Orber Application

version 3.6

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8.5 Document System.

Contents

Chapter 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.

1.2 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

1.2.1 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).

1.2.2 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.

1.2.3 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.

1.2.4 CosNaming Service

Orber contains a CosNaming compliant service.

1.2.5 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).

1.2.6 Orber Stub/Skeleton

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

1.2.7 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.

1.2.8 Interceptors

Descibes how to implement and activate interceptors.

1.2.9 OrberWeb

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

1.2.10 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.

Chapter 2

Introduction to Orber and the IFR

This chapter contains an introduction to Orber and the IFR (Interface Repository).

2.1 Introduction to Orber

2.1.1 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:

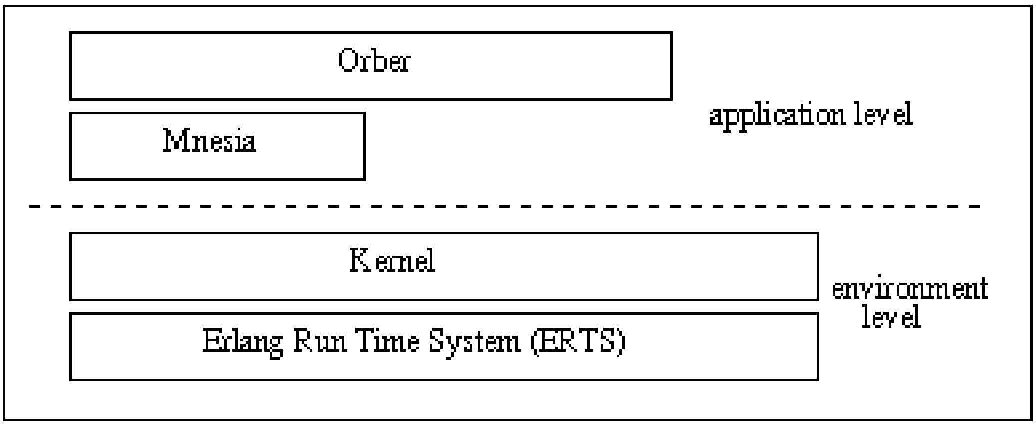

Figure 2.1: Figure 1: Orber Dependencies and Structure.

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.

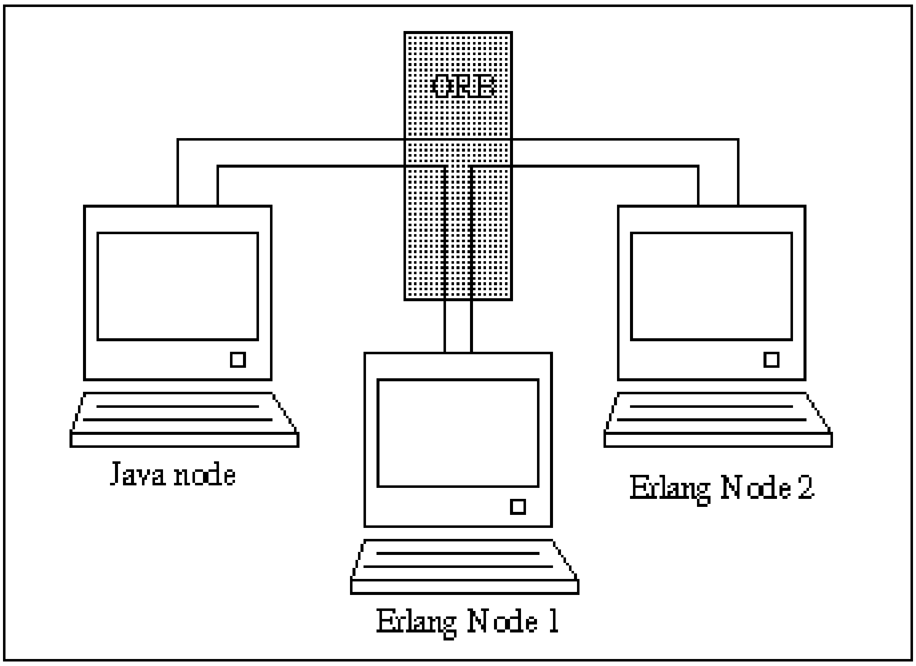

Figure 2.2: Figure 2: ORB interface between Java and Erlang Environment Nodes.

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 achive 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*.

2.2 The Orber Application

2.2.1 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.

2.2.2 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:

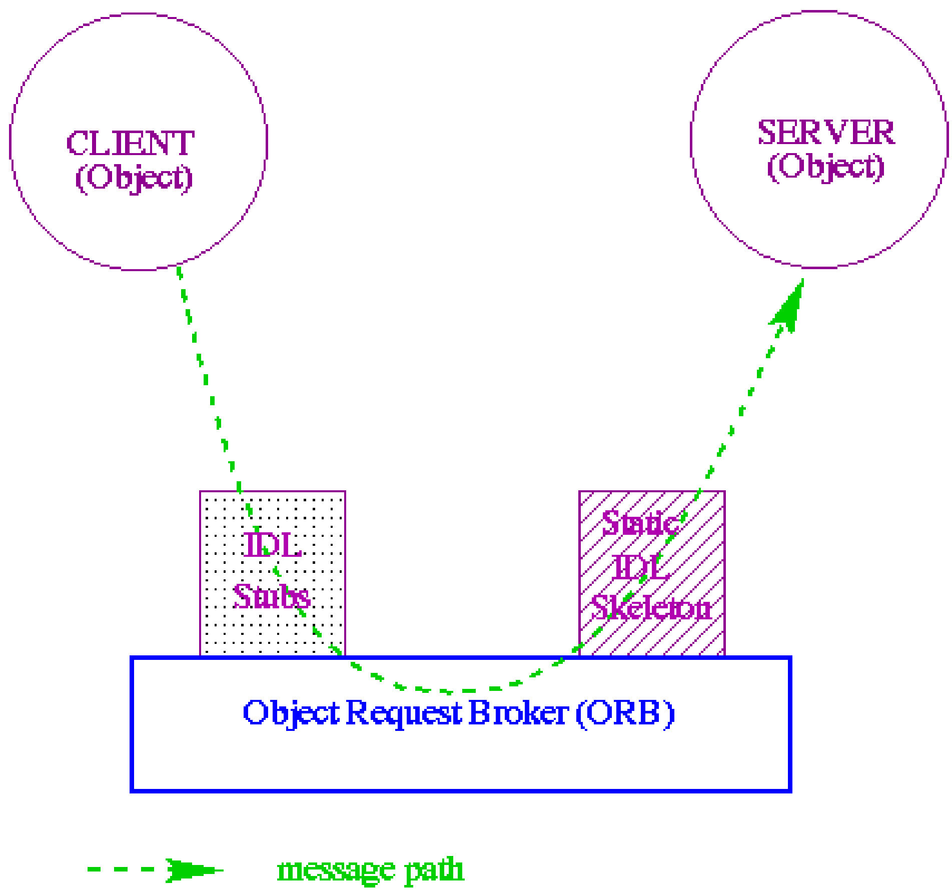

Figure 2.3: Figure 1: How the Object Request Broker works.

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.

2.2.3 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:

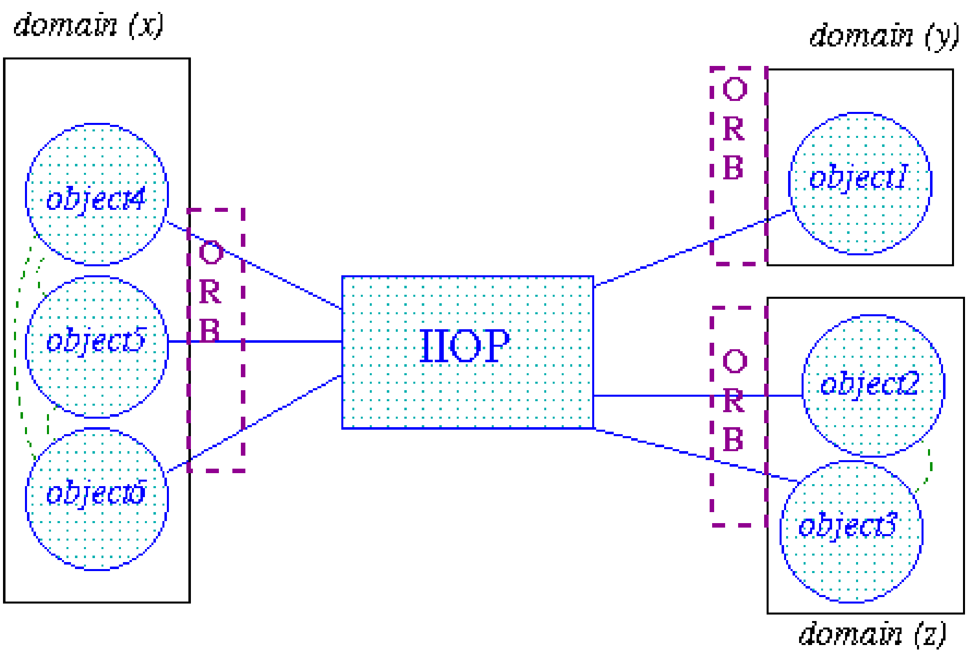

Figure 2.4: Figure 2: IIOP communication between domains and objects.

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.

2.3 Interface Repository

2.3.1 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.

Chapter 3

3.1 Installation Process

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

3.1.1 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` domain. Consult the configuration settings [page 12] section. If you encounter any problems; see the chapter about *Debugging* in this User's Guide.

3.1.2 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.

3.1.3 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 disk 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, "MyDiskSingleNodeORB"}]).
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 130]. But, you *should* at least set the IIOP timeout parameters.

3.1.4 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 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().
```

3.1.5 Install Disk 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 disk 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, "MyDiskMultiNodeORB"}]).
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, "MyDiskMultiNodeORB"}]).
nodeB@hostB> orber:start().
```

3.2 Configuration

It is essential that one configure Orber properly, to avoid, for example, malicious attacks and automatically terminate IIOP connections no longer in use. An easy way to extract information about Orber's configuration parameters is to invoke the operation orber:info/1/2 [page 151]. Orber offer the following configuration parameters:

Key	*Range*	*Default*
domain	string()	"ORBER"
iiop_port	integer() >= 0	4001
nat_iiop_port	integer() > 0 | {local, integer(), [{integer(), integer()}]}	The same as `iiop_port`
iiop_out_ports	0 | {integer(),integer()}	0
iiop_max_fragments	integer() > 0 | infinity	infinity
iiop_max_in_requests	integer() > 0 | infinity	infinity
iiop_max_in_connections	integer() > 0	infinity
iiop_backlog	integer() > 0	5
iiop_packet_size	integer() > 0 | infinity	infinity
ip_address	string() | {multiple, [string()]}	All interfaces
ip_address_local	string()	Defined by the underlying system
nat_ip_address	string() | {multiple, [string()]} | {local, string(), [{string(), string()}]}	The same as `ip_address`
objectkeys_gc_time	integer() > 0 | infinity	infinity
giop_version	{1,0} | {1,1} | {1,2}	{1,1}
iiop_setup_connection_timeout	integer() > 0 | infinity	infinity
iiop_connection_timeout	integer() > 0 | infinity	infinity
iiop_in_connection_timeout	integer() > 0 | infinity	infinity
iiop_out_keepalive	true | false	false
iiop_in_keepalive	true | false	false
iiop_timeout	integer() > 0 | infinity	infinity
interceptors	{native, [atom()]}	-
local_interceptors	{native, [atom()]}	-
orbInitRef	[string()] | undefined	undefined
orbDefaultInitRef	string() | undefined	undefined
orber_debug_level	0 - 10	0
flags	integer() >= 0	0
iiop_acl	[{atom(), string()}] | [{atom(), string(), [string()]}]	[]
secure	no | ssl	no
ssl_generation	2 | 3	2
iiop_ssl_port	integer() >= 0	4002

continued ...

... continued

iiop_ssl_accept_timeout	integer() > 0 | infinity	infinity
iiop_ssl_backlog	integer() > 0	5
iiop_ssl_ip_address_local	string()	Defined by the underlying system
nat_iiop_ssl_port	integer() > 0 | {local, integer(), [{integer(), integer()}]}	The same as `iiop_ssl_port`
ssl_server_cacertfile	string()	-
ssl_server_certfile	string()	-
ssl_server_verify	0 | 1 | 2	-
ssl_server_depth	integer()	-
ssl_server_password	string()	-
ssl_server_keyfile	string()	-
ssl_server_ciphers	string()	-
ssl_server_cachetimeout	integer() | infinity	infinity
ssl_client_cacertfile	string()	-
ssl_client_certfile	string()	-
ssl_client_verify	0 | 1 | 2	-
ssl_client_depth	integer()	-
ssl_client_password	string()	-
ssl_client_keyfile	string()	-
ssl_client_ciphers	string()	-
ssl_client_cachetimeout	integer() | infinity	infinity
iiop_ssl_out_keepalive	true | false	false
iiop_ssl_in_keepalive	true | false	false

Table 3.1: Orber Configuration Parameters

Comments on the table 'Orber Configuration Parameters':

domain Since Orber domains, they are supposed to communicate via IIOP, *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 If set to 0 the OS will pick any vacant port.
*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.

nat_iiop_port The value is either an integer or `{local, DefaultNATPort, [{Port, NATPort}]}`. See also Firewall Configuration [page 18].

iiop_out_ports When set to 0 any available port will be used. If a range is specified, 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. If communicating via SSL, make sure you use a version that supports the local `{port, Port}` option. See also Firewall Configuration [page 18].

iiop_max_fragments Limits the number of IIOP fragments allowed per request.

iiop_max_in_requests Limits the number of concurrent incoming requests per incoming connection.

iiop_max_in_connections Limits the number of concurrent incoming connections.

iiop_backlog Defines the maximum length the queue of pending incoming connections may grow to.

iiop_packet_size Defines the maximum size of incoming requests. If this limit is exceeded, the connection is closed.

ip_address This option is used if orber only should listen on a specific ip interface on a multi-interface host or if exported IOR:s should contain multiple components. The value is the IPv4 or IPv6 address as a string or {`multiple, IPList`}. The latter requires that the object is available via the all IP addresses found in the list.

ip_address_local This option defines the default local interface Orber will use when connecting to another ORB via IIOP, i.e., Orber act as the client side ORB. The value is a IPv4 or IPv6 address as a string. It is possible to override `ip_address_local` by defining `iiop_acl` or passing the Orber generic `interface` Context. If this option is not used, the underlying OS will choose which interface to use. For more information, see the Interface Configuration [page 20] section.

nat_ip_address The value is the ip address as a string (IPv4 or IPv6), {`multiple, IPList`} or {`local, DefaultNATIPAddress, [`{`IPAddress, NATIPAddress`}`]`}. See also Firewall Configuration [page 18].

objectkeys_gc_time This option should be set if objects are started using the option {`persistent, true`}. The value is `integer()` seconds.

giop_version Defines the default GIOP protocol version.

iiop_setup_connection_timeout 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 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_in_connection_timeout The same as for `iiop_connection_timeout`. The only difference is that this option only affects incoming connections (i.e. Orber act as server-side ORB).

iiop_out_keepalive Enables periodic transmission on a connected socket, when no other data is being exchanged. If the other end does not respond, the connection is considered broken and will be terminated. When enabled the SO_KEEPALIVE socket level option is set.

iiop_in_keepalive The same as for `iiop_out_keepalive`. The only difference is that this option only affects incoming connections.

iiop_timeout 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 a system exception, e.g. `TIMEOUT`, 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.

local_interceptors If defined, its value will be used when activating local interceptors via Orber Environment Flags [page 17]. If not defined, but the flag is set, Orber will use the value of the `interceptors` parameter.

orbInitRef 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 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 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.

flags No flags are activated in the default case. The available configuration settings are described in Orber Environment Flags [page 17].

iiop_acl This option must be activated by setting Orber Environment Flags [page 17] parameter. The value of this parameter shall be a list of `[{Direction, Filter}]` and/or `[{Direction, Filter, [Interfaces]}]`. The `Direction`, `tcp_in`, `ssl_in`, `tcp_out` or `ssl_out`, determines if the Access Control List (ACL) applies to incoming or outgoing connections and IIOP or IIOP over SSL. The `Filter` uses a extended format of Classless Inter Domain Routing (CIDR). For example, `"123.123.123.10"` limits the connection to that particular host, while `"123.123.123.10/17"` allows connections to or from any host equal to the 17 most significant bits. Orber also allow the user to specify a certain port or port range, for example, `"123.123.123.10/17#4001"` and `"123.123.123.10/17#4001/5001"` respectively. IPv4 or none compressed IPv6 strings are accepted.
The list of `Interfaces`, IPv4 or IPv6 strings, may only contain *one* address for outgoing connections. For incoming connections, the `Interfaces` list may contain several IP strings. If set for outgoing connections, and access is granted, Orber will use that local interface when connecting to the server-side ORB. For incoming connections, the client-side ORB is required to use one of the listed interfaces locally. If it fail to do so, access will be denied. The module orber_acl [page 161] provides operations for evaluating the access control for filters and addresses. See also the Interface Configuration [page 20] and Firewall Configuration [page 18] chapters.

secure Determines the security mode Orber will use, which is either `no` if it is an insecure domain or the type of security mechanism used. Currently, per default, Orber is compliant with `CSIv1` level 0, which means IIOP via SSL/TLS. The security chapter later in this manual describes how to get security in Orber and how the options are used.

ssl_generation Defines which SSL version, i.e. available API, is installed. The default value, `2`, refers to SSL-3.1 or later, but earlier than SSL-4.0. If set to `3` SSL-4.0, or later, must be available. Currently it not possible to use `1`, it is only reserved for future use.

iiop_ssl_port If set, the value must be an integer greater than zero and not equal to *iiop_port*.

iiop_ssl_accept_timeout The value is an integer (timeout in seconds) or the atom infinity and determine how long the SSL handshake may take. This option should be set to avoid if a client never initiate the handshake.

iiop_ssl_backlog Defines the maximum length the queue of pending incoming connections may grow to.

iiop_ssl_ip_address_local This option defines the default local interface Orber will use when connecting to another ORB via IIOP SSL, i.e., Orber act as the client side ORB. The value is a IPv4 or IPv6 address as a string. It is possible to override `iiop_ssl_ip_address_local` by defining `iiop_acl` or passing the Orber generic `interface` Context. If this option is not used, the underlying OS will choose which interface to use. For more information, see the Interface Configuration [page 20] section.

nat_iiop_ssl_port If set, the value must be an integer greater than zero or `{local, DefaultNATPort, [{Port, NATPort}]}`. See also Firewall Configuration [page 18].

ssl_server_cacertfile the file path to a server side CA certificate.

ssl_server_certfile The path to a file containing a chain of PEM encoded certificates.

ssl_server_verify The type of verification used by SSL during authentication of the other peer for incoming calls.

ssl_server_depth The SSL verification depth for outgoing calls.

ssl_server_password The server side key string.

ssl_server_keyfile The file path to a server side key.

ssl_server_ciphers The server side cipher string.

ssl_server_cachetimeout The server side cache timeout.

ssl_client_cacertfile The file path to a client side CA certificate.

ssl_client_certfile The path to a file containing a chain of PEM encoded certificates.

ssl_client_verify The type of verification used by SSL during authentication of the other peer for outgoing calls.

ssl_client_depth The SSL verification depth for incoming calls.

ssl_client_password The client side key string.

ssl_client_keyfile The file path to a client side key.

ssl_client_ciphers The client side cipher string.

ssl_client_cachetimeout The client side cache timeout.

iiop_ssl_out_keepalive Enables periodic transmission on a connected socket, when no other data is being exchanged. If the other end does not respond, the connection is considered broken and will be terminated. When enabled the SO_KEEPALIVE socket level option is set. Requires that the installed SSL version support the *keepalive* option and that the *ssl_generation* points to this version.

iiop_ssl_in_keepalive The same as for `iiop_ssl_out_keepalive`. The only difference is that this option only affects incoming connections.

