

ic

Copyright © 1998-2017 Ericsson AB. All Rights Reserved.
ic 4.4
December 1, 2017

Copyright © 1998-2017 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 1, 2017

1 IC User's Guide

The *IC* application is an Erlang implementation of an IDL compiler.

1.1 Using the IC Compiler

1.1.1 Introduction

The IC application is an IDL compiler implemented in Erlang. The IDL compiler generates client stubs and server skeletons. Several back-ends are supported, and they fall into three main groups.

The first group consists of a CORBA back-end:

IDL to Erlang CORBA

> This back-end is for CORBA communication and implementation, and the generated code uses the CORBA specific protocol for communication between clients and servers. See the *Orber* application User's Guide and manuals for further details.

The second group consists of a simple Erlang back-end:

IDL to plain Erlang

> This back-end provides a very simple Erlang client interface. It can only be used within an Erlang node, and the communication between client and "server" is therefore in terms of ordinary function calls.
>
> This back-end can be considered a short-circuit version of the IDL to Erlang gen_server back-end (see further below).

The third group consists of backends for Erlang, C, and Java. The communication between clients and servers is by the Erlang distribution protocol, facilitated by *erl_interface* and *jinterface* for C and Java, respectively.

All back-ends of the third group generate code compatible with the Erlang gen_server behavior protocol. Thus generated client code corresponds to `call()` or `cast()` of an Erlang `gen_server`. Similarly, generated server code corresponds to `handle_call()` or `handle_cast()` of an Erlang `gen_server`.

The back-ends of the third group are:

IDL to Erlang gen_server

> Client stubs and server skeletons are generated. Data types are mapped according to the IDL to Erlang mapping described in the *Orber User's Guide*.

IDL to C client

> Client stubs are generated. The mapping of data types is described further on in the C client part of this guide.

IDL to C server

> Server skeletons are generated. The mapping of data types is described further on in the C server part of this guide.

IDL to Java

> Client stubs and server skeletons are generated. The mapping of data types is described further on in the Java part of this guide.

1.1.2 Compilation of IDL Files

The IC compiler is invoked by executing the generic `erlc` compiler from a shell:

```
%> erlc +'{be,BackEnd}' File.idl
```

where `BackEnd` is according to the table below, and `File.idl` is the IDL file to be compiled.

Back-end	`BackEnd`option
IDL to CORBA	`erl_corba`
IDL to CORBA template	`erl_template`
IDL to plain Erlang	`erl_plain`
IDL to Erlang gen_server	`erl_genserv`
IDL to C client	`c_client`
IDL to C server	`c_server`
IDL to Java	`java`

Table 1.1: Compiler back-ends and options

For more details on IC compiler options consult the ic(3) manual page.

1.2 OMG IDL

1.2.1 OMG IDL - Overview

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

Since the `C` and `Java` IC backends only supports a subset of the IDL types supported by the other backends, the mapping is divided into different parts. For more information about IDL to Erlang mapping, i.e., `CORBA`, plain Erlang and generic Erlang Server, see the Orber User's Guide. How to use the plain Erlang and generic Erlang Server is found in this User's Guide.

Reserved Compiler Names and Keywords

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

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

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

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

The OMG defines a set of reserved words, shown below, for use as keywords. These may *not* be used as, for example, identifiers.

abstract	double	local	raises	typedef
any	exception	long	readonly	unsigned
attribute	enum	module	sequence	union
boolean	factory	native	short	ValueBase
case	FALSE	Object	string	valuetype
char	fixed	octet	struct	void
const	float	oneway	supports	wchar
context	in	out	switch	wstring
custom	inout	private	TRUE	
default	interface	public	truncatable	

Table 2.1: OMG IDL keywords

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

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

With Escaped Identifiers the code will look like:

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

1.3 IC Protocol

The purpose of this chapter is to explain the bits and bytes of the IC protocol, which is a composition of the Erlang distribution protocol and the Erlang/OTP gen_server protocol. If you do not intend to replace the Erlang distribution protocol, or replace the gen_server protocol, skip over this chapter.

1.3.1 Introduction

The IDL Compiler (IC) transforms Interface Definition Language (IDL) specifications files to interface code for Erlang, C, and Java. The Erlang language mapping is described in the Orber documentation, while the other mappings are described in the IC documentation (they are of course in accordance with the CORBA C and Java language mapping specifications, with some restrictions).

The most important parts of an IDL specification are the operation declarations. An operation defines what information a client provides to a server, and what information (if any) the client gets back from the server. We consider IDL operations and language mappings in section 2.

What we here call the IC protocol, is the description of messages exchanged between IC end-points (client and servers). It is valid for all IC back-ends, except the 'erl_plain' and 'erl_corba' back-ends. The IC protocol is in turn embedded into the Erlang gen_server protocol, which is described below. Finally, the gen_server protocol is embedded in the Erlang distribution protocol. Pertinent parts of that protocol is described further below.

1.3.2 Language mappings and IDL operations

IDL Operations

An IDL operation is declared as follows:

```
[oneway] RetType Op(in IType1 I1, in IType2 I2, ..., in ITypeN IN,
out OType1 O1, out OType2 O2, ..., out OTypeM OM)
N, M = 0, 1, 2, ...        (2.1.1)
```

`Op' is the operation name, RetType is the return type, and ITypei, i = 1, 2, ..., N, and OTypej, j = 1, 2, ..., M, are the `in' types and `out' types, respectively. The values I1, I2, ..., IN are provided by the caller, and the value of RetType, and the values O1, O2, ..., OM, are provided as results to the caller.

The types can be any basic types or derived types declared in the IDL specification of which the operation declaration is a part.

If the RetType has the special name `void' there is no return value (but there might still be result values O1, 02, ..., OM).

The `in' and `out' parameters can be declared in any order, but for clarity we have listed all `in' parameters before the `out' parameters in the declaration above.

If the keyword `oneway' is present, the operation is a cast, i.e. there is no confirmation of the operation, and consequently there must be no result values: RetType must be equal to `void', and M = 0 must hold.

Otherwise the operation is a call, i.e. it is confirmed (or else an exception is raised).

Note carefully that an operation declared without `oneway' is always a call, even if RetType is `void' and M = 0.

Language Mappings

There are several CORBA Language Mapping specifications. These are about mapping interfaces to various programming languages. IC supports the CORBA C and Java mapping specifications, and the Erlang language mapping specified in the Orber documentation.

Excerpt from "6.4 Basic OMG IDL Types" in the Orber User's Guide:

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

Excerpt from "6.13 Invocations of Operations" in the Orber User's Guide:

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

Hence the function that is mapped from an IDL operation to Erlang always have a return value (an Erlang function always has). That fact has influenced the IC protocol, in that there is always a return value (which is 'ok' if the return type was declared 'void').

1.3.3 IC Protocol

Given the operation declaration (2.1.1) the IC protocol maps to messages as follows, defined in terms of Erlang terms.

Call (Request/Reply, i.e. not oneway)

```
request:         Op                      atom()     N = 0
                {Op, I1, I2, ..., IN}    tuple()    N > 0
                                                    (3.1.1)

reply:          Ret                                 M = 0
                {Ret, O1, O2, ..., OM}              M > 0
                                                    (3.1.2)
```

Notice: Even if the RetType of the operation Op is declared to be 'void', a return value 'ok' is returned in the reply message. That return value is of no significance, and is therefore ignored (note however that a C server back-end returns the atom 'void' instead of 'ok').

Cast (oneway)

```
notification:    Op                      atom()     N = 0
                 {Op, I1, I2, ..., IN}   tuple()    N > 0
                                                    (3.2.1)
```

(There is of course no return message).

1.3.4 Gen_server Protocol

Most of the IC generated code deals with encoding and decoding the gen_server protocol.

Call

```
request:    {'$gen_call', {self(), Ref}, Request}   (4.1.1)

reply:      {Ref, Reply}                            (4.1.2)
```

where Request and Reply are the messages defined in the previous chapter.

Cast

```
notification:    {'$gen_cast', Notification}        (4.2.1)
```

where Notification is the message defined in the previous chapter.

1.3.5 Erlang Distribution Protocol

Messages (of interest here) between Erlang nodes are of the form:

```
      Len(4), Type(1), CtrlBin(N), MsgBin(M)          (5.1)
```

Type is equal to 112 = PASS_THROUGH.

CtrlBin and MsgBin are Erlang terms in binary form (as if created by term_to_binary/1), whence for each of them the first byte is equal to 131 = VERSION_MAGIC.

CtrlBin (of interest here) contains the SEND and REG_SEND control messages, which are binary forms of the Erlang terms

```
      {2, Cookie, ToPid} ,                            (5.2)
```

and

```
      {6, FromPid, Cookie, ToName} ,                  (5.3)
```

respectively.

The CtrlBin(N) message is read and written by erl_interface code (C), j_interface code (Java), or the Erlang distribution implementation, which are invoked from IC generated code.

The MsgBin(N) is the "real" message, i.e. of the form described in the previous section.

1.4 Using the Plain Erlang Back-end

1.4.1 Introduction

The mapping of OMG IDL to the Erlang programming language when Plain Erlang is the back-end of choice is similar to the one used in pure Erlang IDL mapping. The only difference is on the generated code and the extended use of pragmas for code generation: IDL functions are translated to Erlang module function calls.

1.4.2 Compiling the Code

In the Erlang shell type :

ic:gen(`<filename>, [{be, erl_plain}]`).

