlang integer	
wchar	Erlang integer	
boolean	Erlang atoms true or false	
octet	Erlang integer	
any	Erlang record #any{typecode, value}	
long double	Not supported	
Object	Orber object reference	
void	Erlang atom ok	

Table 1.1: OMG IDL basic types

The any value is written as a record with the field typecode which contains the *Type Code* representation, see also the Type Code table [page 18], and the value field itself.

Functions with return type void will return the atom ok.

1.6.4 Constructed OMG IDL Types

Constructed types all have native mappings as shown in the table below.

string	Erlang string
wstring	Erlang list of Integers
struct	Erlang record
union	Erlang record
enum	Erlang atom
sequence	Erlang list
array	Erlang tuple

Table 1.2: OMG IDL constructed types

Below are examples of values of constructed types.

Type	IDL code	Erlang code
<i>string</i>	<code>typedef string S; void op(in S a);</code>	<code>ok = op(Obj, "Hello World"),</code>
<i>struct</i>	<code>struct S {long a; short b;}; void op(in S a);</code>	<code>ok = op(Obj, #'S'{a=300, b=127}),</code>
<i>union</i>	<code>union S switch(long) { case 1: long a;}; void op(in S a);</code>	<code>ok = op(Obj, #'S'{label=1, value=66}),</code>
<i>enum</i>	<code>enum S {one, two}; void op(in S a);</code>	<code>ok = op(Obj, one),</code>
<i>sequence</i>	<code>typedef sequence <long, 3> S; void op(in S a);</code>	<code>ok = op(Obj, [1, 2, 3]),</code>
<i>array</i>	<code>typedef string S[2]; void op(in S a);</code>	<code>ok = op(Obj, {"one", "two"}),</code>

Table 1.3: Typical values

Warning:

Every field in, for example, a struct must be initiated. Otherwise it will be set to the atom `undefined`, which Orber cannot encode when communicating via IIOP. In the example above, invoking the operation with `#'S'{a=300}` will fail (equal to `#'S'{a=300, b=undefined}`)

1.6.5 References to Constants

Constants are generated as Erlang functions, and are accessed by a single function call. The functions are put in the file corresponding to the scope where they are defined. There is no need for an object to be started to access a constant.

Example:

```
// IDL
module M {
    const long c1 = 99;
};
```

Would result in the following conceptual code:

```
-module('M').
-export([c1/0]).

c1() -> 99.
```

1.6.6 References to Objects Defined in OMG IDL

Objects are accessed by object references. An object reference is an opaque Erlang term created and maintained by the ORB.

Objects are implemented by providing implementations for all operations and attributes of the Object, see operation implementation [page 16].

1.6.7 Invocations of Operations

A function call will invoke an operation. The first parameter of the function should be the object reference and then all `in` and `inout` parameters follow in the same order as specified in the IDL specification. The result will be a return value unless the function has `inout` or `out` parameters specified; in which case, a tuple of the return value, followed by the parameters will be returned.

Example:

```
// IDL
interface i1 {
    long op1(in short a);
    long op2(in char c, inout string s, out long count);
    void op3(out long Id);
};
```

Is used in Erlang as :

```
% Erlang
f() ->
...
Obj = ... %% get object reference
R1 = i1:op1(Obj, 55),
{R2, S, Count} = i1:op2(Obj, $a, "hello"),
{ok, Id} = i1:op3(Obj),
...
```

Note how the `inout` parameter is passed *and* returned. There is no way to use a single occurrence of a variable for this in Erlang. Also note, that you must return `ok`, Orber's representation of the IDL-type `void`, must be returned by `op3`. These operations can be implemented in the call-back module as:

```
op1(State, AShort) ->
    {reply, ALong, State}.

op2(State, AShort, AString) ->
    {reply, {ALong, "MyString", ALong}, State}.

op3(State) ->
    {reply, {ok, AId}, State}.
```

The operations may require more arguments (depends on IC options used). For more information, see [Stubs/Skeletons \[page 40\]](#) and [Module.Interface \[page 104\]](#).

Operation Implementation

A standard Erlang `gen_server` behavior is used for object implementation. The `gen_server` state is then used as the object internal state. Implementation of the object function is achieved by implementing its methods and attribute operations. These functions will usually have the internal state as their first parameter, followed by any `in` and `inout` parameters.

Do not confuse the object internal state with its object reference. The object internal state is an Erlang term which has a format defined by the user.

Note:

It is not always the case that the internal state will be the first parameter, as stubs can use their own object reference as the first parameter (see the IC documentation).

The special function `init/1` is called at object start time and is expected to return the tuple `{ok, InitialInternalState}`.

See also the stack example. [page 20]

1.6.8 Exceptions

Exceptions are handled as Erlang catch and throws. Exceptions are translated to messages over an IIOP bridge but converted back to a throw on the receiving side. Object implementations that invoke operations on other objects must be aware of the possibility of a non-local return. This includes invocation of ORB and IFR services. See also the Exceptions [page 44] section.

Exception parameters are mapped as an Erlang record and accessed as such.

An object implementation that raises an exception will use the `corba:raise/1` function, passing the exception record as parameter.

1.6.9 Access to Attributes

Attributes are accessed through their access functions. An attribute implicitly defines the `_get` and `_set` operations. The `_get` operation is defined as a read-only attribute. These operations are handled in the same way as normal operations.

1.6.10 Typecode, Identity and Name Access Functions

As mentioned in a previous section, `struct`, `union` and `exception` types yield to record definitions and access code for that record. For `struct`, `union`, `exception`, `array` and `sequence` types, a special file is generated that holds access functions for `TypeCode`, `Identity` and `Name`. These functions are put in the file corresponding to the scope where they are defined :

- `tc` - returns the type code for the record.
- `id` - returns the identity of the record.
- `name` - returns the name of the record.

For example:

```
// IDL
module m {

    struct s {
        long x;
        long y;
    };

};
```

Would result in the following code on file `m_s.erl`:

```
-module(m_s).

-include("m.hrl").

-export([tc/0,id/0,name/0]).

%% returns type code
tc() -> {tk_struct,"IDL:m/s:1.0","s",[{x,tk_long},{y,tk_long}]}.

%% returns id
id() -> "IDL:m/s:1.0".

%% returns name
name() -> m_s.
```

1.6.11 Type Code Representation

Type Codes are used in any values. The table below corresponds to the table on page 12-11 in the OMG CORBA specification.

Type Code	Example
tk_null	
tk_void	
tk_short	
tk_long	
tk_longlong	
tk_ushort	
tk_ulong	
tk_ulonglong	
tk_float	
tk_double	
tk_boolean	
tk_char	
tk_wchar	

continued ...

... continued

tk_octet	
tk_any	
tk_TypeCode	
tk_Principal	
{tk_objref, IFRId, Name}	{tk_objref, "IDL:M1\I1:1.0", "I1"}
{tk_struct, IFRId, Name, [{ElemName, ElemTC}]}	{tk_struct, "IDL:M1\S1:1.0", "S1", [{ "a", tk_long}, {"b", tk_char}]}
{tk_union, IFRId, Name, DiscrTC, DefaultNr, [{Label, ElemName, ElemTC}]} Note: DefaultNr tells which of tuples in the case list that is default, or -1 if no default	{tk_union, "IDL:U1:1.0", "U1", tk_long, 1, [{1, "a", tk_long}, {default, "b", tk_char}]}
{tk_enum, IFRId, Name, [ElemName]}	{tk_enum, "IDL:E1:1.0", "E1", [{"a1", "a2"}]}
{tk_string, Length}	{tk_string, 5}
{tk_wstring, Length}	{tk_wstring, 7}
{tk_sequence, ElemTC, Length}	{tk_sequence, tk_long, 4}
{tk_array, ElemTC, Length}	{tk_array, tk_char, 9}
{tk_alias, IFRId, Name, TC}	{tk_alias, "IDL:T1:1.0", "T1", tk_short}
{tk_except, IFRId, Name, [{ElemName, ElemTC}]}	{tk_except, "IDL:Exc1:1.0", "Exc1", [{ "a", tk_long}, {"b", {tk_string, 0}}]}

Table 1.4: Type Code tuples

1.6.12 Scoped Names

Various scopes exist in OMG IDL. Modules, interfaces and types define scopes. However, Erlang has only two levels of scope; module and function:

- Function Scope:
used for constants, operations and attributes.
- Erlang Module Scope:
The Erlang module scope handles the remaining OMG IDL scopes.

Syntax Specific Structures for Scoped Names

An Erlang module, corresponding to an IDL global name, is derived by converting occurrences of "::" to underscore, and eliminating the leading "::".

For example, an operation `op1` defined in interface `I1` which is defined in module `M1` would be written in IDL as `M1::I1::op1` and as `'M1_I1':op1` in Erlang, where `op1` is the function name and `'M1_I1'` is the name of the Erlang module.

Files

Several files can be generated for each scope.

- An Erlang source code file (`.erl`) is generated for top level scope as well as the Erlang header file.
- An Erlang header file (`.hrl`) will be generated for each scope. The header file will contain record definitions for all `struct`, `union` and `exception` types in that scope.

- Modules that contain at least one constant definition, will produce Erlang source code files (.erl). That Erlang file will contain constant functions for that scope. Modules that contain no constant definitions are considered empty and no code will be produced for them, but only for their included modules/interfaces.
- Interfaces will produce Erlang source code files (.erl), this code will contain all operation stub code and implementation functions.
- In addition to the scope-related files, an Erlang source file will be generated for each definition of the types `struct`, `union` and `exception` (these are the types that will be represented in Erlang as records). This file will contain special access functions for that record.
- The top level scope will produce two files, one header file (.hrl) and one Erlang source file (.erl). These files are named as the IDL file, prefixed with `oe_`.

Example:

```
// IDL, in the file "spec.idl"
module m {

    struct s {
        long x;
        long y;
    };

    interface i {

        void foo( in s a, out short b );

    };

};
```

This will produce the following files:

- `oe_spec.hrl` and `oe_spec.erl` for the top scope level.
- `m.hrl` for the module `m`.
- `m_i.hrl` and `m_i.erl` for the interface `i`.
- `m_s.erl` for the structure `s` in module `m`.

A Mapping Example

This is a small example of a simple stack. There are two operations on the stack, `push` and `pop`. The example shows all generated files as well as conceptual usage of a stack object.

```
// The source IDL file

interface stack {
    exception overflow {};
    void push(in long val);
    long pop() raises (overflow);
};
```

When this file is compiled it produces four files, two for the top scope and two for the stack interface scope. The generated Erlang code for the stack object server is shown below:

```
-module(stack).
-export([push/2, pop/1]).

init(Env) ->
    stack_impl:init(Env).

%% This is the stub code used by clients
push(THIS, Val) ->
    corba:call(THIS, push, [Val]).

pop(THIS) ->
    corba:call(THIS, pop, []).

%% gen_server handle_calls
handle_call({THIS, push, [Val]}, From, State) ->
    case catch stack_impl:push(State, Val) of
        {'EXCEPTION', E} ->
            {reply, {'EXCEPTION', E}, State};
        {reply, Reply, NewState} ->
            {reply, Reply, NewState}
    end;

handle_call({THIS, pop, []}, From, State) ->
    case catch stack_impl:pop(State) of
        {'EXCEPTION', E} ->
            {reply, {'EXCEPTION', E}, State};
        {reply, Reply, NewState} ->
            {reply, Reply, NewState}
    end.
```

The Erlang code has been simplified but is conceptually correct. The generated `stack` module is the Erlang representation of the stack interface. Note that the variable `THIS` is the object reference and the variable `State` is the internal state of the object.

So far the example only deals with interfaces and call chains. It is now time to implement the stack. The example represents the stack as a simple list. The push operation then is just to add a value on to the front of the list and the pop operation is then to return the head of the list.

In this simple representation the internal state of the object becomes just a list. The initial value for the state is the empty list as shown in the `init/1` function below.

The implementation is put into a file called `stack_impl.erl`.

```
-module(stack_impl).

-include("stack.hrl").

-export([push/2, pop/1, init/1]).

init(_) ->
    {ok, []}.
```

```
push(Stack, Val) ->
    {reply, ok, [Val | Stack]}.
```

```
pop([Val | Stack]) ->
    {reply, Val, Stack};
```

```
pop([]) ->
    corba:raise(#stack_overflow{}).
```

The stack object can be accessed client code. This example shows a typical add function from a calculator class:

```
-module(calc_impl).

-export([add/1]).

add({Stack, Memory}) ->
    Sum = stack:pop(Stack)+stack:pop(Stack),
    stack:push(Stack, Sum),
    {ok, {Stack, Memory}}.
```

Note that the Stack variable above is an object reference and not the internal state of the stack.

1.7 CosNaming Service

1.7.1 Overview of the CosNaming Service

The CosNaming Service is a service developed to help users and programmers identify objects by human readable names rather than by a reference. By binding a name to a naming context (another object), a contextual reference is formed. This is helpful when navigating in the object space. In addition, identifying objects by name allows you to evolve and/or relocate objects without client code modification.

The CosNaming service has some concepts that are important:

- *name binding* - a name to object association.
- *naming context* - is an object that contains a set of name bindings in which each name is unique. Different names can be bound to the same object.
- *to bind a name* - is to create a name binding in a given context.
- *to resolve a name* - is to determine the object associated with the name in a given context.

A name is always resolved in a context, there no absolute names exist. Because a context is like any other object, it can also be bound to a name in a naming context. This will result in a naming graph (a directive graph with nodes and labeled edges). The graph allows more complex names to refer to an object. Given a context, you can use a sequence to reference an object. This sequence is henceforth referred to as *name* and the individual elements in the sequence as *name components*. All but the last name component are bound to naming contexts.

The diagram in figure 1 illustrates how the Naming Service provides a contextual relationship between objects, NamingContexts and NameBindings to create an object locality, as the object itself, has no name.

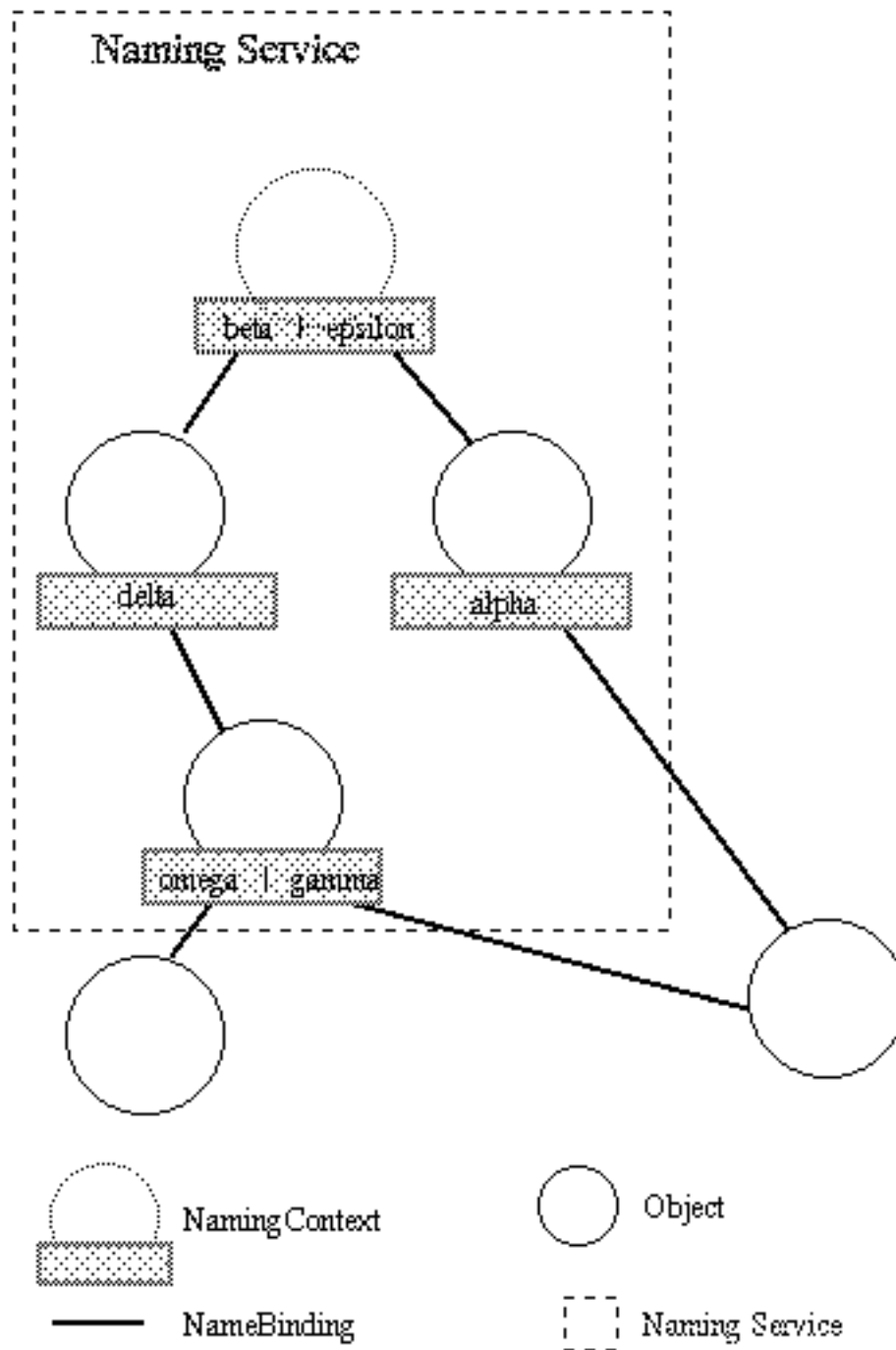


Figure 1.5: Figure 1: Contextual object relationships using the Naming Service.

The naming contexts provide a directory of contextual reference and naming for objects (an object can appear to have more than one name).

In figure 1 the object to the right can either be called `alpha` from one context or `gamma` from another.

The Naming Service has an initial naming context, which is shown in the diagram as the top-most object in the naming graph. It has two names `beta` and `epsilon`, which are bound to other naming contexts. The initial naming context is a well known location used to share a common name space between multiple programs. You can traverse the naming graph until you reach a name, which is bound to an object, which is not a naming context.

We recommend reading *chapter 12, CORBA Fundamentals and Programming*, for detailed information regarding the Naming Service.

1.7.2 The Basic Use-cases of the Naming Service

The basic use-cases of the Naming Service are:

- Fetch initial reference to the naming service.
- Creating a naming context.
- Binding and unbinding names to objects.
- Resolving a name to an object.
- Listing the bindings of a naming context.
- Destroying a naming context.

Fetch Initial Reference to the Naming Service

In order to use the naming service you have to fetch an initial reference to it. This is done with:

```
NS = corba:resolve_initial_reference("NameService").
```

Note:

NS in the other use-cases refers to this initial reference.

Creating a Naming Context

There are two functions for creating a naming context. The first function, which only creates a naming context object is:

```
NC = 'CosNaming_NamingContext':new_context(NS).
```

The other function creates a naming context and binds it to a name in an already existing naming context (the initial context in this example):

```
NC = 'CosNaming_NamingContext':bind_new_context(NS, lname:new(["new"])).
```

Binding and Unbinding Names to Objects

The following steps illustrate how to bind/unbind an object reference to/from a name. For the example below, assume that the NamingContexts in the path are already bound to the name /workgroup/services, and that reference to the services context are in the variable Sc.

1. Use the naming library functions to create a name
`Name = lname:new(["object"]).`
2. Use `CosNaming::NamingContext::bind()` to bind a name to an object
`'CosNaming_NamingContext':bind(Sc, Name, Object).`
3. Use `CosNaming::NamingContext::unbind()` to remove the NameBinding from an object
`'CosNaming_NamingContext':unbind(Sc, Name).`

Note:

Objects can have more than one name, to indicate different paths to the same object.

Resolving a Name to an Object

The following steps show how to retrieve the object reference to the service context above (/workgroup/services).

1. Use the naming library functions to create a name path:
`Name = lname:new(["workgroup", "services"]).`
2. Use `CosNaming::NamingContext::resolve()` to resolve the name to an object
`Sc = 'CosNaming_NamingContext':resolve(NS, Name).`

An alternative is to use:

```
Sc = corba:string_to_object("corbaname:rir:/NameService#workgroup/services/").
```

The corbaname schema is described further in the Interoperable Naming Service section.

Listing the Bindings in a NamingContext

1. Use `CosNaming::NamingContext::list()` to list all the bindings in a context

The following code retrieves and lists up to 10 bindings from a context.

```
{BList, BIterator} = 'CosNaming_NamingContext':list(Sc, 10).

lists:foreach(fun({{Id, Kind}, BindingType}) -> case BindingType of
    nobject ->
        io:format("id: %s, kind: %s, type: object~n", [Id, Kind]);
    _ ->
        io:format("id: %s, kind: %s, type: ncontext~n", [Id, Kind])
end end,
BList).
```

Note:

Normally a *BindingIterator* is helpful in situations where you have a large number of objects in a list, as the programmer then can traverse it more easily. In Erlang it is not needed, because lists are easily handled in the language itself.

Warning:

Remember that the *BindingIterator* (BIterator in the example) is an object and therefore *must be removed* otherwise dangling processes will occur. Use `CosNaming::BindingIterator::destroy()` to remove it.

```
'CosNaming_NamingContext':destroy(BIterator).
```

Destroying a Naming Context

The naming contexts are persistent and must be explicitly removed. (they are also removed if all Orber nodes in the domain are stopped).

1. Use `CosNaming::NamingContext::destroy()` to remove a `NamingContext`

```
'CosNaming_NamingContext':destroy(Sc).
```

1.7.3 Interoperable Naming Service

The OMG specifies URL schemes, which represent a CORBA object and a CORBA object bound in a `NamingContext`, for resolving references from other ORB:s. As of today, three schemes are defined:

- IOR
- corbaloc
- corbaname

IOR

A stringified IOR is a valid URL format but difficult for humans to handle through non-electronic means. This URL format does not depend on a specific Name Service and, thus, is robust and insulates the client from the encapsulated transport information and object key used to reference the object.

corbaloc

The notation of this scheme is similar to the more well known URL http, and the full corbaloc BNF is:

```

<corbaloc>          = "corbaloc:"<obj_addr_list>["/"<key_string>]
<obj_addr_list>     = [<obj_addr>","]*<obj_addr>
<obj_addr>          = <prot_addr> | <future_prot_addr>
<prot_addr>         = <rir_prot_addr> | <iiop_prot_addr>
<rir_prot_addr>     = <rir_prot_token>":"
<rir_prot_token>    = rir
<future_prot_addr>  = <future_prot_id><future_prot_addr>
<future_prot_id>    = <future_prot_token>":"
<iiop_prot_addr>    = <iiop_id><iiop_addr>
<iiop_id>           = <iiop_default> | <iiop_prot_token>":"
<iiop_default>      = ":"
<iiop_prot_token>   = "iiop"
<iiop_addr>         = <version><host>[":"<port>]
<host>              = DNS-style Host Name | ip_address
<version>           = <major> "." <minor> "@" | empty_string
<port>              = number
<major>             = number
<minor>             = number
<key_string>        = for example NameService

```

The corbaloc scheme consists of 3 parts:

- Protocol - as of today iiop or rir is supported. Using rir means that we will resolve the given Key locally, i.e., the same as using `corba:resolve_initial_references("NameService")`.
- IIOP address - this address can be divided into Version, Host and Port. If the version or port are left out they will be set to the default values 1.0 and 2089 respectively.
- KeyString - an object key, e.g., "NameService". If no Key is supplied the default value "NameService" will be used.

A corbaloc can be passed used together with

`corba:string_to_object("corbaloc:1.0@erlang.org:4001/NameService")` or set as the configuration variables `orbInitilRef` or `orbDefaultInitilRef` and calling `corba:resolve_initial_references("NameService")`. For more information see the Orber installation chapter. `corbaloc` can also be used together with `corbaname` to gain an easy access to a Name Service.

Currently, the OMG defines a set of reserved keys and the type of object, listed below, they should be associated with. The `NameService` key may *not* be changed in Orber. If you want to add one of the reserved keys as an initial service, simply use:

```

1> Factory = cosNotificationApp:start_global_factory().
2> corba:add_initial_service("NotificationService", Factory).

```

This object can then be easily resolved by any other ORB, supporting the Interoperable Naming Service, by using:

```

3> NF = corba:string_to_object("corbaloc:1.0@erlang.org:4001/NotificationService").

```

<i>String Name</i>	<i>Object Type</i>
RootPOA	PortableServer::POA
POACurrent	PortableServer::Current
InterfaceRepository	CORBA::Repository
NameService	CosNaming::NamingContext
TradingService	CosTrading::Lookup
SecurityCurrent	SecurityLevel1::Current/SecurityLevel2::Current
TransactionCurrent	CosTransaction::Current
DynAnyFactory	DynamicAny::DynAnyFactory
ORBPolicyManager	CORBA::PolicyManager
PolicyCurrent	CORBA::PolicyCurrent
NotificationService	CosNotifyChannelAdmin::EventChannelFactory
TypedNotificationService	CosTypedNotifyChannelAdmin::TypedEventChannelFactory
CodecFactory	IOP::CodecFactory
PICurrent	PortableInterceptors::Current

Table 1.5: Currently reserved key strings

corbaname

The corbaname URL scheme is an extension of the corbaloc scheme, and the full corbaname BNF is:

```

<corbaname>      = "corbaname:"<obj_addr_list>["/"<key_string>] ["#"<string_name>]
<obj_addr_list> = as described above.
<key_string>     = as described above.

```

The `string_name`, concatenated to the `corbaloc` string, identifies a binding in a naming context. A name component consists of two parts, i.e., `id` and `kind`, which is represented as follows:

<i>String Name</i>	<i>Name Sequence</i>	<i>Comment</i>
"id1/.id3.kind3"	[{"id1", ""}, {"", ""}, {"id3", "kind3"}]	The first component has no kind defined while the second component's both fields are empty.
"id1//id3.kind3"	ERROR	Not allowed, must insert a '.' between the '//'.
"id1.kind1/."	[{"id1", "kind1"}, {"", ""}]	The first component's fields are both set while the second component's both fields are empty.
"id1.kind1/id2."	ERROR	An Id with a trailing '.' is not allowed.
"i\\d1/i\\d2"	[{"i/d1", ""}, {"i.d2", ""}]	Since '.' and '/' are used to separate the components, these tokens must be escaped to be correctly converted.

Table 1.6: Stringified Name representation

After creating a stringified Name we can either use:

```
NameStr = "org.erlang",
NS       = corba:resolve_initial_references("NameService"),
Obj      = 'CosNaming_NamingContextExt':resolve_str(NS, NameStr),
```

or concatenate the Name String using:

```
NameStr = "Swedish/Soccer/Champions",
Address = "corbaname:iiop:1.0@www.aik.se:2000/NameService",
NS       = corba:resolve_initial_references("NameService"),
URLStr   = 'CosNaming_NamingContextExt':to_url(NS, Address, NameStr),
Obj      = corba:string_to_object(URLStr),
```

Using the first alternative, the configuration variables `orbInitilRef` and `orbDefaultInitilRef`, will determine which other ORB's or the local Name Service Orber will try to resolve the given string from. The second alternative allows us to override any settings of the configuration variables.

The function `to_url/3` will perform any necessary escapes compliant with IETF/RFC 2396. US-ASCII alphanumeric characters and `" | " | "/" | ":" | "?" | "@" | "&" | "=" | "+" | "$" | ";" | "-" | "_" | "." | "!" | "~" | "*" | "'" | "(" | ")"` are not escaped.

1.8 How to use security in Orber

1.8.1 Security in Orber

Introduction

Orber SSL provides authentication, privacy and integrity for your Erlang applications. Based on the Secure Sockets Layer protocol, the Orber SSL ensures that your Orber clients and servers can communicate securely over any network. This is done by tunneling IIOP through an SSL connection. To get the node secure you will also need to have a firewall which only lets through connections to certain ports.

Enable Usage of Secure Connections

To enable a secure Orber domain you have to set the configuration variable *secure* which currently only can have one of two values; *no* if no security for IIOP should be used and *ssl* if secure connections is needed (*ssl* is currently the only supported security mechanism).

