IC Application

version 4.0

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

Contents

Chapter 1

IC User's Guide

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

1.1 Using the IC compiler

Introduction

The IC application is an Erlang implementation of an IDL compiler. Several back-ends are supported. The IDL compiler generates server behaviors and client stubs according to the IDL to Erlang mapping. Interface inheritance is supported. The compiler also performs a limited subset of the IDL semantic checks.

Six back-ends are currently supported:

- IDL to Erlang CORBA.
- IDL to (plain) Erlang.
- IDL to generic Erlang Server.
- IDL to generic Erlang Server with C clients.
- IDL to C server switch with generic Erlang Server functionality.
- IDL to Java mapping, where Java client stubs and server skeletons are generated.

While the first back-end (IDL to Erlang CORBA) is intented for pure CORBA functionality, the rest are specially designed to allow portable and efficient links between different languages and virtual machines.

Compiling IDL files.

The compiler is used by calling ic:gen/1 or ic:gen/2 functions in an Erlang shell:

- ic:gen/1 is used to compile files with only default settings.
- ic:gen/2 is used to compile files with an additional option list.

Example compiling a file `example.idl`:

- This will generate code for the `default` back-end.

```
1> ic:gen(example).
Erlang IDL compiler version 2.5.1
ok
2>
```

- This will generate code for the generic Erlang server.

```
1> ic:gen(example,[{be,erl_genserv}]).
Erlang IDL compiler version 2.5.1
ok
2>
```

Compiler configuration.

There are a number of compiler options available to the user, which can be configured by either:

- Using a configuration file, or
- By using command line options on ic:gen/2.

Please read the manual page for information about valid options and use of the configuration file.

1.2 OMG IDL Mapping

OMG IDL Mapping - Overview

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

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

OMG IDL mapping elements

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

All mapping must define the following elements:

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

Reserved compiler names

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

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

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

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

Refer to the IC documentation for further details.

Basic OMG IDL types

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

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

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

Table 1.1: OMG IDL basic types

The `any` value is written as a record with the field typecode which contains the *Type Code* representation, see also the Type Code table [page 9], and the value field itself.

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

Constructed OMG IDL types

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

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

Table 1.2: OMG IDL constructed types

Below are examples of values of constructed types.

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

Table 1.3: Typical values

References to constants

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

Example:

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

Would result in the following conceptual code:

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

c1() -> 99.
```

References to objects defined in OMG IDL

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

Objects are implemented by providing implementations for all operations and attributes of the Object, see operation implementation [page 7].

Invocations of operations

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

Example:

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

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 11]

Exceptions

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

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

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

Access to attributes

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

Typecode, Identity and Name access functions.

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

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

For example:

```
// IDL
module m {

  struct s {
    long x;
    long y;
   };

};
```

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

```
-module(m_s).

-include("m.hrl").

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


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

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

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

Type Code representation

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

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

continued ...

... continued

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

Table 1.4: Type Code tuples

Scoped names

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

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

Syntax Specific structures for scoped names

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

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

Files

Several files can be generated for each scope.

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

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

Example:

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

  struct s {
    long x;
    long y;
  };

  interface i {

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

 };

};
```

This will produce the following files:

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

A mapping example

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

```
// The source IDL file

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

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

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

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

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

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

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

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

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

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

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

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

```
-module(stack_impl).

-include("stack.hrl").

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

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

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

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

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

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

```
-module(calc_impl).

-export([add/1]).

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

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

1.3 C IDL language mapping

Introduction

This chapter describes the mapping of OMG IDL constructs to the C programming language for the generation of native C - 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

Mapping pecularities

Names reserved by the compiler

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 the use of underscores in identifiers.

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 {
        ...
    };
    ...
};
```

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.

Basic OMG IDL types

The mapping of basic types is flexible to allow type adjustment. This can be used when porting to machines with different architectures.

OMG IDL type	C type	Implementation	Adjustable
float	CORBA_float	float	yes
double	CORBA_double	double	yes
short	CORBA_short	short	yes
unsigned short	CORBA_unsigned_short	unsigned short	yes
long	CORBA_long	long	yes
long long	CORBA_long_long	long	yes
unsigned long	CORBA_unsigned_long	unsigned long	yes
unsigned long long	CORBA_unsigned_long_long	unsigned long	yes
char	CORBA_char	char	yes
wchar	CORBA_wchar	unsigned long	yes
boolean	CORBA_boolean	unsigned char	yes
octet	CORBA_octet	char	yes
any	Not supported		

continued ...

... continued

long double	Not supported		
Object	Not supported		
void	void	void	no

Table 1.5: OMG IDL basic types

Constructed OMG IDL types

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

Mapping for string

OMG IDL strings are mapped to C CORBA_char*.

Mapping for wstring

OMG IDL wstrings are mapped to C CORBA_wchar*.

Mapping for struct

An OMG IDL structure is mapped directly onto a C struct.

Mapping for union

An OMG IDL union is mapped directly onto a C discriminated union.

Mapping for enum

An OMG IDL enum is directly mapped onto a C enum.

Mapping for sequence

OMG IDL sequences are mapped to a C struct that represents the sequence.
Consider the following IDL declaration:

```
typedef sequence<long> lseq;
```

Which in C is represented as:

```
typedef struct {
  CORBA_unsigned_long _maximum;
  CORBA_unsigned_long _length;
  CORBA_long* _buffer;
} lseq;
```

Mapping for array

OMG IDL arrays are mapped directly to C arrays.

Mapping for constants

Constants are mapped to C `#define`.

For example:

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

Would result in the following define:

```
#define M1_c1 99
```

Invocations of operations

Operation invocation is achieved through a function call. The function calls have two default parameters, the *interface object* and the *environment* parameter. The result of the function is returned the usual way, while in and out parameters lie between the two default parameters in the same order as they appear in the IDL file.

Default parameters:

- < interface object > *oe_obj* - defined as CORBA_Object, always located as the first parameter of the operation. In the current implementation there is no use for this parameter.
- CORBA_Environment* *oe_env* - the environment structure. It is defined under *ic.h* and has the following public fields:
  - CORBA_Exception_type *_major* - will indicate whether the invocation terminated successfully, 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 allways 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 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.

Exceptions

While 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.

Access to attributes

Not Supported

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 1.6: Basic Argument and Result passing

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

OMG IDL type	Out	Return

continued ...

... continued

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 1.7: 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 1.8: 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.

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 wished size of input buffer
  - *outbufsz* is the wished 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.

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.

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.

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.

1.4 Using the Erlang Generic Server back-end

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 on the generated code, a client stub and server skeleton to an Erlang `gen_server`.

Compiling the code

In the Erlang shell type :

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

Writing the implementation file

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

- Create the coresponding 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 takes as the 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.

An example

In this example, a file "random.idl" is generates code for the plain erlang backend :

- Main file : "random.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_genserv}]).
 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/1, init/4]).

%% Type identification function
-export([typeID/0]).

%% Used to start server
-export([oe_create/0, oe_create_link/0, oe_create/1]).
-export([oe_create_link/1, oe_create/2, oe_create_link/2]).

-export([start/2, start_link/3]).

%% gen server export stuff
-behaviour(gen_server).
-export([init/1, terminate/2, handle_call/3]).
-export([handle_cast/2, handle_info/2]).


%%------------------------------------------------------------
%%
%% Object interface functions.
%%
%%------------------------------------------------------------



%%%% Operation: produce
%%
%%   Returns: RetVal
%%
produce(OE_THIS) ->
```

```
    gen_server:call(OE_THIS, produce, infinity).

%%%% Operation: init
%%
%%   Returns: RetVal
%%
init(OE_THIS, Seed1, Seed2, Seed3) ->
    gen_server:cast(OE_THIS, {init, Seed1, Seed2, Seed3}).




%%------------------------------------------------------------
%%
%% Server implementation.
%%
%%------------------------------------------------------------


%%------------------------------------------------------------
%%
%% Function for fetching the interface type ID.
%%
%%------------------------------------------------------------

typeID() ->
    "IDL:rmod/random:1.0".


%%------------------------------------------------------------
%%
%% Server creation functions.
%%
%%------------------------------------------------------------

oe_create() ->
    start([], []).

oe_create_link() ->
    start_link([], []).

oe_create(Env) ->
    start(Env, []).

oe_create_link(Env) ->
    start_link(Env, []).

oe_create(Env, RegName) ->
    start(RegName, Env, []).

oe_create_link(Env, RegName) ->
    start_link(RegName, Env, []).
```

```
%%--------------------------------------------------------------
%%
%% Start functions.
%%
%%--------------------------------------------------------------

start(Env, Opt) ->
    gen_server:start(?MODULE, Env, Opt).

start_link(Env, Opt) ->
    gen_server:start_link(?MODULE, Env, Opt).

start(RegName, Env, Opt) ->
    gen_server:start(RegName, ?MODULE, Env, Opt).

start_link(RegName, Env, Opt) ->
    gen_server:start_link(RegName, ?MODULE, Env, Opt).

init(Env) ->
%% Call to implementation init
    rmod_random_impl:init(Env).

terminate(Reason, State) ->
    rmod_random_impl:terminate(Reason, State).


%%%% Operation: produce
%%
%%   Returns: RetVal
%%
handle_call(produce, OE_From, OE_State) ->
    rmod_random_impl:produce(OE_State);

%%%% Standard Operation: oe_get_interface
%%
handle_call({OE_THIS, oe_get_interface, []}, From, State) ->
    {reply, [{"produce",{tk_double,[],[]}},
             {"init",{tk_void,[tk_long,tk_long,tk_long],[]}}], State};



%%%% Standard gen_server call handle
%%
handle_call(stop, From, State) ->
    {stop, normal, ok, State}.


%%%% Operation: init
%%
%%   Returns: RetVal
%%
handle_cast({init, Seed1, Seed2, Seed3}, OE_State) ->
    rmod_random_impl:init(OE_State, Seed1, Seed2, Seed3);
```

```
    %%%% Standard gen_server cast handle
    %%
    handle_cast(stop, State) ->
        {stop, normal, State}.




    %%%% Standard gen_server handles
    %%
    handle_info(X, State) ->
        {noreply, State}.
```

