ORB Interface 4

The ORB Interface chapter has been updated based on the CORE changes from (ptc/98-09-04) and the Objects by Value documents (ptc/98-07-06) and (orbos/98-01-18). Changes from RTF 2.4 (ptc/99-03-01) and policy management related material from the Messaging specification (orbos/98-05-05) have also been incorporated.

Contents

This chapter contains the following sections.

4.1 Overview

This chapter introduces the operations that are implemented by the ORB core, and describes some basic ones, while providing reference to the description of the remaining operations that are described elsewhere. The ORB interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. The Object interface contains operations that are implemented by the ORB, and are accessed as implicit operations of the Object Reference. The ValueBase interface contains operations that are implemented by the ORB, and are accessed as implicit operations of the ValueBase Reference.

Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they are described that way for the Object or ValueBase interface operations and the language binding will, for consistency, make them appear that way.

4.2 The ORB Operations

The ORB interface contains the operations that are available to both clients and servers. These operations do not depend on any specific object adapter or any specific object reference.

```
module CORBA {

    interface NVList;           // forward declaration
    interface OperationDef;     // forward declaration
    interface TypeCode;         // forward declaration

    typedef short PolicyErrorCode;
    // for the definition of consts see “PolicyErrorCode” on page 4-22

    interface Request;                  // forward declaration
    typedef sequence <Request> RequestSeq;

    native AbstractBase;

    exception PolicyError {PolicyErrorCode reason;};

    typedef string RepositoryId;
    typedef string Identifier;

    // StructMemberSeq defined in Chapter 10
    // UnionMemberSeq defined in Chapter 10
    // EnumMemberSeq defined in Chapter 10

    typedef unsigned short ServiceType;
    typedef unsigned long ServiceOption;
    typedef unsigned long ServiceDetailType;
```

```
    const ServiceType Security = 1;

    struct ServiceDetail {
        ServiceDetailType service_detail_type;
        sequence <octet> service_detail;
    };

    struct ServiceInformation {
        sequence <ServiceOption> service_options;
        sequence <ServiceDetail> service_details;
    };

    native ValueFactory;

    interface ORB {                                          // PIDL
#pragma version ORB 2.3

        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};

        string object_to_string (
            in Object           obj
        );

        Object string_to_object (
            in string           str
        );

        // Dynamic Invocation related operations

        void create_list (
            in long             count,
            out NVList          new_list
        );

        void create_operation_list (
            in OperationDef     oper,
            out NVList          new_list
        );

        void get_default_context (
            out Context          ctx
        );

        void send_multiple_requests_oneway(
            in RequestSeq       req
        );

        void send_multiple_requests_deferred(
```

```
        in RequestSeq       req
    );

    boolean poll_next_response();

    void get_next_response(
        out Request         req
    );

    // Service information operations

    boolean get_service_information (
        in ServiceType service_type,
        out ServiceInformation service_information
    );

    ObjectIdList list_initial_services ();

    // Initial reference operation

    Object resolve_initial_references (
        in ObjectId identifier
    ) raises (InvalidName);

    // Type code creation operations

    TypeCode create_struct_tc (
        in RepositoryId id,
        in Identifier name,
        in StructMemberSeq members
    );

    TypeCode create_union_tc (
        in RepositoryId id,
        in Identifier name,
        in TypeCode discriminator_type,
        in UnionMemberSeq members
    );

    TypeCode create_enum_tc (
        in RepositoryId id,
        in Identifier name,
        in EnumMemberSeq members
    );

    TypeCode create_alias_tc (
        in RepositoryId id,
        in Identifier name,
        in TypeCode original_type
    );
```

```
TypeCode create_exception_tc (
    in RepositoryId id,
    in Identifier name,
    in StructMemberSeq members
);

TypeCode create_interface_tc (
    in RepositoryId id,
    in Identifier name
);

TypeCode create_string_tc (
    in unsigned long bound
);

TypeCode create_wstring_tc (
    in unsigned long bound
);

TypeCode create_fixed_tc (
    in unsigned short digits,
    in short scale
);

TypeCode create_sequence_tc (
    in unsigned long bound,
    in TypeCode element type
);

TypeCode create_recursive_sequence_tc // deprecated
    in unsigned long bound,
    in unsigned long offset
);

TypeCode create_array_tc (
    in unsigned long length,
    in TypeCode element_type
);

TypeCode create_value_tc (
    in RepositoryId         id,
    in Identifier           name,
    in ValueModifier        type_modifier,
    in TypeCode             concrete_base,
    in ValueMembersSeq      members
);

TypeCode create_value_box_tc (
    in RepositoryId         id,
    in Identifier           name,
    in TypeCode             boxed_type
```

```
    );
    TypeCode create_native_tc (
        in RepositoryId         id,
        in Identifier           name
    );

    TypeCode create_recursive_tc(
        in RepositoryId         id
    );

    TypeCode create_abstract_interface_tc(
        in RepositoryId         id,
        in Identifier           name
    );

    // Thread related operations

    boolean work_pending( );

    void perform_work();

    void run();

    void shutdown(
        in boolean          wait_for_completion
    );

    void destroy();

    // Policy related operations

    Policy create_policy(
        in PolicyType       type,
        in any              val
    ) raises (PolicyError);

    // Dynamic Any related operations deprecated and removed
    // from primary list of ORB operations

    // Value factory operations

    ValueFactory register_value_factory(
        in RepositoryId id,
        in ValueFactory factory
    );

    void unregister_value_factory(in RepositoryId id);

    ValueFactory lookup_value_factory(in RepositoryId id);
};
```

```
};
```