It is possible to invoke operations using the extra timeout parameter:

```
erl> module_interface:function(ObjRef, Timeout, ..Arguments..).
erl> module_interface:function(ObjRef, [{timeout, Timeout}], ..Arguments..).
erl> module_interface:function(ObjRef, ..Arguments..).
```

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`, `iiop_connection_timeout` and `iiop_in_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`.

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.

3.2.1 Orber Environment Flags

The `Environment Flags` allows the user to activate debugging facilities or change Orber's behavior. The latter may result in that Orber is no longer compliant with the OMG standard, which may be necessary when communicating with a non-compliant ORB.

Hexadecimal Value	*OMG Compliant*	*Description*
0001	no	Exclude CodeSet Component
0002	yes	Local Typechecking
0004	yes	Use Host Name in IOR
0008	yes	Enable NAT
0020	yes	Local Interceptors
0080	yes	Light IFR
0100	yes	Use IPv6
0200	yes	EXIT Tolerance
0400	yes	Enable Incoming ACL
0800	yes	Enable Outgoing ACL
1000	yes	Use Current Interface in IOR

Table 3.2: Orber Environment Flags

Any combination of the flags above may be used and changes the behavior as follows:

- *Exclude CodeSet Component* - instruct Orber to exclude the CodeSet component in exported IOR:s. When activated, no negotiating regarding character and wide character conversions between the client and the server will occur. This flag will, most likely, cause problems if your IDL specification contains the data types wchar and/or wstring.
- *Local Typechecking* - If activated, parameters, replies and raised exceptions will be checked to ensure that the data is correct. If an error occurs, the `error_logger` is used to generate reports. One *MAY NOT* use this option for delivered systems due to the extra overhead. Since this option activates typechecking for all objects generated on the target node, it is also possible to use the option {`local_typecheck, boolean()`}, when invoking `oe_create/2`, `oe_create_link/2`, `corba:create/4` or `corba:create_link/4`, to override the configuration parameter.
- *Use Host Name in IOR* - normally Orber inserts the IP-number in IOR:s when they are exported. In some cases, this will cause the clients to open two connections instead of one.
- *Enable NAT* - if this flag is set, it is possible to use the NAT (Network Address Translation) configuration parameters (`nat_iiop_port`, `nat_iiop_ssl_port` and `nat_ip_address`).
- *Local Interceptors* - use interceptors for local invocations.
- *Light IFR* - if the IFR is not explicitly used and this flag is set, Orber will use a minimal IFR to reduce memory usage and installation time.
- *Use IPv6* - when this option is activated, Orber will use `IPv6` for inter-ORB communication.
- *EXIT Tolerance* - servers will survive even though the call-back module caused an EXIT.
- *Enable Incoming ACL* - activates access control for incoming connections.
- *Enable Outgoing ACL* - activates access control for outgoing connections.

- *Use Current Interface in IOR* - when set, Orber will add the interface the request came via to exported local IOR:s.

Invoking the operation orber:info/1/2 [page 151] will display the currently set flags in a readable way.

3.3 Firewall Configuration

Firewalls are used to protect objects from clients in other networks or sub-networks, but also to restrict which hosts internal objects may connect to (i.e. `inbound protection` and `outbound protection`). A firewall can limit access based on:

- *Transport Level* - performs access control decisions based on address information in TCP headers.
- *Application Level* - understands GIOP messages and the specific transport level inter-ORB Protocol supported e.g. IIOP.

This section describes how to configure a `Transport Level` firewall. It must have prior knowledge of the source to destination mappings, and conceptually has a configuration table containing tuples of the form: (`{inhost:inport}`, `{outhost:outport}`). If there are no port restrictions it is rather easy to configure the firewall. Otherwise, we must consider the following alternatives:

- *Incoming Requests* - Orber only uses the port-numbers specified by the configuration parameters *iiop_port* and *iiop_ssl_port*. Other ORB's may use several ports but it should be possible to change this behavior. Consult the other ORBs documentation.
- *Outgoing Requests* - Most ORB's, Orber included, ask the OS to supply a vacant local port when connecting to the server-side ORB. It is possible to change this behavior when using Orber (i.e. set the configuration parameter *iiop_out_ports*).

Warning:
Using the option `iiop_out_ports` may result in that Orber runs out of valid ports numbers. For example, other applications may steal some of the ports or the number of concurrent outgoing connections to other ORBs may be higher than expected. To reduce, but not eliminate, the risk you should use `iiop_connection_timeout`.

Firewall configuration example:

```
# "Plain" IIOP
To: Orber-IPNo:(iiop_port)     From: ORB-IPNo:X
To: ORB-IPNo:Z                 From: Orber-IPNo:(iiop_out_ports | Any Port)

# IIOP via SSL
To: Orber-IPNo:(iiop_port)     From: ORB-IPNo:X
To: Orber-IPNo:(iiop_ssl_port) From: ORB-IPNo:Y
To: ORB-IPNo:Z                 From: Orber-IPNo:(iiop_out_ports | Any Port)
```

If the communication take place via a TCP Firewall with NAT [page 19] (Network Address Translation), we must activate this behavior and define the external address and/or ports.

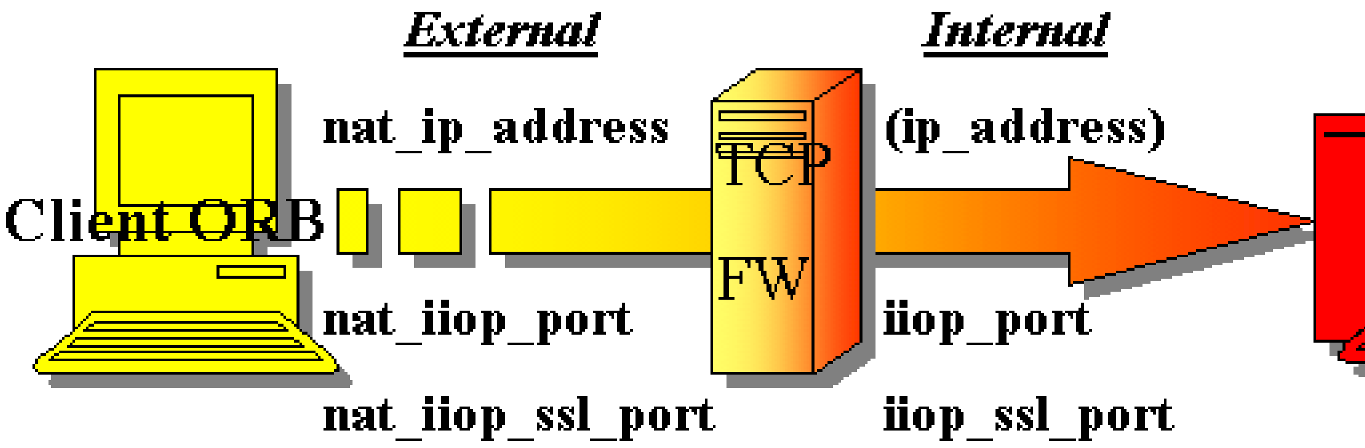

Figure 3.1: TCP Firewall With NAT

Using NAT makes it possible to use different host data for different network domains. This way we can share Internet Protocol address resources or obscure resources. To enable this feature the Enable NAT [page 17] flag must be set and `nat_iiop_port`, `nat_iiop_ssl_port` and `nat_ip_address` configured, which maps to `iiop_port`, `iiop_ssl_port` and `ip_address` respectively. Hence, the firewall must be configured to translate the external to the internal representation correctly. If these NAT parameters are assigned a single port number or IP address, only those will be used when an IOR is exported to another ORB. When `ip_address` is set to `{multiple, [IPAddress]}`, `nat_ip_address` should be configured in the same way, so that each NAT IP address can be translated to a valid address by the firewall. If objects are supposed to be accessible via different interfaces and port, see also Interface Configuration [page 20], the options `{local, DefaultNATIPAddress, [{IPAddress, NATIPAddress}]}` and/or `{local, DefaultNATPort, [{Port, NATPort}]}` shall be used. The default NAT IP address and port, should be translated to the value of `ip_address_local` and the default listen port by the firewall. If the IP address and/or port is not found in the list, the default values will be inserted in the IOR. The firewall must be able to translate these correctly.

If it is necessary to limit the access to an ORB within a secure network, but other applications running on the same host may not be blocked out, one can use a *Application Level* firewall or Orber Access Control List (ACL). The latter makes it possible for the user to define which hosts may communicate, either as server or client, with Orber. This is achieved by defining the configuration parameter iiop_acl [page 12]. The Classless Inter Domain Routing (CIDR) `Filter` determines which peer interfaces and ports the other ORB may use.

Filter	*Peer Interface(s)*	*Peer Port(s)*
”10.1.1.1”	10.1.1.1	any
”10.1.1.1/8”	10.0.0.0-10.255.255.255	any
”10.1.1.1/8#4001”	10.0.0.0-10.255.255.255	4001
”10.1.1.1/8#4001/5001”	10.0.0.0-10.255.255.255	4001-5001

Table 3.3: Orber ACL Filters

Orber ACL, also allows the user to define which local interface(s) may be used, but will not detect `spoofing`. The operation orber_acl:match/2/3 [page 161] makes it easy to verify whether access would be granted or not. For example, if Orber would be started with the ACL `[{tcp_out, "10.1.1.1/8#4001/5001"}]`, then `orber_acl:match/2` would behave as follows:

```
erl> orber_acl:match({11,1,1,1}, tcp_out).
false

erl> orber_acl:match({10,1,1,1}, tcp_out).
true

erl> orber_acl:match({11,1,1,1}, tcp_out, true).
{false,[],0}

erl> orber_acl:match({10,1,1,1}, tcp_out, true).
{true,[],{4001,5001}}
```

Only if the returned boolean is true the extra return values makes a difference. In the example above, `{true,[],{4001,5001}}` means that Orber may connect to `"10.1.1.1"`, using any local interface, if the server-side ORB listens for incoming connect requests on a port within the range 4001-5001. Note, invoking the `orber_acl:match/2/3` operation, will not result in a connect attempt by Orber. The reason for this, is that this function may be used on a live node as well as in test environment. Hence, if a local interface is currently not available or no server-side ORB available via the given host/port(s), will not be detected by Orber.

3.4 Interface Configuration

In many cases it is sufficient to simply configure the underlying OS which local interfaces shall be used for all applications. But, in some cases it is required, due to, for example, the firewall configuration, that different local interfaces are used for different applications. Some times, it is even necessary to use a specific interface for a single CORBA object. This section describe how one can alter this in different ways.

The default behavior is that Orber lets the OS configuration decide which interface will be added in IOR:s exported to another ORB and the local interface used when connecting to another ORB (Orber act as client side ORB). The latter can be overridden by setting the configuration parameters `iiop_ssl_ip_address_local` and/or `ip_address_local`, which will affect IIOP via SSL and IIOP respectively. These parameters can be overridden by using the Orber generic `interface` Context or defining an ACL (Access Control List). The latter always takes precedence if a local interface is included (e.g. `[{tcp_out, "10.0.0.0/8", ["10.0.0.1"]}]`). If the interface is excluded (e.g. `[{tcp_out, "10.0.0.0/8"}]`), the interface chosen will, in the following order, be determined by

#'IOP_ServiceContext'{}, ip_address_local/iiop_ssl_ip_address_local or the configuration of the underlying system.

Adding the interface context, for generated stubs/skeletons, is done in the following way:

```
Ctx = #'IOP_ServiceContext'{context_id = ?ORBER_GENERIC_CTX_ID,
                            context_data = {interface, "10.0.0.1"}},
'CosNaming_NamingContext':resolve(NS, [{context, [Ctx]}], Name),
```

It is also possible to add the context to corba:string_to_object/2, corba:resolve_initial_references/2, corba:resolve_initial_references_remote/3, corba:list_initial_services_remote/2, corba_object:not_existent/2, corba_object:non_existent/2 and corba_object:is_a/3. The operations exported by corba_object are affected if the supplied IOR is external. The function corba:string_to_object/2 might require the interface context if a corbaloc or a corbaloc string is passed (See the INS [page 49] chapter), while corba:resolve_initial_references_remote/3 and corba:list_initial_services_remote/2 always connect to another ORB and it might be necessary to add the context. The remaining corba operations are affected if calls are re-directed by setting the orbInitRef and/or orbDefaultInitRef configuration parameters. For more information, see the Reference Manual for each module.

Configuring which interface(s) that shall be used when exporting an IOR to another ORB, is determined by nat_ip_address, setting the flag 16#1000 [page 17] and ip_address, in that order. Orber listens for incoming connections either via all interfaces or the interface defined by ip_address. It is also possible to add and remove extra listen interfaces by using orber:add_listen_interface/2/3 and orber:remove_listen_interface/1. In this case one should set the 16#1000 flag and, if necessary, set the configuration parameters {local, DefaultNATIPAddress, [{IPAddress, NATIPAddress}]} and/or {local, DefaultNATPort, [{Port, NATPort}]}.

Chapter 4

4.1 OMG IDL to Erlang Mapping - Overview

The purpose of OMG IDL, *Interface Definition Language*, mapping is to act as translator between platforms and languages. An IDL specification is supposed to describe data types, object types etc.

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.

4.2 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.

4.3 Getting Started

To begin with, we should decide which type of objects (i.e. servers) we need and if two, or more, should export the same functionality. Let us assume that we want to create a system for DB (database) access for different kind of users. For example, anyone with a valid password may extract data, but only a few may update the DB. Usually, an application is defined within a `module`, and all global datatypes are defined on the top-level. To begin with we create a module and the interfaces we need:

```
// DB IDL
#ifndef _DB_IDL_
#define _DB_IDL_
// A module is simply a container
module DB {

  // An interface maps to a CORBA::Object.
  interface CommonUser {

  };

  // Inherit the Consumer interface
  interface Administrator : CommonUser {

  };

  interface Access {

  };

};
#endif
```

Since the `Administrator` should be able to do the same things as the `CommonUser`, the previous inherits from the latter. The `Access` interface will grant access to the DB. Now we are ready to define the functionality and data types we need. But, this requires that we know a little bit more about the OMG IDL.

> **Note:**
> The OMG defines a set of reserved case insensitive key-words, which may *NOT* be used as identifiers (e.g. module name). For more information, see Reserved Compiler Names and Keywords [page 42]

4.4 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	-2^15 .. 2^15-1
unsigned short	Erlang integer	0 .. 2^16-1
long	Erlang integer	-2^31 .. 2^31-1

continued ...

... continued

unsigned long	Erlang integer	0 .. 2^32-1
long long	Erlang integer	-2^63 .. 2^63-1
unsigned long long	Erlang integer	0 .. 2^64-1
char	Erlang integer	ISO-8859-1
wchar	Erlang integer	UTF-16 (ISO-10646-1:1993)
boolean	Erlang atom	true/false
octet	Erlang integer	
any	Erlang record	#any{typecode, value}
long double	Not supported	
Object	Orber object reference	Internal Representation
void	Erlang atom	ok

Table 4.1: 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 43], and the value field itself.

Functions with return type `void` will return the atom `ok`.

4.5 Template OMG IDL Types and Complex Declarators

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

Type	*IDL code*	*Maps to*	*Erlang code*
string	typedef string S; void op(in S a);	Erlang string	ok = op(Obj, ”Hello World”),
wstring	typedef wstring S; void op(in S a);	Erlang list of Integers	ok = op(Obj, ”Hello World”),
sequence	typedef sequence <long, 3> S; void op(in S a);	Erlang list	ok = op(Obj, [1, 2, 3]),
array	typedef string S[2]; void op(in S a);	Erlang tuple	ok = op(Obj, {”one”, ”two”}),
fixed	typedef fixed<3,2> myFixed; void op(in myFixed a);	Erlang tuple	MF = fixed:create(3, 2, 314), ok = op(Obj, MF),

Table 4.2: OMG IDL Template and Complex Declarators

4.5.1 String/WString Data Types

A `string` consists of all possible 8-bit quantities except null. Most ORB:s uses, including Orber, the character set Latin-1 (ISO-8859-1). The `wstring` type is represented as a list of integers, where each integer represents a wide character. In this case Orber uses, as most other ORB:s, the UTF-16 (ISO-10646-1:1993) character set.

When defining a a string or wstring they can be of limited length or null terminated:

```
// Null terminated
typedef string myString;
typedef wstring myWString;
// Maximum length 10
typedef string<10> myString10;
typedef wstring<10> myWString10;
```

If we want to define a char/string or wchar/wstring constant, we can use octal (\OOO - one, two or three octal digits), hexadecimal (\xHH - one or two hexadecimal digits) and unicode (\uHHHH - one, two, three or four hexadecimal digits.) representation as well. For example:

```
const string  SwedensBestSoccerTeam = "\101" "\x49" "\u004B";
const wstring SwedensBestHockeyTeam = L"\101\x49\u004B";
const char  aChar  = '\u004B';
const wchar aWchar = L'\u004C';
```

Naturally, we can use `"Erlang"`, `L"Rocks"`, `'A'` and `L'A'` as well.

4.5.2 Sequence Data Type

A sequence can be defined to be of a maximum length or unbounded, and may contain Basic and Template types and scoped names:

```
typedef sequence <short, 1> aShortSequence;
typedef sequence <long> aLongSequence;
typedef sequence <aLongSequence> anEvenLongerSequence;
```

4.5.3 Array Data Type

Arrays are multidimensional, fixed-size arrays. The indices is language mapping specific, which is why one should not pass them as arguments to another ORB.

```
typedef long myMatrix[2][3];
```

4.5.4 Fixed Data Type

A Fixed Point literal consists of an integer part (decimal digits), decimal point and a fraction part (decimal digits), followed by a `D` or `d`. Either the integer part or the fraction part may be missing; the decimal point may be missing, but not d/D. The integer part must be a positive integer less than 32. The Fraction part must be a positive integer less than or equal to the Integer part.

```
const fixed myFixed1 = 3.14D;
const fixed myFixed2 = .14D;
const fixed myFixed3 = 0.14D;
const fixed myFixed4 = 3.D;
const fixed myFixed5 = 3D;
```

It is also possible to use unary (+-) and binary (+-*/) operators:

```
const fixed myFixed6 = 3D + 0.14D;
const fixed myFixed7 = -3.14D;
```

The Fixed Point examples above are, so called, *anonymous* definitions. In later CORBA specifications these have been deprecated as function parameters or return values. Hence, we strongly recommend that you do not use them. Instead, you should use:

```
typedef fixed<5,3> myFixed53;
const myFixed53 myFixed53constant = 03.140d;
typedef fixed<3,2> myFixed32;
const myFixed32 myFixed32constant = 3.14d;

myFixed53 foo(in myFixed32 MF); // OK
void bar(in fixed<5,3> MF); // Illegal
```

For more information, see Fixed [page 140] in Orber's Reference Manual.