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

```
-module('rmod_random_impl').
-export([init/1, terminate/2]).
-export([produce/1,init/4]).


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

terminate(From, Reason) ->
    ok.


produce(_Random)  ->
    case catch random:uniform() of
        {'EXIT',_} ->
            true;
        RUnif ->
            {reply,RUnif,[]}
     end.


init(_Random,S1,S2,S3)  ->
    case catch random:seed(S1,S2,S3) of
        {'EXIT',_} ->
            true;
        _ ->
            {noreply,[]}
    end.
```

Compiling the code :

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

Running the example :

```
3> {ok,R} =  rmod_random:oe_create().
{ok,<0.30.0>
4> rmod_random:init(R,1,2,3).
ok
5> rmod_random:produce(R).
1.97963e-4
6>
```

1.5 Using the Plain Erlang back-end

Introduction

The mapping of OMG IDL to the Erlang programming language when Plain Erlang is the bac-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.

Compiling the code

In the Erlang shell type :

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

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 takes as the 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.

An example

In this example, a file “random.idl” is generates code for the plain erlang backend :

- 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>
```

1.6 Using the C Client back-end

Introduction

The mapping of OMG IDL to the C programming language when C Server switch is the back-end of choice is identical to the one used in C IDL mapping. The only difference is on the generated code and that the idl functions are translated to C functions for the C client.

When to use the C-Client?

A C-client uses the same communication protocol as a Erlang client to genservers, as it is actually an C-genserver client. Therefore, the C-client can be used for:

- Calling functions served by C-servers generated by the C-server back-end.
- Calling functions served by Erlang-genservers generated by the Erlang genserver back-end.

What kind of code is produced?

The code produced is a collection of:

- C source files that contain interface code.
  These files are named after the < `Scoped Interface Name` >`s.c` convention
- C source files that contain code for:
  - type conversion
  - memory allocation
  - data encoding / decoding into buffers
- C header files that contain function headers and type definitions.

All functions found in the code are exported. The user is free to define his own client if there is a need for this. The basic client generated is a `synchronous` client, but an `asynchronous` client can be implemented by proper use of exported functions.

What does this code do when used?

The main functionality of a C client is to:

- Encode call request messages.
- Write messages to a specified file descriptor.
- Read from a specified file descriptor.
- Decode the reply messages.

- Return output values

What is the interface of the functions produced?

The C source defines the following functions:

- One client function for each IDL function.
- One specific message encoder function for each IDL function.
- One specific call function for each function defined in the interface.
- One generic reply message decoder function for each IDL interface.
- One specific return value decoder function for each IDL function.

The interface for the client function is:

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

Where:

- < `Return Value` > is the return value is the value to be returned as defined by the IDL specification for the operation.
- < `Interface Object` > `oe_obj` is the client interface object.
- < `Parameters` > are the parameters to the operation in the same order as defined by the IDL specdication for the operation.
- `CORBA_Environment *oe_env` is a pointer to the current client environment as described in section 3.6.

The interface for the message encoding functions is:

int < Scoped Function Name >__client_enc(< Interface Object > oe_obj, < Input Parameters > CORBA_Environment *oe_env);

Where:

- < `Interface Object` > `oe_obj` is the client interface object.
- < `Input Parameters` > are all the `input`parameters to the operation in the same order as defined by the IDL specification for the operation.
- `CORBA_Environment *oe_env` is a pointer to the current client environment as described in section 3.6.
- the return value for the client is an `int` which is positive or zero when the call is succeed, negative otherwise

The interface for the specific result value decoder is:

int < Scoped Function Name >__client_dec(< Interface Object > oe_obj, < Return/Out Values > CORBA_Environment *oe_env);

Where:

- < `Interface Object` > `oe_obj` is the client interface object.
- < `Return/Out Values` > are return values in order similar to the IDL defined function's.
- `CORBA_Environment *oe_env` is a pointer to the current client environment as described in section 3.6.

- the return value for the client is an `int` which is positive or zero when the call is succeed, negative otherwise

The interface for the generic decoding function is:

int < Scoped Interface Name >__receive_info(< Interface Object > oe_obj, CORBA_Environment *oe_env);

Where:

- < `Interface Object` > `oe_obj` is the client interface object.
- `CORBA_Environment *oe_env` is a pointer to the current client environment as described in section 3.6.
- the return value for the client is an `int` which is positive or zero when the call is succeed, negative otherwise

Functions used for internal purposes

Depending on the data defined and used in the IDL code, C-source files may be generated that hold functions used internally. This is the case when other types than the elementary IDL types are used by the IDL file definitions. All these files must be compiled and linked to the other code.

Which header files to include?

The only header files that must be included are :

- the interface files, the files named < `Scoped Interface Name` >`.h`.

Which directories/libraries/options must be included under C-compiling?

Under compilation you will have to include :

- the directory `$OTPROOT/ usr/ include`

Under linking you will have to link with :

- the libraries under `$OTPROOT/ usr/ lib`
- -lerl_interface -lei -lnsl -lsocket -lic

Compiling the code

In the Erlang shell type:

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

An example

In this example, a file “random.idl” is generates code for the plain erlang backend:

- Main file: “random.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, c_client}]).
Erlang IDL compiler version 3.2
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:.

- C file for interface: “rmod_random.c”

```
#include <stdlib.h>
#include <string.h>
#include "ic.h"
#include "erl_interface.h"
#include "ei.h"
#include "rmod_random.h"



/*
 *  Object interface function "rmod_random_produce"
 */

CORBA_double rmod_random_produce(rmod_random oe_obj,
CORBA_Environment *oe_env) {

  CORBA_double oe_result;
  int oe_msgType = 0;
  erlang_msg oe_msg;
```

```
/* Initiating the message reference */
strcpy(oe_env->_unique.node,erl_thisnodename());
oe_env->_unique.creation = erl_thiscreation();
oe_env->_unique.id = 0;

/* Initiating exception indicator */
oe_env->_major = CORBA_NO_EXCEPTION;

/* Creating call message */
if (rmod_random_produce__client_enc(oe_obj, oe_env) < 0) {
  if (oe_env->_major == CORBA_NO_EXCEPTION)
    CORBA_exc_set(oe_env,
                  CORBA_SYSTEM_EXCEPTION,
                  MARSHAL,
                  "Cannot encode message");
  return oe_result;
}

/* Sending call request */
if (strlen(oe_env->_regname) == 0) {
  if (ei_send_encoded(oe_env->_fd,
                      oe_env->_to_pid,
                      oe_env->_outbuf,
                      oe_env->_iout) < 0) {
      CORBA_exc_set(oe_env,
                    CORBA_SYSTEM_EXCEPTION,
                    NO_RESPONSE,
      "Cannot connect to server");
    return oe_result;
  }
}
else if (ei_send_reg_encoded(oe_env->_fd,
                             oe_env->_from_pid,
                             oe_env->_regname,
                             oe_env->_outbuf,
                             oe_env->_iout) < 0) {
       CORBA_exc_set(oe_env,
                     CORBA_SYSTEM_EXCEPTION,
                     NO_RESPONSE,
                     "Cannot connect to server");
  return oe_result;
}

