Orber Application

version 2.2

Lars Thorsén, Peter Lundell, Per Danielsson

1998-04-25

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 has been 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 of 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
- CosEvent Service (only untyped events)
- 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.

CosEvent Service

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

Tutorial - creating a simple service

The tutorial demonstrates the step by step creation of a simple service.

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
- CosEvent 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:

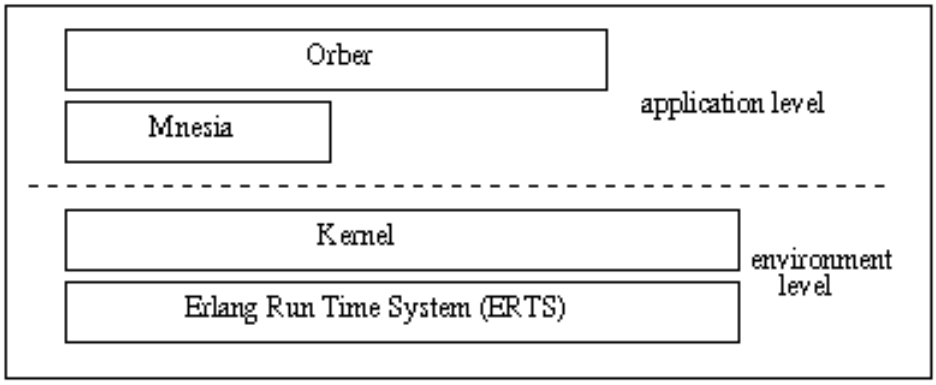

Figure 1.1: Figure 1: Orber Dependencies and Structure.

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.

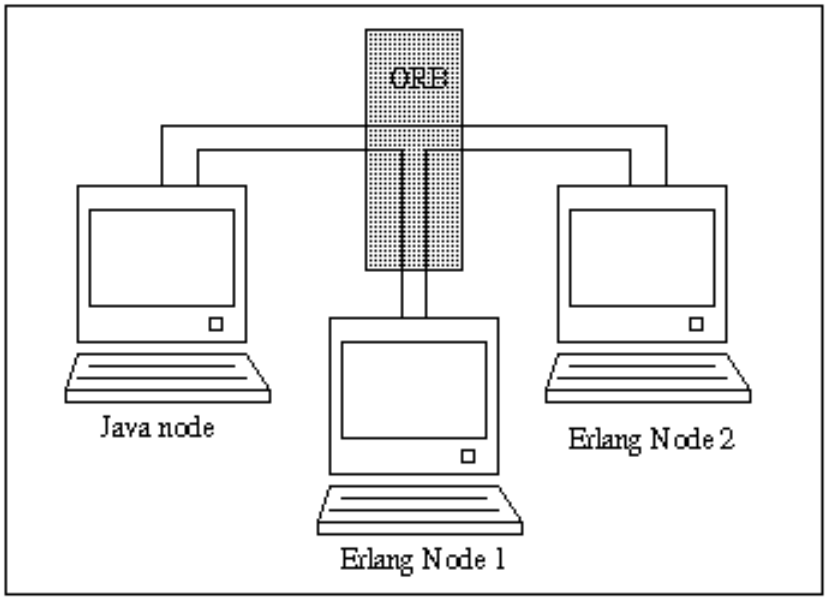

Figure 1.2: Figure 2: ORB interface between Java and Erlang Environment Nodes.

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:

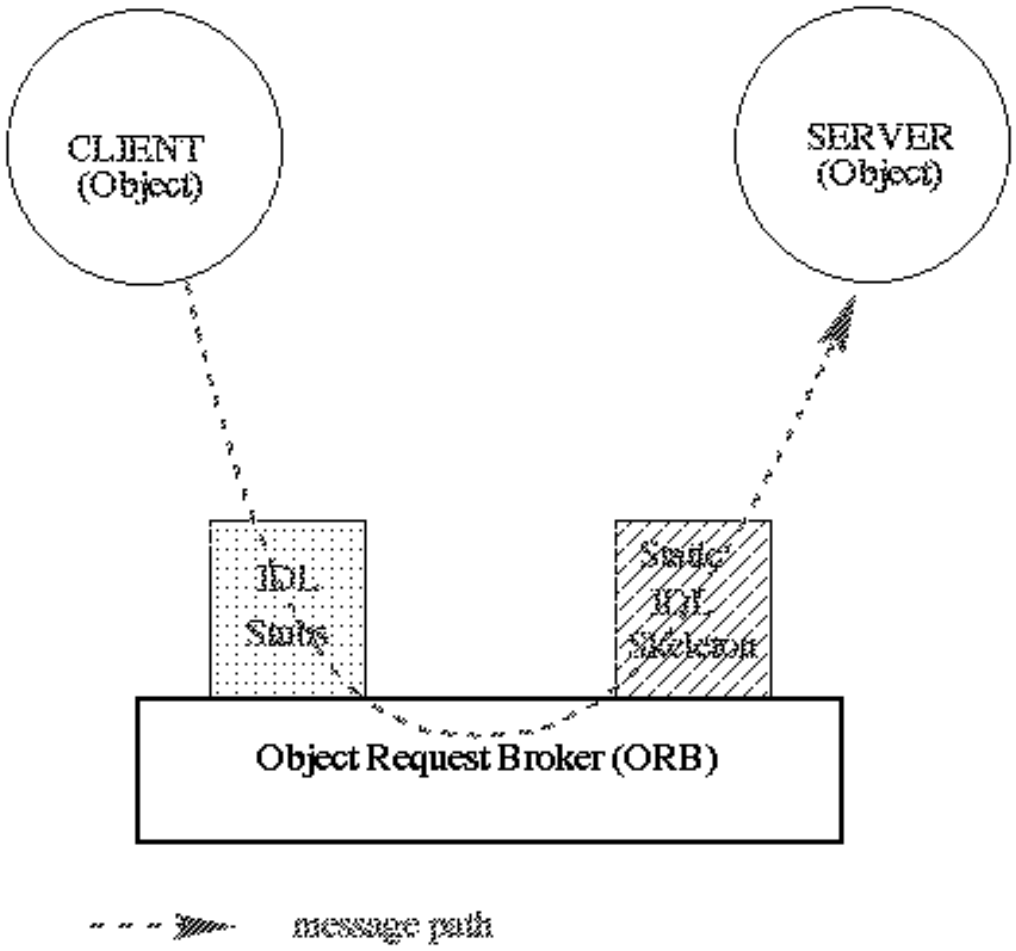

Figure 1.3: Figure 1: How the Object Request Broker works.