Now we continue to work on our IDL specification. To begin with, we want to limit the size of the logon parameters (Id and password). Since the `UserID` and `Password` paremeters, only will be used when invoking operations on the `Access` interface, we may choose to define them within the scope that interface. To keep it simple our DB will contain employee information. Hence, as the DB key we choose an integer (`EmployeeNo`).

```
// DB IDL
#ifndef _DB_IDL_
#define _DB_IDL_
module DB {

  typedef unsigned long EmployeeNo;

  interface CommonUser {

     any lookup(in EmployeeNo ENo);

  };

  interface Administrator : CommonUser {

     void delete(in EmployeeNo ENo);

  };

  interface Access {

      typedef string<10> UserID;
      typedef string<10> Password;

      CommonUser logon(in UserID ID, in Password PW);

  };

};
#endif
```

But what should, for example, the `lookup` operation return? One option is to use the `any` data type. But, depending on what kind of data it encapsulates, this datatype can be rather expensive to use. We might find a solution to our problems among the `Constructed` IDL types.

4.6 Constructed OMG IDL Types

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

Type	*IDL code*	*Maps to*	*Erlang code*
struct	struct myStruct { long a; short b; }; void op(in myStruct a);	Erlang record	ok = op(Obj, #'myStruct'{a=300, b=127}),
union	union myUnion switch(long) { case 1: long a; }; void op(in myUnion a);	Erlang record	ok = op(Obj, #'myUnion'{label=1, value=66}),
enum	enum myEnum {one, two}; void op(in myEnum a);	Erlang atom	ok = op(Obj, one),

Table 4.3: OMG IDL constructed types

4.6.1 Struct Data Type

A `struct` may have Basic, Template, Scoped Names and Constructed types as members.

4.6.2 Enum Data Type

The maximum number of identifiers which may defined in an enumeration is 2. The order in which the identifiers are named in the specification of an enumeration defines the relative order of the identifiers.

4.6.3 Union Data Type

A `union` may consist of:

- Identifier
- Switch - may be an integer, char, boolean, enum or scoped name.
- Body - with or without a `default` case; may appear at most once.

A case label must match the defined type of the discriminator, and may only contain a default case if the values given in the non-default labels do not cover the entire range of the union's discriminant type. For example:

```
// Illegal default; all cases covered by
// non-default cases.
union BooleanUnion switch(boolean) {
  case TRUE:  long TrueValue;
  case FALSE: long FalseValue;
  default: long DefaultValue;
};
```

```
// OK
union BooleanUnion2 switch(boolean) {
  case TRUE:  long TrueValue;
  default: long DefaultValue;
};
```

It is not necessary to list all possible values of the union discriminator in the body. Hence, the value of a union is the value of the discriminator and, in given order, one of the following:

1. If the discriminator match a label, explicitly listed in a case statement, the value must be of the same type.
2. If the union contains a default label, the value must match the type of the default label.
3. No value. Orber then inserts the Erlang atom `undefined` in the value field when receiving a union from an external ORB.

The above can be summed up to:

```
// If the discriminator equals 1 or 2 the value
// is a long. Otherwise, the atom undefined.
union LongUnion switch(long) {
  case 1:
  case 2:  long TrueValue;
};
// If the discriminator equals 1 or 2 the value
// is a long. Otherwise, a boolean.
union LongUnion2 switch(long) {
  case 1:
  case 2:  long TrueValue;
  default: boolean DefaultValue;
};
```

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 opertion with #'myStruct'{a=300} will fail (equal to #'myStruct'{a=300, b=undefined})

Now we can continue to work on our IDL specification. To begin with, we should determine the return value of the `lookup` operation. Since the `any` type can be rather expensive we can use a `struct` or a `union` instead. If we intend to return the same information about a employee every time we can use a struct. Let us assume that the DB contains the name, address, employee number and department.

```
// DB IDL
#ifndef _DB_IDL_
#define _DB_IDL_
module DB {

  typedef unsigned long EmployeeNo;
```

```
  enum Department {Department1, Department2};

  struct employee {
        EmployeeNo No;
        string Name;
        string Address;
        Department Dpt;
  };

  typedef employee EmployeeData;

  interface CommonUser {

     EmployeeData lookup(in EmployeeNo ENo);

  };

  interface Administrator : CommonUser {

     void delete(in EmployeeNo ENo);

  };

  interface Access {

      typedef string<10> UserID;
      typedef string<10> Password;

      // Since Administrator inherits from CommonUser
      // the returned Object can be of either type.
      CommonUser logon(in UserID ID, in Password PW);

  };

};
#endif
```

We can also define exceptions (i.e. not system exception) thrown by each interface. Since exceptions are thoroughly described in the chapter System and User Defined Exceptions [page 71], we choose not to. Hence, we are now ready to compile our IDL-file by invoking:

```
$ erlc DB.idl
```

or:

```
$ erl
Erlang (BEAM) emulator version 5.1.1 [threads:0]

Eshell V5.1.1  (abort with ^G)
1> ic:gen('DB').
ok
```

```
2> halt().
```

The next step is to implement our servers. But, to be able to do that, we need to know how we can access data type definitions. For example, since a struct is mapped to an Erlang record we must include an hrl-file in our callback module.

4.7 Scoped Names and Generated Files

4.7.1 Scoped Names

Within a scope all identifiers must be unique. The following kinds of definitions form scopes in the OMG IDL:

- *module*
- *interface*
- *operation*
- *valuetype*
- *struct*
- *union*
- *exception*

For example, since enumerants do not form a scope, the following IDL code is not valid:

```
module MyModule {
     // 'two' is not unique
     enum MyEnum {one, two};
     enum MyOtherEnum {two, three};
};
```

But, since Erlang only has two levels of scope, *module* and *function*, the OMG IDL scope is mapped as follows:

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

An Erlang module, corresponding to an IDL global name, is derived by converting occurencies of “::” to underscore, and eliminating the leading “::”. Hence, accessing `MyEnum` from another module, one use `MyModule::MyEnum`

For example, an operation `foo` defined in interface `I`, which is defined in module `M`, would be written in IDL as `M::I::foo` and as `'M_I':foo` in Erlang - `foo` is the function name and `'M_I'` is the name of the Erlang module. Applying this knowledge to a stripped version of the DB.idl gives:

```
// DB IDL
#ifndef _DB_IDL_
#define _DB_IDL_
// ++ topmost scope ++
// IC generates oe_XX.erl and oe_XX.hrl.
// XX is equal to the name of the IDL-file.
// Tips: create one IDL-file for each top module
// and give the file the same name (DB.idl).
// The oe_XX.erl module is used to register data
// in the IFR.
module DB {

  // ++ Module scope ++
  // To access 'EmployeeNo' from another scope, use:
  // DB::EmployeeNo, DB::Access etc.
  typedef unsigned long EmployeeNo;

  enum Department {Department1, Department2};

  // Definitions of this struct is contained in:
  // DB.hrl
  // Access functions exported by:
  // DB_employee.erl
  struct employee {
     ... CUT ...
  };

  typedef employee EmployeeData;

  ... CUT ...

  // If this interface should inherit an interface
  // in another module (e.g. OtherModule) use:
  // interface Access : OtherModule::OtherInterface
  interface Access {

      // ++ interface scope ++
      // Types within this scope is accessible via:
      // DB::Access::UserID
      // The Stub/Skeleton for this interface is
      // placed in the module:
      // DB_Access.erl
      typedef string<10> UserID;
      typedef string<10> Password;

      // Since Administrator inherits from CommonUser
      // the returned Object can be of either type.
      // This operation is exported from:
      // DB_Access.erl
      CommonUser logon(in UserID ID, in Password PW);

  };
```

```
};
#endif
```

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. For example, the following definition would generate two structures namned x_y_z.

```
module x {

    struct y_z {
        ...
    };

    interface y {

        struct z {
            ...
        };
    };
};
```

4.7.2 Generated 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_.

After compiling DB.idl, the following files have been generated:

- oe_DB.hrl and oe_DB.erl for the top scope level.
- DB.hrl for the module DB.
- DB_Access.hrl and DB_Access.erl for the interface DB_Access.
- DB_CommonUser.hrl and DB_CommonUser.erl for the interface DB_CommonUser.
- DB_Administrator.hrl and DB_Administrator.erl for the interface DB_Administrator.
- DB_employee.erl for the structure employee in module DB.

Since the employee struct is defined in the top level scope, the Erlang record definition is found in DB.hrl. IC also generates stubs/skeletons (e.g. DB_CommonUser.erl) and access functions for some datatypes (e.g. DB_employee.erl). How the stubs/skeletons are used is thoroughly described in Stubs/Skeletons [page 64] and Module_Interface [page 122].

4.8 Typecode, Identity and Name Access Functions

As mentioned in a previous section, struct, union and exception types yield 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. For example, the module DB_employee.erl, representing the employee struct, exports the following functions:

- tc/0 - returns the type code for the struct.
- id/0 - returns the IFR identity of the struct. In this case the returned value is "IDL:DB/employee:1.0", but if the struct was defined in the scope of CommonUser, the result would be "IDL:DB/CommonUser/employee:1.0". However, the user usually do not need to know the Id, just which Erlang module contains the correct Id.
- name/0 - returns the scoped name of the struct. The employee struct name is "DB_employee".

Type Codes are, for example, used in Any [page 128] values. Hence, we can encapsulate the employee struct in an any type by:

```
%% Erlang code
....
AnEmployee = #'DB_employee'{'No'      = 1,
                            'Name'    = "Adam Ivan Kendall",
                            'Address' = "Rasunda, Solna",
                            'Dpt'     = 'Department1'},
EmployeeTC = 'DB_employee':tc(),
EmployeeAny = any:create(EmployeeTC, AnEmployee),
....
```

For more information, see the Type Code listing [page 43].

4.9 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:

```
// m.idl
module m {
    const float pi = 3.14;

    interface i {
        const float pi = 3.1415;
    };
};
```

Since the two constants are defined in different scopes, the IDL code above is valid, but not necessarily a good approach. After compiling `m.idl`, the constant definitions can be extracted by invoking:

```
$ erlc m.idl
$ erlc m.erl
$ erl
Erlang (BEAM) emulator version 5.1.1 [threads:0]

Eshell V5.1.1  (abort with ^G)
1> m:pi().
3.14
2> m_i:pi().
3.1415
3> halt().
```

4.10 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 36].

4.11 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 71] 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.

4.12 Access to Attributes

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

```
readonly attribute long RAttribute;
attribute long RWAttribute;
```

The `RAttribute` requires that you implement, in your call-back module, `_get_RAttribute`. For the `RWAttribute` it is necessary to implement `_get_RWAttribute` and `_set_RWAttribute`.

4.13 Invocations of Operations

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 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).

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
module m {
  interface i {
      readonly attribute long RAttribute;
      attribute long RWAttribute;
      long foo(in short a);
      long bar(in char c, inout string s, out long count);
      void baz(out long Id);
  };
};
```

Is used in Erlang as :

```
%% Erlang code
....
Obj = ...    %% get object reference
RAttr  = m_i:'_get_RAttribute'(Obj),
RWAttr = m_i:'_get_RWAttribute'(Obj),
ok = m_i:'_set_RWAttribute'(Obj, Long),
R1 = m_i:foo(Obj, 55),
{R2, S, Count} = m_i:bar(Obj, $a, "hello"),
....
```

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 `ok`, Orber's representation of the IDL-type `void`, must be returned by `baz` and `'_set_RWAttribute'`. These operations can be implemented in the call-back module as:

```
'_set_RWAttribute'(State, Long) ->
    {reply, ok, State}.

'_get_RWAttribute'(State) ->
    {reply, Long, State}.

'_get_RAttribute'(State) ->
    {reply, Long, State}.

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

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

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

The operations may require more arguments (depends on IC options used). For more information, see Stubs/Skeletons [page 64] and Module_Interface [page 122].

Warning:
A function can also be defined to be `oneway`, i.e. asynchronous. But, since the behavior of a oneway operation is not defined in the OMG specifications (i.e. the behavior can differ depending on which other ORB Orber is communicating with), one should avoid using it.

4.14 Implementing the DB Application

Now we are ready to implement the call-back modules. There are three modules we must create:

- DB_Access_impl.erl
- DB_CommonUser_impl.erl
- DB_Administrator_impl.erl

An easy way to accomplish that, is to use the IC backend `erl_template`, which will generate a complete call-back module. One should also add the same compile options, for example `this` or `from`, used when generating the stub/skeleton modules:

```
$> erlc +"{be,erl_template}" DB.idl
```

We begin with implementing the `DB_Access_impl.erl` module, which, if we used `erl_template`, will look like the following. All we need to do is to add the logic to the `logon` operation.

```
%%----------------------------------------------------------------------
%% <LICENSE>
%%
%%     $Id$
%%
%%----------------------------------------------------------------------
%% Module       : DB_Access_impl.erl
%%
%% Source       : /home/user/example/DB.idl
%%
%% Description  :
%%
%% Creation date: 2005-05-20
%%
%%----------------------------------------------------------------------
-module('DB_Access_impl').

-export([logon/3]).

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

%%----------------------------------------------------------------------
%% Include Files
%%----------------------------------------------------------------------


%%----------------------------------------------------------------------
%% Macros
%%----------------------------------------------------------------------


%%----------------------------------------------------------------------
%% Records
%%----------------------------------------------------------------------
-record(state, {}).

%%======================================================================
%% API Functions
%%======================================================================
%%----------------------------------------------------------------------
%% Function   : logon/3
%% Arguments  : State - term()
%%              ID = String()
%%              PW = String()
%% Returns    : ReturnValue = OE_Reply
%%              OE_Reply = Object_Ref()
%% Raises     :
```

```
%% Description:
%%----------------------------------------------------------------------
logon(State, ID, PW) ->
        %% Check if the ID/PW is valid and what
        %% type of user it is (Common or Administrator).
        OE_Reply
            = case check_user(ID, PW) of
                 {ok, administrator} ->
                    'DB_Administrator':oe_create();
                 {ok, common} ->
                    'DB_CommonUser':oe_create();
                 error ->
                    %% Here we should throw an exception
                   corba:raise(....)
        end,
        {reply, OE_Reply, State}.

%%======================================================================
%% Internal Functions
%%======================================================================
%%----------------------------------------------------------------------
%% Function   : init/1
%% Arguments  : Env = term()
%% Returns    : {ok, State}          |
%%              {ok, State, Timeout} |
%%              ignore               |
%%              {stop, Reason}
%% Raises     : -
%% Description: Initiates the server
%%----------------------------------------------------------------------
init(_Env) ->
        {ok, #state{}}.


%%----------------------------------------------------------------------
%% Function   : terminate/2
%% Arguments  : Reason = normal | shutdown | term()
%%              State = term()
%% Returns    : ok
%% Raises     : -
%% Description: Invoked when the object is terminating.
%%----------------------------------------------------------------------
terminate(_Reason, _State) ->
        ok.


%%----------------------------------------------------------------------
%% Function   : code_change/3
%% Arguments  : OldVsn = undefined | term()
%%              State = NewState = term()
%%              Extra = term()
%% Returns    : {ok, NewState}
%% Raises     : -
```

```
%% Description: Invoked when the object should update its internal state
%%              due to code replacement.
%%----------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
        {ok, State}.


%%----------------------------------------------------------------------
%% Function   : handle_info/2
%% Arguments  : Info = normal | shutdown | term()
%%              State = NewState = term()
%% Returns    : {noreply, NewState}          |
%%              {noreply, NewState, Timeout} |
%%              {stop, Reason, NewState}
%% Raises     : -
%% Description: Invoked when, for example, the server traps exits.
%%----------------------------------------------------------------------
handle_info(_Info, State) ->
        {noreply, State}.
```

Since `DB_Administrator` inherits from `DB_CommonUser`, we must implement `delete` in the `DB_Administrator_impl.erl` module, and `lookup` in `DB_Administrator_impl.erl`*and*`DB_CommonUser_impl.erl`. But wait, is that really necessary? Actually, it is not. We simple use the IC compile option *impl*:

```
$ erlc +'{{impl, "DB::CommonUser"}, "DBUser_impl"}'  +'{{impl, "DB::Administrator"}, "DBUser_impl"}' DB.
$ erlc *.erl
```

Instead of creating, and not the least, maintaining two call-back modules, we only have to deal with `DBUser_impl.erl`. If we generated the templates, we simply rename `DB_Administrator_impl.erl` to `DBUser_impl.erl`. See also the Exceptions [page 71] chapter. In the following example, only the implementation of the API functios are shown:

```
%%======================================================================
%% API Functions
%%======================================================================
%%----------------------------------------------------------------------
%% Function   : delete/2
%% Arguments  : State - term()
%%              ENo = unsigned_Long()
%% Returns    : ReturnValue = ok
%% Raises     :
%% Description:
%%----------------------------------------------------------------------
delete(State, ENo) ->
        %% How we access the DB, for example mnesia, is not shown here.
        case delete_employee(No) of
            ok ->
                {reply, ok, State};
            error ->
                %% Here we should throw an exception if
```

```
                %% there is no match.
                corba:raise(....)
        end.

%%----------------------------------------------------------------------
%% Function   : lookup/2
%% Arguments  : State - term()
%%              ENo = unsigned_Long()
%% Returns    : ReturnValue = OE_Reply
%%              OE_Reply = #'DB_employee'{No,Name,Address,Dpt}
%%              No = unsigned_Long()
%%              Name = String()
%%              Address = String()
%%              Dpt = Department
%%              Department = 'Department1' | 'Department2'
%% Raises     :
%% Description:
%%----------------------------------------------------------------------
lookup(State, ENo) ->
        %% How we access the DB, for example mnesia, is not shown here.
        case lookup_employee(ENo) of
            %% We assume that we receive a 'DB_employee' struct
            {ok, Employee} ->
                OE_Reply = Employee,
                {reply, OE_Reply, State};
            error ->
                %% Here we should throw an exception if
                %% there is no match.
                corba:raise(....)
        end.
```

After you have compiled both call-back modules, and implemented the missing functionality (e.g. lookup_employee/1), we can test our application:

```
%% Erlang code
....
%% Create an Access object
Acc = 'DB_Access':oe_create(),

%% Login is Common user and Administrator
Adm = 'DB_Access':logon(A, "admin", "pw"),
Com = 'DB_Access':logon(A, "comm", "pw"),

%% Lookup existing employee
Employee = 'DB_Administrator':lookup(Adm, 1),
Employee = 'DB_CommonUser':lookup(Adm, 1),

%% If we try the same using the DB_CommonUser interface
%% it result in an exit since that operation is not exported.
{'EXIT', _} = (catch 'DB_CommonUser':delete(Adm, 1)),

%% Try to delete the employee via the CommonUser Object
{'EXCEPTION', _} = (catch 'DB_Administrator':delete(Com, 1)),
```

```
%% Invoke delete operation on the Administrator object
ok = 'DB_Administrator':delete(Adm, 1),
....
```

4.15 Reserved Compiler Names and Keywords

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 `OE_` 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.