/* Receiving reply message */
do {
  if ((oe_msgType =
        ei_receive_encoded(oe_env->_fd,
                           &oe_env->_inbuf,
                           &oe_env->_inbufsz,
                           &oe_msg,
                           &oe_env->_iin)) < 0) {
        CORBA_exc_set(oe_env,
                      CORBA_SYSTEM_EXCEPTION,
```

```
                        MARSHAL,
                        "Cannot decode message");
            return oe_result;
      }
    } while (oe_msgType != ERL_SEND && oe_msgType != ERL_REG_SEND);

    /* Extracting message header */
    if (rmod_random__receive_info(oe_obj, oe_env) < 0) {
      CORBA_exc_set(oe_env,
                    CORBA_SYSTEM_EXCEPTION,
                    MARSHAL,
                    "Bad message");
      return oe_result;
    }

    /* Extracting return value(s) */
    if (rmod_random_produce__client_dec(oe_obj,
                                        &oe_result,
                                        oe_env) < 0) {
      CORBA_exc_set(oe_env,
                    CORBA_SYSTEM_EXCEPTION,
                    DATA_CONVERSION,
                    "Bad return/out value(s)");
    }

    return oe_result;
}


/*
 *  Encodes the function call for "rmod_random_produce"
 */

int rmod_random_produce__client_enc(rmod_random oe_obj,
                                    CORBA_Environment *oe_env) {

    int oe_error_code = 0;
    oe_env->_iout = 0;

    oe_ei_encode_version(oe_env);
    oe_ei_encode_tuple_header(oe_env, 3);
    oe_ei_encode_atom(oe_env, "$gen_call");
    oe_ei_encode_tuple_header(oe_env, 2);

    if ((oe_error_code =
            oe_ei_encode_pid(oe_env,oe_env->_from_pid)) < 0)
      return oe_error_code;

    if ((oe_error_code =
            oe_ei_encode_ref(oe_env,&oe_env->_unique)) < 0)
      return oe_error_code;

    oe_ei_encode_atom(oe_env, "produce");
```

```
  return 0;
}


/*
 *  Decodes the return value for "rmod_random_produce"
 */

int rmod_random_produce__client_dec(rmod_random oe_obj,
                                    CORBA_double* oe_result,
                                    CORBA_Environment *oe_env) {

  int oe_error_code = 0;

  /* Decode result value: CORBA_double* oe_result */
  if ((oe_error_code =
          ei_decode_double(oe_env->_inbuf,
                           &oe_env->_iin,
                           oe_result)) < 0)
    return oe_error_code;


  return 0;
}



/*
 *  Object interface function "rmod_random_init"
 */

void rmod_random_init(rmod_random oe_obj,
                      CORBA_long seed1,
                      CORBA_long seed2,
                      CORBA_long seed3,
                      CORBA_Environment *oe_env) {

  /* Initiating exception indicator */
  oe_env->_major = CORBA_NO_EXCEPTION;

  /* Creating call message */
  if (rmod_random_init__client_enc(oe_obj,
                                   seed1,
                                   seed2,
                                   seed3,
                                   oe_env) < 0) {
    if (oe_env->_major == CORBA_NO_EXCEPTION)
      CORBA_exc_set(oe_env,
                    CORBA_SYSTEM_EXCEPTION,
                    MARSHAL,
                    "Cannot encode message");
  }
```

```
  /* Sending call request */
  if (oe_env->_major == CORBA_NO_EXCEPTION) {
    if (strlen(oe_env->_regname) == 0) {
      if (ei_send_encoded(oe_env->_fd,
                          oe_env->_to_pid,
                          oe_env->_outbuf,
                          oe_env->_iout) < 0) {
        CORBA_exc_set(oe_env,
                      CORBA_SYSTEM_EXCEPTION,
                      NO_RESPONSE,
                      "Cannot connect to server");
      }
    }
    else if (ei_send_reg_encoded(oe_env->_fd,
                                 oe_env->_from_pid,
                                 oe_env->_regname,
                                 oe_env->_outbuf,
                                 oe_env->_iout) < 0) {
      CORBA_exc_set(oe_env,
                    CORBA_SYSTEM_EXCEPTION,
                    NO_RESPONSE,
                    "Cannot connect to server");
    }
  }

}

/*
 *  Encodes the function call for "rmod_random_init"
 */

int rmod_random_init__client_enc(rmod_random oe_obj,
                                 CORBA_long seed1,
                                 CORBA_long seed2,
                                 CORBA_long seed3,
                                 CORBA_Environment *oe_env) {

  int oe_error_code = 0;
  oe_env->_iout = 0;

  oe_ei_encode_version(oe_env);
  oe_ei_encode_tuple_header(oe_env, 2);

  oe_ei_encode_atom(oe_env, "$gen_cast");
  oe_ei_encode_tuple_header(oe_env, 4);
  oe_ei_encode_atom(oe_env, "init");

  /* Encode parameter: CORBA_long seed1 */
  if ((oe_error_code = oe_ei_encode_long(oe_env, seed1)) < 0)
    return oe_error_code;

  /* Encode parameter: CORBA_long seed2 */
```

```
  if ((oe_error_code = oe_ei_encode_long(oe_env, seed2)) < 0)
    return oe_error_code;

  /* Encode parameter: CORBA_long seed3 */
  if ((oe_error_code = oe_ei_encode_long(oe_env, seed3)) < 0)
    return oe_error_code;

  return 0;
}



/*
 *  Generic function, used to return received message information.
 *  Not used by oneways. Allways generated.
 */

int rmod_random__receive_info(rmod_random oe_obj,
                              CORBA_Environment *oe_env) {

  int oe_error_code = 0;
  int oe_rec_version = 0;
  erlang_ref oe_unq;
  oe_env->_iin = 0;
  oe_env->_received = 0;

  if ((oe_error_code =
        ei_decode_version(oe_env->_inbuf,
                          &oe_env->_iin,
                          &oe_rec_version)) < 0)
    return oe_error_code;

  if ((oe_error_code =
       ei_decode_tuple_header(oe_env->_inbuf,
                              &oe_env->_iin,
                              &oe_env->_received)) < 0)
    return oe_error_code;

  if ((oe_error_code =
        ei_decode_ref(oe_env->_inbuf,
                      &oe_env->_iin,
                      &oe_unq)) < 0)
    return oe_error_code;

  /* Checking message reference*/
  if(strcmp(oe_env->_unique.node,oe_unq.node) != 0)
    return -1;

  if(oe_env->_unique.id != oe_unq.id)
    return -1;

