Spectra ORB
C Edition
Reference Guide

Version 2.1

PRISMTECH
Productivity Tools & Middleware

Guide edition: 04 (1 September 2016)

Table of Contents

1 Preface

1.1 About the Reference Guide

The Reference Guide is the reference for the Spectra ORB C Edition's Application Programmers Interface (API) function signatures and the standard exceptions.

The Reference Guide is intended to be used with the Spectra ORB C Edition, Product, IDL and User Guides, as well as the other documents included with the product; please refer to the Product Guide for a complete list of documents.

1.1.1 Intended Audience

The *Reference Guide* is intended to be used by developers and engineers working in a distributed computing environment using Spectra ORB.

1.1.2 Organisation

The *Introduction* describes the conventions used for the API definitions.

Chapter *2, General Information*, lists definitions and exception codes.

Chapter *3, Standard API Reference*, describes the functions and exceptions.

1.2 Conventions

The conventions listed below are used to guide and assist the reader in understanding this Guide.

Item of special significance or where caution needs to be taken

i Item contains helpful hint or special information.

WIN Information applies to Windows (*e.g.* XP, Vista, Windows 7) only.

UNIX Information applies to Unix based systems (*e.g.* Solaris) only.

C C language specific.

C++ C++ language specific.

Java Java language specific.

Hypertext links are shown as *blue italic underlined*.

On-Line (PDF) versions of this document: Items shown as cross references to other parts of the document, *e.g.* *1.3 Contacts* on page 7, behave as hypertext links: jump to that section of the document by clicking on the cross reference.

```
% Commands or input which the user enters on the command
line of their computer  terminal
```

`Courier`, **`Courier Bold`**, or *`Courier Italic`* fonts indicate programming code. The `Courier` font can also indicate file names.

Code fragments are shown as small `Courier` font in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to "example" and
// kind field to an empty string

newName[0] = new NameComponent ("example", "");
rootContext.bind (newName, demoObject);
```

Italics and ***Italic Bold*** indicate new terms, or emphasise an item.

Sans-serif Bold indicates user-related actions, *e.g.* **File > Save** (a sequence of selections from menus, or buttons or check-boxes).

Step 1: one of several steps required to complete a task.

1.3 Contacts

PrismTech can be contacted at the following contact points.

Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900

Fax: +44 (0)191 497 9901

Web: *http://www.prismtech.com*

Technical questions: *technical-support@prismtech.com*

Sales enquiries: *sales@prismtech.com*

2 General Information

This section provides common mappings, definitions and exceptions.

2.1 Common Mappings for IDL Types

Table 1 is provided here for convenience: please refer to the *Spectra ORB C Edition IDL Handbook* for the complete mapping descriptions.

Table 1 Common IDL Type Definitions

Type	CORBA Type
unsigned 32 bit integer	CORBA_unsigned_long
signed 32 bit integer	CORBA_long
8 bit character	CORBA_char
signed 16 bit integer	CORBA_short
unsigned 16 bit integer	CORBA_unsigned_short
unsigned 8 bit integer	CORBA_octet
unsigned 8 bit integer	CORBA_boolean
unsigned 64 bit integer	CORBA_unsigned_long_long
signed 64 bit integer	CORBA_long_long
32 bit floating point	CORBA_float
64 bit floating point	CORBA_double

2.2 Definitions

Table 2 Definitions

Item	Definition
boolean type	`#define TRUE 1` `#define FALSE 0`
NIL object	`#define CORBA_OBJECT_NIL NULL`
CORBA ORB	`typedef CORBA_Object CORBA_ORB;`

Exception completion status codes	`typedef CORBA_unsigned_long CORBA_completion_status;`
	
	`#define CORBA_COMPLETE_YES 0`
	`#define CORBA_COMPLETE_NO 1`
	`#define CORBA_COMPLETE_MAYBE 2`

CORBA system exception	`typedef struct`
	`{`
	`  CORBA_unsigned_long minor;`
	`  CORBA_completion_status completed;`
	`}`
	`CORBA_SystemException;`

Major Exception Codes	`#define CORBA_NO_EXCEPTION 0`
	`#define CORBA_USER_EXCEPTION 1`
	`#define CORBA_SYSTEM_EXCEPTION 2`

ObjectId (strings that identify the object whose reference is required)	`typedef struct`
	`{`
	`  CORBA_unsigned_long _maximum;`
	`  CORBA_unsigned_long _length;`
	`  CORBA_ORB_ObjectId * _buffer;`
	`}`
	`CORBA_ORB_ObjectIdList;`

ORB Id	`typedef char * CORBA_ORBid;`

Any Type	`typedef struct`
	`{`
	`  CORBA_TypeCode _type;`
	`  void * _value;`
	`  CORBA_boolean _release;`
	`}`
	`CORBA_any;`

3 Standard API Reference

This section contains the standard function signatures and exceptions for the ORB's Application Programmers Interface (API). The section also provides information about policies and the `Policy` *interface which allows access to policies that affect its operation.*

i The ORB conforms to the Minimum CORBA Specification v1.0, as aligned to the full CORBA Specification v.2.3, and the OMG's IDL to C Mapping Specification. Please note that although the ORB conforms to the 2.3 version of the CORBA Specification, it also contains functions and other elements which are:

- included in later (i.e. newer) OMG CORBA specifications
- proprietary to Spectra ORB C Edition

The API descriptions provided in this section will include a note of the CORBA specification version for those functions or elements supported by CORBA Specifications which are newer than the 2.3 version. Functions which are proprietary ORB functions generally have names which are prefixed with *`EORB_`*, for example, *`EORB_`*`alloc`.

3.1 Function Signatures

3.1.1 CORBA Module

3.1.1.1 Environment

Header: `CORBA/Environment.h`

CORBA_Environment__alloc

```
CORBA_Environment * CORBA_Environment__alloc (void)
```

Allocates storage and initializes the `CORBA_Environment` structure. The C mapping does not define a way to initialize an environment structure declared on the stack. This can be done by using the `memset` function to initialize the variable to zero.

CORBA_Environment__free

```
void CORBA_Environment_free (CORBA_Environment * ev)
```

Frees any storage that was allocated to the `CORBA_Environment` structure by the `CORBA_Environment__alloc` function.

3.1.1.2 Exceptions

Header: `CORBA/exception.h`

CORBA_exception_as_any

```
CORBA_any * CORBA_exception_as_any (CORBA_Environment * ev)
```

CORBA_exception_as_any returns a pointer to a CORBA_any containing the exception. If invoked on a CORBA_Environment which identifies a non-exception, a null pointer is returned. Note that the ownership of the returned pointer does not transfer to the caller; instead the pointer remains valid until CORBA_exception_free is called.

CORBA_exception_free

```
void CORBA_exception_free (CORBA_Environment * ev)
```

CORBA_exception_free clears any exception associated with the the environment and releases any associated storage.

CORBA_exception_id

```
CORBA_char * CORBA_exception_id (CORBA_Environment * ev)
```

CORBA_exception_id returns a pointer to the character string identifying the exception. The character string contains the repository ID for the exception. If invoked on a environment variable which identifies a non-exception, (_major equals CORBA_NO_EXCEPTION) a null pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until CORBA_exception_free is called.

CORBA_exception_set

```
void CORBA_exception_set
(
   CORBA_Environment * ev,
   CORBA_exception_type major,
   const CORBA_char * except_repos_id,
   void * param
)
```

CORBA_exception_set allows a function implementation to raise an exception. The ev parameter is the environment parameter passed into the function. The caller must supply a value for the major parameter. The value of the major parameter constrains the other parameters in the call as follows:

- If the major parameter has a value of CORBA_NO_EXCEPTION, then this is a normal outcome to the operation. In this case, both except_repos_id and param must be NULL. Note that it is *not* necessary to invoke CORBA_exception_set to indicate a normal outcome; it is the default behaviour if the function simply returns.
- For any other value of major it specifies either a user-defined or system exception. The except_repos_id parameter is the repository ID representing the exception type. If the exception is declared to have members, the param parameter must be the address of an instance of the exception struct containing the parameters according to the C language mapping, coerced to a void*. In this case, the exception struct must be allocated using the appropriate T__alloc function, and the CORBA_exception_set function adopts the allocated memory and frees it when it no longer needs it. Once the allocated exception struct is passed to CORBA_exception_set, the application is not allowed to access it because it no longer owns it. If the exception takes no parameters, param must be NULL.

If the CORBA_Environment argument to CORBA_exception_set already has an exception set in it, that exception is properly freed before the new exception information is set. See the exception example code for typical usage.

CORBA_exception_value

```
void * CORBA_exception_value (CORBA_Environment * ev)
```

CORBA_exception_value returns a pointer to the structure corresponding to this exception. If invoked on a CORBA_Environment that identifies a non-exception or an exception for which there is no associated information, a null pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until CORBA_exception_free is called.

3.1.1.3 Fixed Types

Header: CORBA/fixed.h

CORBA_fixed_alloc

```
void * CORBA_fixed_alloc (CORBA_unsigned_short digits)
```

CORBA_fixed type storage allocator. Allocates storage for the CORBA_fixed type.

Parameters

digits	number of digits in fixed value

CORBA_fixed_fraction_part

```
CORBA_long CORBA_fixed_fraction_part (const void * fixed)
```

Returns the fraction (scale) part of a fixed point decimal.

CORBA_fixed_integer_part

```
CORBA_long CORBA_fixed_integer_part (const void * fixed)
```

Returns the integer (digits) part of a fixed point decimal.

CORBA_fixed_set

```
void CORBA_fixed_set
(
   void * fixed,
   const CORBA_long ipart,
   const CORBA_long fpart
)
```

Sets the integer (digits) and fractional (scale) part of a fixed point decimal.

Parameters

fixed	pointer to a fixed point decimal
ipart	integer (digits) part
fpart	fraction (scale) part

CORBA_fixed_add

```
void CORBA_fixed_add (void * rp, const void * f1p, const void * f2p)
```

Performs the addition of 2 fixed-point decimals and stores the result in `rp`.

Parameters

rp	pointer to a fixed point decimal (will hold result of operation)
f1p	pointer to a fixed point decimal
f2p	pointer to a fixed point decimal

CORBA_fixed_sub

```
void CORBA_fixed_sub (void * rp, const void * f1p, const void * f2p)
```

Performs the subtraction of two fixed-point decimals and stores the result in `rp`.

Parameters

rp	pointer to a fixed point decimal (will hold result of operation)
f1p	pointer to a fixed point decimal
f2p	pointer to a fixed point decimal

CORBA_fixed_mul

```
void CORBA_fixed_mul (void * rp, const void * f1p, const void * f2p)
```

Performs the multiplication of two fixed-point decimals and stores the result in `rp`.

Parameters

rp	pointer to a fixed point decimal (will hold the result of operation)
f1p	pointer to a fixed point decimal
f2p	pointer to a fixed point decimal

CORBA_fixed_div

```
void CORBA_fixed_div (void * rp, const void * f1p, const void * f2p)
```

Performs the division of two fixed-point decimals and stores the result in `rp`.

Parameters

rp	pointer to a fixed point decimal (will hold the result of operation)
f1p	pointer to a fixed point decimal
f2p	pointer to a fixed point decimal

3.1.1.4 General

CORBA_free

Header: `CORBA.h`

```
void CORBA_free (void * ptr)
```

This function frees memory allocated with one of the ORB allocation functions and calls any associated destructor function. Note the destructor function is called before the memory is actually freed.

CORBA_PolicyList__alloc

Header: `CORBA/Policy.h`

```
CORBA_PolicyList * CORBA_PolicyList__alloc (void)
```

Allocate storage for a policy list.

CORBA_PolicyList_allocbuf

Header: `CORBA/Policy.h`

```
CORBA_Policy * CORBA_PolicyList_allocbuf (CORBA_unsigned_long len)
```

Allocate storage for `len` elements of a policy list.

Parameters

`len` number of elements to allocate storage for

3.1.1.5 Sequences

Header: `CORBA/sequence.h`

CORBA_PolicyTypeSeq__alloc

Header: `CORBA/Object.h`

```
CORBA_PolicyTypeSeq * CORBA_PolicyTypeSeq__alloc (void)
```

Allocates storage for a sequence of policy types.

CORBA_sequence_set_release

```
void CORBA_sequence_set_release (void * sequence, CORBA_boolean release)
```

Sequence types support the notion of ownership of their `_buffer` members. By setting a release flag in the sequence when a buffer is installed, programmers can control ownership of the memory pointed to by `_buffer`. The location of this release flag is implementation-dependent. The two ORB-supplied functions allow for the setting and checking of the sequence release flag:

`CORBA_sequence_set_release` can be used to set the state of the release flag. If the flag is set to `TRUE`, the sequence effectively "owns" the storage pointed to by `_buffer`; if `FALSE`, the programmer is responsible for the storage. If, for example, a sequence is returned from an operation with its release flag set to `FALSE`, calling `CORBA_free` on the returned sequence pointer will not deallocate the memory pointed to by `_buffer`. Before calling `CORBA_free` on the `_buffer` member of a sequence directly, the programmer should check the release flag using `CORBA_sequence_get_release`. If it returns `FALSE`, the programmer should not invoke `CORBA_free` on the `_buffer` member; doing so produces undefined behaviour. Also, passing a null pointer or a pointer to something other than a sequence type to either `CORBA_sequence_set_release` or `CORBA_sequence_get_release` produces undefined behaviour.

CORBA_sequence_set_release should only be used by the creator of a sequence. If it is not called for a given sequence instance, then the default value of the release flag for that instance is FALSE.

Two sequence types are the same type if their sequence element type and size arguments are identical.

3.1.1.6 Strings

Header: CORBA/string.h

CORBA_string_alloc