The OMG defines a set of reserved words, shown below, for use as keywords. These may *not* be used as, for example, identifiers. The keywords which are not in bold face was introduced in the OMG CORBA-3.0 specification.

abstract	*exception*	*inout*	provides	*truncatable*
any	emits	*interface*	*public*	*typedef*
attribute	*enum*	*local*	publishes	typeid
boolean	eventtype	*long*	*raises*	typeprefix
case	*factory*	*module*	*readonly*	*unsigned*
char	*FALSE*	multiple	setraises	*union*
component	finder	*native*	*sequence*	uses
const	*fixed*	*Object*	*short*	*ValueBase*
consumes	*float*	*octet*	*string*	*valuetype*
context	getraises	*oneway*	*struct*	*void*
custom	home	*out*	*supports*	*wchar*
default	import	primarykey	*switch*	*wstring*
double	*in*	*private*	*TRUE*	

Table 4.4: OMG IDL keywords

The keywords listed above must be written exactly as shown. Any usage of identifiers that collide with a keyword is illegal. For example, *long* is a valid keyword; *Long* and *LONG* are illegal as keywords and identifiers. But, since the OMG must be able to expand the IDL grammar, it is possible to use *Escaped Identifiers*. For example, it is not unlikely that `native` have been used in IDL-specifications as identifiers. One option is to change all occurances to `myNative`. Usually, it is necessary to change programming language code that depends upon that IDL as well. Since Escaped Identifiers just disable type checking (i.e. if it is a reserved word or not) and leaves everything else unchanged, it is only necessary to update the IDL-specification. To escape an identifier, simply prefix it with _. The following IDL-code is illegal:

```
typedef string native;
interface i {
   void foo(in native Arg);
   };
};
```

With Escaped Identifiers the code will look like:

```
typedef string _native;
interface i {
   void foo(in _native Arg);
   };
};
```

4.16 Type Code Representation

Type Codes are used in `any` values. To avoid mistakes, you should use access functions exported by the Data Types modules (e.g. struct, union etc) or the orber_tc [page 179] module.

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	
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_fixed, Digits, Scale}	{tk_fixed, 3, 2}

continued ...

... continued

{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 4.5: Type Code tuples

Chapter 5

5.1 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 directed graph with notes 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.

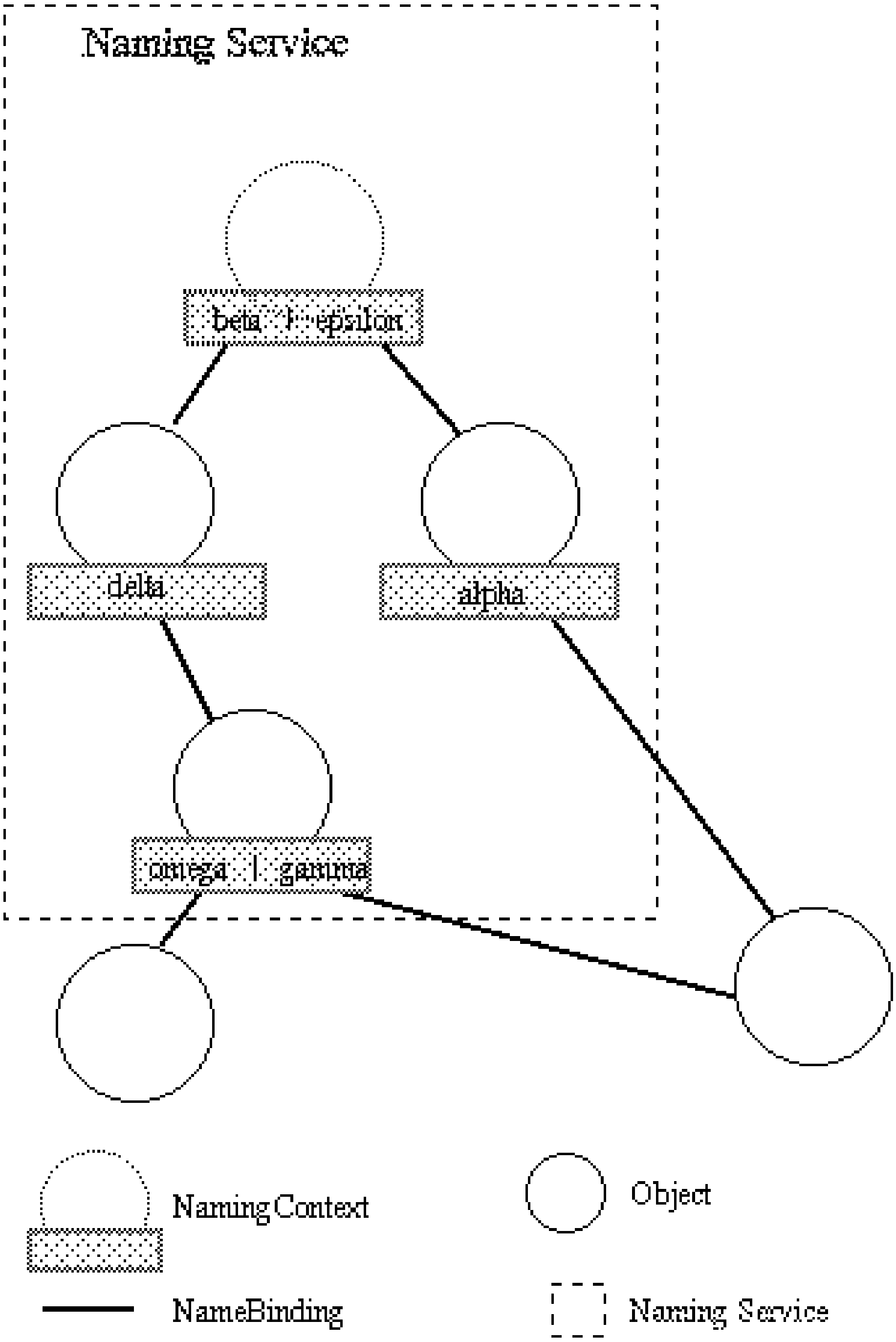

Figure 5.1: Figure 1: Contextual object relationships using the Naming Service.

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.

5.2 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.

5.2.1 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_references("NameService").
```

> **Note:**
> NS in the other use-cases refers to this initial reference.

5.2.2 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"])).
```

5.2.3 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.

5.2.4 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 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.

5.2.5 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).
```

5.2.6 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).
   ```

5.3 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

5.3.1 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.

5.3.2 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 2809 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").
```

String Name	*Object Type*
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 5.1: Currently reserved key strings

5.3.3 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:

String Name	*Name Sequence*	*Comment*
”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 5.2: 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.

Chapter 6

6.1 Security in Orber

6.1.1 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.

6.1.2 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.

6.1.3 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_cacertfile* - 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.
- *ssl_server_keyfile* - which is a path to a file containing a PEM encoded key for the Orber domain as server.
- *ssl_server_password* - only used if the private keyfile is password protected.
- *ssl_server_ciphers* - which is string of ciphers as a colon separated list of ciphers.
- *ssl_server_cachetimeout* - which is the session cache timeout in seconds.

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

- orber:ssl_server_certfile/0
- orber:ssl_server_cacertfile/0
- orber:ssl_server_verify/0
- orber:ssl_server_depth/0
- orber:ssl_server_keyfile/0
- orber:ssl_server_password/0
- orber:ssl_server_ciphers/0
- orber:ssl_server_cachetimeout/0

6.1.4 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_cacertfile* - which is a path to a file containing a chain of PEM encoded CA 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.
- *ssl_client_keyfile* - which is a path to a file containing a PEM encoded key when Orber act as client side ORB.
- *ssl_client_password* - only used if the private keyfile is password protected.
- *ssl_client_ciphers* - which is string of ciphers as a colon separated list of ciphers.
- *ssl_client_cachetimeout* - which is the session cache timeout in seconds.

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_cacertfile/0
- orber:ssl_client_verify/0
- orber:ssl_client_depth/0
- orber:ssl_client_keyfile/0
- orber:ssl_client_password/0
- orber:ssl_client_ciphers/0
- orber:ssl_client_cachetimeout/0

Modify functions:

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

Chapter 7

Service Implementation

This chapter describe how to implement Orber based CORBA services.

7.1 Orber Examples

7.1.1 A Tutorial on How to Create a Simple Service

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`.

```
%%----------------------------------------------------------------------
%%
%% %CopyrightBegin%
%%
%% Copyright Ericsson AB 1997-2009. All Rights Reserved.
%%
%% The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved online at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% %CopyrightEnd%
%%
%%
%%----------------------------------------------------------------------
%% StackModule_Stack_impl example file.

-module('StackModule_Stack_impl').
-include_lib("orber/include/corba.hrl").
-include("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.

```
%%----------------------------------------------------------------------
%%
%% %CopyrightBegin%
%%
%% Copyright Ericsson AB 1997-2009. All Rights Reserved.
%%
%% The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved online at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% %CopyrightEnd%
%%
%%
%%----------------------------------------------------------------------
%% 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.

```
%%----------------------------------------------------------------------
%%
%% %CopyrightBegin%
%%
%% Copyright Ericsson AB 1997-2009. All Rights Reserved.
%%
%% The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved online at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% %CopyrightEnd%
%%
%%
%%-------------------------------------------------------------------
%% 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.

```
%%----------------------------------------------------------------------
%%
```

```
%% %CopyrightBegin%
%%
%% Copyright Ericsson AB 1997-2009. All Rights Reserved.
%%
%% The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved online at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% %CopyrightEnd%
%%
%%
%%-----------------------------------------------------------------
%% stack_client example file.

-module('stack_client').

-export([run/0]).


run() ->
    case catch corba:string_to_object("corbaname:rir:/NameService#StackFactory") of
        {'EXCEPTION', _E} ->
            io:format("The stack factory server is not registered~n",[]);
        SF ->
            %% Create the stack
            SS = 'StackModule_StackFactory':create_stack(SF),

            '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:

```
/*
 * %CopyrightBegin%
 *
 * Copyright Ericsson AB 1997-2009. All Rights Reserved.
 *
 * The contents of this file are subject to the Erlang Public License,
 * Version 1.1, (the "License"); you may not use this file except in
 * compliance with the License. You should have received a copy of the
 * Erlang Public License along with this software. If not, it can be
 * retrieved online at http://www.erlang.org/.
 *
 * Software distributed under the License is distributed on an "AS IS"
 * basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
 * the License for the specific language governing rights and limitations
 * under the License.
 *
 * %CopyrightEnd%
 *
 */
/*
 * Stack example.
 */

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;
       }
    }
}
```

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 (e.g. JavaIDL). The example below is based on JDK-1.4.

```
fingolfin 127> erl
Erlang (BEAM) emulator version 5.5.4.3 [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.5.4.3  (abort with ^G)
1> ic:gen(stack).
Erlang IDL compiler version 4.2.12
ok
2> make:all().
Recompile: StackModule_EmptyStack
Recompile: StackModule_Stack
Recompile: StackModule_StackFactory
Recompile: StackModule_StackFactory_impl
Recompile: StackModule_Stack_impl
```

```
Recompile: oe_stack
Recompile: stack_client
Recompile: stack_factory
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> idlj stack.idl
fingolfin 129> javac StackModule/*.java
fingolfin 130> javac *.java
fingolfin 131> cp StackClient.class StackModule/
```

How to Run Everything

Below is a short transcript on how to run Orber.

```
fingolfin 143> erl
Erlang (BEAM) emulator version 5.5.4.3 [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.5.4.3  (abort with ^G)
1> orber:jump_start([{interceptors, {native, [orber_iiop_tracer_silent]}}]).
ok
2> oe_stack:oe_register().
ok
3> stack_factory:start().
ok
4> stack_client:run().
1
1
7
4
ok
5>
```

Before testing the Java part of this example generate and compile Java classes for `orber/examples/stack.idl` as seen in the build example. To run the Java client use the following command:

```
fingolfin 38> java StackModule.StackClient "corbaname::localhost:4001#StackFactory"
1
1
7
4
Empty stack
fingolfin 39>
```

7.2 Orber Stubs/Skeletons

7.2.1 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 a system exception is raised.
- 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 a system exception is raised.
- 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 the 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 server reply `Timeout` has no effect.
- The compile option `from` has 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 ..) ->
%% IC options this and from
own_function(This, From, State, .. Arguments ..) ->
%% IC option from
own_function(From, State, .. Arguments ..) ->
    %% Send explicit reply to client.
    corba:reply(From, Reply),
    %%--- Possible replies ---
    {noreply, State}
    {noreply, State, Timeout}


%% 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 ------------------------------
```

Chapter 8

8.1 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 operations raises exception IDL-definition.

8.1.1 Status Field

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

Status	*Description*
'COMPLETED_YES'	The operation was invoked on the target object but an error occurred after the object replied. This occur, 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 occur, 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 8.1: Table 1: System Exceptions Status

8.1.2 Minor Field

The minor field contains an integer (VMCID), which is related to a more specific reason why an invocation failed. The function `orber:exception_info/1` can be used to map the minor code to a string. Note, for VMCID:s not assigned by the OMG or Orber, the documentation for that particular ORB must be consulted.

8.1.3 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 has been invoked operations 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 are allowed to create.
- *'INTERNAL'* - an internal failure occurred in an ORB, which is unrecognized. You may consider contacting the ORB providers 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 objet 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.

8.2 User Defined Exceptions

User exceptions is defined in IDL-files and is listed in operations 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();
  };
};
```

8.3 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.
```