1.4.3 Writing the Implementation File

For each IDL interface `<interface name>` defined in the IDL file:

- Create the corresponding Erlang file that will hold the Erlang implementation of the IDL definitions.
- Call the implementation file after the scope of the IDL interface, followed by the suffix `_impl`.
- Export the implementation functions.

For each function defined in the IDL interface :

- Implement an Erlang function that uses as arguments in the same order, as the input arguments described in the IDL file, and returns the value described in the interface.
- When using the function, follow the mapping described in chapter 2.

1.4.4 An Example

In this example, a file "random.idl" is generates code for the plain Erlang back-end :

- Main file : "plain.idl"

```
module rmod {
  interface random {
    double produce();
    oneway void init(in long seed1, in long seed2, in long seed3);
  };
};
```

Compile the file :

```
  Erlang (BEAM) emulator version 4.9

  Eshell V4.9  (abort with ^G)
  1> ic:gen(random,[{be, erl_plain}]).
  Erlang IDL compiler version 2.5.1
  ok
  2>
```

When the file "random.idl" is compiled it produces five files: two for the top scope, two for the interface scope, and one for the module scope. The header files for top scope and interface are empty and not shown here. In this case only the file for the interface `rmod_random.erl` is important :.

- Erlang file for interface : "rmod_random.erl"

```
-module(rmod_random).


%% Interface functions
-export([produce/0, init/3]).

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

```
%% Operation: produce
%%
%%   Returns: RetVal
%%
produce() ->
    rmod_random_impl:produce().

%%--------------------------------------------------------------
%% Operation: init
%%
%%   Returns: RetVal
%%
init(Seed1, Seed2, Seed3) ->
    rmod_random_impl:init(Seed1, Seed2, Seed3).
```

The implementation file should be called `rmod_random_impl.erl` and could look like this:

```
-module('rmod_random_impl').

-export([produce/0,init/3]).


produce()  ->
  random:uniform().


init(S1,S2,S3)  ->
  random:seed(S1,S2,S3).
```

Compiling the code :

```
2> make:all().
Recompile: rmod_random
Recompile: oe_random
Recompile: rmod_random_impl
up_to_date
```

Running the example :

```
3>  rmod_random:init(1,2,3).
ok
4>  rmod_random:produce().
1.97963e-4
5>
```

1.5 Using the Erlang Generic Server Back-end

1.5.1 Introduction

The mapping of OMG IDL to the Erlang programming language when Erlang generic server is the back-end of choice is similar to the one used in the chapter 'OMG IDL Mapping'. The only difference is in the generated code, a client stub and server skeleton to an Erlang `gen_server`. Orber's User's Guide contain a more detailed description of IDL to Erlang mapping.

1.5.2 Compiling the Code

The `ic:gen/2` function can be called from the command line as follows:

```
shell> erlc "+{be, erl_genserv}" MyFile.idl
```

1.5.3 Writing the Implementation File

For each IDL interface `<interface name>` defined in the IDL file :

- Create the corresponding Erlang file that will hold the Erlang implementation of the IDL definitions.
- Call the implementation file after the scope of the IDL interface, followed by the suffix `_impl`.
- Export the implementation functions.

For each function defined in the IDL interface :

- Implement an Erlang function that uses as arguments in the same order, as the input arguments described in the IDL file, and returns the value described in the interface.
- When using the function, follow the mapping described in chapter 2.

1.5.4 An Example

In this example, a file `random.idl` generates code for the Erlang gen_server back-end:

```
// Filename random.idl
module rmod {

  interface random {
    // Generate a new random number
    double produce();
    // Initialize random generator
    oneway void init(in long seed1, in long seed2, in long seed3);

  };
};
```

When the file "random.idl" is compiled (e.g., `shell> erlc "+{be, erl_genserv}" random.idl`) five files are produced; two for the top scope, two for the interface scope, and one for the module scope. The header files for top scope and interface are empty and not shown here. In this case, the stub/skeleton file `rmod_random.erl` is the most important. This module exports two kinds of operations:

- *Administrative* - used when, for example, creating and terminating the server.
- *IDL dependent* - operations defined in the IDL specification. In this case, `produce` and `init`.

Administrative Operations

To create a new server instance, one of the following functions should be used:

- *oe_create/0/1/2* - create a new instance of the object. Accepts `Env` and `RegName`, in that order, as parameters. The former is passed uninterpreted to the initialization operation of the call-back module, while the latter must be as the `gen_server` parameter `ServerName`. If `Env` is left out, an empty list will be passed.
- *oe_create_link/0/1/2* - similar to `oe_create/0/1/2`, but create a linked server.
- *typeID/0* - returns the scooped id compliant with the OMG standard. In this case the string `"IDL:rmod/random:1.0"`.
- *stop/1* - asynchronously terminate the server. The required argument is the return value from any of the start functions.

IDL Dependent Operations

Operations can either be synchronous or asynchronous (i.e., `oneway`). These are, respectively, mapped to `gen_server:call/2/3` and `gen_server:cast/2`. Consult the `gen_server` documentation for valid return values.

The IDL dependent operations in this example are listed below. The first argument must be the whatever the create operation returned.

- *init(ServerReference, Seed1, Seed2, Seed3)* - initialize the random number generator.
- *produce(ServerReference)* - generate a new random number.

If the compile option `timeout` is used a timeout must be added (e.g., `produce(ServerReference, 5000)`). For more information, see the `gen_server` documentation.

Implementation Module

The implementation module shall, unless the compile option `impl` is used, be named `rmod_random_impl.erl`. and could look like this:

```
-module('rmod_random_impl').
%% Mandatory gen_server operations
-export([init/1, terminate/2, code_change/3]).
%% Add if 'handle_info' compile option used
-export([handle_info/2]).
%% API defined in IDL specification
-export([produce/1,init/4]).

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

terminate(From, Reason) ->
    ok.

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

%% Optional
handle_info(Info, State) ->
    {noreply, NewState}.

%% IDL specification
produce(State)  ->
    case catch random:uniform() of
     {'EXIT',_} ->
         {stop, normal, "random:uniform/0 - EXIT", State};
```

```
     RUnif ->
            {reply, RUnif, State}
     end.


init(State, S1, S2, S3)  ->
    case catch random:seed(S1, S2, S3) of
     {'EXIT',_} ->
         {stop, normal, State};
     _ ->
            {noreply, State}
    end.
```

Compile the code and run the example:

```
1> make:all().
Recompile: rmod_random
Recompile: oe_random
Recompile: rmod_random_impl
up_to_date
2> {ok,R} = rmod_random:oe_create().
{ok,<0.30.0>}
3> rmod_random:init(R, 1, 2, 3).
ok
4> rmod_random:produce(R).
1.97963e-4
5>
```

1.6 IDL to C mapping

1.6.1 Introduction

The IC C mapping (used by the C client and C server back-ends) follows the *OMG C Language Mapping Specification*.

The C mapping supports the following:

- All OMG IDL basic types except `long double` and `any`.
- All OMG IDL constructed types.
- OMG IDL constants.
- Operations with passing of parameters and receiving of results. `inout` parameters are not supported.

The following is not supported:

- Access to attributes.
- User defined exceptions.
- User defined objects.

1.6.2 C Mapping Characteristics

Reserved Names

The IDL compiler reserves all identifiers starting with `OE_` and `oe_` for internal use.

Scoped Names

The C programmer must always use the global name for a type, constant or operation. The C global name corresponding to an OMG IDL global name is derived by converting occurrences of "::" to underscore, and eliminating the leading "::". So, for example, an operation `op1` defined in interface `I1` which is defined in module `M1` would be written as `M1::I1::op1` in IDL and as `M1_I1_op1` in C.

> **Warning:**
>
> If underscores are used in IDL names it can lead to ambiguities due to the name mapping described above, therefore it is advisable to avoid underscores in identifiers.

Generated Files

Two files will be generated for each scope. One set of files will be generated for each module and each interface scope. An extra set is generated for those definitions at top level scope. One of the files is a header file(`.h`), and the other file is a C source code file (`.c`). In addition to these files a number of C source files will be generated for type encodings, they are named according to the following template: `oe_code_<type>.c`.

For example:

```
// IDL, in the file "spec.idl"
module m1 {

    typedef sequence<long> lseq;

    interface i1 {
        ...
    };
    ...
};
```

XXX This is C client specific. Will produce the files `oe_spec.h` and `oe_spec.c` for the top scope level. Then the files `m1.h` and `m1.c` for the module `m1` and files `m1_i1.h` and `m1_i1.c` for the interface `i1`. The typedef will produce `oe_code_m1_lseq.c`.

The header file contains type definitions for all `struct` types and sequences and constants in the IDL file. The c file contains all operation stubs if the the scope is an interface.

In addition to the scope-related files a C source file will be generated for encoding operations of all `struct` and sequence types.

1.6.3 Basic OMG IDL Types

The mapping of basic types is as follows.

OMG IDL type	*C type*	*Mapped to C type*
float	CORBA_float	float
double	CORBA_double	double

short	CORBA_short	short
unsigned short	CORBA_unsigned_short	unsigned short
long	CORBA_long	long
long long	CORBA_long_long	long
unsigned long	CORBA_unsigned_long	unsigned long
unsigned long long	CORBA_unsigned_long_long	unsigned long
char	CORBA_char	char
wchar	CORBA_wchar	unsigned long
boolean	CORBA_boolean	unsigned char
octet	CORBA_octet	char
any	Not supported	
long double	Not supported	
Object	Not supported	
void	void	void

Table 6.1: OMG IDL Basic Types

XXX Note that several mappings are not according to OMG C Language mapping.

1.6.4 Constructed OMG IDL Types

Constructed types have mappings as shown in the following table.

OMG IDL type	Mapped to C type
string	CORBA_char*
wstring	CORBA_wchar*
struct	struct
union	union
enum	enum
sequence	struct (see below)

array	array

Table 6.2: OMG IDL Constructed Types

An OMG IDL sequence (an array of variable length),