```
CORBA_char * CORBA_string_alloc (CORBA_unsigned_long length)
```

Strings are dynamically allocated using this function, which allocates len+1 bytes (length of the string plus the NULL termination character). The client should never pass an inout string parameter that was not allocated using CORBA_string_alloc. The client is responsible for freeing the allocated storage using CORBA_string_free, regardless of whether or not a reallocation was necessary.

Strings allocated in this manner are freed using CORBA_string_free.

CORBA_string_dup

```
CORBA_char * CORBA_string_dup (CORBA_char * str)
```

The string_dup function dynamically allocates enough space to hold a copy of its string argument, including the NULL character, copies its string argument into that memory, and returns a pointer to the new string. If allocation fails, a null pointer is returned.

CORBA_string_free

```
void CORBA_string_free (CORBA_char * str)
```

Frees the storage for a string allocated by the CORBA_string_alloc function.

3.1.1.7 Any Interface

Header: CORBA/any.h

CORBA_any_alloc

```
CORBA_any * CORBA_any_alloc (void)
```

This allocates a new any type. The memory for this is freed by calling CORBA_free.

CORBA_any_get_release

```
CORBA_boolean CORBA_any_get_release (CORBA_any * any)
```

Gets the ownership of the value contained within the any. See *CORBA_any_set_release* below for details.

CORBA_any_set_release

```
void CORBA_any_set_release (CORBA_any * any, CORBA_boolean release)
```

CORBA_any_set_release can be used to set the state of the release flag. If the flag is set to TRUE, then the any owns the memory storage pointed to by _value and must therefore manage it; if FALSE, then the programmer is responsible for the memory storage. For example, if an any is returned from an operation with its release flag set to FALSE, then calling CORBA_free on the returned any will not deallocate the memory pointed to by _value. Before calling CORBA_free on the _value member of an any directly, then the programmer should check the release flag using CORBA_any_get_release. If it returns FALSE, then the programmer should not invoke CORBA_free on the _value member; doing so produces undefined behaviour. Also, passing a null pointer to either CORBA_any_set_release or CORBA_any_get_release produces undefined behaviour. If CORBA_any_set_release is never called for a given instance of any, the default value of the release flag for that instance is FALSE.

3.1.1.8 Object Interface

Header: CORBA/Object.h

CORBA_Object_duplicate

```
CORBA_Object CORBA_Object_duplicate
(
   CORBA_Object obj,
   CORBA_Environment * ev
)
```

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference. If more than one copy of an object reference is needed, the client may create a duplicate. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request.

When the object is no longer needed, you must call CORBA_Object_release to release it.

CORBA_Object_get_client_policy

```
CORBA_Policy CORBA_Object_get_client_policy
(
   CORBA_Object obj,
   CORBA_PolicyType policy_type,
   CORBA_Environment * ev
)
```

Returns the effective overriding Policy for the object reference. The effective override is obtained by first checking for an override of the given PolicyType at the Object scope, then at the Current scope, and finally at the ORB scope. If no override is present for the requested PolicyType, a system-dependent default value for that PolicyType may be returned. A NULL Policy reference may also be returned to indicate that there is no default for the policy. Portable applications are expected to set the desired “defaults” at the ORB scope since default Policy values are not specified.

CORBA_Object_get_policy

```
CORBA_Policy CORBA_Object_get_policy
```

```
(
   CORBA_Object obj,
   CORBA_PolicyType policy_type,
   CORBA_Environment *ev
)
```

INV_POLICY is raised with standard minor code 1. The absence of a Policy value in the IOR implies that any legal value may be used. Invoking non_existent on an object reference prior to get_policy ensures the accuracy of the returned effective Policy. If get_policy is invoked prior to the object reference being bound, a compliant implementation shall attempt a binding and then return the effective Policy. If the binding attempt fails it shall pass through the system exception returned from the binding attempt. Note that the effective Policy may change from invocation to invocation due to transparent rebinding.

Parameters

policy_type The type of policy to be obtained.

CORBA_Object_get_policy_overrides

```
CORBA_PolicyList * CORBA_Object_get_policy_overrides
(
   CORBA_Object obj,
   CORBA_PolicyTypeSeq * ts,
   CORBA_Environment * ev
)
```

Returns the list of Policy overrides (of the specified policy types) set at the Object scope. If the specified sequence is empty, all Policy overrides at this scope will be returned. If none of the requested PolicyTypes are overridden at the Object scope, an empty sequence is returned.

CORBA_Object_is_equivalent

```
CORBA_boolean CORBA_Object_is_equivalent
(
   CORBA_Object obj,
   CORBA_Object other_object,
   CORBA_Environment * ev
)
```

The is_equivalent operation is used to determine if two object references are equivalent, in so far that the ORB can easily determine if the object reference passed as the *obj* parameter is equivalent to reference passed as the *other_object* parameter. then TRUE is returned FALSE is returned if the are not equivalent.

If two object references are identical, they are equivalent. Two different object references which refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt to determine if two distinct object references refer to the same object. In general, the existence of reference translation and encapsulation, in the absence of an omniscient topology service, can make such determination impractically expensive. In other words, a FALSE return value may only indicating that the object *references* are distinct, but not necessarily that the *objects* are distinct.

CORBA_Object_is_a

```
CORBA_boolean CORBA_Object_is_a
(
```

```
   CORBA_Object obj,
   char * id,
   CORBA_Environment * ev
)
```

Facilitate maintaining type-safety for object references over the scope of an ORB.

The *id* is a string denoting a shared type identifier. The operation returns true if the object is really an instance of that type, including if that type is an ancestor of the most derived type of that object. Determining whether an object's type is compatible with the *id* may require contacting a remote ORB or interface repository. Such an attempt may fail at either the local or the remote end. If `is_a` cannot make a reliable determination of type compatibility due to failure, then it raises an exception in the calling application code. This enables the application to distinguish among the `TRUE`, `FALSE`, and indeterminate cases.

This operation exposes to application programmers functionality that must already exist in ORBs which support type safe narrow and allows programmers working in environments that do not have compile time type checking to explicitly maintain type safety.

Note that this operation is not supported under strict minimum CORBA or the CORBA/e compact profile.

CORBA_Object_is_nil

```
CORBA_boolean CORBA_Object_is_nil
(
   CORBA_Object obj,
   CORBA_Environment * ev
)
```

Returns `TRUE` if the object reference does not refer to a servant.

CORBA_Object_is_non_existent

```
CORBA_boolean CORBA_Object_non_existent
(
   CORBA_Object obj,
   CORBA_Environment * ev
)
```

Returns `TRUE` if the object implementation no longer exists. Note that this operation is not supported under strict minimum CORBA or the CORBA/e compact profile.

CORBA_Object_get_orb

```
CORBA_ORB CORBA_Object_non_existent
(
   CORBA_Object obj,
   CORBA_Environment * ev
)
```

Returns the ORB associated with an object. This will always be the ORB singleton.

CORBA_Object_release

```
void CORBA_Object_release
(
   CORBA_Object obj,
   CORBA_Environment * ev
```

```
)
```

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference.

When an object reference is no longer needed by a program, its storage may be reclaimed by use of the *release* operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the `release` operation.

CORBA_Object_set_policy_overrides

```
CORBA_Object CORBA_Object_set_policy_overrides
(
   CORBA_Object obj,
   CORBA_PolicyList * policies,
   CORBA_SetOverrideType set_add,
   CORBA_Environment * ev
)
```

The *set_policy_overrides* operation returns a new object reference with the new policies associated with it. It takes two input parameters. The *policies* parameter is a sequence of references to `Policy` objects. The parameter *set_add* parameter indicates whether these policies should be added onto any other overrides that already exist (*ADD_OVERRIDE*) in the object reference, or they should be added to a clean override free object reference (*SET_OVERRIDE*). This operation associates the policies passed in the *policies* parameter with a newly created object reference that it returns. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempts to override any other policy will result in the raising of the *CORBA_NO_PERMISSION* exception.

Parameters

policies A sequence of `Policy` objects that are to be associated with the new copy of the object reference returned by this operation. If the sequence contains two or more `Policy` objects with the same *PolicyType* value, the operation raises the standard system exception *BAD_PARAM* with minor code 30.

set_add Whether the association is in addition to (*ADD_OVERRIDE*) or as a replacement of (*SET_OVERRIDE*) any existing overrides already associated with the object reference. If the value of this parameter is `SET_OVERRIDE`, the supplied policies completely replace all existing overrides associated with the object reference. If the value of this parameter is `ADD_OVERRIDE`, the supplied policies are added to the existing overrides associated with the object reference, except that if a supplied Policy object has the same `PolicyType` value as an existing override, the supplied Policy object replaces the existing override.

CORBA_Object_validate_connection

```
CORBA_boolean CORBA_Object_validate_connection
(
   CORBA_Object obj,
   CORBA_PolicyList ** inconsistent_policies,
   CORBA_Environment * ev
)
```

Function to validate connection with respect to effective policies. Returns the value `TRUE` if the current effective policies for the `Object` will allow an invocation to be made. If the object reference is not yet bound, a binding will occur as part of this operation. If the object reference is already bound, but current policy overrides have changed or for any other reason the binding is no longer valid, a rebind will be attempted regardless of the setting of any `RebindPolicy` override.

If the current effective policies are incompatible, the out parameter `inconsistent_policies` contains those policies causing the incompatibility (not exhaustive). Returns the value `FALSE` if the current effective policies would cause an invocation to raise a system exception.

Parameters

`inconsistent_policies` policy list

3.1.1.9 ORB Interface

Header: `CORBA/ORB.h`

Types

```
typedef char * CORBA_ORB_ObjectId
typedef CORBA_Object CORBA_ORB
typedef CORBA_sequence_string CORBA_ORB_ObjectIdList
```

CORBA_ORB_create_policy

```
CORBA_Policy CORBA_ORB_create_policy
(
   CORBA_ORB orb,
   CORBA_PolicyType type,
   CORBA_any * any,
   CORBA_Environment * ev
)
```

The ORB operation `create_policy` can be invoked to create new instances of policy objects of a specific type with a specified internal state. If `create_policy` fails to instantiate a new Policy object due to its inability to interpret the requested type and content of the policy, then it raises one of the `PolicyError` exceptions listed under *Policy Error Codes* on page 57.

Parameters

`type` The type of policy object to be created

`any` Value for setting the initial state of the policy object that is created.

CORBA_ORB_destroy

```
void CORBA_ORB_destroy (CORBA_ORB orb, CORBA_Environment * ev)
```

This operation destroys the ORB and enables resources to be reclaimed. The `destroy` function destroys the memory associated with the `CORBA_Environment`. This function should only be called once in an application when the ORB is no longer required since Spectra ORB does not currently support multiple ORB instances.

CORBA_ORB_shutdown

```
void CORBA_ORB_shutdown
(
   CORBA_ORB orb,
```

```
   CORBA_boolean wait_for_completion,
   CORBA_Environment * ev
)
```

This operation shuts down the ORB, deactivating the POA if installed. If the `wait_for_completion` parameter is set to `TRUE` then this function will block until all pending requests on any installed POAs are completed.

CORBA_ORB_init