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:

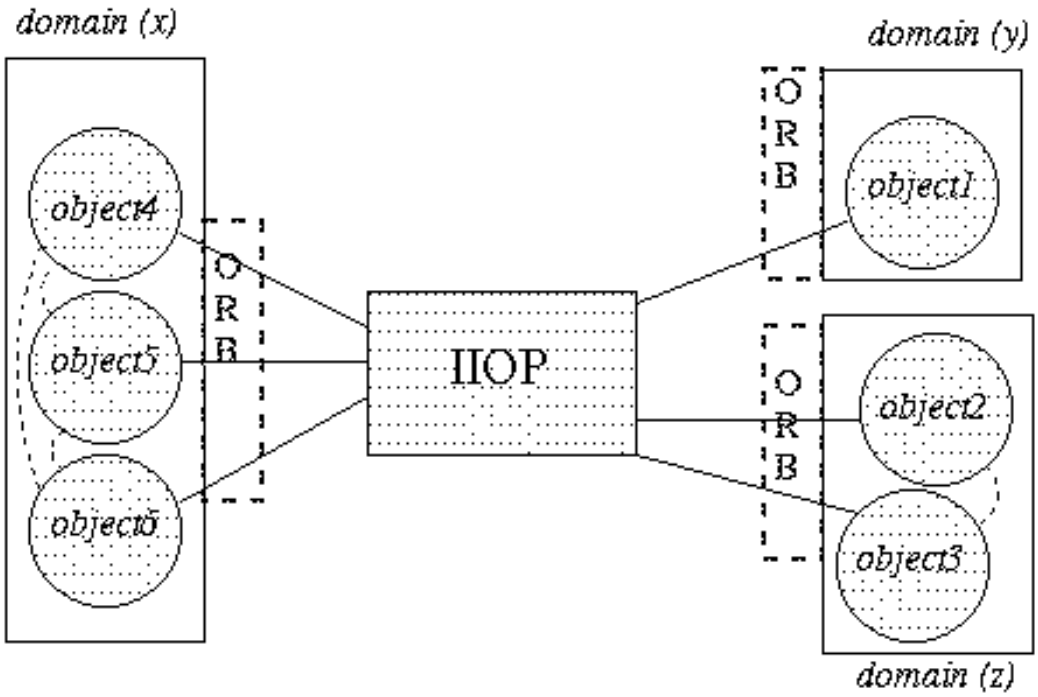

Figure 1.4: Figure 2: IIOP communication between domains and objects.

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.

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.
- *iiop_timeout* - default is infinity. The value is an integer (timeout in seconds) or the atom infinity

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.

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. 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 illegal 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.

If underscores are used in IDL names it 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 straight forward. 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	
unsigned long	Erlang integer	
char	Erlang integer	
boolean	Erlang atoms true or false	
octet	Erlang integer	
any	Erlang record #any{typecode, value}	
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 19], and the value field itself.

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

Constructed OMG IDL types

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

string	Erlang string
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 is accessed by a single function call. The functions are put in the file corresponding to the scope where it is 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 16].

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 shall 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 21]

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 shall 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.

Record access functions.

As mentioned in a previous section, `struct`,`union` and `exception` types yield to record definitions and access code for that record. The functions are put in the file corresponding to the scope where it is defined. Three functions are accessible for each record :

