Orber Application

version 3.1

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

Orber User's Guide

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

1.1 The Orber Application

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.

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

ORB kernel and IIOP support

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

Interface Repository

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

IDL to Erlang mapping

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

CosNaming Service

Orber contains a CosNaming compliant service.

Resolving initial references from Java or C++

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

Resolving initial reference from Java
A class with only one method which returns an *IOR* on the external string format to the INIT object (see “Interoperable Naming Service” specification).

Resolving initial reference from C++
A class (and header file) with only one method which returns an IOR on the external string format to the INIT object (see “Interoperable Naming Service” specification).

Orber stub/skeleton

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

1.2 Introduction to Orber

Overview

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

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

The Orber application contains the following parts:

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

Benefits

Orber provides CORBA functionality in an Erlang environment that enables:

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

Purpose and Dependencies

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

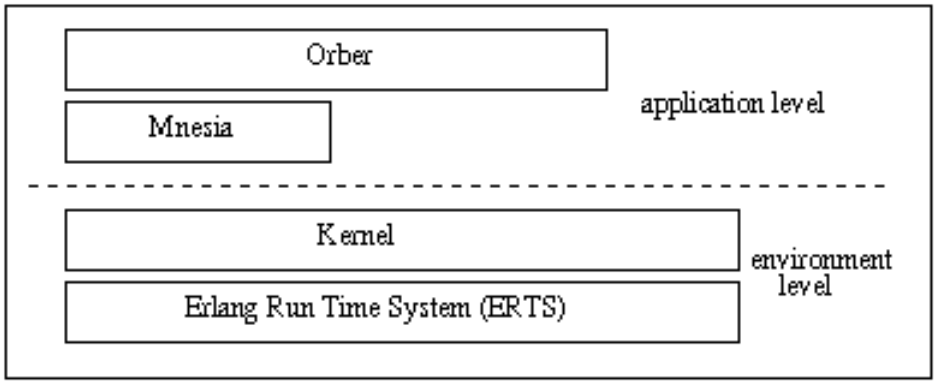

Figure 1.1: Figure 1: Orber Dependencies and Structure.

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

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

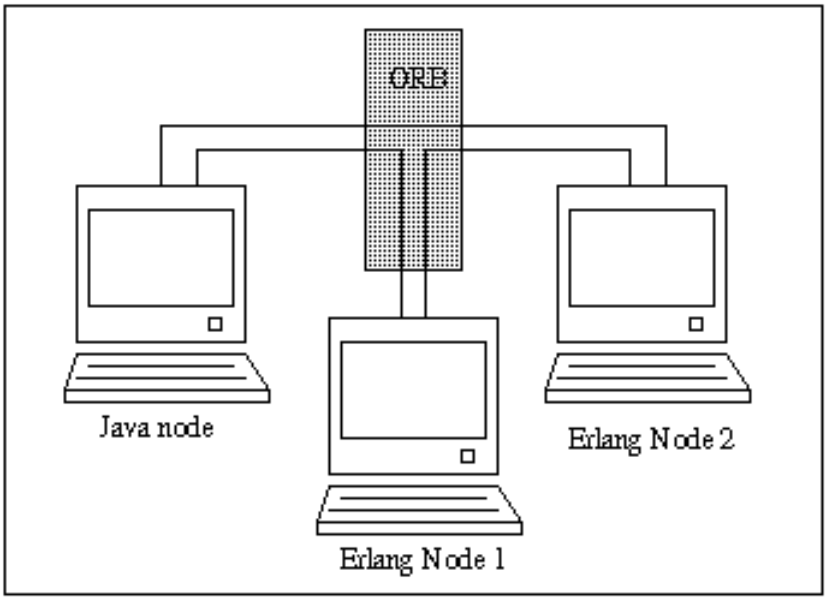

Figure 1.2: Figure 2: ORB interface between Java and Erlang Environment Nodes.

This simplified illustration in figure 2 demonstrates how Orber can facilitate communication in a heterogeneous environment. The Erlang Nodes running OTP and the other Node running applications written in Java can communicate via an *ORB* (Object Request Broker). Using Orber means that CORBA functions can be used to 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 *CORBA, Fundamentals and Programming - Jon Siegel* and *Open Telecom Platform Documentation Set*. It is also helpful to have read *Concurrent Programming in Erlang*.

1.3 The Orber Application

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.

The Object Request Broker (ORB)

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

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

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

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

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

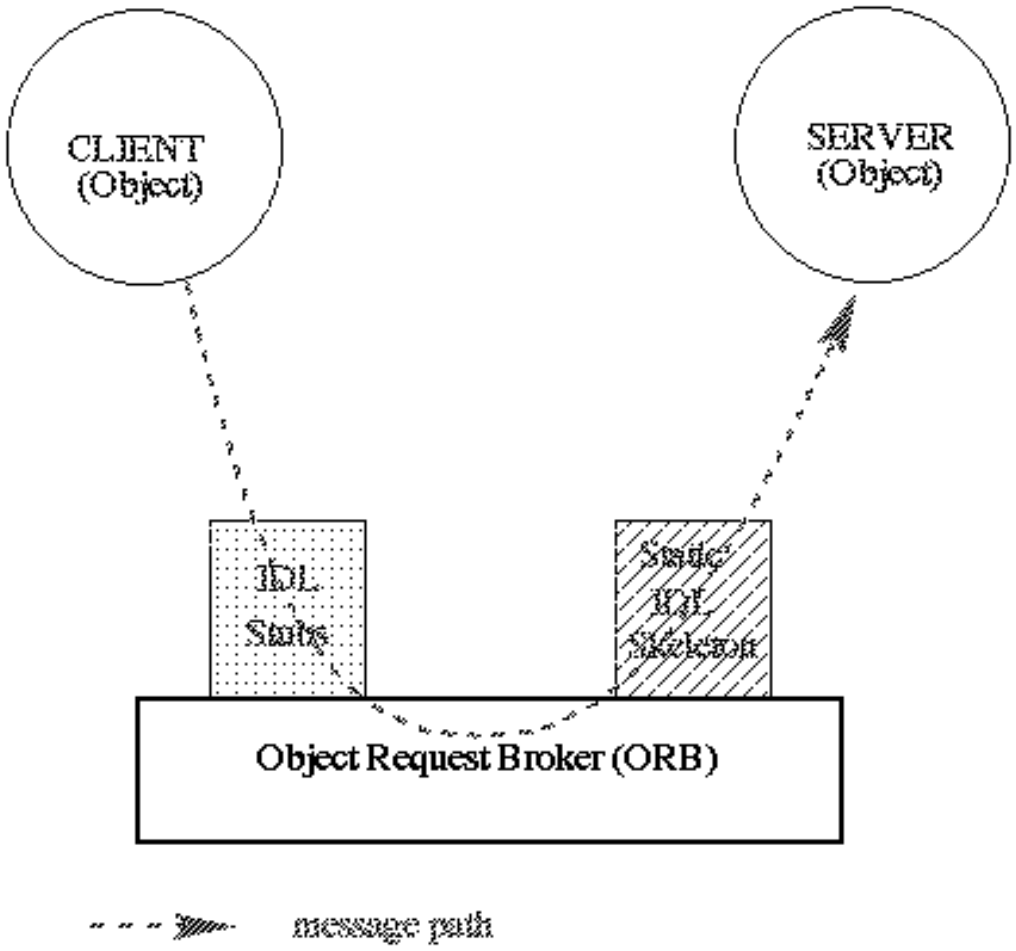

Figure 1.3: Figure 1: How the Object Request Broker works.

The domain in Orber gives an extra aspect to the distributed object environment as each domain has one ORB, but it is distributed over a number of object in different nodes. The domain binds objects on nodes more closely than distributed objects in different domains. The advantage of a domain is that a faster communication exists between nodes and objects of the same domain. An internal communication protocol (other than IIOP) allows a more efficient communication between these objects.

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

Internet Inter-Object Protocol (IIOP)

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

Figure 2 below demonstrates how IIOP works between objects:

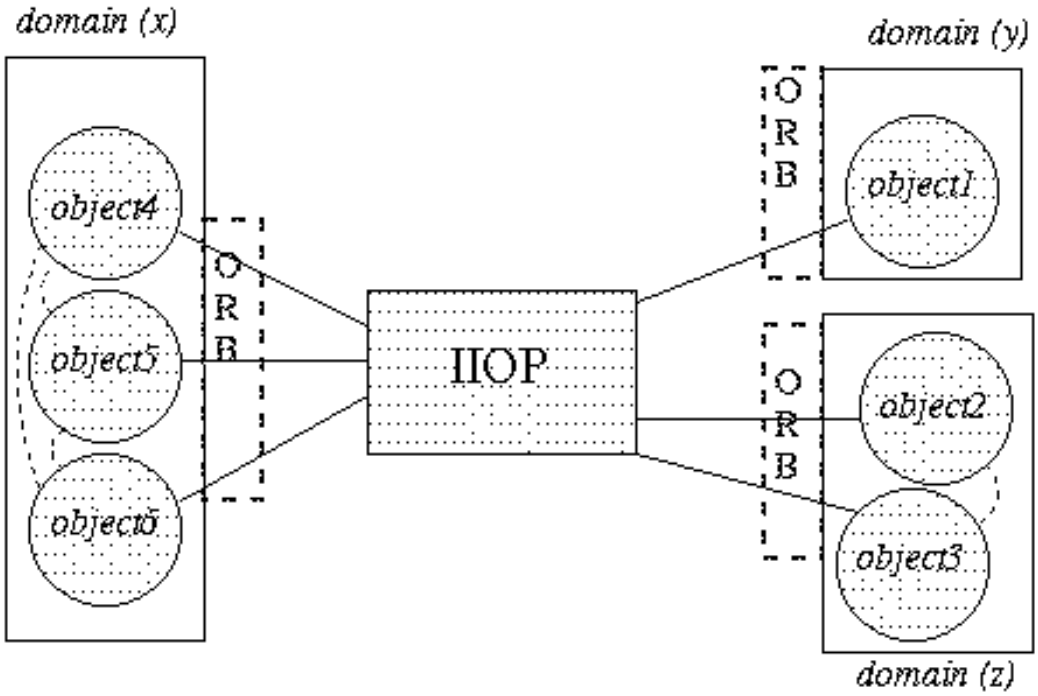

Figure 1.4: Figure 2: IIOP communication between domains and objects.

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

1.4 Interface Repository

Interface Repository(IFR)

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

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

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

1.5 Installing Orber

Installation Process

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

Preparation

Before beginning the installation process for Orber, a Mnesia database schema must exist. This schema will contain information about the location of the Erlang nodes where Orber is planned to be run.

The Mnesia schema can be created by calling the following code in an Erlang shell:

```
Mnesia:create_schema(NodeList)
```

`NodeList` is the list of Erlang node names.

Installing Orber

The next step is to actually install Orber. When the installation is completed Orber will automatically create a few Orber specific Mnesia tables and load them with data.

The installation process will differ slightly depending on whether Orber is running on one or many nodes or if Mnesia is currently running.

Functions to choose from are:

- `orber:install(NodeList).`
- `orber:install(NodeList, Options).`

Installation `Options` is a choice between multi-node or single node installation.

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

Single Node Installation Single node (non-Distributed) installation means that Orber processes will be installed and started on only one node.

In this case, Orber still facilitates external communication with other ORBs through the IIOP protocol.

Single node installation of Orber is suitable in cases where:

- Capacity is greater than load (volume of traffic)
- Distributed system architecture requires an *Orber installation* on only one node.

Below, is an example of a one node installation where Mnesia is not installed. It is not necessary to have Mnesia running when installing Orber on a single node, as Orber will start Mnesia automatically.

Open an Erlang shell and install the application by typing:

```
1> mnesia:create_schema([]).

2> orber:install([]).
```

Note:
In the above example the node list is empty, as the default option is the current node.

Multi-node installation For a multi-node installation there are two extra steps. All nodes must be started and Mnesia must be running.