All types defined in this chapter are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by "**CORBA::**".

The operations **object_to_string** and **string_to_object** are described in "Converting Object References to Strings" on page 4-7.

For a description of the **create_list** and **create_operation_list** operations, see Section 7.4, "List Operations," on page 7-10. The **get_default_context** operation is described in the section Section 7.6.1, "get_default_context," on page 7-14. The **send_multiple_requests_oneway** and **send_multiple_requests_deferred** operations are described in the section Section 7.3.2, "send_multiple_requests," on page 7-9. The **poll_next_response** and **get_next_response** operations are described in the section Section 7.3.5, "get_next_response," on page 7-10.

The **list_intial_services** and **resolve_initial_references** operations are described in "Obtaining Initial Object References" on page 4-18.

The Type code creation operations with names of the form **create_<type>_tc** are described in Section 10.7.3, "Creating TypeCodes," on page 10-53.

The **work_pending**, **perform_work**, **shutdown**, **destroy** and **run** operations are described in "Thread-Related Operations" on page 4-33.

The **create_policy** operations is described in "Create_policy" on page 4-23.

The **register_value_factory**, **unregister_value_factory** and **lookup_value_factory** operations are described in Section 5.4.3, "Language Specific Value Factory Requirements," on page 5-9.

4.2.1 Converting Object References to Strings

4.2.1.1 object_to_string

```
string object_to_string (
    in Object           obj
);
```

4.2.1.2 string_to_object

```
Object string_to_object (
    in string           str
);
```

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems

must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

An object reference may be translated into a string by the operation **object_to_string**. The value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the **string_to_object** operation will accept a string produced by **object_to_string** and return the corresponding object reference.

To guarantee that an ORB will understand the string form of an object reference, that ORB's **object_to_string** operation must be used to produce the string. For all conforming ORBs, if **obj** is a valid reference to an object, then **string_to_object(object_to_string(obj))** will return a valid reference to the same object, if the two operations are performed on the same ORB. For all conforming ORB's supporting IOP, this remains true even if the two operations are performed on different ORBs.

4.2.2 Getting Service Information

4.2.2.1 get_service_information

```
boolean get_service_information (
    in ServiceType service_type;
    out ServiceInformation service_information;
);
```

The **get_service_information** operation is used to obtain information about CORBA facilities and services that are supported by this ORB. The service type for which information is being requested is passed in as the in parameter **service_type**, the values defined by constants in the CORBA module. If service information is available for that type, that is returned in the out parameter **service_information**, and the operation returns the value TRUE. If no information for the requested services type is available, the operation returns FALSE (i.e., the service is not supported by this ORB).

4.3 Object Reference Operations

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object reference, although the interfaces actually depend on the language binding. As above, where we used interface **Object** to represent the object reference, we define an interface for **Object**:

```
module CORBA {

    interface DomainManager;            // forward declaration
    typedef sequence <DomainManager> DomainManagersList;
```

```
interface Policy;                   // forward declaration
typedef sequence <Policy> PolicyList;
typedef unsigned long PolicyType;

interface Context;                  // forward declaration

typedef string Identifier;
interface Request;                  // forward declaration
interface NVList;                   // forward declaration
struct NamedValue{};                // an implicitly well known type
typedef unsigned long Flags;
interface InterfaceDef;

enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

interface Object {                                          // PIDL

    InterfaceDef get_interface ();

    boolean is_nil();

    Object duplicate ();

    void release ();

    boolean is_a (
        in string               logical_type_id
    );

    boolean non_existent();

    boolean is_equivalent (
        in Object               other_object
    );

    unsigned long hash(
        in unsigned long        maximum
    );

    void create_request (
        in Context              ctx
        in Identifier           operation,
        in NVList               arg_list,
        inout NamedValue        result,
        out Request             request,
        in Flags                req_flag
    );

    Policy get_policy (
        in PolicyType           policy_type
    );
```

```
        DomainManagersList get_domain_managers ();

        Object set_policy_overrides(
            in PolicyList       policies,
            in SetOverrideType  set_add
        );
    };
};
```