```
// IDL
typedef sequence <IDL_TYPE> NAME;
```

is mapped to a C struct as follows:

```
/* C */
typedef struct {
  CORBA_unsigned_long _maximum;
  CORBA_unsigned_long _length;
  C_TYPE* _buffer;
} C_NAME;
```

where `C_TYPE` is the mapping of `IDL_TYPE`, and where `C_NAME` is the scoped name of `NAME`.

1.6.5 OMG IDL Constants

An IDL constant is mapped to a C constant through a C `#define` macro, where the name of the macro is scoped. Example:

```
// IDL
module M1 {
    const long c1 = 99;
};
```

results in the following:

```
/* C */
#define M1_c1 99
```

1.6.6 OMG IDL Operations

An OMG IDL operation is mapped to C function. Each C operation function has two mandatory parameters: a first parameter of *interface object* type, and a last parameter of *environment* type.

In a C operation function the the `in` and `out` parameters are located between the first and last parameters described above, and they appear in the same order as in the IDL operation declaration.

Notice that `inout` parameters are not supported.

The return value of an OMG IDL operation is mapped to a corresponding return value of the C operation function.

Mandatory C operation function parameters:

- `CORBA_Object oe_obj` - the first parameter of a C operation function. This parameter is required by the *OMG C Language Mapping Specification*, but in the current implementation there is no particular use for it.
- `CORBA_Environment* oe_env` - the last parameter of a C operation function. The parameter is defined in the C header file `ic.h` and has the following public fields:
  - `CORBA_Exception_type _major` - indicates if an operation invocation was successful which will be one of the following:
    - CORBA_NO_EXCEPTION
    - CORBA_SYSTEM_EXCEPTION
  - int *_fd* - a file descriptor returned from *erl_connect* function.
  - int *_inbufsz* - size of input buffer.
  - char* *_inbuf* - pointer to a buffer used for input.
  - int *_outbufsz* - size of output buffer.
  - char* *_outbuf* - pointer to a buffer used for output.
  - int *_memchunk* - expansion unit size for the output buffer. This is the size of memory chunks in bytes used for increasing the output in case of buffer expansion. The value of this field must be always set to >= 32, should be at least 1024 for performance reasons.
  - char *regname[256]* - a registered name for a process.
  - erlang_pid* *_to_pid* - an Erlang process identifier, is only used if the registered_name parameter is the empty string.
  - erlang_pid* *_from_pid* - your own process id so the answer can be returned

  Beside the public fields, other private fields are internally used but are not mentioned here.

Example:

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

Is mapped to the following C functions

```
/* C */
CORBA_long i1_op1(i1 oe_obj, CORBA_long a, CORBA_Environment* oe_env)
{
    ...
}
CORBA_long i1_op2(i1 oe_obj, CORBA_char* s, CORBA_long *count,
CORBA_Environment* oe_env)
{
    ...
}
```

Operation Implementation

There is no standard CORBA mapping for the C-server side, as it is implementation-dependent but built in a similar way. The current server side mapping is different from the client side mapping in several ways:

- Argument mappings
- Result values
- Structure
- Usage
- Exception handling

1.6.7 Exceptions

Although exception mapping is not implemented, the stubs will generate CORBA system exceptions in case of operation failure. Thus, the only exceptions propagated by the system are built in system exceptions.

1.6.8 Access to Attributes

Not Supported

1.6.9 Summary of Argument/Result Passing for the C-client

The user-defined parameters can only be `in` or `out` parameters, as `inout` parameters are not supported.

This table summarize the types a client passes as arguments to a stub, and receives as a result.

OMG IDL type	In	Out	Return
short	CORBA_short	CORBA_short*	CORBA_short
long	CORBA_long	CORBA_long*	CORBA_long
long long	CORBA_long_long	CORBA_long_long*	CORBA_long_long
unsigned short	CORBA_unsigned_short	CORBA_unsigned_short*	CORBA_unsigned_short
unsigned long	CORBA_unsigned_long	CORBA_unsigned_long*	CORBA_unsigned_long
unsigned long long	CORBA_unsigned_long_long	CORBA_unsigned_long_long*	CORBA_unsigned_long_long
float	CORBA_float	CORBA_float*	CORBA_float
double	CORBA_double	CORBA_double*	CORBA_double
boolean	CORBA_boolean	CORBA_boolean*	CORBA_boolean
char	CORBA_char	CORBA_char*	CORBA_char
wchar	CORBA_wchar	CORBA_wchar*	CORBA_wchar
octet	CORBA_octet	CORBA_octet*	CORBA_octet
enum	CORBA_enum	CORBA_enum*	CORBA_enum

struct, fixed	struct*	struct*	struct
struct, variable	struct*	struct**	struct*
union, fixed	union*	union*	union
union, variable	union*	union**	union*
string	CORBA_char*	CORBA_char**	CORBA_char*
wstring	CORBA_wchar*	CORBA_wchar**	CORBA_wchar*
sequence	sequence*	sequence**	sequence*
array, fixed	array	array	array_slice*
array, variable	array	array_slice**	array_slice*

Table 6.3: Basic Argument and Result passing

A client is responsible for providing storage of all arguments passed as *in* arguments.

OMG IDL type	Out	Return
short	1	1
long	1	1
long long	1	1
unsigned short	1	1
unsigned long	1	1
unsigned long long	1	1
float	1	1
double	1	1
boolean	1	1
char	1	1
wchar	1	1
octet	1	1
enum	1	1
struct, fixed	1	1

struct, variable	2	2
string	2	2
wstring	2	2
sequence	2	2
array, fixed	1	3
array, variable	3	3

Table 6.4: Client argument storage responsibility

Case	Description
1	Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself.
2	The caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. The caller is responsible for releasing the returned storage. Following completion of a request, the caller is not allowed to modify any values in the returned storage. To do so the caller must first copy the returned instance into a new instance, then modify the new instance.
3	The caller allocates a pointer to an array slice which has all the same dimensions of the original array except the first, and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the array. The caller is responsible for releasing the returned storage. Following completion of a request, the caller is not allowed to modify any values in the returned storage. To do so the caller must first copy the returned instance into a new instance, then modify the new instance.

Table 6.5: Argument passing cases

The returned storage in case 2 and 3 is allocated as one block of memory so it is possible to deallocate it with one call of CORBA_free.

1.6.10 Supported Memory Allocation Functions

- *CORBA_Environment* can be allocated from the user by calling *CORBA_Environment_alloc()*.

  The interface for this function is

  ```
  CORBA_Environment *CORBA_Environment_alloc(int inbufsz, int outbufsz);
  ```

where :

- *inbufsz* is the desired size of input buffer
- *outbufsz* is the desired size of output buffer
- return value is a *pointer* to an allocated and initialized *CORBA_Environment* structure

- Strings can be allocated from the user by calling *CORBA_string_alloc()*.

  The interface for this function is

  ```
  CORBA_char *CORBA_string_alloc(CORBA_unsigned_long len);
  ```

  where :

  - *len* is the length of the string to be allocated.

Thus far, no other type allocation function is supported.

1.6.11 Special Memory Deallocation Functions

- `void CORBA_free(void *storage)`

  This function will free storage allocated by the stub.
- `void CORBA_exception_free(CORBA_environment *ev)`

  This function will free storage allocated under exception propagation.

1.6.12 Exception Access Functions

- `CORBA_char *CORBA_exception_id(CORBA_Environment *ev)`

  This function will return raised exception identity.
- `void *CORBA_exception_value(CORBA_Environment *ev)`

  This function will return the value of a raised exception.

1.6.13 Special Types

- The erlang binary type has some special features.

  While the `erlang::binary` idl type has the same C-definition as a generated sequence of octets :

```
module erlang
{

....

// an erlang binary
typedef sequence<octet> binary;

};
```

  it provides a way on sending trasparent data between C and Erlang.

  The C-definition (ic.h) for an erlang binary is :

```
typedef struct {
CORBA_unsigned_long _maximum;
```

```
    CORBA_unsigned_long _length;
    CORBA_octet* _buffer;
    } erlang_binary;                        /* ERLANG BINARY */
```

The differences (between `erlang::binary` and `sequence< octet >`) are :

- on the erlang side the user is sending/receiving typical built in erlang binaries, using `term_to_binary() / binary_to_term()` to create / extract binary structures.
- no encoding/decoding functions are generated
- the underlying protocol is more efficient than usual sequences of octets

The erlang binary IDL type is defined in `erlang.idl`, while its C definition is located in the `ic.h` header file, both in the `IC-< vsn >/include` directory. The user will have to include the file `erlang.idl` in order to use the `erlang::binary` type.

1.6.14 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 the stack.