Below is an example of a multi-node installation where Mnesia is installed:

```
1> orber:install([a@machine1, b@machine2]).
```

Running Java clients against Orber. If you intend to run Java clients, a specific

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

must be added to your *CLASSPATH* variable to allow Orber support for the initial references.

Configuration

The following configuration parameters exist:

- *domain* - default is “ORBER”. The value is a string. As Orber domains must have unique names, problems can arise if two domains have the same name.
- *iiop_port* - default 4001. The value is an integer.
  *Note:*On a UNIX system it is preferable to have a IIOP port higher than 1023, since it is not recommended to run Erlang as a root user.
- *bootstrap_port* - It is used for fetching initial service references and has the IIOP port as the default setting. The value is an integer.
- *orber_nodes* - default is the current Erlang node (this must be set if Orber shall execute on more than one Erlang node). The value is a list of Erlang node names.
- *ip_address* - default is all interfaces. This option is used if orber only should listen on a specific ip interface on a multiinterface host. The value is the ip address as a string or a tuple of four integers.
- *objectkeys_gc_time* - default is `infinity`. This option is should be set if objects are started using the option {`persistent, true`}. The value is, `integer()`, seconds. four integers.
- *giop_version* - default is IIOP 1.1. IIOP Version 1.0 is still usable but you have to this configuration variable, e.g., `erl -orber giop_version "{1,0}"`
- *iiop_connection_timeout* - default is infinity. The value is an integer (timeout in seconds between 0 and 1000000) or the atom infinity. This option is only valid for client object connections, i.e., will have no effect on server connections. Setting this options 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 seceonds have passed.
- *iiop_timeout* - default is infinity. The value is an integer (timeout in seconds between 0 and 1000000) or the atom infinity. This option is only valid on the client side. Setting this option, cause all intra-ORB requests to timeout and raise the `COMM_FAILURE` system exception if no replies are delivered within the given time limit.
- *lightweight* - default is false. This option must be set if Orber is supposed to be started as lightweight. The value is a list of RemoteModifiers, equal to the `orber:resolve_initial_references_remote/2` argument. The list must contain Orber nodes addresses, to which we have access and are not started as lightweight.
- *orbInitRef* - default is undefined. Setting this option, e.g., `erl -orber orbInitRef \"NameService=corbaloc::host.com/NameService\"` or `erl -orber orbInitRef \"IOR:1234567890...\"`, will alter the location from where `orber:resolve_initial_references(Key)` tries to find an object matching the given Key. This variable overrides `orbDefaultInitRef`
- *orbDefaultInitRef* - default is undefined. If a matching Key for `orbInitRef` is not found, and this variable is set, it determines the location from where `orber:resolve_initial_references(Key)` tries to find an object matching the given Key. Usage: `erl -orber orbDefaultInitRef \"corbaloc::host.com\"`

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

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

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

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

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

1.6 OMG IDL Mapping

OMG IDL Mapping - Overview

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

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

OMG IDL mapping elements

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

All mapping must define the following elements:

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

Reserved compiler names

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

IC reserves all identifiers starting with `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.

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

Refer to the IC documentation for further details.

Basic OMG IDL types

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

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

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

Table 1.1: OMG IDL basic types

The `any` value is written as a record with the field typecode which contains the *Type Code* representation, see also the Type Code table [page 20], and the value field itself.

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

Constructed OMG IDL types

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

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

Table 1.2: OMG IDL constructed types

Below are examples of values of constructed types.

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

Table 1.3: Typical values

References to constants

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

Example:

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

Would result in the following conceptual code:

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

c1() -> 99.
```

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 18].

Invocations of operations

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

Example:

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

Is used in Erlang as :

```
%% Erlang
f() ->
    ...
    Obj = ...    %% get object reference
    R1 = i1:op1(Obj, 55),
    {R2, S, Count} = i1:op2(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.

Operation implementation

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

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

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

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

See also the stack example. [page 22]

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.

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.

Access to attributes

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

Typecode, Identity and Name access functions.

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

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

For example:

```
// IDL
module m {

  struct s {
    long x;
    long y;
   };

};
```

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

```
-module(m_s).

-include("m.hrl").

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


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

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

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

Type Code representation

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

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

continued ...

... continued

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

Table 1.4: Type Code tuples

Scoped names

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

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

Syntax Specific structures for scoped names

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

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

Files

Several files can be generated for each scope.

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

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

Example:

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

  struct s {
    long x;
    long y;
  };

  interface i {

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

 };

};
```

This will produce the following files:

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

A mapping example

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

```
// The source IDL file

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

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

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

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

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

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

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

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

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

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

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

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

```
-module(stack_impl).

-include("stack.hrl").

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

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

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

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

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

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

```
-module(calc_impl).

-export([add/1]).

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

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

1.7 CosNaming Service

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 allways resolved in a context, there no absolute names exist. Because a context is like any other object, it can also be bound to a name in a naming context. This will result in a naming graph (a directive graph with 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 refered to as *name* and the individual elements in the sequence as *name components*. All but the last name component are bound to naming contexts.

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

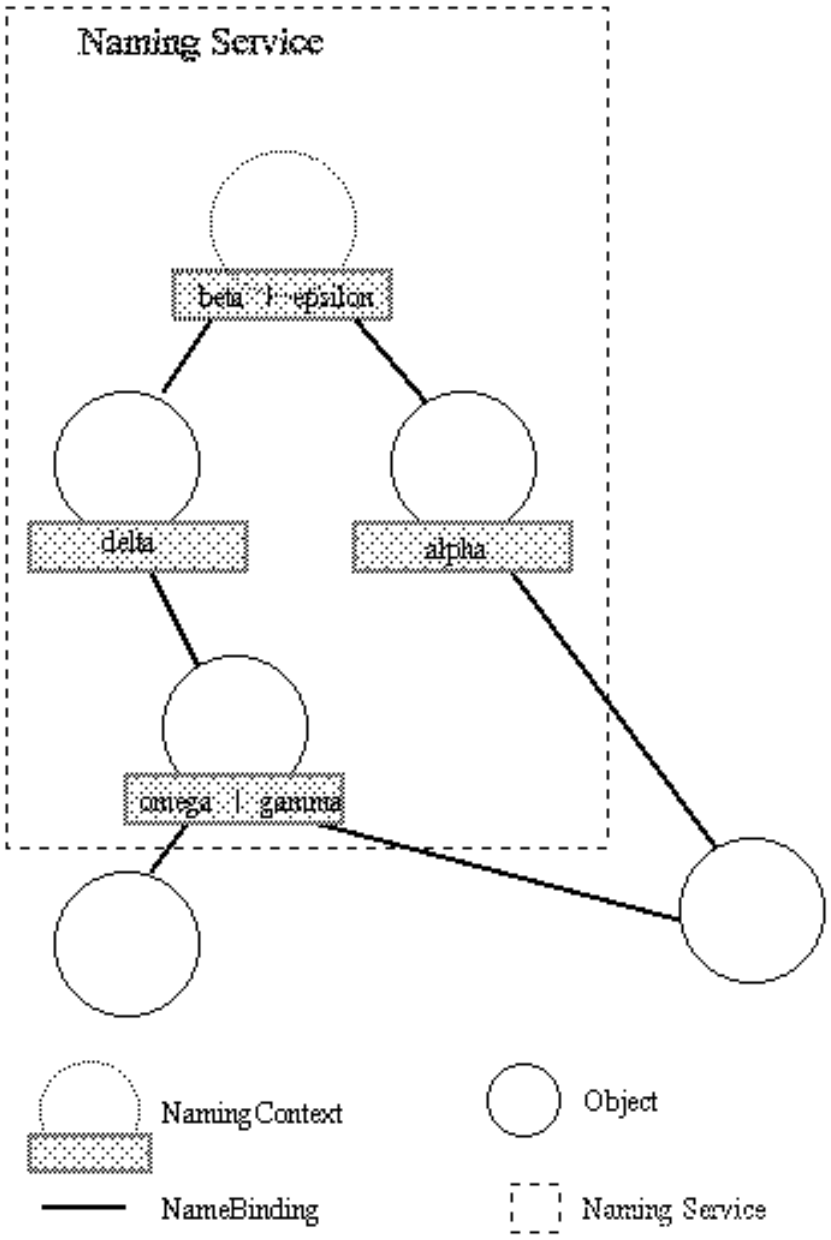

Figure 1.5: Figure 1: Contextual object relationships using the Naming Service.

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

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

The Naming Service has an initial naming context, which is shown in the diagram as the top-most object in the naming graph. It has two names `beta` and `epsilon`, which are bound to other naming contexts. The initial naming context is a well known location used to share a common name space

between multiple programs. You can traverse the naming graph until you reach a name, which is bound to an object, which is not a naming context.

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

The Basic Use-cases of the Naming Service

The basic use-cases of the Naming Service are:

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

Fetch initial reference to the naming service

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

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

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

Creating a naming context

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

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

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

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

Binding and unbinding names to objects

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

1. Use the naming library functions to create a name

   ```
   Name = lname:new(["object"]).
   ```

2. Use CosNaming::NamingContext::bind() to bind a name to an object

   ```
   'CosNaming_NamingContext':bind(Sc, Name, Object).
   ```

3. Use CosNaming::NamingContext::unbind() to remove the NameBinding from an object

   ```
   'CosNaming_NamingContext':unbind(Sc, Name).
   ```

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

Resolving a name to an object

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

1. Use the naming library functions to create a name path:

   ```
   Name = lname:new(["workgroup", "services"]).
   ```

2. Use CosNaming::NamingContext::resolve() to to resolve the name to an object

   ```
   Sc = 'CosNaming_NamingContext':resolve(NS, Name).
   ```

Listing the bindings in a NamingContext

1. Use CosNaming::NamingContext::list() to list all the bindings in a context
   The following code retrieves and lists up to 10 bindings from a context.

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

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

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

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

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

Destroying a naming context

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

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

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

Interoperable Naming Service

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

- IOR
- corbaloc
- corbaname

IOR

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

corbaloc

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

```
<corbaloc>          =  "corbaloc:"<obj_addr_list>["/"<key_string>]
<obj_addr_list>     = [<obj_addr>","]*<obj_addr>
<obj_addr>          = <prot_addr> | <future_prot_addr>
<prot_addr>         = <rir_prot_addr> | <iiop_prot_addr>
<rir_prot_addr>     = <rir_prot_token>":"
<rir_prot_token>    = rir
<future_prot_addr> = <future_prot_id><future_prot_addr>
<future_prot_id>    = <future_prot_token>":"
<iiop_prot_addr>    = <iiop_id><iiop_addr>
<iiop_id>           = <iiop_default> | <iiop_prot_token>":"
<iiop_default>      = ":"
<iiop_prot_token>   = "iiop"
<iiop_addr>         = <version><host>[":"<port>]
```

```
<host>              = DNS-style Host Name | ip_address
<version>           = <major>"."<minor>"@"  | empty_string
<port>              = number
<major>             = number
<minor>             = number
```

The corbaloc scheme consists of 3 parts:

- Protocol - as of today iiop or rir is supported. Using rir means that we will resolve the given Key locally, i.e., the same as using corba:resolve_initial_references("NameService").
- IIOP address - this address can be divided into Version, Host and Port. If the version or port are left out they will be set to the default values 1.0 and 2089 respectively.
- KeyString - a stringified 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.

corbaname

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

```
<corbaname>         = "corbaname:"<corbaloc>["#"<string_name>]
<corbaloc>          = 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 1.5: Table 1: Stringified Name represenation

After creating a stringified Name we can either use:

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

or concatenate the Name String using:

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

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

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

1.8 How to use security in Orber

Security in Orber

Introduction

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

Enable usage of secure connections

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

The default is no security.