8.4 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 bee 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.
```

Chapter 9

9.1 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.

9.1.1 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.

One can also active and deactivate an interceptor during run-time, but this will only affect currently existing connections. For more information, consult Orber's Reference Manual regarding the operations `orber:activate_audit_trail/0/1` and `orber:activate_audit_trail/0/1`.

9.1.2 Creating Interceptors

Each supplied interceptor *must* export the following functions:

- *new_out_connection/3/5* - one of these operations is called when a client application calls an object residing on remote ORB. If an interceptor exports both versions, arity 3 and 5, which operation that will be invoked is Orber internal.
- *new_in_connection/3/5* - one of these operations is invoked when a client side ORB tries to set up a connection to the target ORB. If an interceptor exports both versions, arity 3 and 5, which operation that will be invoked is Orber internal.
- *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 encode.
- *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.

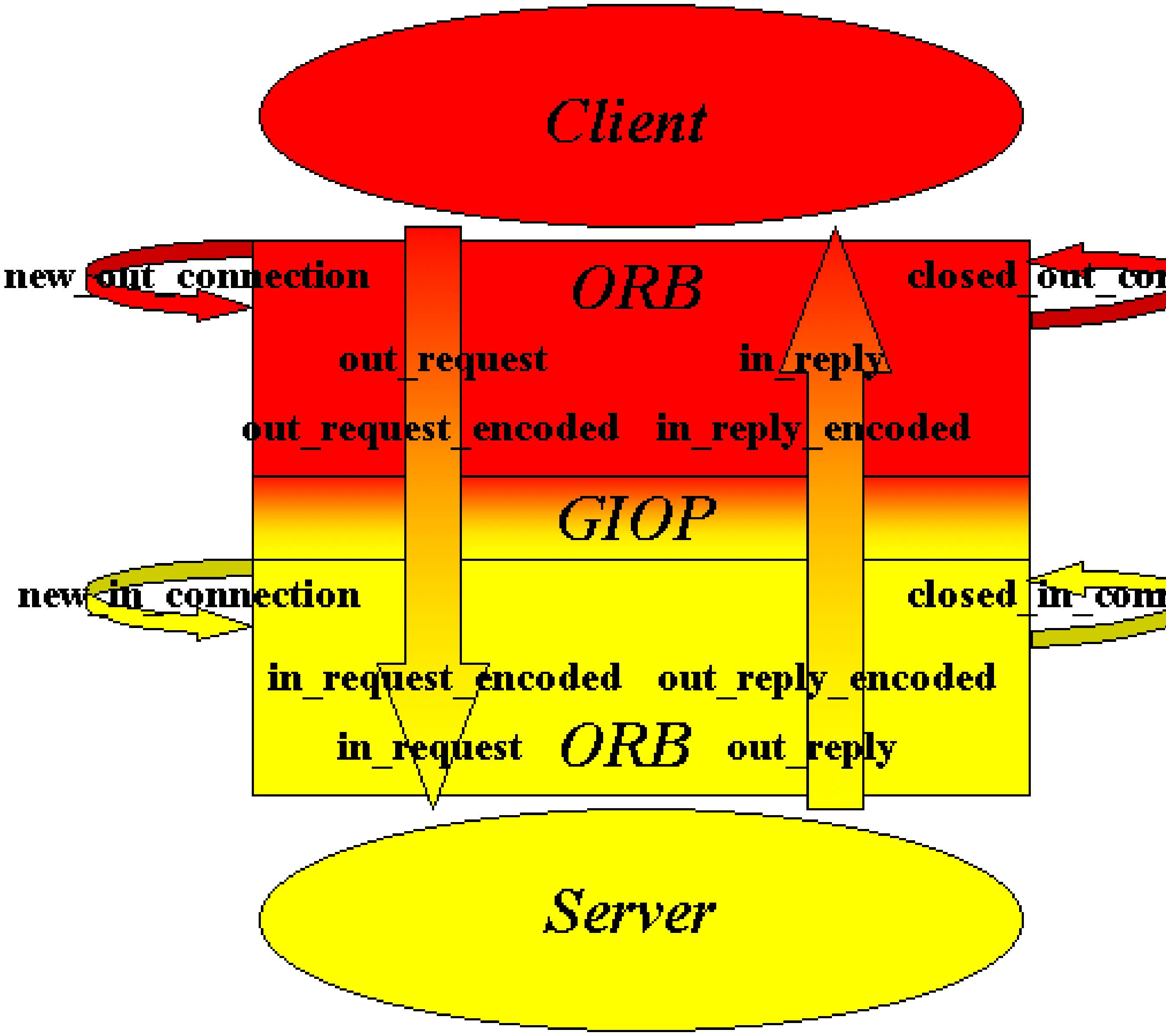

Figure 9.1: The Invocation Order of Interceptor Functions.

9.2 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");
         [SomeData] ->
            {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 behavior *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");
         [SomeData] ->
            {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.

Chapter 10

Tools, Debugging and FAQ

This chapter describe the available tools and debugging facilities for Orber. Also contain a FAQ listing of the most common mistakes.

10.1 OrberWeb

10.1.1 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 to access `Orber` nodes supporting the Interoperable Naming Service. How to start OrberWeb is described in Starting OrberWeb [page 90]

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.

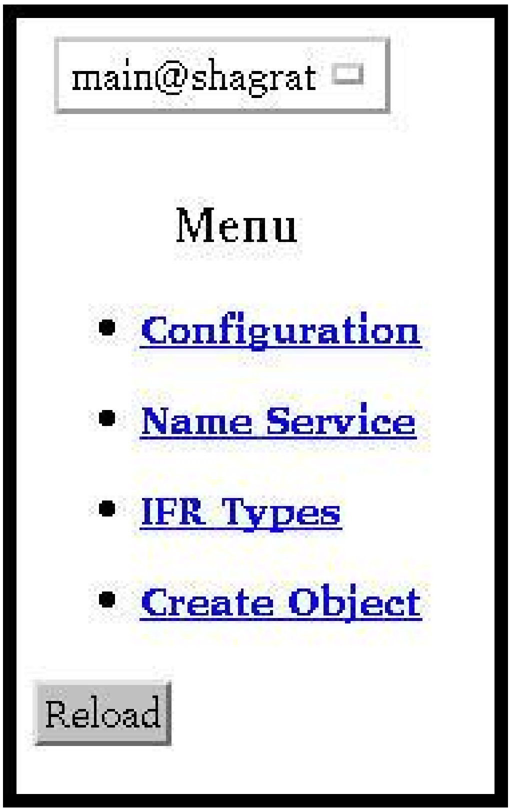

Figure 10.1: The Menu Frame.

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 12] for the target node.

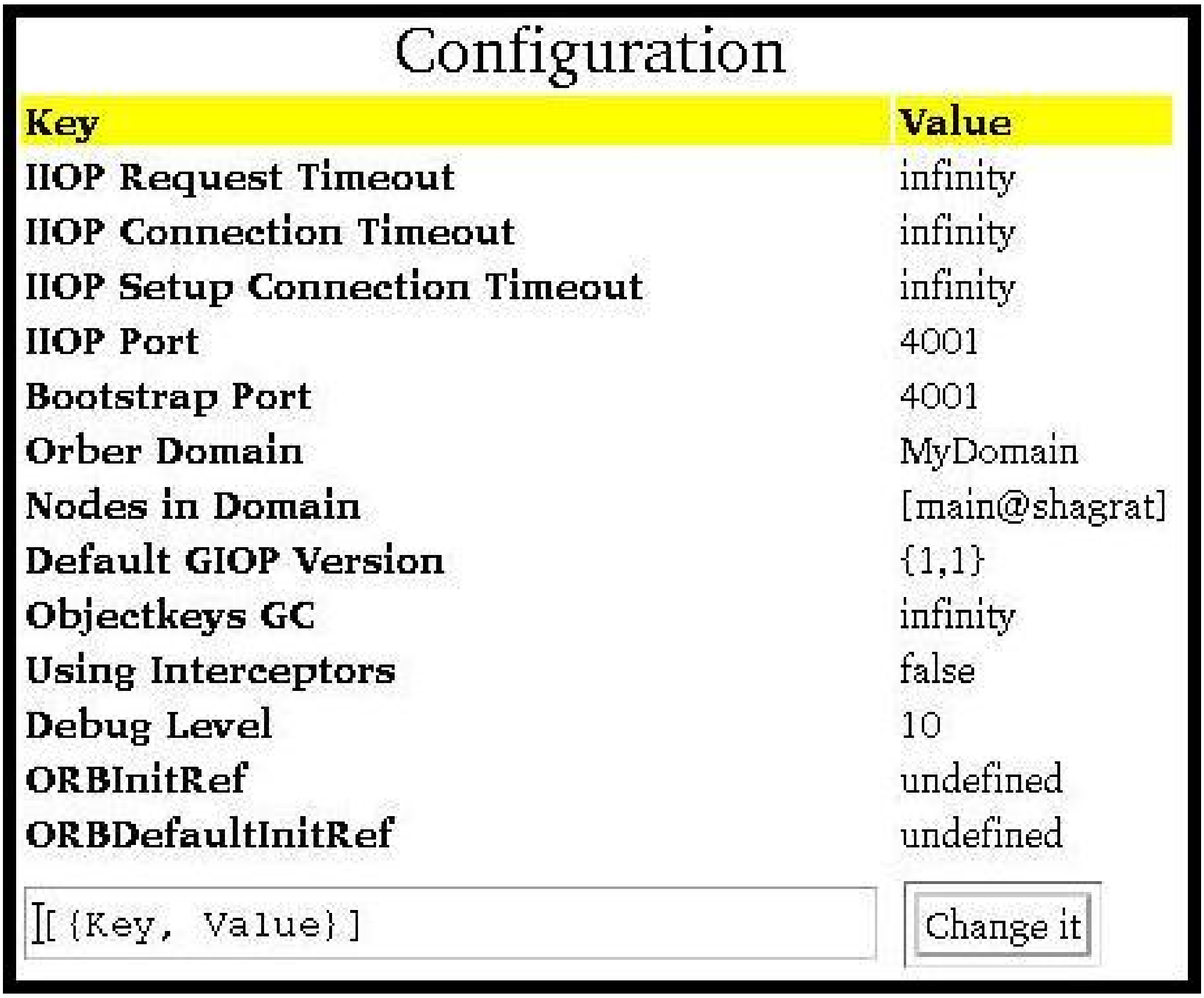

Figure 10.2: Configuration Settings.

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 frame:

Figure 10.3: 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.

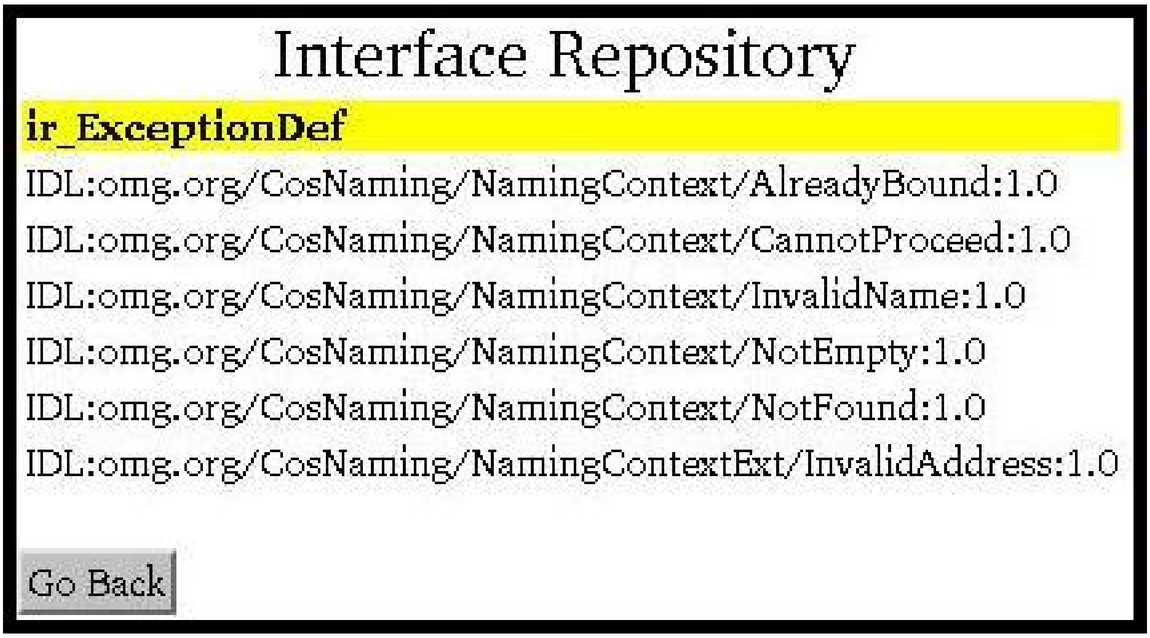

Figure 10.4: List Registered Exceptions.

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 45]. 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 successful, 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*.

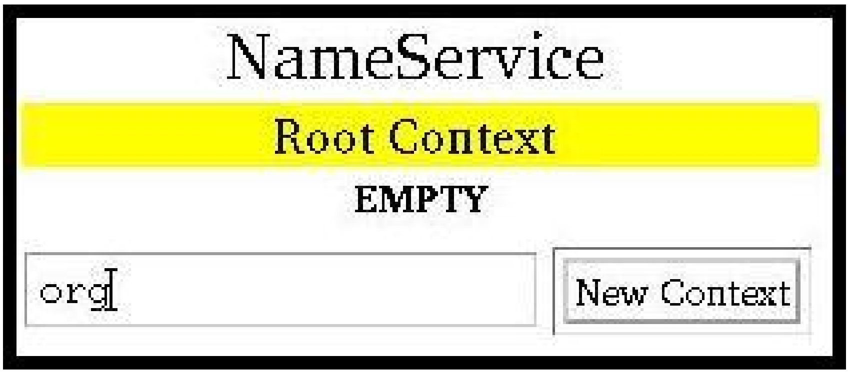

Figure 10.5: Add a New Context.

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.

Figure 10.6: 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 88]. For now, we will just assume that an object have been created and bound as *TheObjectName*.

Figure 10.7: 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*.

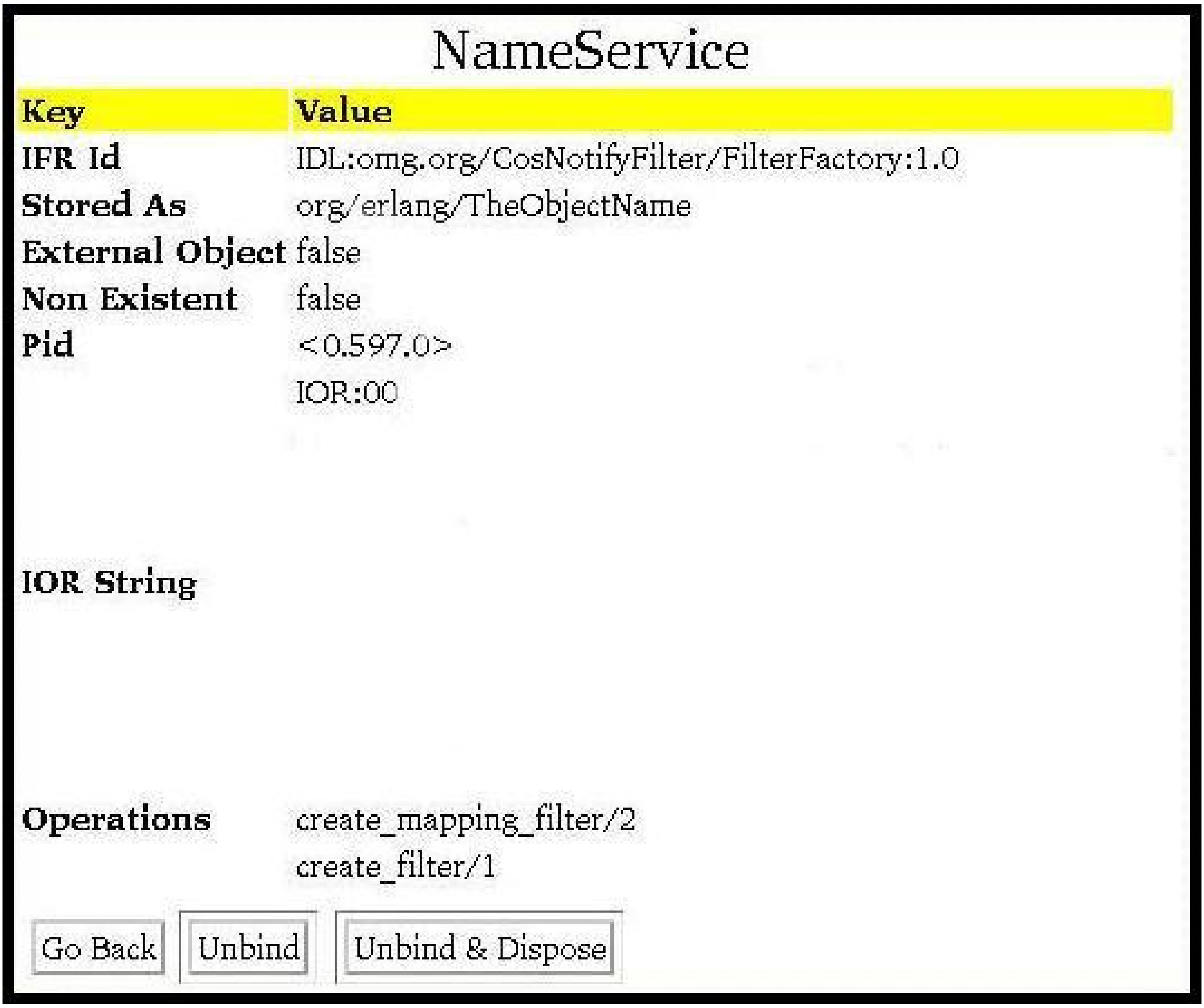

Figure 10.8: Object Data.

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 and 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.

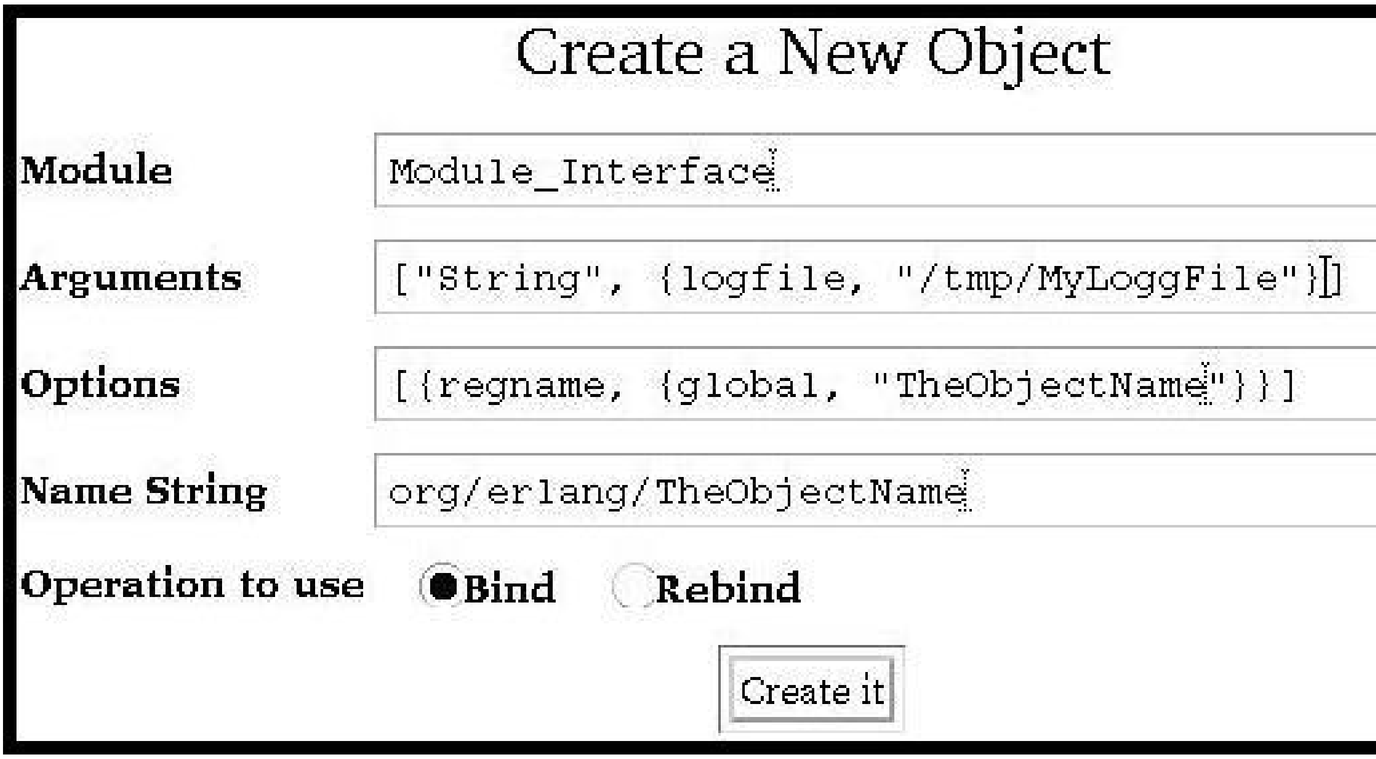

Figure 10.9: Create a New Object.

- *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 Module_Interface [page 122] 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 behavior 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.

10.1.2 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 availible 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.

10.2 Debugging

10.2.1 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 i 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,truce}]);
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 easy to forget to, for example, set all fields in a struct, which one may not discover when developing an application using Orber. When using a typed language, such faults would cause a compile time error. To avoid these mistakes, Orber allows the user to activate automatic typechecking of all local invocations of CORBA Objects. For this feature to be really useful, the user must create test suites which cover as much as possible. For example, invoking an operation with invalid or incorrect arguments should also be tested. This option can be activated for one object or all object via:

- *'MyModuyle_MyInterface':oe_create(Env, [{local_typecheck, true}])* - This approach will only activate, or deactivate, typechecking for the returned instance. Naturally, this option can also be passed to `oe_create_link/2`, `corba:create/4` and `corba:create_link/4`.
- *erl -orber flags 2* - can be added to a start-script. All object invocations will be typechecked, unless overridden by the previous option.
- *corba:orb_init([{flags, 16#0002}])* - this operation must be invoked *before* starting Orber. Behaves as the previous option.

If incorrect data is passed or returned, Orber uses the `error_logger` to generate logs, which can look like:

```
=ERROR REPORT==== 10-Jul-2002::12:36:09 ===
========= Orber Typecheck Request =========
Invoked......: MyModule_MyInterface:foo/1
Typecode.....: [{tk_enum,"IDL:MyModule/enumerant:1.0",
                         "enumerant",
                         ["one","two"]}]
Arguments....: [three]
Result.......: {'EXCEPTION',{'MARSHAL',[],102,'COMPLETED_NO'}}
===========================================
```

Note, that the arity is equivalent to the IDL-file. In the example above, an undefined enumerant was used. In most cases, it is useful to set the configuration parameter `orber_debug_level 10` as well. Due to the extra overhead, this option *MAY ONLY* be used during testing and development. For more information, see also configuration settings [page 12].

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 75]. Normally, the users must implement the interceptor themselves, but for your convenience Orber includes three pre-compiled interceptors called `orber_iiop_tracer`, `orber_iiop_tracer_silent` and `orber_iiop_tracer_stealth`.

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

The `orber_iiop_tracer` and `orber_iiop_tracer_silent` interceptors 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, if you use the `orber_iiop_tracer` interceptor, reports which looks like:

```
=INFO REPORT==== 13-Jul-2005::18:22:39 ===
=============== new_out_connection =======
Node      : myNode@myHost
From      : 192.0.0.10:47987
To        : 192.0.0.20:4001
==========================================