  return 0;
```

```
}
```

Compiling the code:

- Please read the `ReadMe` file att the `ic-3.2/examples/c-client` directory
  In the same directory you can find all the code for this example

> **Note:**
> Due to changes to allow buffer expansion, a new receiving function some changes in CORBA_Environment initialization are applied. The example in the *ic-3.2/examples/c-client* directory demonstrates these changes.

Running the example:

- Please check the `ReadMe` file att the `ic-3.2/examples/c-client` directory
  In the same directory you can find all the code for this example

1.7 Using the C Server back-end

Introduction

The mapping of OMG IDL to the C programming language when C Server switch is the back-end of choice is identical to the one use in C IDL mappning. The only difference is on the generated code and that the idl functions are translated to C function calls for the C Server.

What is the C-server good for ?

The C-server uses the same communication protocol as for the Erlang genservers, it is actually a C-genserver. So the C-server can be used for :

- Serving C-clients generated by the C-client back-end.
- Serving Erlang genserver-clients generated by the Erlang genserver back-end.

What kind of code is produced ?

The code produced is a collection of :

- C source files that contain interface code.
  These files are named after the < `Scoped Interface Name` >`_s.c` convension
- C source files that contain code for :
  - type conversion
  - memory allocation
  - data enoding / decoding into buffers
- C header files that contain function headers and type definitions.

All functions found in the code are exported. The user is free to define its own switches if there is a need for this.

What does this code do when used ?

The main functionality of a C server switch is to :

- Decode call requests stored in buffers
- Recognize the function noted in a request
- Call the calback function that implements the request with the parameters folowed in the message
- Collect the output from the callback function (if the function defined is not a `cast`)
- Encode the output value to an output buffer

- Call the restore function (if defined) that frees memory or/and sets up a server state

What is the interface of the functions produced ?

The C source defines the following functions :

- One server switch for each interface.
- One generic message decoder for each switch.
- One specific call function for each function defined in the interface.
- At most one specific parameter decoding function for each call function.
- One callback function for each call function.
- At most one specific return value encoding function for each call function.

The interface for the server switch is :

int < Scoped Interface Name >_switch(< Interface Object > oe_obj, CORBA_Environment *oe_env);

Where :

- `< Interface Object > oe_obj` is the client interface object.
- `CORBA_Environment *oe_env` is a pointer to the current client environment as described in section 3.6.
- the return value for is an `int` which is positiv or zero when the call is succed, negative otherwize

The interface for the generic message decoder is :

int < Scoped Interface Name >_call_info((< Interface Object > oe_obj, CORBA_Environment *oe_env);

Where :

- `< Interface Object > oe_obj` is the client interface object.
- `CORBA_Environment *oe_env` is a pointer to the current client environment as described in section 3.6.
- the return value for is an `int` which is positiv or zero when the call is succed, negative otherwize

The interface for the specific call function definition is :

int < Scoped Function Name >__exec(< Interface Object > oe_obj, CORBA_Environment *oe_env);

Where :

- `< Interface Object > oe_obj` is the client interface object.
- `CORBA_Environment *oe_env` is a pointer to the current client environment as described in section 3.6.
- the return value for is an `int` which is positiv or zero when the call is succed, negative otherwize

The interface for the specific parameter decoder function is :

int < Scoped Function Name >__dec(< Interface Object > oe_obj, < Parameters > CORBA_Environment *oe_env);

Where :

- `< Interface Object > oe_obj` is the client interface object.

- < Parameters > are pointers to parameters for the function call to be decoded. The order of apearence is similar to the IDL definition of the function.
- CORBA_Environment *oe_env is a pointer to the current client environment as described in section 3.6.
- the return value for is an int which is positiv or zero when the call is succeed, negative otherwize

The interface for the specific callback function is :

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

Where :

- < Interface Object > oe_obj is the client interface object.
- < Parameters > are pointers to in/out-parameters for the function call. The order of apearence is similar to the IDL definition of the function.
- CORBA_Environment *oe_env is a pointer to the current client environment as described in section 3.6.
- the return value for is a pointer to the restore function which can be NULL when the restore function is not defined, initiated to point the restore function otherwize

Callback functions are implementation dependent and in order to make things work, the following rule must be followed when passing arguments to callback functions :

- in parameters of variable storage type are passed as is.
- out parameters of variable storage type are passed by a pointer to their value.
- in / out parameters of fixed storage type are passed by a pointer to their value.
- return values are always passed by a pointer to their value.

The interface for the specific message encoder function is :

int < Scoped Function Name >__enc(< Interface Object > oe_obj, < Parameters > CORBA_Environment *oe_env);

Where :

- < Interface Object > oe_obj is the client interface object.
- < Parameters > are pointers to parameters for the return message to be encoded. The order of apearence is similar to the IDL definition of the function.
- CORBA_Environment *oe_env is a pointer to the current client environment as described in section 3.6.
- the return value for is an int which is positiv or zero when the call is succeed, negative otherwize

The encoder function is generated only for usual call IDL-functions (not oneways)

The interface for the specific restore function is :

void < Scoped Function Name >__rs(< Interface Object > oe_obj, < Parameters > CORBA_Environment *oe_env);

Where :

- < Interface Object > oe_obj is the client interface object.
- < pointers to result values / parameters > are pointers to in/out-parameters for the function call. The order of apearence is similar to the IDL definition of the function.

- `CORBA_Environment *oe_env` is a pointer to the current client environment as described in section 3.6.

The restore function type definition is recorded on the interface header file. It is unique for each IDL defined interface function

Functions used for internal purposes

Depending on the data defined and used in the IDL code, C-source files may be generated that hold functions used internally. This is the case when other types than the elementary IDL types are used by the IDL file definitions. All these files must be compiled and linked to the other code.

Which header files to include ?

The only header files that must be included are the interface files, the files named < `Scoped Interface Name` >`_s.h`

Which directories/libraries/options must be included under C-compiling?

Under compilation you will have to include :

- the directory `$OTPROOT/ usr/ include`

Under linking you will have to link with :

- the libraries under `$OTPROOT/ usr/ lib`
- -lerl_interface -lei -lnsl -lsocket -lic

Compiling the code

In the Erlang shell type :

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

Implementing the callback functions

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

- Create the coresponding C file that will hold the C callback functions for the IDL defined functions.
- The implementation file does not need a special naming.

For each function defined in the IDL interface :

- Implement an C function that takes as the 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 3.

An example

In this example, a file “random.idl” is generates code for the plain erlang backend :

- Main file : “random.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, c_server}]).
Erlang IDL compiler version 3.2
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 theis case only the file for the interface `rmod_random.erl` is important :.

- C file for interface : “rmod_random_s.c”

```
#include <string.h>
#include "ic.h"
#include "erl_interface.h"
#include "ei.h"
#include "rmod_random__s.h"


/*
 * Main switch
 */

int rmod_random__switch(rmod_random oe_obj, CORBA_Environment *oe_env) {

  int status=0;

  /* Initiating exception indicator */
  oe_env->_major = CORBA_NO_EXCEPTION;

  /* Call switch */
  if ((status = rmod_random__call_info(oe_obj, oe_env)) >= 0) {

    if (strcmp(oe_env->_operation, "produce") == 0)
      return rmod_random_produce__exec(oe_obj, oe_env);

    if (strcmp(oe_env->_operation, "init") == 0)
      return rmod_random_init__exec(oe_obj, oe_env);

    /* Bad call */
    CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, BAD_OPERATION,
                  "Invalid operation");
    return -1;
  }


  /* Exit */
  return status;
}



/*
 * Returns call identity
 */

int rmod_random__call_info(rmod_random oe_obj,
                           CORBA_Environment *oe_env) {

  char gencall_atom[10];
  int error_code = 0;
  int rec_version = 0;
  oe_env->_iin = 0;
  oe_env->_received = 0;
```

```
ei_decode_version(oe_env->_inbuf, &oe_env->_iin, &rec_version);
ei_decode_tuple_header(oe_env->_inbuf, &oe_env->_iin,
                       &oe_env->_received);
ei_decode_atom(oe_env->_inbuf, &oe_env->_iin, gencall_atom);

if (strcmp(gencall_atom, "$gen_cast") == 0) {

  if ((error_code = ei_decode_atom(oe_env->_inbuf, &oe_env->_iin,
                                   oe_env->_operation)) < 0) {
    ei_decode_tuple_header(oe_env->_inbuf, &oe_env->_iin,
                           &oe_env->_received);
    if ((error_code = ei_decode_atom(oe_env->_inbuf, &oe_env->_iin,
                                     oe_env->_operation)) < 0) {
      CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, BAD_OPERATION,
                    "Bad Message, cannot extract operation");
      return error_code;
    }
    oe_env->_received -= 1;
  } else
    oe_env->_received -= 2;

  return 0;
}

if (strcmp(gencall_atom, "$gen_call") == 0) {

  ei_decode_tuple_header(oe_env->_inbuf, &oe_env->_iin,
                         &oe_env->_received);

  if ((error_code = ei_decode_pid(oe_env->_inbuf, &oe_env->_iin,
                                  &oe_env->_caller)) < 0) {
    CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, MARSHAL,
                  "Bad Message, bad caller identity");
    return error_code;
  }

  if ((error_code = ei_decode_ref(oe_env->_inbuf, &oe_env->_iin,
                                  &oe_env->_unique)) < 0) {
    CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, MARSHAL,
                  "Bad Message, bad message reference");
    return error_code;
  }

  if ((error_code = ei_decode_atom(oe_env->_inbuf, &oe_env->_iin,
                                   oe_env->_operation)) < 0) {

    ei_decode_tuple_header(oe_env->_inbuf, &oe_env->_iin,
                           &oe_env->_received);

    if ((error_code = ei_decode_atom(oe_env->_inbuf, &oe_env->_iin,
                                     oe_env->_operation)) < 0) {
      CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, BAD_OPERATION,
                    "Bad Message, cannot extract operation");
```

```
        return error_code;
      }

      oe_env->_received -= 1;
      return 0;
    }
    else {
      oe_env->_received -= 2;
      return 0;
    }
  }

  CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, MARSHAL,
                "Bad message, neither cast nor call");
  return -1;
}



int rmod_random_produce__exec(rmod_random oe_obj,
                              CORBA_Environment *oe_env) {

  if (oe_env->_received != 0) {
    CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, BAD_PARAM,
                  "Wrong number of operation parameters");
    return -1;
  }
  else {
    rmod_random_produce__rs* oe_restore = NULL;
    CORBA_double oe_result = 0;

    /* Callback function call */
    oe_restore = rmod_random_produce__cb(oe_obj, &oe_result, oe_env);

    /* Encoding reply message */
    rmod_random_produce__enc(oe_obj, oe_result, oe_env);

    /* Restore function call */
    if (oe_restore != NULL)
      (*oe_restore)(oe_obj, &oe_result, oe_env);