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

Configurations when Orber is used on the server side

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

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

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

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

Configurations when Orber is used on the client side

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

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

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

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

Access functions:

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

Modify functions:

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

1.9 Orber Examples

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.

```
%% <!--
%% ‘‘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 via the world wide web 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.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.’’
%%
%%     $Id$
%%--> StackModule_Stack_impl example file.

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


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

terminate(From, Reason) ->
    ok.

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

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

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

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

```
%% <!--
%% ‘‘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 via the world wide web 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.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%--> 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.

```
%% <!--
%% ``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 via the world wide web 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.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
```

```
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%--> 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.

```
%% <!--
%% ''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 via the world wide web 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.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%--> stack_client example file.

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

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


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

```
    run_1(NS).

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


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

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

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

    end.
```

Writing a client in Java

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

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

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

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

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

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

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

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

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

```
/*
 * Stack example.
 */
```

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

public class StackClient
{
  public static void main(String args[])
    {
      NamingContext nsContext;
      org.omg.CORBA.Object objRef;
      StackFactory sfRef = null;
      Stack sRef = null;
      org.omg.CORBA.Object nsRef, initRef;
      NameComponent[] name = new NameComponent[1];
      Orber.InitialReference ir = new Orber.InitialReference();
      Orber.InitialReferences init;
      String srvHost = new String(args[0]);
      Integer srvPort = new Integer(args[1]);

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

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

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

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

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

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

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

          sfRef.destroy_stack(sRef);

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

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

Building the example

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

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

Eshell V4.9  (abort with ^G)
1> ic:gen(stack).
Erlang IDL compiler version 20
ok
2> make:all().
Recompile: oe_stack
Recompile: StackModule_StackFactory
Recompile: StackModule_Stack
Recompile: StackModule
Recompile: stack_client
```

```
Recompile: stack_factory
Recompile: StackModule_StackFactory_impl
Recompile: StackModule_Stack_impl
up_to_date
3>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a

fingolfin 128> idl stack.idl
fingolfin 129> idl InitialReferences.idl
fingolfin 130> idl <OTP_INSTALLATIONPATH>/lib/orber-<Orber Version>/COSS/
                   CosNaming/cos_naming.idl
fingolfin 131>
fingolfin 132> cd java_output/
fingolfin 133> javac *.java
fingolfin 134> cd CosNaming/
fingolfin 135> javac *.java
fingolfin 136> cd ../_NamingContext/
fingolfin 137> cd javac *.java../_NamingContext/
fingolfin 138> cd ../../CORBA/
fingolfin 139> javac *.java
fingolfin 140> cd ../StackModule/
fingolfin 141> javac *.java
fingolfin 142> cd ../..
fingolfin 143> javac *.java
fingolfin 144> cp StackClient.class java_output/StackModule/.
```

How to run everything

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

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

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

```
1
7
4
ok
7>
```

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

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

```
fingolfin 38> java StackModule.StackClient fingolfin 4001
[New Connection (fingolfin,4001, null,null,pid=0) ]
[New Connection (fingolfin.du.etx.ericsson.se,4001, null,null,pid=0) ]
1
1
7
4
Empty stack
fingolfin 39>
```

A tutorial on how to start Orber as lightweight

Preparation

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

Lightweight Orber do not allow us to:

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

With lightweight Orber we do not:

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

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

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

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

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

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

Eshell V4.9    (abort with ^G)
1> orber:start_lightweight().
2> NS=orber:resolve_initial_references_remote("NameService", ["iiop://hostX:port"]).
3> NC=lname_component:set_id(lname_component:create(), "myFactory").
4> N =lname:insert_component(lname:create(), 1, NC).
5> Factory='CosNaming_NamingContext':resolve(NS, N)).
6> Obj=MyFactory_Creater:MyObject(Factory, Args).
7> MyObject:myFunction(Obj,Args2).
```

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

1.10 Orber Stubs/Skeletons

Orber stubs and skeletons description.

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

Server start

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

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

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

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

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

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

- Initial data can be passed using `Env`.
- Cannot be used as a supervisor child start function.
- Accessible through the object reference returned by the start function. If the option {`regname, RegName`} is used the object reference stays valid even if the server has been restarted.
- If the options {`persistent, true`} and {`regname, {global, Name}`} is used, the result from an object invocation will be the exception 'OBJECT_NOT_EXIST' only if the object has terminated with reason `normal` or `shutdown`. If the object is in the process of restarting, the result will be {`error, Reason`} or the exception `'COMM_FAILURE'`.
- The option {`pseudo, true`} makes it possible to start create non-server objects. There are, however, some limitations, which are further described in the `Pseudo objects` section.

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

- Initial data can be passed using `Env`.
- Can be used as a supervisor child start function if the option {`sup_child, true`} used.
- Accessible through the object reference returned by the start function. If the option {`regname, RegName`} is used the object reference stays valid even if the server has been restarted.
- If the options {`persistent, true`} and {`regname, {global, Name}`} is used, the result from an object invocation will be the exception 'OBJECT_NOT_EXIST' only if the object has terminated with reason `normal` or `shutdown`. If the object is in the process of restarting, the result will be {`error, Reason`} or the exception `'COMM_FAILURE'`.

- For starting a server as a supervisor child you should use the options [{persistent, true}, {regname, {global, Name}}, {sup_child, true}] and of type *transient*. This configuration allows you to delegate restarts to the supervisor and still be able to use the same object reference and be able to see if the server is permanently terminated. Please note you must use *supervisor/stdlib-1.7* or later and that 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 gen_server reply Timeout have no effect.
- The option {pseudo, true} overrides all other start options.
- Only the functions, besides own definitions, init/2 (called via oe_create*/2) and terminate/2 (called via corba:dispose/1) must be implemented.

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

To create a pseudo object do the following:

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

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

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

Call-back module

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

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

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

-module('Module_Interface_impl').

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

%%--------------- EXPORTS------------------------------------
%% Arity depends on IC configuration parameters and the IDL
%% specification.
-export([own_function/X]).


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

%%----------------------------------------------------------
%% function : server specific
%%----------------------------------------------------------
init(InitialData) ->
```

```
    %% 'trap_exit' optional (have no effect if pseudo object).
    process_flag(trap_exit,true),

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

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

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

terminate(Reason, State) ->
    ok.

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

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

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

%% If not use IC option {this, "Module:Interface"}
own_function(State, .. Arguments ..) ->
    %%--- Possible replies ---
    %% Reply and await next request
    {reply, Reply, State}

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

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

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

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

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

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

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

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

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

1.11 Orber Release Notes

Orber 3.1.8, Release Notes

Improvements and new features

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

Fixed bugs and malfunctions

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

Incompatibilities

-

Known bugs and problems

- The same as in last release.

Orber 3.1.7, Release Notes

Improvements and new features

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

Fixed bugs and malfunctions

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

Incompatibilities

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

Known bugs and problems

- The same as in last release.

Orber 3.1.6, Release Notes

Improvements and new features

- Cosmetic update of internal functions.
  Own id: -

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

- The same as in last release.

Orber 3.1.5, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

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

Incompatibilities

-

Known bugs and problems

- The same as in last release.

Orber 3.1.4, Release Notes

Improvements and new features

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

Fixed bugs and malfunctions

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

Incompatibilities

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

Known bugs and problems

- The same as in last release.

Orber 3.1.3, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

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

Incompatibilities

-

Known bugs and problems

- The same as in last release.

Orber 3.1.2, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

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

Incompatibilities

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

Known bugs and problems

- The same as in last release.

Orber 3.1.1, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

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

Incompatibilities

-

Known bugs and problems

- The same as in last release.

Orber 3.1, Release Notes

Improvements and new features

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

Fixed bugs and malfunctions

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

Incompatibilities

-

Known bugs and problems

- The same as in last release.

Orber 3.0.1, Release Notes

Improvements and new features

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

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

- The same as in last release.

Orber 3.0, Release Notes

Improvements and new features

- It is now possible to use secure IIOP connections to and from Orber. Orber currently only supports security with the help of SSL and not SECIOP.
  Own Id: OTP-1510
- It is now possible to start Orber objects as supervisor childs using Module_Interface:oe_create_link/2 or corba:create_link/4 as the start function.
  Own Id: -
- It is now possible to start a Orber object and be able to tell apart if it is in the process of being restarted or has permanently terminated. This is also the reason for introducing `objectkeys_gc_time` configuration parameter.
  Own Id: -
- The service CosEvent has been removed from orber and become its own application, called cosEvent.
  Own Id: -
- The service CosTransactions is now available as a separate application, called cosTransactions.
  Own Id: OTP-1741
- Three new system exceptions, 'TRANSACTION_REQUIRED', 'TRANSACTION_ROLLEDBACK' and 'INVALID_TRANSACTION', introduced. Required by the cosTransactions application.
  Own Id: -
- An configuration variable ip_address has been added, so it's possible to listen on a specific ip interface on a multi interface host. The value is the ip address as a string or a tuple of four integers, default value is all interfaces.
  Own Id: OTP-3294

Fixed bugs and malfunctions

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

Incompatibilities

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

Known bugs and problems

- The same as in last release.

Orber 2.2.2, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- Allignment error in the IIOP decoding/encoding of doubles fixed.
  Own Id: OTP-3185
- Removed a to strict guard on float/double cdr encoding.
  Own Id: OTP-3186
- Orber now accepts parallell requests on the same socket.
  Own Id: OTP-3198

Incompatibilities

-

Known bugs and problems

- The same as in last release.

Orber 2.2.1, Release Notes

Improvements and new features

- In this version of Orber we have added orber:add_node/2 and orber:remove_node/1\n to make it possible to add/remove an Orber node to/from a set of running Orber nodes.
  Own Id: OTP-3103
- A global timeout on outgoing IIOP calls have been added as a configuration variable to Orber. It has the name `iiop_timeout` and can be set to a value in seconds. If not set it will have the value infinity.
  Own Id: OTP-3151

Fixed bugs and malfunctions

- An error when decoding locate requests from IIOP is fixed.
  Own Id: OTP-3149
- There was always a negative response for a locate request on the initial reference (INIT) because of an error in the existence check function. This is now fixed.
  Own Id: OTP-3150
- `InitialReferences.idl` was not according to the standard. The modules name is now changed from `Orber` to `CORBA`. This will affect code which are using this interface. The idl specification must be recompiled and then `Orber` must be changed to `CORBA` in the client.
  Own Id: OTP-3155

Incompatibilities

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

Known bugs and problems

ORB

- The CORBA dynamic interfaces (DII and DSI) are not supported.
- Orber only supports persistent object startup behaviour.
- There are a number of functions in the BOA and CORBA interfaces that are not implemented but are mostly used only when implementing the ORB, and generating IDL compiler stubs and skeletons. These functions are not used by application designers.

Interface Repository

- For the moment, the Interface Repository cannot be used from another ORB.
- IFR will register corruption when trying to register on already defined IDs. This is a problem that appears when trying to call the registration function without unregistering old IFR-objects with the same ID.

Resolving initial reference from C++ The returned IOR is the same for both C++ and Java. However, we have only tested on a client implemented in C++. That is an Orbix C++ client accessing an Orber server.

Orber 2.2, Release Notes

Improvements and new features

- In this version of Orber we have added IIOP 1.1 as default protocol to other ORB's. IIOP 1.0 is still usable but you have to set a configuration variable *giop_version* to get it. We don't support all the new IIOP types because the IDL compiler is not updated yet, but all the headers are updated so the protocol works.
  Own Id: OTP-3092
- The omg.org prefix has been added to CosNaming and CosEvent specifications. This means that the IDL types for these two services now have changed and are incompatible but the names are now according to the CORBA standard.
  Own Id: OTP-3093
- A couple of name creation functions have been added to the naming library. These are not in the CosNaming standard but they are easier to use in the Erlang environment. It doesn't matter that they're not standard because the objects in the naming library are just pseodo objects and are never sent to other ORB's. The changes are in the modules lname and lname_component and the functions are described in the reference manual.
  Own Id: OTP-3094

Fixed bugs and malfunctions

-

Incompatibilities

- IIOP 1.1 is now default protocol version but orber can be configured to run 1.0.
- The omg.org prefix which all standard IDL specification must have has been added. This means that CosEvent and CosNaming now have new type names for all their definitions.