```
CORBA_ORB CORBA_ORB_init
(
   int * argc,
   char ** argv,
   CORBA_ORBid orb_identifier,
   CORBA_Environment * ev
)
```

Create and initialize a new ORB instance. This function may be called whenever a reference to the ORB is required. The implementation of this function only supports a single ORB instance, since Spectra ORB is a minimum- CORBA ORB, so the *`orb_identifier`* argument is ignored. The first call to this function will create and initialize the singleton ORB instance, all subsequent calls will simply return a reference to this singleton ORB.

The C mapping for `ORB_init` deviates from the OMG IDL PIDL in the way it handles the *`arg_list`* parameter. This is intended to provide a meaningful PIDL definition of the initialization API, which has a natural C binding. To this end, the `arg_list` structure is replaced with `argv` and `argc` parameters.

The `argv` parameter is defined as an array of strings (`char**`) and the number of strings in the array is passed in the `argc (int*)` parameter.

If an empty `ORBid` string is used then `argc` arguments can be used to determine which ORB should be returned. This is achieved by searching the `argv` parameters for one tagged `ORBid`, for example, `-ORBid "`*`ORBid_example`*`"`. If an empty `ORBid` string is used and no ORB is indicated by the `argv` parameters, the default ORB is returned.

Regardless of whether an empty or non-empty `ORBid` string is passed to `ORB_init`, the `argv` arguments are examined to determine if any ORB parameters are given. If a non-empty `ORBid` string is passed to `ORB_init`, all *`-ORBid`* parameters in the `argv` are ignored. All other *`-ORB<suffix>`* parameters may be of significance during the ORB initialization process.

For C, the order of consumption of `argv` parameters may be significant to an application. To ensure that applications are not required to handle `argv` parameters they do not recognize, the ORB initialization function must be called before the remainder of the parameters are consumed. Therefore, after the `ORB_init` call, the `argv` and `argc` parameters will have been modified to remove the ORB understood arguments. It is important to note that the `ORB_init` call can only reorder or remove references to parameters from the `argv` list; this restriction is made to avoid potential memory management problems caused by trying to free parts of the `argv` list or extending the `argv` list of parameters.

ORB Initialization Arguments

The following arguments can be passed as command-line parameters into the `CORBA_ORB_init` function in order to initialize the ORB.

`-ORBInitRef <name>=<ior>`

Specifies an object reference that can be resolved by the ORB *resolve_initial_references* function. The `<name>` argument is the name by which the reference will be resolved and the `<ior>` argument is an object reference URL.

`-ORBDefaultInitRef <name>`

Specifies a default name prefix that is applied to a name resolved by the ORB *resolve_initial_references* function.

`-ORBid <name>`

Set the ORB name. The '`orb_identifier`' argument will override this value if set.

`-ORBFT`

Enable client side fault tolerant semantics. If this flag is set and a client connection fails with a TRANSIENT, COMM_FAILURE, NO_RESPONSE or OBJ_ADAPTER system exception then the connection is re-bound and an attempt made to re-invoke the call unless a COMPLETED_MAYBE completion status was returned (to ensure at most once semantics).

`-ORBServerId <name>`

Specifies an ORB server identity that is added to object keys created for persistent object references. This allows the ORB to distinguish between object references created in different processes with the same POA name and object id.

`-ORBSyncServer`

Sets the default messaging sync scope for one way calls, to sync with server (only return when server has received request).

`-ORBSyncTarget`

Sets the default messaging sync scope for one way calls, to sync with target (only return when servant has received request).

`-ORBLogLevel <debug | stats | warn | error | fatal>`

Sets the default ORB logging level. If no log level is specified the default level is for warnings and above (logging is only available in debug builds).

`-ORBConfigFile <file>`

Specifies a file from which other ORB initialization arguments can be read. Each initialization argument should be on a separate line of the file and a single character should be used to separate arguments and their values. For example:

```
-ORBLogLevel:DEBUG
-ORBServerId:Steve
-ORBDefaultInitRef:com.prismt
```

`-ORBNoColocation`

Specifies that requests to co-located objects are made via a transport and not via the normal, more optimal, direct request.

`-ORBHost`

Specifies the host name or IP address to be used for IIOP/DIOP object profiles. Useful for multi-homed hosts where multiple network interfaces are supported with different addresses.

`-ORBInterop`

Specifies that the ORB will be interoperating with a foreign ORB. This disables the use of any ORB specific GIOP components, such as service contexts.

`-ORBProtocol <protocol>`

Specifies the default protocol to use when an IOR contains multiple profiles. The <*protocol*> string is the name of the protocol to select, for example '*uiop*' or '*iiop*'. If not set, then the ORB by default will always select the IIOP protocol.

`-ORBListenEndpoints <list>`

Specifies the default set of protocol listen endpoints. These are specified as a comma separated list of endpoint identifiers. For example:

```
iiop://<host>:<port>,diop:<host>:<port>,uiop:<file>
```

`-ORBPOAEndpoints <list>`

Specifies transport listener endpoints on a per POA basis. These are specified as a comma separated list of endpoint identifiers. The syntax is the same as for standard endpoints (see above), except that the POA name is applied as a prefix with a colon separator. For example:

```
NameService:iiop:<host>:<port>
```

`-ORBThreadPoolStack <value>`

Sets the default stack size, in bytes, for threads created by the ORB. By default the stack size for created threads is determined by the operating system.

`-ORBThreadPoolPriority <value>`

Sets the default native priority for threads created by the ORB. By default threads are created without a set priority.

`-ORBThreadPoolSize <value>`

Sets the initial size of a thread pool (number of threads created on initialization). By default this is set to one so the thread pool is initialized with a single thread.

`-ORBThreadPoolMax <value>`

Sets the maximum size of a thread pool. By default this is zero so there is no maximum for the number of threads that may be created in a pool.

`-ORBThreadPoolQueue <value>`

Sets the maximum number of requests that can be queued for processing by a thread pool. By default this is zero so there is no maximum for the number of jobs that may be queued for processing by the thread pool.

`-ORBConnCacheSize <value>`

Sets the maximum size of the client transport connection cache. Connection caching is disabled if this is set to zero or the real-time ORB is installed. The default maximum for the connection cache is four.

`-ORBPrivateServerConn`

Sets the default server connection model to private.

`-ORBPrivateClientConn`

Sets the default client connection model to private. By default, client connections are shared unless the realtime ORB has been installed.

`-ORBListenerPriority <priority>`

When running in realtime mode sets the default priority for transport listener threads. The priority value should be a valid CORBA priority (0 – 32767).

`-ORBRequestTimeout <timeout>`

Set the timeout (in milliseconds), used when the ORB tries makes a request. By default this is not set and the ORB will wait for ever for a request to complete.

`-ORBConnectTimeout <timeout>`

Set the timeout (in milliseconds), used when the ORB tries to establish a new client side transport connection. By default this is not set and the ORB will wait for ever for a connection to be established.

`-ORBFragmentSize <size>`

This enables GIOP fragmentation for messages above the specified size. The fragment size includes the GIOP header so the minimum for this value is 24. This will be the maximum request size sent to the underlying transport so can be used where a transport implementation may have a fixed maximum request size.

`-ORBRegistry <registries>`

This argument configures registries used for the ORB register_initial_reference operation. Supported registries currently include standard output and file. See the 'Object Reference Resolution' section of the user guide for more detail.

`-ORBLocate`

When an object reference is first used this flag makes the ORB make an initial GIOP locate request as opposed to a normal request. This can be useful when working with an implementation repository or other location agent, where the endpoint in the object reference in not for the eventual target server. This behaviour can also be implemented on a per interface basis by compiling with the '`-locate`' IDL compiler flag.

`-ORBCorbaloc`

When converting an object reference to a string using the `object_to_string` operation this flag causes the generation of a `corbaloc` format string.

`-ORBReuseAddr`

When creating a socket based endpoint (for example for IIOP or DIOP protocols), this flag sets the socket reuse address policy. Is normally of use when running a server on fixed endpoints, in the situation where a server is restarting and the listener socket may not have timed out.

`-ORBKeepAlive`

When using a socket based transport such as IIOP this flag sets the socket keep alive policy on connections. Setting this option allows connections to detect when a remote connection has failed and generate an exception for clients and close the connection for servers.

`-ORBCloseClientConn`

When a client side transport connection is closed and this flag is set the ORB will send a `GIOP::CloseConnection` message.

`-ORBCloseServerConn`

When a server side transport connection is closed and this flag is set the ORB will send a `GIOP::CloseConnection` message.

`-ORBFixedPOA`

If this flag is set then all servant lookups in a POA are unlocked, giving faster request handling. This should not be enabled if servants are dynamically added or removed from a POA after it has been activated.

`-ORBShutdownDelay <milliseconds>`

When the ORB is destroyed, this sets the delay before the ORB run operation returns. This can be useful when memory profiling the ORB to allow time for pool threads to exit, to avoid the generation of spurious memory leak reports. The default value for this delay can vary depending on the operating system, although for most systems the default value is 400 milliseconds. To check for the value for a particular system look for `EORB_ORB_SHUTDOWN_DELAY` in the system specific include directory under `include/eOrbC/os`.

`-ORBClientCDRBufferSize <bytes>`

This value sets the initial size for the CDR marshalling buffer for the construction of client requests. The buffer will be reused to receive a reply message. The default value is 512 and the minimum value is 256. Note that the maximum size of CDR buffers is shown when ORB logging is set to report stats.

`-ORBServerCDRBufferSize <bytes>`

This value sets the initial size for the CDR marshalling buffer for a server to receive requests. The buffer will be reused to construct reply messages. The default value is 512 and the minimum value is 256. Note that the maximum size of CDR buffers is shown when ORB logging is set to report stats.

`-ORBCDRBufferResize <number>`

When a CDR marshalling buffer needs to be increased in size the new size will be rounded up to a power of 2. This value sets the required power of 2. The default value is 12 (allocating in 4K byte chunks) and the maximum value is 24.

`-ORBZeroCopyThreshold <bytes>`

If a transport supports zero copy then this value sets the threshold size below which data is copied as opposed to zero copied. For small data sizes, zero copying may be less performant than simply copying.

`-ORBZeroCopyBuffMax <number>`

If a transport supports zero copy then this value sets the maximum number of data buffers that will be presented by the ORB to the transport. Some transports may have a limit on the number of zero copy buffers they can handle.

`-ORBPOAPersistent`

Sets the default lifespan policy for the root POA to be persistent.

`-ORBPOAMultipleId`

Sets the default id uniqueness policy for the root POA to be multiple id.

`-ORBPOAUserId`

Sets the default id assignment policy for the root POA to be user id.

CORBA_ORB_list_initial_services

```
CORBA_ORB_ObjectIdList * CORBA_ORB_list_initial_services
(
   CORBA_ORB orb,
   CORBA_Environment * ev
)
```

This function returns an `ObjectIdList` of the object IDs of those objects which are listed in the *initial references* list. `ObjectIdList` is a sequence of object identifiers and are typed as strings. Objects whose IDs are in this list can be resolved using the `resolve_initial_references` function.

See *CORBA_ORB_resolve_initial_references below*.

CORBA_ORB_object_to_string

```
CORBA_char * CORBA_ORB_object_to_string
(
   CORBA_ORB orb,
   const CORBA_Object orb,
   CORBA_Environment * ev
)
```

Returns a string representation (a *stringified IOR*) of a CORBA object's IOR. This string can be persisted and is one of the methods which are available for publishing or making an object available to other objects. Also see `CORBA_ORB_string_to_object`. The generated IOR string will be of generic format (starting with "`IOR:`") unless the orb argument is passed as `NULL`, in which case a `corbaloc` format string (starting with "`corbaloc:`") will be generated. This latter case is Spectra ORB-specific behaviour.

CORBA_ORB_register_initial_reference

```
void CORBA_ORB_register_initial_reference
(
   CORBA_ORB orb,
   const char * name,
   CORBA_Object obj,
   CORBA_Environment * ev
)
```

Adds a unique named object to the list used by resolve_initial_references for resolving named objects. See `CORBA_ORB_resolve_initial_references` below.

If a reference for a local servant is registered as an initial reference, then this name can be used as part of a `corbaloc` format object reference string to resolve the service. For example the name service can be resolved via the following type URI:

`corbaloc:iiop:<host>:<port>/NameService`

The ORB also supports pluggable registries via the '`-ORBRegistry`' ORB initialization argument. This allows a registered reference to be written to a number of formats. Currently supported registries include file and standard output. See the 'Object Reference Resolution' section of the user guide for more detail.

If an attempt is made to register a reference using a name that has already been registered then an `InvalidName` exception is thrown unless the service name ends in "Service" in which case the old registration will be overwritten with the new.

CORBA_ORB_resolve_initial_references

```
CORBA_Object CORBA_ORB_resolve_initial_references
(
   CORBA_ORB orb,
   const char * identifier,
   CORBA_Environment * ev
)
```

Returns the object reference for objects or service listed in an initial references list. This list uses object IDs (strings) to identify the objects. The ids listed is only for a subset of standard CORBA core objects, services and other, non-standard objects. The objects listed in the initial references list depend on the ORB's configuration. Each standard CORBA object and service has a reserved ID.

The reserved IDs (type `ObjectId`) for the CORBA core objects are:

```
RootPOA
POACurrent
InterfaceRepository
PolicyManager
PolicyCurrent
```

The reserved IDs (type `ObjectId`) for the CORBA services are:

```
NameService
EventService
```

The default ORB configuration supports only the `RootPOA`, `POACurrent`, `PolicyManager` and `PolicyCurrent` objects. Also see *CORBA_ORB_register_initial_reference* .

This function will also resolve objects registered via an `-ORBInitRef` ORB initialisation argument or explicitly via the `ORB register_initial_reference` function.

CORBA_ORB_run

```
void CORBA_ORB_run (CORBA_ORB orb, CORBA_Environment * ev)
```

Instructs the ORB to process ORB events (requests and replies). Does not return until the ORB is shutdown. Calling `run` from the main thread is useful to ensure that the process does not exit until the ORB is shut down.

CORBA_ORB_string_to_object

```
CORBA_Object CORBA_ORB_string_to_object
(
   CORBA_ORB orb,
   const CORBA_char * str,
   CORBA_Environment * ev
)
```

Uses a string containing a *stringified IOR*, the *`str`* parameter, to return the IOR for the object specified by the *`orb`* parameter. Throws `BAD_PARAM` exception on failure.

The *`orb`* parameter can be `NULL`, this is necessary when a fully initialized ORB is not available when using `PortableInterceptor_ORBInitInfo_register_initial_reference` in a *`pre_init`* initializer of a `PortableInterceptor_ORBInitializer`.

3.1.1.10 Policy Interface

CORBA_Policy__get_policy_type

```
CORBA_PolicyType CORBA_Policy__get_policy_type
(
   CORBA_Policy obj,
   CORBA_Environment * ev
)
```

Returns the constant value of type `PolicyType` that corresponds to the type of the policy object `obj`.

CORBA_Policy_copy