  }
  return 0;
}


int rmod_random_produce__enc(rmod_random oe_obj,
                             CORBA_double oe_result,
                             CORBA_Environment *oe_env) {
  int oe_error_code;
  oe_env->_iout = 0;

  oe_ei_encode_version(oe_env);
```

```
  oe_ei_encode_tuple_header(oe_env, 2);
  oe_ei_encode_ref(oe_env, &oe_env->_unique);

  /* Encode parameter: CORBA_double oe_result */
  if ((oe_error_code = oe_ei_encode_double(oe_env, oe_result)) < 0) {
    CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, BAD_PARAM,
                  "Bad operation parameter on encode");
    return oe_error_code;
  }

  return 0;
}


int rmod_random_init__exec(rmod_random oe_obj,
                           CORBA_Environment *oe_env) {

  if (oe_env->_received != 3) {
    CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, BAD_PARAM,
                  "Wrong number of operation parameters");
    return -1;
  }
  else {
    int oe_error_code = 0;
    rmod_random_init__rs* oe_restore = NULL;
    CORBA_long seed1;
    CORBA_long seed2;
    CORBA_long seed3;


    /* Decode parameters */
    if((oe_error_code = rmod_random_init__dec(oe_obj, &seed1, &seed2,
                                              &seed3, oe_env)) < 0) {
      CORBA_exc_set(oe_env, CORBA_SYSTEM_EXCEPTION, BAD_PARAM,
                    "Bad parameter on decode");
      return oe_error_code;
  }

    /* Callback function call */
    oe_restore = rmod_random_init__cb(oe_obj, &seed1, &seed2, &seed3,
                 oe_env);

    /* Restore function call */
    if (oe_restore != NULL)
      (*oe_restore)(oe_obj, &seed1, &seed2, &seed3, oe_env);

  }
  return 0;
}

int rmod_random_init__dec(rmod_random oe_obj, CORBA_long* seed1,
                          CORBA_long* seed2, CORBA_long* seed3,
                          CORBA_Environment *oe_env) {
```

```
      int oe_error_code;

      if ((oe_error_code = ei_decode_long(oe_env->_inbuf,
                                          &oe_env->_iin, seed1)) < 0)
        return oe_error_code;


      if ((oe_error_code = ei_decode_long(oe_env->_inbuf,
                                          &oe_env->_iin, seed2)) < 0)
        return oe_error_code;


      if ((oe_error_code = ei_decode_long(oe_env->_inbuf,
                                          &oe_env->_iin, seed3)) < 0)
        return oe_error_code;


      return 0;
    }
```

The implementation file must be a c file, in this example we use a file called`callbacks.c`. This file must be implemented implemented in a similar way :

```
#include <stdlib.h>
#include "rmod_random__s.h"


rmod_random_produce__rs* rmod_random_produce__cb(rmod_random
                                                 oe_obj, double *rs,
                                                 CORBA_Environment *oe_env)
{
  *rs = (double) rand();

  return (rmod_random_produce__rs*) NULL;
}


rmod_random_init__rs* rmod_random_init__cb(rmod_random oe_obj,
                                           long* seed1, long* seed2,
                                           long* seed3,
                                           CORBA_Environment *oe_env)
{
  srand(*seed1 * *seed2 * *seed3);

  return (rmod_random_init__rs*) NULL;
}
```

Compiling the code :

- Please read the *ReadMe* file att the *ic-3.2/examples/c-server* directory
  In the same directory you can find all the code for this example

Note:
Due to changes in *Erl_Interface*, to allow buffer expansion, a new receiving function *ei_receive_encoded/5* is created, while changes in CORBA_Environment initialization are applied. You *must* consider and adapt these. The example in the *ic-3.2/examples/c-server* directory demonstrates the changes.

Running the example :

- Please check the *ReadMe* file att the *ic-3.2/examples/c-server* directory
  In the same directory you can find all the code for this example

1.8 Programming your own composit function in C

CORBA_Environment setting

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 outpuy
    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 semantically divided into three areas :

- The Corba Compatibility area, the area demanded by the standard OMG IDL mapping v2.0
- The External Implementation area, the implementation part used for standard implementation of the generated client/server model.
- The Internal Implementation area, the implementation part usefull for those who wish to define their own composit/switch functions.

Observe that the advanced user wishing to write own composit functions must have good knowledge of Erl_interface or/and EI-* functions.

The Corba Compatibility area of CORBA_Environment

Containes only one (1) field, the _major which is defined as a CORBA_Exception_type. The CORBA_Exception type is forced to be an integer type due to implementation details and in the current version 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 looks like :

```
#ifndef CORBA_NO_EXCEPTION
#define CORBA_NO_EXCEPTION      0
#endif

#ifndef CORBA_SYSTEM_EXCEPTION
#define CORBA_SYSTEM_EXCEPTION -1
#endif
```

The External Implementation area of CORBA_Environment

This area contains nine (*9*) 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 allways 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

The Internal Implementation area of CORBA_Environment

This area contains eight (8) 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 composit 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 that defines its own composit/switch functions have to update/support these values at a way similat to the use of these in the generated code.

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 self set this value to a value bigger than 32.
- *ev->_to_pid = NULL*;
- *ev->_from_pid = NULL*;

- By using *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 wished size of input buffer
  - *outbufsz* is the wished 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*.

Setting System Exceptions

If the advanced 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 on 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 on 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”

Guidlines for the advanced user :

Here are some guidelines for the composit function programmer:

- Try to define buffers for input/output that are big enough to host the coresponding data. If the buffers are not big enough, the stub will reallocate the buffers which cost under runtime.
- Set the exceptions by using the function CORBA_exc_set/4
- Set exceptions only when really needed. Do not overuse system exceptions.

- Always free the CORBA_Environment exception fields by use of CORBA_exception_free/1 after a system failure.
- Look att the examples in the *examples/c-client* and *examples/c-server* directories. The code is tested and follows the suggested application paradigm.

1.9 IDL to Java language mapping

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

Specialities in the mapping

Names reserved by the compiler

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

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

continued ...

... continued

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 1.9: OMG IDL basic types

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.

Mapping for constants

Constants are mapped according to the standard.

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;
  }

}
```

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.

Access to attributes

Attributes are supported according to the standard.

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`.

Communication toolbox

The generated client and server stubs are using the classes defined in the `jinterface` package to communicate whith other nodes. Among other, the most important classes are :

- `OtpInputStream` which is the stream class used for incoming message storage
- `OtpOutputStream` which is the stream class used for outcoming 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 contains 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 register the server node to `epmd` daemon.
  - `OtpConnection accept()`, which waits for a connections and returns the OtpConnection object which is unique for each client node.

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 depended classes :
  - Environment
  - Holder
- Erlang compatibility classes :
  - Pid, PidHelper, PidHolder

    The Pid class inherits from `OtpErlangPid` and is used to represend 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 inherits from `OtpErlangRef` and is used to represend 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 inherits from `OtpErlangPort` and is used to represend 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 inherits from `Any` and is used to represend 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`.

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 the same way as the Any type does.

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

- `boolean isAtom()` returns `true` if the Term is an OtpErlangAtom, `false` otherwize
- `boolean isConstant()` returns `true` if the Term is not an OtpErlangList nor an OtpErlangTuple, `false` otherwize
- `boolean isFloat()` returns `true` if the Term is an OtpErlangFloat, `false` otherwize
- `boolean isInteger()` returns `true` if the Term is an OtpErlangInt, `false` otherwize
- `boolean isList()` returns `true` if the Term is an OtpErlangList, `false` otherwize
- `boolean isString()` returns `true` if the Term is an OtpErlangString, `false` otherwize
- `boolean isNumber()` returns `true` if the Term is an OtpErlangInteger or an OtpErlangFloat, `false` otherwize

- `boolean isPid()` returns `true` if the Term is an OtpErlangPid or Pid, `false` otherwize
- `boolean isPort()` returns `true` if the Term is an OtpErlangPort or Port, `false` otherwize
- `boolean isReference()` returns `true` if the Term is an OtpErlangRef, `false` otherwize
- `boolean isTuple()` returns `true` if the Term is an OtpErlangTuple, `false` otherwize
- `boolean isBinary()` returns `true` if the Term is an OtpErlangBinary, `false` otherwize

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

Client stub initialization, methods exported

The recommended constructor function for client stubs will take 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 refering to. This is usefull when building asynchroinous clients.
- `java.lang.Object __server()`, which returns the server for the current connection.

Server skeleton initialization, server stub implementation, methods exported

The constructor function for server skeleton will take no parameters.

The server skeleton file will contain 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 will export 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.