Known bugs and problems

ORB

- The CORBA dynamic interfaces (DII and DSI) are not supported.
- Orber only supports persistent object startup behaviour.
- There are a number of functions in the BOA and CORBA interfaces that are not implemented but are mostly used only when implementing the ORB, and generating IDL compiler stubs and skeletons. These functions are not used by application designers.

Interface Repository

- For the moment, the Interface Repository cannot be used from another ORB.
- IFR will register corruption when trying to register on already defined IDs. This is a problem that appears when trying to call the registration function without unregistering old IFR-objects with the same ID.

Resolving initial reference from C++ The returned IOR is the same for both C++ and Java. However, we have only tested on a client implemented in C++ ie.an Orbix C++ client accessing an Orber server.

Orber 2.1, Release Notes

Improvements and new features

In this version of Orber we have added IIOP 1.1, not all types but the protocol headers should be handled correct. IIOP 1.0 is still the default protocol so orber is fully compatible with previous version, but in OTP R5A IIOP 1.1 will be default protocol (it will be possible to configure the system for 1.0).

Fixed bugs and malfunctions

- Orber now handles the functions _is_a and _not_existent over IIOP.
  Own Id: OTP-2230
- A new function orber:uninstall/0 is added so one can clean up an orber installation.
  Own Id: OTP-3027
- Orber has an improved error message if orber:start is run before orber:install.
  Own Id: OTP-3028

Incompatibilities

-

Known bugs and problems

ORB

- The CORBA dynamic interfaces (DII and DSI) are not supported.
- Orber only supports persistent object startup behaviour.
- There are a number of functions in the BOA and CORBA interfaces that are not implemented but are mostly used only when implementing the ORB,and generating IDL compiler stubs and skeletons. These functions are not used by application designers.

Interface Repository

- For the moment, the Interface Repository cannot be used from another ORB.
- IFR will register corruption when trying to register on already defined IDs. This is a problem that appears when trying to call the registration function without unregistering old IFR-objects with the same ID.

Resolving initial reference from C++ The returned IOR is the same for both C++ and Java. However, we have only tested on a client implemented in C++ ie.an Orbix C++ client accessing an Orber server.

Orber 2.0.2, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- Communication problems under NT, caused by erranous closing of a socket when using long version of hostname when accessing a remote NameService.
  Own Id: OTP-2757
- Hangings related to orber usage, caused by erranous closing of a socket when using long version of hostname when accessing a remote NameService.
  Own Id: OTP-2758
- Private fields - CORBA objects. This was just an error in the example code for the stack client.
  Own Id: OTP-2859

Incompatibilities

-

Known bugs and problems

ORB

- The CORBA dynamicinterfaces (DII and DSI) are not supported.
- Orber only supports persistent object startup behaviour.
- There are a number of functions in the BOA and CORBA interfaces that are not implemented but are mostly used only when implementing the ORB,and generating IDL compiler stubs and skeletons. These functions are not used by application designers.

Interface Repository

- For the moment, the Interface Repository cannot be used from another ORB.
- IFR will register corruption when trying to register on already defined IDs. This is a problem that appears when trying to call the registration function without unregistering old IFR-objects with the same ID.

Resolving initial reference from C++ The returned IOR is the same for both C++ and Java. However, we have only tested on a client implemented in C++ ie.an Orbix C++ client accessing an Orber server.

Orber 2.0.1, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- The application environment variable domain in orber can now be sent as an atom when starting the Erlang node. Example: erl -orber domain Name
  Own Id: OTP-2745
- An error in Orber iwhich resulted in a crash when an exception was sent over IIOP is fixed.
  Own Id: OTP-2931
- Problems in C++ with narrow of initial reference returned by the InitialReference class fixed. Both the C++ and Java implementations of the InitialReference class used the ‘ old module name ORBER instead of Orber. OrbixWeb (java) worked anyway but Orbix (C++) got an exception.
  Own Id: OTP-2935

Incompatibilities

-

Known bugs and problems

ORB

- The dynamic interfaces are not supported and won't be in the first release of Orber.
- Orber only supports persistent object startup behaviour.
- There are a number of functions in the BOA and CORBA interfaces that are not implemented but are mostly used only when implementing the ORB,and generating IDL compiler stubs and skeletons. These functions are not used by application designers.

Interface Repository

- For the moment, the Interface Repository cannot be used from another ORB.
- IFR will register corruption when trying to register on already defined IDs. This is a problem that appears when trying to call the registration function without unregistering old IFR-objects with the same ID.

Resolving initial reference from C++ The returned IOR is the same for both C++ and Java. However, we have only tested on a client implemented in C++ ie.an Orbix C++ client accessing an Orber server.

orber 2.0, Release Notes

Improvements and new features

- It is now possible to start an corba object with a registered name, this can be a local name known only in the same Erlang node or a global name which can be seen in the whole system. This functionality is useful when one is designing application which will be restarted on other nodes when one the first node is going down.
  Own Id: OTP-2486
- It is now possible to install orber so the Interface Repository uses RAM base mnesia tables instead of disc based.
  Own Id: OTP-2484
- The IDL compiler has been removed from orber and become its own application, called ic.
  Own Id: OTP-2483
- It Is now possible to have different Orber nodes talking to each other with IIOP instead of just Erlang distribution. This is solved through a configuration parameter called domain. If the server objects object key has a domain name that differs from the senders domain name IIOP is used.
  Own Id: OTP-2397
- There is now a possibility to have sub objects in an orber object. These sub objects are not distinguishable from ordinary objects from the outside. This functionality can be useful when one just wants one process to handle a number of objects of the same type.
  Own Id: OTP-2396
- Performance tuning, the calls internal in an Erlang node to an orber object is now more efficient. The overhead that Corba adds is minimised so it will especially visible on calls with a small amount of data.
  Own Id: OTP-2111

Fixed bugs and malfunctions

- A bug in orber_ifr:lookup/2 have been fixed.
  Own Id: OTP-2172
- The encoding problem with arrays in IIOP is now fixed.
  Own Id: OTP-2367
- A Marshalling error in the IIOP encoding of any objects corrected. It existed for all the complex types, tk_objref, tk_struct, tk_union, tk_enum, tk_array, tk_sequence tk_alias and tk_exception.
  Own Id: OTP-2391
- A crash under IFR registration and unregistration when modules with inherited interfaces is now fixed.
  Own Id: OTP-2254

Incompatibilities

- There are a number of modules which now are prefixed, but object.erl is the only one which is included in the external interface (it is changed to corba_object.erl). The data type “any” is the only module without prefix now.
  Own Id: OTP-2305
- A hidden field which contains the IFR id in the record definitions will be removed. This will require a regeneration of all IDL specs.
  Own Id: OTP-2480
- The any type is now represented as a record and not just a two tuple which makes it possible to check the type in guards. The two tuple {<TypeCode>, <Value>} is now defined as: `-record(any,{typecode, value}).`
  Own Id: OTP-2480
- IDL unions are represented as Erlang records in the same manner as IDL structs which makes it possible to use the names in guards.
  Own Id: OTP-2481
- The prefix OE_ which has been used on some modules and functions have been changed to oe_.
  Own Id: OTP-2440
- The corba:create function is renamed to corba:create_link and a new corba:create function have been added. This means that corba:create have changed its semantics a bit and if the old behaviour is wanted corba:create_link should be used. These functions are now the corba similar to gen_server:start and gen_server:start_link in behaviour.
  The IDL compiler now also generates create functions (oe_create and oe_create_link with different number of parameters) in the api module which are more convenient to call than the create functions in the corba module because they have less parameters but does the same thing.
  Own Id: OTP-2442

Known bugs and problems

ORB

- The dynamic interfaces are not supported and won’t be in the first release of Orber.
- Orber only support the persistent object startup behaviour.
- There are a number of function in the boa and corba interfaces that not are implemented but they are mostly used when implementing the ORB and in the stubs and skeletons generated by the IDL compiler and not used by application designers.

Interface Repository

- The Interface Repository cannot be used from another ORB for the moment.
- IFR register corruption when trying to register on already defined id's. This is a problem that appears when trying to call the registration function without unregistering old ifr-objects with the same id's.

Resolving initial reference from C++ The returned IOR is correct and the same as for the java implementation but we have for the moment just tested with a client implemented in C++, ie an Orbix C++ client accessing an Orber server.

Orber 1.0.3, Release Notes

Fixed bugs and malfunctions

- Inherited interfaces are now registered correctly in the Interface Repository. This means that object:get_interface/1 now work properly.
  Own Id: OTP-2134
- The generated function which unregisters IDL specifications from the Interface repository crashed when when modules contained interfaces which inherited other interfaces.
  Own Id: OTP-2254

Incompatibilities

One needs to recompile the IDL files to get the inherited interfaces correctly in the IFR register/unregister functions.

Known bugs and problems

ORB

- The dynamic interfaces are not supported and won't be in the first release of Orber.
- Orber only support the persistent object startup behaviour.
- There are a number of function in the boa and corba interfaces that not are implemented but they are mostly used when implementing the ORB and in the stubs and skeletons generated by the IDL compiler and not used by application designers.

IDL compiler

- Defining interface repository identifiers by the use of compiler pragmas is not supported. The `ID`, `version` or `prefix` compiler pragmas are not supported. This is an add on to the standard.
- No checks are made to ensure reference integrity. IDL specifies that identifiers must have one and only one meaning in each scope.
- Files are not closed properly when the compiler has detected errors. This may result in an `emfiles` error code from the Erlang runtime system when the maximum number of open files have been exceeded. The solution is to restart the Erlang emulator when the file error occurs.

- If inline enumerator discriminator types are used, then the name of the enumeration is on the same scope as the name of the union type. This does not apply to the case where the discriminator type is written using a type reference.
- The IFR registration of interface operations does not register any raised exceptions.
- When running the type code registration functions (OE_register) for the IFR and have included files the specifications must be registered in the correct order. There is for the moment no check if that have been done which can give some bad registrations, but an unregistered followed by a register of the superior specification will solve it.

Interface Repository

- The Interface Repository cannot be used from another ORB for the moment.

Resolving initial reference from C++ The returned IOR is correct and the same as for the java implementation but we have for the moment just tested with a client implemented in C++, ie an Orbix C++ client accessing an Orber server.

Orber 1.0.2, Release Notes

Fixed bugs and malfunctions

- The idl compiler generated wrong type registration code for the IFR when an IDL specification included another IDL specification. One could get exceptions from the IFR for trying to double register something (for example a module or interface).
  Own Id: OTP-2133
- Two type errors in internal IDL specified interfaces corrected.
  Own Id: OTP-2121, OTP-2122
- object:get_interface/1 didn't work properly.
  Own Id: OTP-2025
- IDL compiler: Error in handle call code generation in server stub. The compiler stopped generating handle_call clauses when there was a ONEWAY function. In the example below there was no code generated for the function h. If the oneway functions were last in the interface definition all worked fine.

  ```
  interface i {
     short f();
     oneway void g(in char c);
     long h();
  }
  ```

  Own Id: OTP-2057
- Badly choosen module name in the IDL example file InitialReferences.idl, the module name is changed from ORBER to Orber.
  Own Id: OTP-2069
- Documentation error in the description of the IDL mapping to Erlang. The example in chapter 2.7 was wrong.
  Own Id: OTP-2108

- pull() function in ProxyPullSupplier interface had a wrong return vaue of {Value, BOOL} instead of Value.
  Own Id: OTP-2150
- 'Disconnected' exceptions were missing from calls to ProxyPullSupplier:pull(), ProxyPullSupplier:try_pull() and ProxyPushConsumer:push(). This exception should be thrown in case if communication has been disconnected.
  Own Id: OTP-2151

Incompatibilities

One needs to recompile the IDL files to get the corrections in some cases.

There are one incompatibility, the package name for the Java InitialReferences class has been changed. see bugfix id OTP-2069 above.