- 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.
```

Record access functions.

As mentioned in a previous section, the types `struct`,`union` and `exception` yield to record definitions and access code for that record. The functions are placed in the file corresponding to the scope where it is defined. Three functions are accessible for each record:

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

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_ushort	
tk_ulong	
tk_float	
tk_double	
tk_boolean	
tk_char	
tk_octet	
tk_any	
tk_TypeCode	
tk_Principal	
{tk_objref, IFRId, Name}	{tk_objref, "IDL:M1\I1:1.0", "I1"}
{tk_struct, IFRId, Name, [{ElemName, ElemTC}]}	{tk_struct, "IDL:M1\S1:1.0", "S1", [{"a", tk_long}, {"b", tk_char}]}
{tk_union, IFRId, Name, DiscrTC, DefaultNr, [{Label, ElemName, ElemTC}]} Note: DefaultNr tells which of tuples in the case list that is default, or -1 if no default	{tk_union, "IDL:U1:1.0", "U1", tk_long, 1, [{1, "a", tk_long}, {default, "b", tk_char}]}
{tk_enum, IFRId, Name, [ElemName]}	{tk_enum, "IDL:E1:1.0", "E1", ["a1", "a2"]}
{tk_string, Length}	{tk_string, 5}
{tk_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 from now refered 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.

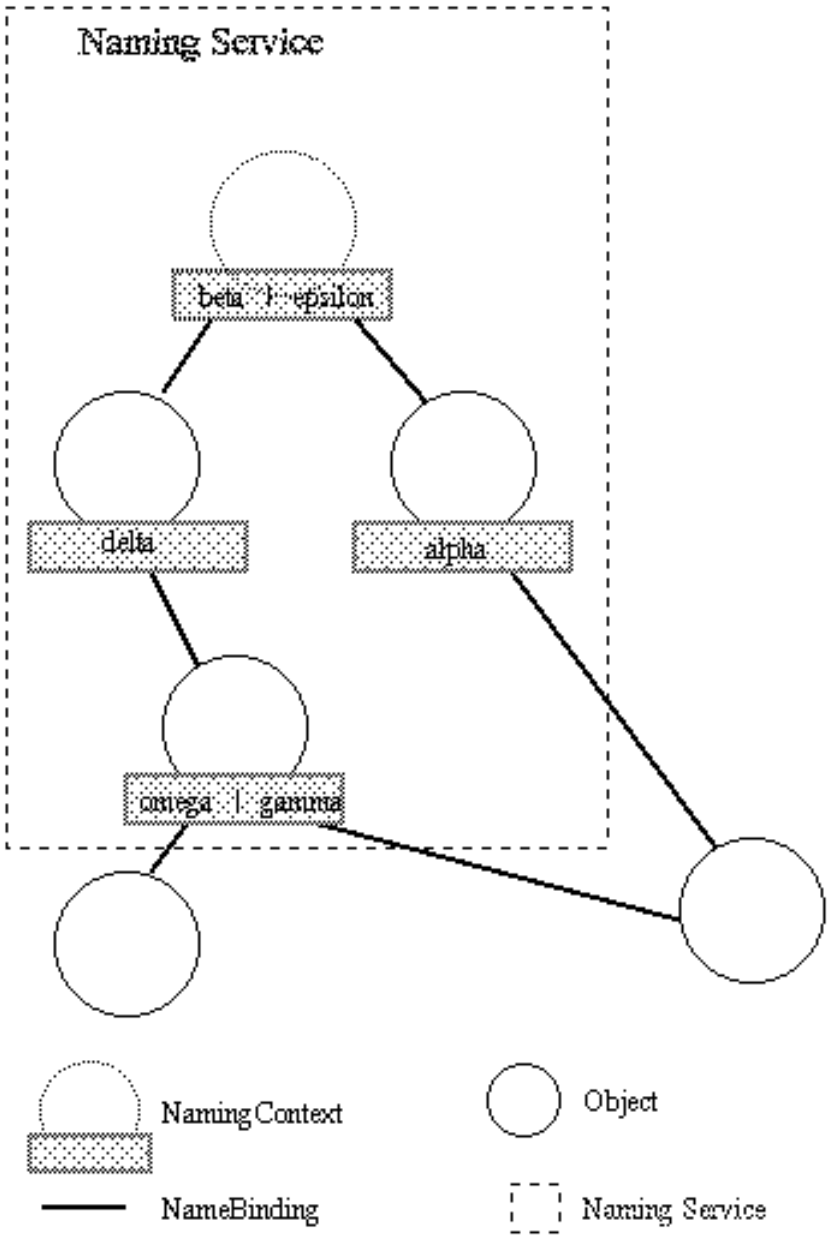

Figure 1.5: Figure 1: Contextual object relationships using the Naming Service.

The naming contexts provides 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 below example, 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 be removed. (they are also removed if all Orber nodes in the domain are stopped).

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

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

1.8 Event Service

Overview of the CosEvent Service

The Event service allows programmers to subscribe to information channels. Suppliers can generate events without knowing the consumer identities and the consumer can receive events without knowing the supplier identity. Both push and pull event delivery are supported. The Event service will queue information and processes.

The CORBA Event service provides a flexible model for asynchronous, decoupled communication between objects. This chapter outlines communication models and the roles and relationships of key components in the CosEvent service. It shows a simple example of use of this service.

Event Service Components

There are five components in the OMG CosEvent service architecture. These are described below:

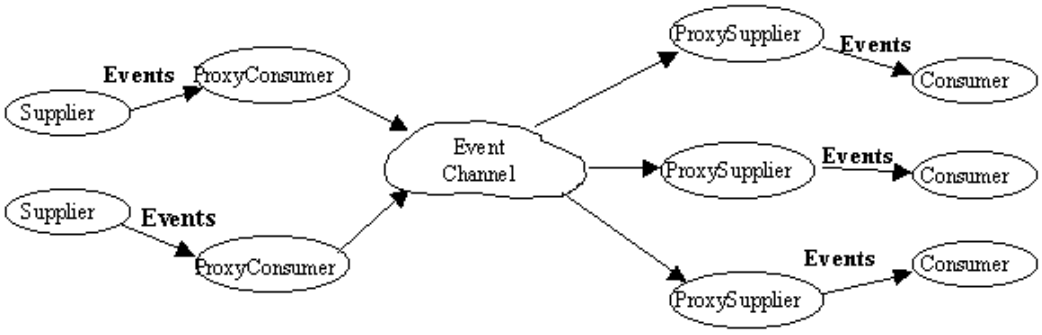

Figure 1.6: Figure 1: Event service Components

- *Suppliers and consumers:* Consumers are the ultimate targets of events generated by suppliers. Consumers and suppliers can both play active and passive roles. There could be two types of consumers and suppliers: push or pull. A PushSupplier object can actively push an event to a passive PushConsumer object. Likewise, a PullSupplier object can passively wait for a PullConsumer object to actively pull an event from it.
- *EventChannel:* The central abstraction in the CosEvent service is the EventChannel which plays the role of a mediator between consumers and suppliers. Consumers and suppliers register their interest with the EventChannel. It can provide many-to-many communication. The channel consumes events from one or more suppliers, and supplies events to one or more consumers. An EventChannel can support consumers and suppliers using different communication models.
- *ProxySuppliers and ProxyConsumers:* ProxySuppliers act as middlemen between consumers and the EventChannel. A ProxySupplier is similar to a normal supplier, but includes an additional method for connecting a consumer to the ProxySupplier. Likewise, ProxyConsumers act as

middlemen between suppliers and the EventChannel. A ProxyConsumer is similar to a normal consumer, but includes an additional method for connecting a supplier to the ProxyConsumer.

- *Supplier and consumer administrations:* Consumer administration acts as a factory for creating ProxySuppliers. Supplier administration acts as a factory for creating ProxyConsumers.

Event Service Communication Models

There are four general models of component collaboration in the OMG CosEvent service architecture. The following describes these models: (Please note that proxies are not shown in the diagrams for simplicity).

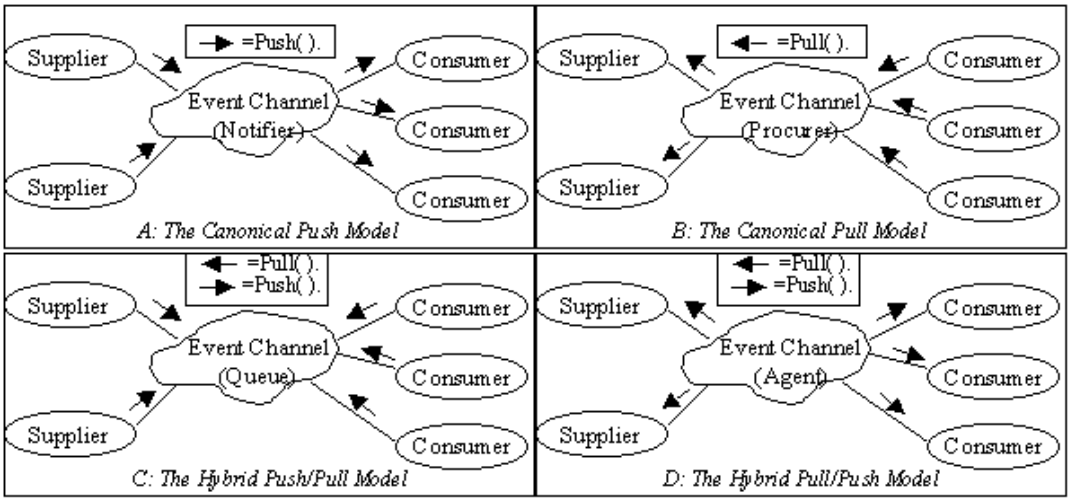

Figure 1.7: Figure 2: Event service Communication Models

- *The Canonical Push Model:* The Canonical push model shown in figure 2(A) allows the suppliers of events to initiate the transfer of event data to consumers. In this model, suppliers are active initiators and consumers are the passive targets of the requests. EventChannels play the role of `Notifier`. Thus, active suppliers use EventChannels to push data to passive consumers that have registered with the EventChannels.
- *The Canonical Pull Model:* The Canonical pull model shown in figure 2(B) allows consumers to request events from suppliers. In this model, Consumers are active initiators and suppliers are the passive targets of the pull requests. EventChannel plays the role of `Procurer` since they procure events on behalf of consumers. Thus, active consumers can explicitly pull data from passive suppliers via the EventChannels.
- *The Hybrid Push/Pull Model:* The push/pull model shown in figure 2(C) is a hybrid that allows consumers to request events queued at an EventChannel by suppliers. In this model, both suppliers and consumers are active initiators of the requests. EventChannels play the role of `Queue`. Thus, active consumers can explicitly pull data deposited by active suppliers via the EventChannels.

- *The Hybrid Pull/Push Model:* The pull/push model shown in figure 2(D) is another hybrid that allows the channel to pull events from suppliers and push them to consumers. In this model, suppliers are passive targets of pull requests and consumers are passive targets of pushes. EventChannels play the role of `Intelligent Agent`. Thus, active EventChannels can pull data from passive suppliers and push that data to passive consumers.

Creating an EventChannel

An EventChannel can be created by using the `EventChannelFactory` interface, which is implemented by `OrberEventChannel_EventChannelFactory`.

To start the factory server one needs to make a call to `corba:create/2` which could look like this:

```
-module(event_channel_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() ->
  ECFok = 'OrberEventChannel_EventChannelFactory':oe_create(),
  NS = corba:resolve_initial_references("Nameservice"),
  NC = lname_component:set_id(lname_component:create(),
                              "EventChannelFactory"),
  N = lname:insert_component(lname:create(), 1, NC),
  'CosNaming_NamingContext':bind(NS, N, ECFok).
```

Now an EventChannelFactory is created and registered in the CosNaming service and could be found by consumers and suppliers.

Using the Event Service

This section shows an example of usage of the Event service in order to decouple communication between a measurements collector and a safety controller.

Using the Consumer interface for safety controller

The safety controller plays the role of a PushConsumer. It is interested in the data provided by the measurements collector, which plays the role of a PushSupplier. Safety controller is responsible for the action to be taken in case if some measurements exceed the safety limits.

First, the safety controller creates a PushConsumer itself, and then obtains an EventSupplier channel object reference using the `EventChannelFactory`, as follows:

```
// The safety controller creates a PushConsumer object
MyPushConsumer = my_push_consumer_srv:create(),

// EventChannel created through EventChannelFactory
// EventChannelFactory obtained from the CosNaming service (not shown)
// EventChannel registered in the CosNaming service (not shown)
EventChannel = 'OrberEventChannel_EventChannelFactory':
                create_event_channel(ECFactory),
```

This code assumes that the `MyPushConsumer` supports the `PushConsumer` interface and implements the appropriate safety controller logic.

Note: If no support exists for the push consumer the process will crash.

Next, the safety controller connects itself to the `EventChannel`:

```
// first step: obtain ConsumerAdmin object reference
ConsumerAdmin = 'CosEventChannelAdmin_EventChannel'
                 :for_consumers(EventChannel),
// obtain ProxyPushSupplier from the ConsumerAdmin object
PPhS = 'CosEventChannelAdmin_ConsumerAdmin'
        :obtain_push_supplier(ConsumerAdmin),

// second step: connect our PushConsumer to the ProxyPushSupplier
'CosEventChannelAdmin_ProxyPushSupplier'
         :connect_push_consumer(PPhS, MyPushConsumer)
```

When an event arrives in the EventChannel, it will invoke the `push` operation on the registered `PushConsumer` object reference.

Using the supplier interface for measurements collector

Measurements collector sends data containing information about current measurement of the system to the EventChannel in order to keep safety controller informed of any changes.

As with the safety controller, the measurements collector needs an object reference to an `EventChannel` and to a PushSupplier to connect to the channel. This accomplished as follows:

```
// measurements collector creates a PushSupplier
MyPushSupplier = my_push_supplier_srv:create(),

// EventChannel obtained from the Naming service (not shown)
EventChannel = //...

// obtain SupplierAdmiin object reference
SupplierAdmin = 'CosEventChannelAdmin_EventChannel':for_suppliers(EventChannel),

// obtain ProxyPushConsumer from SupplierAdmin object
PPhC = 'CosEventChannelAdmin_SupplierAdmin':obtain_push_consumer(SupplierAdmin),

// connect our PushSupplier to the ProxyPushConsumer
'CosEventChannelAdmin_ProxyPushConsumer':connect_push_supplier(PPhC, MyPushSupplier),
```

Once the consumer and the supplier registration code get executed, both the safety controller and the measurements collector are connected to the EventChannel. At this point, safety controller will automatically receive measurements data that are pushed by the measurements collector.

Exchanging and processing event data

The events exchanged between supplier and consumer must always be specified in OMG IDL so that they can be stored into an any type variable. Consider the following data example sent by the measurements controller:

```
        record(measurements, {temperature, pressure, water_level}).
```

In order to push an event, the measurements collector must create and initialize this record, put it into CORBA::any, and call push on the EventChannel PushConsumer interface:

```
// create some data
EventRecord = #measurements{temperature = 150, pressure = 100,
                                          water_level = 200},
EventData = { measurements:tc(),EventRecord},

// push the event to consumer
'CosEventChannelAdmin_ProxyPushConsumer':push(PPhC, EventData),
```

Once the EventChannel receives an event from the measurements collector, it pushes the event data to the consumer by invoking the push operation on registered PushConsumer object reference.

The implementation of the safety controller consumer push could look like this:

```
push(Data) ->
{
  if
    Data#measurements.temperature > 300 ->
      // some logic to set alarm
      ;
    Data#measurements.water_level < 50 ->
      // some logic to get more water
      ;

    ......etc
  end.
}
```

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 (more information needed???). This will produce among other files a stack API module called `StackModule_Stack.erl` which contains the client stub the server skeleton.

Implementation of interface

After generating the API stubs and the server skeletons it's 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.

```
-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.

```
-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.

```
-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.

```
-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`, we have tested with OrbixWeb. To support this, a Java package named `Orber` is included with our product. It contains just one class, `InitialReference` which can be used in to get the initial reference to Orber's naming service. The Java client will then look like this:

```
//
//

package StackModule;

import CosNaming._NamingContextRef;
import CosNaming.Name;
import IE.Iona.Orbix2._CORBA;
import IE.Iona.Orbix2.CORBA.SystemException;
import IE.Iona.Orbix2.CORBA._ObjectRef;

public class StackClient
{
  public static void main(String args[])
    {
      CORBA._InitialReferencesRef init;
      _NamingContextRef nsContext;
      Name name;
      _ObjectRef initRef, nsRef, objRef;
      _StackFactoryRef sfRef = null;
      _StackRef sRef = null;
      Orber.InitialReference ir = new Orber.InitialReference();

      int i;
      String srvHost = new String(args[0]);
      Integer srvPort = new Integer(args[1]);

      try
        {
          // For an explanation about initial reference handling see
          // the "Interoperable Naming Service" specification.
```

```
    // Create Initial reference (objectkey "INIT").
    String s = ir.stringified_ior(srvHost, srvPort.intValue());

    initRef = _CORBA.Orbix.string_to_object(s);

    init = CORBA.InitialReferences._narrow(initRef);
    // Fetch name service reference.
    nsRef = init.get("NameService");

    nsContext = CosNaming.NamingContext._narrow(nsRef);

    // Create a name
    name = new Name(1);
    name.buffer[0] = new CosNaming.NameComponent("StackFactory", "");

    try
      {
        objRef = nsContext.resolve(name);
      }
    catch(IE.Iona.Orbix2.CORBA.UserException n)
      {
        System.out.println("Unexpected exception: " + n.toString());
        return;
      }
    sfRef = StackFactory._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 supply CosNaming it is possible to use this package instead. We have tested with Sun Microsystems Java IDL (import org.omg.CosNaming.*;)

Building the example

To build the example for access from a Java client you need a Java enabled ORB. In the build log below OrbixWeb's IDL compiler was used.

```
fingolfin 127> erl
Erlang (JAM) emulator version 4.6

Eshell V4.5.3  (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's used as default in OTP R3A, in R2 it's 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 (JAM) emulator version 4.6

Eshell V4.5.3  (abort with ^G)
1> mnesia:create_schema([]).
ok
2> orber:install([]).
ok
3> orber:start().
ok
4> oe_stack:oe_register().
ok
5> stack_factory:start().
ok
6> stack_client:run().
1
1
7
4
ok
7>
```

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

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

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

1.10 Orber Release Notes

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 a 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 an 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++ ie.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 theese 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's 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's 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 it's own application, called ic.
  Own Id: OTP-2483
- It's 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 it's 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