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 {
        :
        :
```

```
   };

};
```

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.

1.10 IDL Compiler Release Notes

IC 4.0.5, Release Notes

Improvements and new features

- The pragma code option is extended to point specific functions on NOC backent, not only interfaces.

Fixed bugs and malfunctions

- -

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 4.0.4, Release Notes

Improvements and new features

- -

Fixed bugs and malfunctions

- A bug in pragma prefix when including IDL files is fixed. This caused problems for erlang-corba IFR registrations.
  Own Id: OTP-3620

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 4.0.3, Release Notes

Improvements and new features

- Limited support on multiple file module definitions.
  The current version supports multiple file module definitions all backends except the c oriented backends.
  Own Id: OTP-3550

Fixed bugs and malfunctions

- -

Incompatibilities

-

Known bugs and problems

- Multiple file definition of a module is not supported on c oriented backends.
- Type definitions on multiple file module level are limited to containers, such as modules and interfaces. This is true on corba and Erlang backends.

IC 4.0.2, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- A bug is fixed on Erlang backends.
  The (recently) introduced generation of files describing sequence and array files were even true for included interfaces. In the case of some Erlang backends this were unnecessary.
  Own Id: OTP-3485

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 4.0.1, Release Notes

Improvements and new features

- New functionality added on Java and Erl_genserv backends.
  - On the Java client stub :
    - The Java client have now one more costructor function, that allows to continue with an allready started connection.
    - `void __stop()` which sends a stop cast call to the server. While this causes the Erlang server to terminate, it sets a stop flag to the Java server environment, requesting the server to terminate.
    - `void __reconnect()` which closes the current client connection if open and then connects to the same server.

    The Environment variable is now declared as `public`.
  - On the Java server skeleton :
    - `boolean __isStopped()` which returns true if a `stop` message where received, false otherwise. The user must check if this function returns true, and in this case exit the implemented server loop.

    The Environment variable is now declared as `protected` which allows the implementation that extends the stub to access it.
  - On the Erlang gen_server stub :
    - `stop(Server)` which yields to a cast call to the standard gen_server `stop` function. This will always terminate the Erlang gen_server, while it will set the stop flag for the Java server stub.

  Own Id: OTP-3433

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 4.0, Release Notes

Improvements and new features

- New types handled by IC.
  The following OMG-IDL types are added in this compiler version :

  - long long
    unsigned long long
    wchar
    wstring

  Own Id: OTP-3331

- TypeCode as built in type and access code files for array and sequence types.
  - As TypeCode is a psevdo-interface, it is now is a built-in type on IC.
  - Access code files which contain information about TypeCode, ID and Name are now generated for user defined arrays and sequences.

  Own Id: OTP-3392

Fixed bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.8.2, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

A bug is fixed on preprocessor directive expansion.

When nested #ifdef - #ifndef directives, a bug caused improper included file expansion. This is fixed by repairing the preprocessor expansion function.

Own Id: OTP-3472

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.8.1, Release Notes

Improvements and new features

- Build in Erlang types support for java-backends
  The built-in Erlang types `term, port, ref` and `pid` are needed in Java backends in order to support an efficient mapping between the two languages. The new types are also supported by additional helpers and holders to match with OMGs Java mapping As a result of this, the following classes are added to the `com.ericsson.otp.ic` interface :
  - `Term,TermHelper,TermHolder` which represents the built-in Erlang type `term`
  - `Ref,RefHelper,RefHolder` which represents the built-in Erlang type `ref`
  - `Port,PortHelper, PortHolder` which represents the built-in Erlang type `port`
  - `Pid, PidHelper and PidHolder` which represents the built-in Erlang type `pid`

  Own Id: OTP-3348
- Compile time preprocessor macro variable definitions
  The preprocessor lacked possibility to accept user defined variables other than the one defined in IDL files. This limited the use of command-ruled IDL specifications. Now the build-in preprocessor allows the user to set variables by using the “preproc_flags” option the same way as using the “gcc” preprocessor.
  Supported flags :
  - `"-D< Variable >"` which defines a variable
  - `"-U< Variable >"` which undefines a variable

  Own Id: OTP-3349

Fixed bugs and malfunctions

A bug on comment type expansion is fixed.

The comment type expansion were errornous when inherited types (NOC backend). This is now fixed and the type naming agree with the scope of the inheritor interface.

Own Id: OTP-3346

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.8, Release Notes

Improvements and new features

- The code generated for java backend is optimized due to use of streams insead for tuple classes when (un)marshalling message calls. Support for building clients using asynchronous client calls and effective multithreaded servers.
  Own Id: OTP-3310
- The any type is now supported for java backend.
  Own Id: OTP-3311

A bug on C generated constants is fixed

While the constants are evaluated and behave well when used inside an IDL specification their C-export were not working properly. The constant export definitions were not generated well :

- the declared C definition were errornous (the name did not always agree with the scope the constant were declared in).
- there were no C- definition generated for the c-server backend when the constants were declared inside an interface.

Own Id: OTP-3219

Incompatibilities

Due to optimizations in java backend, the stub initialization and usage differs than the previous version.

Client stub interface changes:

- Client disconnects by calling the `__disconnect()` function instead for the old `_closeConnection()`
- All `marshal` operation functions have now the interface :
  `void _< OpName >_marshal(Environment<, Param |, Params >)`
  instead for
  `OtpErlangTuple _< OpName >_marshal(< Param, | Params, >OtpErlangPid, OtpErlangRef)`
- All `unmarshal` operation functions have now the interface :
  `< Ret value > _< OpName >_unmarshal(Environment<, Param |, Params >)`
  instead for
  `< Ret value > _< OpName >_unmarshal(< Param, | Params, >OtpErlangTuple, OtpErlangRef)`
- Call reference extraction is available by the client function :
  `OtpErlangRef __getRef()`
  instead for previous function :
  `OtpErlangRef _getReference(OtpErlangTuple)`

Server skeleton interface changes:

- The implementation function no longer have to contain the two (2) contructor functions (with `super()`). This is due to the fact that there is only one contructor function for each skeleton file :
  `public _< interface name >ImplBase()`

- The parameter for the caller identity extraction function `_getCallerPid` is now an `Environment` variable instead for an `OtpErlangTuple`.
- There is a new `invoke` function :
  `OtpOutputStream invoke(OtpInputStream)`
  instead for the old one :
  `OtpErlangTuple invoke(OtpErlangTuple)`
- The `OtpConnection` class function used for receiving messages is now :
  `OtpInputStream receiveBuf()`
  instead for the old one :
  `OtpErlangTuple receive()`
- The `OtpConnection` class function used for sending messages is now :
  `void sendBuf(OtpErlangPid, OtpOutputStream)`
  instead for the old one :
  `void send(OtpErlangPid, OtpErlangTuple)`

Known bugs and problems

- The same as in previous version.

IC 3.7.1, Release Notes

Improvements and new features

Some memory usage optimizations for the compiler were done.

Fixed bugs and malfunctions

- A bug is fixed when C backend is used.
  When C-union with enumerant discriminator, the size calculation of the discriminator value were errornous. This lead to the sideeffect that only the first case of the union were alowed. The error were fixed by fixing the size calculation of the discriminator.
  Own Id: OTP-3215

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.7, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- A bug is fixed when C backend is used.
  When unions with enumerant discriminator were decoded, an error encountered in the union size calculation.
  Own Id: OTP-3209

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.6, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- A bug is fixed when NOC backend is used.
  When several functions with the same name were found in the included file tree, a compile time failure occured.
  Own Id: OTP-3203

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.5, Release Notes

Improvements and new features

- Noc backend optimization
  When NOC backend is choosed, the type code information on the stub functions is reduced to a single atom “no_tk”. This is the default behaviour. The typecode generation is enabled by the “use_tk” switch.
  Own Id: OTP-3196

Fixed bugs and malfunctions

- General java backend bugfixes
  Protocol errors on user defined structures and union types are corrected.

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.4, Release Notes

Improvements and new features

- Semantic test enhancements.
  The compiler detects now semantic errors when enumerant values colide with user defined types on the same name scope.
  Own Id: OTP-3157

Fixed bugs and malfunctions

- General java backend bugfixes
  Several bugs were fixed on user defined types.
  - Union discriminators work better when all possible case values are defined.
  - A bug on Interface inherited operations is fixed that cause errors on generated server switch.
  - Type definitions on included files are better generated.

  Own Id: OTP-3156

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.3, Release Notes

Improvements and new features

- A new back-end which generates Java code according to the CORBA IDL to Java mapping for communication with the Erlang distribution protocol has been added to IC. For the moment there is no support for the Erlang types Pid, Ref, Port and Term but this will be added later.
  Own Id: OTP-2779

Fixed bugs and malfunctions

- Fixed the bug that the c code backends sometimes generated incorrect code for struct arguments. They shall always be pointers.
  Own Id: OTP-2732
- The code generation is fixed so the array parameters now follow the CORBA V2.0 C mapping.
  Own Id: OTP-2873
- Fixed the problem that the checking of the numbers of outparameters always was true.
  Own Id: OTP-2944
- Fixed the bug that some temporary variables was not declared when c code.
  Own Id: OTP-2950

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.2.2, Release Notes

Improvements and new features

- Unions are now supported to agree with OMG's C mapping.
  Own Id: OTP-2868
- There is now a possibility to use pre- and postcondition methods on the server side for IC generated Corba Objects. The compiler option is documented in the ic reference manual and an example of how the pre- and postcondition methods should be designed and used is added to ic example directory (an ReadMe.txt file exists with some instructions for running the example code).
  Own Id: OTP-3068

Fixed bugs and malfunctions

- The compiler ignores unknown/non supported pragma directives. A warning is raised while the generated code will then be the same as if the corresponding (unkown) pragma directive were missing.
  Own Id: OTP-3052

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.2.1, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- Wrong C code was generated for limited strings when they where included from another IDL specification.
  Own Id: OTP-3033

Incompatibilities

-

Known bugs and problems

- The same as in previous version.

IC 3.2, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- The buffers for in/output used by C-stubs are now expandable. This fixes buffer overflow problems when messages received/sended do not fit in buffers.
  Own Id: OTP-3001

Incompatibilities

The CORBA_Environment structure has now two new fields, the buffers for in/output must now be dynamically allocated.

Known bugs and problems

- The same as in previous version.

IC 3.1.2, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

- The generated IFR registration function for constants has been fixed so the parameters are correct.
  Own Id: OTP-2856
- Error in the C code generation of ONEWAY operations without parameters The bug was an decoding error in the operation header. The generated code expected one parameter instead of zero. This is now fixed.
  Own Id: OTP-2909
- Type problems on floats and booleans fixed.
  Erroneous code for runtime checks on float was removed and the internal format of the data representing the boolean value is uppgraded.
  Own Id: OTP-2925
- The generated code for arrays of typedefined strings were erroneous in the C-backends due to a failure in the compiler internal type checking.
  Own Id: OTP-2936
- The generated code for typedefined nested sequences were erroneous in the C-backends. Pointer mismatches caused compilation failure.
  Own Id: OTP-2937

Incompatibilities

The IDL specifications must be regenerated for C due to changes in the code generation.

One must regenerate IDL specifications for Erlang CORBA if there are constants in the specification due to previous errors in the IFR registration functions (OTP-2856).

Known bugs and problems

- OMG IDL - C mapping is not consistent on sequnce naming.
  There is som inconsistencies around sequence naming in the specification which must be investigated further.
- Problems with nested sequences
  Nested sequences on the form:

  ```
  typedef sequence&lt;sequence&lt;long&gt; &gt; ex;
  ```

  are not generated correctly.
  Nested sequences can be used if the innermost sequence is separately typedefined.

  ```
  typedef sequence&lt;long&gt; lseq;
  typedef sequence&lt;lseq&gt; ex;
  ```

IC 3.1.1, Release Notes

Improvements and new features

- Improvements on error repport on unsupported types by propagating warning when declaring unions in C -backends

Fixed bugs and malfunctions

- A bug is fixed when arrays that contained variable size data on C-backends
  The compiler generated errornous code when IDL defined arrays that contained variable size data such as strings, varible size structs or sequences.
  Own Id: OTP-2900
- A bug is fixed when sequences that contained variable size data on C_backends
  The compiler generated errornous code when IDL defined arrays that contained variable size data such as strings, variable size structs or other sequences.
  Own Id: OTP-2901
- A bug concerning bounded strings on C-backends is fixed.
  The compiler generated errornous code for IDL defined bounded strings. Syntax errors were generated in special cases of typdedefined strings.
  Own Id: OTP-2898
- A runtime error when sequences that contained integer types is fixed.
  When C-clients/server that communicated with Erlang clients/servers, and the data send by Erlang part were a list of small numbers, the Erlang runtime compacts the list to a string. This caused a runtime error when sending sequences of integer types and all had value less than 256.
  Own Id: OTP-2899
- An OMG IDL - C mapping problem on enumerant values is fixed.
  The enumerant values names is now prefixed by the current scope, as defined in the specification.
  Own Id: OTP-2902
- A problem when using constants in array declarations is fixed.
  Array dimentions declared with contants generated erroneous code.
  Own Id: OTP-2864

Incompatibilities

- Changes in C-generation on enumerant values.

Known bugs and problems

- OMG IDL - C mapping is not consistent on sequnce naming.
  There is som inconsistencies around sequence naming in the specification which must be investigated further.

IC 3.1, Release Notes

Improvements and new features

- No new features are added

Changes in compiler usage and code generation.

- No changes since last version.

Fixed bugs and malfunctions

- A bug is fixed on the generated structures.
  The generated C code for the structures corresponds now to direct mapping of C-structs.
  Own Id: OTP-2843

Incompatibilities

- Included structures inside a struct are no longer pointers.

Known bugs and problems

- Runtime error when list that contain longs, shorts
  When C-clients/server that communicates with Erlang clients/servers, and the data send by Erlang part is a list of small numbers, the Erlang runtime compacts the list to a string
  This is only actual in case of numbers with value less than 256
- Compiler failure when arrays that contain dynamic data.
  The compiler fails to compile IDL defined arrays that contain complex data.

IC 3.0, Release Notes

Improvements and new features

- Interface change for C-backends
  Major interface change. The new interface is CORBA 2.0 compliant.
  Own Id: OTP-2845
- The C-backends functionality is improved
  - Due to interface change and some unneeded error checks,the C-generated code is fairly optimized.

Changes in compiler usage and code generation.

- No changes since last version.

Fixed bugs and malfunctions

- Several serious bugs on decoding and memory allocation are fixed.

Incompatibilities

- Interface change on the C-backends
  In order to be CORBA 2.0 compatible, the new version generates fully incompatible C code.

Known bugs and problems

- The same as in version 2.5.1

IC 2.5.1, Release Notes

Improvements and new features

- A new backend is added : C-server
  This back-ends can be used to create servers, compatible to c-clients, and Erlang genserver clients. The code produced is a collection of functions for encoding and decoding messages and a switch that coordinates them. These parts can be used to create other servers as well. All functions are exported to header files.
  Own Id: OTP-2713
- The C-client functionality is improved
  - The static buffer used for input/output is removed along with the `memset` function that initiated it. The new client is at least 20-30 procent faster.
  - The internal structure of the client is changed. The client fuctions are now a collection of encoding and decoding message functions ruled by a specific call function. While the basic client generated is a synchronous client, the exported functions support the implementation of threaded asynchonous clients.
  - The static buffer used for input/output is remove along with the `memset` function that initiated it. The new client is at least 20-30 procent faster.
  - The code generated is generally improved, warnings are (almost) eliminated, while no unidentified variable errors occur.
  - The IDL types unsigned shorts, shorts, floats are supported now.
  - All generated functions are exported in client header files..

  Own Id: OTP-2712

Changes in compiler usage and code generation.

- A new option is added for the C-server back-end : `c_server`.
- A new option is added : `scoped_op_calls`.

Fixed bugs and malfunctions

- A bug oneway operations on erl_corba and erl_genserv that caused en exit due to internal interface error is fixed.
- A bug on oneway operations on c_genserv back-end that caused several variables to be unidentifined is fixed.

Incompatibilities

- Interface change on the C-client
  The client functions are called with two extra variables, a pointer to an array of char - used for storage and an integer - the array size
- The IDL type `attribute` is disabled, due to some implementation problems.

Known bugs and problems

- The same as in version 2.1

IC 2.1, Release Notes

Improvements and new features

- The compiler now provides more in depth information (outprints) when errors occur.
  In some cases the compiler stops compiling due to an abnormal exit or incompatible input. In this situation, a “fatal error” may occur but the compiler will generate information explaining the problem.
  Own Id: OTP-2565

Changes in compiler usage and code generation.

- No changes since version 2.0

Fixed bugs and malfunctions

- No changes since version 2.0

Incompatibilities

- The same as in version 2.0

Known bugs and problems

- The same as in version 2.0

IC 2.0, Release Notes

Improvements and new features

- The IDL compiler is now a separate application and is longer a part of Orber.
- Pragma handling implementation.
  Pragma ID, prefix and version are implemented to agree with CORBA revision 2.0. The compiler accepts and applies these on the behavior of the compiled code.
  In this implementation, pragmas are accepted by the parser and applied by the use of ic_pragma functions.
  All IFR-identity handling now passes through pragma table. As pragma handling in OMG-IDL is affecting the identity of an ifr-object, all identity handling and registration is now controlled by pragma functions. A hash table called “pragmatab” contains vital identity information used under compilation.
  There two major pragma categories :
  - Normal pragmas, are used in the code where basic definitions and statements appear.

– Under certain circumstances, ugly pragmas can now appear inside code, parameter lists, structure definitions ... etc.
It is quite challenging to allow ugly pragmas, but the effects of unlimited ugly pragma implementation on the parser can be enormous. Ugly pragmas can cause the parser source code to become time consuming and user unreadable.
In order to allow ugly pragmas but not destroy the current structure of the parser, the use of ugly pragmas is limited. Multiple pragma directives are allowed inside parameter lists, unions, exceptions, enumerated type, structures... as long as they are do not appear between two keywords or between keywords and identifiers.

The pragma effect is the same for both scope and basic pragma rules.
When compiling, an IFR-identity must be looked up several times but by storing identity aliases inside the pragma table there this an increase in both speed and flexibility.
Own Id: OTP-2128

- Code for interface inheritance registration for the IFR registration code .
Inherited interfaces can now be registered as a list of interface descriptions by entering code for inherited interface registration under new interface creation. This is achieved by correcting the function reg2/6 and adding two more functions, get_base_interfaces/2 and call_fun_str/2
Own Id: OTP-2134
- IFR registration checks for included IDL files.
All top level definitions (with respect to the scope) - modules, interfaces, constants, types or exceptions - found in an IDL file are either defined inside the compiled IDL file or inside included files. By having an extended registration of all top level definitions it becomes possible to simply produce checks for those included by the current IDL file. A function call include_reg_test/1 is added in all OE_* files that checks for IFR-registration on all included IDL files. The code for that function is added inside the OE_* file, while the function is called under OE_*:OE_register/0 operation.