The default is no security.

Setting of a CA certificate file with an option does not work due to weaknesses in the SSLeay package. A work-around in the ssl application is to set the OS environment variable `SSL_CERT_FILE` before SSL is started. However, then the CA certificate file will be global for all connections (both incoming and outgoing calls).

Configurations when Orber is Used on the Server Side

The following three configuration variables can be used to configure Orber's SSL behavior on the server side.

- *ssl_server_certfile* which is a path to a file containing a chain of PEM encoded certificates for the Orber domain as server.
- *ssl_server_verify* which specifies type of verification: 0 = do not verify peer; 1 = verify peer, verify client once, 2 = verify peer, verify client once, fail if no peer certificate. The default value is 0.
- *ssl_server_depth* which specifies verification depth, i.e. how far in a chain of certificates the verification process shall proceed before the verification is considered successful. The default value is 1.

There also exist a number of API functions for accessing the values of these variables:

- orber:ssl_server_certfile/0
- orber:ssl_server_verify/0
- orber:ssl_server_depth/0

Configurations when Orber is Used on the Client Side

When the Orber enabled application is the client side in the secure connection the different configurations can be set per client process instead and not for the whole domain as for incoming calls.

One can use configuration variables to set default values for the domain but they can be changed per client process. Below is the list of client configuration variables.

- *ssl_client_certfile* which is a path to a file containing a chain of PEM encoded certificates used in outgoing calls in the current process.
- *ssl_client_verify* which specifies type of verification: 0 = do not verify peer; 1 = verify peer, verify client once, 2 = verify peer, verify client once, fail if no peer certificate. The default value is 0.
- *ssl_client_depth* which specifies verification depth, i.e. how far in a chain of certificates the verification process shall proceed before the verification is considered successful. The default value is 1.

There also exist a number of API functions for accessing and changing the values of this variables in the client processes.

Access functions:

- orber:ssl_client_certfile/0
- orber:ssl_client_verify/0
- orber:ssl_client_depth/0

Modify functions:

- orber:set_ssl_client_certfile/1
- orber:set_ssl_client_verify/1
- orber:set_ssl_client_depth/1

1.9 Orber Examples

1.9.1 A Tutorial on How to Create a SimpleSservice

Interface Design

This example uses a very simple stack server. The specification contains two interfaces: the first is the Stack itself and the other is the StackFactory which is used to create new stacks. The specification is in the file `stack.idl`.

```
#ifndef _STACK_IDL
#define _STACK_IDL

module StackModule {

    exception EmptyStack {};

    interface Stack {

        long pop() raises(StackModule::EmptyStack);

        void push(in long value);

        void empty();

    };

    interface StackFactory {

        StackModule::Stack create_stack();
        void destroy_stack(in StackModule::Stack s);

    };

};

#endif
```

Generating Erlang Code

Run the IDL compiler on this file by calling the `ic:gen/1` function

```
1> ic:gen("stack").
```

This will produce the client stub and server skeleton. Among other files a stack API module named `StackModule_Stack.erl` will be produced. This will produce among other files a stack API module called `StackModule_Stack.erl` which contains the client stub and the server skeleton.

Implementation of Interface

After generating the API stubs and the server skeletons it is time to implement the servers and if no special options are sent to the IDL compiler the file name should be `<global interface name>_impl.erl`, in our case `StackModule_Stack_impl.erl`.

```
%% StackModule_Stack_impl example file.

-module('StackModule_Stack_impl').
-include_lib("orber/include/corba.hrl").
-include_lib("orber/examples/Stack/StackModule.hrl").
-export([pop/1, push/2, empty/1, init/1, terminate/2]).

init(Env) ->
    {ok, []}.

terminate(From, Reason) ->
    ok.

push(Stack, Val) ->
    {reply, ok, [Val | Stack]}.

pop([Val | Stack]) ->
    {reply, Val, Stack};
pop([]) ->
    corba:raise(#'StackModule_EmptyStack'{}).

empty(_) ->
    {reply, ok, []}.
```

We also have the factory interface which is used to create new stacks and that implementation is in the file `StackModule_StackFactory_impl.erl`.

```
%% StackModule_StackFactory_impl example file.

-module('StackModule_StackFactory_impl').
-include_lib("orber/include/corba.hrl").
-export([create_stack/1, destroy_stack/2, init/1, terminate/2]).

init(Env) ->
    {ok, []}.

terminate(From, Reason) ->
    ok.

create_stack(State) ->
    %% Just a create we don't want a link.
    {reply, 'StackModule_Stack':oe_create(), State}.

destroy_stack(State, Stack) ->
    {reply, corba:dispose(Stack), State}.
```

To start the factory server one executes the function `StackModule.StackFactory:oe_create/0` which in this example is done in the module `stack_factory.erl` where the started service is also registered in the name service.

```
%% stack_factory example file.

-module('stack_factory').
-include_lib("orber/include/corba.hrl").
-include_lib("orber/COSS/CosNaming/CosNaming.hrl").
-include_lib("orber/COSS/CosNaming/lname.hrl").

-export([start/0]).

start() ->
    SFok = 'StackModule_StackFactory':oe_create(),
    NS = corba:resolve_initial_references("NameService"),
    NC = lname_component:set_id(lname_component:create(), "StackFactory"),
    N = lname:insert_component(lname:create(), 1, NC),
    'CosNaming_NamingContext':bind(NS, N, SFok).
```

Writing a Client in Erlang

At last we will write a client to access our service.

```
%% stack_client example file.

-module('stack_client').
-include_lib("orber/include/corba.hrl").
-include_lib("orber/COSS/CosNaming/CosNaming.hrl").
-include_lib("orber/COSS/CosNaming/lname.hrl").

-export([run/0, run/1]).

run() ->
    NS = corba:resolve_initial_references("NameService"),
    run_1(NS).

run(HostRef) ->
    NS = corba:resolve_initial_references_remote("NameService", HostRef),
    run_1(NS).

run_1(NS) ->
    NC = lname_component:set_id(lname_component:create(), "StackFactory"),
    N = lname:insert_component(lname:create(), 1, NC),
    case catch 'CosNaming_NamingContext':resolve(NS, N) of
        {'EXCEPTION', E} ->
            io:format("The stack factory server is not registered~n", []);
        SF ->
            %% Create the stack
            SS = 'StackModule_StackFactory':create_stack(SF),
```

```
%% io:format("SS pid ~w~n",[iop_ior:get_key(SS)]),
'StackModule_Stack':push(SS, 4),
'StackModule_Stack':push(SS, 7),
'StackModule_Stack':push(SS, 1),
'StackModule_Stack':push(SS, 1),
Res = 'StackModule_Stack':pop(SS),
io:format("~w~n", [Res]),
Res1 = 'StackModule_Stack':pop(SS),
io:format("~w~n", [Res1]),
Res2 = 'StackModule_Stack':pop(SS),
io:format("~w~n", [Res2]),
Res3 = 'StackModule_Stack':pop(SS),
io:format("~w~n", [Res3]),

%% Remove the stack
'StackModule_StackFactory':destroy_stack(SF, SS)

end.
```

Writing a Client in Java

To write a Java client for Orber you must have another ORB that uses IIOP for client-server communication and supports a Java language mapping. It must also have support for IDL:CosNaming/NamingContext or IDL:CosNaming/NamingContextExt. If the client ORB support Interoperable Naming Service the Java Client can look like:

```
/*
 * Stack example using Interoperable Naming Service.
 */

package StackModule;
import org.omg.CORBA.*;
import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORB.*;

public class StackClient
{
    public static void main(String args[])
    {
        org.omg.CORBA.Object objRef;
        StackFactory sfRef = null;
        Stack sRef = null;
        // The argument can look like
        // "corbaname::host:4001/#StackFactory"
        String corbaName = new String(args[0]);
        try{
            ORB orb = ORB.init(args, null);

            objRef = orb.string_to_object(corbaName);
            sfRef = StackFactoryHelper.narrow(objRef);
            sRef = sfRef.create_stack();
        }
```



```

        sRef.push(4);
        sRef.push(7);
        sRef.push(1);
        sRef.push(1);

        try{
            System.out.println(sRef.pop());
            System.out.println(sRef.pop());
            System.out.println(sRef.pop());
            System.out.println(sRef.pop());
            // The following operation shall
            // return an EmptyStack exception
            System.out.println(sRef.pop());
        }
        catch(EmptyStack es) {
            System.out.println("Empty stack");
        };

        sfRef.destroy_stack(sRef);
    }
    catch(SystemException se)
    {
        System.out.println("Unexpected exception: " + se.toString());
        return;
    }
}
}

```

If the Client ORB does not support Interoperable Naming Service, a Java package named `Orber` is included with our product. It contains just one class, `InitialReference` which can be used to get the initial reference to Orber's naming service. The Java client will then look like this:

```

/*
 * Stack example.
 */

package StackModule;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORB.*;

public class StackClient
{
    public static void main(String args[])
    {
        NamingContext nsContext;
        org.omg.CORBA.Object objRef;
        StackFactory sfRef = null;
        Stack sRef = null;
        org.omg.CORBA.Object nsRef, initRef;
        NameComponent[] name = new NameComponent[1];
    }
}

```

```
Orber.InitialReference ir = new Orber.InitialReference();
Orber.InitialReferences init;
String srvHost = new String(args[0]);
Integer srvPort = new Integer(args[1]);

try
{
    ORB orb = ORB.init(args, null);

    // Create Initial reference (objectkey "INIT").
    String s = ir.stringified_ior(srvHost, srvPort.intValue());
    initRef = orb.string_to_object(s);
    init = Orber.InitialReferencesHelper.narrow(initRef);

    // Fetch name service reference.
    nsRef = init.get("NameService");
    nsContext = NamingContextHelper.narrow(nsRef);
    // Create a name
    name[0] = new NameComponent("StackFactory", "");

    try
    {
        objRef = nsContext.resolve(name);
    }
    catch(Exception n)
    {
        System.out.println("Unexpected exception: " + n.toString());
        return;
    }

    sfRef = StackFactoryHelper.narrow(objRef);
    sRef = sfRef.create_stack();

    sRef.push(4);
    sRef.push(7);
    sRef.push(1);
    sRef.push(1);

    try
    {
        System.out.println(sRef.pop());
        System.out.println(sRef.pop());
        System.out.println(sRef.pop());
        System.out.println(sRef.pop());
        // The following operation shall return an EmptyStack exception
        System.out.println(sRef.pop());
    }
    catch(EmptyStack es)
    {
        System.out.println("Empty stack");
    };

    sfRef.destroy_stack(sRef);
}
```

```

    }
    catch(SystemException se)
    {
        System.out.println("Unexpected exception: " + se.toString());
        return;
    }
}
}
}

```

Note:

If an ORB does not support CosNaming at all the `cos_naming.idl` file must be compiled and imported.

Building the Example

To build the example for access from a Java client you need a Java enabled ORB. The build log below, using OrbixWeb's IDL compiler, describes the scenario where the Client ORB does not support Naming Service.

```

fingolfin 127> erl
Erlang (BEAM) emulator version 4.9

Eshell V4.9 (abort with ^G)
1> ic:gen(stack).
Erlang IDL compiler version 20
ok
2> make:all().
Recompile: oe_stack
Recompile: StackModule_StackFactory
Recompile: StackModule_Stack
Recompile: StackModule
Recompile: stack_client
Recompile: stack_factory
Recompile: StackModule_StackFactory_impl
Recompile: StackModule_Stack_impl
up_to_date
3>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a

fingolfin 128> idl stack.idl
fingolfin 129> idl InitialReferences.idl
fingolfin 130> idl <OTP_INSTALLATIONPATH>/lib/orber-<Orber Version>/COSS/
                  CosNaming/cos_naming.idl
fingolfin 131>
fingolfin 132> cd java_output/
fingolfin 133> javac *.java

```

```
fingolfin 134> cd CosNaming/
fingolfin 135> javac *.java
fingolfin 136> cd ../_NamingContext/
fingolfin 137> cd javac *.java
fingolfin 138> cd ../../Orber/
fingolfin 139> javac *.java
fingolfin 140> cd ../StackModule/
fingolfin 141> javac *.java
fingolfin 142> cd ../../
fingolfin 143> javac *.java
fingolfin 144> cp StackClient.class java_output/StackModule/.
```

How to Run Everything

Below is a short transcript on how to run Orber. The commands for starting the new socket communication package will not be necessary when it is used as default in OTP R3A. In R2 it is only available unsupported, and without documentation but Orber uses this for better IIOP performance. An example `.inetrc` can also be found in Orber's example directory and is named `inetrc` (without the starting `.`).

```
fingolfin 143> erl
Erlang (BEAM) emulator version 4.9
```

```
Eshell V4.9      (abort with ^G)
1> mnesia:create_schema([]).
ok
2> orber:install([]).
ok
3> orber:start().
ok
4> oe_stack:oe_register().
ok
5> stack_factory:start().
ok
6> stack_client:run().
1
1
7
4
ok
7>
```

Before testing the Java part of this example generate and compile Java classes for `orber/examples/stack.idl`, `orber/examples/InitialReferences.idl` and `orber/COSS/CosNaming/cos_naming.idl` as seen in the build example. We have tested with OrbixWeb.

To run the Java client use the following command (the second parameter is the port number for the bootstrap port):

```
fingolfin 38> java StackModule.StackClient fingolfin 4001
[New Connection (fingolfin,4001, null,null,pid=0) ]
[New Connection (fingolfin.du.etx.ericsson.se,4001, null,null,pid=0) ]
```

```

1
1
7
4
Empty stack
fingolfin 39>

```

1.9.2 A Tutorial on How to Start Orber as Lightweight

Preparation

When starting Erlang the configuration parameter `lightweight` must be used. The value is set to a list of remote modifiers, equal to the `orber:resolve_initial_references_remote/2` argument, i.e., “`iiop://host:port`”. On these given nodes, all necessary `oe_X:oe_register()` calls must be done before running a Orber lightweight.

Lightweight Orber do not allow us to:

- Create objects locally
- Accept incoming requests
- Access local NameService
- Register data in the IFR

With lightweight Orber we do not:

- Start Mnesia
- Run `orber:install/1`

To be able to start objects we must supply a factory on a non-lightweight node(s) which can start necessary objects. One way to accomplish this is:

```

smaug 125> erl -orber domain "ORBER_MAIN"
Erlang (BEAM) emulator version 4.9

Eshell V4.9      (abort with ^G)
1> mnesia:create_schema([]).
2> orber:install([]).
3> orber:start().
4> oe_MyFactory:oe_register().
5> oe_MyObjects:oe_register().  %% Do this for all objects necessary.
6> Factory=MyFactory_Creater:oe_create().
7> NS=orber:resolve_initial_references("NameService").
8> NC=lname_component:set_id(lname_component:create(), "myFactory").
9> N =lname:insert_component(lname:create(), 1, NC).
10> 'CosNaming_NamingContext':bind(NS, N, Factory)).

```

Now we have a factory we can access from, hence, we can now start a lightweight Orber:

```
fingolfin 14> erl -orber lightweight ["iiop://host1:port\",
                                     \"iiop://host2:port\"]
               -orber domain \"ORBER_LIGHT\"
Erlang (BEAM) emulator version 4.9

Eshell V4.9      (abort with ^G)
1> orber:start_lightweight().
2> Fac=corba:string_to_object(\"corbaname::Host:Port/NameService#myFactory\"),
3> Obj=MyFactory_Creater:MyObject(Fac, Args).
4> MyObject:myFunction(Obj,Args2).
```

It is not necessary to start both Orber types using the configuration parameter `domain`, but at least one of them.

1.10 Orber Stubs/Skeletons

1.10.1 Orber Stubs and Skeletons Description

This example describes the API and behavior of Orber stubs and skeletons.

Server Start

Orber servers can be started in several ways. The chosen start functions determines how the server can be accessed and its behavior.

Using `Module_Interface:oe_create()` or `oe_create_link()`:

- No initial data can be passed.
- Cannot be used as a supervisor child start function.
- Only accessible through the object reference returned by the start function. The object reference is no longer valid if the server dies and is restarted.

Using `Module_Interface:oe_create(Env)` or `oe_create_link(Env)`:

- Initial data can be passed using `Env`.
- Cannot be used as a supervisor child start function.
- Only accessible through the object reference returned by the start function. The object reference is no longer valid if the server dies and is restarted.

Using `Module_Interface:oe_create(Env, Options)`:

- Initial data can be passed using `Env`.
- Cannot be used as a supervisor child start function.
- Accessible through the object reference returned by the start function. If the option `{regname, RegName}` is used the object reference stays valid even if the server has been restarted.
- If the options `{persistent, true}` and `{regname, {global, Name}}` is used, the result from an object invocation will be the exception `'OBJECT_NOT_EXIST'` only if the object has terminated with reason `normal` or `shutdown`. If the object is in the process of restarting, the result will be `{error, Reason}` or the exception `'COMM_FAILURE'`.

- The option `{pseudo, true}` makes it possible to start create non-server objects. There are, however, some limitations, which are further described in the `Pseudo objects` section.

Using `Module_Interface:oe_create_link(Env, Options)`:

- Initial data can be passed using `Env`.
- Can be used as a supervisor child start function if the option `{sup_child, true}` used.
- Accessible through the object reference returned by the start function. If the option `{regname, RegName}` is used the object reference stays valid even if the server has been restarted.
- If the options `{persistent, true}` and `{regname, {global, Name}}` is used, the result from an object invocation will be the exception `'OBJECT_NOT_EXIST'` only if the object has terminated with reason `normal` or `shutdown`. If the object is in the process of restarting, the result will be `{error, Reason}` or the exception `'COMM_FAILURE'`.
- For starting a server as a supervisor child you should use the options `[{persistent, true}, {regname, {global, Name}}, {sup_child, true}]` and of type *transient*. This configuration allows you to delegate restarts to the supervisor and still be able to use the same object reference and be able to see if the server is permanently terminated. Please note you must use *supervisor/stdlib-1.7* or later and that it returns `{ok, Pid, Object}` instead of just `Object`.
- Using the option `{pseudo, true}` have the same effect as using `oe_create/2`.

Warning:

To avoid flooding Orber with old object references start erlang using the flag `-orber objectkeys_gc_time Time`, which will remove all object references related to servers being dead for `Time` seconds. To avoid extra overhead, i.e., performing garbage collect if no persistent objects are started, the `objectkeys_gc_time` default value is *infinity*. For more information, see the `orber` and `corba` documentation.

Warning:

Orber still allow `oe_create(Env, {Type, RegName})` and `oe_create_link(Env, {Type, RegName})` to be used, but may not in future releases.

Pseudo Objects

This section describes Orber pseudo objects.

The Orber stub can be used to start a `pseudo` object, which will create a non-server implementation. A pseudo object introduce some limitations:

- The functions `oe_create_link/2` is equal to `oe_create/2`, i.e., no link can or will be created.
- The BIF:s `self()` and `process_flag(trap_exit, true)` behaves incorrectly.
- The IC option `{{impl, "M:I"}, "other_impl"}` has no effect. The call-back functions must be implemented in a file called `M.I.impl.erl`
- The call-back functions must be implemented as if the IC option `{this, "M:I"}` was used.
- The `gen_server` State changes have no effect. The user can provide information via the `Env` start parameter and the State returned from `init/2` will be the State passed in following invocations.

- The `gen_server` reply Timeout have no effect.
- The option `{pseudo, true}` overrides all other start options.
- Only the functions, besides own definitions, `init/2` (called via `oe_create*/2`) and `terminate/2` (called via `corba:dispose/1`) must be implemented.

By adopting the rules for pseudo objects described above we can use `oe_create/2` to create server or pseudo objects, by excluding or including the option `{pseudo, true}`, without changing the call-back module.

To create a pseudo object do the following:

```
fingolfin 127> erl
Erlang (BEAM) emulator version 4.9

Eshell V4.9 (abort with ^G)
1> ic:gen(myDefinition, [{this, "MyModule::MyInterface"}]).
Erlang IDL compiler version 20
ok
2> make:all().
Recompile: oe_MyDefinition
Recompile: MyModule_MyInterface
Recompile: MyModule_MyInterface_impl
up_to_date
3> PseudoObj = MyModule_MyInterface:oe_create(Env, [{pseudo, true}]).
```

The call-back functions must be implemented as `MyFunction(OE.THIS, State, Args)`, and called by `MyModule_MyInterface:MyFunction(PseudoObj, Args)`.

Call-back Module

This section provides an example of how a call-back module may be implemented.

Note:

Arguments and Replies are determined by the IDL-code and, hence, not further described here.

```
%%%-----
%%% File      : Module_Interface_impl.erl
%%% Author   :
%%% Purpose  :
%%% Created  :
%%%-----

-module('Module_Interface_impl').

%%----- INCLUDES -----
-include_lib("orber/include/corba.hrl").
-include_lib("../..").

%%----- EXPORTS-----
%% Arity depends on IC configuration parameters and the IDL
```

```

%% specification.
-export([own_function/X]).

%%----- gen_server specific -----
-export([init/1, terminate/2, code_change/3, handle_info/2]).

%%-----
%% function : server specific
%%-----
init(InitialData) ->
    %% 'trap_exit' optional (have no effect if pseudo object).
    process_flag(trap_exit,true),

    %%--- Possible replies ---
    %% Reply and await next request
    {ok, State}.

    %% Reply and if no more requests within Time the special
    %% timeout message should be handled in the
    %% Module_Interface_impl:handle_info/2 call-back function (use the
    %% IC option {{handle_info, "Module::Interface"}, true}).
    {ok, State, Timeout}

    %% Return ignore in order to inform the parent, especially if it is a
    %% supervisor, that the server, as an example, did not start in
    %% accordance with the configuration data.
    ignore
    %% If the initializing procedure fails, the reason
    %% is supplied as StopReason.
    {stop, StopReason}

terminate(Reason, State) ->
    ok.

code_change(OldVsn, State, Extra) ->
    {ok, NewState}.

%% If use IC option {{handle_info, "Module::Interface"}, true}.
%% (have no effect if pseudo object).
handle_info(Info, State) ->
    %%--- Possible replies ---
    %% Await the next invocation.
    {noreply, State}.
    %% Stop with Reason.
    {stop, Reason, State}.

%%--- two-way -----
%% If use IC option {this, "Module:Interface"}
%% (Required for pseudo objects)
own_function(This, State, .. Arguments ..) ->

%% If not use IC option {this, "Module:Interface"}

```

```
own_function(State, .. Arguments ..) ->
    %%--- Possible replies ---
    %% Reply and await next request
    {reply, Reply, State}

    %% Reply and if no more requests within Time the special
    %% timeout message should be handled in the
    %% Module_Interface_impl:handle_info/2 call-back function (use the
    %% IC option {{handle_info, "Module::Interface"}, true}).
    {reply, Reply, State, Timeout}

    %% Stop the server and send Reply to invoking object.
    {stop, StopReason, Reply, State}

    %% Stop the server and send no reply to invoking object.
    {stop, StopReason, State}

    %% Raise exception. Any changes to the internal State is lost.
    corba:raise(Exception).

%%--- one-way -----
%% If use IC option {this, "Module:Interface"}
%% (Required for pseudo objects)
own_function(This, State, .. Arguments ..) ->

%% If not use IC option {this, "Module:Interface"}
own_function(State, .. Arguments ..) ->
    %%--- Possible results ---
    {noreply, State}

    %% Release and if no more requests within Time the special
    %% timeout message should be handled in the
    %% Module_Interface_impl:handle_info/2 call-back function (use the
    %% IC option {{handle_info, "Module::Interface"}, true}).
    {noreply, State, Timeout}

    %% Stop the server with StopReason.
    {stop, StopReason, State}

%%----- END OF MODULE -----
```

1.11 CORBA System and User Defined Exceptions

1.11.1 System Exceptions

Orber, or any other ORB, may raise a System Exceptions. These exceptions contain status- and minor-fields and may not appear in the operation's raises exception IDL-definition.

Status Field

The status field indicates if the request was completed or not and will be assigned one of the following erlang atoms:

<i>Status</i>	<i>Description</i>
'COMPLETED_YES'	The operation was invoked on the target object but an error occurred after the object replied. This occurs, for example, if a server replies but Orber is not able to marshal and send the reply to the client ORB.
'COMPLETED_NO'	Orber failed to invoke the operation on the target object. This occurs, for example, if the object no longer exists.
'COMPLETED_MAYBE'	Orber invoked the operation on the target object but an error occurred and it is impossible to decide if the request really reached the object or not.

Table 1.7: Table 1: System Exceptions Status

Minor Field

The minor field contains an integer, which the OMG initially defined to be opaque. In later CORBA specifications explicit values have been defined for the minor field, but only for a handful of the exceptions, to enable a more exact description. Currently, Orber still uses the old settings, which is why the minor field should not be taken into consideration for exceptions raised by Orber.

Supported System Exceptions

The OMG CORBA specification defines the following exceptions:

- *'BAD_CONTEXT'* - if a request does not contain a correct context this exception is raised.
- *'BAD_INV_ORDER'* - this exception indicates that operations have been invoked in the wrong order, which would cause, for example, a dead-lock.
- *'BAD_OPERATION'* - raised if the target object exists, but that the invoked operation is not supported.
- *'BAD_PARAM'* - is thrown if, for example, a parameter is out of range or otherwise considered illegal.
- *'BAD_TYPECODE'* - if illegal type code is passed, for example, encapsulated in an any data type the *'BAD_TYPECODE'* exception will be raised.
- *'BAD_QOS'* - raised whenever an object cannot support the required quality of service.
- *'CODESET_INCOMPATIBLE'* - raised if two ORB's cannot communicate due to different representation of, for example, char and/or wchar.
- *'COMM_FAILURE'* - raised if an ORB is unable to setup communication or it is lost while an operation is in progress.
- *'DATA_CONVERSION'* - raised if an ORB cannot convert data received to the native representation. See also the *'CODESET_INCOMPATIBLE'* exception.
- *'FREE_MEM'* - the ORB failed to free dynamic memory and failed.
- *'IMP_LIMIT'* - an implementation limit was exceeded in the ORB at run time. A object factory may, for example, limit the number of object clients allowed to create.

- *'INTERNAL'* - an internal failure occurred in an ORB, which is unrecognized. You may consider contacting the ORB provider's support.
- *'INTF_REPOS'* - the ORB was not able to reach the interface repository, or some other failure relating to the interface repository is detected.
- *'INITIALIZE'* - the ORB initialization failed due to, for example, network or configuration error.
- *'INVALID_TRANSACTION'* - is raised if the request carried an invalid transaction context.
- *'INV_FLAG'* - an invalid flag was passed to an operation, which caused, for example, a connection to be closed.
- *'INV_IDENT'* - this exception indicates that an IDL identifier is incorrect.
- *'INV_OBJREF'* - this exception is raised if an object reference is malformed or a nil reference (see also `corba:create_nil_objref/0`).
- *'INV_POLICY'* - the invocation cannot be made due to an incompatibility between policy overrides that apply to the particular invocation.
- *'MARSHAL'* - this exception may be raised by the client- or server-side when either ORB is unable to marshal/unmarshal requests or replies.
- *'NO_IMPLEMENT'* - if the operation exists but no implementation exists, this exception is raised.
- *'NO_MEMORY'* - the ORB has run out of memory.
- *'NO_PERMISSION'* - the caller has insufficient privileges, such as, for example, bad SSL certificate.
- *'NO_RESOURCES'* - a general platform resource limit exceeded.
- *'NO_RESPONSE'* - no response available of a deferred synchronous request.
- *'OBJ_ADAPTER'* - indicates administrative mismatch; the object adapter is not able to associate an object with the implementation repository.
- *'OBJECT_NOT_EXIST'* - the object have been disposed or terminated; clients should remove all copies of the object reference and initiate desired recovery process.
- *'PERSIST_STORE'* - the ORB was not able to establish a connection to its persistent storage or data contained in the the storage is corrupted.
- *'REBIND'* - a request resulted in, for example, a *'LOCATION_FORWARD'* message; if the policies are incompatible this exception is raised.
- *'TIMEOUT'* - raised if a request fail to complete within the given time-limit.
- *'TRANSACTION_MODE'* - a transaction policy mismatch detected.
- *'TRANSACTION_REQUIRED'* - a transaction is required for the invoked operation but the request contained no transaction context.
- *'TRANSACTION_ROLLEDBACK'* - the transaction associated with the request has already been rolled back or will be.
- *'TRANSACTION_UNAVAILABLE'* - no transaction context can be supplied since the ORB is unable to contact the Transaction Service.
- *'TRANSIENT'* - the ORB could not determine the current status of an object since it could not be reached. The error may be temporary.
- *'UNKNOWN'* - is thrown if an implementation throws a non-CORBA, or unrecognized, exception.