Short Summaries

- Erlang Module CosEventChannelAdmin [page ??] – The CosEventChannelAdmin defines a set if event service interfaces that enables decoupled asynchronous communication between objects and implements generic (untyped) version of the OMG COSS standard event service.
- Erlang Module CosEventChannelAdmin_ConsumerAdmin [page ??] – This module implements a ConsumerAdmin interface, which allows consumers to be connected to the event channel.
- Erlang Module CosEventChannelAdmin_EventChannel [page ??] – This module implements an Event Channel interface, which plays the role of a mediator between consumers and suppliers.
- Erlang Module CosEventChannelAdmin_ProxyPullConsumer [page ??] – This module implements a ProxyPullConsumer interface which acts as a middleman between pull supplier and the event channel.
- Erlang Module CosEventChannelAdmin_ProxyPullSupplier [page ??] – This module implements a ProxyPullSupplier interface which acts as a middleman between pull consumer and the event channel.
- Erlang Module CosEventChannelAdmin_ProxyPushConsumer [page ??] – This module implements a ProxyPushConsumer interface which acts as a middleman between push supplier and the event channel.
- Erlang Module CosEventChannelAdmin_ProxyPushSupplier [page ??] – This module implements a ProxyPushSupplier interface which acts as a middleman between push consumer and the event channel.
- Erlang Module CosEventChannelAdmin_SupplierAdmin [page ??] – This module implements a SupplierAdmin interface, which allows suppliers to be connected to the event channel.
- Erlang Module CosNaming [page ??] – The CosNaming service is a collection of interfaces that together define the naming service.
- Erlang Module CosNaming_BindingIterator [page ??] – This interface supports iteration over a name binding list.
- Erlang Module CosNaming_NamingContext [page ??] – This interface supports different bind and access functions for names in a context.
- Erlang Module OrberEventChannel [page ??] – The OrberEventChannel defines an interface that enables the user to create event channel objects.

- Erlang Module OrberEventChannel_EventChannelFactory [page ??] – This module implements the EventChannelFactory interface that enables the user to create event channel objects.
- Erlang Module any [page 89] – the corba any type
- Erlang Module corba [page 91] – The functions on CORBA module level
- Erlang Module corba_object [page 95] – The Corba Object interface functions
- Erlang Module lname [page 97] – Interface that supports the name pseudo-objects.
- Erlang Module lname_component [page 99] – Interface that supports the name pseudo-objects.
- Erlang Module orber [page 101] – The main module of the Orber application
- Erlang Module orber_ifr [page 104] – The Interface Repository stores representations of IDL information
- Erlang Module orber_tc [page 118] – help functions for IDL typecodes

CosEventChannelAdmin

No functions are exported

CosEventChannelAdmin_ConsumerAdmin

The following functions are exported:

- `obtain_push_supplier(Object) -> Return`
  [page 69] Creates a ProxyPushSupplier object
- `obtain_pull_supplier(Object) -> Return`
  [page 69] Creates a ProxyPullSupplier object

CosEventChannelAdmin_EventChannel

The following functions are exported:

- `for_consumers(Object) -> Return`
  [page 70] Returns a ConsumerAdmin object
- `for_suppliers(Object) -> Return`
  [page 70] Returns a SupplierAdmin object
- `destroy(Object) -> Return`
  [page 70] Destroys the event channel

CosEventChannelAdmin_ProxyPullConsumer

The following functions are exported:

- `connect_pull_supplier(Object, PullSupplier) -> Return`
  [page 72] Connects pull supplier to the proxy pull consumer
- `disconnect_pull_consumer(Object) -> Return`
  [page 72] Disconnects the ProxyPullConsumer object from the event channel.

CosEventChannelAdmin_ProxyPullSupplier

The following functions are exported:

- `connect_pull_consumer(Object, PullConsumer) -> Return`
  [page 73] Connects pull consumer to the proxy pull supplier
- `disconnect_pull_supplier(Object) -> Return`
  [page 73] Disconnects the ProxyPullSupplier object from the event channel.
- `pull(Object) -> Return`
  [page 73] Transmits data from suppliers to consumers.
- `try_pull(Object) -> Return`
  [page 74] Transmits data from suppliers to consumers.

CosEventChannelAdmin_ProxyPushConsumer

The following functions are exported:

- `connect_push_supplier(Object, PushSupplier) -> Return`
  [page 75] Connects push supplier to the proxy push consumer
- `disconnect_push_consumer(Object) -> Return`
  [page 75] Disconnects the ProxyPushConsumer object from the event channel.
- `push(Object, Data) -> Return`
  [page 75] Communicates event data to the consumers.

CosEventChannelAdmin_ProxyPushSupplier

The following functions are exported:

- `connect_push_consumer(Object, PushConsumer) -> Return`
  [page 77] Connects push consumer to the proxy push supplier
- `disconnect_push_supplier(Object) -> Return`
  [page 77] Disconnects the ProxyPushSupplier object from the event channel.

CosEventChannelAdmin_SupplierAdmin

The following functions are exported:

- `obtain_push_consumer(Object) -> Return`
  [page 78] Creates a ProxyPushConsumer object
- `obtain_pull_consumer(Object) -> Return`
  [page 78] Creates a ProxyPullConsumer object

CosNaming

No functions are exported

CosNaming_BindingIterator

The following functions are exported:

- `next_one(BindinIterator) -> Return`
  [page 82] Returns a binding
- `next_n(BindinIterator, HowMany) -> Return`
  [page 82] Returns a binding list
- `destroy(BindingIterator) -> Return`
  [page 82] destroys the iterator object

CosNaming_NamingContext

The following functions are exported:

- `bind(NamingContext, Name, Object) -> Return`
  [page 85] Bind a Name to an Object
- `rebind(NamingContext, Name, Object) -> Return`
  [page 85] Bind an Object to the Name even if the Name already is bound
- `bind_context(NamingContext1, Name, NamingContex2) -> Return`
  [page 85] Bind a Name to an NamingContext
- `rebind_context(NamingContext1, Name, NamingContex2) -> Return`
  [page 85] Bind an NamingContext to the Name even if the Name already is bound
- `resolve(NamingContext, Name) -> Return`
  [page 85] Retrieve an Object bound to Name
- `unbind(NamingContext, Name) -> Return`
  [page 86] Remove the binding for a Name
- `new_context(NamingContext) -> Return`
  [page 86] Create a new NamingContext
- `bind_new_context(NamingContext, Name) -> Return`
  [page 86] Create a new NamingContext and bind it to a Name

- `destroy(NamingContext) -> Return`
  [page 86] Destroy a NamingContext
- `list(NamingContext, HowMany) -> Return`
  [page 86] List returns a all bindings in the context

OrberEventChannel

No functions are exported

OrberEventChannel_EventChannelFactory

The following functions are exported:

- `create_event_channel(Object) -> Return`
  [page 88] Creates a event channel object

any

The following functions are exported:

- `create() -> Result`
  [page 89] creates an any record
- `create(Typecode, Value) -> Result`
  [page 89] creates an any record
- `set_typecode(A, Typecode) -> Result`
  [page 89] sets the typecode field
- `get_typecode(A) -> Result`
  [page 89] fetches the typecode
- `set_value(A, Value) -> Result`
  [page 90] sets the value field
- `get_value(A) -> Result`
  [page 90] fetches the value

corba

The following functions are exported:

- `create(Module, TypeID) -> Object`
  [page 91] create and start a new server object
- `create(Module, TypeID, Env) -> Object`
  [page 91] create and start a new server object
- `create(Module, TypeID, Env, ServerName) -> Object`
  [page 91] create and start a new server object