```
CORBA_Policy CORBA_Policy_copy (CORBA_Policy obj, CORBA_Environment * ev)
```

Creates a copy of the policy object `obj`. The copy does not retain any relationships that the policy had with any domain or object.

CORBA_Policy_destroy

```
void CORBA_Policy_destroy (CORBA_Policy obj, CORBA_Environment *ev)
```

Destroys the policy object `obj`. It is the responsibility of the policy object to determine whether it can be destroyed.

3.1.1.11 PolicyCurrent Interface

Header: `CORBA/Policy.h`

CORBA_PolicyCurrent_get_policy_overrides

```
CORBA_PolicyList * CORBA_PolicyCurrent_get_policy_overrides
(
   CORBA_PolicyCurrent obj,
   CORBA_PolicyTypeSeq * ts,
   CORBA_Environment * ev
)
```

Returns policies with thread-level scope.

See `CORBA_PolicyManager_get_policy_overrides` for details.

CORBA_PolicyCurrent_set_policy_overrides

```
void CORBA_PolicyCurrent_set_policy_overrides
(
   CORBA_PolicyCurrent obj,
   CORBA_PolicyList * policies,
   CORBA_SetOverrideType set_add,
   CORBA_Environment *ev
)
```

Modifies policies with Thread-level scope.

See `CORBA_PolicyManager_set_policy_overrides` for details.

3.1.1.12 PolicyManager Interface

Header: `CORBA/Policy.h`

CORBA_PolicyManager_get_policy_overrides

```
CORBA_PolicyList * CORBA_PolicyManager_get_policy_overrides
(
   CORBA_PolicyManager obj,
   CORBA_PolicyTypeSeq * ts,
   CORBA_Environment * ev
)
```

Returns a `PolicyList` containing the overridden `Policies` for the requested `PolicyTypes`. If the specified sequence is empty, then all `Policy` overrides at this scope will be returned. If none of the requested `PolicyTypes` are overridden at the target `PolicyManager`, then an empty sequence is returned. This accessor returns only those `Policy` overrides that have been set at the specific scope corresponding to the target `PolicyManager` (no evaluation is done with respect to overrides at other scopes).

Parameters

ts *A* sequence of overridden policy types identifying the policies that are to be retrieved.

CORBA_PolicyManager_set_policy_overrides

```
void CORBA_PolicyManager_set_policy_overrides
(
```

```
   CORBA_PolicyManager obj,
   CORBA_PolicyList * policies,
   CORBA_SetOverrideType set_add,
   CORBA_Environment * ev
)
```

Modifies the current set of overrides with the requested list of `Policy` overrides.

The *`policies`* parameter is a sequence of references to `Policy` objects. The *`set_add`* parameter indicates whether these policies should be added onto any other overrides that already exist (*`ADD_OVERRIDE`*) in the `PolicyManager`, or they should be added to a clean `PolicyManager` free of any other overrides (*`SET_OVERRIDE`*). Invoking `set_policy_overrides` with an empty sequence of policies and a mode of `SET_OVERRIDE` removes all overrides from a `PolicyManager`. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempts to override any other policy will result in the raising of the *`CORBA_NO_PERMISSION`* exception. If the request would put the set of overriding policies for the target `PolicyManager` in an inconsistent state, no policies are changed or added, and the *`InvalidPolicies`* exception is raised. There is no evaluation of compatibility with policies set within other `PolicyManagers`.

Parameters

`policies` A sequence of `Policy` objects that are to be associated with the `PolicyManager` object. If the sequence contains two or more `Policy` objects with the same `PolicyType` value, the operation raises the standard system exception `BAD_PARAM` with standard minor code 30.

`set_add` Whether the association is in addition to (`ADD_OVERRIDE`) or as a replacement of (`SET_OVERRIDE`) any existing overrides already associated with the `PolicyManager` object. If the value of this parameter is `SET_OVERRIDE`, the supplied policies completely replace all existing overrides associated with the `PolicyManager` object. If the value of this parameter is `ADD_OVERRIDE`, the supplied policies are added to the existing overrides associated with the `PolicyManager` object, except that if a supplied `Policy` object has the same `PolicyType` value as an existing override, the supplied `Policy` object replaces the existing override.

3.1.1.13 TypeCode Interface

Header: `CORBA/TypeCode.h`

CORBA_TypeCode_BadKind__alloc

```
CORBA_TypeCode_BadKind * CORBA_TypeCode_BadKind__alloc (void)
```

CORBA_boolean CORBA_TypeCode_equal

```
CORBA_boolean CORBA_TypeCode_equal
(
   CORBA_TypeCode tc1,
   CORBA_TypeCode tc2,
   CORBA_Environment * ev
)
```

`CORBA_TypeCode_equal` determines if two `TypeCodes` describe the same basic abstract data type. Equivalent `TypeCodes` produce the same results when `TypeCode` methods are invoked on them.

CORBA_TypeCode_duplicate

```
void CORBA_TypeCode_duplicate (CORBA_TypeCode tc)
```

Creates a duplicate instance of `CORBA_TypeCode`.

CORBA_TypeCode_id

```
CORBA_RepositoryId CORBA_TypeCode_id
(
   CORBA_TypeCode tc,
   CORBA_Environment * ev
)
```

Returns the repository id of the type code.

CORBA_TypeCode_kind

```
CORBA_TCKind CORBA_TypeCode_kind
(
   CORBA_TypeCode tc,
   CORBA_Environment * ev
)
```

Returns the `TCKind` value of a type code. The return value describes what kind of type is described by the type code. The `TCKind` value also determines which other operations on the type code you can call to extract more details.

CORBA_TypeCode_name

```
CORBA_Identifier CORBA_TypeCode_name
(
   CORBA_TypeCode tc,
   CORBA_Environment * ev
)
```

Returns the identifier of the type code.

3.1.1.14 LocalObject Interface

Header: `CORBA/LocalObject.h`

CORBA_LocalObject__add_ref

```
CORBA_LocalObject__add_ref (CORBA_LocalObject obj)
```

Adds a reference to an allocated instance of a local object.

CORBA_LocalObject__remove_ref

```
CORBA_LocalObject__remove_ref (CORBA_LocalObject obj)
```

Removes a reference from an allocated instance of a local object and frees the instance if the reference count reaches zero.

3.1.1.15 DataInputStream Value Type

Header `CORBA/DataInputStream.h`

CORBA_DataInputStream__alloc

```
CORBA_DataInputStream CORBA_DataInputStream__alloc
   (CORBA_DataOutputStream os)
```

Creates a data output stream from a data input stream. The contents of the output stream are copied into the newly created input stream.

CORBA_DataInputStream_read_boolean

```
CORBA_boolean CORBA_DataInputStream_read_boolean
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads a boolean value from a data input stream.

CORBA_DataInputStream_read_char

```
CORBA_char CORBA_DataInputStream_read_char
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads a char value from a data input stream.

CORBA_DataInputStream_read_octet

```
CORBA_octet CORBA_DataInputStream_read_octet
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads an octet value from a data input stream.

CORBA_DataInputStream_read_short

```
CORBA_short CORBA_DataInputStream_read_short
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads a short value from a data input stream.

CORBA_DataInputStream_read_ushort

```
CORBA_unsigned_short CORBA_DataInputStream_read_ushort
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads an unsigned short value from a data input stream.

CORBA_DataInputStream_read_long

```
CORBA_long CORBA_DataInputStream_read_long
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads a long value from a data input stream.

CORBA_DataInputStream_read_ulong

```
CORBA_unsigned_long CORBA_DataInputStream_read_ulong
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads an unsigned long value from a data input stream.

CORBA_DataInputStream_read_string

```
char * CORBA_DataInputStream_read_string
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads a string value from a data input stream.

CORBA_DataInputStream_read_float

```
CORBA_float CORBA_DataInputStream_read_float
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads a float value from a data input stream.

CORBA_DataInputStream_read_double

```
CORBA_double CORBA_DataInputStream_read_double
(
   CORBA_DataInputStream is,
   CORBA_Environment * ev
)
```

Reads a double value from a data input stream.

CORBA_DataInputStream_read_octet_array

```
CORBA_double CORBA_DataInputStream_read_octet_array
(
   CORBA_DataInputStream is,
   CORBA_OctetSeq * seq,
   CORBA_unsigned_long offset,
   CORBA_unsigned_long length,
   CORBA_Environment * ev
)
```

Reads an octet array from a data input stream into an octet sequence. The length argument is the size of the octet array to copy and the offset argument is the offset within the sequence to which the array is copied. The sequence buffer is extended if required.

3.1.1.16 DataOutputStream Value Type

Header `CORBA/DataOutputStream.h`

CORBA_DataOutputStream__alloc

```
CORBA_DataOutputStream CORBA_DataOutputStream__alloc (void)
```

Creates a data output stream.

CORBA_DataOutputStream_write_boolean

```
void CORBA_DataOutputStream_write_boolean
(
   CORBA_DataOutputStream os,
   CORBA_boolean value,
   CORBA_Environment * ev
)
```

Writes a boolean value to a data output stream.

CORBA_DataOutputStream_write_char

```
void CORBA_DataOutputStream_write_char
(
   CORBA_DataOutputStream os,
   CORBA_char value,
   CORBA_Environment * ev
)
```

Writes a char value to a data output stream.

CORBA_DataOutputStream_write_octet

```
void CORBA_DataOutputStream_write_octet
(
   CORBA_DataOutputStream os,
   CORBA_octet value,
   CORBA_Environment * ev
)
```

Writes an octet value to a data output stream.

CORBA_DataOutputStream_write_short

```
void CORBA_DataOutputStream_write_short
(
   CORBA_DataOutputStream os,
   CORBA_short value,
   CORBA_Environment * ev
)
```

Writes a short value to a data output stream.

CORBA_DataOutputStream_write_ushort

```
void CORBA_DataOutputStream_write_ushort
(
   CORBA_DataOutputStream os,
   CORBA_unsigned_short value,
   CORBA_Environment * ev
)
```

Writes an unsigned short value to a data output stream.

CORBA_DataOutputStream_write_long

```
void CORBA_DataOutputStream_write_long
(
   CORBA_DataOutputStream os,
   CORBA_long value,
   CORBA_Environment * ev
)
```

Writes a long value to a data output stream.

CORBA_DataOutputStream_write_ulong

```
void CORBA_DataOutputStream_write_ulong
(
   CORBA_DataOutputStream os,
   CORBA_unsigned_long value,
   CORBA_Environment * ev
)
```

Writes an unsigned long value to a data output stream.

CORBA_DataOutputStream_write_string

```
void CORBA_DataOutputStream_write_string
(
   CORBA_DataOutputStream os,
   char * value,
   CORBA_Environment * ev
)
```

Writes a string value to a data output stream.

CORBA_DataOutputStream_write_float

```
void CORBA_DataOutputStream_write_float
(
   CORBA_DataOutputStream os,
   CORBA_float value,
   CORBA_Environment * ev
)
```

Writes a float value to a data output stream.

CORBA_DataOutputStream_write_double

```
void CORBA_DataOutputStream_write_double
(
   CORBA_DataOutputStream os,
```

```
   CORBA_double value,
   CORBA_Environment * ev
)
```

Writes a double value to a data output stream.

CORBA_DataOutputStream_write_octet_array

```
void CORBA_DataOutputStream_write_octet_array
(
   CORBA_DataOutputStream os,
   CORBA_OctetSeq * seq,
   CORBA_unsigned_long offset,
   CORBA_unsigned_long length,
   CORBA_Environment * ev
)
```

Writes an octet array contained in a sequence to a data output stream. The length argument determines the length of array to be written and the offset argument the array start offset within the sequence.

3.1.2 PortableInterceptor Module

Header: `PortableInterceptor.h`

PortableInterceptor_register_orb_initializer

```
PortableInterceptor_register_orb_initializer
(
   PortableInterceptor_ORBInitializer * init,
   CORBA_Environment * ev
)
```

This function registers a portable ORB initializer. Initializers should be registered before the ORB is initialized as the ORB initialization calls back on registered initializers.

3.1.2.1 ORBInitializer Local Interface

Header: `PortableInterceptor/ORBInitializer.h`

```
typedef struct PortableInterceptor_ORBInitializer
{
   void (*pre_init)
   (
      PortableInterceptor_ORBInitInfo info,
      CORBA_Environment * ev
   );

   void (*post_init)
   (
      PortableInterceptor_ORBInitInfo info,
      CORBA_Environment * ev
   );

   void (*fini)
   (
```

```
      PortableInterceptor_ORBInitInfo info,
      CORBA_Environment * ev
   );
}
PortableInterceptor_ORBInitializer;
```

The `ORBInitializer` local interface is implemented as a `struct` containing function pointers to the defined ORB initialization functions (*`pre_init`* and *`post_init`*). An additional Spectra ORB-specific function is also provided (*`fini`*) to support initializer clean up on ORB shutdown.

PortableInterceptor_ORBInitializer__alloc

```
PortableInterceptor_ORBInitializer *
   PortableInterceptor_ORBInitializer__alloc (void)
```

This function allocates a new instance of an `ORBInitializer` local interface. This can be registered as an ORB initializer.