=INFO REPORT==== 29-Nov-2001::15:26:28 ===
=============== out_request ==============
Connection: {"192.0.0.20",4001,"192.0.0.10",47987}
Operation : resolve
Parameters: [[{'CosNaming_NameComponent',
                "AIK","SwedishIcehockeyChampions"}]]
Context   : [{'IOP_ServiceContext',1,
               {'CONV_FRAME_CodeSetContext',65537,65801}}]
==========================================
```

The `orber_iiop_tracer_silent` will not log GIOP encoded data. To activate one the interceptors, 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_silent]}}])* - this operation must be invoked *before* starting Orber.

It is also possible to active and deactivate an interceptor during run-time, but this will only affect currently existing connections. For more information, consult Orber's Reference Manual regarding the operations `orber:activate_audit_trail/0/1` and `orber:activate_audit_trail/0/1`.

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 12]). 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 12]).

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.

Q: I cannot run Orber as a multi-node ORB?

A: Make sure that the Erlang distribution have been started for each node and the `cookies` are correct. For more information, consult the `System Documentation`

Orber Reference Manual

Short Summaries

CosNaming

No functions are exported.

CosNaming_BindingIterator

The following functions are exported:

- `next_one(BindinIterator) -> Return`
  [page 114] Return a binding
- `next_n(BindinIterator, HowMany) -> Return`
  [page 114] Return a binding list
- `destroy(BindingIterator) -> Return`
  [page 114] Destroy the iterator object

CosNaming_NamingContext

The following functions are exported:

- `bind(NamingContext, Name, Object) -> Return`
  [page 117] Bind a Name to an Object
- `rebind(NamingContext, Name, Object) -> Return`
  [page 117] Bind an Object to the Name even if the Name already is bound
- `bind_context(NamingContext1, Name, NamingContex2) -> Return`
  [page 117] Bind a Name to an NamingContext
- `rebind_context(NamingContext1, Name, NamingContex2) -> Return`
  [page 117] Bind an NamingContext to the Name even if the Name already is bound
- `resolve(NamingContext, Name) -> Return`
  [page 117] Retrieve an Object bound to Name
- `unbind(NamingContext, Name) -> Return`
  [page 118] Remove the binding for a Name
- `new_context(NamingContext) -> Return`
  [page 118] Create a new NamingContext
- `bind_new_context(NamingContext, Name) -> Return`
  [page 118] Create a new NamingContext and bind it to a Name
- `destroy(NamingContext) -> Return`
  [page 118] Destroy a NamingContext
- `list(NamingContext, HowMany) -> Return`
  [page 118] List returns a all bindings in the context

CosNaming_NamingContextExt

The following functions are exported:

- `to_string(NamingContext, Name) -> Return`
  [page 120] Stringify a `Name`sequence to a string
- `to_name(NamingContext, NameString) -> Return`
  [page 120] Convert a stringified `Name`to a `Name`sequence
- `to_url(NamingContext, AddressString, NameString) -> Return`
  [page 120] Return an URL string constructed from the given Address and Name strings
- `resolve_str(NamingContext, NameString) -> Return`
  [page 120] Return the object associated, if any, with the given name string

Module_Interface

The following functions are exported:

- `Module_Interface:typeID() -> TypeId`
  [page 123] Return the Type ID related to this stub/skeleton
- `Module_Interface:oe_create() -> ObjRef`
  [page 123] Start a Orber server.
- `Module_Interface:oe_create_link() -> ObjRef`
  [page 123] Start a linked Orber server.
- `Module_Interface:oe_create(Env) -> ObjRef`
  [page 123] Start a Orber server.
- `Module_Interface:oe_create_link(Env) -> ObjRef`
  [page 123] Start a linked Orber server.
- `Module_Interface:oe_create(Env, Options) -> ObjRef`
  [page 123] Start a Orber stub/skeleton
- `Module_Interface:oe_create_link(Env, Options) -> Return`
  [page 124] Start a Orber stub/skeleton
- `Module_Interface:own_functions(ObjRef, Arg1, ..., ArgN) -> Reply`
  [page 125] User defined function which is not a part of Orber
- `Module_Interface:own_functions(ObjRef, Options, Arg1, ..., ArgN) -> Reply`
  [page 125] User defined function which is not a part of Orber
- `Module_Interface_impl:init(Env) -> CallReply`
  [page 126] User defined function which is not a part of Orber
- `Module_Interface_impl:terminate(Reason, State) -> ok`
  [page 126] User defined function which is not a part of Orber
- `Module_Interface_impl:code_change(OldVsn, State, Extra) -> CallReply`
  [page 126] User defined function which is not a part of Orber
- `Module_Interface_impl:handle_info(Info, State) -> CallReply`
  [page 126] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(State, Arg1, ..., ArgN) -> CallReply`
  [page 127] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(This, State, Arg1, ..., ArgN) -> CallReply`
  [page 127] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(This, From, State, Arg1, ..., ArgN) -> ExtCallReply`
  [page 127] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(From, State, Arg1, ..., ArgN) -> ExtCallReply`
  [page 127] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(State, Arg1, ..., ArgN) -> CastReply`
  [page 127] User defined function which is not a part of Orber
- `Module_Interface_impl:own_functions(This, State, Arg1, ..., ArgN) -> CastReply`
  [page 127] User defined function which is not a part of Orber

any

The following functions are exported:

- `create() -> Result`
  [page 128] Create an any record
- `create(Typecode, Value) -> Result`
  [page 128] Create an any record
- `set_typecode(A, Typecode) -> Result`
  [page 128] Set the typecode field
- `get_typecode(A) -> Result`
  [page 128] Fetch the typecode
- `set_value(A, Value) -> Result`
  [page 129] Set the value field
- `get_value(A) -> Result`
  [page 129] Fetch the value

corba

The following functions are exported:

- `create(Module, TypeID) -> Object`
  [page 130] Create and start a new server object
- `create(Module, TypeID, Env) -> Object`
  [page 130] Create and start a new server object
- `create(Module, TypeID, Env, Optons1) -> Object`
  [page 130] Create and start a new server object
- `create_link(Module, TypeID) -> Object`
  [page 130] Create and start a new server object
- `create_link(Module, TypeID, Env) -> Object`
  [page 130] Create and start a new server object
- `create_link(Module, TypeID, Env, Options2) -> Reply`
  [page 130] Create and start a new server object
- `dispose(Object) -> ok`
  [page 131] Stop a server object
- `create_nil_objref() -> Object`
  [page 131] Stop a server object
- `create_subobject_key(Object, Key) -> Result`
  [page 131] Add an Erlang term to a private key field
- `get_subobject_key(Object) -> Result`
  [page 132] Fetch the contents of the private key field
- `get_pid(Object) -> Result`
  [page 132] Get the process id from an object key
- `raise(Exception)`
  [page 132] Generate an Erlang throw
- `reply(To, Reply) -> true`
  [page 132] Send explicit reply to client

- `resolve_initial_references(ObjectId) -> Object`
  [page 132] Return the object reference for the given object id
- `resolve_initial_references(ObjectId, Contexts) -> Object`
  [page 132] Return the object reference for the given object id
- `add_initial_service(ObjectId, Object) -> boolean()`
  [page 133] Add a new initial service and associate it with the given id
- `remove_initial_service(ObjectId) -> boolean()`
  [page 133] Remove association between the given id and service
- `list_initial_services() -> [ObjectId]`
  [page 133] Return a list of supported object id's
- `resolve_initial_references_remote(ObjectId, Address) -> Object`
  [page 133] Return the object reference for the given object id
- `resolve_initial_references_remote(ObjectId, Address, Contexts) -> Object`
  [page 133] Return the object reference for the given object id
- `list_initial_services_remote(Address) -> [ObjectId]`
  [page 134] Return a list of supported object id's
- `list_initial_services_remote(Address, Contexts) -> [ObjectId]`
  [page 134] Return a list of supported object id's
- `object_to_string(Object) -> IOR_string`
  [page 134] Convert the object reference to the external string representation
- `string_to_object(IOR_string) -> Object`
  [page 135] Convert the external string representation to an object reference
- `string_to_object(IOR_string, Contexts) -> Object`
  [page 135] Convert the external string representation to an object reference
- `print_object(Data [, Type]) -> ok | {'EXCEPTION', E} | {'EXIT', R} | string()`
  [page 135] Print the supplied object
- `add_alternate_iiop_address(Object, Host, Port) -> NewObject | {'EXCEPTION', E}`
  [page 136] Add ALTERNATE_IIOP_ADDRESS component to the supplied local object
- `orb_init(KeyValueList) -> ok | {'EXIT', Reason}`
  [page 136] Configure Orber before starting it

corba_object

The following functions are exported:

- `get_interface(Object) -> InterfaceDef`
  [page 137] Fetch the interface description
- `is_nil(Object) -> boolean()`
  [page 137] Return true, if the given object is a NIL object reference, otherwise false
- `is_a(Object, Logical_type_id) -> Return`
  [page 137] Return true if the target object is an, or inherit from, object of the given type

- `is_a(Object, Logical_type_id, Contexts) -> Return`
  [page 137] Return true if the target object is an, or inherit from, object of the given type
- `is_remote(Object) -> boolean()`
  [page 138] Determine whether or not an object reference is remote
- `non_existent(Object) -> Return`
  [page 138] Return false if the target object do not exist, otherwise true
- `non_existent(Object, Contexts) -> Return`
  [page 138] Return false if the target object do not exist, otherwise true
- `not_existent(Object) -> Return`
  [page 138] Return false if the target object do not exist, otherwise true
- `not_existent(Object, Contexts) -> Return`
  [page 138] Return false if the target object do not exist, otherwise true
- `is_equivalent(Object, OtherObject) -> boolean()`
  [page 139] Return true if the target object and the supplied object easily can be determined to be equal, otherwise false
- `hash(Object, Maximum) -> int()`
  [page 139] Return a hash value based on the target object

fixed

The following functions are exported:

- `create(Digits, Scale, Value) -> Result`
  [page 141] Create a fixed type
- `get_typecode(Fixed) -> Result`
  [page 141] Create TypeCode representing the supplied fixed type
- `add(Fixed1, Fixed2) -> Result`
  [page 141] Add the supplied Fixed types
- `subtract(Fixed1, Fixed2) -> Result`
  [page 141] Subtract Fixed2 from Fixed1
- `multiply(Fixed1, Fixed2) -> Result`
  [page 141] Multiply Fixed1 with Fixed2
- `divide(Fixed1, Fixed2) -> Result`
  [page 141] Divide Fixed1 with Fixed2
- `unary_minus(Fixed) -> Result`
  [page 141] Negate the supplied Fixed Type

interceptors

The following functions are exported:

- `new_in_connection(Ref, PeerHost, PeerPort) -> NewRef`
  [page 143] Invoke when a new client ORB wants to setup a connection
- `new_in_connection(Ref, PeerHost, PeerPort, SocketHost, SocketPort) -> NewRef`
  [page 143] Invoke when a new client ORB wants to setup a connection

- `new_out_connection(Ref, PeerHost, PeerPort) -> NewRef`
  [page 143] Invoke when setting up a new connection to a server side ORB
- `new_out_connection(Ref, PeerHost, PeerPort, SocketHost, SocketPort) -> NewRef`
  [page 143] Invoke when setting up a new connection to a server side ORB
- `closed_in_connection(Ref) -> NewRef`
  [page 143] Invoke when an existing connection to a client side ORB have been terminated
- `closed_out_connection(Ref) -> NewRef`
  [page 143] Invoke when an existing connection to a server side ORB have been terminated
- `in_reply(Ref, Obj, Ctx, Op, Data, Extra) -> Reply`
  [page 144] Invoke when replies arrives at the client side ORB
- `in_reply_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`
  [page 144] Invoke when replies arrives at the client side ORB with undecoded reply body
- `in_request(Ref, Obj, Ctx, Op, Args, Extra) -> Reply`
  [page 144] Invoke when requests arrive at the server side ORB
- `in_request_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`
  [page 144] Invoke when requests arrive at the server side ORB with undecoded request body
- `out_reply(Ref, Obj, Ctx, Op, Data, Extra) -> Reply`
  [page 145] Invoke after the target object replied
- `out_reply_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`
  [page 145] Invoke after the target object replied with the reply encoded
- `out_request(Ref, Obj, Ctx, Op, Args, Extra) -> Reply`
  [page 145] Invoke on the client side ORB before encoding and sending the request
- `out_request_encoded(Ref, Obj, Ctx, Op, Bin, Extra) -> Reply`
  [page 145] Invoke on the client side ORB before sending the request

lname

The following functions are exported:

- `create() -> Return`
  [page 147] Create a new name
- `insert_component(Name, N, NameComponent) -> Return`
  [page 147] Insert a new name component in a name
- `get_component(Name, N) -> Return`
  [page 147] Get a name component from a name
- `delete_component(Name, N) -> Return`
  [page 148] Delete a name component from a name
- `num_components(Name) -> Return`
  [page 148] Count the number of name components in a name
- `equal(Name1, Name2) -> Return`
  [page 148] Test if two names are equal
- `less_than(Name1, Name2) -> Return`
  [page 148] Test if one name is lesser than the other

- `to_idl_form(Name) -> Return`
  [page 148] Transform a pseudo name to an IDL name
- `from_idl_form(Name) -> Return`
  [page 148] Transform an IDL name to a pseudo name

lname_component

The following functions are exported:

- `create() -> Return`
  [page 149] Create a new name component
- `get_id(NameComponent) -> Return`
  [page 149] Get the id field of a name component
- `set_id(NameComponent, Id) -> Return`
  [page 149] Set the id field of a name component
- `get_kind(NameComponent) -> Return`
  [page 149] Get the kind field of a name component
- `set_kind(NameComponent, Kind) -> Return`
  [page 150] Set the kind field of a name component

orber

The following functions are exported:

- `start() -> ok`
  [page 151] Start the Orber application
- `start(Type) -> ok`
  [page 151] Start the Orber application
- `jump_start(Attributes) -> ok | {'EXIT', Reason}`
  [page 151] Start the Orber application during tests
- `stop() -> ok`
  [page 151] Stop the Orber application
- `info() -> ok`
  [page 151] Generate Info Report, which contain Orber's configuration settings
- `info(IoType) -> ok | {'EXIT', Reason} | string()`
  [page 151] Generate Info Report, which contain Orber's configuration settings
- `exception_info(Exception) -> {ok, string()} | {error, Reason}`
  [page 152] Return a printable string, which describes the supplied exception
- `is_system_exception(Exception) -> true | false`
  [page 152] Return true if the supplied exception is a system defined exception otherwise false
- `get_tables() -> [Tables]`
  [page 152] Get the Mnesia tables Orber uses.
- `get_ORBInitRef() -> string() | undefined`
  [page 152] Get the initial reference address.
- `get_ORBDefaultInitRef() -> string() | undefined`
  [page 152] Get the initial reference address.

- `domain() -> string()`
  [page 152] Display the Orber domain name
- `iiop_port() -> int()`
  [page 152] Display the IIOP port number
- `iiop_out_ports() -> 0 | {Min, Max}`
  [page 152] Display the ports Orber may use when connecting to another ORB
- `iiop_ssl_port() -> int()`
  [page 152] Display the IIOP port number used for secure connections
- `iiop_timeout() -> int() (milliseconds)`
  [page 152] Display the IIOP timeout value
- `iiop_connection_timeout() -> int() (milliseconds)`
  [page 153] Display the IIOP connection timeout value
- `iiop_connections() -> Result`
  [page 153] List all existing connections to/from other ORB's
- `iiop_connections(Direction) -> Result`
  [page 153] List all existing connections to/from other ORB's
- `iiop_connections_pending() -> Result`
  [page 153] List all connections to another ORB currently being set up
- `iiop_in_connection_timeout() -> int() (milliseconds)`
  [page 154] Display the IIOP connection timeout value for incoming connections
- `iiop_acl() -> Result`
  [page 154] Return the ACL configuration
- `activate_audit_trail() -> Result`
  [page 154] Activate IIOP audit/trail
- `activate_audit_trail(Verbosity) -> Result`
  [page 154] Activate IIOP audit/trail
- `deactivate_audit_trail() -> Result`
  [page 154] Deactivate IIOP audit/trail
- `add_listen_interface(IP, Type) -> Result`
  [page 155] Add a new listen process for incoming connection
- `add_listen_interface(IP, Type, Port) -> Result`
  [page 155] Add a new listen process for incoming connection
- `add_listen_interface(IP, Type, ConfigurationParameters) -> Result`
  [page 155] Add a new listen process for incoming connection
- `remove_listen_interface(Ref) -> ok`
  [page 155] Terminate listen process for incoming connection
- `close_connection(Connection) -> Result`
  [page 156] Terminate outgoing connection(s)
- `close_connection(Connection, Interface) -> Result`
  [page 156] Terminate outgoing connection(s)
- `secure() -> no | ssl`
  [page 156] Display the security mode Orber is running in
- `ssl_server_certfile() -> string()`
  [page 156] Display the path to the server certificate
- `ssl_client_certfile() -> string()`
  [page 156] Display the path to the client certificate

- `set_ssl_client_certfile(Path) -> ok`
  [page 156] Set the value of the client certificate
- `ssl_server_verify() -> 0 | 1 | 2`
  [page 157] Display the SSL verification type for incoming calls
- `ssl_client_verify() -> 0 | 1 | 2`
  [page 157] Display the SSL verification type for outgoing calls
- `set_ssl_client_verify(Value) -> ok`
  [page 157] Set the value of the SSL verification type for outgoing calls
- `ssl_server_depth() -> int()`
  [page 157] Display the SSL verification depth for incoming calls
- `ssl_client_depth() -> int()`
  [page 157] Display the SSL verification depth for outgoing calls
- `set_ssl_client_depth(Depth) -> ok`
  [page 157] Sets the value of the SSL verification depth for outgoing calls
- `objectkeys_gc_time() -> int() (seconds)`
  [page 157] Display the Object Keys GC time value
- `orber_nodes() -> RetVal`
  [page 157] Displays which nodes that this orber domain consist of.
- `install(NodeList) -> ok`
  [page 158] Install the Orber application
- `install(NodeList, Options) -> ok`
  [page 158] Install the Orber application
- `uninstall() -> ok`
  [page 158] Uninstall the Orber application
- `add_node(Node, Options) -> RetVal`
  [page 159] Add a new node to a group of Orber nodes.
- `remove_node(Node) -> RetVal`
  [page 159] Removes a node from a group of Orber nodes.
- `configure(Key, Value) -> ok | {'EXIT', Reason}`
  [page 159] Change Orber configuration.

orber_acl

The following functions are exported:

- `match(IP, Direction) -> boolean()`
  [page 161] Verify if the IP address versus the current configuration
- `match(IP, Direction, GetInfo) -> Reply`
  [page 161] Verify if the IP address versus the current configuration
- `verify(IP, Filter, Family) -> Reply`
  [page 161] Verify if the IP address versus the Filter
- `range(Filter, Family) -> Reply`
  [page 162] Get range of Filter

orber_diagnostics

The following functions are exported:

- `nameservice() -> Result`
  [page 163] Display all objects stored in the Name Service
- `nameservice(Flags) -> Result`
  [page 163] Display all objects stored in the Name Service
- `missing_modules() -> Count`
  [page 163] Echo missing modules required by Orber

orber_ifr

The following functions are exported:

- `init(Nodes,Timeout) -> ok`
  [page 164] Intialize the IFR
- `find_repository() -> #IFR_Repository_objref`
  [page 164] Find the IFR object reference for the Repository
- `get_def_kind(Objref) -> Return`
  [page 165] Return the definition kind of the IFR object
- `destroy(Objref) -> Return`
  [page 165] Destroy, except IRObject, Contained and Container, target object and its contents
- `get_id(Objref) -> Return`
  [page 165] Return the target object's repository id
- `set_id(Objref,Id) -> ok`
  [page 165] Set the target object's repository id
- `get_name(Objref) -> Return`
  [page 165] Return the name of the target object
- `set_name(Objref,Name) -> ok`
  [page 165] Set given name to target object
- `get_version(Objref) -> Return`
  [page 166] Return the version of the target object
- `set_version(Objref,Version) -> ok`
  [page 166] Set given version of the target object
- `get_defined_in(Objref) -> Return`
  [page 166] Return the Container the target object is contained in
- `get_absolute_name(Objref) -> Return`
  [page 166] Return the absolute name of the target object
- `get_containing_repository(Objref) -> Return`
  [page 166] Get the most derived Contained object associated with the target object
- `describe(Objref) -> Return`
  [page 166] Return a tuple which describe the target object
- `move(Objref,New_container,New_name,New_version) -> Return`
  [page 167] Move the target object from its current location to given Container, name and version

- `lookup(Objref,Search_name) -> Return`
  [page 167] Return the IFR object identified by the given name
- `contents(Objref,Limit_type,Exclude_inherited) -> Return`
  [page 167] 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 167] Return a list of IFR objects matching the given name
- `describe_contents(Objref,Limit_type,Exclude_inherited,Max_returned_objs) -> Return`
  [page 168] Return a list of descriptions of the IFR objects contained by the target Container object
- `create_module(Objref,Id,Name,Version) -> Return`
  [page 168] Create an IFR object of given type
- `create_constant(Objref,Id,Name,Version,Type,Value) -> Return`
  [page 168] Create a ConstantDef IFR object
- `create_struct(Objref,Id,Name,Version,Members) -> Return`
  [page 168] Create a StructDef IFR object
- `create_union(Objref,Id,Name,Version,Discriminator_type,Members) -> Return`
  [page 169] Create a UnionDef IFR object
- `create_enum(Objref,Id,Name,Version,Members) -> Return`
  [page 169] Create a EnumDef IFR object
- `create_alias(Objref,Id,Name,Version,Original_type) -> Return`
  [page 169] Create a AliasDef IFR object
- `create_interface(Objref,Id,Name,Version,Base_interfaces) -> Return`
  [page 169] Create a InterfaceDef IFR object
- `create_exception(Objref,Id,Name,Version,Members) -> Return`
  [page 170] Create a ExceptionDef IFR object
- `get_type(Objref) -> Return`
  [page 170] Return the typecode of the target object
- `lookup_id(Objref,Search_id) -> Return`
  [page 170] Return the IFR object matching the given id
- `get_primitive(Objref,Kind) -> Return`
  [page 170] Return a PrimitiveDef of the specified kind
- `create_string(Objref,Bound) -> Return`
  [page 170] Create an IFR objref of the type StringDef
- `create_wstring(Objref,Bound) -> Return`
  [page 171] Create an IFR objref of the type WstringDef
- `create_fixed(Objref,Digits,Scale) -> Return`
  [page 171] Create an IFR objref of the type FixedDef
- `create_sequence(Objref,Bound,Element_type) -> Return`
  [page 171] Create an IFR objref of the type SequenceDef
- `create_array(Objref,Length,Element_type) -> Return`
  [page 171] Create an IFR objref of the type ArrayDef
- `create_idltype(Objref,Typecode) -> Return`
  [page 171] Create an IFR objref of the type IDLType

- `get_type_def(Objref) -> Return`
  [page 171] Return an IFR object of the type IDLType describing the type of the target object
- `set_type_def(Objref,TypeDef) -> Return`
  [page 172] Set given TypeDef of the target object
- `get_value(Objref) -> Return`
  [page 172] Return the value attribute of the target ConstantDef object
- `set_value(Objref,Value) -> Return`
  [page 172] Set the value attribute of the target ConstantDef object
- `get_members(Objref) -> Return`
  [page 172] Return the members of the target object
- `set_members(Objref,Members) -> Return`
  [page 172] Set the members attribute of the target object
- `get_discriminator_type(Objref) -> Return`
  [page 173] Get the discriminator typecode of the target object
- `get_discriminator_type_def(Objref) -> Return`
  [page 173] Return IDLType object describing the discriminator type of the target object
- `set_discriminator_type_def(Objref,TypeDef) -> Return`
  [page 173] Set the attribute discriminator_type_def for the target object to the given TypeDef
- `get_original_type_def(Objref) -> Return`
  [page 173] Return an IFR object of the type IDLType describing the original type
- `set_original_type_def(Objref,TypeDef) -> Return`
  [page 173] Set the original_type_def attribute which describes the original type
- `get_kind(Objref) -> Return`
  [page 173] Return an atom describing the primitive type
- `get_bound(Objref) -> Return`
  [page 174] Get the maximum size of the target object
- `set_bound(Objref,Bound) -> Return`
  [page 174] Set the maximum size of the target object
- `get_element_type(Objref) -> Return`
  [page 174] Return the typecode of the elements in the IFR object
- `get_element_type_def(Objref) -> Return`
  [page 174] Return an IFR object of the type IDLType describing the type of the elements in Objref
- `set_element_type_def(Objref,TypeDef) -> Return`
  [page 174] Set the element_type_def attribute of the target object to the given TypeDef
- `get_length(Objref) -> Return`
  [page 174] Return the number of elements in the array
- `set_length(Objref,Length) -> Return`
  [page 175] Set the number of elements in the array
- `get_mode(Objref) -> Return`
  [page 175] Get the mode of the target object (AttributeDef or OperationDef)
- `set_mode(Objref,Mode) -> Return`
  [page 175] Set the mode of the target object (AttributeDef or OperationDef) to the given mode

- `get_result(Objref) -> Return`
  [page 175] Return typecode describing the type of the value returned by the operation
- `get_result_def(Objref) -> Return`
  [page 175] Return an IFR object of the type IDLType describing the type of the result
- `set_result_def(Objref,ResultDef) -> Return`
  [page 175] Set the type_def attribute of the target object to the given ResultDef
- `get_params(Objref) -> Return`
  [page 176] Return a list of parameter description records describing the parameters of the target OperationDef
- `set_params(Objref,Params) -> Return`
  [page 176] Set the params attibute of the target object to the given parameter description records
- `get_contexts(Objref) -> Return`
  [page 176] Return a list of context identifiers for the operation
- `set_contexts(Objref,Contexts) -> Return`
  [page 176] Set the context attribute for the operation
- `get_exceptions(Objref) -> Return`
  [page 176] Return a list of exception types that can be raised by the target object
- `set_exceptions(Objref,Exceptions) -> Return`
  [page 176] Set the exceptions attribute for the target object
- `get_base_interfaces(Objref) -> Return`
  [page 177] Return a list of InterfaceDefs from which the target InterfaceDef object inherit
- `set_base_interfaces(Objref,BaseInterfaces) -> Return`
  [page 177] Set the BaseInterfaces attribute
- `is_a(Objref,Interface_id) -> Return`
  [page 177] Return a boolean if the target InterfaceDef match or inherit from the given id
- `describe_interface(Objref) -> Return`
  [page 177] Return a full inter face description record describing the InterfaceDef
- `create_attribute(Objref,Id,Name,Version,Type,Mode) -> Return`
  [page 177] 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 177] Create an IFR object of the type OperationDef contained in the target InterfaceDef object

orber_tc

The following functions are exported:

- `null() -> TC`
  [page 179] Return the IDL typecode
- `void() -> TC`
  [page 179] Return the IDL typecode

- `short() -> TC`
  [page 179] Return the IDL typecode
- `unsigned_short() -> TC`
  [page 179] Return the IDL typecode
- `long() -> TC`
  [page 179] Return the IDL typecode
- `unsigned_long() -> TC`
  [page 179] Return the IDL typecode
- `long_long() -> TC`
  [page 179] Return the IDL typecode
- `unsigned_long_long() -> TC`
  [page 179] Return the IDL typecode
- `wchar() -> TC`
  [page 179] Return the IDL typecode
- `float() -> TC`
  [page 179] Return the IDL typecode
- `double() -> TC`
  [page 179] Return the IDL typecode
- `boolean() -> TC`
  [page 179] Return the IDL typecode
- `char() -> TC`
  [page 179] Return the IDL typecode
- `octet() -> TC`
  [page 179] Return the IDL typecode
- `any() -> TC`
  [page 179] Return the IDL typecode
- `typecode() -> TC`
  [page 179] Return the IDL typecode
- `principal() -> TC`
  [page 179] Return the IDL typecode
- `object_reference(Id, Name) -> TC`
  [page 179] Return the object_reference IDL typecode
- `struct(Id, Name, ElementList) -> TC`
  [page 179] Return the struct IDL typecode
- `union(Id, Name, DiscrTC, Default, ElementList) -> TC`
  [page 180] Return the union IDL typecode
- `enum(Id, Name, ElementList) -> TC`
  [page 180] Return the enum IDL typecode
- `string(Length) -> TC`
  [page 180] Return the string IDL typecode
- `wstring(Length) -> TC`
  [page 181] Return the wstring IDL typecode
- `fixed(Digits, Scale) -> TC`
  [page 181] Return the fixed IDL typecode
- `sequence(ElemTC, Length) -> TC`
  [page 181] Return the sequence IDL typecode

- `array(ElemTC, Length) -> TC`
  [page 181] Return the array IDL typecode
- `alias(Id, Name, AliasTC) -> TC`
  [page 181] Return the alias IDL typecode
- `exception(Id, Name, ElementList) -> TC`
  [page 181] Return the exception IDL typecode
- `get_tc(Object) -> TC`
  [page 182] Fetch typecode
- `get_tc(Id) -> TC`
  [page 182] Fetch typecode
- `check_tc(TC) -> boolean()`
  [page 182] 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 the 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 and a boolean. The latter is set to true if the binding is valid otherwise false. If the boolean is false there are no more bindings to retrieve.

`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, NamingContex2) -> 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, NamingContex2) -> 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 up-to HowMany from the context. It also returns a BindinIterator which can be used to step through the list. If the total number of existing bindings are less than, or equal to, the `HowMany` parameter a NIL object reference is returned.

Note:
One must destroy the BindingIterator, unless it is a NIL object reference, by using '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 116]

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 IC option from has no effect.
- 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

`Module_Interface:typeID() -> TypeId`

Types:

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

Returns the Type ID related to this stub/skeleton

`Module_Interface:oe_create() -> ObjRef`

Types:

- ObjRef = #object reference

Start a Orber server.

`Module_Interface:oe_create_link() -> ObjRef`

Types:

- ObjRef = #object reference

Start a linked Orber server.

`Module_Interface:oe_create(Env) -> ObjRef`

Types:

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

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

`Module_Interface:oe_create_link(Env) -> ObjRef`

Types:

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

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

`Module_Interface:oe_create(Env, Options) -> ObjRef`

Types:

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

- Options = [{sup_child, false} | {persistent, Bool} | {regname, RegName} | {pseudo, Bool} | {local_typecheck, Bool} | {survive_exit, Bool} | {create_options, [CreateOpts]}]
- Bool = true | false
- RegName = {global, term()} | {local, atom()}
- CreateOpts = {debug, [Dbg]} | {timeout, Time}
- Dbg = trace | log | statistics | {log_to_file, FileName}

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*.

The option `{local_typecheck, boolean()}`, which overrides the Local Typechecking [page 17] environment flag, turns on or off typechecking. If activated, parameters, replies and raised exceptions will be checked to ensure that the data is correct, when invoking operations on CORBA Objects within the same Orber domain. Due to the extra overhead, this option *MAY ONLY* be used during testing and development.

`{survive_exit, boolean()}` overrides the EXIT Tolerance [page 17] environment flag. If activated, the server will not terminate, even though the call-back module returns EXIT.

`Time` specifies how long time, in milliseconds, the server is allowed to spend initializing. For more information about the `Dbg` options, see the `sys` module.

`Module_Interface: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} | {local_typecheck, Bool} | {survive_exit, Bool} | {create_options, [CreateOpts]}]
- Bool = true | false
- RegName = {global, term()} | {local, atom()}
- CreateOpts = {debug, [Dbg]} | {timeout, Time}
- Dbg = trace | log | statistics | {log_to_file, FileName}
-
-
-

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).

The options {`local_typecheck, boolean()`} and {`survive_exit, boolean()`} behaves in the same way as for `oe_create/2`.

`Time` specifies how long time, in milliseconds, the server is allowed to spend initializing. For more information about the `Dbg` options, see the `sys` module.

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

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

Types:

- ObjRef = #object reference
- Options = [Option] | Timeout
- Option = {timeout, Timeout} | {context, [Context]}
- Timeout = infinity | integer(milliseconds)
- Context = #'IOP_ServiceContext'{context_id = CtxId, context_data = CtxData}
- CtxId = ?ORBER_GENERIC_CTX_ID
- CtxData = {interface, Interface} | {userspecific, term()} | {configuration, Options}
- Interface = string()
- Options = [{Key, Value}]
- Key = ssl_client_verify | ssl_client_depth | ssl_client_certfile | ssl_client_cacertfile | ssl_client_password | ssl_client_keyfile | ssl_client_ciphers | ssl_client_cachetimeout
- Value = allowed value associated with the given key
- ArgX = specified in the IDL-code.
- Reply = specified in the IDL-code.

The default value for the `Timeout` option is `infinity`. IPv4 or IPv6 addresses are accepted as local Interface.

The *configuration* context is used to override the global SSL client side configuration [page 12].

To gain access to `#'IOP_ServiceContext'`{} record and the `?ORBER_GENERIC_CTX_ID` macro, you must add `-include_lib("orber/include/corba.hrl").` to your module.

CALLBACK FUNCTIONS

The following functions should be exported from a `CORBA` callback module. Note, a complete template of the call-back module can be generated automatically by compiling the IDL-file with the IC option {`be,erl_template`}. One should also add the same compile options, for example `this` or `from`, used when generating the stub/skeleton modules.

Exports

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

Types:

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

All two-way functions must return one of the listed replies or raise any of the exceptions listed in the IDL code (i.e. raises(...)). If the IC compile options *this* and/or *from* are used, the implementation must accept the *This* and/or *From* parameters.

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

Types:

- This = the servers #object reference
- State = term()
- CastReply = {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()

All one-way functions must return one of the listed replies. If the IC compile option *this* is used, the implementation must accept the *This* parameter.

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, Optons1) -> 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} | {local_typecheck, Bool}]
- Options2 = [{sup_child, Bool} | {persistent, Bool} | {regname, RegName} | {pseudo, Bool} | {local_typecheck, 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_link2, 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 children they should be *transient* and the *objectkeys_gc_time* should be modified (default equals `infinity`).

The option {`local_typecheck, boolean()`}, which overrides the Local Typechecking [page 17] environment flag, turns on or off typechecking. If activated, parameters, replies and raised exceptions will be checked to ensure that the data is correct, when invoking operations on CORBA Objects within the same Orber domain. Due to the extra overhead, this option *MAY ONLY* be used during testing and development.

`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 a always a binary, if it was an Erlang term that was stored with *create_subobject_key/2* one can to 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*}.

`reply(To, Reply) -> true`

Types:

- To = client reference
- Reply = IDL type

This function can be used by a CORBA object to explicitly send a reply to a client that invoked a two-way operation. If this operation is used, it is *not* possible to return a reply in the call-back module.
To must be the *From* argument provided to the callback function, which requires that the IC option *from* was used when compiling the IDL-file.

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

Types:

- ObjectId = string()
- Contexts = [Context]

- Context = #'IOP_ServiceContext'{context_id = CtxId, context_data = CtxData}
- CtxId = ?ORBER_GENERIC_CTX_ID
- CtxData = {interface, Interface} | {userspecific, term()} | {configuration, Options}
- Interface = string()
- Options = [{Key, Value}]
- Key = ssl_client_verify | ssl_client_depth | ssl_client_certfile | ssl_client_cacertfile | ssl_client_password | ssl_client_keyfile | ssl_client_ciphers | ssl_client_cachetimeout
- Value = allowed value associated with the given key
- 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`.

The *configuration* context is used to override the global SSL client side configuration [page 12].

`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 49]. 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`
`resolve_initial_references_remote(ObjectId, Address, Contexts) -> Object`

Types:

- ObjectId = string()
- Address = [RemoteModifier]
- RemoteModifier = string()
- Contexts = [Context]
- Context = #'IOP_ServiceContext'{context_id = CtxId, context_data = CtxData}
- CtxId = ?ORBER_GENERIC_CTX_ID

- CtxData = {interface, Interface} | {userspecific, term()} | {configuration, Options}
- Interface = string()
- Options = [{Key, Value}]
- Key = ssl_client_verify | ssl_client_depth | ssl_client_certfile | ssl_client_cacertfile | ssl_client_password | ssl_client_keyfile | ssl_client_ciphers | ssl_client_cachetimeout
- Value = allowed value associated with the given key
- 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"`.

The *configuration* context is used to override the global SSL client side configuration [page 12].

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

`list_initial_services_remote(Address) -> [ObjectId]`

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

Types:

- Address = [RemoteModifier]
- RemoteModifier = string()
- Contexts = [Context]
- Context = #'IOP_ServiceContext'{context_id = CtxId, context_data = CtxData}
- CtxId = ?ORBER_GENERIC_CTX_ID
- CtxData = {interface, Interface} | {userspecific, term()} | {configuration, Options}
- Interface = string()
- Options = [{Key, Value}]
- Key = ssl_client_verify | ssl_client_depth | ssl_client_certfile | ssl_client_cacertfile | ssl_client_password | ssl_client_keyfile | ssl_client_ciphers | ssl_client_cachetimeout
- Value = allowed value associated with the given key
- ObjectId = string()

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

The *configuration* context is used to override the global SSL client side configuration [page 12].

> **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`
`string_to_object(IOR_string, Contexts) -> Object`

Types:

- IOR_string = string()
- Contexts = [Context]
- Context = #'IOP_ServiceContext'{context_id = CtxId, context_data = CtxData}
- CtxId = ?ORBER_GENERIC_CTX_ID
- CtxData = {interface, Interface} | {userspecific, term()} | {configuration, Options}
- Interface = string()
- Options = [{Key, Value}]
- Key = ssl_client_verify | ssl_client_depth | ssl_client_certfile | ssl_client_cacertfile | ssl_client_password | ssl_client_keyfile | ssl_client_ciphers | ssl_client_cachetimeout
- Value = allowed value associated with the given key
- 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@123.0.0.012:4001/NameService"`

We can also resolve an object from the NameService by using
`"corbaname:iiop:1.2@123.0.0.012:4001/NameService#org/Erlang/MyObj"`

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

The *configuration* context is used to override the global SSL client side configuration [page 12].

How to handle the interface context is further described in the User's Guide.

`print_object(Data [, Type]) -> ok | {'EXCEPTION', E} | {'EXIT', R} | string()`

Types:

- Data = IOR_string | #objref (local or external) | corbaloc/corbaname string
- Type = IoDevice | error_report | {error_report, Reason} | info_msg | {info_msg, Comment} | string
- IoDevice = see the io-module
- Reason = Comment = string()

The object represented by the supplied data is dissected and presented in a more readable form. The Type parameter is optional; if not supplied standard output is used. For `error_report` and `info_msg` the `error_logger` module is used, with or without Reason or Comment. If the atom `string` is supplied this function will return a flat list. The `IoDevice` is passed to the operation `io:format/2`.

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.

Note:
Make sure that the given `Object` is accessible via the alternate Host/port. For example, if the object is correctly started as `local` or `pseudo`, the object should be available on all nodes within a multi-node Orber installation. Since only one instance exists for other object types, it will not be possible to access it if the node it was started on terminates.

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

Types:

- KeyValueList = [{Key, Value}]
- Key = any key listed in the configuration chapter
- 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 12] 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`
`is_a(Object, Logical_type_id, Contexts) -> Return`

Types:

- Object = #objref
- Logical_type_id = string()
- Contexts = [Context]
- Context = #'IOP_ServiceContext'{context_id = CtxId, context_data = CtxData}
- CtxId = ?ORBER_GENERIC_CTX_ID
- CtxData = {interface, Interface} | {userspecific, term()} | {configuration, Options}
- Interface = string()
- Options = [{Key, Value}]
- Key = ssl_client_verify | ssl_client_depth | ssl_client_certfile | ssl_client_cacertfile | ssl_client_password | ssl_client_keyfile | ssl_client_ciphers | ssl_client_cachetimeout
- Value = allowed value associated with the given key
- Return = boolean() | {'EXCEPTION', E}

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.

The *configuration* context is used to override the global SSL client side configuration [page 12].

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`
`non_existent(Object, Contexts) -> Return`

Types:

- Object = #objref
- Contexts = [Context]
- Context = #'IOP_ServiceContext'{context_id = CtxId, context_data = CtxData}
- CtxId = ?ORBER_GENERIC_CTX_ID
- CtxData = {interface, Interface} | {userspecific, term()} | {configuration, Options}
- Interface = string()
- Options = [{Key, Value}]
- Key = ssl_client_verify | ssl_client_depth | ssl_client_certfile | ssl_client_cacertfile | ssl_client_password | ssl_client_keyfile | ssl_client_ciphers | ssl_client_cachetimeout
- Value = allowed value associated with the given key
- Return = boolean() | {'EXCEPTION', E}

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.

The *configuration* context is used to override the global SSL client side configuration [page 12].

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`
`not_existent(Object, Contexts) -> Return`

Types:

- Object = #objref
- Contexts = [Context]
- Context = #'IOP_ServiceContext'{context_id = CtxId, context_data = CtxData}

- CtxId = ?ORBER_GENERIC_CTX_ID
- CtxData = {interface, Interface} | {userspecific, term()} | {configuration, Options}
- Interface = string()
- Options = [{Key, Value}]
- Key = ssl_client_verify | ssl_client_depth | ssl_client_certfile | ssl_client_cacertfile | ssl_client_password | ssl_client_keyfile | ssl_client_ciphers | ssl_client_cachetimeout
- Value = allowed value associated with the given key
- Return = boolean() | {'EXCEPTION', E}

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.

The *configuration* context is used to override the global SSL client side configuration [page 12].

`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.

fixed

Erlang Module

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

The type `Fixed` used below is defined as:

```
-record(fixed, {digits, scale, value}).
```

where `digits` is the total amount of digits it consists of and `scale` is the number of fractional digits. The `value` field contains the actual Fixed value represented as an integer. The limitations of each field are:

- Digits - integer(), -1 > Digits < 32
- Scale - integer(), -1 > Scale =< Digits
- Value - integer(), range (31 digits): 9999999999999999999999999999999

Since the Value part is represented by an integer, it is vital that the Digits and Scale values are correct. This also means that trailing zeros cannot be left out in some cases:

- fixed<5,3> eq. 03.140d eq. 3140
- fixed<3,2> eq. 3.14d eq. 314

Leading zeros can be left out.

For your convenience, this module exports functions which handle unary (`-`) and binary (`+-*/`) operations legal for the Fixed type. Since a unary `+` have no effect, this module do not export such a function. Any of the binary operations may cause an overflow (i.e. more than 31 significant digits; leading and trailing zeros are not considered significant). If this is the case, the Digit and Scale values are adjusted and the Value truncated (no rounding performed). This behavior is compliant with the OMG CORBA specification. Each binary operation have the following upper bounds:

- *Fixed1 + Fixed2* - `fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>`
- *Fixed1 - Fixed2* - `fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>`
- *Fixed1 * Fixed2* - `fixed<d1+d2, s1+s2>`
- *Fixed1 / Fixed2* - `fixed<(d1-s1+s2) + Sinf ,Sinf >`

A quotient may have an arbitrary number of decimal places, which is denoted by a scale of Sinf.

Exports

`create(Digits, Scale, Value) -> Result`

Types:

- Result = Fixed Type | {'EXCEPTION', #'BAD_PARAM'{}}

This function creates a new instance of a `Fixed Type`. If the limitations is not fulfilled (e.g. overflow) an exception is raised.

`get_typecode(Fixed) -> Result`

Types:

- Result = TypeCode | {'EXCEPTION', #'BAD_PARAM'{}}

Returns the TypeCode which represents the supplied Fixed type. If the parameter is not of the correct type, an exception is raised.

`add(Fixed1, Fixed2) -> Result`

Types:

- Result = Fixed1 + Fixed2 | {'EXCEPTION', #'BAD_PARAM'{}}

Performs a Fixed type addition. If the parameters are not of the correct type, an exception is raised.

`subtract(Fixed1, Fixed2) -> Result`

Types:

- Result = Fixed1 - Fixed2 | {'EXCEPTION', #'BAD_PARAM'{}}

Performs a Fixed type subtraction. If the parameters are not of the correct type, an exception is raised.

`multiply(Fixed1, Fixed2) -> Result`

Types:

- Result = Fixed1 * Fixed2 | {'EXCEPTION', #'BAD_PARAM'{}}

Performs a Fixed type multiplication. If the parameters are not of the correct type, an exception is raised.

`divide(Fixed1, Fixed2) -> Result`

Types:

- Result = Fixed1 / Fixed2 | {'EXCEPTION', #'BAD_PARAM'{}}

Performs a Fixed type division. If the parameters are not of the correct type, an exception is raised.

`unary_minus(Fixed) -> Result`

Types:

- Result = -Fixed | {'EXCEPTION', #'BAD_PARAM'{}}

Negates the supplied Fixed type. If the parameter is not of the correct type, an exception is raised.

interceptors

Erlang Module

This module contains the mandatory functions for user supplied native interceptors and their intended behavior. 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, PeerHost, PeerPort) -> NewRef`
`new_in_connection(Ref, PeerHost, PeerPort, SocketHost, SocketPort) -> NewRef`

Types:

- Ref = term() | undefined
- PeerHost = SocketHost = string(), e.g., ”myHost@myServer” or ”192.0.0.10”
- PeerPort = SocketPort = 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 PeerHost and PeerPort variables supplied data of the client ORB which requested a new connection. SocketHost and SocketPort are the local interface and port the client connected to.

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

`new_out_connection(Ref, PeerHost, PeerPort) -> NewRef`
`new_out_connection(Ref, PeerHost, PeerPort, SocketHost, SocketPort) -> NewRef`

Types:

- Ref = term() | undefined
- PeerHost = SocketHost = string(), e.g., ”myHost@myServer” or ”192.0.0.10”
- PeerPort = SocketPort = 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 PeerHost and PeerPort variables identifies the target ORB’s bootstrap data and SocketHost and SocketPort are the local interface and port the client ORB connected via.

`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, ..., OutPramaterN]
- 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 value from the target object and the rest is a 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, ..., OutPramaterN]
- 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 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.

lname

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 lname interface supports handling of names e.g. adding and removing name components.

Note that the lname 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`
`start(Type) -> ok`

Types:

- Type = temporary | permanent

Starts the Orber application (it also starts mnesia if it is not running). Which `Type` parameter is supplied determines the behavior. If not supplied Orber is started as `temporary`. See the Reference Manual *application(3)* for further information.

`jump_start(Attributes) -> ok | {'EXIT', Reason}`

Types:

- Attributes = Port | Options
- Port = integer()
- Options = [{Key, Value}]
- Key = any key listed in the configuration chapter
- Value = allowed value associated with the given key

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. Theses 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.

`info() -> ok`
`info(IoType) -> ok | {'EXIT', Reason} | string()`

Types:

- IoType = info_msg | string | io | {io, IoDevice}

Generates an Info Report, which contain Orber's configuration settings. If no `IoType` is supplied, `info_msg` is used (see the error_logger documentation). When the atom string is supplied this function will return a flat list. For `io` and `{io, IoDevice}`, `io:format/1` and `io:format/3` is used respectively.

`exception_info(Exception) -> {ok, string()} | {error, Reason}`

Returns a printable string, which describes the supplied exception in greater detail. Note, this function is mainly intended for system exceptions.

`is_system_exception(Exception) -> true | false`

Returns true if the supplied exception is a system defined exception, otherwise false.

`get_tables() -> [Tables]`

Returns a list of the Orber specific Mnesia tables. This list is required to restore Mnesia if it has been partitioned.

`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 12] 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 to 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 system exception, e.g. *TIMEOUT*, is raised.

Note: the iiop_timeout configuration parameter (TimeVal) may only range between 0 and 1000000 seconds. Otherwise, the default value is used.

Note: Earlier IC versions required that the compile option `{timeout,"module::interface"}`, was used, which allow the user to add an extra timeout parameter, e.g., `module_interface:function(ObjRef, Timeout, ... Arguments ...)` or `module_interface:function(ObjRef, [{timeout, Timeout}], ... Arguments ...)`, instead of `module_interface:function(ObjRef, ... Arguments ...)`. This is no longer the case and if the extra Timeout is used, 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() -> Result`

`iiop_connections(Direction) -> Result`

Types:

- Direction = in | out | inout
- Result = [{Host, Port}] | [{Host, Port, Interface}] | {'EXIT',Reason}
- Host = string()
- Port = integer()
- Interface = string()
- Reason = term()

The list returned by this operation contain tuples of remote hosts/ports Orber is currently connected to. If no Direction is not supplied, both incoming and outgoing connections are included.

If a specific local interface has been defined for the connection, this will be added to the returned tuple.

`iiop_connections_pending() -> Result`

Types:

- Result = [{Host, Port}] | [{Host, Port, Interface}] | {'EXIT',Reason}
- Host = string()
- Port = integer()
- Interface = string()
- Reason = term()

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.

If a specific local interface has been defined for the connection, this will be added to the returned tuple.

`iiop_in_connection_timeout() -> int() (milliseconds)`

This function returns the timeout value after which incoming IIOP connections terminate. It can be configured by setting the application variable *iiop_in_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_in_connection_timeout configuration parameter (TimeVal) may only range between 0 and 1000000 seconds. Otherwise, the default value is used.

`iiop_acl() -> Result`

Types:

- Result = [{Direction, Filter}] | [{Direction, Filter, [Interface]}]
- Direction = tcp_in | ssl_in | tcp_out | ssl_out
- Filter = string()
- Interface = string()

Returns the ACL configuration. The `Filter` uses a extended format of Classless Inter Domain Routing (CIDR). For example, `"123.123.123.10"` limits the connection to that particular host, while `"123.123.123.10/17"` allows connections to or from any host equal to the 17 most significant bits. Orber also allow the user to specify a certain port or port range, for example, `"123.123.123.10/17#4001"` and `"123.123.123.10/17#4001/5001"` respectively. IPv4 or none compressed IPv6 strings are accepted.
The list of `Interfaces`, IPv4 or IPv6 strings, are currently only used for outgoing connections and may only contain *one* address. If set and access is granted, Orber will use that local interface when connecting to the other ORB. The module orber_acl [page 161] provides operations for evaluating the access control for filters and addresses.

`activate_audit_trail() -> Result`
`activate_audit_trail(Verbosity) -> Result`

Types:

- Verbosity = stealth | normal | verbose
- Result = ok | {error, Reason}
- Reason = string()

Activates audit/trail for all existing incoming and outgoing IIOP connections. The `Verbosity` parameter, `stealth`, `normal` or `verbose`, determines which of the built in interceptors is used (`orber_iiop_tracer_stealth`, `orber_iiop_tracer_silent` or `orber_iiop_tracer` respectively). If no verbosity level is supplied, then the `normal` will be used.

In case Orber is configured to use other interceptors, the audit/trail interceptors will simply be added to that list.

`deactivate_audit_trail() -> Result`

Types:

- Result = ok | {error, Reason}
- Reason = string()

Deactivates audit/trail for all existing incoming and outgoing IIOP connections. In case Orber is configured to use other interceptors, those will still be used.

`add_listen_interface(IP, Type) -> Result`

`add_listen_interface(IP, Type, Port) -> Result`

`add_listen_interface(IP, Type, ConfigurationParameters) -> Result`

Types:

- IP = string
- Type = normal | ssl
- Port = integer() > 0
- ConfigurationParameters = [{Key, Value}]
- Key = flags | iiop_in_connection_timeout | iiop_max_fragments | iiop_max_in_requests | interceptors | iiop_port | iiop_ssl_port
- Value = as described in the User's Guide
- Result = {ok, Ref} | {error, Reason} | {'EXCEPTION', #'BAD_PARAM'{}}
- Ref = #Ref
- Reason = string()

Create a new process that handle requests for creating a new incoming IIOP connection via the given interface and port. If the latter is excluded, Orber will use the value of the `iiop_port` or `iiop_ssl_port` configuration parameters. The `Type` parameter determines if it is supposed to be IIOP or IIOP via SSL. If successful, the returned `#Ref` shall be passed to `orber:remove_listen_interface/1` when the connection shall be terminated.

It is also possible to supply configuration parameters that override the global configuration. The *iiop_in_connection_timeout*, *iiop_max_fragments*, *iiop_max_in_requests* and *interceptors* parameters simply overrides the global counterparts (See the Configuration [page 12] chapter in the User's Guide). But the following parameters there are a few restrictions:

- *flags* - currently it is only possible to override the global setting for the `Use Current Interface in IOR` and `Exclude CodeSet Component` flags.
- *iiop_port* - requires that `Use Current Interface in IOR` is activated and the supplied `Type` is `ssl`. If so, exported IOR:s will contain the IIOP port defined by this configuration parameter. Otherwise, the global setting will be used.
- *iiop_ssl_port* - almost equivalent to `iiop_port`. The difference is that `Type` shall be `normal` and that exported IOR:s will contain the IIOP via SSL port defined by this configuration parameter.

If it is not possible to add a listener based on the supplied interface and port, the error message is one of the ones described in `inet` and/or `ssl` documentation.

`remove_listen_interface(Ref) -> ok`

Types:

- Ref = #Ref

Terminates the listen process, associated with the supplied `#Ref`, for incoming a connection. The Ref parameter is the return value from the `orber:add_listen_interface/2/3` operation. When terminating the connection, all associated requests will not deliver a reply to the clients.

`close_connection(Connection) -> Result`

`close_connection(Connection, Interface) -> Result`

Types:

- Connection = Object | [{Host, Port}]
- Object = #objref (external)
- Host = string()
- Port = string()
- Interface = string()
- Result = ok | {'EXCEPTION', #'BAD_PARAM'{}}

Will try to close all outgoing connections to the host/port combinations found in the supplied object reference or the given list of hosts/ports. If a `#'IOP_ServiceContext'`{} containing a local interface has been used when communicating with the remote object (see also Module_Interface [page 122]), that interface shall be passed as the second argument. Otherwise, connections via the default local interface, will be terminated.

Note:

Since several clients maybe communicates via the same connection, they will be affected when invoking this operation. Other clients may re-create the connection by invoking an operation on the target object.

`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 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.

`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} | {initialreferences_storage_type, TableType} | {load_order, Priority}
- Timeout = infinity | integer()
- TableType = disc_copies | ram_copies
- Priority = integer()

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.
- {`initialreferences_storage_type, TableType`} - this option sets the type of table used for storing initial references. The TableType can be disc_copies or ram_copies. Default is ram_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.
- {`load_order, Priority`} - per default the priority is set to 0. Using this option it will change the priority of in which order Mnesia will load Orber internal tables. For more information, consult the Mnesia documentation.

`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, Options) -> RetVal`

Types:

- Node = node()
- Options = IFRStorageType | [KeyValue]
- IFRStorageType = StorageType
- StorageType = disc_copies | ram_copies
- KeyValue = {ifr_storage_type, StorageType} | {initialreferences_storage_type, StorageType} | {nameservice_storage_type, StorageType} | {type, Type}
- Type = temporary | permanent
- 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 | iiop_in_connection_timeout | objectkeys_gc_time | orber_debug_level
- 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 the keys specified above.

Any other key must 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 12] 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_acl

Erlang Module

This module contains functions intended for analyzing Access Control List (ACL) filters. The filters uses a extended format of Classless Inter Domain Routing (CIDR). For example, `"123.123.123.10"` limits the connection to that particular host, while `"123.123.123.10/17"` allows connections to or from any host equal to the 17 most significant bits. Orber also allow the user to specify a certain port or port range, for example, `"123.123.123.10/17#4001"` and `"123.123.123.10/17#4001/5001"` respectively. IPv4 or none compressed IPv6 strings are accepted.

Exports

`match(IP, Direction) -> boolean()`
`match(IP, Direction, GetInfo) -> Reply`

Types:

- IP = tuple() | [integer()]
- Direction = tcp_in | ssl_in | tcp_out | ssl_out
- GetInfo = boolean()
- Reply = boolean() | {boolean(), [Interface], PortInfo}
- Interface = string()
- PortInfo = integer() | {integer(), integer()}

If `GetInfo` is not supplied or set to false, this operation returns a boolean which tells if the IPv4 or IPv6 address would pass the ACL filter, defined by the `iiop_acl` configuration parameter, or not. When `GetInfo` is set to true, a tuple which, besides the boolean that tells if access was granted, also include the defined interfaces and port(s). This operation requires that Orber is running and can be used on a live node to determine if Orber has been properly configured.

`verify(IP, Filter, Family) -> Reply`

Types:

- IP = string()
- Filter = string()
- Family = inet | inet6
- Reply = true | {false, From, To} | {error, string()}
- From = string()
- To = string()

This operation returns true if the IPv4 or IPv6 address would pass the supplied ACL. If that is not the case, a tuple containing the accepted range is returned. This operation should only be used for test purposes.

`range(Filter, Family) -> Reply`

Types:

- Filter = string()
- Family = inet | inet6
- Reply = {ok, From, To} | {error, string()}
- From = string()
- To = string()

Returns the range of accepted IP addresses based on the supplied filter. This operation should only be used for test purposes.

orber_diagnostics

Erlang Module

This module contains functions which makes it possible to run simple tests.

> **Warning:**
> Functions exported by this module may only be used during test and development phase.

Exports

`nameservice() -> Result`
`nameservice(Flags) -> Result`

Types:

- Flags = integer()
- Result = ok | {'EXCEPTION', E}

Displays all objects stored in the NameService. Existent checks are, per default, also performed on all local objects. This can also be activated for external objects by setting the flag `16#01`. The displayed information is the stringified Name described in CosNaming_NamingContextExt [page 120], non existent status (true | false | external | undefined) and the IFR-Id:

```
host/
host/resources/
host/resources/MyObj/ [false] IDL:MyMod/MyIntf:1.0
```

`missing_modules() -> Count`

Types:

- Count = integer()

This operation list missing modules generated by IC and required by Orber. Requires that all API:s are registered in the IFR.

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* 2.3. 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.

Initialization of the IFR

The following functions are used to initialize 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 initialize 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 know, 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_Wstring, dk_Fixed, 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 IRObject, 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_Wstring, dk_Fixed, 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_Wstring, dk_Fixed, 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_Wstring, dk_Fixed, 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_wstring, pk_fixed, 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_wstring(Objref,Bound) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Bound = integer() (unsigned long /= 0)
- Return = #IFR_WstringDef_objref

Creates an IFR objref of the type WstringDef.

`create_fixed(Objref,Digits,Scale) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Digits = Scale = integer()
- Return = #IFR_FixedDef_objref

Creates an IFR objref of the type FixedDef.

`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 parameter description records)
- Return = ok | {exception, _}

Sets the params attribute, a list of parameter description 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 parameter description 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
long_long() -> TC
unsigned_long_long() -> TC
wchar() -> 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.

`wstring(Length) -> TC`

Types:

- Length = integer()
  the length of the wstring (0 means unbounded)

Function returns the IDL typecode for wstring.

`fixed(Digits, Scale) -> TC`

Types:

- Digits = Scale = integer()
  the digits and scale parameters of a Fixed type

Function returns the IDL typecode for fixed.

`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 and 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(TC) -> boolean()`

Function checks the syntax of an IDL typecode.

List of Figures

List of Tables

Glossary

BindingIterator

The binding iterator (Like a book mark) indicates which objects have been read from the list.
Local for chapter 5.

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 2.

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 2.

IDL

Interface Definition Language - IDL is the OMG specified interface definition language, used to define the CORBA object interfaces.
Local for chapter 2.

IIOP

Internet-Inter ORB Protocol
Local for chapter 2.

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 2.

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 3.

Orber installation

is the structure of the ORB or ORBs as defined during the install process is called the "installation".
Local for chapter 3.

Type Code

Type Code is a full definition of a type
Local for chapter 4.

Type Codes

Type codes give a complete description of the type including all its components and structure.
Local for chapter 4.

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.