1.11.2 User Defined Exceptions

User exceptions is defined in IDL-files and is listed in operation's raises exception listing. For example, if we have the following IDL code:

```
module MyModule {  
  
    exception MyException {};  
    exception MyExceptionMsg { string ExtraInfo; };  
  
    interface MyInterface {  
  
        void foo()  
            raises(MyException);  
  
        void bar()  
            raises(MyException, MyExceptionMsg);  
  
        void baz();  
    };  
};
```

1.11.3 Throwing Exceptions

To be able to raise `MyException` or `MyExceptionMsg` exceptions, the generated `MyModule.hrl` must be included, and typical usage is:

```
-module('MyModule_MyInterface_impl').  
-include("MyModule.hrl").  
  
bar(State) ->  
    case TestingSomething of  
        ok ->  
            {reply, ok, State};  
        {error, Reason} when list(Reason) ->  
            corba:raise('#MyModule_MyExceptionMsg'{'ExtraInfo' = Reason});  
        error ->  
            corba:raise('#MyModule_MyException'{})  
    end.
```

1.11.4 Catching Exceptions

Depending on which operation we invoke we must be able to handle:

- foo - `MyException` or a system exception.
- bar - `MyException`, `MyExceptionMsg` or a system exception.
- baz - a system exception.

Catching and matching exceptions can be done in different ways:

```
case catch 'MyModule_MyInterface':bar(MIReference) of
  ok ->
    %% The operation raised no exception.
    ok;
  {'EXCEPTION', #MyModule_MyExceptionMsg{'ExtraInfo' = Reason}} ->
    %% If we want to log the Reason we must extract 'ExtraInfo'.
    error_logger:error_msg("Operation 'bar' raised: ~p~n", [Reason]),
    ... do something ...;
  {'EXCEPTION', E} when record(E, 'OBJECT_NOT_EXIST') ->
    ... do something ...;
  {'EXCEPTION', E} ->
    ... do something ...
end.
```

1.12 Orber Interceptors

1.12.1 Using Interceptors

For Inter-ORB communication, e.g., via IIOP, it is possible to intercept requests and replies. To be able to use Interceptors Orber the configuration parameter `interceptors` must be defined.

Configure Orber to Use Interceptors

The configuration parameter `interceptors` must be defined, e.g., as command line option:

```
erl -orber interceptors "{native, ['myInterceptor']}"
```

It is possible to use more than one interceptor; simply add them to the list and they will be invoked in the same order as they appear in the list.

Creating Interceptors

Each supplied interceptor *must* export the following functions:

- *new_out_connection/3* - this operation is called when a client application calls an object residing on remote ORB.
- *new_in_connection/3* - invoked when a client side ORB tries to set up a connection to the target ORB.
- *out_request/6* - supplies all request data on the client side ORB.
- *out_request_encoded/6* - similar to *out_request* but the request body is encoded.
- *in_request_encoded/6* - after a new request arrives at the target ORB the request data is passed to the interceptor in encoded format.
- *in_request/6* - prior to invoking the operation on the target object, the interceptor *in_request* is called.
- *out_reply/6* - after the target object replied the *out_reply* operation is called with the result of the object invocation.
- *out_reply_encoded/6* - before sending a reply back to the client side ORB this operation is called with the result in encoded format.

- *in_reply_encoded/6* - after the client side ORB receives a reply this function is called with the reply in encoded format.
- *in_reply/6* - before delivering the reply to the client this operation is invoked.
- *closed_in_connection/1* - when a connection is terminated on the client side this function is called.
- *closed_out_connection/1* - if an outgoing connection is terminated this operation will be invoked.

The operations *new_out_connection*, *new_in_connection*, *closed_in_connection* and *closed_out_connection* operations are only invoked *once* per connection. The remaining operations are called, as shown below, for every Request/Reply to/from remote CORBA Objects.

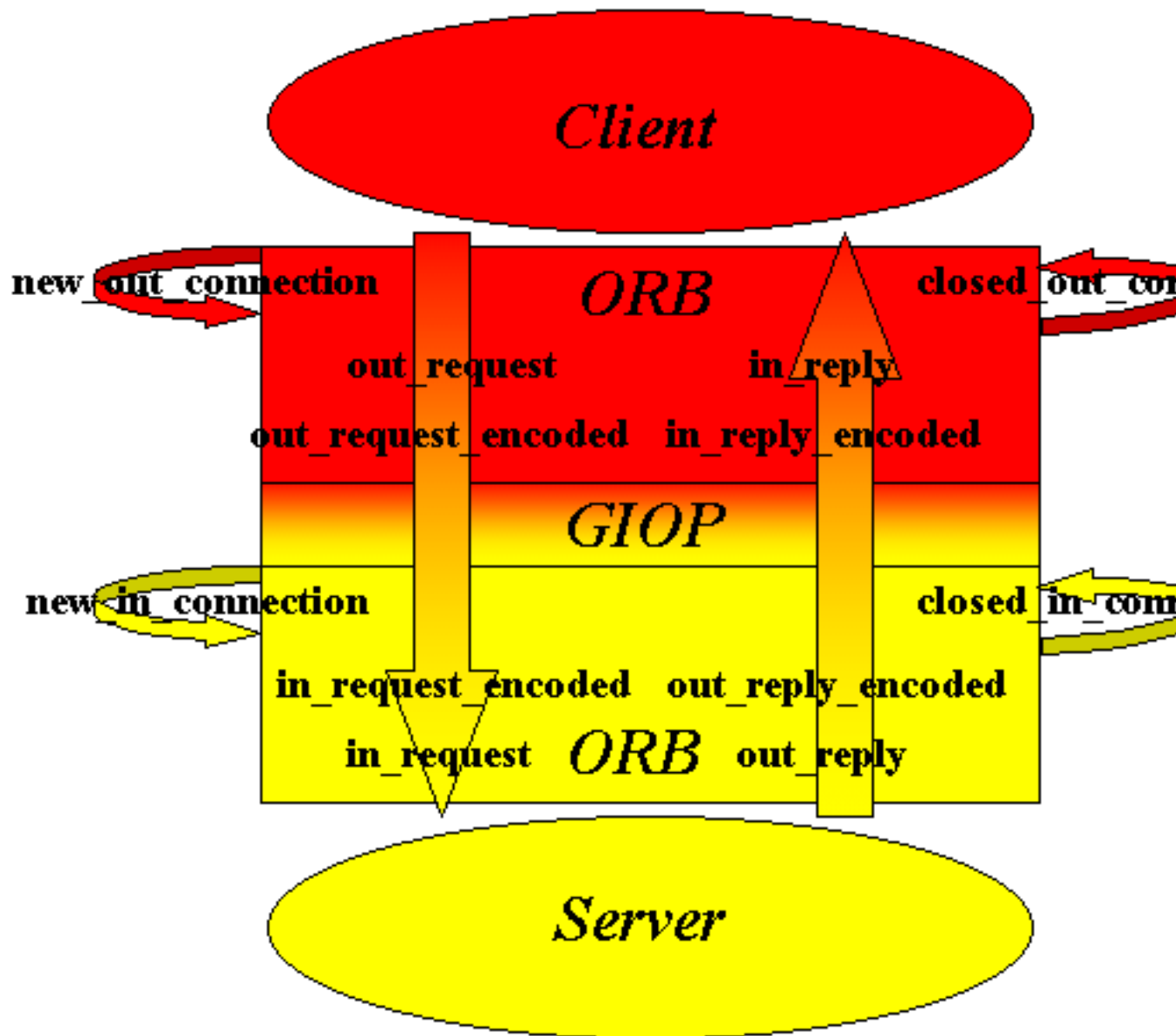


Figure 1.6: The Invocation Order of Interceptor Functions.

1.12.2 Interceptor Example

Assume we want to create a simple access service which purpose is to:

- Only allow incoming request from ORB's residing on a certain set of nodes.
- Restrict the objects any client may invoke operations on.
- Only allow outgoing requests to call a limited set of external ORB's.
- Add a checksum to each binary request/reply body.

To restricts the access we use a protected and named ets-table holding all information. How the ets-table is initiated and maintained is implementation specific, but it contain {Node, ObjectTable, ChecksumModule} where Node is used as ets-key, ObjectTable is a reference to another ets-table in which we store which objects the clients are allowed to invoke operations on and ChecksumModule determines which module we should use to handle the checksums.

```
new_in_connection(Arg, Host, Port) ->
  %% Since we only use one interceptor we do not care about the
  %% input Arg since it is set do undefined by Orber.
  case ets:lookup(in_access_table, Host) of
    [] ->
      %% We may want to log the Host/Port to see if someone tried
      %% to hack in to our system.
      exit("Access not granted");
    [{Host, ObjTable, ChecksumModule}] ->
      {ObjTable, ChecksumModule}
  end.
```

The returned tuple, i.e., {ObjTable, ChecksumModule}, will be passed as the first argument whenever invoking one of the interceptor functions. Unless the connection attempt did not fail we are now ready for receiving requests from the client side ORB.

When a new request comes in the first interceptor function to be invoked is `in_request_encoded`. We will remove the checksum from the coded request body in the following way:

```
in_request_encoded({ObjTable, ChecksumModule}, ObjKey, Ctx, Op, Bin, Extra) ->
  NewBin = ChecksumModule:remove_checksum(Bin),
  {NewBin, Extra}.
```

If the checksum check fails the ChecksumModule should invoke `exit/1`. But if the check succeeded we are now ready to check if the client-ORB objects are allowed to invoke operations on the target object. Please note, it is possible to run both checks in `in_request_encoded`. Please note, the checksum calculation must be relatively fast to ensure a good throughput.

If we want to we can restrict any clients to only use a subset of operations exported by a server:

```
in_request({ObjTable, ChecksumModule}, ObjKey, Ctx, Op, Params, Extra) ->
  case ets:lookup(ObjTable, {ObjKey, Op}) of
    [] ->
      exit("Client tried to invoke illegal operation");
    [SomData] ->
      {Params, Extra}
  end.
```


At this point Orber are now ready to invoke the operation on the target object. Since we do not care about what the reply is the `out_reply` function do nothing, i.e.:

```
out_reply(_, _, _, _, Reply, Extra) ->
    {Reply, Extra}.
```

If the client side ORB expects a checksum to be added to the reply we add it by using:

```
out_reply_encoded({ObjTable, ChecksumModule}, ObjKey, Ctx, Op, Bin, Extra) ->
    NewBin = ChecksumModule:add_checksum(Bin),
    {NewBin, Extra}.
```

Warning:

If we manipulate the binary as above the behaviour *must* be `Bin == remove_checksum(add_checksum(Bin))`.

For outgoing requests the principle is the same. Hence, it is not further described here. The complete interceptor module would look like:

```
-module(myInterceptor).

%% Interceptor functions.
-export([new_out_connection/3,
        new_in_connection/3,
        closed_in_connection/1,
        closed_out_connection/1,
        in_request_encoded/6,
        in_reply_encoded/6,
        out_reply_encoded/6,
        out_request_encoded/6,
        in_request/6,
        in_reply/6,
        out_reply/6,
        out_request/6]).

new_in_connection(Arg, Host, Port) ->
    %% Since we only use one interceptor we do not care about the
    %% input Arg since it is set do undefined by Orber.
    case ets:lookup(in_access_table, Host) of
        [] ->
            %% We may want to log the Host/Port to see if someone tried
            %% to hack in to our system.
            exit("Access not granted");
        [{Host, ObjTable, ChecksumModule}] ->
            {ObjTable, ChecksumModule}
    end.

new_out_connection(Arg, Host, Port) ->
    case ets:lookup(out_access_table, Host) of
```

```
    [] ->
        exit("Access not granted");
    [{Host, ObjTable, ChecksumModule}] ->
        {ObjTable, ChecksumModule}
end.

in_request_encoded({_, ChecksumModule}, ObjKey, Ctx, Op, Bin, Extra) ->
    NewBin = ChecksumModule:remove_checksum(Bin),
    {NewBin, Extra}.

in_request({ObjTable, _}, ObjKey, Ctx, Op, Params, Extra) ->
    case ets:lookup(ObjTable, {ObjKey, Op}) of
        [] ->
            exit("Client tried to invoke illegal operation");
        [SomData] ->
            {Params, Extra}
    end.

out_reply(_, _, _, _, Reply, Extra) ->
    {Reply, Extra}.

out_reply_encoded({_, ChecksumModule}, ObjKey, Ctx, Op, Bin, Extra) ->
    NewBin = ChecksumModule:add_checksum(Bin),
    {NewBin, Extra}.

out_request({ObjTable, _}, ObjKey, Ctx, Op, Params, Extra) ->
    case ets:lookup(ObjTable, {ObjKey, Op}) of
        [] ->
            exit("Client tried to invoke illegal operation");
        [SomeData] ->
            {Params, Extra}
    end.

out_request_encoded({_, ChecksumModule}, ObjKey, Ctx, Op, Bin, Extra) ->
    NewBin = ChecksumModule:add_checksum(Bin),
    {NewBin, Extra}.

in_reply_encoded({_, ChecksumModule}, ObjKey, Ctx, Op, Bin, Extra) ->
    NewBin = ChecksumModule:remove_checksum(Bin),
    {NewBin, Extra}.

in_reply(_, _, _, _, Reply, Extra) ->
    {Reply, Extra}.

closed_in_connection(Arg) ->
    %% Nothing to clean up.
    Arg.

closed_out_connection(Arg) ->
    %% Nothing to clean up.
    Arg.
```

Note:

One can also use interceptors for debugging purposes, e.g., print which objects and operations are invoked with which arguments and the outcome of the operation. In conjunction with the configuration parameter `orber_debug_level` it is rather easy to find out what went wrong or just to log the traffic.

1.13 OrberWeb

1.13.1 Using OrberWeb

OrberWeb is intended to make things easier when developing and testing applications using Orber. The user is able to interact with Orber via a GUI by using a web browser.

OrberWeb requires that the application `WebTool` is available and started on at least one node; if so OrberWeb can usually be used to access Orber nodes supporting the Interoperable Naming Service. How to start OrberWeb is described in [Starting OrberWeb \[page 62\]](#)

The OrberWeb GUI consists of a *Menu Frame* and a *Data Frames*.

The Menu Frame

The menu frame consists of:

- *Node List* - which node to access.
- *Configuration* - see how Orber on the current node is configured.
- *Name Service* - browse the NameService and add/remove a Context/Object.
- *IFR Types* - see which types are registered in IFR.
- *Create Object* - create a new object and, possibly, store it in the NameService.



Figure 1.7: The Menu Frame.

Which nodes we can access is determined by what is returned when invoking `[node() | nodes()]`. If you cannot see a desired node in the list, you have to call `net_adm:ping(Node)`. But this requires that the node is started with the distribution switched on (e.g. `erl -sname myNode`); this also goes for the node OrberWeb is running on.

The Configuration Data Frame

When accessing the *Configuration* page OrberWeb presents a table containing the configuration settings [page 11] for the target node.

Configuration	
Key	Value
IIOP Request Timeout	infinity
IIOP Connection Timeout	infinity
IIOP Setup Connection Timeout	infinity
IIOP Port	4001
Bootstrap Port	4001
Orber Domain	MyDomain
Nodes in Domain	[main@shagrat]
Default GIOP Version	{1,1}
Objectkeys GC	infinity
Using Interceptors	false
Debug Level	10
ORBInitRef	undefined
ORBDefaultInitRef	undefined
[[{Key, Value}]]	
Change it	

Figure 1.8: Configuration Settings.

It is also possible to change those configuration parameters which can be changed when Orber is already started. The Key-Value pairs is given as a list of tuples, e.g., `[[{orber_debug_level, 5}, {iiop_timeout, 60}, {giop_version, {1,2}}]`. If one tries to update a parameter which may not be changed an error message will be displayed.

The IFR Data Frame

All types registered in the IFR (Interface Repository) which have an associated IFR-id can be viewed via the IFR Data Frame. This gives the user an easy way to confirm that all necessary IDL-specifications have been properly registered. All available types are listed when choosing IFR Types in the menu fram:



Figure 1.9: Select Type.

After selecting a type all definitions of that particular type will be displayed. If no such bindings exists the table will be empty.

Since Orber adds definitions to the IFR when it is installed (e.g. CosNaming), not only types defined by the user will show up in the table. In the figure below you find the the NameService exceptions listed.

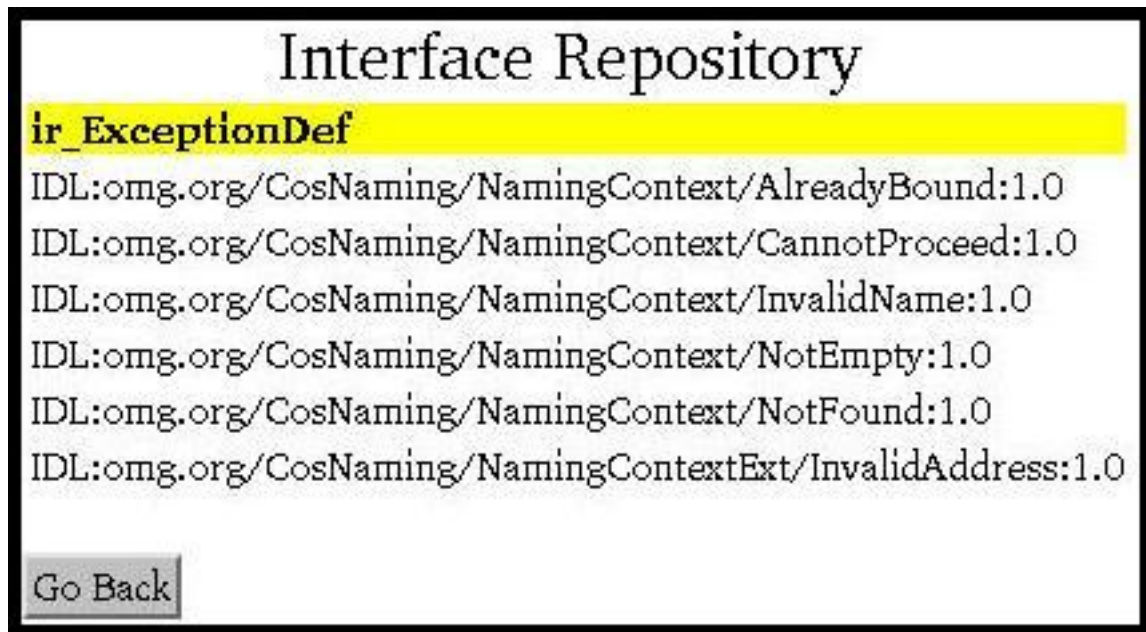


Figure 1.10: List Registered Exceptions.

The NameService Data Frame

The NameService main purpose is to make possible to bind object references, which can client applications can resolve and invoke operations on. Initially, the NameService is empty. The most common scenario, is that user applications create Contexts and add objects in the NameService. OrberWeb allows the user to do the very same thing.

When referencing an object or context you must use stringified NameComponents. For more information see the Interoperable Naming Service [page 22]. In the following example we will use the string *org/erlang/TheObjectName*, where *org* and *erlang* will be contexts and *TheObjectName* the name the object will be bound to.

Since the NameService is empty in the beginning, the only thing we can do is creating a new context. Simply write *org* in the input field and press *New Context*. If OrberWeb was able to create the context or not, is shown in the completion message. If succesfull, just press the *Go Back* button. Now, a link named *org* should be listed in the table. In the right column the context type is displayed. Contexts are associated with *ncontext* and objects with *nobject*.



The screenshot shows a web interface titled "NameService". Below the title is a yellow horizontal bar containing the text "Root Context". Underneath this bar, the word "EMPTY" is centered. At the bottom of the interface, there is a text input field containing the text "org" and a button labeled "New Context" to its right.

Figure 1.11: Add a New Context.

To create the next level context (i.e. *erlang*), simply follow the link and repeat the procedure. If done correctly, a table containing the same data as the following figure should be the result if you follow the *erlang* link. Note, that the path is displayed in the yellow field.

If a context does not contain any sub-contexts or object bindings, it is possible to delete the context. If these requirements are met, a `Delete Context` button will appear. A completion status message will be displayed after deleting the context.



The screenshot shows a web interface titled "NameService". Below the title is a yellow horizontal bar containing the text "org/erlang". Underneath this bar, the word "EMPTY" is centered. To the right of "EMPTY" is a button labeled "Delete Context". At the bottom of the interface, there is a text input field containing the text "I" and a button labeled "New Context" to its right. In the bottom left corner, there is a button labeled "Go Back".

Figure 1.12: Delete Context.

Now it is possible to bind an object using the complete name string. To find out how this is done using OrberWeb see Object Creation [page 60]. For now, we will just assume that an object have been created and bound as *TheObjectName*.



Figure 1.13: Object Stored in the NameService.

If you follow the *TheObjectName* link, data about the bound object will be presented. Note, depending on which type of object it is, the information given differs. It would, for example, not be possible to display a Pid for all types of objects since it might reside on a Java-ORB. In the figure below a CosNotification FilterFactory have been bound under the name *org/erlang/TheObjectName*.

NameService

Key	Value
IFR Id	IDL:omg.org/CosNotifyFilter/FilterFactory:1.0
Stored As	org/erlang/TheObjectName
External Object	false
Non Existent	false
Pid	<0.597.0> IOR:00

IOR String

Operations create_mapping_filter/2
 create_filter/1

Figure 1.14: Object Data.

OrberWeb also makes it possible to remove a binding and dispose the associated object. Pressing *Unbind* the binding will be removed but the object will still exist. But, if the *Unbind & Dispose* button is pressed, the binding will be removed and the object terminated.

The Object Creation Data Frame

This part makes it possible to create a new object and, if wanted, store it the NameService.

Create a New Object

Module	<input type="text" value="Module_Interface"/>
Arguments	<input]"="" type="text" value='["String", {logfile, "/tmp/MyLoggFile"}]'/>
Options	<input]"="" type="text" value='[{regname, {global, "TheObjectName"}}]'/>
Name String	<input type="text" value="org/erlang/TheObjectName"/>
Operation to use	<input checked="" type="radio"/> Bind <input type="radio"/> Rebind
<input type="button" value="Create it"/>	

Figure 1.15: Create a New Object.

- *Module* - simply type the name of the module of the object type you want to create. If the module begins with a capital letter, we normally must write 'Module_Interface'. But, when using OrberWeb, you shall *NOT*. Since we cannot create linked objects this is not an option.
- *Arguments* - the supplied arguments must be written as a single Erlang term. That is, as a list or tuple containing other Erlang terms. The arguments will be passed to the `init` function of the object. It is, however, not possible to use Erlang records. If OrberWeb is not able to parse the arguments, an error message will be displayed. If left empty, an empty list will be passed.
- *Options* - the options can be the ones listed under ModuleInterface [page 104] in Orber's Reference manual. Hence, they are not further described here. But, as an example, in the figure above we started the object as globally registered. If no options supplied the object will be started as default.
- *Name String* - if left empty the object will *not* be registered in the NameService. Hence, it is important that you can access the object in another way, otherwise a zombie process is created. In the previous section we used the name string *org/erlang/TheObjectName*. If we choose the same name here, the listed contexts (i.e. *org* and *erlang*) must be created *before* we can create and bind the object to *TheObjectName*. If this requirement is not met, OrberWeb cannot bind the object. Hence, the object will be terminated and an error message displayed.
- *Operation to use* - which option choosed will determine the behaviour of OrberWeb. If you choose *bind* and a binding already exists an error message will be displayed and the newly started object terminated. But if you choose *rebind* any existing binding will over-written.

1.13.2 Starting OrberWeb

You may choose to start OrberWeb on node, on which Orber is running or not. But the Erlang distribution must be started (e.g. by using `-sname aNodeName`). Now, all you have to do is to invoke:

```
erl> webtool:start().  
WebTool is available at http://localhost:8888/  
Or http://127.0.0.1:8888/
```

Type one of the URL:s in your web-browser. If you want to access the WebTool application from different machine, just replace `localhost` with its name. For more information, see the WebTool documentation.

1.14 Debugging

1.14.1 Tools and FAQ

Persons who use Orber for the first time may find it hard to tell what goes wrong when trying to setup communication between an Orber-ORB and ORB:s supplied by another vendor or another Orber-ORB. The purpose of this chapter is to inform about the most common mistakes and what tools one can use to overcome these problems.

Tools

To begin with, Orber can be configured to run in debug mode. There are four ways to set this parameter:

- `erl -orber orber_debug_level 10` - can be added to a start-script.
- `corba:orb_init([{ orber_debug_level, 10 }])` - this operation must be invoked *before* starting Orber.
- `orber:configure(orber_debug_level, 10)` - this operation can be invoked at any time.
- *OrberWeb* - via the Configuration menu one can easily change the configuration. For more information, see the OrberWeb chapter in this User's Guide.

When Orber runs in debug mode, printouts will be generated if anything abnormal occurs (not necessarily an error). An error message typically looks like:

```
=ERROR REPORT==== 29-Nov-2001::14:09:55 ===  
===== Orber =====  
[410] corba:common_create(orber_test_server, [{pseudo,true}]);  
not a boolean(truce).  
=====
```

In the example above, we tried to create an object with an incorrect option (i.e. should have been `{pseudo,true}`).

If you are not able to solve the problem, you should include all generated reports when contacting support or using the erlang-questions mailing list.

It is also possible to trace all communication between an Orber-ORB and, for example, a Java-ORB, communicating via IIOP. All you need to do is to activate an interceptor [page 48]. Normally, the users must implement the interceptor themselves, but for your convenience Orber includes a pre-compiled interceptor called `orber_iiop_tracer`.

Warning:

Logging all traffic is *expensive*. Hence, only use the supplied interceptor during test and development.

The `orber_iiop_tracer` interceptor uses the `error_logger` module to generate the logs. If the traffic is intense you probably want to write the reports to a log-file. This is done by, for example, invoking:

```
erl> error_logger:tty(false).
erl> error_logger:logfile({open, "/tmp/IIOPTrace"}).
```

The `IIOPTrace` file will contain reports which looks like:

```
=INFO REPORT==== 29-Nov-2001::15:26:28 ===
===== new_out_connection =====
Node       : 'myNode@myHost'
To Host    : "123.456.789.012"
To Port    : 2000
=====

=INFO REPORT==== 29-Nov-2001::15:26:28 ===
===== out_request =====
Connection: {"123.456.789.012",2000}
Operation  : resolve
Parameters: [[{'CosNaming_NameComponent',
               "AIK", "SwedishIcehockeyChampions"}]]
Context    : [{ 'IOP_ServiceContext', 1,
                 {'CONV_FRAME_CodeSetContext', 65537, 65801}}]
=====
```

To activate the `orber_iiop_tracer` interceptor, you have two options:

- `erl -orber interceptors "{native,[orber_iiop_tracer]}"` - can be added to a start-script.
- `corba:orb_init([interceptors, {native, [orber_iiop_tracer]}])` - this operation must be invoked *before* starting Orber.

FAQ

Q: When my client, typically written in C++ or Java, invoke narrow on an Orber object reference it fails?

A: You must register your application in the IFR by invoking `oe_register()`. If the object was created by a COS-application, you must run `install` (e.g. `cosEventApp:install()`).

A: Confirm, by consulting the IDL specifications, that the received object reference really inherit from the interface you are trying to narrow it to.