Known bugs and problems

ORB

- The dynamic interfaces are not supported and won't be in the first release of Orber.
- Orber only support the persistent object startup behaviour.
- There are a number of function in the boa and corba interfaces that not are implemented but they are mostly used when implementing the ORB and in the stubs and skeletons generated by the IDL compiler and not used by application designers.

IDL compiler

- Defining interface repository identifiers by the use of compiler pragmas is not supported. The `ID`, `version` or `prefix` compiler pragmas are not supported. This is an add on to the standard.
- No checks are made to ensure reference integrity. IDL specifies that identifiers must have one and only one meaning in each scope.
- Files are not closed properly when the compiler has detected errors. This may result in an `emfiles` error code from the Erlang runtime system when the maximum number of open files have been exceeded. The solution is to restart the Erlang emulator when the file error occurs.
- If inline enumerator discriminator types are used, then the name of the enumeration is on the same scope as the name of the union type. This does not apply to the case where the discriminator type is written using a type reference.
- The IFR registration of interface operations does not registerany raised exceptions.
- When running the type code registration funcctions (OE_register) for the IFR and have included files the specifications must be registered in the correct order. There is for the moment no check if that have been done which can give some bad registrations, but an unregistered followed by a register of the superior specification will solve it.

Interface Repository

- The Interface Repository cannot be used from another ORB for the moment.

Resolving initial reference from C++ The returned IOR is correct and the same as for the java implementation but we have for the moment just tested with a client implemented in C++, ie an Orbix C++ client accessing an Orber server.

Orber 1.0.1, Release Notes

Fixed bugs and malfunctions

- Default count in the Type Kind structs where always -1.
  Own Id: OTP-2007
- CosNaming::NamingContext::list() returned wrong return value and bad format of out parameters.
  Own Id: OTP-2023
- corba::string_to_object previously returned an internal structure. This has been remedied and the function now returns an object reference.
  Own Id: OTP-2024

Orber 1.0, Release Notes

Improvements and new features

Orber is a new application which allows OTP applications to interact with other programs written in other languages through the CORBA standard.

The orber release contains the following parts:

- `Orb kernel and IIOP support`
- `IDL compiler`
- `Interface Repository`
- `Orber CosNaming Service`
- `Orber CosEvent Service (only untyped events)`
- `Resolving initial reference from Java`
- `Resolving initial reference from C++`
- `A small example`

Implemented work packages are: OTP-1508, OTP-1509 (not typed event).

Orb kernel and IIOP support There is an ORB kernel with IIOP support which allows creating persistent server objects in Erlang and access them from Erlang and java. For the moment one need a java enabled Orb to generate java from idl and use java server objects (we have tested with OrbixWeb).

IDL compiler The IDL compiler generates server behaviours and client stubs according to the IDL to Erlang mapping. Interface inheritance is supported. The idl compiler *requires gcc* because it's used as preprocessor. (It's possible to run the compiler without preprocessor if for example you don't use include statements)

Interface Repository The Interface Repository (IFR) is fully implemented. The module `orber_ifr` is the interface to it. The IFR is used for some type checking when coding/decoding IIOP and therefore all interfaces must be registered in the IFR.

Orber CosNaming service This is the first version of the CosNaming compliant service which also includes two modules `lname` and `lname_component` which supports the naming library interface in erlang.

Orber CosEvent Service Orber contains an Event Service that is compliant with the untyped part of the CosEvent service specification.

Resolving initial reference from Java A class with just 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 just one method which returns an IOR on the external string format to the INIT object (see “Interoperable Naming Service” specification).

A small example A small programming example is contributed which shows how Orber can be used. It is an implementation of a Stack service which shows how Erlang services can be accessed from both Erlang and java.

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

General

- Operation attribute `oneway` is implemented but not tested.

ORB

- The dynamic interfaces are not supported and won’t be in the first release of Orber.
- Orber only support the persistent object startup behaviour.
- There are a number of function in the boa and corba interfaces that not are implemented but they are mostly used when implementing the ORB and in the stubs and skeletons generated by the IDL compiler and not used by application designers.

IDL compiler

- Defining interface repository identifiers by the use of compiler pragmas is not supported. The `ID`, `version` or `prefix` compiler pragmas are not supported. This is an add on to the standard.
- No checks are made to ensure reference integrity. IDL specifies that identifiers must have one and only one meaning in each scope.
- Files are not closed properly when the compiler has detected errors. This may result in an `emfiles` error code from the Erlang runtime system when the maximum number of open files have been exceeded. The solution is to restart the Erlang emulator when the file error occurs.
- If inline enumerator discriminator types are used, then the name of the enumeration is on the same scope as the name of the union type. This does not apply to the case where the discriminator type is written using a type reference.
- The IFR registration of interface operations does not registerany raised exceptions.

Interface Repository

- The Interface Repository cannot be used from another ORB for the moment.

Resolving initial reference from C++ The returned IOR is correct and the same as for the java implementation but we have for the moment just tested with a client implemented in C++, ie an Orbix C++ client accessing an Orber server.

Orber Reference Manual

Short Summaries

- Erlang Module CosNaming [page 83] – The CosNaming service is a collection of interfaces that together define the naming service.
- Erlang Module CosNaming_BindingIterator [page 86] – This interface supports iteration over a name binding list.
- Erlang Module CosNaming_NamingContext [page 88] – This interface supports different bind and access functions for names in a context.
- Erlang Module CosNaming_NamingContextExt [page 91] – This interface contains operation for converting a Name sequence to a string and back.
- Erlang Module Module_Interface [page 93] – Orber generated stubs/skeletons.
- Erlang Module any [page 99] – the corba any type
- Erlang Module corba [page 101] – The functions on CORBA module level
- Erlang Module corba_object [page 105] – The Corba Object interface functions
- Erlang Module lname [page 107] – Interface that supports the name pseudo-objects.
- Erlang Module lname_component [page 109] – Interface that supports the name pseudo-objects.
- Erlang Module orber [page 111] – The main module of the Orber application
- Erlang Module orber_ifr [page 117] – The Interface Repository stores representations of IDL information
- Erlang Module orber_tc [page 131] – help functions for IDL typecodes

CosNaming

No functions are exported.

CosNaming_BindingIterator

The following functions are exported:

- next_one(BindinIterator) -> Return
  [page 86] Returns a binding
- next_n(BindinIterator, HowMany) -> Return
  [page 86] Returns a binding list
- destroy(BindingIterator) -> Return
  [page 86] destroys the iterator object

CosNaming_NamingContext

The following functions are exported:

- bind(NamingContext, Name, Object) -> Return
  [page 89] Bind a Name to an Object
- rebind(NamingContext, Name, Object) -> Return
  [page 89] Bind an Object to the Name even if the Name already is bound
- bind_context(NamingContext1, Name, NamingContex2) -> Return
  [page 89] Bind a Name to an NamingContext
- rebind_context(NamingContext1, Name, NamingContex2) -> Return
  [page 89] Bind an NamingContext to the Name even if the Name already is bound
- resolve(NamingContext, Name) -> Return
  [page 89] Retrieve an Object bound to Name
- unbind(NamingContext, Name) -> Return
  [page 90] Remove the binding for a Name
- new_context(NamingContext) -> Return
  [page 90] Create a new NamingContext
- bind_new_context(NamingContext, Name) -> Return
  [page 90] Create a new NamingContext and bind it to a Name
- destroy(NamingContext) -> Return
  [page 90] Destroy a NamingContext
- list(NamingContext, HowMany) -> Return
  [page 90] List returns a all bindings in the context

CosNaming_NamingContextExt

The following functions are exported:

- to_string(NamingContext, Name) -> Return
  [page 91]
- to_name(NamingContext, NameString) -> Return
  [page 91]
- to_url(NamingContext, AddressString, NameString) -> Return
  [page 91]
- resolve_str(NamingContext, NameString) -> Return
  [page 91]

Module_Interface

The following functions are exported:

- `typeID() -> TypeId`
  [page 94] Returns the Type ID related to this stub/skeleton
- `oe_create() -> ObjRef`
  [page 94] Start a Orber server.
- `oe_create_link() -> ObjRef`
  [page 94] Start a linked Orber server.
- `oe_create(Env) -> ObjRef`
  [page 94] Start a Orber server.
- `oe_create_link(Env) -> ObjRef`
  [page 94] Start a linked Orber server.
- `oe_create(Env, Options) -> ObjRef`
  [page 94] Start a Orber stub/skeleton
- `oe_create_link(Env, Options) -> Return`
  [page 95] Start a Orber stub/skeleton
- `Module_Interface:own_functions(ObjRef, Arg1, ..., ArgN) -> Reply`
  [page 95]
- `Module_Interface:own_functions(ObjRef, Timeout, Arg1, ..., ArgN) -> Reply`
  [page 95]
- `Module_Interface_impl:init(Env) -> CallReply`
  [page 96]
- `Module_Interface_impl:terminate(Reason, State) -> ok`
  [page 96]
- `Module_Interface_impl:code_change(OldVsn, State, Extra) -> CallReply`
  [page 96] Update the internal `State`.
- `Module_Interface_impl:handle_info(Info, State) -> CallReply`
  [page 96]
- `Module_Interface_impl:own_functions(This, State, Arg1, ..., ArgN) -> CallReply`
  [page 97]
- `Module_Interface_impl:own_functions(State, Arg1, ..., ArgN) -> CallReply`
  [page 97]
- `Module_Interface_impl:own_functions(This, State, Arg1, ..., ArgN) -> CallReply`
  [page 97]
- `Module_Interface_impl:own_functions(State, Arg1, ..., ArgN) -> CallReply`
  [page 97]

any

The following functions are exported:

- `create() -> Result`
  [page 99] creates an any record
- `create(Typecode, Value) -> Result`
  [page 99] creates an any record
- `set_typecode(A, Typecode) -> Result`
  [page 99] sets the typecode field
- `get_typecode(A) -> Result`
  [page 99] fetches the typecode
- `set_value(A, Value) -> Result`
  [page 100] sets the value field
- `get_value(A) -> Result`
  [page 100] fetches the value

corba

The following functions are exported:

- `create(Module, TypeID) -> Object`
  [page 101] create and start a new server object
- `create(Module, TypeID, Env) -> Object`
  [page 101] create and start a new server object
- `create(Module, TypeID, Env, Optons1) -> Object`
  [page 101] create and start a new server object
- `create_link(Module, TypeID) -> Object`
  [page 101] create and start a new server object
- `create_link(Module, TypeID, Env) -> Object`
  [page 101] create and start a new server object
- `create_link(Module, TypeID, Env, Options2) -> Reply`
  [page 101] create and start a new server object
- `dispose(Object) -> ok`
  [page 102] stops a server object
- `create_subobject_key(Object, Key) -> Result`
  [page 102] adds an Erlang term to a private key field
- `get_subobject_key(Object) -> Result`
  [page 102] fetch the contents of the private key field
- `get_pid(Object) -> Result`
  [page 103] get the process id from an object key
- `raise(Exception)`
  [page 103] generates an Erlang throw
- `resolve_initial_references(ObjectId) -> Object`
  [page 103] returns the object reference for the given object id

- `list_initial_services() -> [ObjectId]`
  [page 103] returns a list of supported object id's
- `resolve_initial_references_remote(ObjectId, Address) -> Object`
  [page 103] returns the object reference for the given object id
- `list_initial_services_remote(Address) -> [ObjectId]`
  [page 103] returns a list of supported object id's
- `object_to_string(Object) -> IOR_string`
  [page 104] converts the object reference to the external string representation
- `string_to_object(IOR_string) -> Object`
  [page 104] converts the external string representation to an object reference

corba_object

The following functions are exported:

- `get_interface(Object) -> InterfaceDef`
  [page 105] Fetch the interface description
- `is_nil(Object) -> boolean()`
  [page 105]
- `is_a(Object, Logical_type_id) -> Return`
  [page 105]
- `is_remote(Object) -> boolean()`
  [page 105] Determines whether or not an object reference is remote.
- `non_existent(Object) -> Return`
  [page 106]
- `not_existent(Object) -> Return`
  [page 106]
- `is_equivalent(Object, OtherObject) -> boolean()`
  [page 106]
- `hash(Object, Maximum) -> int()`
  [page 106]

Iname

The following functions are exported:

- `create() -> Return`
  [page 107] creates a new name
- `insert_component(Name, N, NameComponent) -> Return`
  [page 107] inserts a new name component in a name
- `get_component(Name, N) -> Return`
  [page 107] get a name component from a name
- `delete_component(Name, N) -> Return`
  [page 108] deletes s name component from a name

- num_components(Name) -> Return
  [page 108] counts the number of name components in a name
- equal(Name1, Name2) -> Return
  [page 108] tests if two names are equal
- less_than(Name1, Name2) -> Return
  [page 108] tests if one name is lesser than the other
- to_idl_form(Name) -> Return
  [page 108] transforms a pseudo name to an IDL name
- from_idl_form(Name) -> Return
  [page 108] transforms an IDL name to a pseudo name

lname_component

The following functions are exported:

- create() -> Return
  [page 109] creates a new name component
- get_id(NameComponent) -> Return
  [page 109] get the id field of a name component
- set_id(NameComponent, Id) -> Return
  [page 109] set the id field of a name component
- get_kind(NameComponent) -> Return
  [page 109] get the kind field of a name component
- set_kind(NameComponent, Kind) -> Return
  [page 110] set the kind field of a name component

orber

The following functions are exported:

- start() -> ok
  [page 111] Start the Orber application
- start_lightweight() -> ok
  [page 111] Start the Orber application as lightweight
- start_lightweight(Addresses) -> ok
  [page 111] Start the Orber application as lightweight
- stop() -> ok
  [page 111] Stops the Orber application
- is_lightweight() -> boolean()
  [page 112] Is the application started as lightweight?
- get_lightweight_nodes() -> RemoteModifierList | false
  [page 112] Get the Remote Modifier list.
- get_ORBInitRef() -> string() | undefined
  [page 112] Get the initial reference address.

- `get_ORBDefaultInitRef() -> string() | undefined`
  [page 112] Get the initial reference address.
- `domain() -> string()`
  [page 112] Display the Orber domain name
- `iiop_port() -> int()`
  [page 112] Display the IIOP port number
- `iiop_ssl_port() -> int()`
  [page 112] Display the IIOP port number used for secure connections
- `iiop_timeout() -> int() (milliseconds)`
  [page 112] Display the IIOP timeout value
- `iiop_connection_timeout() -> int() (milliseconds)`
  [page 113] Display the IIOP connection timeout value
- `secure() -> no | ssl`
  [page 113] Display the security mode Orber is running in
- `ssl_server_certfile() -> string()`
  [page 113] Display the path to the server certificate
- `ssl_client_certfile() -> string()`
  [page 113] Display the path to the client certificate
- `set_ssl_client_certfile(Path) -> ok`
  [page 113] Sets the value of the client certificate
- `ssl_server_verify() -> 0 | 1 | 2`
  [page 113] Display the SSL verification type for incoming calls
- `ssl_client_verify() -> 0 | 1 | 2`
  [page 113] Display the SSL verification type for outgoing calls
- `set_ssl_client_verify(Value) -> ok`
  [page 113] Sets the value of the SSL verification type for outgoing calls
- `ssl_server_depth() -> int()`
  [page 114] Display the SSL verification depth for incoming calls
- `ssl_client_depth() -> int()`
  [page 114] Display the SSL verification depth for outgoing calls
- `set_ssl_client_depth(Depth) -> ok`
  [page 114] Sets the value of the SSL verification depth for outgoing calls
- `objectkeys_gc_time() -> int() (seconds)`
  [page 114] Display the Object Keys GC time value
- `bootstrap_port() -> int()`
  [page 114] Display the bootstrap protocol port number
- `orber_nodes() -> RetVal`
  [page 114] Displays which nodes that this orber domain consist of.
- `install(NodeList) -> ok`
  [page 114] Installs the Orber application
- `install(NodeList, Options) -> ok`
  [page 114] Installs the Orber application
- `uninstall() -> ok`
  [page 115] Uninstall the Orber application
- `add_node(Node, StorageType) -> RetVal`
  [page 115] Adds a new node to a group of Orber nodes.
- `remove_node(Node) -> RetVal`
  [page 116] Removes a node from a group of Orber nodes.

orber_ifr

The following functions are exported:

- `init(Nodes,Timeout) -> ok`
  [page 117] Intialize the IFR
- `find_repository() -> #IFR_Repository_objref`
  [page 117]
- `get_def_kind(Objref) -> Return`
  [page 118]
- `destroy(Objref) -> Return`
  [page 118]
- `get_id(Objref) -> Return`
  [page 118]
- `set_id(Objref,Id) -> ok`
  [page 118]
- `get_name(Objref) -> Return`
  [page 118]
- `set_name(Objref,Name) -> ok`
  [page 118]
- `get_version(Objref) -> Return`
  [page 119]
- `set_version(Objref,Version) -> ok`
  [page 119]
- `get_defined_in(Objref) -> Return`
  [page 119]
- `get_absolute_name(Objref) -> Return`
  [page 119]
- `get_containing_repository(Objref) -> Return`
  [page 119]
- `describe(Objref) -> Return`
  [page 119]
- `move(Objref,New_container,New_name,New_version) -> Return`
  [page 120]
- `lookup(Objref,Search_name) -> Return`
  [page 120]
- `contents(Objref,Limit_type,Exclude_inherited) -> Return`
  [page 120]
- `lookup_name(Objref,Search_name,Levels_to_search, Limit_type, Exclude_inherited) -> Return`
  [page 120]
- `describe_contents(Objref,Limit_type,Exclude_inherited,Max_returned_objs) -> Return`
  [page 121]
- `create_module(Objref,Id,Name,Version) -> Return`
  [page 121]

- is_a(Objref,Interface_id) -> Return
  [page 129]
- describe_interface(Objref) -> Return
  [page 130]
- create_attribute(Objref,Id,Name,Version,Type,Mode) -> Return
  [page 130]
- create_operation(Objref,Id,Name,Version,Result,Mode,Params,
  Exceptions,Contexts) -> Return
  [page 130]

orber_tc

The following functions are exported:

- null() -> TC
  [page 131] get the IDL typecode
- void() -> TC
  [page 131] get the IDL typecode
- short() -> TC
  [page 131] get the IDL typecode
- unsigned_short() -> TC
  [page 131] get the IDL typecode
- long() -> TC
  [page 131] get the IDL typecode
- unsigned_long() -> TC
  [page 131] get the IDL typecode
- float() -> TC
  [page 131] get the IDL typecode
- double() -> TC
  [page 131] get the IDL typecode
- boolean() -> TC
  [page 131] get the IDL typecode
- char() -> TC
  [page 131] get the IDL typecode
- octet() -> TC
  [page 131] get the IDL typecode
- any() -> TC
  [page 131] get the IDL typecode
- typecode() -> TC
  [page 131] get the IDL typecode
- principal() -> TC
  [page 131] get the IDL typecode
- object_reference(Id, Name) -> TC
  [page 131] the object_reference IDL typecode
- struct(Id, Name, ElementList) -> TC
  [page 131] the struct IDL typecode

- `union(Id, Name, DiscrTC, Default, ElementList) -> TC`
  [page 132] the union IDL typecode
- `enum(Id, Name, ElementList) -> TC`
  [page 132] the enum IDL typecode
- `string(Length) -> TC`
  [page 132] the string IDL typecode
- `sequence(ElemTC, Length) -> TC`
  [page 133] the sequence IDL typecode
- `array(ElemTC, Length) -> TC`
  [page 133] the array IDL typecode
- `alias(Id, Name, AliasTC) -> TC`
  [page 133] the alias IDL typecode
- `exception(Id, Name, ElementList) -> TC`
  [page 133] the exception IDL typecode
- `get_tc(Object) -> TC`
  [page 133] fetch typecode
- `get_tc(Id) -> TC`
  [page 133] fetch typecode
- `check(TC) -> boolean()`
  [page 134] syntax check of an IDL typecode

CosNaming (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 (Module)

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

The type `NameComponent` used below is defined as:

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

`id` and `kind` are strings.

The type `Binding` used below is defined as:

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

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

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

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

Exports

`next_one(BindinIterator) -> Return`

Types:

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

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

`next_n(BindinIterator, HowMany) -> Return`

Types:

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

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

`destroy(BindingIterator) -> Return`

Types:

- BindingIterator = #objref
- Return = ok

This operation destroys the binding iterator.

CosNaming_NamingContext (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` ia 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 upto HowMany from the context. It also returns a BindinIterator which can be used to step through the list.

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

CosNaming_NamingContextExt (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 88]

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 (Module)

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

Starting a Orber server can be done in three ways:

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

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

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

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

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

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

Exports

`typeID() -> TypeId`

Types:

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

Returns the Type ID related to this stub/skeleton

`oe_create() -> ObjRef`

Types:

- ObjRef = #object reference

Start a Orber server.

`oe_create_link() -> ObjRef`

Types:

- ObjRef = #object reference

Start a linked Orber server.

`oe_create(Env) -> ObjRef`

Types:

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

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

`oe_create_link(Env) -> ObjRef`

Types:

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

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

`oe_create(Env, Options) -> ObjRef`

Types:

- Env = term()

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

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

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

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

`oe_create_link(Env, Options) -> Return`

Types:

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

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

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

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

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

Types:

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

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

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

Types:

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

- Reply = specified in the IDL-code.

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

`Module_Interface_impl:init(Env) -> CallReply`

Types:

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

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

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

Types:

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

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

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

Types:

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

Update the internal `State`.

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

Types:

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

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

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

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

Types:

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

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

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

Types:

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

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

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

Types:

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

- StopReason = normal | shutdown | term()

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

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

Types:

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

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

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

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

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

Module is the name of the interface API module.

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

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

A server started with create_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 childs they should be *transient* and the *objectkeys_gc_time* should be modified (default equals infinity).

Example:

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

`dispose(Object) -> ok`

Types:

- Object = #objref

This function is used for terminating the execution of a server object.

`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*}.

`resolve_initial_references(ObjectId) -> Object`

Types:

- ObjectId = string()
- Object = #objref

This function returns the object reference for the object id asked for (just now only the "NameService").

`list_initial_services() -> [ObjectId]`

Types:

- ObjectId = string()

This function returns a list of allowed object id's (just now only the "NameService").

`resolve_initial_references_remote(ObjectId, Address) -> Object`

Types:

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

This function returns the object reference for the object id asked for (depends on the orb, for orber it is just the "NameService"). The remote modifier string has the following format: `"iiop://host:port"`.

`list_initial_services_remote(Address) -> [ObjectId]`

Types:

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

This function returns a list of allowed object id's (depends on the orb, for orber it is just the "NameService"). The remote modifier string has the following format: `"iiop://host:port"`.

`object_to_string(Object) -> IOR_string`

Types:

- Object = #objref
- IOR_string = string()

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

`string_to_object(IOR_string) -> Object`

Types:

- IOR_string = string()
- Object = #objref

This function takes an IOR on the external string representation and returns the object reference.

corba_object (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 test and no object implementation is involved in the test.

`is_a(Object, Logical_type_id) -> Return`

Types:

- Object = #objref
- Logical_type_id = string()

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

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

`is_remote(Object) -> boolean()`

Types:

- Object = #objref

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

`non_existent(Object) -> Return`

Types:

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

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

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

`not_existent(Object) -> Return`

Types:

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

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

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

Types:

- Object = #objref
- OtherObject = #objref

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

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

Types:

- Object = #objref
- Maximum = int()

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

lname (Module)

This interface is a part of the names library which is used to hide the representation of names. In orbers Erlang mapping the pseodo-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 compnent 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 (Module)

This interface is a part of the name library, which is used to hide the representation of names. In orbers Erlang mapping the pseodo-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 (Module)

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

Exports

`start() -> ok`

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

`start_lightweight() -> ok`

Starts the Orber application as lightweight.