- `create_link(Module, TypeID) -> Object`
  [page 91] create and start a new server object
- `create_link(Module, TypeID, Env) -> Object`
  [page 91] create and start a new server object
- `create_link(Module, TypeID, Env, ServerName) -> Object`
  [page 91] create and start a new server object
- `dispose(Object) -> ok`
  [page 92] stops a server object
- `dispose(Object) -> ok`
  [page 92] stops a server object
- `create_subobject_key(Object, Key) -> Result`
  [page 92] adds an erlang term to a private key field
- `get_subobject_key(Object) -> Result`
  [page 92] fetch the contents of the private key field
- `get_pid(Object) -> Result`
  [page 92] get the process id from an object key
- `raise(Exception)`
  [page 92] generates an erlang throw
- `resolve_initial_references(ObjectId) -> Object`
  [page 93] returns the object reference for the given object id
- `list_initial_services() -> [ObjectId]`
  [page 93] returns a list of supported object id's
- `resolve_initial_references_remote(ObjectId, Address) -> Object`
  [page 93] returns the object reference for the given object id
- `list_initial_services_remote(Address) -> [ObjectId]`
  [page 93] returns a list of supported object id's
- `object_to_string(Object) -> IOR_string`
  [page 93] converts the object reference to the external string representation
- `string_to_object(IOR_string) -> Object`
  [page 93] converts the external string representation to an object reference

corba_object

The following functions are exported:

- `get_interface(Object) -> InterfaceDef`
  [page 95] Fetch the interface description
- `is_nil(Object) -> boolean()`
  [page 95]
- `is_a(Object, Logical_type_id) -> Return`
  [page 95]
- `is_remote(Object) -> boolean()`
  [page 95] Determines whether or not an object reference is remote.
- `non_existent(Object) -> Return`
  [page 96]

- `is_equivalent(Object, OtherObject) -> boolean()`
  [page 96]
- `hash(Object, Maximum) -> int()`
  [page 96]

Iname

The following functions are exported:

- `create() -> Return`
  [page 97] creates a new name
- `insert_component(Name, N, NameComponent) -> Return`
  [page 97] inserts a new name component in a name
- `get_component(Name, N) -> Return`
  [page 97] get a name component from a name
- `delete_component(Name, N) -> Return`
  [page 98] deletes s name component from a name
- `num_components(Name) -> Return`
  [page 98] counts the number of name components in a name
- `equal(Name1, Name2) -> Return`
  [page 98] tests if two names are equal
- `less_than(Name1, Name2) -> Return`
  [page 98] tests if one name is lesser than the other
- `to_idl_form(Name) -> Return`
  [page 98] transforms a pseudo name to an IDL name
- `from_idl_form(Name) -> Return`
  [page 98] transforms an IDL name to a pseudo name

Iname_component

The following functions are exported:

- `create() -> Return`
  [page 99] creates a new name component
- `get_id(NameComponent) -> Return`
  [page 99] get the id field of a name component
- `set_id(NameComponent, Id) -> Return`
  [page 99] set the id field of a name component
- `get_kind(NameComponent) -> Return`
  [page 99] get the kind field of a name component
- `set_kind(NameComponent, Kind) -> Return`
  [page 100] set the kind field of a name component

orber

The following functions are exported:

- `start() -> ok`
  [page 101] Start the Orber application
- `stop() -> ok`
  [page 101] Stops the Orber application
- `domain() -> string()`
  [page 101] Display the Orber domain name
- `iiop_port() -> int()`
  [page 101] Display the IIOP port number
- `iiop_timeout() -> int() (milliseconds)`
  [page 101] Display the IIOP timeout value
- `bootstrap_port() -> int()`
  [page 101] Display the bootstrap protocol port number
- `orber_nodes() -> RetVal`
  [page 102] Displays which nodes that this orber domain consist of.
- `install(NodeList) -> ok`
  [page 102] Installs the Orber application
- `install(NodeList, Options) -> ok`
  [page 102] Installs the Orber application
- `uninstall() -> ok`
  [page 102] Uninstall the Orber application
- `add_node(Node, StorageType) -> RetVal`
  [page 103] Adds a new node to a group of Orber nodes.
- `remove_node(Node) -> RetVal`
  [page 103] Removes a node from a group of Orber nodes.

orber_ifr

The following functions are exported:

- `init(Nodes,Timeout) -> ok`
  [page 104] Intialize the IFR
- `find_repository() -> #IFR_Repository_objref`
  [page 104]
- `get_def_kind(Objref) -> Return`
  [page 105]
- `destroy(Objref) -> Return`
  [page 105]
- `get_id(Objref) -> Return`
  [page 105]
- `set_id(Objref,Id) -> ok`
  [page 105]

- get_type(Objref) -> Return
  [page 110]
- lookup_id(Objref,Search_id) -> Return
  [page 110]
- get_primitive(Objref,Kind) -> Return
  [page 110]
- create_string(Objref,Bound) -> Return
  [page 110]
- create_sequence(Objref,Bound,Element_type) -> Return
  [page 111]
- create_array(Objref,Length,Element_type) -> Return
  [page 111]
- create_idltype(Objref,Typecode) -> Return
  [page 111]
- get_type_def(Objref) -> Return
  [page 111]
- set_type_def(Objref,TypeDef) -> Return
  [page 111]
- get_value(Objref) -> Return
  [page 111]
- set_value(Objref,Value) -> Return
  [page 112]
- get_members(Objref) -> Return
  [page 112]
- set_members(Objref,Members) -> Return
  [page 112]
- get_discriminator_type(Objref) -> Return
  [page 112]
- get_discriminator_type_def(Objref) -> Return
  [page 112]
- set_discriminator_type_def(Objref,TypeDef) -> Return
  [page 112]
- get_original_type_def(Objref) -> Return
  [page 113]
- set_original_type_def(Objref,TypeDef) -> Return
  [page 113]
- get_kind(Objref) -> Return
  [page 113]
- get_bound(Objref) -> Return
  [page 113]
- set_bound(Objref,Bound) -> Return
  [page 113]
- get_element_type(Objref) -> Return
  [page 113]
- get_element_type_def(Objref) -> Return
  [page 114]

- set_element_type_def(Objref,TypeDef) -> Return
  [page 114]
- get_length(Objref) -> Return
  [page 114]
- set_length(Objref,Length) -> Return
  [page 114]
- get_mode(Objref) -> Return
  [page 114]
- set_mode(Objref,Mode) -> Return
  [page 114]
- get_result(Objref) -> Return
  [page 115]
- get_result_def(Objref) -> Return
  [page 115]
- set_result_def(Objref,ResultDef) -> Return
  [page 115]
- get_params(Objref) -> Return
  [page 115]
- set_params(Objref,Params) -> Return
  [page 115]
- get_contexts(Objref) -> Return
  [page 115]
- set_contexts(Objref,Contexts) -> Return
  [page 116]
- get_exceptions(Objref) -> Return
  [page 116]
- set_exceptions(Objref,Exceptions) -> Return
  [page 116]
- get_base_interfaces(Objref) -> Return
  [page 116]
- set_base_interfaces(Objref,BaseInterfaces) -> Return
  [page 116]
- is_a(Objref,Interface_id) -> Return
  [page 116]
- describe_interface(Objref) -> Return
  [page 117]
- create_attribute(Objref,Id,Name,Version,Type,Mode) -> Return
  [page 117]
- create_operation(Objref,Id,Name,Version,Result,Mode,Params,
  Exceptions,Contexts) -> Return
  [page 117]

orber_tc

The following functions are exported:

- `null() -> TC`
  [page 118] get the IDL typecode
- `void() -> TC`
  [page 118] get the IDL typecode
- `short() -> TC`
  [page 118] get the IDL typecode
- `unsigned_short() -> TC`
  [page 118] get the IDL typecode
- `long() -> TC`
  [page 118] get the IDL typecode
- `unsigned_long() -> TC`
  [page 118] get the IDL typecode
- `float() -> TC`
  [page 118] get the IDL typecode
- `double() -> TC`
  [page 118] get the IDL typecode
- `boolean() -> TC`
  [page 118] get the IDL typecode
- `char() -> TC`
  [page 118] get the IDL typecode
- `octet() -> TC`
  [page 118] get the IDL typecode
- `any() -> TC`
  [page 118] get the IDL typecode
- `typecode() -> TC`
  [page 118] get the IDL typecode
- `principal() -> TC`
  [page 118] get the IDL typecode
- `object_reference(Id, Name) -> TC`
  [page 118] the object_reference IDL typecode
- `struct(Id, Name, ElementList) -> TC`
  [page 118] the struct IDL typecode
- `union(Id, Name, DiscrTC, Default, ElementList) -> TC`
  [page 119] the union IDL typecode
- `enum(Id, Name, ElementList) -> TC`
  [page 119] the enum IDL typecode
- `string(Length) -> TC`
  [page 119] the string IDL typecode
- `sequence(ElemTC, Length) -> TC`
  [page 120] the sequence IDL typecode
- `array(ElemTC, Length) -> TC`
  [page 120] the array IDL typecode
- `alias(Id, Name, AliasTC) -> TC`
  [page 120] the alias IDL typecode
- `exception(Id, Name, ElementList) -> TC`
  [page 120] the exception IDL typecode

- `get_tc(Object) -> TC`
  [page 120] fetch typecode
- `get_tc(Id) -> TC`
  [page 120] fetch typecode
- `check(TC) -> boolean()`
  [page 121] syntax check of an IDL typecode

CosEventChannelAdmin (Module)

The event service defines two roles for objects: the supplier role and the consumer role. Suppliers supply event data to the event channel and consumers receive event data from the channel. Suppliers do not need to know the identity of the consumers, and vice versa. Consumers and suppliers are connected to the event channel via proxies, which are managed by ConsumerAdmin and SupplierAdmin objects.

There are four general models of communication. These are:

- The canonical push model. It allows the suppliers of events to initiate the transfer of event data to consumers. Event channels play the role of `Notifier`. Active suppliers use event channel to push data to passive consumers registered with the event channel.
- The canonical pull model. It allows consumers to request events from suppliers. Event channels play the role of `Procure` since they procure events on behalf of consumers. Active consumers can explicitly pull data from passive suppliers via the event channels.
- The hybrid push/pull model. It allows consumers request events queued at a channel by suppliers. Event channels play the role of `Queue`. Active consumers explicitly pull data deposited by active suppliers via the event channels.
- The hybrid pull/push model. It allows the channel to pull events from suppliers and push them to consumers. Event channels play the role of `Intelligent agent`. Active event channels can pull data from passive suppliers to push it to passive consumers.

To get access to the record definitions for the structuress use:
`-include_lib("orber/COSS/CosEvent/CosEventChannelAdmin.hrl")..`

There are seven different interfaces supported in the service:

- ProxyPushConsumer
- ProxyPullSupplier
- ProxyPullConsumer
- ProxyPushSupplier
- ConsumerAdmin
- SupplierAdmin
- EventChannel

IDL specification for CosEventChannelAdmin:

```
#ifndef _COSEVENTCHANELADMIN_IDL
#define _COSEVENTCHANELADMIN_IDL

#include "CosEventComm.idl"

#pragma prefix "omg.org"

module CosEventChannelAdmin
{
        exception AlreadyConnected{};
        exception TypeError{};

        interface ProxyPushConsumer: CosEventComm::PushConsumer
        {
                void connect_push_supplier(in CosEventComm::PushSupplier push_supplie
        };

        interface ProxyPullSupplier: CosEventComm::PullSupplier
        {
                void connect_pull_consumer(in CosEventComm::PullConsumer pull_consume
        };

        interface ProxyPullConsumer: CosEventComm::PullConsumer
        {
                void connect_pull_supplier(in CosEventComm::PullSupplier pull_supplie
        };

        interface ProxyPushSupplier: CosEventComm::PushSupplier
        {
                void connect_push_consumer(in CosEventComm::PushConsumer push_consume
        };

        interface ConsumerAdmin
        {
                ProxyPushSupplier obtain_push_supplier();
                ProxyPullSupplier obtain_pull_supplier();
        };

        interface SupplierAdmin
        {
                ProxyPushConsumer obtain_push_consumer();
                ProxyPullConsumer obtain_pull_consumer();
        };

        interface EventChannel
        {
                ConsumerAdmin for_consumers();
                SupplierAdmin for_suppliers();
                void destroy();
        };

};
```

```
#endif
```

CosEventChannelAdmin_ConsumerAdmin (Module)

The ConsumerAdmin interface defines the first step for connecting consumers to the event channel. It acts as a factory for creating proxy suppliers. Both consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers.

Any object that possesses an object reference that supports the EventChannelFactory interface can perform the following operations:

Exports

`obtain_push_supplier(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ProxyPushSupplier object reference.

`obtain_pull_supplier(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ProxyPullSupplier object reference.

CosEventChannelAdmin_EventChannel (Module)

An event channel is an object that allows multiple suppliers to communicate with multiple consumers in a highly decoupled, asynchronous manner. The event channel is built up incrementally. An event channel factory could be used for creating an event channel. This factory could be found in OrberEventChannel_EventChannelFactory module. When an event channel is created no suppliers or consumers are connected to it. Event Channel can implement group communication by serving as a replicator, broadcaster, or multicaster that forward events from one or more suppliers to multiple consumers.

It is up to the user to decide when an event channel is created and how references to the event channel are obtained. By representing the event channel as an object, it has all of the properties that apply to objects. One way to manage an event channel is to register it in a naming context, or export it through an operation on an object.

Any object that possesses an object reference that supports the ProxyPullConsumer interface can perform the following operations:

Exports

`for_consumers(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ConsumerAdmin object reference. If ConsumerAdmin object does not exist already it creates one.

`for_suppliers(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a SupplierAdmin object reference. If SupplierAdmin object does not exist already it creates one.

`destroy(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

CosEventChannelAdmin_ProxyPullConsum (Module)

The ProxyPullConsumer interface defines the second step for connecting pull suppliers to the event channel. A proxy consumer is similar to a normal consumer, but includes an additional method for connecting a supplier to the proxy consumer.

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

- AlreadyConnected is defined as `-record('AlreadyConnected', {}).`
- CORBA standard BAD_PARAM is defined as `-record('BAD_PARAM', {'OE_ID', minor, completion_status}).`

The first exception is defined in the file `event_service.hrl` and the second one in the file `corba.hrl`.

Any object that possesses an object reference that supports the ProxyPullConsumer interface can perform the following operations:

Exports

`connect_pull_supplier(Object, PullSupplier) -> Return`

Types:

- Object = #objref
- PullSupplier = #objref of PullSupplier type
- Return = void

This operation connects PullSupplier object to the ProxyPullConsumer object. If a nil object reference is passed CORBA standard `BAD_PARAM` exception is raised. If the ProxyPullConsumer is already connected to a PullSupplier, then the `AlreadyConnected` exception is raised.