```
// The source IDL file: stack.idl

struct s {
      long l;
      string s;
};

interface stack {
    void push(in s val);
    s pop();
};
```

When this file is compiled it produces four files, two for the top scope and two for the stack interface scope. The important parts of the generated C code for the stack API is shown below.

stack.c

```
void push(stack oe_obj, s val, CORBA_Environment* oe_env) {
  ...
}


s* pop(stack oe_obj, CORBA_Environment* oe_env) {
  ...
}
```

oe_stack.h

```
#ifndef OE_STACK_H
#define OE_STACK_H
```

```
/*------------------------------------------------------------
 * Struct definition: s
 */
typedef struct {
   long l;
   char *s;
} s;



#endif
```

stack.h just contains an include statement of `oe_stack.h`.

oe_code_s.c

```
int oe_sizecalc_s(CORBA_Environment
      *oe_env, int* oe_size_count_index, int* oe_size) {
  ...
}

int oe_encode_s(CORBA_Environment *oe_env, s* oe_rec) {
  ...
}

int oe_decode_s(CORBA_Environment *oe_env, char *oe_first,
                int* oe_outindex, s *oe_out) {
  ...
}
```

The only files that are really important are the `.h` files and the stack.c file.

1.7 The C Client Back-end

1.7.1 Introduction

With the option `{be, c_client}` the IDL Compiler generates C client stubs according to the IDL to C mapping, on top of the Erlang distribution and gen_server protocols.

The developer has to write additional code, that together with the generated C client stubs, form a hidden Erlang node. That additional code uses `erl_interface` functions for defining the hidden node, and for establishing connections to other Erlang nodes.

1.7.2 Generated Stub Files

The generated stub files are:

- For each IDL interface, a C source file, the name of which is `<Scoped Interface Name>.c`. Each operation of the IDL interface is mapped to a C function (with scoped name) in that file;
- C source files that contain functions for type conversion, memory allocation, and data encoding/decoding;
- C header files that contain function prototypes and type definitions.

All C functions are exported (i.e. not declared static).

1.7.3 C Interface Functions

For each IDL operation a C interface function is generated, the prototype of which is:

`<Return Value> <Scoped Function Name>(<Interface Object> oe_obj, <Parameters>, CORBA_Environment *oe_env);`

where

- `<Return Value>` is the value to be returned as defined by the IDL specification;
- `<Interface Object> oe_obj` is the client interface object;
- `<Parameters>` is a list of parameters of the operation, defined in the same order as defined by the IDL specification;
- `CORBA_Environment *oe_env` is a pointer to the current client environment. It contains the current file descriptor, the current input and output buffers, etc. For details see *CORBA_Environment C Structure*.

1.7.4 Generating, Compiling and Linking

To generate the C client stubs type the following in an appropriate shell:

`erlc -I ICROOT/include "+{be, c_client}" File.idl`,

where `ICROOT` is the root of the IC application. The `-I ICROOT/include` is only needed if `File.idl` refers to `erlang.idl`.

When compiling a generated C stub file, the directories `ICROOT/include` and `EICROOT/include`, have to be specified as include directories, where `EIROOT` is the root directory of the Erl_interface application.

When linking object files the `EIROOT/lib` and `ICROOT/priv/lib` directories have to be specified.

1.7.5 An Example

In this example the IDL specification file "random.idl" is used for generating C client stubs (the file is contained in the IC `/examples/c-client` directory):

```
module rmod {

  interface random {

    double produce();

    oneway void init(in long seed1, in long seed2, in long seed3);

  };

};
```

Generate the C client stubs:

```
erlc '+{be, c_client}' random.idl
Erlang IDL compiler version X.Y.Z
```

Six files are generated.

Compile the C client stubs:

Please read the `ReadMe` file att the `examples/c-client` directory

In the same directory you can find all the code for this example.

In particular you will find the `client.c` file that contains all the additional code that must be written to obtain a complete client.

In the `examples/c-client` directory you will also find source code for an Erlang server, which can be used for testing the C client.

1.8 The C Server Back-end

1.8.1 Introduction

With the option `{be, c_server}` the IDL Compiler generates C server skeletons according to the IDL to C mapping, on top of the Erlang distribution and gen_server protocols.

The developer has to write additional code, that together with the generated C server skeletons, form a hidden Erlang node. That additional code contains implementations of call-back functions that implement the true server functionality, and also code uses `erl_interface` functions for defining the hidden node and for establishing connections to other Erlang nodes.

1.8.2 Generated Stub Files

The generated stub files are:

- For each IDL interface, a C source file, the name of which is `<Scoped Interface Name>__s.c`. Each operation of the IDL interface is mapped to a C function (with scoped name) in that file;
- C source files that contain functions for type conversion, memory allocation, and data encoding/decoding;
- C header files that contain function prototypes and type definitions.

All C functions are exported (i.e. not declared static).

1.8.3 C Skeleton Functions

For each IDL operation a C skeleton function is generated, the prototype of which is `int <Scoped Function Name>__exec(<Interface Object> oe_obj, CORBA_Environment *oe_env)`, where `<Interface Object>`, and `CORBA_Environment` are of the same type as for the generated C client stubs code.

Each `<Scoped Function Name>__exec()` function calls the call-back function

```
<Scoped Function Name>_rs* <Scoped Function Name>__cb(<Interface Object> oe_obj,
<Parameters>, CORBA_Environment *oe_env)
```

where the arguments are of the same type as those generated for C client stubs.

The return value `<Scoped Function Name>_rs*` is a pointer to a function with the same signature as the call-back function `<Scoped Function Name>_cb`, and is called after the call-back function has been evaluated (provided that the pointer is not equal to `NULL`).

1.8.4 The Server Loop

The developer has to implement code for establishing connections with other Erlang nodes, code for call-back functions and restore functions.

In addition, the developer also has to implement code for a server loop, that receives messages and calls the relevant `__exec` function. For that purpose the IC library function `oe_server_receive()` function can be used.

1.8.5 Generating, Compiling and Linking

To generate the C server skeletons type the following in an appropriate shell:

`erlc -I ICROOT/include "+{be, c_server}" File.idl`,

where `ICROOT` is the root of the IC application. The `-I ICROOT/include` is only needed if `File.idl` refers to `erlang.idl`.

When compiling a generated C skeleton file, the directories `ICROOT/include` and `EICROOT/include`, have to be specified as include directories, where `EIROOT` is the root directory of the Erl_interface application.

When linking object files the `EIROOT/lib` and `ICROOT/priv/lib` directories have to be specified.

1.8.6 An Example

In this example the IDL specification file "random.idl" is used for generating C server skeletons (the file is contained in the IC `/examples/c-server` directory):

```
module rmod {

  interface random {

    double produce();

    oneway void init(in long seed1, in long seed2, in long seed3);

  };

};
```

Generate the C server skeletons:

```
erlc '+{be, c_server}' random.idl
Erlang IDL compiler version X.Y.Z
```

Six files are generated.

Compile the C server skeletons:

Please read the `ReadMe` file in the `examples/c-server` directory.

In the same directory you can find all the code for this example. In particular you will find the `server.c` file that contains all the additional code that must be written to obtain a complete server.

In the `examples/c-server` directory you will also find source code for an Erlang client, which can be used for testing the C server.

1.9 CORBA_Environment C Structure

This chapter describes the CORBA_Environment C structure.

1.9.1 C Structure

Here is the complete definition of the CORBA_Environment C structure, defined in file "ic.h" :

```
```

```
/* Environment definition */
typedef struct {

  /*----- CORBA compatibility part ------------------------*/
  /* Exception tag, initially set to CORBA_NO_EXCEPTION ---*/
  CORBA_exception_type   _major;

  /*----- External Implementation part - initiated by the user ---*/
  /* File descriptor                                              */
  int                    _fd;
  /* Size of input buffer                                         */
  int                    _inbufsz;
  /* Pointer to always dynamically allocated buffer for input     */
  char                   *_inbuf;
  /* Size of output buffer                                        */
  int                    _outbufsz;
  /* Pointer to always dynamically allocated buffer for output    */
  char                   *_outbuf;
 /* Size of memory chunks in bytes, used for increasing the output
    buffer, set to >= 32, should be around >= 1024 for performance
    reasons                                                      */
 int                    _memchunk;
 /* Pointer for registered name                                  */
  char                   _regname[256];
 /* Process identity for caller                                  */
  erlang_pid             *_to_pid;
  /* Process identity for callee                                 */
  erlang_pid             *_from_pid;

  /*- Internal Implementation part - used by the server/client ---*/
  /* Index for input buffer                                       */
  int                    _iin;
  /* Index for output buffer                                      */
  int                    _iout;
  /* Pointer for operation name                                   */
  char                   _operation[256];
   /* Used to count parameters                                    */
  int                    _received;
  /* Used to identify the caller                                  */
  erlang_pid             _caller;
 /* Used to identify the call                                    */
  erlang_ref             _unique;
  /* Exception id field                                           */
  CORBA_char            *_exc_id;
  /* Exception value field                                        */
   void                  *_exc_value;


} CORBA_Environment;
```

The structure is divided into three parts:

- The CORBA Compatibility part, demanded by the standard OMG IDL mapping v2.0.
- The external implementation part used for generated client/server code.
- The internal part useful for those who wish to define their own functions.

1.9.2 The CORBA Compatibility Part

Contains only one field `_major` defined as a CORBA_Exception_type. The CORBA_Exception type is an integer which can be one of:

- *CORBA_NO_EXCEPTION*, by default equal to 0, can be set by the application programmer to another value.

- *CORBA_SYSTEM_EXCEPTION*, by default equal to -1, can be set by the application programmer to another value.

The current definition of these values are:

```
#define CORBA_NO_EXCEPTION      0
#define CORBA_SYSTEM_EXCEPTION -1
```

1.9.3 The External Part

This part contains the following fields:

- int *_fd* - a file descriptor returned from erl_connect. Used for connection setting.
- char* *_inbuf* - pointer to a buffer used for input. Buffer size checks are done under runtime that prevent buffer overflows. This is done by expanding the buffer to fit the input message. In order to allow buffer reallocation, the output buffer must always be dynamically allocated. The pointer value can change under runtime in case of buffer reallocation.
- int *_inbufsz* - start size of input buffer. Used for setting the input buffer size under initialization of the Erl_Interface function ei_receive_encoded/5. The value of this field can change under runtime in case of input buffer expansion to fit larger messages
- int *_outbufsz* - start size of output buffer. The value of this field can change under runtime in case of input buffer expansion to fit larger messages
- char* *_outbuf* - pointer to a buffer used for output. Buffer size checks prevent buffer overflows under runtime, by expanding the buffer to fit the output message in cases of lack of space in buffer. In order to allow buffer reallocation, the output buffer must always be dynamically allocated. The pointer value can change under runtime in case of buffer reallocation.
- int *_memchunk* - expansion unit size for the output buffer. This is the size of memory chunks in bytes used for increasing the output in case of buffer expansion. The value of this field must be always set to >= 32, should be at least 1024 for performance reasons.
- char *regname[256]* - a registered name for a process.
- erlang_pid* *_to_pid* - an Erlang process identifier, is only used if the registered_name parameter is the empty string.
- erlang_pid* *_from_pid* - your own process id so the answer can be returned.

1.9.4 The Internal Part

This part contains the following fields:

- int *_iin* - Index for input buffer. Initially set to zero. Updated to agree with the length of the received encoded message.
- int *_iout* - Index for output buffer Initially set to zero. Updated to agree with the length of the message encoded to the communication counterpart.
- char *_operation[256]* - Pointer for operation name. Set to the operation to be called.
- int *_received* - Used to count parameters. Initially set to zero.
- erlang_pid *_caller* - Used to identify the caller. Initiated to a value that identifies the caller.
- erlang_ref *_unique* - Used to identify the call. Set to a default value in the case of generated functions.
- CORBA_char* *_exc_id* - Exception id field. Initially set to NULL to agree with the initial value of _major (CORBA_NO_EXCEPTION).
- void* *_exc_value* - Exception value field Initially set to *NULL* to agree with the initial value of _major (CORBA_NO_EXCEPTION).

The advanced user who defines his own functions has to update/support these values in a way similar to how they are updated in the generated code.

1.9.5 Creating and Initiating the CORBA_Environment Structure

There are two ways to set the CORBA_Environment structure:

- Manually

  The following default values must be set to the CORBA_Environment **ev* fields, when buffers for input/output should have the size *inbufsz*/ *outbufsz*:

  - *ev->_inbufsz = inbufsz*;

    The value for this field can be between 0 and maximum size of a signed integer.
  - *ev->_inbuf* = malloc(*inbufsz*);

    The size of the allocated buffer must be equal to the value of its corresponding index, _inbufsz.
  - *ev->_outbufsz = outbufsz*;

    The value for this field can be between 0 and maximum size of a signed integer.
  - *ev->_outbuf* = malloc(*outbufsz*);

    The size of the allocated buffer must be equal to the value of its corresponding index, _outbufsz.
  - *ev->_memchunk = __OE_MEMCHUNK__*;

    Please note that __OE_MEMCHUNK__ is equal to *1024*, you can set this value to a value bigger than 32 yourself.
  - *ev->_to_pid = NULL*;
  - *ev->_from_pid = NULL*;

- By using the *CORBA_Environment_alloc*/2 function.

  The CORBA_Environment_alloc function is defined as:

```
CORBA_Environment *CORBA_Environment_alloc(int inbufsz,
                                           int outbufsz);
```

  where:

  - *inbufsz* is the desired size of input buffer
  - *outbufsz* is the desired size of output buffer
  - return value is a *pointer* to an allocated and initialized *CORBA_Environment* structure.

  This function will set all needed default values and allocate buffers equal to the values passed, but will not allocate space for the _to_pid and _from_pid fields.

  To free the space allocated by CORBA_Environment_alloc/2:

  - First call CORBA_free for the input and output buffers.
  - After freeing the buffer space, call CORBA_free for the CORBA_Environment space.

Note:

Remember to set the fields *_fd*, *_regname*, **_to_pid* and/or **_from_pid* to the appropriate application values. These are not automatically set by the stubs.

Warning:

Never assign static buffers to the buffer pointers. Never set the *_memchunk* field to a value less than *32*.

1.9.6 Setting System Exceptions

If the user wishes to set own system exceptions at critical positions on the code, it is strongly recommended to use one of the current values:

- CORBA_NO_EXCEPTION upon success. The value of the _exc_id field should be then set to NULL. The value of the _exc_value field should be then set to NULL.
- CORBA_SYSTEM_EXCEPTION upon system failure. The value of the _exc_id field should be then set to one of the values defined in "ic.h" :

```
#define UNKNOWN          "UNKNOWN"
#define BAD_PARAM        "BAD_PARAM"
#define NO_MEMORY        "NO_MEMORY"
#define IMPL_LIMIT       "IMP_LIMIT"
#define COMM_FAILURE     "COMM_FAILURE"
#define INV_OBJREF       "INV_OBJREF"
#define NO_PERMISSION    "NO_PERMISSION"
#define INTERNAL         "INTERNAL"
#define MARSHAL          "MARSHAL"
#define INITIALIZE       "INITIALIZE"
#define NO_IMPLEMENT     "NO_IMPLEMENT"
#define BAD_TYPECODE     "BAD_TYPECODE"
#define BAD_OPERATION    "BAD_OPERATION"
#define NO_RESOURCES     "NO_RESOURCES"
#define NO_RESPONSE      "NO_RESPONSE"
#define PERSIST_STORE    "PERSIST_STORE"
#define BAD_INV_ORDER    "BAD_INV_ORDER"
#define TRANSIENT        "TRANSIENT"
#define FREE_MEM         "FREE_MEM"
#define INV_IDENT        "INV_IDENT"
#define INV_FLAG         "INV_FLAG"
#define INTF_REPOS       "INTF_REPOS"
#define BAD_CONTEXT      "BAD_CONTEXT"
#define OBJ_ADAPTER      "OBJ_ADAPTER"
#define DATA_CONVERSION  "DATA_CONVERSION"
#define OBJ_NOT_EXIST    "OBJECT_NOT_EXIST"
```

The value of the _exc_value field should be then set to a string that explains the problem in an informative way. The user should use the functions CORBA_exc_set/4 and CORBA_exception_free/1 to free the exception. The user has to use CORBA_exception_id/1 and CORBA_exception_value/1 to access exception information. Prototypes for these functions are declared in "ic.h"

1.10 IDL to Java language Mapping

1.10.1 Introduction

This chapter describes the mapping of OMG IDL constructs to the Java programming language for the generation of native Java - Erlang communication.

This language mapping defines the following:

- All OMG IDL basic types
- All OMG IDL constructed types
- References to constants defined in OMG IDL
- Invocations of operations, including passing of parameters and receiving of result
- Access to attributes

1.10.2 Specialties in the Mapping

Names Reserved by the Compiler

The IDL compiler reserves all identifiers starting with `OE_` and `oe_` for internal use.

1.10.3 Basic OMG IDL Types

The mapping of basic types are according to the standard. All basic types have a special Holder class.

OMG IDL type	Java type
float	float
double	double
short	short
unsigned short	short
long	int
long long	long
unsigned long	long
unsigned long long	long
char	char
wchar	char
boolean	boolean
octet	octet
string	java.lang.String
wstring	java.lang.String

any	Any
long double	Not supported
Object	Not supported
void	void

Table 10.1: OMG IDL basic types

1.10.4 Constructed OMG IDL Types

All constructed types are according to the standard with three (3) major exceptions.

- The IDL Exceptions are not implemented in this Java mapping.
- The functions used for read/write to streams, defined in `Helper` functions are named unmarshal (instead for read) and marshal (instead for write).
- The streams used in `Helper` functions are `OtpInputStream` for input and `OtpOutputStream` for output.

1.10.5 Mapping for Constants

Constants are mapped according to the standard.

1.10.6 Invocations of Operations

Operation invocation is implemented according to the standard. The implementation is in the class `_<nterfacename>Stub.java` which implements the interface in `<nterfacename>.java`.

```
test._iStub client;

client.op(10);
```

Operation Implementation

The server is implemented through extension of the class `_<nterfacename>ImplBase.java` and implementation of all the methods in the interface.