Preconditions:

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

`start_lightweight(Addresses) -> ok`

Types:

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

Starts the Orber application as lightweight.

Preconditions:

- If Erlang is started using the configuration parameter -orber lightweight, e.g., erl -orber lightweight Address, the argument supplied to this function will override the configuration parameter. Hence, this function must be used carefully.
- The `Addresses` must be a list of `RemoteModifiers`, equal to the `orber:resolve_initial_references_remote/2` argument. The list must contain Orber nodes addresses, to which we have access and are not started as lightweight.

`stop() -> ok`

Stops the Orber application.

`is_lightweight() -> boolean()`

This function returns the true if Orber is started as lightweight, false otherwise.

`get_lightweight_nodes() -> RemoteModifierList | false`

This function returns false if Orber is not started as lightweight, otherwise a list of Remote Modifiers.

`get_ORBInitRef() -> string() | undefined`

This function returns undefined if we will resolve references locally, otherwise a string describing which host we will contact if the Key given to `corba:resolve_initial_references/1` matches the Key set in this configuration variable. For more information see the user's guide.

`get_ORBDefaultInitRef() -> string() | undefined`

This function returns undefined if we will resolve references locally, otherwise a string describing which host, or hosts, from which we will try to resolve the Key given to `corba:resolve_initial_references/1`. For more information see the user's guide.

`domain() -> string()`

This function returns the domain name of the current Orber domain as a string.

`iiop_port() -> int()`

This function returns the port-number, which is used by the IIOP protocol. It can be configured by setting the application variable *iiop_port*, if it is not set it will have the default number 4001.

`iiop_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 *COMM_FAILURE* exception is raised.

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

Note: IC supply the compile option `ic:gen(IdlFile, [{timeout,"module::interface"}])`, which allow the user to add an extra timeout parameter, e.g., `module_interface(ObjRef, Timeout, ... Arguments ...)`, instead of `module_interface(ObjRef, ... Arguments ...)`. If, a stub is compiled with the timeout option, the extra Timeout argument will override the configuration

parameter `iiop_timeout`. It is, however, not possible to use `infinity` to override the Timeout parameter. The Timeout option is also valid for objects which resides within the same Orber domain.

`iiop_connection_timeout() -> int() (milliseconds)`

This function returns the timeout value after which outgoing IIOP connections terminate. It can be configured by setting the application variable *iiop_connection_timeout TimeVal (seconds)*, if it is not set it will have the default value *infinity*. The connection will not be terminated if there are pending requests.

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

`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.

`bootstrap_port() -> int()`

This function returns the port-number, which is used by the CORBA bootstrapping protocol. This protocol is used to fetch an initial reference from another ORB. It can be configured by setting the application variable *bootstrap_port*, if it is not set it will use the iiop port.

Note: In the future it will use the port number which is set in the standard (the suggestion is 900). Because the standard is not ready in this area we in the meantime uses a port number, which do not require root permissions in Unix.

`orber_nodes() -> RetVal`

Types:

- RetVal = [node()]

This function returns the list of node names that this orber domain consists of.

`install(NodeList) -> ok`
`install(NodeList, Options) -> ok`

Types:

- NodeList = [node()]
- Options = [Option]
- Option = {install_timeout, Timeout} | {ifr_storage_type, TableType}

- Timeout = infinity | integer()
- TableType = disc_copies | ram_copies

This function installs all the necessary mnesia tables and load default data in some of them. If one or more Orber tables already exists the installation fails. The function *uninstall* may be used, if it is safe, i.e., no other application is running Orber.

Preconditions:

- a mnesia schema must exist before the installation
- mnesia is running on the other nodes if the new installation shall be a multi node domain

Mnesia will be started by the function if it is not already running on the installation node and if it was started it will be stopped afterwards.

The options that can be sent to the installation program is:

- {`install_timeout, Timeout`} - this timeout is how long we will wait for the tables to be created. The Timeout value can be *infinity* or an integer number in milliseconds. Default is infinity.
- {`ifr_storage_type, TableType`} - this option sets the type of tables used for the interface repository. The TableType can be disc_copies or ram_copies. Default is disc_copies. (All other tables in Orber are ram copies).

`uninstall() -> ok`

This function stops the Orber application, terminates all server objects and removes all Orber related mnesia tables.

Note: Since other applications may be running on the same node using mnesia *uninstall* will not stop the mnesia application.

`add_node(Node, StorageType) -> RetVal`

Types:

- Node = node()
- StorageType = disc_copies | ram_copies
- RetVal = ok | exit()

This function add given node to a existing Orber node group and starts Orber on the new node. `orber:add_node` is called from a member in the Orber node group.

Preconditions for new node:

- Erlang started on the new node using the option `-mnesia extra_db_nodes`, e.g., `erl -sname new_node_name -mnesia extra_db_nodes ConnectToNodes_List`
- 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.

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.

orber_ifr (Module)

This module contains functions for managing the Interface Repository (IFR). This documentation should be used in conjunction with the documentation in chapter 6 of *CORBA* 2.0. Whenever the term IFR object is used in this manual page, it refers to a pseudo object used only for interaction with the IFR rather than a CORBA object.

Initialisation of the IFR

The following functions are used to initialise the Interface Repository and to obtain the initial reference to the repository.

Exports

`init(Nodes,Timeout) -> ok`

Types:

- Nodes = list()
- Timeout = integer() | infinity

This function should be called to initialise the IFR. It creates the necessary mnesia-tables. A mnesia schema should exist, and mnesia must be running.

`find_repository() -> #IFR_Repository_objref`

Find the IFR object reference for the Repository. This reference should be used when adding objects to the IFR, and when extracting information from the IFR. The first time this function is called, it will create the repository and all the primitive definitions.

General methods

The following functions are the methods of the IFR. The first argument is always an #IFR_objref, i.e. the IFR (pseudo)object on which to apply this method. These functions are useful when the type of IFR object is not 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_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_Sequence, dk_Array, dk_Repository)
- Exclude_inherited = atom() (true or false)
- Return = list() (a list of IFR#_objects)

Objref is an IFR object of any kind that inherits from Container. Returns the contents of that IFR object.

`lookup_name(Objref,Search_name,Levels_to_search, Limit_type, Exclude_inherited) -> Return`

Types:

- Objref = #IFR_objref
- Search_name = string()
- Levels_to_search = integer()
- Limit_type = atom() (one of dk_none, dk_all, dk_Attribute, dk_Constant, dk_Exception, dk_Interface, dk_Module, dk_Operation, dk_Typedef, dk_Alias, dk_Struct, dk_Union, dk_Enum, dk_Primitive, dk_String, dk_Sequence, dk_Array, dk_Repository)
- Exclude_inherited = atom() (true or false)

- Return = list() (a list of #IFR_objects)

Objref is an IFR object of any kind that inherits from Container. Returns a list of #IFR_objects with an id matching Search_name.

`describe_contents(Objref,Limit_type,Exclude_inherited,Max_returned_objs) -> Return`

Types:

- Objref = #IFR_objref
- Limit_type = atom() (one of dk_none, dk_all, dk_Attribute, dk_Constant, dk_Exception, dk_Interface, dk_Module, dk_Operation, dk_Typedef, dk_Alias, dk_Struct, dk_Union, dk_Enum, dk_Primitive, dk_String, dk_Sequence, dk_Array, dk_Repository)
- Exclude_inherited = atom() (true or false)
- Return = list() (a list of tuples (contained_description records) | {exception, _}

Objref is an IFR object of any kind that inherits from Container. Returns a list of descriptions of the IFR objects in this Container's contents.

`create_module(Objref,Id,Name,Version) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Return = #IFR_ModuleDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type ModuleDef.

`create_constant(Objref,Id,Name,Version,Type,Value) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Type = #IFR_IDLType_objref
- Value = any()
- Return = #IFR_ConstantDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type ConstantDef.

`create_struct(Objref,Id,Name,Version,Members) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Members = list() (list of structmember records)

- Return = #IFR_StructDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type StructDef.

`create_union(Objref,Id,Name,Version,Discriminator_type,Members) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Discriminator_type = #IFR_IDLType_Objref
- Members = list() (list of unionmember records)
- Return = #IFR_UnionDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type UnionDef.

`create_enum(Objref,Id,Name,Version,Members) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Members = list() (list of strings)
- Return = #IFR_EnumDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type EnumDef.

`create_alias(Objref,Id,Name,Version,Original_type) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Original_type = #IFR_IDLType_Objref
- Return = #IFR_AliasDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type AliasDef.

`create_interface(Objref,Id,Name,Version,Base_interfaces) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()

- Base_interfaces = list() (a list of IFR_InterfaceDef_objrefs that this interface inherits from
- Return = #IFR_InterfaceDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type InterfaceDef.

`create_exception(Objref,Id,Name,Version,Members) -> Return`

Types:

- Objref = #IFR_objref
- Id = string()
- Name = string()
- Version = string()
- Members = list() (list of structmember records)
- Return = #IFR_ExceptionDef_objref

Objref is an IFR object of any kind that inherits from Container. Creates an IFR object of the type ExceptionDef.

`get_type(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = tuple() (a typecode tuple)

Objref is an IFR object of any kind that inherits from IDLType or an IFR object of the kind ConstantDef, ExceptionDef or AttributeDef. Returns the typecode of the IFR object.

`lookup_id(Objref,Search_id) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Search_id = string()
- Return = #IFR_objref

Returns an IFR object matching the Search_id.

`get_primitive(Objref,Kind) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Kind = atom() (one of pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong, pk_float, pk_double, pk_boolean, pk_char, pk_octet, pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref)
- Return = #IFR_PrimitiveDef_objref

Returns a PrimitiveDef of the specified kind.

`create_string(Objref,Bound) -> Return`

Types:

- Objref = #IFR_Repository_objref

- Bound = integer() (unsigned long /= 0)
- Return = #IFR_StringDef_objref

Creates an IFR objref of the type StringDef.

`create_sequence(Objref,Bound,Element_type) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Bound = integer() (unsigned long)
- Element_type = #IFR_IDLType_objref
- Return = #IFR_SequenceDef_objref

Creates an IFR objref of the type SequenceDef.

`create_array(Objref,Length,Element_type) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Bound = integer() (unsigned long)
- Element_type = #IFR_IDLType_objref
- Return = #IFR_ArrayDef_objref

Creates an IFR objref of the type ArrayDef.

`create_idltype(Objref,Typecode) -> Return`

Types:

- Objref = #IFR_Repository_objref
- Typecode = tuple() (a typecode tuple)
- Return = #IFR_IDLType_objref

Creates an IFR objref of the type IDLType.

`get_type_def(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = #IFR_IDLType_objref

Objref is an IFR object of the kind ConstantDef or AttributeDef. Returns an IFR object of the type IDLType describing the type of the IFR object.

`set_type_def(Objref,TypeDef) -> Return`

Types:

- Objref = #IFR_objref
- TypeDef = #IFR_IDLType_objref
- Return = ok | {exception, _}

Objref is an IFR object of the kind ConstantDef or AttributeDef. Sets the type_def of the IFR Object.

`get_value(Objref) -> Return`

Types:

- Objref = #IFR_ConstantDef_objref
- Return = any()

Returns the value attribute of an IFR Object of the type ConstantDef.

`set_value(Objref,Value) -> Return`

Types:

- Objref = #IFR_ConstantDef_objref
- Value = any()
- Return = ok | {exception, _}

Sets the value attribute of an IFR Object of the type ConstantDef.

`get_members(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = list()

Objref is an IFR object the kind StructDef, UnionDef, EnumDef or ExceptionDef. For StructDef, UnionDef and ExceptionDef: Returns a list of structmember records that are the constituent parts of the object. For EnumDef: Returns a list of strings describing the enumerations.