`disconnect_pull_consumer(Object) -> Return`

Types:

- Object = #objref
- Return = void

This operation disconnects proxy pull consumer from the event channel and sends a notification about the loss of the connection to the pull supplier attached to it.

CosEventChannelAdmin_ProxyPullSuppl (Module)

The ProxyPullSupplier interface defines the second step for connecting pull consumers to the event channel. A proxy supplier is similar to a normal supplier, but includes an additional method for connecting a consumer to the proxy supplier.

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

- AlreadyConnected is defined as `-record('AlreadyConnected', {}).`
- Disconnected is defined as `-record('Disconnected', {}).`

These exceptions are defined in the file `event_service.hrl`.

Any object that possesses an object reference that supports the ProxyPullSupplier interface can perform the following operations:

Exports

`connect_pull_consumer(Object, PullConsumer) -> Return`

Types:

- Object = #objref
- PullConsumer = #objref of PullConsumer type
- Return = void

This operation connects PullConsumer object to the ProxyPullSupplier object. A nil object reference can be passed to this operation. If so a channel cannot invoke the disconnect_pull_consumer operation on the consumer; the consumer may be disconnected from the channel without being informed. If the ProxyPullSupplier is already connected to a PullConsumer, then the `AlreadyConnected exception` is raised.

`disconnect_pull_supplier(Object) -> Return`

Types:

- Object = #objref
- Return = void

This operation disconnects proxy pull supplier from the event channel. It sends a notification about the loss of the connection to the pull consumer attached to it, unless nil object reference was passed at the connection time.

`pull(Object) -> Return`

Types:

- Object = #objref
- Return = any

This operation blocks until the event data is available or the `Disconnected` exception is raised. It returns the event data to the consumer.

`try_pull(Object) -> Return`

Types:

- Object = #objref
- Return = {any, bool()}

This operation does not block: if the event data is available, it returns the event data and sets the data availability flag to true; otherwise it returns a long with a value of 0 and sets the data availability to false. If the event communication has already been disconnected, the `Disconnected` exception is raised.

CosEventChannelAdmin_ProxyPushCons (Module)

The ProxyPushConsumer interface defines the second step for connecting push suppliers to the event channel. A proxy consumer is similar to a normal consumer, but includes an additional method for connecting a supplier to the proxy consumer.

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

- AlreadyConnected is defined as `-record('AlreadyConnected', {}).`
- Disconnected is defined as `-record('Disconnected', {}).`

These exceptions are defined in the file `event_service.hrl`.

Any object that possesses an object reference that supports the ProxyPushConsumer interface can perform the following operations:

Exports

`connect_push_supplier(Object, PushSupplier) -> Return`

Types:

- Object = #objref
- PushSupplier = #objref of PushSupplier type
- Return = void

This operation connects PushSupplier object to the ProxyPushConsumer object. A nil object reference can be passed to this operation. If so a channel cannot invoke the disconnect_push_supplier operation on the supplier; the supplier may be disconnected from the channel without being informed. If the ProxyPushConsumer is already connected to a PushSupplier, then the `AlreadyConnected exception` is raised.

`disconnect_push_consumer(Object) -> Return`

Types:

- Object = #objref
- Return = void

This operation disconnects proxy push consumer from the event channel. Sends a notification about the loss of the connection to the push supplier attached to it, unless nil object reference was passed at the connection time.

`push(Object, Data) -> Return`

Types:

- Object = #objref
- Data = any
- Return = void

This operation sends event data to all connected consumers via the event channel. If the event communication has already been disconnected, the `Disconnected` is raised.

CosEventChannelAdmin_ProxyPushSupp (Module)

The ProxyPushSupplier interface defines the second step for connecting push consumers to the event channel. A proxy supplier is similar to a normal supplier, but includes an additional method for connecting a consumer to the proxy supplier.

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

- AlreadyConnected is defined as `-record('AlreadyConnected', {}).`
- CORBA standard BAD_PARAM is defined as `-record('BAD_PARAM', {'OE_ID', minor, completion_status}).`

The first exception is defined in the file `event_service.hrl` and the second one in the file `corba.hrl`.

Any object that possesses an object reference that supports the ProxyPushSupplier interface can perform the following operations:

Exports

`connect_push_consumer(Object, PushConsumer) -> Return`

Types:

- Object = #objref
- PushConsumer = #objref of PushConsumer type
- Return = void

This operation connects PushConsumer object to the ProxyPushSupplier object. If a nil object reference is passed CORBA standard `BAD_PARAM` exception is raised. If the ProxyPushSupplier is already connected to a PushConsumer, then the `AlreadyConnected exception` is raised.

`disconnect_push_supplier(Object) -> Return`

Types:

- Object = #objref
- Return = void

This operation disconnects proxy push supplier from the event channel and sends a notification about the loss of the connection to the push consumer attached to it.

CosEventChannelAdmin_SupplierAdmin (Module)

The SupplierAdmin interface defines the first step for connecting suppliers to the event channel. It acts as a factory for creating proxy consumers. Both consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers.

Any object that possesses an object reference that supports the EventChannelFactory interface can perform the following operations:

Exports

`obtain_push_consumer(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ProxyPushConsumer object reference.

`obtain_pull_consumer(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ProxyPullConsumer object reference.

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 Specificatio
// 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 = {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.

OrberEventChannel (Module)

There is only one interface supported in the service:

- EventChannelFactory

IDL specification for OrberEventChannel:

```
#ifndef _EVENT_CHANNEL_FACTORY_IDL
#define _EVENT_CHANNEL_FACTORY_IDL

#include "OrberEventChannelAdmin.idl"

#pragma prefix "omg.org"

module OrberEventChannel
{
        interface EventChannelFactory
        {
                OrberEventChannelAdmin::EventChannel create_event_channel();
        };
};

#endif
```

OrberEventChannel_EventChannelFactory (Module)

Any object that possesses an object reference that supports the EventChannelFactory interface can perform the following operations:

Exports

`create_event_channel(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns an EventChannel object reference.

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, ServerName) -> Object
create_link(Module, TypeID) -> Object
create_link(Module, TypeID, Env) -> Object
create_link(Module, TypeID, Env, ServerName) -> Object
```

Types:

- Module = atom()
- TypeID = string()
- Env = term()
- ServerName = {local, atom()} | {global, atom()}
- Object = #objref

These functions start a new server object. If you start it without *ServerName* it can only be accessed through the returned object key. Started with a *ServerName* 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* callback 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 callback 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.

```
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.

`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 want's 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's a binary it will be stored as is and otherwise it's 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

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

`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 won't 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 doesn't 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 won't 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 doesn't 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's not running).

`stop() -> ok`

Stops the Orber application.

`domain() -> string()`

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

`iiop_port() -> int()`

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

`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's 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.

`bootstrap_port() -> int()`

This function returns the portnumber, 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's 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 isn't ready in this area we in the meantime uses a port number, which don't require root permissions in Unix.

```
orber_nodes() -> RetVal
```

Types:

- RetVal = [node()]

This function returns the list of nodenames 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 multinode domain

Mnesia will be started by the function if it's 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() <v>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's 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

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.

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 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`.