3.1.2.2 ORBInitInfo Local Interface

Header: `PortableInterceptor/ORBInitInfo.h`

```
typedef CORBA_LocalObject  PortableInterceptor_ORBInitInfo;
typedef CORBA_char *  PortableInterceptor_ORBInitInfo_ObjectId;
```

PortableInterceptor_ORBInitInfo__get_arguments

```
CORBA_sequence_string * PortableInterceptor_ORBInitInfo__get_arguments
(
   PortableInterceptor_ORBInitInfo info,
   CORBA_Environment * ev
)
```

This function returns the list of ORB initialization arguments from an `ORBInitInfo` local interface.

PortableInterceptor_ORBInitInfo__get_orb_id

```
CORBA_char * PortableInterceptor_ORBInitInfo__get_orb_id
(
   PortableInterceptor_ORBInitInfo info,
   CORBA_Environment * ev
)
```

This function returns the ORB id of the ORB being initialized.

PortableInterceptor_ORBInitInfo_register_initial_reference

```
void PortableInterceptor_ORBInitInfo_register_initial_reference
(
   PortableInterceptor_ORBInitInfo info,
   PortableInterceptor_ORBInitInfo_ObjectId id.
   CORBA_Object obj,
   CORBA_Environment * ev
)
```

This function registers an initial named (*id*) reference (*obj*) with the ORB being initialized.

Throws BAD_PARAM exception if the obj parameter is NULL or CORBA_OBJECT_NIL.

PortableInterceptor_ORBInitInfo_resolve_initial_references

```
CORBA_Object PortableInterceptor_ORBInitInfo_resolve_initial_references
(
   PortableInterceptor_ORBInitInfo info,
   PortableInterceptor_ORBInitInfo_ObjectId id.
   CORBA_Environment * ev
)
```

This function resolves an initial named (*id*) reference with the ORB being initialized. The InvalidName exception is raised if the reference cannot be resolved.

3.1.3 EORB Module

EORB_alloc

```
void * EORB_alloc (size_t len)
```

This allocates memory of size len.

EORB_allocBuffer

```
void * EORB_allocBuffer (EORB_DtorFN dtor, size_t size, size_t count)
```

This allocates memory for a sequence of count elements of length size with a destructor function, dtor, that is invoked for each element when the memory is released with CORBA_free.

EORB_allocVar

```
void * EORB_allocVar (EORB_DtorFN dtor, size_t len)
```

This allocates memory with a destructor function, dtor, that is invoked when the memory is released with CORBA_free.

EORB_add_ref

```
void EORB_add_ref (void * ptr)
```

Adds a reference to an ORB allocated object.

EORB_remove_ref

```
void EORB_remove_ref (void * ptr)
```

Removes a reference from an ORB allocated object and frees it if the reference count reaches zero.

EORB_get_ref_count

```
CORBA_unsigned_long EORB_get_ref_count (void * ptr)
```

Returns the reference count of an ORB allocated object.

EORB_Object_dump

```
void EORB_Object_dump (CORBA_Object obj)
```

The `EORB_Object_dump` outputs or *dumps* information about the object passed to the function as *`obj`* parameter. This function can be useful for debugging applications.

EORB_POA_dump

```
void EORB_POA_dump (PortableServer_POA poa)
```

The `EORB_POA_dump` outputs or *dumps* information about the object in the POA passed to the function as the *`poa`* parameter. If a null value is passed, then information about the objects of all of the server's POAs will be output. This function can be useful for debugging the server component of applications.

EORB_Any_extract

```
void EORB_Any_extract
(
   CORBA_any * src,
   CORBA_any * dst,
   CORBA_Environment * ev
)
```

This operation extracts the value and type from the *`src`* any into the *`dst`* any. The value must have been heap allocated. The `src` any has its type set to `TC_null` and value set to `NULL`. The release flag of the `src` any must also be *`TRUE`*. An exception is thrown if the value is not heap allocated.

3.1.4 PortableServer Module

Header: `PortableServer.h`

PortableServer_ObjectId_to_string

```
CORBA_char* PortableServer_ObjectId_to_string
(
   PortableServer_ObjectId * oid,
   CORBA_Environment * ev
)
```

Converts and `ObjectId` to a string. If conversion of the `ObjectId` to the string produces illegal characters (such as a `NULL`), then the `CORBA_BAD_PARAM` exception is raised.

PortableServer_string_to_ObjectId

```
PortableServer_ObjectId* PortableServer_string_to_ObjectId
(
   CORBA_char * str,
   CORBA_Environment * ev
)
```

Converts a string to an `ObjectId`. This can be used for creating object references before activating them.

3.1.4.1 POA Interface

PortableServer_POA_activate_object

```
PortableServer_ObjectId* PortableServer_POA_activate_object
(
   PortableServer_POA poa,
   PortableServer_Servant servant,
   CORBA_Environment * ev
)
```

The `activate_object` operation activates objects. It generates an ObjectId of type `ObjectId`, enters the ID and the specified servant in the Active Object Map and returns a pointer to the object's `ObjectId`.

Parameters

servant The servant instance

PortableServer_POA_activate_object_with_id

```
void PortableServer_POA_activate_object_with_id
(
   PortableServer_POA poa,
   const PortableServer_ObjectId * oid,
   PortableServer_Servant servant,
   CORBA_Environment * ev
)
```

Enables a servant to be activated with a user-defined `ObjectId`, as specified by the `oid` parameter. The `RETAIN` policy must be present in order for `activate_object` to work: a `WrongPolicy` exception is thrown if `RETAIN` is not present. If the POA has another active object which has the same Object ID value, then the `ObjectAlreadyActive` exception is raised. If the POA has the `UNIQUE_ID` policy and the servant is already in the Active Object Map, then the `ServantAlreadyActive` exception is raised. Otherwise, the `activate_object_with_id` operation enters an association between the specified Object Id and the specified servant in the Active Object Map.

If the POA has the `SYSTEM_ID` policy and it detects that the Object ID value was not generated by the system or for this POA, then the `activate_object_with_id` operation may raise the `BAD_PARAM` system exception. A portable application must not invoke `activate_object_with_id` on a POA that has the `SYSTEM_ID` policy with an Object ID value that was not previously generated by the system for that POA, or, if the POA also has the `PERSISTENT` policy, for a previous instantiation of the same POA.

PortableServer_POA_create_id_assignment_policy

```
PortableServer_IdAssignmentPolicy
   PortableServer_POA_create_id_assignment_policy
(
   PortableServer_POA poa,
   PortableServer_IdAssignmentPolicyValue value,
   CORBA_Environment * ev
)
```

Objects with the IdAssignmentPolicy interface are obtained using the create_id_assignment_policy operation and passed to the *PortableServer_POA_create_POA* operation to specify whether Object Ids in the created POA are generated by the application or by the ORB. The following values can be supplied:

- *PortableServer_USER_ID* - Objects created with that POA are assigned ObjectIds only by the application.
- *PortableServer_SYSTEM_ID* - Objects created with that POA are assigned ObjectIds only by the POA. If the POA also has the PERSISTENT policy, assigned ObjectIds must be unique across all instantiations of the same POA.

If this policy is not supplied, it defaults to PortableServer_SYSTEM_ID.

PortableServer_POA_create_id_uniqueness_policy

```
PortableServer_IdUniquenessPolicy
   PortableServer_POA_create_id_uniqueness_policy
(
   PortableServer_POA poa,
   PortableServer_IdUniquenessPolicyValue value,
   CORBA_Environment * ev
)
```

Objects with the PortableServer_IdUniquenessPolicyValue interface are obtained using the create_id_uniqueness_policy operation and passed to the Portable_POA_create_POA operation to specify whether the servants activated in the created POA must have unique object identities. The following values can be supplied:

- *PortableServer_UNIQUE_ID* - servants activated with that POA support exactly one ObjectId.
- *PortableServer_MULTIPLE_ID* - a servant activated with that POA may support one or more ObjectIds.

If no PortableServer_IdUniquenessPolicyValue is specified at POA creation, then the default PortableServer_UNIQUE_ID value is used.

PortableServer_POA_create_lifespan_policy

```
PortableServer_LifespanPolicy
   PortableServer_POA_create_lifespan_policy
(
   PortableServer_POA poa,
   PortableServer_LifespanPolicyValue value,
   CORBA_Environment * ev
)
```

Objects with the *LifespanPolicy* interface are obtained using the PortableServer_POA_create_lifespan_policy operation and passed to the *PortableServer_POA_create_POA* operation to specify the lifespan of the objects implemented in the created POA. The following values can be supplied:

- *PortableServer_TRANSIENT* - The objects implemented in the POA cannot outlive the POA instance in which they are first created. Once the POA enters the deactivated state, any requests received by this POA will cause the POA to raise an *OBJECT_NOT_EXIST* system exception with standard minor code 4.

- *PortableServer_PERSISTENT* - The objects implemented in the POA can outlive the process in which they are first created.

- Persistent objects have a POA associated with them (the POA that created them).

  When the ORB receives a request on a persistent object, it first searches for the matching POA, based on the names of the POA and all of its ancestors.

  - Administrative action beyond the scope of this specification may be necessary to inform the ORB's location service of the creation and eventual termination of existence of this POA, and optionally to arrange for on-demand activation of a process implementing this POA.

  - POA names must be unique within their enclosing scope (the parent POA). A portable program can assume that POA names used in other processes will not conflict with its own POA names.

The default policy is PortableServer_TRANSIENT.

PortableServer_POA_create_POA

```
PortableServer_POA PortableServer_POA_create_POA
(
   PortableServer_POA poa,
   const char * name,
   PortableServer_POAManager manager,
   CORBA_PolicyList * policies,
   CORBA_Environment * ev
)
```

This operation creates a new POA as a child of the target POA. The specified name identifies the new POA with respect to other POAs with the same parent POA. If the target POA already has a child POA with the specified name, then the AdapterAlreadyExists exception is raised.

If the manager parameter is null, then a new POAManager object is created and associated with the new POA. Otherwise, the specified POAManager object is associated with the new POA. The POAManager object can be obtained using the the_POAManager attribute name.

The specified policy objects are associated with the POA and used to control its behaviour. The policy objects are effectively copied before this operation returns, so the application is free to destroy them while the POA is in use. Policies are not inherited from the parent POA.

If any of the policy objects specified are not valid for the ORB implementation, if conflicting policy objects are specified, or if any of the specified policy objects require prior administrative action that has not been performed, an InvalidPolicy exception is raised containing the index in the policies parameter value of the first offending policy object.

PortableServer_POA_create_reference

```
CORBA_Object PortableServer_POA_create_reference
(
   PortableServer_POA poa,
   const CORBA_char * repository_id,
   CORBA_Environment * ev
)
```

This operation creates an object reference that encapsulates a POA-generated Object Id value and the specified interface repository id, `repository_id`. The specified repository id, which may be a null string, will become the `type_id` of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behaviour.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the appropriate servant manager to be invoked, if one is available. The generated Object Id value may be obtained by invoking `POA_reference_to_id` with the created reference.

This operation requires the `SYSTEM_ID POA` policy; if not set, then the `WrongPolicy` exception is raised.

PortableServer_POA_create_reference_with_id

```
CORBA_Object PortableServer_POA_create_reference_with_id
(
   PortableServer_POA poa,
   const PortableServer_ObjectId * oid,
   const CORBA_char * repository_id,
   CORBA_Environment * ev
)
```

This operation creates an object reference that encapsulates the specified Object Id and interface repository Id values. The specified repository id, which may be a null string, will become the `type_id` of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behaviour.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the object to be activated if necessary, or the default servant used, depending on the applicable policies.

If the POA has the `SYSTEM_ID` policy and it detects that the Object Id value was not generated by the system or for this POA, the `create_reference_with_id` operation may raise the `BAD_PARAM` system exception. A portable application must not invoke this operation on a POA that has the `SYSTEM_ID` policy with an Object Id value that was not previously generated by the system for that POA, or, if the POA also has the `PERSISTENT` policy, for a previous instantiation of the same POA.

PortableServer_POA_deactivate_object

```
void PortableServer_POA_deactivate_object
(
   PortableServer_POA poa,
   const PortableServer_ObjectId * oid,
   CORBA_Environment * ev
)
```

The `deactivate_object` operation requires the `RETAIN` policy; if not present, the `WrongPolicy` exception is raised.

This operation causes the object with an `ObjectId` specified by the `oid` parameter to be deactivated. An object which has been deactivated continues to process requests until there are no active requests for that object. A deactivated object is removed from the Active Object Map after all running requests for that object are completed.

If a servant manager is associated with the POA, `ServantActivator_etherealize` is invoked with the `oid` and the associated servant after the object has been removed from the Active Object Map. Reactivation for the object blocks until etherealization (if necessary) is complete. This includes implicit activation (as described in etherealize) and explicit activation using `activate_object_with_id`. After an object has been removed from the Active Object Map and etherealized (if necessary), it may then be reactivated through the usual mechanisms. The operation does not wait for requests or etherealization to complete and always returns immediately after deactivating the object.

If the servant associated with the `oid` is serving multiple Object IDs, the `ServantActivator_etherealize` may be invoked multiple times with the same servant when the other objects are deactivated. It is the responsibility of the object implementation to refrain from destroying the servant while it is active with any ID.