Q: I am trying to register my application in the IFR but it fails. Why?

A: If one, or more, interface in your IDL-specification inherits from other interface(s), you must register them before registering your application. Note, this also apply when you inherit interfaces supported by a COS-application. Hence, they must be installed prior to registration of your application.

Q: I have a Orber client and server residing on two different Orber instances but I only get the 'OBJECT_NOT_EXIST' exception, even though I am sure that the object is still alive?

A: If the two Orber-ORB's are not intended to be a part of multi-node ORB, make sure that the two Orber-ORB's have different *domain* names set (see configuration settings [page 11]). The easiest way to confirm this is to invoke `orber:info()` on each node.

Q: When I'm trying to install and/or start Orber it fails?

A: Make sure that no other Orber-ORB is already running on the same node. If so, change the `iiop_port` configuration parameter (see configuration settings [page 11]).

Q: My Orber server is invoked via IIOP but Orber cannot marshal the reply?

A: Consult your IDL file to confirm that your replies are of the correct type. If it is correct and the return type is, for example, a struct, make sure you have set every field in the struct. If you do not do that it will be set to the atom 'undefined', which most certainly is not correct.

A: Check that you handle `inout` and `out` parameters correctly (see the IDL specification). For example, a function which have one out-parameter and should return void, then your call-back module should return `{reply, {ok, OutParam}, State}`. Note, even though the return value is void (IDL) you must reply with `ok`.

1.15 Orber Release Notes

1.15.1 Orber 3.2.12, Release Notes

Improvements and new features

- Orber now check if an external IOR contains any `TAG_ALTERNATE_IIOP_ADDRESS` components when trying to setup a connection to another ORB.
Own id: OTP-4294
- It is now possible to add `TAG_ALTERNATE_IIOP_ADDRESS` components to a local object reference. See `corba:add_alternate_iiop_address/3`.
Own id: OTP-4294
- Orber now allows unions with no default value defined and a discriminator out of range to be sent via IIOP. The value-field is set to the atom `undefined`.
Own id: OTP-4295
- The `corba` module now exports a function, `print_object/1/2`, which prints IOR's in a more readable form.
Own id: OTP-4296

Fixed bugs and malfunctions

- Since “all” ORB’s accept ISO-8859-1 encoding of chars and strings, Orber assumed that it could be used at all time to reduce the overhead. The JDK-1.3 only accepts ISO 646:1991 IRV (US-ASCII), even though ISO-8859-1 is default, which is why Orber now checks which codeset is accepted.
Own Id: OTP-4298
- When invoking a Locate Request, Orber in some cases did not reply with a Locate Reply header (used a Reply header).
Own Id: OTP-4293

Incompatibilities

-

Known bugs and problems

- The same as in the last release.

1.15.2 Orber 3.2.11, Release Notes

Improvements and new features

- If the underlying OS was not configured to allow Erlang to use the fully qualified host name the result could an incorrect IP-address. Hence, if you want to upgrade to Orber-3.2.11 and the fully qualified name must be used you must upgrade your kernel version. Most likely, this change will NOT cause any problems, but if in doubt please contact support or use the mailing-list. See also the release notes for Orber-3.2.6.
Own id: OTP-3966

Fixed bugs and malfunctions

- When looking up Initial Service (e.g. using `corbaloc`, `corbaname` or `corba:resolve_initial_references_remote/2`) and communicating via SSL, Orber used the wrong port number.
Own Id: OTP-4264

Incompatibilities

-

Known bugs and problems

- The same as in the last release.

1.15.3 Orber 3.2.10, Release Notes

Improvements and new features

- It is now possible to add new initial references, which can, for example, be accessed via `corba:resolve_initial_references/1` or the `corbaloc` schema.
Own Id: OTP-4258
- The orber module now exports functions, `iiop_connections` and `iiop_connections_pending`, which, respectively, list all currently open connections to other ORB's and connections which are in process of being set up to another ORB.
Own Id: OTP-4262
- Orber now allows the user to define an interval of ports which Orber is may use (i.e. ports on the local machine) when trying to connect to another ORB. This behavior is usefull if Orber resides behind a firewill which only allow applications to use certain ports when communicating with the outside world. If this option is set, it is absolutely necessary to set `iiop_connection_timeout`. If not, there is risk that Orber run out of ports, which will result in communicatio failure. This option cannot be used when using SSL since it does not support this feature. The default behavior is that any available port will be used (as before).
Own Id: OTP-4260
- One can now install Orber's NameService as `disc_copies`, but the default behavior is that Orber uses `ram_copies`.
Own Id: OTP-4259
- A pre-compiled IIOP-trace interceptor is now included in the Orber release. For more information, see the Debugging chapter in the User's Guide.
Own Id: OTP-4257
- It is now possible to set Orber's configuration parameters in, for example, an Erlang shell. Consult `corba:orb_init/1` and `orber:configure/2`.
Own Id: OTP-4261
- The Orber release now include OrberWeb, which is an extension of the WebTool application (first released in R8B). Hence, WebTool must be installed to enable this feature. For more information, see the chapter OrberWeb in the User's Guide. OrberWeb is intended to be used during test and development.
Own Id: OTP-4257

Fixed bugs and malfunctions

- When setting up two Orber ORB's, where one of the ORB's domain name was a prefix of the other ORB's, communication via IIOP would fail. To eliminate any further configuration problems, one may not use `^G` (i.e. `"\007"`) in the domain name. Due to this change it is not possible to upgrade during run-time.
Own Id: OTP-4229
- When using a mix of IIOP-versions (1.0 vs 1.1/1.2) and sending/receiving IOR's to/from Orber could result in a MARSHAL exception.
Own Id: OTP-4230
- Encoding/decoding of `wchar/wstring` when using IIOP-1.2 do now follow the OMG standard. If your ORB do not follow the standard, contact support for information how to make a work-around to solve this problem. In most cases it is sufficient to configure the ORB's to communicate via IIOP-1.1.
Own Id: OTP-4263

Incompatibilities

- The encoding/decoding of wchar/wstring when using IIOP-1.2 have been updated. See above.
Own Id: OTP-4263
- Orber no longer returns a 'EXIT' message when trying to install Orber, i.e., invoking orber:install/1/2, on a disc-less node. But if the installation fails due to any other reason, Orber still return a 'EXIT' message.
Own Id: OTP-4256

Known bugs and problems

- OrberWeb only tested with Netscape-4.75. Furthermore, until WebTool reaches version 1.0 OrberWeb should also be considered to be a beta version.
Own Id: OTP-4257
- OrberWeb do not escape arguments passed when, for example, creating a new context. Hence, for now you are recommended to only use letters.
Own Id: OTP-4257

1.15.4 Orber 3.2.9, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- External IOR:s containing unsupported or incorrectly placed TaggedComponents was corrupted when Orber forwarded the IOR via IIOP. This bug was introduced in 3.2.6.
Own Id: OTP-4170

Incompatibilities

-

Known bugs and problems

-

1.15.5 Orber 3.2.8, Release Notes

Improvements and new features

- Orber now support interceptors.
Own Id: -

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

-

1.15.6 Orber 3.2.7, Release Notes

Improvements and new features

- When Orber acted as server-side and communicating via IIOP the overhead was unreasonably large and memory consuming (depended on the IDL-specification). This have now been fixed and will, especially, improve the performance when invoking operations with no, or simple, arguments on a complex interface. This change will have little effect on objects started as pseudo since this problem did not affect them.
Own Id: OTP-4063
- When Orber tried to set up a connection to another ORB which did not respond all IIOP access where blocked until the TCP protocol generated a timeout. Now only requests to that particular ORB are queued.
Own Id: OTP-4060

Fixed bugs and malfunctions

- It was not possible to invoke the operation `CosNaming_BindingIterator:next_one` via IIOP if no more bindings existed.
Own id: OTP-4004

Incompatibilities

-

Known bugs and problems

-

1.15.7 Orber 3.2.6, Release Notes

Improvements and new features

- Registering data in the IFR overhead reduced.
Own id: OTP-3904
- The overhead for the function `is_a/1` have been reduced, which also affects remote `narrow` operations for inherited interfaces (e.g. using Java or C++ ORBs).
Own id: OTP-3904
- If the underlying OS was not configured to allow Erlang to lookup the host-name by using the short-name the result was always the IP-adress 127.0.0.1 (loop-back). Now Orber uses the full name. Hence, make sure the `net_adm:localhost/0` and `inet:getaddr/2` return proper values.
Own id: OTP-3966

Fixed bugs and malfunctions

- The `CONV_FRAME_CodeSetComponentInfo` struct was not placed correctly in IOR:s. Each profile must be self-sustained which is why this information must be duplicated in each profile. Currently this only applies for the IOP-profile but will also concern future protocols.
Own id: OTP-3992

Incompatibilities

-

Known bugs and problems

-

1.15.8 Orber 3.2.5, Release Notes

Improvements and new features

- Orber now defines the configuration variable, `iiop_setup_connection_timeout`, which makes it possible to timeout connection attempts to another ORB before the OS TCP timeout is activated.
Own id: OTP-3961
- It is now possible to configure Orber to generate reports when abnormal situations occurs. For more information consult the User's Guide regarding the configuration parameter `orber_debug_level`. Note, it is not recommended to use this option for delivered systems since some of the reports is not to be considered as errors.
Own id: OTP-3962
- Orber now accepts a list of addresses as value for the configuration parameter `orbInitRef`.
Own id: OTP-3945
- Orber now includes services defined by the configuration parameter `orbInitRef` when invoking `corba:list_initial_services/0`.
Own id: OTP-3946

Fixed bugs and malfunctions

- When using the configuration variable '`orbDefaultInitRef`' with a value pointing to another Orber-ORB it was not possible to install Orber since Orber used to create default `NamingContexts`. Orber no longer add these contexts.
Own id: OTP-3943
- Orber accessed `corbaloc` addresses in reverse order. Now fixed.
Own id: OTP-3944

Incompatibilities

- When installing Orber no default `NamingContext`'s, i.e., `host`, `hosts`, `resources`, `development`, `factories` and `workgroup`, will be added. These contexts was defined in a cancelled specification.
Own id: OTP-3942
- `corbaloc` addresses are now accessed in FIFO order (instead of LIFO).
Own id: OTP-3944

Known bugs and problems

-

1.15.9 Orber 3.2.4, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- When communicating via IIOP using version 1.2 Orber used incorrect offset for reply bodies containing system exceptions, exceptions and location forward.
Own id: OTP-3912
- Orber did not return correct IFR Id:s when raising system exceptions via IIOP.
Own id: OTP-3911
- If two different processes concurrently manipulated a `CosNaming::NamingContext` the data could become corrupted. For single-node Orber this error occurred in version 3.2.1, 3.2.2 and 3.2.3. For multi-node Orber this behaviour have been present at all time.
Own id: OTP-3910

Incompatibilities

- Since Orber now returns a different, and correct, IFR-id for systems exceptions other ORB:s and older versions of Orber might raise a different exception, probably MARSHAL or UNKNOWN. This only occurs when communicating via IIOP. It is not possible to upgrade during runtime. Use `orber:stop()`, load new version and restart Orber by invoking `orber:start()`.
Own id: OTP-3911

Known bugs and problems

-

1.15.10 Orber 3.2.3, Release Notes

Improvements and new features

- Improved performance for all types, simple and complex, when communicating via IIOP. It is not possible to upgrade during runtime. Use `orber:stop()`, load new version and restart Orber by invoking `orber:start()`.
Own id: OTP-3905

Fixed bugs and malfunctions

- If a psuedo object raises an exception or exits the exception was only returned, not thrown.
Own id: OTP-3907
- Orber defined an incorrect ID for CodeSets. This may cause INV_OBJREF or DATA_CONVERSION exceptions to be thrown, it depends on the other ORB.
Own id: OTP-3899

Incompatibilities

-

Known bugs and problems

-

1.15.11 Orber 3.2.2, Release Notes

Improvements and new features

- The behaviour of Orber when receiving unsupported or incorrect messages have now been improved.
Own id: OTP-3903
- Time consumed by `oe.MyModule:oe_register()` decreased.
Own id: OTP-3904

Fixed bugs and malfunctions

- When Orber received a 'location_forward' reply, the result from the second invocation was never delivered to the client. Now fixed.
Own id: OTP-3814

Incompatibilities

-

Known bugs and problems

-

1.15.12 Orber 3.2.1, Release Notes

Improvements and new features

- It is now possible to use external `NamingContexts` when, for example, using `'CosNaming.NamingContextExt':bind_context/3`.
Own id: OTP-3902

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

-

1.15.13 Orber 3.2, Release Notes

Improvements and new features

- Orber now supports IIOP-version 1.2.
Own id: OTP-3901
- Improved encoding and decoding performance for IIOP requests containing `struct`, `union` or user defined `exceptions`.
Own id: OTP-3900

Fixed bugs and malfunctions

- Setting the `bootstrap_port` configuration parameter to a value less than 1024 made it impossible to start Orber properly. Now fixed.
Own id: OTP-3898

Incompatibilities

-

Known bugs and problems

-

1.15.14 Orber 3.1.8, Release Notes

Improvements and new features

- Orber now accepts `Indirection/Repeated CORBA::TypeCode` as input and/or return value when communicating via IIOP.
Own id: -

Fixed bugs and malfunctions

- When another ORB replied with `location forward` Orber failed to decode this. Now fixed.
Own id: OTP-3709
- Orber failed to encode `CORBA::TypeCode` containing `tk_alias`, e.g., sending an `#any{}` which encapsulates data defined by `typedef`.
Own id: OTP-3689

Incompatibilities

-

Known bugs and problems

-

1.15.15 Orber 3.1.7, Release Notes

Improvements and new features

- Earlier, Orber did not use the IIOP/GIOP version specified in an external object key when invoking an intra-ORB request.
Own id: OTP-3663
- The OMG standard now support an Interoperable Naming Service. Initially there where two proposals of which Orber earlier supported one of them. Now both standards are supported.
Own id: OTP-3664
- The OMG have redefined the operator, used when encoding requests via IIOP, for the function `corba_object:non_existent/1`. CORBA version 2.0 and 2.2 compliant ORB:s is supposed to support the old definition, while later versions, i.e., 2.3, is supposed to use the new operator (`_non_existent` instead of `_not_existent`). Orber accepts both versions.
Own id: OTP-3679

Fixed bugs and malfunctions

- If an Orber node crashed and was restarted the object keys could point to other processes than it should, which may cause problems if, for example, the other process terminates due to it does not handle unknown messages. Now Orber GC object keys for objects residing on the crashed node. If Orber is started as a multi-node ORB of which one or more nodes runs an older Orber version they can still communicate but with an increased overhead. Hence, all nodes should be upgraded during a relatively short time. If Orber is stopped, i.e., `orber:stop()` or a shutdown is generated, objects residing on that node will be terminated.
Own id: OTP-3678
- If an IDL-file contains two interfaces of which the first one contains an exception and the second interface, which inherits the first one, contain an operation which raises this exception the IFR failed since multiple references where found when invoking `orber_ifr:lookup_id/2`. Now fixed.
Own id: OTP-3665

Incompatibilities

- To be able to start Orber as lightweight the mnesia application cannot be listed in the “`orber.app`” file. You might find it necessary to add ‘mnesia’ to the applications-list. For example, you cannot upgrade an older version of Orber (not started as lightweight) to this version without adding mnesia to the application dependencies list.
Own id: OTP-3666
- The function `corba_object:non_existent/1` have been updated to follow the CORBA 2.3 standard. Hence, Intra-ORB communication with ORB:s not supporting this standard will fail. The operation `corba_object:not_existent/1` allow users to use the old standard. Consult the ORB vendor’s documentation to decide which function to use.
Own id: OTP-3679

Known bugs and problems

-

1.15.16 Orber 3.1.6, Release Notes

Improvements and new features

- Cosmetic update of internal functions.
Own id: -

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

-

1.15.17 Orber 3.1.5, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- When decoding TypeCode for an object reference, e.g., as a part of an #any{}, Orber failed. This is no longer the case.
Own id: OTP-3631

Incompatibilities

-

Known bugs and problems

-

1.15.18 Orber 3.1.4, Release Notes

Improvements and new features

- The function `start_lightweight/1` have been added to the `orber` module. This function allow us to start orber as lightweight without, or override, the configuration parameter `-orber_lightweight`.
Own id: -
- A new configuration parameter, `'iiop_connection_timeout Secs'`, is now available. This parameter's purpose, is to terminate the socket connection on the client side if a timespan of Secs seconds have passed. The connection will, however, NOT be terminated if a client still waits for a reply. For the last scenario to happen, the client have been configured to use a larger timeout value than the configuration parameter `'iiop_connection_timeout'` have been set to.
Own id: -
- Up until now, invoking an an operation with an extra Timeout parameter (using the IC option: `ic:gen(IdlFile, [{timeout,"module::interface"}])`), only applied to local Objects. Now, using the IC option above, when compiling the stubs, and adding the extra Timeout parameter, a timeout will also be triggered when calling Objects residing on other ORB:s. The return value, after a timeout has been triggered, have changed from an EXIT message to raising the system exception `COMM_FAILURE`. For more information, about how this feature interacts with the configuration parameter `'iiop_timeout'`, consult the documentation.
Own id: -
- When using invalid intra-ORB configuration, i.e., incorrect Port/IP-address, when trying to connect to another ORB, a CRASH REPORT was generated if the configuration parameter `'-boot_start_sasl'` was used. This behaviour has now changed.
Own id: -

Fixed bugs and malfunctions

- If a client-side ORB terminated the IIOP connection immediately there was a possibility that the server responsible detecting this did not.
Own id: OTP-3593
- Setting the configuration parameter `'iiop_timeout'` did not result in a correct behaviour, i.e., no timeout triggered.
Own id: OTP-3555

Incompatibilities

- When using the IC option, `ic:gen(IdlFile, [{timeout,"module::interface"}])`, an EXIT was the timeout result. Now, the system exception `COMM_FAILURE` is raised.

Known bugs and problems

-

1.15.19 Orber 3.1.3, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- Orber did not ignore unrecognized TaggedProfiles. Other vendors may have registered own TAG's with the OMG. These TAG's are valid but not necessarily handled by other vendors.
Own id: OTP-3514
- When passing Object references over IIOP, decoding local references could fail. Now fixed.
Own id: OTP-3515

Incompatibilities

-

Known bugs and problems

-

1.15.20 Orber 3.1.2, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- Previously the OMG have published two suggestions for Interoperable Name Service, of which, the CORBA 3 specify orbos/98-10-11 to be implemented. Unfortunately, the Interoperable Name Service Orber supports, is the one not chosen. Hence, the `InitialReferences.idl` will not be according to the future standard. The modules name is now changed from CORBA to Orber. This will affect code which are using this interface. The idl specification must be recompiled and then CORBA must be changed to Orber in the client.
Own id: OTP-3468, OTP-3155
- Now possible to run `oe_unregister` when the IDL-specification contains exceptions correctly.
Own Id: OTP-3447
- Now possible to run `oe_unregister` when the IDL-specification contains attributes.
Own Id: OTP-3439

Incompatibilities

The change in `InitialReferences.idl` to clash with the Corba standard implies changes in code that use this interface. See the OTP-3468 and OTP-3155 in the Fixed bugs and malfunctions chapter above.

Known bugs and problems

-

1.15.21 Orber 3.1.1, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- When introducing the configuration parameter `ip_address` it was no longer possible to have the same default behaviour as before. Now fixed.
Own Id: OTP-3431
- The internal request number handling never checked if maximum reached. Now the counter restart at 0 after reaching max.
Own Id: OTP-3415
- Orber did not handle locate-requests correctly, i.e., not able to recognize the new internal representation of object references.
Own Id: OTP-3414

Incompatibilities

-

Known bugs and problems

-

1.15.22 Orber 3.1, Release Notes

Improvements and new features

- It is now possible to start Orber as lightweight.
Own Id: -
- It is now possible to create pseudo objects, i.e., not server objects.
Own Id: -
- One new system exception introduced; 'BAD_QOS'.
Own Id: -
- Orber now supports the types 'long long' and 'unsigned long long'
Own Id: -

Fixed bugs and malfunctions

- Encoding typecode for complex exceptions (non-empty body) was not done correctly.
Own Id: OTP-3390
- `orber_iiop_pm` crashed when it received an 'EXIT'. Now fixed.
Own Id: OTP-3391

Incompatibilities

-

Known bugs and problems

-

1.15.23 Orber 3.0.1, Release Notes

Improvements and new features

- Orber is now able to handle upgrade properly.
Own Id: -

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

-

1.15.24 Orber 3.0, Release Notes

Improvements and new features

- It is now possible to use secure IIOP connections to and from Orber. Orber currently only supports security with the help of SSL and not SECIOP.
Own Id: OTP-1510
- It is now possible to start Orber objects as supervisor childs using `Module_Interface:oe_create_link/2` or `corba:create_link/4` as the start function.
Own Id: -
- It is now possible to start a Orber object and be able to tell apart if it is in the process of being restarted or has permanently terminated. This is also the reason for introducing `objectkeys_gc_time` configuration parameter.
Own Id: -
- The service `CosEvent` has been removed from orber and become its own application, called `cosEvent`.
Own Id: -
- The service `CosTransactions` is now available as a separate application, called `cosTransactions`.
Own Id: OTP-1741
- Three new system exceptions, `'TRANSACTION_REQUIRED'`, `'TRANSACTION_ROLLEDBACK'` and `'INVALID_TRANSACTION'`, introduced. Required by the `cosTransactions` application.
Own Id: -

- An configuration variable `ip_address` has been added, so it's possible to listen on a specific ip interface on a multi interface host. The value is the ip address as a string or a tuple of four integers, default value is all interfaces.
Own Id: OTP-3294

Fixed bugs and malfunctions

- set- and get-operations for the 'any'-module now behaves properly.
Own Id: OTP-3355
- Orber can now handle IORs which contain more than one "Tagged Profile".
Own Id: OTP-3266

Incompatibilities

- CosEvent include paths have changed since it is now a separate application, called `cosEvent`.
- The internal representation of object references have changed. Orber do, however, recognize the old representation. But object references (created by Orber 2.2.2 or older) stored and used through several Orber upgrades may not be supported.
- The functions `oe_create/2` and `oe_create_link/2` now take an options list as its second argument. Orber still allow `oe_create*(Env, {Type,RegName})` to be used, but may not in future releases.

Known bugs and problems

-

Orber Reference Manual

Short Summaries

- Erlang Module **CosNaming** [page 94] – The CosNaming service is a collection of interfaces that together define the naming service.
- Erlang Module **CosNaming.BindingIterator** [page 97] – This interface supports iteration over a name binding list.
- Erlang Module **CosNaming.NamingContext** [page 99] – This interface supports different bind and access functions for names in a context.
- Erlang Module **CosNaming.NamingContextExt** [page 102] – This interface contains operation for converting a Name sequence to a string and back.
- Erlang Module **Module.Interface** [page 104] – Orber generated stubs/skeletons.
- Erlang Module **any** [page 110] – the corba any type
- Erlang Module **corba** [page 112] – The functions on CORBA module level
- Erlang Module **corba_object** [page 118] – The CORBA Object interface functions
- Erlang Module **interceptors** [page 120] – Describe the functions which must be exported by any supplied Orber native interceptor.
- Erlang Module **lname** [page 125] – Interface that supports the name pseudo-objects.
- Erlang Module **lname_component** [page 127] – Interface that supports the name pseudo-objects.
- Erlang Module **orber** [page 129] – The main module of the Orber application
- Erlang Module **orber_ifr** [page 136] – The Interface Repository stores representations of IDL information
- Erlang Module **orber_tc** [page 150] – Help functions for IDL typecodes

CosNaming

No functions are exported.

CosNaming_BindingIterator

The following functions are exported:

- `next_one(BindinIterator)` -> Return
[page 97] Return a binding
- `next_n(BindinIterator, HowMany)` -> Return
[page 97] Return a binding list
- `destroy(BindingIterator)` -> Return
[page 97] Destroy the iterator object

CosNaming_NamingContext

The following functions are exported:

- `bind(NamingContext, Name, Object)` -> Return
[page 100] Bind a Name to an Object
- `rebind(NamingContext, Name, Object)` -> Return
[page 100] Bind an Object to the Name even if the Name already is bound
- `bind_context(NamingContext1, Name, NamingContext2)` -> Return
[page 100] Bind a Name to an NamingContext
- `rebind_context(NamingContext1, Name, NamingContext2)` -> Return
[page 100] Bind an NamingContext to the Name even if the Name already is bound
- `resolve(NamingContext, Name)` -> Return
[page 100] Retrieve an Object bound to Name
- `unbind(NamingContext, Name)` -> Return
[page 101] Remove the binding for a Name
- `new_context(NamingContext)` -> Return
[page 101] Create a new NamingContext
- `bind_new_context(NamingContext, Name)` -> Return
[page 101] Create a new NamingContext and bind it to a Name
- `destroy(NamingContext)` -> Return
[page 101] Destroy a NamingContext
- `list(NamingContext, HowMany)` -> Return
[page 101] List returns a all bindings in the context

CosNaming_NamingContextExt

The following functions are exported:

- `to_string(NamingContext, Name)` -> Return
[page 102] Stringify a Name sequence to a string
- `to_name(NamingContext, NameString)` -> Return
[page 102] Convert a stringified Name to a Name sequence
- `to_url(NamingContext, AddressString, NameString)` -> Return
[page 102] Return an URL string constructed from the given Address and Name strings
- `resolve_str(NamingContext, NameString)` -> Return
[page 102] Return the object associated, if any, with the given name string

Module_Interface

The following functions are exported:

- `typeID()` -> `TypeId`
[page 105] Return the Type ID related to this stub/skeleton
- `oe_create()` -> `ObjRef`
[page 105] Start a Orber server.
- `oe_create_link()` -> `ObjRef`
[page 105] Start a linked Orber server.
- `oe_create(Env)` -> `ObjRef`
[page 105] Start a Orber server.
- `oe_create_link(Env)` -> `ObjRef`
[page 105] Start a linked Orber server.
- `oe_create(Env, Options)` -> `ObjRef`
[page 105] Start a Orber stub/skeleton
- `oe_create_link(Env, Options)` -> `Return`
[page 106] Start a Orber stub/skeleton
- `Module_Interface:own_functions(ObjRef, Arg1, ..., ArgN)` -> `Reply`
[page 106] User defined function which is not a part of Orber
- `Module_Interface:own_functions(ObjRef, Timeout, Arg1, ..., ArgN)` -> `Reply`
[page 106] User defined function which is not a part of Orber
- `Module_Interface_impl:init(Env)` -> `CallReply`
[page 107] User defined function which is not a part of Orber
- `Module_Interface_impl:terminate(Reason, State)` -> `ok`
[page 107] User defined function which is not a part of Orber
- `Module_Interface_impl:code_change(OldVsn, State, Extra)` -> `CallReply`
[page 107] User defined function which is not a part of Orber
- `Module_Interface_impl:handle_info(Info, State)` -> `CallReply`
[page 107] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(This, State, Arg1, ..., ArgN)` -> `CallReply`
[page 108] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(State, Arg1, ..., ArgN)` -> `CallReply`
[page 108] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(This, State, Arg1, ..., ArgN)` -> `CallReply`
[page 108] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(State, Arg1, ..., ArgN)` -> `CallReply`
[page 108] User defined function which is not a part of Orber

any

The following functions are exported:

- `create()` -> Result
[page 110] Create an any record
- `create(Typecode, Value)` -> Result
[page 110] Create an any record
- `set_typecode(A, Typecode)` -> Result
[page 110] Set the typecode field
- `get_typecode(A)` -> Result
[page 110] Fetch the typecode
- `set_value(A, Value)` -> Result
[page 111] Set the value field
- `get_value(A)` -> Result
[page 111] Fetch the value

corba

The following functions are exported:

- `create(Module, TypeID)` -> Object
[page 112] Create and start a new server object
- `create(Module, TypeID, Env)` -> Object
[page 112] Create and start a new server object
- `create(Module, TypeID, Env, Options1)` -> Object
[page 112] Create and start a new server object
- `create_link(Module, TypeID)` -> Object
[page 112] Create and start a new server object
- `create_link(Module, TypeID, Env)` -> Object
[page 112] Create and start a new server object
- `create_link(Module, TypeID, Env, Options2)` -> Reply
[page 112] Create and start a new server object
- `dispose(Object)` -> ok
[page 113] Stop a server object
- `create_nil_objref()` -> Object
[page 113] Stop a server object
- `create_subobject_key(Object, Key)` -> Result
[page 113] Add an Erlang term to a private key field
- `get_subobject_key(Object)` -> Result
[page 114] Fetch the contents of the private key field
- `get_pid(Object)` -> Result
[page 114] Get the process id from an object key
- `raise(Exception)`
[page 114] Generate an Erlang throw
- `resolve_initial_references(ObjectId)` -> Object
[page 114] Return the object reference for the given object id

- `add_initial_service(ObjectId, Object) -> boolean()`
[page 114] Add a new initial service and associate it with the given id
- `remove_initial_service(ObjectId) -> boolean()`
[page 115] Remove association between the given id and service
- `list_initial_services() -> [ObjectId]`
[page 115] Return a list of supported object id's
- `resolve_initial_references_remote(ObjectId, Address) -> Object`
[page 115] Return the object reference for the given object id
- `list_initial_services_remote(Address) -> [ObjectId]`
[page 115] Return a list of supported object id's
- `object_to_string(Object) -> IOR_string`
[page 116] Convert the object reference to the external string representation
- `string_to_object(IOR_string) -> Object`
[page 116] Convert the external string representation to an object reference
- `print_object(Data [, IoDevice]) -> ok | {'EXCEPTION', E}`
[page 116] Print the supplied object
- `add_alternate_iiop_address(Object, Host, Port) -> NewObject | {'EXCEPTION', E}`
[page 116] Add ALTERNATE_IIOB_ADDRESS component to the supplied local object
- `orb_init(KeyValueList) -> ok | {'EXIT', Reason}`
[page 116] Configure Orber before starting it

corba_object

The following functions are exported:

- `get_interface(Object) -> InterfaceDef`
[page 118] Fetch the interface description
- `is_nil(Object) -> boolean()`
[page 118] Return true, if the given object is a NIL object reference, otherwise false
- `is_a(Object, Logical_type_id) -> Return`
[page 118] Return true if the target object is an, or inherit from, object of the given type
- `is_remote(Object) -> boolean()`
[page 118] Determine whether or not an object reference is remote
- `non_existent(Object) -> Return`
[page 119] Return false if the target object do not exist, otherwise true
- `not_existent(Object) -> Return`
[page 119] Return false if the target object do not exist, otherwise true
- `is_equivalent(Object, OtherObject) -> boolean()`
[page 119] Return true if the target object and the supplied object easily can be determined to be equal, otherwise false
- `hash(Object, Maximum) -> int()`
[page 119] Return a hash value based on the target object

interceptors

The following functions are exported:

- `new_in_connection(Ref, Host, Port) -> NewRef`
[page 121] Invoke when a new client ORB wants to setup a connection
- `new_out_connection(Ref, Host, Port) -> NewRef`
[page 121] Invoke when setting up a new connection to a server side ORB
- `closed_in_connection(Ref) -> NewRef`
[page 121] Invoke when an existing connection to a client side ORB have been terminated
- `closed_out_connection(Ref) -> NewRef`
[page 121] Invoke when an existing connection to a server side ORB have been terminated
- `in_reply(Ref, Obj, Ctx, Op, Data, Extra) -> Reply`
[page 122] Invoke when replies arrives at the client side ORB
- `in_reply_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`
[page 122] Invoke when replies arrives at the client side ORB with undecoded reply body
- `in_request(Ref, Obj, Ctx, Op, Args, Extra) -> Reply`
[page 122] Invoke when requests arrive at the server side ORB
- `in_request_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`
[page 122] Invoke when requests arrive at the server side ORB with undecoded request body
- `out_reply(Ref, Obj, Ctx, Op, Data, Extra) -> Reply`
[page 123] Invoke after the target object replied
- `out_reply_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`
[page 123] Invoke after the target object replied with the reply encoded
- `out_request(Ref, Obj, Ctx, Op, Args, Extra) -> Reply`
[page 123] Invoke on the client side ORB before encoding and sending the request
- `out_request_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`
[page 123] Invoke on the client side ORB before sending the request

Iname

The following functions are exported:

- `create() -> Return`
[page 125] Create a new name
- `insert_component(Name, N, NameComponent) -> Return`
[page 125] Insert a new name component in a name
- `get_component(Name, N) -> Return`
[page 125] Get a name component from a name
- `delete_component(Name, N) -> Return`
[page 126] Delete a name component from a name
- `num_components(Name) -> Return`
[page 126] Count the number of name components in a name

- `equal(Name1, Name2) -> Return`
[page 126] Test if two names are equal
- `less_than(Name1, Name2) -> Return`
[page 126] Test if one name is lesser than the other
- `to_idl_form(Name) -> Return`
[page 126] Transform a pseudo name to an IDL name
- `from_idl_form(Name) -> Return`
[page 126] Transform an IDL name to a pseudo name

Iname_component

The following functions are exported:

- `create() -> Return`
[page 127] Create a new name component
- `get_id(NameComponent) -> Return`
[page 127] Get the id field of a name component
- `set_id(NameComponent, Id) -> Return`
[page 127] Set the id field of a name component
- `get_kind(NameComponent) -> Return`
[page 127] Get the kind field of a name component
- `set_kind(NameComponent, Kind) -> Return`
[page 128] Set the kind field of a name component

orber

The following functions are exported:

- `start() -> ok`
[page 129] Start the Orber application
- `start_lightweight() -> ok`
[page 129] Start the Orber application as lightweight
- `start_lightweight(Addresses) -> ok`
[page 129] Start the Orber application as lightweight
- `jump_start(Port) -> ok | {'EXIT', Reason}`
[page 129] Start the Orber application during tests
- `stop() -> ok`
[page 130] Stop the Orber application
- `is_lightweight() -> boolean()`
[page 130] Is the application started as lightweight?
- `get_lightweight_nodes() -> RemoteModifierList | false`
[page 130] Get the Remote Modifier list.
- `get_ORBInitRef() -> string() | undefined`
[page 130] Get the initial reference address.
- `get_ORBDefaultInitRef() -> string() | undefined`
[page 130] Get the initial reference address.
- `domain() -> string()`
[page 130] Display the Orber domain name

- `iiop_port() -> int()`
[page 130] Display the IIOP port number
- `iiop_out_ports() -> 0 | {Min, Max}`
[page 130] Display the ports Orber may use when connecting to another ORB
- `iiop_ssl_port() -> int()`
[page 130] Display the IIOP port number used for secure connections
- `iiop_timeout() -> int() (milliseconds)`
[page 131] Display the IIOP timeout value
- `iiop_connection_timeout() -> int() (milliseconds)`
[page 131] Display the IIOP connection timeout value
- `iiop_connections() -> [{Host, Port}] | {'EXIT', Reason}`
[page 131] List all existing connections to another ORB
- `iiop_connections_pending() -> [{Host, Port}] | {'EXIT', Reason}`
[page 131] List all connections to another ORB currently being set up
- `secure() -> no | ssl`
[page 131] Display the security mode Orber is running in
- `ssl_server_certfile() -> string()`
[page 131] Display the path to the server certificate
- `ssl_client_certfile() -> string()`
[page 132] Display the path to the client certificate
- `set_ssl_client_certfile(Path) -> ok`
[page 132] Set the value of the client certificate
- `ssl_server_verify() -> 0 | 1 | 2`
[page 132] Display the SSL verification type for incoming calls
- `ssl_client_verify() -> 0 | 1 | 2`
[page 132] Display the SSL verification type for outgoing calls
- `set_ssl_client_verify(Value) -> ok`
[page 132] Set the value of the SSL verification type for outgoing calls
- `ssl_server_depth() -> int()`
[page 132] Display the SSL verification depth for incoming calls
- `ssl_client_depth() -> int()`
[page 132] Display the SSL verification depth for outgoing calls
- `set_ssl_client_depth(Depth) -> ok`
[page 132] Sets the value of the SSL verification depth for outgoing calls
- `objectkeys_gc_time() -> int() (seconds)`
[page 132] Display the Object Keys GC time value
- `bootstrap_port() -> int()`
[page 133] Display the bootstrap protocol port number
- `orber_nodes() -> RetVal`
[page 133] Displays which nodes that this orber domain consist of.
- `install(NodeList) -> ok`
[page 133] Install the Orber application
- `install(NodeList, Options) -> ok`
[page 133] Install the Orber application
- `uninstall() -> ok`
[page 134] Uninstall the Orber application

- `add_node(Node, StorageType) -> RetVal`
[page 134] Add a new node to a group of Orber nodes.
- `remove_node(Node) -> RetVal`
[page 134] Removes a node from a group of Orber nodes.
- `configure(Key, Value) -> ok | {'EXIT', Reason}`
[page 135] Change Orber configuration.

orber_ifr

The following functions are exported:

- `init(Nodes, Timeout) -> ok`
[page 136] Initialize the IFR
- `find_repository() -> #IFR_Repository_objref`
[page 136] Find the IFR object reference for the Repository
- `get_def_kind(Objref) -> Return`
[page 137] Return the definition kind of the IFR object
- `destroy(Objref) -> Return`
[page 137] Destroy, except IRObj, Contained and Container, target object and its contents
- `get_id(Objref) -> Return`
[page 137] Return the target object's repository id
- `set_id(Objref, Id) -> ok`
[page 137] Set the target object's repository id
- `get_name(Objref) -> Return`
[page 137] Return the name of the target object
- `set_name(Objref, Name) -> ok`
[page 137] Set given name to target object
- `get_version(Objref) -> Return`
[page 138] Return the version of the target object
- `set_version(Objref, Version) -> ok`
[page 138] Set given version of the target object
- `get_defined_in(Objref) -> Return`
[page 138] Return the Container the target object is contained in
- `get_absolute_name(Objref) -> Return`
[page 138] Return the absolute name of the target object
- `get_containing_repository(Objref) -> Return`
[page 138] Get the most derived Contained object associated with the target object
- `describe(Objref) -> Return`
[page 138] Return a tuple which describe the target object
- `move(Objref, New_container, New_name, New_version) -> Return`
[page 139] Move the target object from its current location to given Container, name and version
- `lookup(Objref, Search_name) -> Return`
[page 139] Return the IFR object identified by the given name

- `contents(Objref,Limit_type,Exclude_inherited)` -> Return
[page 139] Return the content of the target object limited by the given constraints
- `lookup_name(Objref,Search_name,Levels_to_search, Limit_type, Exclude_inherited)` -> Return
[page 139] Return a list of IFR objects matching the given name
- `describe_contents(Objref,Limit_type,Exclude_inherited,Max_returned_objs)` -> Return
[page 140] Return a list of descriptions of the IFR objects contained by the target Container object
- `create_module(Objref,Id,Name,Version)` -> Return
[page 140] Create an IFR object of given type
- `create_constant(Objref,Id,Name,Version,Type,Value)` -> Return
[page 140] Create a ConstantDef IFR object
- `create_struct(Objref,Id,Name,Version,Members)` -> Return
[page 140] Create a StructDef IFR object
- `create_union(Objref,Id,Name,Version,Discriminator_type,Members)` -> Return
[page 141] Create a UnionDef IFR object
- `create_enum(Objref,Id,Name,Version,Members)` -> Return
[page 141] Create a EnumDef IFR object
- `create_alias(Objref,Id,Name,Version,Original_type)` -> Return
[page 141] Create a AliasDef IFR object
- `create_interface(Objref,Id,Name,Version,Base_interfaces)` -> Return
[page 141] Create a InterfaceDef IFR object
- `create_exception(Objref,Id,Name,Version,Members)` -> Return
[page 142] Create a ExceptionDef IFR object
- `get_type(Objref)` -> Return
[page 142] Return the typecode of the target object
- `lookup_id(Objref,Search_id)` -> Return
[page 142] Return the IFR object matching the given id
- `get_primitive(Objref,Kind)` -> Return
[page 142] Return a PrimitiveDef of the specified kind
- `create_string(Objref,Bound)` -> Return
[page 142] Create an IFR objref of the type StringDef
- `create_sequence(Objref,Bound,Element_type)` -> Return
[page 143] Create an IFR objref of the type SequenceDef
- `create_array(Objref,Length,Element_type)` -> Return
[page 143] Create an IFR objref of the type ArrayDef
- `create_idltype(Objref,Typecode)` -> Return
[page 143] Create an IFR objref of the type IDLType
- `get_type_def(Objref)` -> Return
[page 143] Return an IFR object of the type IDLType describing the type of the target object
- `set_type_def(Objref,TypeDef)` -> Return
[page 143] Set given TypeDef of the target object

- `get_value(Objref) -> Return`
[page 143] Return the value attribute of the target ConstantDef object
- `set_value(Objref, Value) -> Return`
[page 144] Set the value attribute of the target ConstantDef object
- `get_members(Objref) -> Return`
[page 144] Return the members of the target object
- `set_members(Objref, Members) -> Return`
[page 144] Set the members attribute of the target object
- `get_discriminator_type(Objref) -> Return`
[page 144] Get the discriminator typecode of the target object
- `get_discriminator_type_def(Objref) -> Return`
[page 144] Return IDLType object describing the discriminator type of the target object
- `set_discriminator_type_def(Objref, TypeDef) -> Return`
[page 145] Set the attribute discriminator_type_def for the target object to the given TypeDef
- `get_original_type_def(Objref) -> Return`
[page 145] Return an IFR object of the type IDLType describing the original type
- `set_original_type_def(Objref, TypeDef) -> Return`
[page 145] Set the original_type_def attribute which describes the original type
- `get_kind(Objref) -> Return`
[page 145] Return an atom describing the primitive type
- `get_bound(Objref) -> Return`
[page 145] Get the maximum size of the target object
- `set_bound(Objref, Bound) -> Return`
[page 145] Set the maximum size of the target object
- `get_element_type(Objref) -> Return`
[page 146] Return the typecode of the elements in the IFR object
- `get_element_type_def(Objref) -> Return`
[page 146] Return an IFR object of the type IDLType describing the type of the elements in Objref
- `set_element_type_def(Objref, TypeDef) -> Return`
[page 146] Set the element_type_def attribute of the target object to the given TypeDef
- `get_length(Objref) -> Return`
[page 146] Return the number of elements in the array
- `set_length(Objref, Length) -> Return`
[page 146] Set the number of elements in the array
- `get_mode(Objref) -> Return`
[page 146] Get the mode of the target object (AttributeDef or OperationDef)
- `set_mode(Objref, Mode) -> Return`
[page 147] Set the mode of the target object (AttributeDef or OperationDef) to the given mode
- `get_result(Objref) -> Return`
[page 147] Return typecode describing the type of the value returned by the operation

- `get_result_def(Objref)` -> Return
[page 147] Return an IFR object of the type IDLType describing the type of the result
- `set_result_def(Objref, ResultDef)` -> Return
[page 147] Set the type_def attribute of the target object to the given ResultDef
- `get_params(Objref)` -> Return
[page 147] Return a list of parameter description records describing the parameters of the target OperationDef
- `set_params(Objref, Params)` -> Return
[page 147] Set the params attribute of the target object to the given parameterdescription records
- `get_contexts(Objref)` -> Return
[page 148] Return a list of context identifiers for the operation
- `set_contexts(Objref, Contexts)` -> Return
[page 148] Set the context attribute for the operation
- `get_exceptions(Objref)` -> Return
[page 148] Return a list of exception types that can be raised by the target object
- `set_exceptions(Objref, Exceptions)` -> Return
[page 148] Set the exceptions attribute for the target object
- `get_base_interfaces(Objref)` -> Return
[page 148] Return a list of InterfaceDefs from which the target InterfaceDef object inherit
- `set_base_interfaces(Objref, BaseInterfaces)` -> Return
[page 148] Set the BaseInterfaces attribute
- `is_a(Objref, Interface_id)` -> Return
[page 148] Return a boolean if the target InterfaceDef match or inherit from the given id
- `describe_interface(Objref)` -> Return
[page 149] Return a full interface description record describing the InterfaceDef
- `create_attribute(Objref, Id, Name, Version, Type, Mode)` -> Return
[page 149] Create an IFR object of the type AttributeDef contained in the target InterfaceDef object
- `create_operation(Objref, Id, Name, Version, Result, Mode, Params, Exceptions, Contexts)` -> Return
[page 149] Create an IFR object of the type OperationDef contained in the target InterfaceDef object

orber_tc

The following functions are exported:

- `null()` -> TC
[page 150] Return the IDL typecode
- `void()` -> TC
[page 150] Return the IDL typecode
- `short()` -> TC
[page 150] Return the IDL typecode

- `unsigned_short()` -> TC
[page 150] Return the IDL typecode
- `long()` -> TC
[page 150] Return the IDL typecode
- `unsigned_long()` -> TC
[page 150] Return the IDL typecode
- `float()` -> TC
[page 150] Return the IDL typecode
- `double()` -> TC
[page 150] Return the IDL typecode
- `boolean()` -> TC
[page 150] Return the IDL typecode
- `char()` -> TC
[page 150] Return the IDL typecode
- `octet()` -> TC
[page 150] Return the IDL typecode
- `any()` -> TC
[page 150] Return the IDL typecode
- `typecode()` -> TC
[page 150] Return the IDL typecode
- `principal()` -> TC
[page 150] Return the IDL typecode
- `object_reference(Id, Name)` -> TC
[page 150] Return the object_reference IDL typecode
- `struct(Id, Name, ElementList)` -> TC
[page 150] Return the struct IDL typecode
- `union(Id, Name, DiscrTC, Default, ElementList)` -> TC
[page 151] Return the union IDL typecode
- `enum(Id, Name, ElementList)` -> TC
[page 151] Return the enum IDL typecode
- `string(Length)` -> TC
[page 151] Return the string IDL typecode
- `sequence(ElemTC, Length)` -> TC
[page 152] Return the sequence IDL typecode
- `array(ElemTC, Length)` -> TC
[page 152] Return the array IDL typecode
- `alias(Id, Name, AliasTC)` -> TC
[page 152] Return the alias IDL typecode
- `exception(Id, Name, ElementList)` -> TC
[page 152] Return the exception IDL typecode
- `get_tc(Object)` -> TC
[page 152] Fetch typecode
- `get_tc(Id)` -> TC
[page 152] Fetch typecode
- `check(TC)` -> `boolean()`
[page 153] Check syntax of an IDL typecode

CosNaming

Erlang Module

The naming service provides the principal mechanism for clients to find objects in an ORB based world. The naming service provides an initial naming context that functions as the root context for all names. Given this context clients can navigate in the name space.

Types that are declared on the CosNaming level are:

```
typedef string Istring;
struct NameComponent {
    Istring id;
    Istring kind;
};

typedef sequence <NameComponent> Name;

enum BindingType {nobject, ncontext};

struct Binding {
    Name      binding_name;
    BindingType binding_type;
};

typedef sequence <Binding> BindingList;
```

To get access to the record definitions for the structs use:

```
-include_lib("orber/COSS/CosNaming.hrl")..
```

Names are not an ORB object but they can be structured in components as seen by the definition above. There are no requirements on names so the service can support many different conventions and standards.

There are two different interfaces supported in the service:

- NamingContext
- BindingIterator

IDL specification for CosNaming:

```
// Naming Service v1.0 described in CORBAservices:
// Common Object Services Specification, chapter 3
// OMG IDL for CosNaming Module, p 3-6
```

```
#pragma prefix "omg.org"
```

```
module CosNaming
{
```

```

typedef string Istring;
struct NameComponent {
    Istring id;
    Istring kind;
};

typedef sequence <NameComponent> Name;

enum BindingType {nobject, ncontext};

struct Binding {
    Name    binding_name;
    BindingType binding_type;
};

typedef sequence <Binding> BindingList;

interface BindingIterator;
interface NamingContext;

interface NamingContext {

    enum NotFoundReason { missing_node, not_context, not_object};

    exception NotFound {
        NotFoundReason why;
        Name rest_of_name;
    };

    exception CannotProceed {
        NamingContext cxt;
        Name rest_of_name;
    };

    exception InvalidName{};
    exception AlreadyBound {};
    exception NotEmpty{};

    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);
    Object resolve (in Name n)
        raises(NotFound, CannotProceed, InvalidName);
    void unbind(in Name n)
        raises(NotFound, CannotProceed, InvalidName);
    NamingContext new_context();
    NamingContext bind_new_context(in Name n)

```

```
        raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
void destroy( )
    raises(NotEmpty);
void list (in unsigned long how_many,
          out BindingList bl,
          out BindingIterator bi);
};

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

CosNaming_BindingIterator

Erlang Module

This interface allows a client to iterate over the Bindinglist it has been initiated with.

The type NameComponent used below is defined as:

```
-record('CosNaming_NameComponent', {id, kind=""}).
```

id and kind are strings.

The type Binding used below is defined as:

```
-record('CosNaming_Binding', {binding_name, binding_type}).
```

binding_name is a Name = [NameComponent] and binding_type is an enum which has the values nobject and ncontext.

Both these records are defined in the file CosNaming.hrl and it is included with:

```
-include_lib("orber/COSS/CosNaming/CosNaming.hrl").
```

Exports

`next_one(BindinIterator) -> Return`

Types:

- BindingIterator = #objref
- Return = {bool(), Binding}

This operation returns the next binding. If there are no more bindings it returns false otherwise true.

`next_n(BindinIterator, HowMany) -> Return`

Types:

- BindingIterator = #objref
- HowMany = int()
- BindingList = [Binding]
- Return = {bool(), BindingList}

This operation returns a binding list with at most HowMany bindings. If there are no more bindings it returns false otherwise true.

`destroy(BindingIterator) -> Return`

Types:

- BindingIterator = #objref
- Return = ok

This operation destroys the binding iterator.

CosNaming_NamingContext

Erlang Module

This is the object that defines name scopes, names must be unique within a naming context. Objects may have multiple names and may exist in multiple naming contexts. Name context may be named in other contexts and cycles are permitted.

The type NameComponent used below is defined as:

```
-record('CosNaming_NameComponent', {id, kind=""}).
```

where id and kind are strings.

The type Binding used below is defined as:

```
-record('CosNaming_Binding', {binding_name, binding_type}).
```

where binding_name is a Name and binding_type is an enum which has the values nobject and ncontext.

Both these records are defined in the file CosNaming.hrl and it is included with:

```
-include_lib("orber/COSS/CosNaming/CosNaming.hrl").
```

There are a number of exceptions that can be returned from functions in this interface.

- NotFound is defined as

```
-record('CosNaming_NamingContext_NotFound',
        {rest_of_name, why}).
```

- CannotProceed is defined as

```
-record('CosNaming_NamingContext_CannotProceed',
        {rest_of_name, cxt}).
```

- InvalidName is defined as

```
-record('CosNaming_NamingContext_InvalidName', {}).
```

- NotFound is defined as

```
-record('CosNaming_NamingContext_NotFound', {}).
```

- AlreadyBound is defined as

```
-record('CosNaming_NamingContext_AlreadyBound', {}).
```

- NotEmpty is defined as

```
-record('CosNaming_NamingContext_NotEmpty', {}).
```

These exceptions are defined in the file CosNaming_NamingContext.hrl and it is included with:

```
-include_lib("orber/COSS/CosNaming/CosNaming_NamingContext.hrl").
```

Exports

`bind(NamingContext, Name, Object) -> Return`

Types:

- `NameContext = #objref`
- `Name = [NameComponent]`
- `Object = #objref`
- `Return = ok`

Creates a binding of a name and an object in the naming context. Naming contexts that are bound using *bind()* do not participate in name resolution.

`rebind(NamingContext, Name, Object) -> Return`

Types:

- `NamingContext = #objref`
- `Name = [NameComponent]`
- `Object = #objref`
- `Return = ok`

Creates a binding of a name and an object in the naming context even if the name is already bound. Naming contexts that are bound using *rebind()* do not participate in name resolution.

`bind_context(NamingContext1, Name, NamingContext2) -> Return`

Types:

- `NamingContext1 = NamingContext2 = #objref`
- `Name = [NameComponent]`
- `Return = ok`

The *bind_context* function creates a binding of a name and a naming context in the current context. Naming contexts that are bound using *bind_context()* participate in name resolution.

`rebind_context(NamingContext1, Name, NamingContext2) -> Return`

Types:

- `NamingContext1 = NamingContext2 = #objref`
- `Name = [NameComponent]`
- `Return = ok`

The *rebind_context* function creates a binding of a name and a naming context in the current context even if the name already is bound. Naming contexts that are bound using *rebind_context()* participate in name resolution.

`resolve(NamingContext, Name) -> Return`

Types:

- `NamingContext = #objref`
- `Name = [NameComponent]`
- `Return = Object`

- Object = #objref

The resolve function is the way to retrieve an object bound to a name in the naming context. The given name must match exactly the bound name. The type of the object is not returned, clients are responsible for narrowing the object to the correct type.

`unbind(NamingContext, Name) -> Return`

Types:

- NamingContext = #objref
- Name = [NameComponent]
- Return = ok

The unbind operation removes a name binding from the naming context.

`new_context(NamingContext) -> Return`

Types:

- NamingContext = #objref
- Return = #objref

The new_context operation creates a new naming context.

`bind_new_context(NamingContext, Name) -> Return`

Types:

- NamingContext = #objref
- Name = [NameComponent]
- Return = #objref

The new_context operation creates a new naming context and binds it to Name in the current context.

`destroy(NamingContext) -> Return`

Types:

- NamingContext = #objref
- Return = ok

The destroy operation disposes the NamingContext object and removes it from the name server. The context must be empty e.g. not contain any bindings to be removed.

`list(NamingContext, HowMany) -> Return`

Types:

- NamingContext = #objref
- HowMany = int()
- Return = {ok, BindingList, BindingIterator}
- BindingList = [Binding]
- BindingIterator = #objref

The list operation returns a BindingList with a number of bindings upto HowMany from the context. It also returns a BindingIterator which can be used to step through the list.

Note that one must remove the BindingIterator with a 'BindingIterator'.destroy() otherwise one can get dangling objects.

CosNaming_NamingContextExt

Erlang Module

To get access to the record definitions for the structures use:

```
-include_lib("orber/COSS/CosNaming/CosNaming.hrl").
```

This module also exports the functions described in:

- CosNaming_NamingContext [page 99]

Exports

`to_string(NamingContext, Name) -> Return`

Types:

- NameContext = #objref
- Name = [NameComponent]
- Return = string() | {'EXCEPTION', NamingContext::InvalidName{}}

Stringifies a Name sequence to a string.

`to_name(NamingContext, NameString) -> Return`

Types:

- NameContext = #objref
- NameString = string()
- Return = [NameComponent] | {'EXCEPTION', NamingContext::InvalidName{}}

Converts a stringified Name to a Name sequence.

`to_url(NamingContext, AddressString, NameString) -> Return`

Types:

- NameContext = #objref
- Address = NameString = string()
- Return = URLString | {'EXCEPTION', NamingContext::InvalidName{}} | {'EXCEPTION', NamingContextExt::InvalidAddress{}}

This operation takes a corbaloc string and a stringified Name sequence as input and returns a fully formed URL string.

`resolve_str(NamingContext, NameString) -> Return`

Types:

- NameContext = #objref

- NameString = string()
- Return = #objref | {'EXCEPTION', NamingContext::InvalidName{}} |
{'EXCEPTION', NamingContext::NotFound{why, rest_of_name}} |
{'EXCEPTION', NamingContext::CannotProceed{cxt, rest_of_name}}

This operation takes a stringified Name sequence as input and returns the associated, if any, object.

Module_Interface

Erlang Module

This module contains the stub/skeleton functions generated by IC.

Starting a Orber server can be done in three ways:

- Normal - when the server dies Orber forgets all knowledge of the server.
- Supervisor child - adding the configuration parameter `{sup_child, true}` the `oe_create_link/2` function returns `{ok, Pid, ObjRef}` which can be handled by the application *supervisor/stdlib-1.7* or later.
- Persistent object reference - adding the configuration parameters `{persistent, true}` and `{regname, {global, term()}}` Orber will remember the object reference until the server terminates with reason *normal* or *shutdown*. Hence, if the server is started as a *transient* supervisor child we do not receive a 'OBJECT_NOT_EXIST' exception when it has crashed and is being restarted.

The Orber stub can be used to start a pseudo object, which will create a non-server implementation. A pseudo object introduce some limitations:

- The functions `oe_create_link/2` is equal to `oe_create/2`, i.e., no link can or will be created.
- The BIF:s `self()` and `process_flag(trap_exit, true)` behaves incorrectly.
- The IC option `{{impl, "M::I"}, "other_impl"}` has no effect. The call-back functions must be implemented in a file called `M_I_impl.erl`
- The call-back functions must be implemented as if the IC option `{this, "M::I"}` was used.
- Server State changes have no effect. The user can provide information via the `Env` start parameter and the State returned from `init/2` will be the State passed in following invocations.
- If a call-back function replies with the `Timeout` parameter set it have no effect.
- Operations defined as `oneway` are blocking until the operation replies.
- The option `{pseudo, true}` overrides all other start options.
- Only the functions, besides own definitions, `init/2` (called via `oe_create*/2`) and `terminate/2` (called via `corba:dispose/1`) must be implemented.

By adopting the rules for pseudo objects described above we can use `oe_create/2` to create server or pseudo objects, by excluding or including the option `{pseudo, true}`, without changing the call-back module.

If you start a object without `{regname, RegName}` it can only be accessed through the returned object key. Started with a `{regname, RegName}` the name is registered locally or globally.

Warning:

To avoid flooding Orber with old object references start erlang using the flag *-orber objectkeys_gc_time Time*, which will remove all object references related to servers being dead for *Time* seconds. To avoid extra overhead, i.e., performing garbage collect if no persistent objects are started, the *objectkeys_gc_time* default value is *infinity*. For more information, see the orber and corba documentation.

Exports

`typeID() -> TypeId`

Types:

- `TypeId = string()`, e.g., "IDL:Module/Interface:1.0"

Returns the Type ID related to this stub/skeleton

`oe_create() -> ObjRef`

Types:

- `ObjRef = #object reference`

Start a Orber server.

`oe_create_link() -> ObjRef`

Types:

- `ObjRef = #object reference`

Start a linked Orber server.

`oe_create(Env) -> ObjRef`

Types:

- `Env = term()`
- `ObjRef = #object reference`

Start a Orber server passing *Env* to *init/1*.

`oe_create_link(Env) -> ObjRef`

Types:

- `Env = term()`
- `ObjRef = #object reference`

Start a linked Orber server passing *Env* to *init/1*.

`oe_create(Env, Options) -> ObjRef`

Types:

- `Env = term()`
- `ObjRef = #object reference`

- Options = [{sup_child, false} | {persistent, Bool} | {regname, RegName} | {pseudo, Bool}]
- Bool = true | false
- RegName = {global, term()} | {local, atom()}

Start a Orber server passing Env to init/1.

If the option {pseudo, true} is used, all other options are overridden. As default, this option is set to false.

This function cannot be used for starting a server as supervisor child. If started as persistent, the options [{persistent, true}, {regname, {global, term()}}] must be used and Orber will only forget the object reference if it terminates with reason *normal* or *shutdown*.

oe_create_link(Env, Options) -> Return

Types:

- Env = term()
- Return = ObjRef | {ok, Pid, ObjRef}
- ObjRef = #object reference
- Options = [{sup_child, Bool} | {persistent, Bool} | {regname, RegName} | {pseudo, Bool}]
- Bool = true | false
- RegName = {global, term()} | {local, atom()}

Start a linked Orber server passing Env to init/1.

If the option {pseudo, true} is used, all other options are overridden and no link will be created. As default, this option is set to false.

This function can be used for starting a server as persistent or supervisor child. At the moment [{persistent, true}, {regname, {global, term()}}] must be used to start a server as persistent, i.e., if a server died and is in the process of being restarted a call to the server will not raise 'OBJECT_NOT_EXIST' exception. Orber will only forget the object reference if it terminates with reason *normal* or *shutdown*, hence, the server must be started as *transient* (for more information see the supervisor documentation).

Module_Interface:own_functions(ObjRef, Arg1, ..., ArgN) -> Reply

Types:

- ObjRef = #object reference
- ArgX = specified in the IDL-code.
- Reply = specified in the IDL-code.

If the configuration parameter {timeout, "Module::Interface"} is *not* passed to IC this function must be called when invoking an operation.

Module_Interface:own_functions(ObjRef, Timeout, Arg1, ..., ArgN) -> Reply

Types:

- ObjRef = #object reference
- Timeout = int() >= 0 | infinity
- ArgX = specified in the IDL-code.
- Reply = specified in the IDL-code.

If the configuration parameter `{timeout, "Module::Interface"}` is passed to IC this function must be called when invoking an operation.

`Module_Interface_impl:init(Env) -> CallReply`

Types:

- `Env = term()`
- `CallReply = {ok, State} | {ok, State, Timeout} | ignore | {stop, StopReason}`
- `State = term()`
- `Timeout = int() >= 0 | infinity`
- `StopReason = term()`

Whenever a new server is started, *init/1* is the first function called in the specified call-back module.

`Module_Interface_impl:terminate(Reason, State) -> ok`

Types:

- `Reason = term()`
- `State = term()`

This call-back function is called whenever the server is about to terminate.

`Module_Interface_impl:code_change(OldVsn, State, Extra) -> CallReply`

Types:

- `OldVsn = undefined | term()`
- `State = term()`
- `Extra = term()`
- `CallReply = {ok, NewState}`
- `NewState = term()`

Update the internal State.

`Module_Interface_impl:handle_info(Info, State) -> CallReply`

Types:

- `Info = term()`
- `State = term()`
- `CallReply = {noreply, State} | {noreply, State, Timeout} | {stop, StopReason, State}`
- `Timeout = int() >= 0 | infinity`
- `StopReason = normal | shutdown | term()`

If the configuration parameter `{{handle_info, "Module::Interface"}, true}` is passed to IC and *process_flag(trap_exit, true)* is set in the *init()* call-back this function must be exported.

Note:

To be able to handle the `Timeout` option in `CallReply` in the call-back module the configuration parameter `{{handle_info, "Module::Interface"}, true}` must be passed to IC.

`Module_Interface_impl:own_functions(This, State, Arg1, ..., ArgN) -> CallReply`

Types:

- This = the servers #object reference
- State = term()
- ArgX = specified in the IDL-code.
- CallReply = {reply, Reply, State} | {reply, Reply, State, Timeout} | {noreply, State} | {noreply, State, Timeout} | {stop, StopReason, Reply, State} | {stop, StopReason, State}
- Reply = specified in the IDL-code.
- Timeout = int() >= 0 | infinity
- StopReason = normal | shutdown | term()

If the configuration parameter {*this*, "*Module::Interface*"} is passed to IC and the function is defined to be two-way this function must be exported.

`Module_Interface_impl:own_functions(State, Arg1, ..., ArgN) -> CallReply`

Types:

- State = term()
- CallReply = {reply, Reply, State} | {reply, Reply, State, Timeout} | {noreply, State} | {noreply, State, Timeout} | {stop, StopReason, Reply, State} | {stop, StopReason, State}
- ArgX = specified in the IDL-code.
- Reply = specified in the IDL-code.
- Timeout = int() >= 0 | infinity
- StopReason = normal | shutdown | term()

If the configuration parameter {*this*, "*Module::Interface*"} is *not* passed to IC and the function is defined to be two-way this function must be exported.

`Module_Interface_impl:own_functions(This, State, Arg1, ..., ArgN) -> CallReply`

Types:

- This = the servers #object reference
- State = term()
- CallReply = {noreply, State} | {noreply, State, Timeout} | {stop, StopReason, State}
- ArgX = specified in the IDL-code.
- Reply = specified in the IDL-code.
- Timeout = int() >= 0 | infinity
- StopReason = normal | shutdown | term()

If the configuration parameter {*this*, "*Module::Interface*"} is passed to IC and the function is defined to be one-way this function must be exported.

`Module_Interface_impl:own_functions(State, Arg1, ..., ArgN) -> CallReply`

Types:

- State = term()
- CallReply = {noreply, State} | {noreply, State, Timeout} | {stop, StopReason, State}
- ArgX = specified in the IDL-code.
- Reply = specified in the IDL-code.

- Timeout = int() >= 0 | infinity
- StopReason = normal | shutdown | term()

If the configuration parameter `{this, "Module::Interface"}` is *not* passed to IC and the function is defined to be one-way this function must be exported.

any

Erlang Module

This module contains functions that gives an interface to the CORBA any type.

Note that the any interface in orber does not contain a destroy function because the any type is represented as an Erlang record and therefor will be removed by the garbage collector when not in use.

The type TC used below describes an IDL type and is a tuple according to the to the Erlang language mapping.

The type Any used below is defined as:

```
-record(any, {typecode, value}).
```

where typecode is a TC tuple and value is an Erlang term of the type defined by the typecode field.

Exports

```
create() -> Result
```

```
create(Typecode, Value) -> Result
```

Types:

- Typecode = TC
- Value = term()
- Result = Any

The create/0 function creates an empty any record and the create/2 function creates an initialized record.

```
set_typecode(A, Typecode) -> Result
```

Types:

- A = Any
- Typecode = TC
- Result = Any

This function sets the typecode of A and returns a new any record.

```
get_typecode(A) -> Result
```

Types:

- A = Any
- Result = TC

This function returns the typecode of *A*.

`set_value(A, Value) -> Result`

Types:

- *A* = Any
- *Value* = `term()`
- *Result* = Any

This function sets the value of *A* and returns a new any record.

`get_value(A) -> Result`

Types:

- *A* = Any
- *Result* = `term()`

This function returns the value of *A*.

corba

Erlang Module

This module contains functions that are specified on the CORBA module level. It also contains some functions for creating and disposing objects.

Exports

```
create(Module, TypeID) -> Object
create(Module, TypeID, Env) -> Object
create(Module, TypeID, Env, Options1) -> Object
create_link(Module, TypeID) -> Object
create_link(Module, TypeID, Env) -> Object
create_link(Module, TypeID, Env, Options2) -> Reply
```

Types:

- Module = atom()
- TypeID = string()
- Env = term()
- Options1 = [{persistent, Bool} | {regname, RegName}]
- Options2 = [{sup_child, Bool} | {persistent, Bool} | {regname, RegName} | {pseudo, Bool}]
- RegName = {local, atom()} | {global, term()}
- Reply = #objref | {ok, Pid, #objref}
- Bool = true | false
- Object = #objref

These functions start a new server object. If you start it without *RegName* it can only be accessed through the returned object key. Started with a *RegName* the name is registered locally or globally.

TypeID is the repository ID of the server object type and could for example look like "IDL:StackModule/Stack:1.0".

Module is the name of the interface API module.

Env is the arguments passed which will be passed to the implementations *init* call-back function.

A server started with *create/2*, *create/3* or *create/4* does not care about the parent, which means that the parent is not handled explicitly in the generic process part.

A server started with *create_link/2*, *create_link/3* or *create_link/4* is initially linked to the caller, the parent, and it will terminate whenever the parent process terminates, and with the same reason as the parent. If the server traps exits, the *terminate/2* call-back

function is called in order to clean up before the termination. These functions should be used if the server is a worker in a supervision tree.

If you use the option `{sup_child, true}` `create_link/4` will return `{ok, Pid, #objref}`, otherwise `#objref`, and make it possible to start a server as a supervisor child (stdlib-1.7 or later).

If you use the option `{persistent, true}` you also must use the option `{regname, {global, Name}}`. This combination makes it possible to tell the difference between a server permanently terminated or in the process of restarting.

The option `{pseudo, true}`, allow us to create an object which is not a server. Using `{pseudo, true}` overrides all other start options. For more information see section `Module_Interface`.

If a server is started using the option `{persistent, true}` the object key will not be removed unless it terminates with reason *normal* or *shutdown*. Hence, if persistent servers is used as supervisor childs they should be *transient* and the *objectkeys_gc_time* should be modified (default equals infinity).

Example:

```
corba:create('StackModule_Stack', "IDL:StackModule/Stack:1.0",
            {10, test})
```

`dispose(Object) -> ok`

Types:

- Object = `#objref`

This function is used for terminating the execution of a server object. Invoking this operation on a NIL object reference, e.g., the return value of `corba:create_nil_objref/0`, always return ok. For valid object references, invoking this operation more than once, will result in a system exception.

`create_nil_objref() -> Object`

Types:

- Object = `#objref` representing NIL.

Creates an object reference that represents the NIL value. Attempts to invoke operations using the returned object reference will return a system exception.

`create_subobject_key(Object, Key) -> Result`

Types:

- Object = `#objref`
- Key = `term()`
- Result = `#objref`

This function is used to create a subobject in a server object. It can for example be useful when one wants unique access to separate rows in a mnesia or an ETS table. The *Result* is an object reference that will be seen as a unique reference to the outside world but will access the same server object where one can use the *get_subobject_key/1* function to get the private key value.

Key is stored in the object reference *Object*. If it is a binary it will be stored as is and otherwise it is converted to a binary before storage.

```
get_subobject_key(Object) -> Result
```

Types:

- Object = #objref
- Result = #binary

This function is used to fetch a subobject key from the object reference *Object*. The result is always a binary, if it was an Erlang term that was stored with *create_subobject_key/2* one can do *binary_to_term/1* to get the real value.

```
get_pid(Object) -> Result
```

Types:

- Object = #objref
- Result = #pid | {error, Reason} | {'EXCEPTION', E}

This function is to get the process id from an object, which is a must when CORBA objects is started/handled in a supervisor tree. The function will throw exceptions if the key is not found or some other error occurs.

```
raise(Exception)
```

Types:

- Exception = record()

This function is used for raising corba exceptions as an Erlang user generated exit signal. It will throw the tuple {'EXCEPTION', *Exception*}.

```
resolve_initial_references(ObjectId) -> Object
```

Types:

- ObjectId = string()
- Object = #objref

This function returns the object reference associated with the given object id. Initially, only "NameService" is available. To add or remove services use *add_initial_service/2* or *remove_initial_service/1*.

```
add_initial_service(ObjectId, Object) -> boolean()
```

Types:

- ObjectId = string()
- Object = #objref

This operation allows us to add initial services, which can be accessed by using `resolve_initial_references/1` or the `corbaloc` schema. If using an Id defined by the OMG, the given object must be of the correct type; for more information see the Interoperable Naming Service [page 26]. Returns `false` if the given id already exists.

```
remove_initial_service(ObjectId) -> boolean()
```

Types:

- `ObjectId = string()`

If we don not want a certain service to be accessible, invoking this function will remove the association. Returns `true` if able to terminate the binding. If no such binding existed `false` is returned.

```
list_initial_services() -> [ObjectId]
```

Types:

- `ObjectId = string()`

This function returns a list of allowed object id's.

```
resolve_initial_references_remote(ObjectId, Address) -> Object
```

Types:

- `Address = [RemoteModifier]`
- `RemoteModifier = string()`
- `ObjectId = string()`
- `Object = #objref`

This function returns the object reference for the object id asked for. The remote modifier string has the following format: `"iiop://host:port"`.

Warning:

This operation is not supported by most ORB's. Hence, use `corba:string_to_object/1` instead.

```
list_initial_services_remote(Address) -> [ObjectId]
```

Types:

- `Address = [RemoteModifier]`
- `RemoteModifier = string()`
- `ObjectId = string()`

This function returns a list of allowed object id's. The remote modifier string has the following format: `"iiop://host:port"`.

Warning:

This operation is not supported by most ORB's. Hence, avoid using it.

`object_to_string(Object) -> IOR_string`

Types:

- Object = #objref
- IOR_string = string()

This function returns the object reference as the external string representation of an IOR.

`string_to_object(IOR_string) -> Object`

Types:

- IOR_string = string()
- Object = #objref

This function takes a corbaname, corbaloc or an IOR on the external string representation and returns the object reference.

To lookup the NameService reference, simply use

"corbaloc:iiop:1.2@123.456.789.012:4001/NameService"

We can also resolve an object from the NameService by using

"corbaname:iiop:1.2@123.456.789.012:4001/NameService#org/Erlang/MyObj"

For more information about corbaname and corbaloc, see the User's Guide (Interoperable Naming Service).

`print_object(Data [, IoDevice]) -> ok | {'EXCEPTION', E}`

Types:

- Data = IOR_string | #objref (local or external) | corbaloc/corbaname string
- IoDevice = see the io-module

The object represented by the supplied data is dissected and presented in a more readable form. The Io-device is optional; if not supplied standard output is used.

If the supplied object is a local reference, the output is equivalent to an object exported from the node this function is invoked on.

`add_alternate_iiop_address(Object, Host, Port) -> NewObject | {'EXCEPTION', E}`

Types:

- Object = NewObject = local #objref
- Host = string()
- Port = integer()

This operation creates a new instance of the supplied object containing an ALTERNATE_IIOP_ADDRESS component. Only the new instance contains the new component. When this object is passed to another ORB, which supports the ALTERNATE_IIOP_ADDRESS, requests will be routed to the alternate address if it is not possible to communicate with the main address.

The ALTERNATE_IIOP_ADDRESS component requires that IIOP-1.2 is used. Hence, make sure both Orber and the other ORB is correctly configured.

`orb_init(KeyValueList) -> ok | {'EXIT', Reason}`

Types:

- KeyValueList = [{Key, Value}]
- Key = orbDefaultInitRef | orbInitRef | giop_version | iiop_timeout | iiop_connection_timeout | iiop_setup_connection_timeout | objectkeys_gc_time | orber_debug_level | iiop_port | interceptors | domain | ip_address | iiop_out_ports
- Value = allowed value associated with the given key

This function allows the user to configure Orber in, for example, an Erlang shell. Orber may *NOT* be started prior to invoking this operation. For more information, see configuration settings [page 11] in the User's Guide.

corba_object

Erlang Module

This module contains the CORBA Object interface functions that can be called for all objects.

Exports

`get_interface(Object) -> InterfaceDef`

Types:

- Object = #objref
- InterfaceDef = term()

This function returns the full interface description for an object.

`is_nil(Object) -> boolean()`

Types:

- Object = #objref

This function checks if the object reference has a nil object value, which denotes no object. It is the reference that is tested and no object implementation is involved in the test.

`is_a(Object, Logical_type_id) -> Return`

Types:

- Object = #objref
- Logical_type_id = string()

The *Logical_type_id* is a string that is a share type identifier (repository id). The function returns true if the object is an instance of that type or an ancestor of the “most derived” type of that object.

Note: Other ORB suppliers may not support this function completely according to the OMG specification. Thus, a *is_a* call may raise an exception or respond unpredictable if the Object is located on a remote node.

`is_remote(Object) -> boolean()`

Types:

- Object = #objref

This function returns true if an object reference is remote otherwise false.

`non_existent(Object) -> Return`

Types:

- Object = #objref
- Return = boolean() | {EXCEPTION, _}

This function can be used to test if the object has been destroyed. It does this without invoking any application level code. The ORB returns true if it knows that the object is destroyed otherwise false.

Note: The OMG have specified two different operators, `_not_existent` (CORBA version 2.0 and 2.2) and `_non_existent` (CORBA version 2.3), to be used for this function. It is not mandatory to support both versions. Thus, a *non_existent* call may raise an exception or respond unpredictable if the Object is located on a remote node. Depending on which version, ORB:s you intend to communicate with supports, you can either use this function or `not_existent/1`.

`not_existent(Object) -> Return`

Types:

- Object = #objref
- Return = boolean() | {EXCEPTION, _}

This function is implemented due to Interoperable purposes. Behaves as `non_existent` except the operator `_not_existent` is used when communicating with other ORB:s.

`is_equivalent(Object, OtherObject) -> boolean()`

Types:

- Object = #objref
- OtherObject = #objref

This function is used to determine if two object references are equivalent so far the ORB easily can determine. It returns *true* if the target object reference is equal to the other object reference and *false* otherwise.

`hash(Object, Maximum) -> int()`

Types:

- Object = #objref
- Maximum = int()

This function returns a hash value based on the object reference that not will change during the lifetime of the object. The *Maximum* parameter denotes the upper bound of the value.

interceptors

Erlang Module

This module contains the mandatory functions for user supplied native interceptors and their intended behaviour. See also the User's Guide.

Warning:

Using `Interceptors` may reduce the through-put significantly if the supplied interceptors invoke expensive operations. Hence, one should always supply interceptors which cause as little overhead as possible.

Warning:

It is possible to alter the `Data`, `Bin` and `Args` parameter for the `in_reply` and `out_reply`, `in_reply_encoded`, `in_request_encoded`, `out_reply_encoded` and `out_request_encoded`, `in_request` and `out_request` respectively. But, if it is done incorrectly, the consequences can be serious.

Note:

The `Extra` parameter is set to 'undefined' by Orber when calling the first interceptor and may be set to any erlang term. If an interceptor change this parameter it will be passed on to the next interceptor in the list uninterpreted.

Note:

The `Ref` parameter is set to 'undefined' by Orber when calling `new_in_connection` or `new_out_connection` using the first interceptor. The user supplied interceptor may set `NewRef` to any erlang term. If an interceptor change this parameter it will be passed on to the next interceptor in the list uninterpreted.

Exports

`new_in_connection(Ref, Host, Port) -> NewRef`

Types:

- Ref = term() | undefined
- Host = string(), e.g., "myHost@myServer" or "012.345.678.910"
- Port = integer
- NewRef = term() | {'EXIT', Reason}

When a new connection is requested by a client side ORB this operation is invoked. If more than one interceptor is supplied, e.g., {native, ['myInterceptor1', 'myInterceptor2']}, the return value from 'myInterceptor1' is passed to 'myInterceptor2' as Ref. Initially, Orber uses the atom 'undefined' as Ref parameter when calling the first interceptor. The return value from the last interceptor, in the example above 'myInterceptor2', is passed to all other functions exported by the interceptors. Hence, the Ref parameter can, for example, be used as a unique identifier to mnesia or ets where information/restrictions for this connection is stored.

The Host and Port variables supplied is the peer data of the client ORB which requested a new connection.

If, for some reason, we do not allow the client ORB to connect simply invoke `exit(Reason)`.

`new_out_connection(Ref, Host, Port) -> NewRef`

Types:

- Ref = term() | undefined
- Host = string(), e.g., "myHost@myServer" or "012.345.678.910"
- Port = integer
- NewRef = term() | {'EXIT', Reason}

When a new connection is set up this function is invoked. Behaves just like `new_in_connection`; the only difference is that the Host and Port variables identifies the target ORB's bootstrap data.

`closed_in_connection(Ref) -> NewRef`

Types:

- Ref = term()
- NewRef = term()

When an existing connection is terminated this operation is invoked. The main purpose of this function is to make it possible for a user to clean up all data associated with the associated connection.

The input parameter Ref is the return value from `new_in_connection/3`.

`closed_out_connection(Ref) -> NewRef`

Types:

- Ref = term()
- NewRef = term()

When an existing connection is terminated this operation is invoked. The main purpose of this function is to make it possible for a user to clean up all data associated with the associated connection.

The input parameter Ref is the return value from `new_out_connection/3`.

`in_reply(Ref, Obj, Ctx, Op, Data, Extra) -> Reply`

Types:

- Ref = term()
- Obj = #objref
- Ctx = [#'IOP_ServiceContext'{}]
- Op = atom()
- Data = [Result, OutParameter1, ..., OutParameterN]
- Reply = {NewData, NewExtra}

When replies are delivered from the server side ORB to the client side ORB this operation is invoked. The Data parameter is a list in which the first element is the return value from the target object and the rest is a list of all parameters defined as out or inout in the IDL-specification.

`in_reply_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`

Types:

- Ref = term()
- Obj = #objref
- Ctx = [#'IOP_ServiceContext'{}]
- Op = atom()
- Bin = #binary
- Reply = {NewBin, NewExtra}

When replies are delivered from the server side ORB to the client side ORB this operation is invoked. The Bin parameter is the reply body still uncoded.

`in_request(Ref, Obj, Ctx, Op, Args, Extra) -> Reply`

Types:

- Ref = term()
- Obj = #objref
- Ctx = [#'IOP_ServiceContext'{}]
- Op = atom()
- Args = [Argument] - defined in the IDL-specification
- Reply = {NewArgs, NewExtra}

When a new request arrives at the server side ORB this operation is invoked.

`in_request_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`

Types:

- Ref = term()
- Obj = #objref
- Ctx = [#'IOP_ServiceContext'{}]
- Op = atom()

- Bin = #binary
- Reply = {NewBin, NewExtra}

When a new request arrives at the server side ORB this operation is invoked before decoding the request body.

`out_reply(Ref, Obj, Ctx, Op, Data, Extra) -> Reply`

Types:

- Ref = term()
- Obj = #objref
- Ctx = [#'IOP_ServiceContext' {}]
- Op = atom()
- Data = [Result, OutParameter1, ..., OutParameterN]
- Reply = {NewData, NewExtra}

After the target object have been invoked this operation is invoked with the result. The Data parameter is a list in which the first element is the return value from the target object and the rest is a all parameters defined as out or inout in the IDL-specification.

`out_reply_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`

Types:

- Ref = term()
- Obj = #objref
- Ctx = [#'IOP_ServiceContext' {}]
- Op = atom()
- Bin = #binary
- Reply = {NewBin, NewExtra}

This operation is similar to `out_reply`; the only difference is that the reply body have been encoded.

`out_request(Ref, Obj, Ctx, Op, Args, Extra) -> Reply`

Types:

- Ref = term()
- Obj = #objref
- Ctx = [#'IOP_ServiceContext' {}]
- Op = atom()
- Args = [Argument] - defined in the IDL-specification
- Reply = {NewArgs, NewExtra}

Before a request is sent to the server side ORB, `out_request` is invoked.

`out_request_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`

Types:

- Ref = term()
- Obj = #objref
- Ctx = [#'IOP_ServiceContext' {}]

- Op = atom()
- Bin = #binary
- Reply = {NewBin, NewExtra}

This operation is similar to `out_request`; the only difference is that the request body have been encoded.

Iname

Erlang Module

This interface is a part of the names library which is used to hide the representation of names. In orbers Erlang mapping the pseudo-object names and the real IDL names have the same representation but it is desirable that the clients uses the names library so they will not be dependent of the representation. The Iname interface supports handling of names e.g. adding and removing name components.

Note that the Iname interface in orber does not contain a destroy function because the Names are represented as standard Erlang lists and therefor will be removed by the garbage collector when not in use.

The type NameComponent used below is defined as:

```
-record('CosNaming_NameComponent', {id, kind=""}).
```

id and kind are strings.

The record is defined in the file CosNaming.hrl and it is included with:

```
-include_lib("orber/COSS/CosNaming/CosNaming.hrl").
```

Exports

`create()` -> Return

Types:

- Return = [NameComponent]

This function returns a new name.

`insert_component(Name, N, NameComponent)` -> Return

Types:

- Name = [NameComponent]
- N = int()
- Return = Name

This function returns a name where the new name component has been inserted as component N in Name.

`get_component(Name, N)` -> Return

Types:

- Name = [NameComponent]
- N = int()
- Return = NameComponent

This function returns the N :th name component in Name.

`delete_component(Name, N) -> Return`

Types:

- Name = [NameComponent]
- N = int()
- Return = Name

This function deletes the N :th name component from Name and returns the new name.

`num_components(Name) -> Return`

Types:

- Name = [NameComponent]
- Return = int()

This function returns a the number of name components in Name.

`equal(Name1, Name2) -> Return`

Types:

- Name1 = Name2 = [NameComponent]
- Return = bool()

This function returns true if the two names are equal and false otherwise.

`less_than(Name1, Name2) -> Return`

Types:

- Name1 = Name2 = [NameComponent]
- Return = bool()

This function returns true if Name1 are lesser than Name2 and false otherwise.

`to_idl_form(Name) -> Return`

Types:

- Name = [NameComponent]
- Return = Name

This function just checks if Name is a correct IDL name before returning it because the name representation is the same for pseudo and IDL names in orber.

`from_idl_form(Name) -> Return`

Types:

- Name = [NameComponent]
- Return = Name

This function just returns the Name because the name representation is the same for pseudo and IDL names in orber.

lname_component

Erlang Module

This interface is a part of the name library, which is used to hide the representation of names. In orbers Erlang mapping the pseudo-object names and the real IDL names have the same representation but it is desirable that the clients uses the names library so they will not be dependent of the representation. The lname_component interface supports handling of name components e.g. set and get of the struct members.

Note that the lname_component interface in orber does not contain a destroy function because the NameComponents are represented as Erlang records and therefor will be removed by the garbage collector when not in use.

The type NameComponent used below is defined as:

```
-record('CosNaming_NameComponent', {id, kind=""}).
```

id and kind are strings.

The record is defined in the file CosNaming.hrl and it is included with:

```
-include_lib("orber/COSS/CosNaming/CosNaming.hrl").
```

Exports

`create()` -> Return

Types:

- Return = NameComponent

This function returns a new name component.

`get_id(NameComponent)` -> Return

Types:

- Return = string()

This function returns the id string of a name component.

`set_id(NameComponent, Id)` -> Return

Types:

- Id = string()
- Return = NameComponent

This function sets the id string of a name component and returns the component.

`get_kind(NameComponent)` -> Return

Types:

- Return = string()

This function returns the id string of a name component.

`set_kind(NameComponent, Kind) -> Return`

Types:

- Kind = string()
- Return = NameComponent

This function sets the kind string of a name component and returns the component.

orber

Erlang Module

This module contains the functions for starting and stopping the application. It also has some utility functions to get some of the configuration information from running application.

Exports

`start() -> ok`

Starts the Orber application (it also starts mnesia if it is not running).

`start_lightweight() -> ok`

Starts the Orber application as lightweight.

Preconditions:

- Erlang started on the node using the option `-orber lightweight`, e.g., `erl -orber lightweight` Addresses.
- The `Addresses` must be a list of `RemoteModifiers`, equal to the `orber:resolve_initial_references_remote/2` argument. The list must contain Orber nodes addresses, to which we have access and are not started as lightweight.

`start_lightweight(Addresses) -> ok`

Types:

- `Addresses = [Address]`
- `Address =`
- `RetVal = ok | exit()`

Starts the Orber application as lightweight.

Preconditions:

- If Erlang is started using the configuration parameter `-orber lightweight`, e.g., `erl -orber lightweight Address`, the argument supplied to this function will override the configuration parameter. Hence, this function must be used carefully.
- The `Addresses` must be a list of `RemoteModifiers`, equal to the `orber:resolve_initial_references_remote/2` argument. The list must contain Orber nodes addresses, to which we have access and are not started as lightweight.

`jump_start(Port) -> ok | {'EXIT', Reason}`

Types:

- Port = integer()

Installs and starts the Orber and the Mnesia applications with the configuration parameters `domain` and `iiop_port` set to "IP-number:Port" and the supplied Port respectively. These settings are in most cases sufficient to ensure that no clash with any other Orber instance occur. If this operation fails, check if the listen port (`iiop_port`) is already in use. This function *MAY ONLY* be used during development and tests; how Orber is configured when using this operation may change at any time without warning.

`stop() -> ok`

Stops the Orber application.

`is_lightweight() -> boolean()`

This function returns the true if Orber is started as lightweight, false otherwise.

`get_lightweight_nodes() -> RemoteModifierList | false`

This function returns false if Orber is not started as lightweight, otherwise a list of Remote Modifiers.

`get_ORBInitRef() -> string() | undefined`

This function returns undefined if we will resolve references locally, otherwise a string describing which host we will contact if the Key given to `corba:resolve_initial_references/1` matches the Key set in this configuration variable. For more information see the user's guide.

`get_ORBDefaultInitRef() -> string() | undefined`

This function returns undefined if we will resolve references locally, otherwise a string describing which host, or hosts, from which we will try to resolve the Key given to `corba:resolve_initial_references/1`. For more information see the user's guide.

`domain() -> string()`

This function returns the domain name of the current Orber domain as a string.

`iiop_port() -> int()`

This function returns the port-number, which is used by the IIOP protocol. It can be configured by setting the application variable *iiop_port*, if it is not set it will have the default number 4001.

`iiop_out_ports() -> 0 | {Min, Max}`

The return value of this operation is what the configuration parameter `iiop_out_ports` [page 11] have been set to.

`iiop_ssl_port() -> int()`

This function returns the port-number, which is used by the secure IIOP protocol. It can be configured by setting the application variable *iiop_ssl_port*, if it is not set it will have the default number 4002 if Orber is configured to run in secure mode. Otherwise it returns -1.

`iiop_timeout() -> int()` (milliseconds)

This function returns the timeout value after which outgoing IIOP requests terminate. It can be configured by setting the application variable *iiop_timeout TimeVal (seconds)*, if it is not set it will have the default value *infinity*. If a request times out a *COMM_FAILURE* exception is raised.

Note: the *iiop_timeout* configuration parameter (TimeVal) may only range between 0 and 1000000 seconds. Otherwise, the default value is used.

Note: IC supply the compile option `ic:gen(IdlFile, [{timeout, "module::interface"}])`, which allow the user to add an extra timeout parameter, e.g., `module_interface(ObjRef, Timeout, ... Arguments ...)`, instead of `module_interface(ObjRef, ... Arguments ...)`. If a stub is compiled with the *timeout* option, the extra *Timeout* argument will override the configuration parameter *iiop_timeout*. It is, however, not possible to use *infinity* to override the *Timeout* parameter. The *Timeout* option is also valid for objects which resides within the same Orber domain.

`iiop_connection_timeout() -> int()` (milliseconds)

This function returns the timeout value after which outgoing IIOP connections terminate. It can be configured by setting the application variable *iiop_connection_timeout TimeVal (seconds)*, if it is not set it will have the default value *infinity*. The connection will not be terminated if there are pending requests.

Note: the *iiop_connection_timeout* configuration parameter (TimeVal) may only range between 0 and 1000000 seconds. Otherwise, the default value is used.

`iiop_connections() -> [{Host, Port}] | {'EXIT', Reason}`

The list returned by this operation contain tuples of remote hosts/ports Orber is currently connected to. If no such connections exists, the list will be empty.

`iiop_connections_pending() -> [{Host, Port}] | {'EXIT', Reason}`

In some cases a connection attempt (i.e. trying to communicate with another ORB) may block due to a number of reasons. This operation allows the user to check if this is the case. The returned list contain tuples of remote hosts/ports. Normally, the list is empty.

`secure() -> no | ssl`

This function returns the security mode Orber is running in, which is either *no* if it is an insecure domain or the type of security mechanism used. For the moment the only security mechanism is *ssl*. This is configured by setting the application variable *secure*.

`ssl_server_certfile() -> string()`

This function returns a path to a file containing a chain of PEM encoded certificates for the Orber domain as server. This is configured by setting the application variable *ssl_server_certfile*.

`ssl_client_certfile() -> string()`

This function returns a path to a file containing a chain of PEM encoded certificates used in outgoing calls in the current process. The default value is configured by setting the application variable *ssl_client_certfile*.

`set_ssl_client_certfile(Path) -> ok`

Types:

- Path = string()

This function takes a path to a file containing a chain of PEM encoded certificates as parameter and sets it for the current process.

`ssl_server_verify() -> 0 | 1 | 2`

This function returns the type of verification used by SSL during authentication of the other peer for incoming calls. It is configured by setting the application variable *ssl_server_verify*.

`ssl_client_verify() -> 0 | 1 | 2`

This function returns the type of verification used by SSL during authentication of the other peer for outgoing calls. The default value is configured by setting the application variable *ssl_client_verify*.

`set_ssl_client_verify(Value) -> ok`

Types:

- Value = 0 | 1 | 2

This function sets the SSL verification type for the other peer of outgoing calls.

`ssl_server_depth() -> int()`

This function returns the SSL verification depth for incoming calls. It is configured by setting the application variable *ssl_server_depth*.

`ssl_client_depth() -> int()`

This function returns the SSL verification depth for outgoing calls. The default value is configured by setting the application variable *ssl_client_depth*.

`set_ssl_client_depth(Depth) -> ok`

Types:

- Depth = int()

This function sets the SSL verification depth for the other peer of outgoing calls.

`objectkeys_gc_time() -> int() (seconds)`

This function returns the timeout value after which terminated object keys, related to servers started with the configuration parameter `{persistent, true}`, will be removed. It can be configured by setting the application variable `objectkeys_gc_time TimeVal (seconds)`, if it is not set it will have the default value *infinity*.

Objects terminating with reason *normal* or *shutdown* are removed automatically.

Note: the `objectkeys_gc_time` configuration parameter (TimeVal) may only range between 0 and 1000000 seconds. Otherwise, the default value is used.

`bootstrap_port() -> int()`

This function returns the port-number, which is used by the CORBA bootstrapping protocol. This protocol is used to fetch an initial reference from another ORB. It can be configured by setting the application variable `bootstrap_port`, if it is not set it will use the `iiop` port.

Note: In the future it will use the port number which is set in the standard (the suggestion is 900). Because the standard is not ready in this area we in the meantime uses a port number, which do not require root permissions in Unix.

`orber_nodes() -> RetVal`

Types:

- RetVal = [node()]

This function returns the list of node names that this orber domain consists of.

`install(NodeList) -> ok`

`install(NodeList, Options) -> ok`

Types:

- NodeList = [node()]
- Options = [Option]
- Option = {install_timeout, Timeout} | {ifr_storage_type, TableType} | {nameservice_storage_type, TableType}
- Timeout = infinity | integer()
- TableType = disc_copies | ram_copies

This function installs all the necessary mnesia tables and load default data in some of them. If one or more Orber tables already exists the installation fails. The function *uninstall* may be used, if it is safe, i.e., no other application is running Orber.

Preconditions:

- a mnesia schema must exist before the installation
- mnesia is running on the other nodes if the new installation shall be a multi node domain

Mnesia will be started by the function if it is not already running on the installation node and if it was started it will be stopped afterwards.

The options that can be sent to the installation program is:

- {install_timeout, Timeout} - this timeout is how long we will wait for the tables to be created. The Timeout value can be *infinity* or an integer number in milliseconds. Default is infinity.

- `{ifr_storage_type, TableType}` - this option sets the type of tables used for the interface repository. The `TableType` can be `disc_copies` or `ram_copies`. Default is `disc_copies`.
- `{nameservice_storage_type, TableType}` - the default behavior of Orber is to install the `NameService` as `ram_copies`. This option makes it possible to change this to `disc_copies`. But the user should be aware of that if a node is restarted, all local object references stored in the `NameService` is not valid. Hence, you cannot switch to `disc_copies` and expect exactly the same behavior as before.

`uninstall()` -> `ok`

This function stops the Orber application, terminates all server objects and removes all Orber related mnesia tables.

Note: Since other applications may be running on the same node using mnesia *uninstall* will not stop the mnesia application.

`add_node(Node, StorageType)` -> `RetVal`

Types:

- `Node = node()`
- `StorageType = disc_copies | ram_copies`
- `RetVal = ok | exit()`

This function add given node to a existing Orber node group and starts Orber on the new node. `orber:add_node` is called from a member in the Orber node group.

Preconditions for new node:

- Erlang started on the new node using the option `-mnesia extra_db_nodes`, e.g., `erl -sname new_node_name -mnesia extra_db_nodes ConnectToNodes_List`
- The new node's domain name is the same for the nodes we want to connect to.
- Mnesia is running on the new node (no new schema created).
- If the new node will use `disc_copies` the schema type must be changed using: `mnesia:change_table_copy_type(schema, node(), disc_copies)`.

Orber will be started by the function on the new node.

Fails if:

- Orber already installed on given node.
- Mnesia not started as described above on the new node.
- Impossible to copy data in Mnesia tables to the new node.
- Not able to start Orber on the new node, due to, for example, the `iiop_port` is already in use.

The function do not remove already copied tables after a failure. Use `orber:remove_node` to remove these tables.

`remove_node(Node)` -> `RetVal`

Types:

- `Node = node()`

- RetVal = ok | exit()

This function removes given node from a Orber node group. The Mnesia application is not stopped.

```
configure(Key, Value) -> ok | {'EXIT', Reason}
```

Types:

- Key = orbDefaultInitRef | orbInitRef | giop_version | iiop_timeout | iiop_connection_timeout | iiop_setup_connection_timeout | objectkeys_gc_time | orber_debug_level | iiop_port | interceptors | domain | ip_address | iiop_out_ports
- Value = allowed value associated with the given key

This function allows the user to configure Orber in, for example, an Erlang shell. It is possible to invoke `configure` at any time for most of the specified keys above except:

- iiop_port
- interceptors
- domain
- ip_address
- iiop_out_ports

The Keys listed above may only be set before installing and starting Orber.

Trying to change the configuration in any other way is *NOT* allowed since it may affect the behavior of Orber.

For more information regarding allowed values, see configuration settings [page 11] in the User's Guide.

Note:

Configuring the IIOP timeout values will not affect already existing connections. If you want a guaranteed uniform behavior, you must set these parameters from the start.

orber_ifr

Erlang Module

This module contains functions for managing the Interface Repository (IFR). This documentation should be used in conjunction with the documentation in chapter 6 of *CORBA 2.0*. Whenever the term IFR object is used in this manual page, it refers to a pseudo object used only for interaction with the IFR rather than a CORBA object.

Initialisation of the IFR

The following functions are used to initialise the Interface Repository and to obtain the initial reference to the repository.

Exports

```
init(Nodes,Timeout) -> ok
```

Types:

- Nodes = list()
- Timeout = integer() | infinity

This function should be called to initialise the IFR. It creates the necessary mnesia-tables. A mnesia schema should exist, and mnesia must be running.

```
find_repository() -> #IFR_Repository_objref
```

Find the IFR object reference for the Repository. This reference should be used when adding objects to the IFR, and when extracting information from the IFR. The first time this function is called, it will create the repository and all the primitive definitions.

General methods

The following functions are the methods of the IFR. The first argument is always an #IFR_objref, i.e. the IFR (pseudo)object on which to apply this method. These functions are useful when the type of IFR object is not known, but they are somewhat slower than the specific functions listed below which only accept a particular type of IFR object as the first argument.

Exports

`get_def_kind(Objref) -> Return`

Types:

- `Objref = #IFR_objref`
- `Return = atom()` (one of `dk_none`, `dk_all`, `dk_Attribute`, `dk_Constant`, `dk_Exception`, `dk_Interface`, `dk_Module`, `dk_Operation`, `dk_Typedef`, `dk_Alias`, `dk_Struct`, `dk_Union`, `dk_Enum`, `dk_Primitive`, `dk_String`, `dk_Sequence`, `dk_Array`, `dk_Repository`)

`Objref` is an IFR object of any kind. Returns the definition kind of the IFR object.

`destroy(Objref) -> Return`

Types:

- `Objref = #IFR_object`
- `Return = tuple()`

`Objref` is an IFR object of any kind except `IRObj`, `Contained` and `Container`. Destroys that object and its contents (if any). Returns whatever `mnesia:transaction` returns.

`get_id(Objref) -> Return`

Types:

- `Objref = #IFR_object`
- `Return = string()`

`Objref` is an IFR object of any kind that inherits from `Contained`. Returns the repository id of that object.

`set_id(Objref,Id) -> ok`

Types:

- `Objref = #IFR_object`
- `Id = string()`

`Objref` is an IFR object of any kind that inherits from `Contained`. Sets the repository id of that object.

`get_name(Objref) -> Return`

Types:

- `Objref = #IFR_object`
- `Return = string()`

`Objref` is an IFR object of any kind that inherits from `Contained`. Returns the name of that object.

`set_name(Objref,Name) -> ok`

Types:

- `Objref = #IFR_object`
- `Name = string()`

Objref is an IFR object of any kind that inherits from Contained. Sets the name of that object.

`get_version(Objref) -> Return`

Types:

- Objref = #IFR_object
- Return = string()

Objref is an IFR object of any kind that inherits from Contained. Returns the version of that object.

`set_version(Objref,Version) -> ok`

Types:

- Objref = #IFR_object
- Version = string()

Objref is an IFR object of any kind that inherits from Contained. Sets the version of that object.

`get_defined_in(Objref) -> Return`

Types:

- Objref = #IFR_object
- Return = #IFR_Container_objref

Objref is an IFR object of any kind that inherits from Contained. Returns the Container object that the object is defined in.

`get_absolute_name(Objref) -> Return`

Types:

- Objref = #IFR_object
- Return = string()

Objref is an IFR object of any kind that inherits from Contained. Returns the absolute (scoped) name of that object.

`get_containing_repository(Objref) -> Return`

Types:

- Objref = #IFR_object
- Return = #IFR_Repository_objref

Objref is an IFR object of any kind that inherits from Contained. Returns the Repository that is eventually reached by recursively following the object's defined_in attribute.

`describe(Objref) -> Return`

Types:

- Objref = #IFR_object
- Return = tuple() (a contained_description record) | {exception, _}

Objref is an IFR object of any kind that inherits from Contained. Returns a tuple describing the object.

`move(Objref, New_container, New_name, New_version) -> Return`

Types:

- `Objref = #IFR_objref`
- `New_container = #IFR_Container_objref`
- `New_name = string()`
- `New_version = string()`
- `Return = ok | {exception, _}`

Objref is an IFR object of any kind that inherits from Contained. `New_container` is an IFR object of any kind that inherits from Container. Removes Objref from its current Container, and adds it to `New_container`. The name attribute is changed to `New_name` and the version attribute is changed to `New_version`.

`lookup(Objref, Search_name) -> Return`

Types:

- `Objref = #IFR_objref`
- `Search_name = string()`
- `Return = #IFR_object`

Objref is an IFR object of any kind that inherits from Container. Returns an IFR object identified by `search_name` (a scoped name).

`contents(Objref, Limit_type, Exclude_inherited) -> Return`

Types:

- `Objref = #IFR_objref`
- `Limit_type = atom()` (one of `dk_none`, `dk_all`, `dk_Attribute`, `dk_Constant`, `dk_Exception`, `dk_Interface`, `dk_Module`, `dk_Operation`, `dk_Typedef`, `dk_Alias`, `dk_Struct`, `dk_Union`, `dk_Enum`, `dk_Primitive`, `dk_String`, `dk_Sequence`, `dk_Array`, `dk_Repository`)
- `Exclude_inherited = atom()` (true or false)
- `Return = list()` (a list of IFR#_objects)

Objref is an IFR object of any kind that inherits from Container. Returns the contents of that IFR object.

`lookup_name(Objref, Search_name, Levels_to_search, Limit_type, Exclude_inherited) -> Return`

Types:

- `Objref = #IFR_objref`
- `Search_name = string()`
- `Levels_to_search = integer()`
- `Limit_type = atom()` (one of `dk_none`, `dk_all`, `dk_Attribute`, `dk_Constant`, `dk_Exception`, `dk_Interface`, `dk_Module`, `dk_Operation`, `dk_Typedef`, `dk_Alias`, `dk_Struct`, `dk_Union`, `dk_Enum`, `dk_Primitive`, `dk_String`, `dk_Sequence`, `dk_Array`, `dk_Repository`)
- `Exclude_inherited = atom()` (true or false)

- Return = list() (a list of #IFR_objects)

Objref is an IFR object of any kind that inherits from Container. Returns a list of #IFR_objects with an id matching Search_name.

`describe_contents(Objref, Limit_type, Exclude_inherited, Max_returned_objs) -> Return`

Types:

- Objref = #IFR_objref
- Limit_type = atom() (one of dk_none, dk_all, dk_Attribute, dk_Constant, dk_Exception, dk_Interface, dk_Module, dk_Operation, dk_Typedef, dk_Alias, dk_Struct, dk_Union, dk_Enum, dk_Primitive, dk_String, dk_Sequence, dk_Array, dk_Repository)
- Exclude_inherited = atom() (true or false)
- Return = list() (a list of tuples (contained_description records) | {exception, _})

Objref is an IFR object of any kind that inherits from Container. Returns a list of descriptions of the IFR objects in this Container's contents.

`create_module(Objref, Id, Name, Version) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Return = #IFR_ModuleDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type ModuleDef.

`create_constant(Objref, Id, Name, Version, Type, Value) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Type = #IFR_IDLType_objref
- Value = any()
- Return = #IFR_ConstantDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type ConstantDef.

`create_struct(Objref, Id, Name, Version, Members) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Members = list() (list of structmember records)

- Return = #IFR_StructDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type StructDef.

```
create_union(Objref,Id,Name,Version,Discriminator_type,Members) -> Return
```

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Discriminator_type = #IFR_IDLType_Objref
- Members = list() (list of unionmember records)
- Return = #IFR_UnionDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type UnionDef.

```
create_enum(Objref,Id,Name,Version,Members) -> Return
```

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Members = list() (list of strings)
- Return = #IFR_EnumDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type EnumDef.

```
create_alias(Objref,Id,Name,Version,Original_type) -> Return
```

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Original_type = #IFR_IDLType_Objref
- Return = #IFR_AliasDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type AliasDef.

```
create_interface(Objref,Id,Name,Version,Base_interfaces) -> Return
```

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()

- Base_interfaces = list() (a list of IFR_InterfaceDef_objrefs that this interface inherits from)
- Return = #IFR_InterfaceDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type InterfaceDef.

`create_exception(Objref,Id,Name,Version,Members) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Members = list() (list of structmember records)
- Return = #IFR_ExceptionDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type ExceptionDef.

`get_type(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = tuple() (a typecode tuple)

Objref is an IFR object of any kind that inherits from IDLType or an IFR object of the kind ConstantDef, ExceptionDef or AttributeDef. Returns the typecode of the IFR object.

`lookup_id(Objref,Search_id) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Search_id = string()
- Return = #IFR_objref

Returns an IFR object matching the Search_id.

`get_primitive(Objref,Kind) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Kind = atom() (one of pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong, pk_float, pk_double, pk_boolean, pk_char, pk_octet, pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref)
- Return = #IFR_PrimitiveDef_objref

Returns a PrimitiveDef of the specified kind.

`create_string(Objref,Bound) -> Return`

Types:

- Objref = #IFR_Repository_objref

- Bound = integer() (unsigned long != 0)
 - Return = #IFR_StringDef_objref
- Creates an IFR objref of the type StringDef.

`create_sequence(Objref,Bound,Element_type) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Bound = integer() (unsigned long)
- Element_type = #IFR_IDLType_objref
- Return = #IFR_SequenceDef_objref

Creates an IFR objref of the type SequenceDef.

`create_array(Objref,Length,Element_type) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Bound = integer() (unsigned long)
- Element_type = #IFR_IDLType_objref
- Return = #IFR_ArrayDef_objref

Creates an IFR objref of the type ArrayDef.

`create_idltype(Objref,Typecode) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Typecode = tuple() (a typecode tuple)
- Return = #IFR_IDLType_objref

Creates an IFR objref of the type IDLType.

`get_type_def(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = #IFR_IDLType_objref

Objref is an IFR object of the kind ConstantDef or AttributeDef. Returns an IFR object of the type IDLType describing the type of the IFR object.

`set_type_def(Objref,TypeDef) -> Return`

Types:

- Objref = #IFR_objref
- TypeDef = #IFR_IDLType_objref
- Return = ok | {exception, _}

Objref is an IFR object of the kind ConstantDef or AttributeDef. Sets the type_def of the IFR Object.

`get_value(Objref) -> Return`

Types:

- Objref = #IFR_ConstantDef_objref
- Return = any()

Returns the value attribute of an IFR Object of the type ConstantDef.

`set_value(Objref, Value) -> Return`

Types:

- Objref = #IFR_ConstantDef_objref
- Value = any()
- Return = ok | {exception, _}

Sets the value attribute of an IFR Object of the type ConstantDef.

`get_members(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = list()

Objref is an IFR object the kind StructDef, UnionDef, EnumDef or ExceptionDef. For StructDef, UnionDef and ExceptionDef: Returns a list of structmember records that are the constituent parts of the object. For EnumDef: Returns a list of strings describing the enumerations.

`set_members(Objref, Members) -> Return`

Types:

- Objref = #IFR_objref
- Members = list()
- Return = ok | {exception, _}

Objref is an IFR object the kind StructDef, UnionDef, EnumDef or ExceptionDef. For StructDef, UnionDef and ExceptionDef: Members is a list of structmember records. For EnumDef: Members is a list of strings describing the enumerations. Sets the members attribute, which are the constituent parts of the exception.

`get_discriminator_type(Objref) -> Return`

Types:

- Objref = #IFR_UnionDef_objref
- Return = tuple() (a typecode tuple)

Returns the discriminator typecode of an IFR object of the type UnionDef.

`get_discriminator_type_def(Objref) -> Return`

Types:

- Objref = #IFR_UnionDef_objref
- Return = #IFR_IDLType_objref

Returns an IFR object of the type IDLType describing the discriminator type of an IFR object of the type UnionDef.

`set_discriminator_type_def(Objref,TypeDef) -> Return`

Types:

- `Objref = #IFR_UnionDef_objref`
- `Return = #IFR_IDLType_objref`

Sets the attribute `discriminator_type_def`, an IFR object of the type `IDLType` describing the discriminator type of an IFR object of the type `UnionDef`.

`get_original_type_def(Objref) -> Return`

Types:

- `Objref = #IFR_AliasDef_objref`
- `Return = #IFR_IDLType_objref`

Returns an IFR object of the type `IDLType` describing the original type.

`set_original_type_def(Objref,TypeDef) -> Return`

Types:

- `Objref = #IFR_AliasDef_objref`
- `Typedef = #IFR_IDLType_objref`
- `Return = ok | {exception, _}`

Sets the `original_type_def` attribute which describes the original type.

`get_kind(Objref) -> Return`

Types:

- `Objref = #IFR_PrimitiveDef_objref`
- `Return = atom()`

Returns an atom describing the primitive type (See CORBA 2.0 p 6-21).

`get_bound(Objref) -> Return`

Types:

- `Objref = #IFR_objref`
- `Return = integer (unsigned long)`

`Objref` is an IFR object the kind `StringDef` or `SequenceDef`. For `StringDef`: returns the maximum number of characters in the string. For `SequenceDef`: Returns the maximum number of elements in the sequence. Zero indicates an unbounded sequence.

`set_bound(Objref,Bound) -> Return`

Types:

- `Objref = #IFR_objref`
- `Bound = integer (unsigned long)`
- `Return = ok | {exception, _}`

`Objref` is an IFR object the kind `StringDef` or `SequenceDef`. For `StringDef`: Sets the maximum number of characters in the string. `Bound` must not be zero. For `SequenceDef`: Sets the maximum number of elements in the sequence. Zero indicates an unbounded sequence.

`get_element_type(Objref) -> Return`

Types:

- `Objref = #IFR_objref`
- `Return = tuple()` (a typecode tuple)

`Objref` is an IFR object the kind `SequenceDef` or `ArrayDef`. Returns the typecode of the elements in the IFR object.

`get_element_type_def(Objref) -> Return`

Types:

- `Objref = #IFR_objref`
- `Return = #IFR_IDLType_objref`

`Objref` is an IFR object the kind `SequenceDef` or `ArrayDef`. Returns an IFR object of the type `IDLType` describing the type of the elements in `Objref`.

`set_element_type_def(Objref,TypeDef) -> Return`

Types:

- `Objref = #IFR_objref`
- `TypeDef = #IFR_IDLType_objref`
- `Return = ok | {exception, _}`

`Objref` is an IFR object the kind `SequenceDef` or `ArrayDef`. Sets the `element_type_def` attribute, an IFR object of the type `IDLType` describing the type of the elements in `Objref`.

`get_length(Objref) -> Return`

Types:

- `Objref = #IFR_ArrayDef_objref`
- `Return = integer()` (unsigned long)

Returns the number of elements in the array.

`set_length(Objref,Length) -> Return`

Types:

- `Objref = #IFR_ArrayDef_objref`
- `Length = integer()` (unsigned long)

Sets the number of elements in the array.

`get_mode(Objref) -> Return`

Types:

- `Objref = #IFR_objref`
- `Return = atom()`

`Objref` is an IFR object the kind `AttributeDef` or `OperationDef`. For `AttributeDef`: Return is an atom ('ATTR_NORMAL' or 'ATTR_READONLY') specifying the read/write access for this attribute. For `OperationDef`: Return is an atom ('OP_NORMAL' or 'OP_ONEWAY') specifying the mode of the operation.

`set_mode(Objref,Mode) -> Return`

Types:

- `Objref = #IFR_objref`
- `Mode = atom()`
- `Return = ok | {exception, _}`

`Objref` is an IFR object the kind `AttributeDef` or `OperationDef`. For `AttributeDef`: Sets the read/write access for this attribute. `Mode` is an atom (`'ATTR_NORMAL'` or `'ATTR_READONLY'`). For `OperationDef`: Sets the mode of the operation. `Mode` is an atom (`'OP_NORMAL'` or `'OP_ONEWAY'`).

`get_result(Objref) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `Return = tuple()` (a typecode tuple)

Returns a typecode describing the type of the value returned by the operation.

`get_result_def(Objref) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `Return = #IFR_IDLType_objref`

Returns an IFR object of the type `IDLType` describing the type of the result.

`set_result_def(Objref,ResultDef) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `ResultDef = #IFR_IDLType_objref`
- `Return = ok | {exception, _}`

Sets the `type_def` attribute, an IFR Object of the type `IDLType` describing the result.

`get_params(Objref) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `Return = list()` (list of parameter description records)

Returns a list of parameter description records, which describes the parameters of the `OperationDef`.

`set_params(Objref,Params) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `Params = list()` (list of parameterdescription records)
- `Return = ok | {exception, _}`

Sets the `params` attribute, a list of parameterdescription records.

`get_contexts(Objref) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `Return = list()` (list of strings)

Returns a list of context identifiers for the operation.

`set_contexts(Objref,Contexts) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `Contexts = list()` (list of strings)
- `Return = ok | {exception, _}`

Sets the context attribute for the operation.

`get_exceptions(Objref) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `Return = list()` (list of `#IFR_ExceptionDef_objrefs`)

Returns a list of exception types that can be raised by this operation.

`set_exceptions(Objref,Exceptions) -> Return`

Types:

- `Objref = #IFR_OperationDef_objref`
- `Exceptions = list()` (list of `#IFR_ExceptionDef_objrefs`)
- `Return = ok | {exception, _}`

Sets the exceptions attribute for this operation.

`get_base_interfaces(Objref) -> Return`

Types:

- `Objref = #IFR_InterfaceDef_objref`
- `Return = list()` (list of `#IFR_InterfaceDef_objrefs`)

Returns a list of InterfaceDefs from which this InterfaceDef inherits.

`set_base_interfaces(Objref,BaseInterfaces) -> Return`

Types:

- `Objref = #IFR_InterfaceDef_objref`
- `BaseInterfaces = list()` (list of `#IFR_InterfaceDef_objrefs`)
- `Return = ok | {exception, _}`

Sets the BaseInterfaces attribute.

`is_a(Objref,Interface_id) -> Return`

Types:

- `Objref = #IFR_InterfaceDef_objref`

- Interface_id = #IFR_InterfaceDef_objref
- Return = atom() (true or false)

Returns true if the InterfaceDef either is identical to or inherits from Interface_id.

`describe_interface(Objref) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref
- Return = tuple() (a fullinterfacedescription record)

Returns a full inter face description record describing the InterfaceDef.

`create_attribute(Objref,Id,Name,Version,Type,Mode) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref
- Id = string()
- Name = string()
- Version = string()
- Type = #IFR_IDLType_objref
- Mode = atom() ('ATTR_NORMAL' or 'ATTR_READONLY')
- Return = #IFR_AttributeDef_objref

Creates an IFR object of the type AttributeDef contained in this InterfaceDef.

`create_operation(Objref,Id,Name,Version,Result,Mode,Params, Exceptions,Contexts) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref
- Id = string()
- Name = string()
- Version = string()
- Result = #IFR_IDLType_objref
- Mode = atom() ('OP_NORMAL' or 'OP_ONEWAY')
- Params = list() (list of parameterdescription records)
- Exceptions = list() (list of #IFR_ExceptionDef_objrefs)
- Contexts = list() (list of strings)
- Return = #IFR_OperationDef_objref

Creates an IFR object of the type OperationDef contained in this InterfaceDef.

orber_tc

Erlang Module

This module contains some functions that gives support in creating IDL typecodes that can be used in for example the any types typecode field. For the simple types it is meaningless to use this API but the functions exist to get the interface complete.

The type TC used below describes an IDL type and is a tuple according to the to the Erlang language mapping.

Exports