Own Id: OTP-2138
- Exception registration under IFR-operation creation.
By entering code for exception registration under operation creation, the exceptions of an operation can be checked now. This is done by correcting the function get_exceptions/4 and adding two more functions, excdef/5 and get_EXC_ID/5 (the last two are cooperating with the first one and all three are defined in the module “ictk”).
Own Id: OTP-2102
- New back-end to IDL compiler : Plain Erlang.
The new back-end just translates IDL specifications to Erlang module calls. No pragmas are allowed.
Own Id: OTP-2471
- New back-end to IDL compiler : generic server.
A new back-end that translates IDL specifications to a standard OTP generic server.
Own Id: OTP-2482
- New back-end to IDL compiler : c client generation
A new back-end that translates IDL specifications to a C API for accessing servers in Erlang.
Own Id: OTP-1511
- All records in generated files reveal own Erlang modules.
In Erlang related back-ends, every structure which generates definition form is a record, (such as union, struct, exception....). These records are held in a generated Erlang files which contain functions that reveal record information.
The Erlang file which contain these functions is named after the scope of the record (similar to the generated module and interface files).

Three functions are available :

- tc/0 - returns the record type code,
- id/0 - returns the record id,
- name - returns the record name.

Own Id: OTP-2473

- Changes in compiler usage and code generation.
  - New compilation flags. New flag be (= back-end) which is used by the compiler to choose back-end. Default back-end is set to erl_corba.
  - Stub files have an extra function oe_dependency/0 indicating file dependency. This helps the user to determine which IDL files should to be compiled beside the compiled file.

  Own Id: OTP-2474
- The IDL generation for CORBA is changed so standard gen_server return values can be used from the implementation module. The change is compatible so that old values remain valid.
  Own Id: OTP-2485
- It's now possible to generate an API to a CORBA object that accepts timeout values in the calls in the same manner as gen_server. The option to the compiler is "timeout".
  Own Id: OTP-2487

Fixed bugs and malfunctions

- Empty file generation problem is fixed. When the IDL module definition did not contain constant definitions, the generated stub file for that module definition was empty. After checking the module body, these files will not be generated anymore.

Incompatibilities

- Changes in generated files.
  Stub-files generated by the compiler had prefix "OE_" and those used by Orber had also a register/unregister function called "OE_register"/"OE_unregister" and a directive "OE_get_interface" passed to the gen_server. This made it difficult/irritating to use, for example call to the register function in Orber would appear as shown below:
  - 'OE_filename':'OE_register'().

  This is changed by using the prefix "oe_" instead for "OE_" for the above. A registration call in Orber is now written:
  - oe_filename:oe_register().

  Own Id: OTP-2440

Known bugs and problems

- No checks are made to ensure reference integrity. IDL specifies that identifiers must have 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 has 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 when the discriminator type is written using a type reference.
- Parser failure with syntax error when “standard” preprocessor directives such as #ifndef and #include (not pragmas) are used in other than top level scope.

Previous Release Notes

For release notes on previous versions see the release notes on Orber (version previous to 1.0.3).

IC Reference Manual

Short Summaries

CORBA_Environment_alloc

The following functions are exported:

- `CORBA_Environment * CORBA_Environment_alloc(inbufsz, outbufsz)`
  Initializes communication

ic

The following functions are exported:

- `ic:gen(FileName) -> Result`
  [page 91] Generate stub and server code according to OMG/CORBA 2.0.
- `ic:gen(FileName, [Option]) -> Result`
  [page 91] Generate stub and server code according to OMG/CORBA 2.0.

CORBA_Environment_alloc (C Module)

The *CORBA_Environment_alloc()* function is the function used to allocate and initiate the *CORBA_Environment* structure.

Exports

`CORBA_Environment * CORBA_Environment_alloc(inbufsz, outbufsz)`

Types:

- int inbufsz;
- int outbufsz;

This funtion is used to create 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 wished size of input buffer.

outbufsize is the wished size of 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 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.

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 outpuy
    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:**
> Remember to set the field values *_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*.

SEE ALSO

ic(3)

ic (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 or C. The compiler generates client stub code and server behaviors.

Two kinds of files are generated for each scope, Erlang/C files and Erlang/C header files. Headers are used to store record definitions, while usual Erlang/C files contain the object interface functions, the object server or access functions for records defined in interfaces.

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()} | {{this, String()}, bool()} |
- {{handle_info, String()}, bool()} | {timeout, String()} |
- {scoped_op_calls, bool()} | {scl, 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, erl_corba | erl_plain | erl_genserv | c_genserv | c_client | c_server } |
-
- DirNAme = string() | atom()
-
- FileName = string() | atom()

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

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:  ic:gen(x, [{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:  ic:gen(x, [{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`

```
Example1:  ic:gen(x, [{preproc_cmd, "erl"}])
Example2:  ic:gen(x, [{preproc_cmd, "gcc -x c++ -E"}])
```

preproc_flags Flags given to the preprocessor.

```
Example:  ic:gen(x, [{preproc_flags, "-I../include"
```

Code options

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:  ic:gen(x, [this])` activates the parameter for all interfaces.

`Example:  ic:gen(x, [{this, "M1::I1"}])` activates the parameter for all functions of `M1::I1`.

`Example:  ic:gen(x, [this, {{this, "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: ic:gen(x, [handle_info])` will activates module implementation `handle_info` for all interfaces in the source file.

`Example: ic:gen(x, [{{handle_info, "M1::I1"}, true}` will activates module implementation `handle_info` for the specified interface.

`Example: ic:gen(x, [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: ic:gen(x, [{time_out,"M::I"}])` produces server stub which will has an extra timeout parameter in the initialization function for that interface.

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 coalition 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: ic:gen(x, [{be, c_genserv},{scoped_op_calls,true}])` produces client stub which sends “scoped” requests to the a gen_server or a c-server.

scl Used for compatibility with previous compiler versions up to `3.3`. Due to better semantic checks on enumerants, the compiler discovers name coalitions 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: ic:gen(x, [{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: ic:gen(x, [{precond, {mod, fun}}])` adds the call of m:f for all operations in the idl file.

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

`Example: ic:gen(x, [{{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: ic:gen(x, [{postcond, {mod, fun}}])` adds the call of m:f for all operations in the idl file.

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

Example: `ic:gen(x, [{{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 Suppress all warnings. Default value is false.

warn_name_shadow Warning appear 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

All back-end options are declared as a tuple {`be,atom()`}, followed eventually by back-end specific options:

erl_corba This option switches to the IDL generation for CORBA.

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_genserv Will produce a C client to the generic Erlang server.

c_client Will produce a C client to the generic Erlang server.

Please note that this option have the same action as the c_genserv option. It is supposed to gradually replace the c_genserv option. For a limited period of time both options will be supported.

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

Default back-end option is {`be, erl_corba`}.

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:

1. Configuration file
2. 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.

List of Tables

Chapter 1: IC User's Guide

Glossary

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`.