```
public class server extends test._iImplBase {

  public void op(int i) throws java.lang.Exception {
    System.out.println("Received call op()");
    o.value = i;
    return i;
  }

}
```

1.10.7 Exceptions

While exception mapping is not implemented, the stubs will generate some Java exceptions in case of operation failure. No exceptions are propagated through the communication.

1.10.8 Access to Attributes

Attributes are supported according to the standard.

1.10.9 Summary of Argument/Result Passing for Java

All types (`in`, `out` or `inout`) of user defined parameters are supported in the Java mapping. This is also the case in the Erlang mappings but *not* in the C mapping. `inout` parameters are not supported in the C mapping so if you are going to do calls to or from a C program `inout` cannot be used in the IDL specifications.

`out` and `inout` parameters must be of Holder types. There is a jar file (`ic.jar`) with Holder classes for the basic types in the `ic` application. This library is in the directory `$OTPROOT/lib/ic_<version number>/priv`.

1.10.10 Communication Toolbox

The generated client and server stubs use the classes defined in the `jinterface` package to communicate with other nodes. The most important classes are :

- `OtpInputStream` which is the stream class used for incoming message storage
- `OtpOutputStream` which is the stream class used for outgoing message storage
- `OtpErlangPid` which is the process identification class used to identify processes inside a java node.

  The recommended constructor function for the OtpErlangPid is `OtpErlangPid(String node, int id, int serial, int creation)` where :

  - `String node`, is the name of the node where this process runs.
  - `int id`, is the identification number for this identity.
  - `int serial`, internal information, must be an 18-bit integer.
  - `int creation`, internal information, must have value in range 0..3.
- `OtpConnection` which is used to define a connection between nodes.

  While the connection object is stub side constructed in client stubs, it is returned after calling the `accept` function from an OtpErlangServer object in server stubs. The following methods used for node connection :

  - `OtpInputStream receiveBuf()`, which returns the incoming streams that contain the message arrived.
  - `void sendBuf(OtpErlangPid client, OtpOutputStream reply)`, which sends a reply message (in an OtpOutputStream form) to the client node.
  - `void close()`, which closes a connection.
- `OtpServer` which is used to define a server node.

  The recommended constructor function for the OtpServer is :

  - `OtpServer(String node, String cookie)`. where :
    - `node` is the requested name for the new java node, represented as a String object.

- `cookie` is the requested cookie name for the new java node, represented as a String object.

The following methods used for node registration and connection acceptance :

- `boolean publishPort()`, which registers the server node to `epmd` daemon.
- `OtpConnection accept()`, which waits for a connection and returns the OtpConnection object which is unique for each client node.

1.10.11 The Package com.ericsson.otp.ic

The package *com.ericsson.otp.ic* contains a number of java classes specially designed for the IC generated java-backends :

- Standard java classes defined through OMG-IDL java mapping :
  - *BooleanHolder*
  - *ByteHolder*
  - *CharHolder*
  - *ShortHolder*
  - *IntHolder*
  - *LongHolder*
  - *FloatHolder*
  - *DoubleHolder*
  - *StringHolder*
  - *Any*, *AnyHelper*, *AnyHolder*
  - *TypeCode*
  - *TCKind*
- Implementation-dependant classes :
  - *Environment*
  - *Holder*
- Erlang compatibility classes :
  - *Pid*, *PidHelper*, *PidHolder*

    The Pid class originates from `OtpErlangPid` and is used to represent the Erlang built-in `pid` type, a process's identity. PidHelper and PidHolder are helper respectively holder classes for Pid.
  - *Ref*, *RefHelper*, *RefHolder*

    The Ref class originates from `OtpErlangRef` and is used to represent the Erlang built-in `ref` type, an Erlang reference. RefHelper and RefHolder are helper respectively holder classes for Ref.
  - *Port*, *PortHelper*, *PortHolder*

    The Port class originates from `OtpErlangPort` and is used to represent the Erlang built-in `port` type, an Erlang port. PortHelper and PortHolder are helper respectively holder classes for Port.
  - *Term*, *TermHelper*, *TermHolder*

    The Term class originates from `Any` and is used to represent the Erlang built-in `term` type, an Erlang term. TermHelper and TermHolder are helper respectively holder classes for Term.

To use the Erlang build-in classes, you will have to include the file `erlang.idl` located under `$OTPROOT/lib/ic/include`.

1.10.12 The Term Class

The `Term` class is intended to represent the Erlang term generic type. It extends the `Any` class and it is basically used in the same way as in the Any type.

The big difference between Term and Any is the use of `guard` methods instead of `TypeCode` to determine the data included in the Term. This is especially true when the Term's value class cannot be determined at compilation time. The guard methods found in Term :

- `boolean isAtom()` returns `true` if the Term is an OtpErlangAtom, `false` otherwise
- `boolean isConstant()` returns `true` if the Term is neither an OtpErlangList nor an OtpErlangTuple, `false` otherwise
- `boolean isFloat()` returns `true` if the Term is an OtpErlangFloat, `false` otherwise
- `boolean isInteger()` returns `true` if the Term is an OtpErlangInt, `false` otherwise
- `boolean isList()` returns `true` if the Term is an OtpErlangList, `false` otherwise
- `boolean isString()` returns `true` if the Term is an OtpErlangString, `false` otherwise
- `boolean isNumber()` returns `true` if the Term is an OtpErlangInteger or an OtpErlangFloat, `false` otherwise
- `boolean isPid()` returns `true` if the Term is an OtpErlangPid or Pid, `false` otherwise
- `boolean isPort()` returns `true` if the Term is an OtpErlangPort or Port, `false` otherwise
- `boolean isReference()` returns `true` if the Term is an OtpErlangRef, `false` otherwise
- `boolean isTuple()` returns `true` if the Term is an OtpErlangTuple, `false` otherwise
- `boolean isBinary()` returns `true` if the Term is an OtpErlangBinary, `false` otherwise

1.10.13 Stub File Types

For each interface, three (3) stub/skeleton files are generated :

- A java interface file, named after the idl interface.
- A client stub file, named after the convention `_< interface name >Stub` which implements the java interface. Example : `_stackStub`.java
- A server stub file, named after the convention `_< interface name >ImplBase` which implements the java interface. Example : `_stackImplBase`.java

1.10.14 Client Stub Initialization, Methods Exported

The recommended constructor function for client stubs accepts four (4) parameters :

- `String selfNode`, the node identification name to be used in the new client node.
- `String peerNode`, the node identification name where the client process is running.
- `String cookie`, the cookie to be used.

- `Object server`, where the java Object can be one of:
  - `OtpErlangPid`, the server's process identity under the node where the server process is running.
  - `String`, the server's registered name under the node where the server process is running.

The methods exported from the generated client stub are :

- `void __disconnect()`, which disconnects the server connection.
- `void __reconnect()`, which disconnects the server connection if open, and then connects to the same peer.
- `void __stop()`, which sends the standard stop termination call. When connected to an Erlang server, the server will be terminated. When connected to a java server, this will set a stop flag that denotes that the server must be terminated.
- `com.ericsson.otp.erlang.OtpErlangRef __getRef()`, will return the message reference received from a server that denotes which call it is referring to. This is useful when building asynchronous clients.
- `java.lang.Object __server()`, which returns the server for the current connection.

1.10.15 Server Skeleton Initialization, Server Stub Implementation, Methods Exported

The constructor function for server skeleton accepts no parameters.

The server skeleton file contains a server `switch` which decodes messages from the input stream and calls implementation (`callback`) functions. As the server skeleton is declared `abstract`, the application programmer will have to create a stub class that `extends` the skeleton file. In this class, all operations defined in the interface class, generated under compiling the idl file, are implemented.

The server skeleton file exports the following methods:

- `OtpOutputStrem invoke(OtpInputStream request)`, where the input stream `request` is unmarshalled, the implementation function is called and a reply stream is marshalled.
- `boolean __isStopped()`, which returns true if a stop message is received. The implementation of the stub should always check if such a message is received and terminate if so.
- `boolean __isStopped(com.ericsson.otp.ic.Environment)`, which returns true if a stop message is received for a certain Environment and Connection. The implementation of the stub should always check if such a message is received and terminate if so.
- `OtpErlangPid __getCallerPid()`, which returns the caller identity for the latest call.
- `OtpErlangPid __getCallerPid(com.ericsson.otp.ic.Environment)`, which returns the caller identity for the latest call on a certain Environment.
- `java.util.Dictionary __operations()`, which returns the operation dictionary which holds all operations supported by the server skeleton.

1.10.16 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 some of the generated files.

```
// The source IDL file: stack.idl

struct s {
      long l;
      string s;
};

interface stack {
    void push(in s val);
    s pop();
};
```

When this file is compiled it produces eight files. Three important files are shown below.

The public interface is in *stack.java*.

```
public interface stack {

/****
 * Operation "stack::push" interface functions
 *
 */

    void push(s val) throws java.lang.Exception;

/****
 * Operation "stack::pop" interface functions
 *
 */

    s pop() throws java.lang.Exception;

}
```

For the IDL struct s three files are generated, a public class in *s.java*.

```
final public class s {
   // instance variables
   public int l;
   public java.lang.String s;

   // constructors
   public s() {};
   public s(int _l, java.lang.String _s) {
     l = _l;
     s = _s;
   };

};
```

A holder class in *sHolder.java* and a helper class in *sHelper.java*. The helper class is used for marshalling.

```

```

```
public class sHelper {

   // constructors
   private sHelper() {};

   // methods
   public static s unmarshal(OtpInputStream in)
      throws java.lang.Exception {
        :
        :
   };

   public static void marshal(OtpOutputStream out, s value)
      throws java.lang.Exception {
        :
        :
   };

};
```

1.10.17 Running the Compiled Code

When using the generated java code you must have added `$OTPROOT/lib/ic_<version number>/priv` and `$OTPROOT/lib/jinterface_<version number>/priv` to your `CLASSPATH` variable to get basic Holder types and the communication classes.

2 Reference Manual

The *IC* application is an Erlang implementation of an IDL compiler.

ic

Erlang module

The ic module is an Erlang implementation of an OMG IDL compiler. Depending on the choice of back-end the code will map to Erlang, C, or Java. The compiler generates client stubs and server skeletons.

Two kinds of files are generated for each scope: Ordinary code files and header files. The latter are used for defining record definitions, while the ordinary files contain the object interface functions.

Exports

```
ic:gen(FileName) -> Result
ic:gen(FileName, [Option]) -> Result
```

Types:

```
Result = ok | error | {ok, [Warning]} | {error, [Warning], [Error]}
Option = [ GeneralOption | CodeOption | WarningOption | BackendOption]
GeneralOption =
{outdir, String()} | {cfgfile, String()} | {use_preproc, bool()} |
{preproc_cmd, String()} | {preproc_flags, String()}
CodeOption =
{gen_hrl, bool()} | {serv_last_call, exception | exit} | {{impl,
String()}, String()} | {light_ifr, bool()}
this | {this, String()} | {{this, String()}, bool()} |
from | {from, String()} | {{from, String()}, bool()} |
handle_info | {handle_info, String()} | {{handle_info, String()}, bool()}
|
timeout | {timeout, String()} | {{timeout, String()}, bool()} |
{scoped_op_calls, bool()} | {scl, bool()} |
{user_protocol, Prefix} |
{c_timeout, {SendTimeout, RecvTimeout}} |
{c_report, bool()} |
{precond, {atom(), atom()}} | {{precond, String()} {atom(), atom()}} |
{postcond, {atom(), atom()}} | {{postcond, String()} {atom(), atom()}}
WarningOption =
{'Wall', bool()} | {maxerrs, int() | infinity} |
{maxwarns, int() | infinity} | {nowarn, bool()} |
{warn_name_shadow, bool()} | {pedantic, bool()} |
{silent, bool()}
BackendOption = {be, Backend}
Backend = erl_corba | erl_template | erl_plain | erl_genserv | c_client |
c_server | java
DirNAme = string() | atom()
FileName = string() | atom()
```

The tuple `{Option, true}` can be replaced by `Option` for boolean values.

The `ic:gen/2` function can be called from the command line as follows:

```
erlc "+Option" ... File.idl
```

Example:

```
erlc "+{be,c_client}" '+{outdir, "../out"}' File.idl
```

General options

outdir

Places all output files in the directory given by the option. The directory will be created if it does not already exist.

Example option: `{outdir, "output/generated"}`.

cfgfile

Uses *FileName* as configuration file. Options will override compiler defaults but can be overridden by command line options. Default value is `".ic_config"`.

Example option: `{cfgfile, "special.cfg"}`.

use_preproc

Uses a preprocessor. Default value is true.

preproc_cmd

Command string to invoke the preprocessor. The actual command will be built as `preproc_cmd++preproc_flags++FileName`

Example option: `{preproc_cmd, "erl"})`.

Example option: `{preproc_cmd, "gcc -x c++ -E"}`.

preproc_flags

Flags given to the preprocessor.

Example option: `{preproc_flags, "-I../include"}`.

Code options

light_ifr

Currently, the default setting is `false`. To be able to use this option Orber must be configured to use Light IFR (see Orber's User's Guide). When this options is used, the size of the generated files used to register the API in the IFR DB are minimized.

Example option: `{light_ifr, true}`.

gen_hrl

Generate header files. Default is true.

serv_last_call

Makes the last `gen_server handle_call` either raise a CORBA exception or just exit plainly. Default is the exception.

{{impl, IntfName}, ModName}

Assumes that the interface with name *IntfName* is implemented by the module with name *ModName* and will generate calls to the *ModName* module in the server behavior. Note that the *IntfName* must be a fully scoped name as in `"M1::I1"`.

this

Adds the object reference as the first parameter to the object implementation functions. This makes the implementation aware of its own object reference.
The option comes in three varieties: `this` which activates the parameter for all interfaces in the source file, `{this, IntfName}` which activates the parameter for a specified interface and `{{this, IntfName}, false}` which deactivates the parameter for a specified interface.

Example option: `this)` activates the parameter for all interfaces.

Example option: `{this, "M1::I1"}` activates the parameter for all functions of `M1::I1`.

Example options: `[this, {{this, "M1::I2"}, false}]` activates the parameter for all interfaces except `M1::I2`.

from

Adds the invokers reference as the first parameter to the object implementation two-way functions. If both `from` and `this` options are used the invokers reference parameter will be passed as the second parameter. This makes it possible for the implementation to respond to a request and continue executing afterwards. Consult the `gen_server` and `Orber` documentation how this option may be used.
The option comes in three varieties: `from` which activates the parameter for all interfaces in the source file, `{from, IntfName}` which activates the parameter for a specified interface and `{{from, IntfName}, false}` which deactivates the parameter for a specified interface.

Example option: `from)` activates the parameter for all interfaces.

Example options: `[{from, "M1::I1"}]` activates the parameter for all functions of `M1::I1`.

Example options: `[from, {{from, "M1::I2"}, false}]` activates the parameter for all interfaces except `M1::I2`.

handle_info

Makes the object server call a function `handle_info` in the object implementation module on all unexpected messages. Useful if the object implementation need to trap exits.

Example option: `handle_info` will activates module implementation `handle_info` for all interfaces in the source file.

Example option: `{{handle_info, "M1::I1"}, true}` will activates module implementation `handle_info` for the specified interface.

Example options: `[handle_info, {{handle_info, "M1::I1"}, false}]` will generate the `handle_info` call for all interfaces except `M1::I1`.

timeout

Used to allow a server response time limit to be set by the user. This should be a string that represents the scope for the interface which should have an extra variable for wait time initialization.

Example option: `{timeout,"M::I"})` produces server stub which will has an extra timeout parameter in the initialization function for that interface.

Example option: `timeout` produces server stub which will has an extra timeout parameter in the initialization function for all interfaces in the source file.

Example options: `[timeout, {{timeout,"M::I"}, false}]` produces server stub which will has an extra timeout parameter in the initialization function for all interfaces except `M1::I1`.

scoped_op_calls

Used to produce more refined request calls to server. When this option is set to true, the operation name which was mentioned in the call is scoped. This is essential to avoid name clashes when communicating with c-servers.

This option is available for the c-client, c-server and the Erlang gen_server back ends. `All` of the parts generated by ic have to agree in the use of this option. Default is `false`.

Example options: `[{be,c_genserv},{scoped_op_calls,true}])` produces client stubs which sends "scoped" requests to a gen_server or a c-server.

user_protocol

Used to define a own protocol different from the default Erlang distribution + gen_server protocol. Currently only valid for C back-ends. For further details see *IC C protocol*.

Example options: `[{be,c_client},{user_protocol, "my_special"}])` produces client stubs which use C protocol functions with the prefix "my_special".

c_timeout

Makes sends and receives to have timeouts (C back-ends only). These timeouts are specified in milliseconds.

Example options: `[{be,c_client},{c_timeout, {10000, 20000}}])` produces client stubs which use a 10 seconds send timeout, and a 20 seconds receive timeout.

c_report

Generates code for writing encode/decode errors to `stderr` (C back-ends only). timeouts are specified in milliseconds.

Example options: `[{be,c_client}, c_report])`.

scl

Used for compatibility with previous compiler versions up to `3.3`. Due to better semantic checks on enumerants, the compiler discovers name clashes between user defined types and enumerant values in the same name space. By enabling this option the compiler turns off the extended semantic check on enumerant values. Default is `false`.

Example option: `{scl,true}`

precond

Adds a precondition call before the call to the operation implementation on the server side.

The option comes in three varieties: `{precond, {M, F}}` which activates the call for operations in all interfaces in the source file, `{{precond, IntfName}, {M, F}}` which activates the call for all operations in a specific interface and `{{precond, OpName}, {M, F}}` which activates the call for a specific operation.

The precondition function has the following signature `m:f(Module, Function, Args)`.

Example option: `{precond, {mod, fun}}` adds the call of m:f for all operations in the idl file.

Example options: `[{{precond, "M1::I"}, {mod, fun}}]` adds the call of `m:f` for all operations in the interface `M1::I1`.

Example options: `[{{precond, "M1::I::Op"}, {mod, fun}}]` adds the call of `m:f` for the operation `M1::I::Op`.

postcond

Adds a postcondition call after the call to the operation implementation on the server side.

The option comes in three varieties: `{postcond, {M, F}}` which activates the call for operations in all interfaces in the source file, `{{postcond, IntfName}, {M, F}}` which activates the call for all operations in a specific interface and `{{postcond, OpName}, {M, F}}` which activates the call for a specific operation.

The postcondition function has the following signature `m:f(Module, Function, Args, Result)`.

Example option: `{postcond, {mod, fun}}` adds the call of m:f for all operations in the idl file.

Example options: `[{{postcond, "M1::I"}, {mod, fun}}]` adds the call of `m:f` for all operations in the interface `M1::I1`.

Example options: `[{{postcond, "M1::I::Op"}, {mod, fun}}]` adds the call of `m:f` for the operation `M1::I::Op`.

Warning options

'Wall'

The option activates all reasonable warning messages in analogy with the gcc -Wall option. Default value is true.

maxerrs

The maximum numbers of errors that can be detected before the compiler gives up. The option can either have an integer value or the atom `infinity`. Default number is 10.

maxwarns

The maximum numbers of warnings that can be detected before the compiler gives up. The option can either have an integer value or the atom `infinity`. Default value is infinity.

nowarn

Suppresses all warnings. Default value is false.

warn_name_shadow

Warning appears whenever names are shadowed due to inheritance; for example, if a type name is redefined from a base interface. Note that it is illegal to overload operation and attribute names as this causes an error to be produced. Default value is true.

pedantic

Activates all warning options. Default value is false.

silent

Suppresses compiler printed output. Default value is false.

Back-End options

Which back-end IC will generate code for is determined by the supplied `{be,atom()}` option. If left out, `erl_corba` is used. Currently, IC support the following back-ends:

erl_corba

This option switches to the IDL generation for CORBA.

erl_template

Generate CORBA call-back module templates for each interface in the target IDL file. Note, will overwrite existing files.

erl_plain

Will produce plain Erlang modules which contain functions that map to the corresponding interface functions on the input file.

erl_genserv

This is an IDL to Erlang generic server generation option.

c_client

Will produce a C client to the generic Erlang server.

c_server

Will produce a C server switch with functionality of a generic Erlang server.

java

Will produce Java client stubs and server skeletons with functionality of a generic Erlang server.

c_genserv

Deprecated. Use `c_client` instead.

Preprocessor

The IDL compiler allows several preprocessors to be used, the `Erlang IDL preprocessor` or other standard `C` preprocessors. Options can be used to provide extra flags such as include directories to the preprocessor. The build in the Erlang IDL preprocessor is used by default, but any standard C preprocessor such as `gcc` is adequate.

The preprocessor command is formed by appending the prepoc_cmd to the preproc_flags option and then appending the input IDL file name.

Configuration

The compiler can be configured in two ways:

- Configuration file
- Command line options

The configuration file is optional and overrides the compiler defaults and is in turn overridden by the command line options. The configuration file shall contain options in the form of Erlang terms. The configuration file is read using `file:consult`.

An example of a configuration file, note the "." after each line.