```
null() -> TC
void() -> TC
short() -> TC
unsigned_short() -> TC
long() -> TC
unsigned_long() -> TC
float() -> TC
double() -> TC
boolean() -> TC
char() -> TC
octet() -> TC
any() -> TC
typecode() -> TC
principal() -> TC
```

These functions return the IDL typecodes for simple types.

```
object_reference(Id, Name) -> TC
```

Types:

- Id = string()
the repository ID
- Name = string()
the type name of the object

Function returns the IDL typecode for object_reference.

```
struct(Id, Name, ElementList) -> TC
```

Types:

- `Id = string()`
the repository ID
- `Name = string()`
the type name of the struct
- `ElementList = [{MemberName, TC}]`
a list of the struct elements
- `MemberName = string()`
the element name

Function returns the IDL typecode for struct.

`union(Id, Name, DiscrTC, Default, ElementList) -> TC`

Types:

- `Id = string()`
the repository ID
- `Name = string()`
the type name of the union
- `DiscrTC = TC`
the typecode for the unions discriminant
- `Default = integer()`
a value that indicates which tuple in the element list that is default (value < 0 means no default)
- `ElementList = [{Label, MemberName, TC}]`
a list of the union elements
- `Label = term()`
the label value should be of the *DiscrTC* type
- `MemberName = string()`
the element name

Function returns the IDL typecode for union.

`enum(Id, Name, ElementList) -> TC`

Types:

- `Id = string()`
the repository ID
- `Name = string()`
the type name of the enum
- `ElementList = [MemberName]`
a list of the enums elements
- `MemberName = string()`
the element name

Function returns the IDL typecode for enum.

`string(Length) -> TC`

Types:

- `Length = integer()`
the length of the string (0 means unbounded)

Function returns the IDL typecode for string.

`sequence(ElemTC, Length) -> TC`

Types:

- ElemTC = TC
the typecode for the sequence elements
- Length = integer()
the length of the sequence (0 means unbounded)

Function returns the IDL typecode for sequence.

`array(ElemTC, Length) -> TC`

Types:

- ElemTC = TC
the typecode for the array elements
- Length = integer()
the length of the array

Function returns the IDL typecode for array.

`alias(Id, Name, AliasTC) -> TC`

Types:

- Id = string()
the repository ID
- Name = string()
the type name of the alias
- AliasTC = TC
the typecode for the type which the alias refer to

Function returns the IDL typecode for alias.

`exception(Id, Name, ElementList) -> TC`

Types:

- Id = string()
the repository ID
- Name = string()
the type name of the exception
- ElementList = [{MemberName, TC}]
a list of the exception elements
- MemberName = string()
the element name

Function returns the IDL typecode for exception.

`get_tc(Object) -> TC`

`get_tc(Id) -> TC`

Types:

- Object = record()
an IDL specified struct, union or exception
- Id = string()
the repository ID

If the `get_tc/1` gets a record that is an IDL specified struct, union or exception as a parameter it returns the typecode.

If the parameter is a repository ID it uses the Interface Repository to get the typecode.

`check(TC) -> boolean()`

Function checks the syntax of an IDL typecode.

List of Figures

1.1	Figure 1: Orber Dependencies and Structure.	4
1.2	Figure 2: ORB interface between Java and Erlang Environment Nodes.	5
1.3	Figure 1: How the Object Request Broker works.	6
1.4	Figure 2: IIOP communication between domains and objects.	7
1.5	Figure 1: Contextual object relationships using the Naming Service.	23
1.6	The Invocation Order of Interceptor Functions.	49
1.7	The Menu Frame.	54
1.8	Configuration Settings.	55
1.9	Select Type.	56
1.10	List Registered Exceptions.	57
1.11	Add a New Context.	58
1.12	Delete Context.	58
1.13	Object Stored in the NameService.	59
1.14	Object Data.	60
1.15	Create a New Object.	61

List of Tables

1.1	OMG IDL basic types	14
1.2	OMG IDL constructed types	14
1.3	Typical values	15
1.4	Type Code tuples	19
1.5	Currently reserved key strings	28
1.6	Stringified Name represenation	28
1.7	Table 1: System Exceptions Status	45

Glossary

BindingIterator

The binding iterator (Like a book mark) indicates which objects have been read from the list.
Local for chapter 1.

CORBA

A specification of an architecture for a distributed object system

CORBA

Common Object Request Broker Architecture is a common communication standard developed by the
OMG (Object Management Group)
Local for chapter 1.

domains

A domain allows a more efficient communication protocol to be used between objects not on the same
node without the need of an ORB
Local for chapter 1.

IDL

Interface Definition Language - IDL is the OMG specified interface definition language, used to define
the CORBA object interfaces.
Local for chapter 1.

IIOP

Internet-Inter ORB Protocol
Local for chapter 1.

IOR

Interoperable Object Reference
Local for chapter 1.

ORB

Object Request Broker - ORB open software bus architecture specified by the OMG which allows object components to communicate in a heterogeneous environment.

Local for chapter 1.

Orber domain

A domain containing several Erlang nodes, which are communicating by using the Erlang internal format. An Orber domain looks as one ORB from the environment.

Local for chapter 1.

Orber installation

is the structure of the ORB or ORBs as defined during the install process is called the "installation".

Local for chapter 1.

Type Code

Type Code is a full definition of a type

Local for chapter 1.

Type Codes

Type codes give a complete description of the type including all its components and structure.

Local for chapter 1.

Index of Modules and Functions

Modules are typed in *this* way.
Functions are typed in *this* way.

add_alternate_iiop_address/3
 corba , 116

add_initial_service/2
 corba , 114

add_node/2
 orber , 134

alias/3
 orber_tc , 152

any
 create/0, 110
 create/2, 110
 get_typecode/1, 110
 get_value/1, 111
 set_typecode/2, 110
 set_value/2, 111

any/0
 orber_tc , 150

array/2
 orber_tc , 152

bind/3
 CosNaming_NamingContext , 100

bind_context/3
 CosNaming_NamingContext , 100

bind_new_context/2
 CosNaming_NamingContext , 101

boolean/0
 orber_tc , 150

bootstrap_port/0
 orber , 133

char/0
 orber_tc , 150

check/1
 orber_tc , 153

closed_in_connection/1
 interceptors , 121

closed_out_connection/1
 interceptors , 121

configure/2
 orber , 135

contents/3
 orber_ifr , 139

corba
 add_alternate_iiop_address/3, 116
 add_initial_service/2, 114
 create/2, 112
 create/3, 112
 create/4, 112
 create_link/2, 112
 create_link/3, 112
 create_link/4, 112
 create_nil_objref/0, 113
 create_subobject_key/2, 113
 dispose/1, 113
 get_pid/1, 114
 get_subobject_key/1, 114
 list_initial_services/0, 115
 list_initial_services_remote/1, 115
 object_to_string/1, 116
 orb_init/1, 116
 print_object/2, 116
 raise/1, 114
 remove_initial_service/1, 115
 resolve_initial_references/1, 114
 resolve_initial_references_remote/2,
 115
 string_to_object/1, 116

corba_object
 get_interface/1, 118
 hash/2, 119
 is_a/2, 118
 is_equivalent/2, 119

- is_nil/1, 118
- is_remote/1, 118
- non_existent/1, 119
- not_existent/1, 119
- CosNaming_BindingIterator*
 - destroy/1, 97
 - next_n/2, 97
 - next_one/1, 97
- CosNaming_NamingContext*
 - bind/3, 100
 - bind_context/3, 100
 - bind_new_context/2, 101
 - destroy/1, 101
 - list/2, 101
 - new_context/1, 101
 - rebind/3, 100
 - rebind_context/3, 100
 - resolve/2, 100
 - unbind/2, 101
- CosNaming_NamingContextExt*
 - resolve_str/2, 102
 - to_name/2, 102
 - to_string/2, 102
 - to_url/3, 102
- create/0
 - any, 110
 - lname, 125
 - lname_component, 127
- create/2
 - any, 110
 - corba, 112
- create/3
 - corba, 112
- create/4
 - corba, 112
- create_alias/5
 - orber_ifr, 141
- create_array/3
 - orber_ifr, 143
- create_attribute/6
 - orber_ifr, 149
- create_constant/6
 - orber_ifr, 140
- create_enum/5
 - orber_ifr, 141
- create_exception/5
 - orber_ifr, 142
- create_idltype/2
 - orber_ifr, 143
- create_interface/5
 - orber_ifr, 141
- create_link/2
 - corba, 112
- create_link/3
 - corba, 112
- create_link/4
 - corba, 112
- create_module/4
 - orber_ifr, 140
- create_nil_objref/0
 - corba, 113
- create_operation/9
 - orber_ifr, 149
- create_sequence/3
 - orber_ifr, 143
- create_string/2
 - orber_ifr, 142
- create_struct/5
 - orber_ifr, 140
- create_subobject_key/2
 - corba, 113
- create_union/6
 - orber_ifr, 141
- delete_component/2
 - lname, 126
- describe/1
 - orber_ifr, 138
- describe_contents/4
 - orber_ifr, 140
- describe_interface/1
 - orber_ifr, 149
- destroy/1
 - CosNaming_BindingIterator*, 97
 - CosNaming_NamingContext*, 101
 - orber_ifr, 137
- dispose/1
 - corba, 113
- domain/0
 - orber, 130
- double/0

<i>orber_tc</i> , 150	<i>orber_ifr</i> , 137
enum/3	get_interface/1
<i>orber_tc</i> , 151	<i>corba_object</i> , 118
equal/2	get_kind/1
<i>lname</i> , 126	<i>lname_component</i> , 127
exception/3	<i>orber_ifr</i> , 145
<i>orber_tc</i> , 152	get_length/1
	<i>orber_ifr</i> , 146
find_repository/0	get_lightweight_nodes/0
<i>orber_ifr</i> , 136	<i>orber</i> , 130
float/0	get_members/1
<i>orber_tc</i> , 150	<i>orber_ifr</i> , 144
from_idl_form/1	get_mode/1
<i>lname</i> , 126	<i>orber_ifr</i> , 146
get_absolute_name/1	get_name/1
<i>orber_ifr</i> , 138	<i>orber_ifr</i> , 137
get_base_interfaces/1	get_ORBDefaultInitRef/0
<i>orber_ifr</i> , 148	<i>orber</i> , 130
get_bound/1	get_ORBInitRef/0
<i>orber_ifr</i> , 145	<i>orber</i> , 130
get_component/2	get_original_type_def/1
<i>lname</i> , 125	<i>orber_ifr</i> , 145
get_containing_repository/1	get_params/1
<i>orber_ifr</i> , 138	<i>orber_ifr</i> , 147
get_contexts/1	get_pid/1
<i>orber_ifr</i> , 148	<i>corba</i> , 114
get_def_kind/1	get_primitive/2
<i>orber_ifr</i> , 137	<i>orber_ifr</i> , 142
get_defined_in/1	get_result/1
<i>orber_ifr</i> , 138	<i>orber_ifr</i> , 147
get_discriminator_type/1	get_result_def/1
<i>orber_ifr</i> , 144	<i>orber_ifr</i> , 147
get_discriminator_type_def/1	get_subobject_key/1
<i>orber_ifr</i> , 144	<i>corba</i> , 114
get_element_type/1	get_tc/1
<i>orber_ifr</i> , 146	<i>orber_tc</i> , 152
get_element_type_def/1	get_type/1
<i>orber_ifr</i> , 146	<i>orber_ifr</i> , 142
get_exceptions/1	get_type_def/1
<i>orber_ifr</i> , 148	<i>orber_ifr</i> , 143
get_id/1	get_typecode/1
<i>lname_component</i> , 127	<i>any</i> , 110
	get_value/1

- any*, 111
- orber_ifr*, 143
- get_version/1*
 - orber_ifr*, 138
- hash/2*
 - corba_object*, 119
- iiop_connection_timeout/0*
 - orber*, 131
- iiop_connections/0*
 - orber*, 131
- iiop_connections_pending/0*
 - orber*, 131
- iiop_out_ports/0*
 - orber*, 130
- iiop_port/0*
 - orber*, 130
- iiop_ssl_port/0*
 - orber*, 130
- iiop_timeout/0*
 - orber*, 131
- in_reply/6*
 - interceptors*, 122
- in_reply_encoded/6*
 - interceptors*, 122
- in_request/6*
 - interceptors*, 122
- in_request_encoded/6*
 - interceptors*, 122
- init/2*
 - orber_ifr*, 136
- insert_component/3*
 - lname*, 125
- install/1*
 - orber*, 133
- install/2*
 - orber*, 133
- interceptors*
 - closed_in_connection/1*, 121
 - closed_out_connection/1*, 121
 - in_reply/6*, 122
 - in_reply_encoded/6*, 122
 - in_request/6*, 122
 - in_request_encoded/6*, 122
 - new_in_connection/3*, 121
 - new_out_connection/3*, 121
 - out_reply/6*, 123
 - out_reply_encoded/6*, 123
 - out_request/6*, 123
 - out_request_encoded/6*, 123
- is_a/2*
 - corba_object*, 118
 - orber_ifr*, 148
- is_equivalent/2*
 - corba_object*, 119
- is_lightweight/0*
 - orber*, 130
- is_nil/1*
 - corba_object*, 118
- is_remote/1*
 - corba_object*, 118
- jump_start/1*
 - orber*, 129
- less_than/2*
 - lname*, 126
- list/2*
 - CosNaming_NamingContext*, 101
- list_initial_services/0*
 - corba*, 115
- list_initial_services_remote/1*
 - corba*, 115
- lname*
 - create/0*, 125
 - delete_component/2*, 126
 - equal/2*, 126
 - from_idl_form/1*, 126
 - get_component/2*, 125
 - insert_component/3*, 125
 - less_than/2*, 126
 - num_components/1*, 126
 - to_idl_form/1*, 126
- lname_component*
 - create/0*, 127
 - get_id/1*, 127
 - get_kind/1*, 127
 - set_id/2*, 127
 - set_kind/2*, 128
- long/0*
 - orber_tc*, 150

lookup/2
 orber_ifr , 139

lookup_id/2
 orber_ifr , 142

lookup_name/5
 orber_ifr , 139

Module_Interface
 Module_Interface:own_functions/4,
 106
 Module_Interface:own_functions/5,
 106
 Module_Interface_impl:code_change/3,
 107
 Module_Interface_impl:handle_info/2,
 107
 Module_Interface_impl:init/1, 107
 Module_Interface_impl:own_functions/4,
 108
 Module_Interface_impl:own_functions/5,
 108
 Module_Interface_impl:terminate/2,
 107
 oe_create/0, 105
 oe_create/1, 105
 oe_create/2, 105
 oe_create_link/0, 105
 oe_create_link/1, 105
 oe_create_link/2, 106
 typeID/0, 105

Module_Interface:own_functions/4
 Module_Interface , 106

Module_Interface:own_functions/5
 Module_Interface , 106

Module_Interface_impl:code_change/3
 Module_Interface , 107

Module_Interface_impl:handle_info/2
 Module_Interface , 107

Module_Interface_impl:init/1
 Module_Interface , 107

Module_Interface_impl:own_functions/4
 Module_Interface , 108

Module_Interface_impl:own_functions/5
 Module_Interface , 108

Module_Interface_impl:terminate/2
 Module_Interface , 107

move/4
 orber_ifr , 139

new_context/1
 CosNaming_NamingContext , 101

new_in_connection/3
 interceptors , 121

new_out_connection/3
 interceptors , 121

next_n/2
 CosNaming_BindingIterator , 97

next_one/1
 CosNaming_BindingIterator , 97

non_existent/1
 corba_object , 119

not_existent/1
 corba_object , 119

null/0
 orber_tc , 150

num_components/1
 lname , 126

object_reference/2
 orber_tc , 150

object_to_string/1
 corba , 116

objectkeys_gc_time/0
 orber , 132

octet/0
 orber_tc , 150

oe_create/0
 Module_Interface , 105

oe_create/1
 Module_Interface , 105

oe_create/2
 Module_Interface , 105

oe_create_link/0
 Module_Interface , 105

oe_create_link/1
 Module_Interface , 105

oe_create_link/2
 Module_Interface , 106

orb_init/1
 corba , 116

orber

- add_node/2, 134
- bootstrap_port/0, 133
- configure/2, 135
- domain/0, 130
- get_lightweight_nodes/0, 130
- get_ORBDefaultInitRef/0, 130
- get_ORBInitRef/0, 130
- iiop_connection_timeout/0, 131
- iiop_connections/0, 131
- iiop_connections_pending/0, 131
- iiop_out_ports/0, 130
- iiop_port/0, 130
- iiop_ssl_port/0, 130
- iiop_timeout/0, 131
- install/1, 133
- install/2, 133
- is_lightweight/0, 130
- jump_start/1, 129
- objectkeys_gc_time/0, 132
- orber_nodes/0, 133
- remove_node/1, 134
- secure/0, 131
- set_ssl_client_certfile/1, 132
- set_ssl_client_depth/1, 132
- set_ssl_client_verify/1, 132
- ssl_client_certfile/0, 132
- ssl_client_depth/0, 132
- ssl_client_verify/0, 132
- ssl_server_certfile/0, 131
- ssl_server_depth/0, 132
- ssl_server_verify/0, 132
- start/0, 129
- start_lightweight/0, 129
- start_lightweight/1, 129
- stop/0, 130
- uninstall/0, 134

orber_ifr

- contents/3, 139
- create_alias/5, 141
- create_array/3, 143
- create_attribute/6, 149
- create_constant/6, 140
- create_enum/5, 141
- create_exception/5, 142
- create_idltype/2, 143
- create_interface/5, 141
- create_module/4, 140
- create_operation/9, 149
- create_sequence/3, 143
- create_string/2, 142
- create_struct/5, 140

- create_union/6, 141
- describe/1, 138
- describe_contents/4, 140
- describe_interface/1, 149
- destroy/1, 137
- find_repository/0, 136
- get_absolute_name/1, 138
- get_base_interfaces/1, 148
- get_bound/1, 145
- get_containing_repository/1, 138
- get_contexts/1, 148
- get_def_kind/1, 137
- get_defined_in/1, 138
- get_discriminator_type/1, 144
- get_discriminator_type_def/1, 144
- get_element_type/1, 146
- get_element_type_def/1, 146
- get_exceptions/1, 148
- get_id/1, 137
- get_kind/1, 145
- get_length/1, 146
- get_members/1, 144
- get_mode/1, 146
- get_name/1, 137
- get_original_type_def/1, 145
- get_params/1, 147
- get_primitive/2, 142
- get_result/1, 147
- get_result_def/1, 147
- get_type/1, 142
- get_type_def/1, 143
- get_value/1, 143
- get_version/1, 138
- init/2, 136
- is_a/2, 148
- lookup/2, 139
- lookup_id/2, 142
- lookup_name/5, 139
- move/4, 139
- set_base_interfaces/2, 148
- set_bound/2, 145
- set_contexts/2, 148
- set_discriminator_type_def/2, 145
- set_element_type_def/2, 146
- set_exceptions/2, 148
- set_id/2, 137
- set_length/2, 146
- set_members/2, 144
- set_mode/2, 147
- set_name/2, 137
- set_original_type_def/2, 145
- set_params/2, 147
- set_result_def/2, 147

```

    set_type_def/2, 143
    set_value/2, 144
    set_version/2, 138
orber_nodes/0
    orber, 133
orber_tc
    alias/3, 152
    any/0, 150
    array/2, 152
    boolean/0, 150
    char/0, 150
    check/1, 153
    double/0, 150
    enum/3, 151
    exception/3, 152
    float/0, 150
    get_tc/1, 152
    long/0, 150
    null/0, 150
    object_reference/2, 150
    octet/0, 150
    principal/0, 150
    sequence/2, 152
    short/0, 150
    string/1, 151
    struct/3, 150
    typecode/0, 150
    union/5, 151
    unsigned_long/0, 150
    unsigned_short/0, 150
    void/0, 150
out_reply/6
    interceptors, 123
out_reply_encoded/6
    interceptors, 123
out_request/6
    interceptors, 123
out_request_encoded/6
    interceptors, 123
principal/0
    orber_tc, 150
print_object/2
    corba, 116
raise/1
    corba, 114
rebind/3
    CosNaming_NamingContext, 100
    rebind_context/3
        CosNaming_NamingContext, 100
remove_initial_service/1
    corba, 115
remove_node/1
    orber, 134
resolve/2
    CosNaming_NamingContext, 100
resolve_initial_references/1
    corba, 114
resolve_initial_references_remote/2
    corba, 115
resolve_str/2
    CosNaming_NamingContextExt, 102
secure/0
    orber, 131
sequence/2
    orber_tc, 152
set_base_interfaces/2
    orber_ifr, 148
set_bound/2
    orber_ifr, 145
set_contexts/2
    orber_ifr, 148
set_discriminator_type_def/2
    orber_ifr, 145
set_element_type_def/2
    orber_ifr, 146
set_exceptions/2
    orber_ifr, 148
set_id/2
    lname_component, 127
    orber_ifr, 137
set_kind/2
    lname_component, 128
set_length/2
    orber_ifr, 146
set_members/2
    orber_ifr, 144
set_mode/2
    orber_ifr, 147
set_name/2
    orber_ifr, 137

```

set_original_type_def/2 <i>orber_ifr</i> , 145	string/1 <i>orber_tc</i> , 151
set_params/2 <i>orber_ifr</i> , 147	string_to_object/1 <i>corba</i> , 116
set_result_def/2 <i>orber_ifr</i> , 147	struct/3 <i>orber_tc</i> , 150
set_ssl_client_certfile/1 <i>orber</i> , 132	to_idl_form/1 <i>lname</i> , 126
set_ssl_client_depth/1 <i>orber</i> , 132	to_name/2 <i>CosNaming_NamingContextExt</i> , 102
set_ssl_client_verify/1 <i>orber</i> , 132	to_string/2 <i>CosNaming_NamingContextExt</i> , 102
set_type_def/2 <i>orber_ifr</i> , 143	to_url/3 <i>CosNaming_NamingContextExt</i> , 102
set_typecode/2 <i>any</i> , 110	typecode/0 <i>orber_tc</i> , 150
set_value/2 <i>any</i> , 111 <i>orber_ifr</i> , 144	typeID/0 <i>Module.Interface</i> , 105
set_version/2 <i>orber_ifr</i> , 138	unbind/2 <i>CosNaming_NamingContext</i> , 101
short/0 <i>orber_tc</i> , 150	uninstall/0 <i>orber</i> , 134
ssl_client_certfile/0 <i>orber</i> , 132	union/5 <i>orber_tc</i> , 151
ssl_client_depth/0 <i>orber</i> , 132	unsigned_long/0 <i>orber_tc</i> , 150
ssl_client_verify/0 <i>orber</i> , 132	unsigned_short/0 <i>orber_tc</i> , 150
ssl_server_certfile/0 <i>orber</i> , 131	void/0 <i>orber_tc</i> , 150
ssl_server_depth/0 <i>orber</i> , 132	
ssl_server_verify/0 <i>orber</i> , 132	
start/0 <i>orber</i> , 129	
start_lightweight/0 <i>orber</i> , 129	
start_lightweight/1 <i>orber</i> , 129	
stop/0 <i>orber</i> , 130	