The **create_request** operation is part of the Object interface because it creates a pseudo-object (a Request) for an object. It is described with the other Request operations in the section Section 7.2, “Request Operations,” on page 7-4.

Unless otherwise stated below, the operations in the IDL above do not require access to remote information.

4.3.1 Determining the Object Interface

4.3.1.1 get_interface

```
InterfaceDef get_interface();
```

An operation on the object reference, **get_interface**, returns an object in the Interface Repository, which provides type information that may be useful to a program. See the Interface Repository chapter for a definition of operations on the Interface Repository. The implementation of this operation may involve contacting the ORB that implements the target object.

4.3.2 Duplicating and Releasing Copies of Object References

4.3.2.1 duplicate

```
Object duplicate();
```

4.3.2.2 release

```
void release();
```

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 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.

4.3.3 Nil Object References

4.3.3.1 is_nil

```
boolean is_nil();
```

An object reference whose value is **OBJECT_NIL** denotes no object. An object reference can be tested for this value by the **is_nil** operation. The object implementation is not involved in the nil test.

4.3.4 Equivalence Checking Operation

4.3.4.1 is_a

```
boolean is_a(
    in RepositoryId          logical_type_id
);
```

An operation is defined to facilitate maintaining type-safety for object references over the scope of an ORB.

The **logical_type_id** is a string denoting a shared type identifier (**RepositoryId**). 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 **logical_type_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, 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.

4.3.5 Probing for Object Non-Existence

4.3.5.1 non_existent

```
boolean non_existent ();
```

The **non_existent** operation may be used to test whether an object (e.g., a proxy object) has been destroyed. It does this without invoking any application level operation on the object, and so will never affect the object itself. It returns true (rather than raising CORBA::OBJECT_NOT_EXIST) if the ORB knows authoritatively that the object does not exist; otherwise, it returns false.

Services that maintain state that includes object references, such as bridges, event channels, and base relationship services, might use this operation in their "idle time" to sift through object tables for objects that no longer exist, deleting them as they go, as a form of garbage collection. In the case of proxies, this kind of activity can cascade, such that cleaning up one table allows others then to be cleaned up.

Probing for object non-existence may require contacting the ORB that implements the target object. Such an attempt may fail at either the local or the remote end. If non-existent cannot make a reliable determination of object existence due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the true, false, and indeterminate cases.

4.3.6 Object Reference Identity

In order to efficiently manage state that include large numbers of object references, services need to support a notion of object reference identity. Such services include not just bridges, but relationship services and other layered facilities.

Two identity-related operations are provided. One maps object references into disjoint groups of potentially equivalent references, and the other supports more expensive pairwise equivalence testing. Together, these operations support efficient maintenance and search of tables keyed by object references.

4.3.6.1 Hashing Object Identifiers

hash

```
unsigned long hash(
    in unsigned long        maximum
);
```

Object references are associated with ORB-internal identifiers which may indirectly be accessed by applications using the **hash** operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are *not* identical.

The **maximum** parameter to the **hash** operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision chained hash table of object references, the more randomly distributed the values are within that range, and the cheaper those values are to compute, the better.

For bridge construction, note that proxy objects are themselves objects, so there could be many proxy objects representing a given “real” object. Those proxies would not necessarily hash to the same value.

4.3.6.2 Equivalence Testing

is_equivalent

```
boolean is_equivalent(
    in Object                   other_object
);
```

The **is_equivalent** operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns TRUE if the target object reference is known to be equivalent to the other object reference passed as its parameter, and FALSE otherwise.

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

ORBs are allowed, but not required, to attempt determination of whether 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. This means that a FALSE return from **is_equivalent** should be viewed as only indicating that the object references are distinct, and not necessarily an indication that the references indicate distinct objects.

A typical application use of this operation is to match object references in a hash table. Bridges could use it to shorten the lengths of chains of proxy object references. Externalization services could use it to “flatten” graphs that represent cyclical relationships between objects. Some might do this as they construct the table, others during idle time.