```
{outdir, gen_dir}.
{{impl, "M1::M2::object"}, "obj"}.
```

Output files

The compiler will produce output in several files depending on scope declarations found in the IDL file. At most three file types will be generated for each scope (including the top scope), depending on the compiler back-end and the compiled interface. Generally, the output per interface will be a header file (`.hrl`/ `.h`) and one or more Erlang/C files (`.erl`/`.c`). Please look at the language mapping for each back-end for details.

There will be at least one set of files for an IDL file, for the file level scope. Modules and interfaces also have their own set of generated files.

ic_clib

C Library

This manual page lists some of the functions in the IC C runtime library.

Allocation and Deallocation Functions

The following functions are used for allocating and deallocating a *CORBA_Environment* structure.

Exports

`CORBA_Environment *CORBA_Environment_alloc(int inbufsz, int outbufsz)`

This function is used to allocate and initiate the `CORBA_Environment` structure. In particular, it is used to dynamically allocate a CORBA_Environment structure and set the default values for the structure's fields.

inbufsize is the initial size of the input buffer.

outbufsize is the initial size of the output buffer.

CORBA_Environment is the CORBA 2.0 state structure used by the generated stub.

This function will set all needed default values and allocate buffers the lengths of which are equal to the values passed, but will not allocate space for the _to_pid and _from_pid fields.

To free the space allocated by CORBA_Environment_alloc() do as follows.

- First call CORBA_free for the input and output buffers.
- After freeing the buffer space, call CORBA_free for the CORBA_Environment space.

`void CORBA_free(void *p)`

Frees allocated space pointed to by `p`.

`CORBA_char *CORBA_string_alloc(CORBA_unsigned_long len)`

Allocates a (simple) CORBA character string of length `len + 1`.

`CORBA_wchar *CORBA_wstring_alloc(CORBA_unsigned_long len)`

Allocates a CORBA wide string of length `len + 1`.

Exception Functions

Functions for retrieving exception ids and values, and for setting exceptions.

Exports

`CORBA_char *CORBA_exception_id(CORBA_Environment *env)`

Returns the exception identity if an exception is set, otherwise it returns `NULL`.

`void *CORBA_exception_value(CORBA_Environment *env)`

Returns the exception value, if an exception is set, otherwise it returns `NULL`.

void CORBA_exc_set(CORBA_Environment *env, CORBA_exception_type Major, CORBA_char *Id, CORBA_char *Value)

Sets the exception type, exception identity, and exception value in the environment pointed to by `env`.

Server Reception

The following function is provided for convenience.

Exports

int oe_server_receive(CORBA_Environment *env, oe_map_t *map)

int oe_server_receive_tmo(CORBA_Environment *env, oe_map_t *map, unsigned int send_ms, unsigned int recv_ms)

Provides a loop that receives one message, executes the operation in question, and in case of a two-way operation sends a reply.

`send_ms` and `recv_ms` specify timeout values in milliseconds for send and receive, respectively.

Generic Execution Switch and Map Merging

Function for searching for server operation function, and for calling it if found. Function for merging maps (see the include file `ic.h` for definitions).

Exports

int oe_exec_switch(CORBA_Object obj, CORBA_Environment *env, oe_map_t *map)

Search for server operation and execute it.

oe_map_t *oe_merge_maps(oe_map_t *maps, int size)

Merge an array of server maps to one single map.

The CORBA_Environment structure

Here is the complete definition of the CORBA_Environment structure, defined in file *ic.h*:

```
/* Environment definition */
typedef struct {

/*----- CORBA compatibility part ------------------------*/
/* Exception tag, initially set to CORBA_NO_EXCEPTION ---*/
CORBA_exception_type   _major;

/*----- External Implementation part - initiated by the user ---*/
/* File descriptor                                              */
int                    _fd;
/* Size of input buffer                                         */
int                    _inbufsz;
/* Pointer to always dynamically allocated buffer for input     */
char                  *_inbuf;
/* Size of output buffer                                        */
int                    _outbufsz;
/* Pointer to always dynamically allocated buffer for output    */
char                  *_outbuf;
```

```
    /* Size of memory chunks in bytes, used for increasing the output
    buffer, set to >= 32, should be around >= 1024 for performance
    reasons                                                        */
    int                    _memchunk;
    /* Pointer for registered name                                  */
    char                   _regname[256];
    /* Process identity for caller                                  */
    erlang_pid            *_to_pid;
    /* Process identity for callee                                 */
    erlang_pid            *_from_pid;

    /*- Internal Implementation part - used by the server/client ---*/
    /* Index for input buffer                                       */
    int                    _iin;
    /* Index for output buffer                                      */
    int                    _iout;
    /* Pointer for operation name                                   */
    char                   _operation[256];
    /* Used to count parameters                                    */
    int                    _received;
    /* Used to identify the caller                                  */
    erlang_pid             _caller;
    /* Used to identify the call                                    */
    erlang_ref             _unique;
    /* Exception id field                                           */
    CORBA_char            *_exc_id;
    /* Exception value field                                        */
    void                  *_exc_value;


    } CORBA_Environment;
```

Note:

Always set the field values *_fd*, *_regname*, *_to_pid* and/or **_from_pid* to appropriate application values. These are not automatically set by the stubs.

Warning:

Never assign static buffers to the buffer pointers, and never set the *_memchunk* field to a value less than *32*.

SEE ALSO

ic(3), ic_c_protocol(3)

ic_c_protocol

C Library

This manual page lists some of the functions of the IC C runtime library that are used internally for the IC protocol.

The listed functions are used internally by generated C client and server code. They are documented here for *the advanced user* that want to replace the default protocol (Erlang distribution + gen_server) by his own protocol, For each set of client or sever functions below with prefix `oe`, the user has to implement his own set of functions, the names of which are obtained by replacing the `oe` prefix by `Prefix`. The `Prefix` has to be set with the option `{user_protocol, Prefix}` at compile time.

The following terminology is used (reflected in names of functions): a *notification* is a message send from client to server, without any reply back (i.e. a *oneway* operation); a *request* is a message sent from client to server, and where a *reply* message is sent back from the server to the client.

In order to understand how the functions work and what they do the user *must* study their implementation in the IC C library (source file is `ic.c`), and also consider how they are used in the C code of ordinary generated client stubs or server skeletons.

Client Protocol Functions

The following functions are used internally by generated C client code.

Exports

`int oe_prepare_notification_encoding(CORBA_Environment *env)`

The result of this function is the beginning of a binary of in external format of the tuple `{'$gen_cast', X}` where `X` is not yet filled in.

In generated client code this function is the first to be called in the encoding function for each oneway operation.

`int oe_send_notification(CORBA_Environment *env)`
`int oe_send_notification_tmo(CORBA_Environment *env, unsigned int send_ms)`

Sends a client notification to a server according to the Erlang distribution + gen_server protocol.

The `send_ms` parameter specified a timeout in milliseconds.

`int oe_prepare_request_encoding(CORBA_Environment *env)`

The result of this function is the beginning of a binary in the external format of the tuple `{'$gen_call', {Pid, Ref}, X}` where `X` is not yet filled in.

In generated client code this function is the first to be called in the encoding function for each twoway operation.

`int oe_send_request_and_receive_reply(CORBA_Environment *env)`
`int oe_send_request_and_receive_reply_tmo(CORBA_Environment *env, unsigned int send_ms, unsigned int recv_ms)`

Sends a client request and receives the reply according to the Erlang distribution + gen_server protocol. This function calls the `oe_prepare_reply_decoding` function in order to obtain the gen_server reply.

`send_ms` and `recv_ms` specify timeouts for send and receive, respectively, in milliseconds.

int oe_prepare_reply_decoding(CORBA_Environment *env)

Decodes the binary version of the tuple `{Ref, X}`, where `X` is to be decoded later by the specific client decoding function.

Server Protocol Functions

The following functions are used internally by generated C server code.

Exports

int oe_prepare_request_decoding(CORBA_Environment *env)

Decodes the binary version of the tuple `{'$gen_cast', Op}` (`Op` an atom), or the tuple `{'$gen_cast', {Op, X}}`, where `Op` is the operation name, and where `X` is to be decoded later by the specific operation decoding function; or

decodes the binary version of the tuple `{'$gen_call', {Pid, Ref}, Op}` (`Op` an atom), or the tuple `{'$gen_call', {Pid, Ref}, {Op, X}}`, where `Op>` is the operation name, and `X` is to be decode later by the specific operation decoding function.

int oe_prepare_reply_encoding(CORBA_Environment *env)

Encodes the beginning of the binary version of the tuple `{{Ref,X}`, where `X` is to be filled in by the specific server encoding function.

SEE ALSO

ic(3), ic_clib(3), *IC Protocol*