PortableServer_POA_destroy

```
void PortableServer_POA_destroy
(
   PortableServer_POA self,
   CORBA_boolean etherealize_objects,
   CORBA_boolean wait_for_completion,
   CORBA_Environment * ev
)
```

This function releases all the resources attached to the POA and frees the POA pointer. All of the objects activated on this POA will be deactivated and released. The etherialize_objects argument is ignored as Minimum CORBA does not support servant managers which implement this functionality. If the wait_for_completion argument is TRUE then this operation will block until all requests currently executing on the POAs servants complete.

PortableServer_POA_find_POA

```
PortableServer_POA PortableServer_POA_find_POA
(
   PortableServer_POA poa,
   const char * name,
   CORBA_boolean activate_it,
   CORBA_Environment * ev
)
```

Returns the POA that matches the given name. The `activate_it` parameter is ignored in Minimum CORBA because there are no `AdapterActivators` in Minimum CORBA. Returns a pointer to the POA. The POA throws the `AdapterNonExistent` system exception if the specified POA is not found

PortableServer_POA__get_the_name

```
CORBA_char * PortableServer_POA__get_the_name
(
   PortableServer_POA poa,
   CORBA_Environment * ev
)
```

Returns the value of the *`the_name`* attribute. This attribute identifies the POA relative to its parent. This name is assigned when the POA is created. The name of the root POA is defined to be '`RootPOA`'.

PortableServer_POA PortableServer_POA__get_the_parent

```
PortableServer_POA PortableServer_POA__get_the_parent
(
   PortableServer_POA poa,
   CORBA_Environment * ev
)
```

This operation gets the POA interface's *the_parent* read-only attribute. NULL is returned it this is called on the root POA.

PortableServer_POA PortableServer_POA__get_the_name

```
CORBA_char * PortableServer_POA__get_the_name
(
   PortableServer_POA poa,
   CORBA_Environment * ev
)
```

This operation gets the POA interface's *the_name* read-only attribute.

PortableServer_POA PortableServer_POA__get_the_POAManager

```
PortableServer_POAManager PortableServer_POA__get_the_POAManager
(
   PortableServer_POA poa,
   CORBA_Environment * ev
)
```

This operation gets the POA interface's *the_POAManager* read-only attribute.

PortableServer_POA PortableServer_POA__get_the_children

```
PortableServer_POAList * PortableServer_POA__get_the_children
(
   PortableServer_POA poa,
   CORBA_Environment * ev
)
```

This operation gets the POA interface's *the_children* read-only attribute.

PortableServer_POA_id_to_reference

```
CORBA_Object PortableServer_POA_id_to_reference
(
   PortableServer_POA self,
   const PortableServer_ObjectId * oid,
   CORBA_Environment * ev
)
```

This operation requires the RETAIN policy; if not present, the WrongPolicy exception is raised. If an object with the specified Object ID value is currently active, a reference encapsulating the information used to activate the object is returned. If the Object ID value is not active in the POA, the an ObjectNotActive exception is raised.

PortableServer_POA_id_to_servant

```
PortableServer_Servant PortableServer_POA_id_to_servant
(
```

```
   PortableServer_POA self,
   const PortableServer_ObjectId * oid,
   CORBA_Environment * ev
)
```

This operation requires the `RETAIN` policy or the `USE_DEFAULT_SERVANT` policy. If neither policy is present, the `WrongPolicy` exception is raised. If the POA has the `RETAIN` policy and the specified `ObjectId` is in the Active Object Map, then this operation returns the servant associated with that object in the Active Object Map. Otherwise, if the POA has the `USE_DEFAULT_SERVANT` policy and a default servant has been registered with the POA, this operation returns the default servant. Otherwise the `ObjectNotActive` exception is raised.

PortableServer_POA_reference_to_id

```
PortableServer_ObjectId * PortableServer_POA_reference_to_id
(
   PortableServer_POA self,
   CORBA_Object ref,
   CORBA_Environment * ev
)
```

Returns a pointer to the object ID for the specified object reference. This operation is valid only if the reference was created by the POA on which the operation is being performed. The object denoted by the reference does not have to be active for this operation to succeed. Throws the `WrongAdapter` exception if the specified object reference was not created by this POA.

PortableServer_POA_reference_to_servant

```
PortableServer_Servant PortableServer_POA_reference_to_servant
(
   PortableServer_POA self,
   CORBA_Object ref,
   CORBA_Environment * ev
)
```

This operation requires the `RETAIN` policy or the `USE_DEFAULT_SERVANT` policy. If neither policy is present, the `WrongPolicy` exception is raised. If the POA has the `RETAIN` policy and the specified object is present in the Active Object Map, then this operation returns the servant associated with that object in the Active Object Map. Otherwise, if the POA has the `USE_DEFAULT_SERVANT` policy and a default servant has been registered with the POA, then this operation returns the default servant. Otherwise, the `ObjectNotActive` exception is raised. If the object reference was not created by this POA, then the `WrongAdapter` exception is raised.

PortableServer_POA_servant_to_id

```
PortableServer_ObjectId* PortableServer_POA_servant_to_id
(
   PortableServer_POA self,
   PortableServer_Servant servant,
   CORBA_Environment * ev
)
```

This operation requires the `USE_DEFAULT_SERVANT` policy or a combination of the `RETAIN` policy and either the `UNIQUE_ID` or `IMPLICIT_ACTIVATION` policies; if not present, the `WrongPolicy` exception is raised.

This operation has four possible behaviours:

1. If the POA has the `UNIQUE_ID` policy and the specified servant is active, then the Object Id associated with that servant is returned.
2. If the POA has the `IMPLICIT_ACTIVATION` policy and either the POA has the `MULTIPLE_ID` policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and that Object Id is returned.
3. If the POA has the `USE_DEFAULT_SERVANT` policy, the servant default servant, and the operation is being invoked in the request on the default servant, then the ObjectId associated invocation is returned.
4. Otherwise, the `ServantNotActive` exception is raised

PortableServer_POA_servant_to_reference

```
CORBA_Object PortableServer_POA_servant_to_reference
(
   PortableServer_POA self,
   PortableServer_Servant servant,
   CORBA_Environment * ev
)
```

This operation requires the `RETAIN` policy and either the `UNIQUE_ID` or `IMPLICIT_ACTIVATION` policies if invoked outside the context of an operation dispatched by this POA. If this operation is not invoked in the context of executing a request on the specified servant and the policies specified previously are not present the `WrongPolicy` exception is raised.

This operation has four possible behaviours:

1. If the POA has both the `RETAIN` and the `UNIQUE_ID` policy and the specified servant is active, an object reference encapsulating the information used to activate the servant is returned.
2. If the POA has both the `RETAIN` and the `IMPLICIT_ACTIVATION` policy and either the POA has the `MULTIPLE_ID` policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and a corresponding object reference is returned.
3. If the operation was invoked in the context of executing a request on the specified servant, the reference associated with the current invocation is returned.
4. Otherwise, the `ServantNotActive` exception is raised.

3.1.4.2 Current Interface

PortableServer_Current_get_object_id

```
PortableServer_ObjectId * PortableServer_Current_get_object_id
(
   PortableServer_Current current,
   CORBA_Environment * ev
)
```

This operation returns the *`ObjectId`* identifying the object in whose context it is called. If called outside the context of a POA-dispatched operation, a *`NoContext`* exception is raised.

PortableServer_Current_get_POA

```
PortableServer_POA PortableServer_Current_get_POA
(
   PortableServer_Current current,
   CORBA_Environment * ev
)
```

This operation returns a reference to the POA implementing the object in whose context it is called. If called outside the context of a POA-dispatched operation, then a `NoContext` exception is raised.

PortableServer_Current_get_reference

```
CORBA_Object PortableServer_Current_get_reference
(
   PortableServer_Current current,
   CORBA_Environment * ev
)
```

This operation returns an object reference to the servant in whose context it is called. If called outside the context of a POA-dispatched operation, then a `NoContext` exception is raised.

PortableServer_Current_get_servant

```
PortableServer_servant PortableServer_Current_get_servant
(
   PortableServer_Current current,
   CORBA_Environment * ev
)
```

This operation returns a reference to the servant in whose context it is called. If called outside the context of a POA-dispatched operation, then a `NoContext` exception is raised.

3.1.4.3 POAManager Interface

PortableServer_POAManager_activate

```
void PortableServer_POAManager_activate
(
   PortableServer_POAManager manager,
   CORBA_Environment * ev
)
```

This function activates all transport listeners so POAs can start handling incoming requests. The ORB *`run`* function has the same effect.

PortableServer_POAManager_get_id

```
CORBA_char * PortableServer_POAManager_get_id
(
   PortableServer_POAManager manager,
   CORBA_Environment * ev
)
```

This function returns the identity of a POAManager.

3.1.5 Messaging Module

Header: `Messaging.h`

Types

```
/* Messaging::RelativeRoundtripTimeoutPolicy */

typedef CORBA_Policy Messaging_RelativeRoundtripTimeoutPolicy;
#define Messaging_RELATIVE_RT_TIMEOUT_POLICY_TYPE 32

/* Messaging::SyncScopePolicy */

typedef CORBA_short Messaging_SyncScope;
typedef CORBA_Policy Messaging_SyncScopePolicy;
#define Messaging_SYNC_SCOPE_POLICY_TYPE 24
#define TC_Messaging_SyncScope TC_short

#define Messaging_SYNC_NONE 0
#define Messaging_SYNC_WITH_TRANSPORT 1
#define Messaging_SYNC_WITH_SERVER 2
#define Messaging_SYNC_WITH_TARGET 3
```

3.1.5.1 SyncScopePolicy Interface

Messaging_SyncScopePolicy__get_synchronization

```
Messaging_SyncScope Messaging_SyncScopePolicy__get_synchronization
(
   Messaging_SyncScopePolicy policy,
   CORBA_Environment * _ev
);
```

This returns the sync scope from a sync scope policy.

3.1.6 IOP::Codec Module

Header: `IOP.h`

Types

```
#define IOP_ENCODING_CDR_ENCAPS 0

typedef CORBA_short IOP_EncodingFormat;
typedef CORBA_LocalObject IOP_CodecFactory;

typedef struct IOP_Encoding
{
   IOP_EncodingFormat format;
   CORBA_octet major_version;
   CORBA_octet minor_version;
}
IOP_Encoding;

const char * ex_IOP_Codec_InvalidTypeForEncoding;
const char * ex_IOP_Codec_FormatMismatch;
const char * ex_IOP_Codec_TypeMismatch;
const char * ex_IOP_CodecFactory_UnknownEncoding;
```

3.1.6.1 CodecFactory Interface

IOP_CodecFactory_create_codec

```
IOP_Codec IOP_CodecFactory_create_codec
(
   IOP_CodecFactory factory,
   IOP_Encoding * enc,
   CORBA_Environment * _ev
);
```

The `CodecFactory` object can be resolved as a named initial reference "`CodecFactory`". The `create_codec` function can then be used to create a `Codec` supporting a given encoding as defined in the `enc` argument. The only supported type of encoding is `IOP_ENCODING_CDR_ENCAPS` and the major version must be 1 or the function will throw an `UnknownEncoding` exception.

3.1.6.2 Codec Interface

IOP_Codec_encode

```
CORBA_OctetSeq * IOP_Codec_encode
(
   IOP_Codec codec,
   CORBA_any * data,
   CORBA_Environment * _ev
);
```

This operation encodes the type and content of the `any` argument as a CDR encapsulation which is returned as a sequence of octets. If the type in the any cannot be encoded then the `InvalidTypeForEncoding` exception will be raised.

IOP_Codec_encode_value

```
CORBA_OctetSeq * IOP_Codec_encode_value
(
   IOP_Codec codec,
   CORBA_any * data,
   CORBA_Environment * _ev
);
```

This operation encodes the content of the `any` argument as a CDR encapsulation which is returned as a sequence of octets. If the type in the any cannot be encoded then the `InvalidTypeForEncoding` exception will be raised. Note that unlike the `encode` operation the type of the any is not encoded.

IOP_Codec_decode

```
CORBA_any * IOP_Codec_decode
(
   IOP_Codec codec,
   CORBA_OctetSeq * data,
   CORBA_Environment * _ev
);
```

This operation decodes the type and content of an `any` from a CDR encapsulation which is provided as a sequence of octets in the `data` argument. A new `any` containing the type and content of the CDR is returned from the operation. If the type in the CDR cannot be decoded then the `FormatMismatch` exception will be raised.

IOP_Codec_decode_value

```
CORBA_any * IOP_Codec_decode
(
   IOP_Codec codec,
   CORBA_OctetSeq * data,
   CORBA_TypeCode tc,
   CORBA_Environment * _ev
);
```

This operation decodes the content of an `any` from a CDR encapsulation which is provided as a sequence of octets in the `data` argument. The type of the data in the CDR must be provided as the `tc` argument A new `any` containing the type and content of the CDR is returned from the operation. If the content in the CDR cannot be decoded then the `FormatMismatch` exception will be raised.