`set_members(Objref,Members) -> Return`

Types:

- Objref = #IFR_objref
- Members = list()
- Return = ok | {exception, _}

Objref is an IFR object the kind StructDef, UnionDef, EnumDef or ExceptionDef. For StructDef, UnionDef and ExceptionDef: Members is a list of structmember records. For EnumDef: Members is a list of strings describing the enumerations. Sets the members attribute, which are the constituent parts of the exception.

`get_discriminator_type(Objref) -> Return`

Types:

- Objref = #IFR_UnionDef_objref
- Return = tuple() (a typecode tuple)

Returns the discriminator typecode of an IFR object of the type UnionDef.

`get_discriminator_type_def(Objref) -> Return`

Types:

- Objref = #IFR_UnionDef_objref
- Return = #IFR_IDLType_objref

Returns an IFR object of the type IDLType describing the discriminator type of an IFR object of the type UnionDef.

`set_discriminator_type_def(Objref,TypeDef) -> Return`

Types:

- Objref = #IFR_UnionDef_objref
- Return = #IFR_IDLType_objref

Sets the attribute discriminator_type_def, an IFR object of the type IDLType describing the discriminator type of an IFR object of the type UnionDef.

`get_original_type_def(Objref) -> Return`

Types:

- Objref = #IFR_AliasDef_objref
- Return = #IFR_IDLType_objref

Returns an IFR object of the type IDLType describing the original type.

`set_original_type_def(Objref,TypeDef) -> Return`

Types:

- Objref = #IFR_AliasDef_objref
- Typedef = #IFR_IDLType_objref
- Return = ok | {exception, _}

Sets the original_type_def attribute which describes the original type.

`get_kind(Objref) -> Return`

Types:

- Objref = #IFR_PrimitiveDef_objref
- Return = atom()

Returns an atom describing the primitive type (See CORBA 2.0 p 6-21).

`get_bound(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = integer (unsigned long)

Objref is an IFR object the kind StringDef or SequenceDef. For StringDef: returns the maximum number of characters in the string. For SequenceDef: Returns the maximum number of elements in the sequence. Zero indicates an unbounded sequence.

`set_bound(Objref,Bound) -> Return`

Types:

- Objref = #IFR_objref
- Bound = integer (unsigned long)
- Return = ok | {exception, _}

Objref is an IFR object the kind StringDef or SequenceDef. For StringDef: Sets the maximum number of characters in the string. Bound must not be zero. For SequenceDef: Sets the maximum number of elements in the sequence. Zero indicates an unbounded sequence.

`get_element_type(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = tuple() (a typecode tuple)

Objref is an IFR object the kind SequenceDef or ArrayDef. Returns the typecode of the elements in the IFR object.

`get_element_type_def(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = #IFR_IDLType_objref

Objref is an IFR object the kind SequenceDef or ArrayDef. Returns an IFR object of the type IDLType describing the type of the elements in Objref.

`set_element_type_def(Objref,TypeDef) -> Return`

Types:

- Objref = #IFR_objref
- TypeDef = #IFR_IDLType_objref
- Return = ok | {exception, _}

Objref is an IFR object the kind SequenceDef or ArrayDef. Sets the element_type_def attribute, an IFR object of the type IDLType describing the type of the elements in Objref.

`get_length(Objref) -> Return`

Types:

- Objref = #IFR_ArrayDef_objref
- Return = integer() (unsigned long)

Returns the number of elements in the array.

`set_length(Objref,Length) -> Return`

Types:

- Objref = #IFR_ArrayDef_objref
- Length = integer() (unsigned long)

Sets the number of elements in the array.

`get_mode(Objref) -> Return`

Types:

- Objref = #IFR_objref
- Return = atom()

Objref is an IFR object the kind AttributeDef or OperationDef. For AttributeDef: Return is an atom ('ATTR_NORMAL' or 'ATTR_READONLY') specifying the read/write access for this attribute. For OperationDef: Return is an atom ('OP_NORMAL' or 'OP_ONEWAY') specifying the mode of the operation.

`set_mode(Objref,Mode) -> Return`

Types:

- Objref = #IFR_objref
- Mode = atom()
- Return = ok | {exception, _}

Objref is an IFR object the kind AttributeDef or OperationDef. For AttributeDef: Sets the read/write access for this attribute. Mode is an atom ('ATTR_NORMAL' or 'ATTR_READONLY'). For OperationDef: Sets the mode of the operation. Mode is an atom ('OP_NORMAL' or 'OP_ONEWAY').

`get_result(Objref) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- Return = tuple() (a typecode tuple)

Returns a typecode describing the type of the value returned by the operation.

`get_result_def(Objref) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- Return = #IFR_IDLType_objref

Returns an IFR object of the type IDLType describing the type of the result.

`set_result_def(Objref,ResultDef) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- ResultDef = #IFR_IDLType_objref
- Return = ok | {exception, _}

Sets the type_def attribute, an IFR Object of the type IDLType describing the result.

`get_params(Objref) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- Return = list() (list of parameter description records)

Returns a list of parameter description records, which describes the parameters of the OperationDef.

`set_params(Objref,Params) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- Params = list() (list of parameterdescription records)
- Return = ok | {exception, _}

Sets the params attribute, a list of parameterdescription records.

`get_contexts(Objref) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- Return = list() (list of strings)

Returns a list of context identifiers for the operation.

`set_contexts(Objref,Contexts) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- Contexts = list() (list of strings)
- Return = ok | {exception, _}

Set the context attribute for the operation.

`get_exceptions(Objref) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- Return = list() (list of #IFR_ExceptionDef_objrefs)

Returns a list of exception types that can be raised by this operation.

`set_exceptions(Objref,Exceptions) -> Return`

Types:

- Objref = #IFR_OperationDef_objref
- Exceptions = list() (list of #IFR_ExceptionDef_objrefs)
- Return = ok | {exception, _}

Sets the exceptions attribute for this operation.

`get_base_interfaces(Objref) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref
- Return = list() (list of #IFR_InterfaceDef_objrefs)

Returns a list of InterfaceDefs from which this InterfaceDef inherits.

`set_base_interfaces(Objref,BaseInterfaces) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref
- BaseInterfaces = list() (list of #IFR_InterfaceDef_objrefs)
- Return = ok | {exception, _}

Sets the BaseInterfaces attribute.

`is_a(Objref,Interface_id) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref

- Interface_id = #IFR_InterfaceDef_objref
- Return = atom() (true or false)

Returns true if the InterfaceDef either is identical to or inherits from Interface_id.

`describe_interface(Objref) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref
- Return = tuple() (a fullinterfacedescription record)

Returns a full inter face description record describing the InterfaceDef.

`create_attribute(Objref,Id,Name,Version,Type,Mode) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref
- Id = string()
- Name = string()
- Version = string()
- Type = #IFR_IDLType_objref
- Mode = atom() ('ATTR_NORMAL' or 'ATTR_READONLY')
- Return = #IFR_AttributeDef_objref

Creates an IFR object of the type AttributeDef contained in this InterfaceDef.

`create_operation(Objref,Id,Name,Version,Result,Mode,Params, Exceptions,Contexts) -> Return`

Types:

- Objref = #IFR_InterfaceDef_objref
- Id = string()
- Name = string()
- Version = string()
- Result = #IFR_IDLType_objref
- Mode = atom() ('OP_NORMAL' or 'OP_ONEWAY')
- Params = list() (list of parameterdescription records)
- Exceptions = list() (list of #IFR_ExceptionDef_objrefs)
- Contexts = list() (list of strings)
- Return = #IFR_OperationDef_objref

Creates an IFR object of the type OperationDef contained in this InterfaceDef.

orber_tc (Module)

This module contains some functions that gives support in creating IDL typecodes that can be used in for example the any types typecode field. For the simple types it is meaningless to use this API but the functions exist to get the interface complete.

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

Exports

```
null() -> TC
void() -> TC
short() -> TC
unsigned_short() -> TC
long() -> TC
unsigned_long() -> TC
float() -> TC
double() -> TC
boolean() -> TC
char() -> TC
octet() -> TC
any() -> TC
typecode() -> TC
principal() -> TC
```

These functions return the IDL typecodes for simple types.

```
object_reference(Id, Name) -> TC
```

Types:

- Id = string()
  the repository ID
- Name = string()
  the type name of the object

Function returns the IDL typecode for object_reference.

```
struct(Id, Name, ElementList) -> TC
```

Types:

- Id = string()
  the repository ID
- Name = string()
  the type name of the struct
- ElementList = [{MemberName, TC}]
  a list of the struct elements
- MemberName = string()
  the element name

Function returns the IDL typecode for struct.

`union(Id, Name, DiscrTC, Default, ElementList) -> TC`

Types:

- Id = string()
  the repository ID
- Name = string()
  the type name of the union
- DiscrTC = TC
  the typecode for the unions discriminant
- Default = integer()
  a value that indicates which tuple in the element list that is default (value < 0 means no default)
- ElementList = [{Label, MemberName, TC}]
  a list of the union elements
- Label = term()
  the label value should be of the *DiscrTC* type
- MemberName = string()
  the element name

Function returns the IDL typecode for union.

`enum(Id, Name, ElementList) -> TC`

Types:

- Id = string()
  the repository ID
- Name = string()
  the type name of the enum
- ElementList = [MemberName]
  a list of the enums elements
- MemberName = string()
  the element name

Function returns the IDL typecode for enum.

`string(Length) -> TC`

Types:

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

Function returns the IDL typecode for string.

`sequence(ElemTC, Length) -> TC`

Types:

- ElemTC = TC
  the typecode for the sequence elements
- Length = integer()
  the length of the sequence (0 means unbounded)

Function returns the IDL typecode for sequence.

`array(ElemTC, Length) -> TC`

Types:

- ElemTC = TC
  the typecode for the array elements
- Length = integer()
  the length of the array

Function returns the IDL typecode for array.

`alias(Id, Name, AliasTC) -> TC`

Types:

- Id = string()
  the repository ID
- Name = string()
  the type name of the alias
- AliasTC = TC
  the typecode for the type which the alias refer to

Function returns the IDL typecode for alias.

`exception(Id, Name, ElementList) -> TC`

Types:

- Id = string()
  the repository ID
- Name = string()
  the type name of the exception
- ElementList = [{MemberName, TC}]
  a list of the exception elements
- MemberName = string()
  the element name

Function returns the IDL typecode for exception.

`get_tc(Object) -> TC`
`get_tc(Id) -> TC`

Types:

- Object = record()
  an IDL specified struct, union or exception
- Id = string()
  the repository ID

If the get_tc/1 gets a record that is 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) -> 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 1.

CORBA

A specification of an architecture for a distributed object system

CORBA

A specification of an architecture for a distributed object system

CORBA

Common Object Request Broker Architecture is a common communication standard developed by the OMG (Object Management Group)
Local for chapter 1.

CORBA

Common Object Request Broker Architecture is a common communication standard developed by the OMG (Object Management Group)
Local for chapter 1.

domains

A domain allows a more efficient communication protocol to be used between objects not on the same node without the need of an ORB
Local for chapter 1.

IDL

Interface Definition Language - IDL is the OMG specified interface definition language, used to define the CORBA object interfaces.
Local for chapter 1.

IIOP

Internet-Inter ORB Protocol
Local for chapter 1.

IOR

Interoperable Object Reference
Local for chapter 1.

ORB

Object Request Broker - ORB open software bus architecture specified by the OMG which allows object components to communicate in a heterogeneous environment.
Local for chapter 1.

Orber domain

A domain containing several Erlang nodes, which are communicating by using the Erlang internal format. An Orber domain looks as one ORB from the environment.
Local for chapter 1.

Orber installation

is the structure of the ORB or ORBs as defined during the install process is called the ”installation”.
Local for chapter 1.

Type Code

Type Code is a full definition of a type
Local for chapter 1.

Type Codes

Type codes give a complete description of the type including all its components and structure. Local for chapter 1.

Index

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