3.2 CORBA Exceptions

Table 3 lists the standard CORBA system exceptions supported by the ORB. *Table 4* lists proprietary ORB-specific exceptions.

i The standard minor codes for the standard system exceptions are prefaced by the VMCID assigned to OMG, namely:

```
#define CORBA_OMGVMCID 0x4f4d0000
```

The minor codes for ORB-specific exceptions are prefaced by the VMCID assigned by the OMG to Spectra ORB:

```
#define EORB_VMCID 0x58500000
```

Table 3 Standard System Exceptions (OMGVMCID)

CORBA Exception	Minor Code	Value	Description
UNKNOWN	EORB_OMG_UNKNOWN_M1	1	Unlisted user exception received by client.
UNKNOWN	EORB_OMG_UNKNOWN_M2	2	Non-standard System Exception not supported.
TRANSIENT	EORB_OMG_TRANSIENT_M1	1	Request discarded by POA.
BAD_PARAM	EORB_OMG_BAD_PARAM_M7	7	String to object conversion failed due to bad scheme name.
BAD_PARAM	EORB_OMG_BAD_PARAM_M8	8	String to object conversion failed due to bad address.
BAD_PARAM	EORB_OMG_BAD_PARAM_M9	9	String to object conversion failed due to bad schema specific part.
BAD_PARAM	EORB_OMG_BAD_PARAM_M10	10	String to object conversion failed due to non specific reason.
BAD_PARAM	EORB_OMG_BAD_PARAM_M27	27	Attempt to call register an initial reference with a null Object.
DATA_CONVERSION	EORB_OMG_DATA_CONVERSION_M2	2	Failure of priority mapping object.
BAD_OPERATION	EORB_OMG_BAD_OPERATION_M2	2	Operation or attribute not known to target object.
BAD_INV_ORDER	EORB_OMG_BAD_INV_ORDER_M3	3	Operation would deadlock.
INV_POLICY	EORB_OMG_INV_POLICY_M2	2	Invalid policy identifier.
MARSHAL	EORB_OMG_MARSHAL_M4	4	Attempt to marshal local object.
NO_IMPLEMENT	EORB_OMG_NO_IMPLEMENT_M3	3	Unable to use any profile in IOR
NO_IMPLEMENT	EORB_OMG_NO_IMPLEMENT_M8	8	Operation not implemented in local object.
NO_RESOURCES	EORB_OMG_NO_RESOURCES_M2	2	No connections for request's priority.

Table 4 ORB-Specific Exceptions (EORB_VMCID)

Exception	Minor Code	Value	Description
BAD_PARAM	EORB_OMG_BAD_PARAM_M1	1	Invalid NULL argument to operation.
BAD_PARAM	EORB_OMG_BAD_PARAM_M2	2	Invalid nil object reference.
BAD_PARAM	EORB_OMG_BAD_PARAM_M3	3	Bad `TypeCode` type.
BAD_PARAM	EORB_OMG_BAD_PARAM_M4	4	Invalid object id.
BAD_PARAM	EORB_OMG_BAD_PARAM_M5	5	Invalid sequence length.
BAD_PARAM	EORB_OMG_BAD_PARAM_M6	6	Invalid policy.
BAD_PARAM	EORB_OMG_BAD_PARAM_M7	7	Invalid environment.
BAD_PARAM	EORB_OMG_BAD_PARAM_M8	8	Already Active.
BAD_PARAM	EORB_OMG_BAD_PARAM_M9	9	Invalid ID.
BAD_PARAM	EORB_OMG_BAD_PARAM_M10	10	Invalid Priority.
BAD_PARAM	EORB_OMG_BAD_PARAM_M11	11	Local call on non local on non initialized local object.
BAD_PARAM	EORB_OMG_BAD_PARAM_M12	12	Call on local object.
BAD_PARAM	EORB_OMG_BAD_PARAM_M13	13	Servant finalizer called with NULL vepv.
BAD_PARAM	EORB_OMG_BAD_PARAM_M14	14	Servant initializer called with NULL vepv.
BAD_PARAM	EORB_OMG_BAD_PARAM_M15	15	Invalid operation on POA with SYSTEM_ID policy.
BAD_PARAM	EORB_OMG_BAD_PARAM_M16	16	Attempt to extract non heap allocated value from any.
BAD_PARAM	EORB_OMG_BAD_PARAM_M17	17	Attempt to activate servant on POA with no transport
BAD_PARAM	EORB_OMG_BAD_PARAM_M18	18	Attempt to marshal uninitialized union
IMPL_LIMIT	EORB_OMG_IMPL_LIMIT_M2	2	POA Table Full.
IMPL_LIMIT	EORB_OMG_IMPL_LIMIT_M3	3	Servant Table Full.
IMPL_LIMIT	EORB_OMG_IMPL_LIMIT_M4	4	Unable to allocate memory.
INTERNAL	EORB_INTERNAL_M1	1	Failed to open file.
INTERNAL	EORB_INTERNAL_M2	2	Failed to read IOR.
INTERNAL	EORB_INTERNAL_M3	3	Bad Servant Table.
INTERNAL	EORB_INTERNAL_M4	4	Bad Reply Type.
INTERNAL	EORB_INTERNAL_M6	6	SCA Port not connected.
INTERNAL	EORB_INTERNAL_M7	7	SCA Component destroyed.
INTERNAL	EORB_INTERNAL_M8	8	SCA Component uninitialized.
INITIALIZE	EORB_INITIALIZE_M1	1	Attempt to reinitialise singleton ORB after it has been destroyed.
INITIALIZE	EORB_INITIALIZE_M2	2	Attempt to destroy singleton ORB with active references.
INITIALIZE	EORB_INITIALIZE_M3	3	Failed to obtain license.
INITIALIZE	EORB_INITIALIZE_M4	4	Bad Component Declaration.
INITIALIZE	EORB_INITIALIZE_M5	5	Bad Object Declaration.
INITIALIZE	EORB_INITIALIZE_M6	6	Bad declaration.

INITIALIZE	EORB_INITIALIZE_M7	7	No environment.
INITIALIZE	EORB_INITIALIZE_M8	8	Bad configuration.
INITIALIZE	EORB_INITIALIZE_M9	9	Bad URL Syntax.
INITIALIZE	EORB_INITIALIZE_M10	10	Name Not Found.
INITIALIZE	EORB_INITIALIZE_M11	11	ORB Not Initialized.
INITIALIZE	EORB_INITIALIZE_M12	12	Component initialization failure.
INITIALIZE	EORB_INITIALIZE_M13	13	Bad Component.
INITIALIZE	EORB_INITIALIZE_M14	14	Transport for profile not found.
INITIALIZE	EORB_INITIALIZE_M15	15	Priority not set.
INITIALIZE	EORB_INITIALIZE_M16	16	Transport connection error.
COMM_FAILURE	EORB_COMM_FAILURE_M1	1	Transport failed to receive data.
COMM_FAILURE	EORB_COMM_FAILURE_M2	2	Transport failed to connect.
COMM_FAILURE	EORB_COMM_FAILURE_M3	3	Transport failed to send.
COMM_FAILURE	EORB_COMM_FAILURE_M4	4	Received GIOP message error.
COMM_FAILURE	EORB_COMM_FAILURE_M5	5	Received unknown GIOP message type.
COMM_FAILURE	EORB_COMM_FAILURE_M9	9	Received unexpected GIOP `LocateReply` message.
COMM_FAILURE	EORB_COMM_FAILURE_M10	10	Attempted 2-Way call on 1-Way transport.
COMM_FAILURE	EORB_COMM_FAILURE_M11	11	Failed to establish transport listener.
COMM_FAILURE	EORB_COMM_FAILURE_M12	12	Unsupported GIOP addressing disposition.
COMM_FAILURE	EORB_COMM_FAILURE_M13	13	Transport failed to send data - connection closed by peer.
COMM_FAILURE	EORB_COMM_FAILURE_M14	14	Transport failed to receive data - connection closed by peer.
COMM_FAILURE	EORB_COMM_FAILURE_M15	15	Transport failed to receive data.
COMM_FAILURE	EORB_COMM_FAILURE_M16	16	Transport failed to open connection socket.
COMM_FAILURE	EORB_COMM_FAILURE_M17	17	Transport non-blocking connection attempt has not succeeded.
COMM_FAILURE	EORB_COMM_FAILURE_M18	18	Transport failed to connect.
COMM_FAILURE	EORB_COMM_FAILURE_M20	20	Transport failed to accept connection.
COMM_FAILURE	EORB_COMM_FAILURE_M21	21	Transport failed to open socket in connection initialisation.
COMM_FAILURE	EORB_COMM_FAILURE_M22	22	Transport failed to bind socket in connection initialisation.
COMM_FAILURE	EORB_COMM_FAILURE_M23	23	Transport failed to write datagram data - connection closed.
COMM_FAILURE	EORB_COMM_FAILURE_M24	24	Transport failed to write datagram data.
COMM_FAILURE	EORB_COMM_FAILURE_M25	25	Transport failed to receive enough datagram data.
COMM_FAILURE	EORB_COMM_FAILURE_M26	26	Transport failed to open datagram connection socket.
COMM_FAILURE	EORB_COMM_FAILURE_M28	28	Transport failed to receive datagram - connection closed.
COMM_FAILURE	EORB_COMM_FAILURE_M29	29	Transport non-blocking datagram read failed

			during wait for accept request.
COMM_FAILURE	EORB_COMM_FAILURE_M30	30	Transport datagram read failed.
COMM_FAILURE	EORB_COMM_FAILURE_M32	32	Transport failed - socket already in shutdown in datagram listen.
COMM_FAILURE	EORB_COMM_FAILURE_M33	33	Transport failed to open socket in datagram connection initialisation.
COMM_FAILURE	EORB_COMM_FAILURE_M34	34	Transport failed to bind socket in datagram connection initialisation.
COMM_FAILURE	EORB_COMM_FAILURE_M36	36	Unrecoverable data loss (not reliable) in message receive pgm.
COMM_FAILURE	EORB_COMM_FAILURE_M37	37	Provided buffer was too small to receive entire APDU.
TIMEOUT	EORB_TIMEOUT_M1	1	Transport receive timeout.
TIMEOUT	EORB_TIMEOUT_M2	2	Transport connect timeout.
TIMEOUT	EORB_TIMEOUT_M3	3	Transport send timeout.
NO_IMPLEMENT	EORB_NO_IMPLEMENT_M1	1	Feature not supported by operating system.
NO_IMPLEMENT	EORB_NO_IMPLEMENT_M2	2	Feature not yet implemented in ORB.
NO_IMPLEMENT	EORB_NO_IMPLEMENT_M3	3	Operation Not Found.
NO_IMPLEMENT	EORB_NO_IMPLEMENT_M4	4	Protocol Not Loaded.
NO_IMPLEMENT	EORB_NO_IMPLEMENT_M6	6	No `TypeCode` Support.
NO_IMPLEMENT	EORB_NO_IMPLEMENT_M8	8	No asynchronous support.
NO_IMPLEMENT	EORB_NO_IMPLEMENT_M11	11	No TCP support.
NO_IMPLEMENT	EORB_NO_IMPLEMENT_M12	12	Realtime operation called without RTORB plugin.
MARSHAL	EORB_MARSHAL_M1	1	`TypeCode` for any marshal not found.
MARSHAL	EORB_MARSHAL_M3	3	Buffer underflow.
MARSHAL	EORB_MARSHAL_M4	4	Invalid GIOP Header.
MARSHAL	EORB_MARSHAL_M5	5	Invalid GIOP Reply Status.
MARSHAL	EORB_MARSHAL_M6	6	Invalid GIOP Message Type.
MARSHAL	EORB_MARSHAL_M9	9	Unsupported type.
MARSHAL	EORB_MARSHAL_M10	10	Any or fixed type support not configured .
TIMEOUT	EORB_TIMEOUT_M1	1	Transport receive timeout.
TIMEOUT	EORB_TIMEOUT_M2	2	Transport connect timeout.
TIMEOUT	EORB_TIMEOUT_M3	3	Transport send timeout.
INV_OBJREF	ORB_INV_OBJREF_M1	1	Can't Bind.
OBJECT_NOT_EXIST	EORB_OBJECT_NOT_EXIST_M1	1	Object Not Found.
INV_POLICY	EORB_INV_POLICY_M1	1	No thread pool found.
INV_POLICY	EORB_INV_POLICY_M2	2	Failed to listen on policy specified transport.
INV_POLICY	EORB_INV_POLICY_M3	3	Failed to bind to policy specified transport.
UNKNOWN	EORB_UNKNOWN_M1	1	Inbound priority transform returned false.
UNKNOWN	EORB_UNKNOWN_M2	1	Outbound priority transform returned false.

3.3 Standard User Exceptions

The ORB's standard User Exceptions are described below. These exceptions may be returned as a result of any operation invocation, regardless of the interface specification. Standard user exceptions may not be listed in `raises` expressions. The ORB's System Exceptions are described under *Standard System Exceptions*.

PortableServer_POA_AdapterAlreadyExists

The `AdapterAlreadyExists` exception is thrown when trying to create a new, child POA on a target POA (using *`POA_create_POA`*) and the target POA already has a child POA with the same name.

PortableServer_POA_AdapterNonExistent

The `AdapterNonExistent` is thrown when trying to find a target POA's named child POA (using *`POA_find_POA`*) and the child POA's name is not found (i.e., the child POA does not exist).

PortableServer_POA_InvalidPolicy

The `InvalidPolicy` exception is thrown when creating a POA (using *`POA_create_POA`*) and if any of the following conditions occur:

- the policy objects specified are not valid for the ORB
- conflicting policy objects are specified
- the specified policy objects require prior administrative action that has not been performed

PortableServer_POA_ObjectAlreadyActive

The `ObjectAlreadyActive` exception is thrown when creating a CORBA object with a user ID (using `POA_activate_object_with_id`) and an existing object with the same ID value is already active in the POA.

PortableServer_POA_ObjectNotActive

The `ObjectNotActive` exception is thrown when trying to access or perform an operation on an object that has not been activated.

PortableServer_POA_ServantAlreadyActive

The `ServantAlreadyActive` is thrown when attempting to activate an object that has already been activated (e.g. when using `POA_activate` or `POA_activate_with_id`).

PortableServer_POA_ServantNotActive

The `ServantNotActive` exception is thrown when performing an operation which requires or expects an activated servant, but the servant is not activated.

PortableServer_POA_WrongAdapter

The `WrongAdapter` exception is thrown when performing an operation from a POA (e.g. `POA_reference_to_servant`) on an object and the object was created by another POA.

PortableServer_POA_WrongPolicy

The `WrongPolicy` exception is thrown when one or more policies, that an operation expects or depends on, has not been set or is not present.

CORBA_ORB_InvalidName

This exception is raised when an operation is passed a name, such as the string value of an `ObjectId`, which is invalid. The name could be invalid because:

- the name value passed to the operation is an empty string
- the name being passed to the operation does not exist (e.g. when using `resolve_intial_references` to identify an object)
- the name being used for an object.

3.4 Standard System Exceptions

The ORB's standard System Exceptions are described below. These exceptions may be returned as a result of any operation invocation, regardless of the interface specification.

CORBA_UNKNOWN

This exception is raised if an operation implementation throws a non-CORBA exception (such as an exception specific to the implementation's programming language), or if an operation raises a user exception that does not appear in the operation's raises expression. `UNKNOWN` is also raised if the server returns a system exception that is unknown to the client. (This can happen if the server uses a later version of CORBA than the client and new system exceptions have been added to the later version.)

CORBA_BAD_PARAM

A parameter passed to a call is out of range or otherwise considered illegal. An ORB may raise this exception if null values or null pointers are passed to an operation (for language mappings where the concept of a null pointers or null values applies). `BAD_PARAM` can also be raised as a result of client generating requests with incorrect parameters using the DII.

CORBA_NO_MEMORY

The ORB cannot allocate new memory..

CORBA_IMP_LIMIT

This exception indicates that an implementation limit was exceeded in the ORB run time. For example, an ORB may reach the maximum number of references it can hold simultaneously in an address space, the size of a parameter may have exceeded the allowed maximum, or an ORB may impose a maximum on the number of clients or servers that can run simultaneously.

CORBA_COMM_FAILURE

This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.

CORBA_INV_OBJREF

This exception indicates that an object reference is internally malformed. For example, the repository ID may have incorrect syntax or the addressing information may be invalid. This exception is raised by `ORB_string_to_object` if the passed string does not decode correctly. An ORB may choose to detect calls via nil references (but is not obliged to do detect them). `INV_OBJREF` is used to indicate this.

CORBA_NO_PERMISSION

An invocation failed because the caller has insufficient privileges.

CORBA_INTERNAL

This exception indicates an internal failure in an ORB, for example, if an ORB has detected corruption of its internal data structures.

CORBA_NO_IMPLEMENT

This exception indicates that even though the operation that was invoked exists (it has an IDL definition), no implementation for that operation exists. `NO_IMPLEMENT` can, for example, be raised by an ORB if a client asks for an object's type definition from the interface repository, but no interface repository is provided by the ORB.

CORBA_BAD_TYPECODE

The ORB has encountered a malformed type code (for example, a type code with an invalid `TCKind` value).

CORBA_BAD_OPERATION

This indicates that an object reference denotes an existing object, but that the object does not support the operation that was invoked.

CORBA_NO_RESOURCES

The ORB has encountered some general resource limitation. For example, the run time may have reached the maximum permissible number of open connections.

CORBA_NO_RESPONSE

This exception is raised if a client attempts to retrieve the result of a deferred synchronous call, but the response for the request is not yet available.

CORBA_PERSIST_STORE

This exception indicates a persistent storage failure, for example, failure to establish a database connection or corruption of a database.

CORBA_BAD_INV_ORDER

This exception indicates that the caller has invoked operations in the wrong order. For example, it can be raised by an ORB if an application makes an ORB-related call without having correctly initialized the ORB first.

CORBA_TRANSIENT

`TRANSIENT` indicates that the ORB attempted to reach an object and failed. It is not an indication that an object does not exist. Instead, it simply means that no further determination of an object's status was possible because it could not be reached. This exception is raised if an attempt to establish a connection fails, for example, because the server or the implementation repository is down.

CORBA_FREE_MEM

The ORB failed in an attempt to free dynamic memory, for example because of heap corruption or memory segments being locked.

CORBA_INV_IDENT

This exception indicates that an IDL identifier is syntactically invalid. It may be raised if, for example, an identifier passed to the interface repository does not conform to IDL identifier syntax, or if an illegal operation name is used with the DII.

CORBA_INV_FLAG

An invalid flag was passed to an operation (for example, when creating a DII request).

CORBA_OBJ_ADAPTER

This exception typically indicates an administrative mismatch. For example, a server may have made an attempt to register itself with an implementation repository under a name that is already in use, or is unknown to the repository. `OBJ_ADAPTER` is also raised by the POA to indicate problems with application-supplied servant managers.

CORBA_DATA_CONVERSION

This exception is raised if an ORB cannot convert the representation of data as marshalled into its native representation or vice-versa. For example, `DATA_CONVERSION` can be raised if wide character code set conversion fails, or if an ORB cannot convert floating point values between different representations.

CORBA_OBJECT_NOT_EXIST

The `OBJECT_NOT_EXIST` exception is raised whenever an invocation on a deleted object was performed. It is an authoritative *hard* fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate “final recovery” style procedures. Bridges forward this exception to clients, also destroying any records they may hold (for example, proxy objects used in reference translation). The clients could in turn purge any of their own data structures.

CORBA_MARSHAL

A request or reply from the network is structurally invalid. This error typically indicates a bug in either the client-side or server-side run time. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception. `MARSHAL` can also be caused by using the DII or DSI incorrectly, for example, if the type of the actual parameters sent does not agree with IDL signature of an operation.

CORBA_INITIALIZE

An ORB has encountered a failure during its initialization, such as failure to acquire networking resources or detecting a configuration error.

3.5 Policies

The ORB allows access to certain choices that affect its operation. This information is accessed in a structured manner using interfaces derived from the `Policy` interface defined in the CORBA module.

```
//IDL Policy interface specification

module CORBA
{
   typedef unsigned long PolicyType;

   interface Policy
   {
      readonly attribute PolicyType policy_type;
      Policy copy ();
      void destroy ();
   };
   typedef sequence <Policy> PolicyList;
};
```

`PolicyType` defines the type of `Policy` object. In general, the constant values that are allocated are defined in conjunction with the definition of the corresponding `Policy` object. The values of `PolicyTypes` for policies that are standardized by OMG are allocated by OMG. `PolicyType`, of type `unsigned long`, consists of a 20-bit *Vendor PolicyType Valueset ID* (`VPVID`) in the high order 20 bits, and a vendor assigned policy value in the low order 12 bits. The `VPVIDs 0` through `\xf` are reserved for the OMG. All values for the standard `PolicyTypes` are allocated within this range by the OMG. For Spectra ORB-specific policies, the `VPVID` is set to `EORB_VPVID`.

3.5.1 Policy Values

Supported CORBA standard policy APIs plus the ORB's specific policy extensions are described below. Policies can be set at `ORB`, `POA`, `Thread` or `Object` scope. The policy for an `Object` is determined by checking for a `Policy` override at the `Object`, `POA`, `Thread` and then finally `ORB` scope. If no override is found, then the default `Policy` value is used. *Table 5* gives the scope of each policy. The following policies are supported by the ORB:

- *`RELATIVE_RT_TIMEOUT_POLICY_TYPE`* - The duration allowed for a request or its reply to be delivered. When the duration is exceeded, then the request if cancelled (if a reply has not yet been received) or the reply is discarded (if one has been received).

- *`EORB_POLICY_CONNECT_TIMEOUT`* - The time out for when a client initially connects (binds) to a server.

- *`EORB_POLICY_IMPL_NAME`* - Used to set the server name for a process. This is encoded within a persistent IOR and is used to disambiguate objects created with the same id and POA name but within different server processes.

- *RTCORBA_SERVER_PROTOCOL_POLICY_TYPE* - Used to configure a POA with a specific set of profiles, for example *IIOP/DIOP*. A protocol object in the list contained by the *ServerProtocolPolicy* will be used for the selection of transports that the POA should use. It may have its *transport_protocol_properties* set to Null (if additional configuration of the transport is not required), otherwise *transport_protocol_properties* points at an instance of a subtype of *ProtocolProperties*, in which case this will be used for the transport configuration. For example, it could be an instance of *RTCORBA TCPProtocolProperties*. Refer to the *ServerProtocolPolicy* section of the *Real-time CORBA Specification* for additional information. Functions, in addition to those in the standard TCPProtocolProperties interface, are provided in Spectra ORB which enable additional transport values to be set. These additional functions are listed under Additional *Transport Value Setting Functions* . The persistent example in the product distribution demonstrates how to set RTCORBA policies.

- EORB_POLICY_CONNECT_TIMEOUT Sets the ORB transport connection time-out.

- *RELATIVE_RT_TIMEOUT_POLICY_TYPE* and *RTCORBA_SERVER_PROTOCOL_POLICY_TYPE* are standard CORBA policies.

All other policies are proprietary Spectra ORB policies.

Table 5 Policies

Policy	Scope	Type
EORB_POLICY_CONNECT_TIMEOUT	Object	TC_TimeBase_TimeT
RELATIVE_RT_TIMEOUT_POLICY_TYPE	Object	TC_TimeBase_TimeT

3.5.1.1 Additional Transport Value Setting Functions

```
void EORB_TCP (void)

CORBA_boolean EORB_TCP_ProtocolProperties__get_cloexec
(
   EORB_TCP_ProtocolProperties props,
   CORBA_Environment * ev
)

void EORB_TCP_ProtocolProperties__set_cloexec
(
   EORB_TCP_ProtocolProperties props,
   CORBA_boolean cloexec,
   CORBA_Environment * ev
)

char * EORB_TCP_ProtocolProperties__get_host
(
   EORB_TCP_ProtocolProperties props,
   CORBA_Environment * ev
)

void EORB_TCP_ProtocolProperties__set_host
(
   EORB_TCP_ProtocolProperties props,
   char * host,
```

```
   CORBA_Environment * ev
)

CORBA_unsigned_short EORB_TCP_ProtocolProperties__get_port
(
   EORB_TCP_ProtocolProperties props,
   CORBA_Environment * ev
)

void EORB_TCP_ProtocolProperties__set_port
(
   EORB_TCP_ProtocolProperties props,
   CORBA_unsigned_short port,
   CORBA_Environment * ev
)

CORBA_boolean EORB_TCP_ProtocolProperties__get_reuse
(
   EORB_TCP_ProtocolProperties props,
   CORBA_Environment * ev
)

void EORB_TCP_ProtocolProperties__set_reuse
(
   EORB_TCP_ProtocolProperties props,
   CORBA_boolean reuse,
   CORBA_Environment * ev
)

CORBA_boolean EORB_TCP_ProtocolProperties__get_linger
(
   EORB_TCP_ProtocolProperties props,
   CORBA_Environment * ev
)

void EORB_TCP_ProtocolProperties__set_linger
(
   EORB_TCP_ProtocolProperties props,
   CORBA_boolean linger,
   CORBA_Environment * ev
)

CORBA_long EORB_TCP_ProtocolProperties__get_lingertime
(
   EORB_TCP_ProtocolProperties props,
   CORBA_Environment * ev
)

void EORB_TCP_ProtocolProperties__set_lingertime
(
   EORB_TCP_ProtocolProperties props,
   CORBA_long ltime,
   CORBA_Environment * ev
)

CORBA_boolean EORB_TCP_ProtocolProperties__get_debug
(
   EORB_TCP_ProtocolProperties props,
   CORBA_Environment * ev
)
```

```
void EORB_TCP_ProtocolProperties__set_debug
(
   EORB_TCP_ProtocolProperties props,
   CORBA_boolean debug,
   CORBA_Environment * ev
)
```

3.5.2 Policy Errors

3.5.2.1 Policy Error Exception

The `PolicyError` exception is raised to indicate problems with parameter values passed to the CORBA_ORB_create_policy operation.

3.5.2.2 Policy Error Codes

A request to create a Policy may be invalid. The reasons, `PolicyErrorCodes`, are listed below:

- *`CORBA_BAD_POLICY`* - the requested Policy is not understood by the ORB.
- *`CORBA_UNSUPPORTED_POLICY`* - the requested Policy is understood to be valid by the ORB, but is not currently supported.
- *`CORBA_BAD_POLICY_TYPE`* - The type of the value requested for the Policy is not valid for that `PolicyType`.
- *`CORBA_BAD_POLICY_VALUE`* - The value requested for the Policy is of a valid type but is not within the valid range for that type.
- *`CORBA_UNSUPPORTED_POLICY_VALUE`* - The value requested for the Policy is of a valid type and within the valid range for that type, but this valid value is not currently supported.