4.3.7 Getting Policy Associated with the Object

4.3.7.1 get_policy

The **get_policy** operation returns the policy object of the specified type (see “Policy Object” on page 4-20), which applies to this object. It returns the *effective* **Policy** for the object reference. The effective **Policy** is the one that would be used if a request were made. This **Policy** is determined first by obtaining the *effective override* for the **PolicyType** as returned by **get_client_policy**. The effective override is then compared with the **Policy** as specified in the IOR. The effective **Policy** is the

intersection of the values allowed by the effective override and the IOR-specified **Policy**. If the intersection is empty, the system exception INV_POLICY is raised. Otherwise, a **Policy** with a value legally within the intersection is returned as the effective **Policy**. 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, the returned effective **Policy** is implementation dependent. In that situation, a compliant implementation may do any of the following: raise the system exception BAD_INV_ORDER, return some value for that **PolicyType** which may be subject to change once a binding is performed, or attempt a binding and then return the effective **Policy**. Note that if the effective **Policy** may change from invocation to invocation due to transparent rebinding.

```
Policy get_policy (
    in PolicyType       policy_type
);
```

Parameter(s)

policy_type - The type of policy to be obtained.

Return Value

A **Policy** object of the type specified by the **policy_type** parameter.

Exception(s)

CORBA::INV_POLICY - raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

The implementation of this operation may involve remote invocation of an operation (e.g. **DomainManager::get_domain_policy** for some security policies) for some policy types.

4.3.8 Overriding Associated Policies on an Object Reference

4.3.8.1 set_policy_overrides

The **set_policy_overrides** operation returns a new object reference with the new policies associated with it. It takes two input parameters. The first parameter **policies** is a sequence of references to **Policy** objects. The second parameter **set_add** of type **SetOverrideType** 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 first 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.

```
enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

Object set_policy_overrides(
    in PolicyList           policies,
    in SetOverrideType      set_add
);
```

Parameter(s)

policies - a sequence of **Policy** objects that are to be associated with the new copy of the object reference returned by this operation

set_add - whether the association is in addition to (**ADD_OVERRIDE**) or as replacement of (**SET_OVERRIDE**) any existing overrides already associated with the object reference.

Return Value

A copy of the object reference with the overrides from **policies** associated with it in accordance with the value of **set_add**.

Exception(s)

CORBA::NO_PERMISSION - raised when an attempt is made to override any policy that cannot be overridden.

4.3.9 Getting the Domain Managers Associated with the Object

4.3.9.1 get_domain_managers

The get_domain_managers operation allows administration services (and applications) to retrieve the domain managers (see “Management of Policy Domains” on page 4-28), and hence the security and other policies applicable to individual objects that are members of the domain.

```
typedef sequence <DomainManager> DomainManagersList;

DomainManagersList get_domain_managers ();
```

Return Value

The list of immediately enclosing domain managers of this object. At least one domain manager is always returned in the list since by default each object is associated with at least one domain manager at creation.

The implementation of this operation may involve contacting the ORB that implements the target object.

4.4 ValueBase Operations

ValueBase serves a similar role for value types that **Object** serves for interfaces. Its mapping is language-specific and must be explicitly specified for each language.

Typically it is mapped to a concrete language type which serves as a base for all value types. Any operations that are required to be supported for all values are conceptually defined on **ValueBase**, although in reality their actual mapping depends upon the specifics of any particular language mapping.

Analogous to the definition of the **Object** interface for implicit operations of object references, the implicit operations of **ValueBase** are defined on a pseudo-**valuetype** as follows:

```
module CORBA {
    valuetype ValueBase{                                    PIDL
        ValueDef get_value_def();
    };
};
```

The **get_value_def()** operation returns a description of the value's definition as described in the interface repository (Section 10.5.24, "ValueDef," on page 10-34).

4.5 ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and possibly the object adapter (POA) environments.
- Get references to ORB pseudo-object (for use in future ORB operations) and perhaps other objects (including the root POA or some Object Adapter objects).

The following operations are provided to initialize applications and obtain the appropriate object references:

- Operations providing access to the ORB. These operations reside in the CORBA module, but not in the ORB interface and are described in Section 4.6, "ORB Initialization," on page 4-16.
- Operations providing access to Object Adapters, Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in Section 4.7, "Obtaining Initial Object References," on page 4-18.

4.6 ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get the ORB pseudo-object reference and possibly an OA object reference (such as the root POA). This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB pseudo-object reference and the OA object reference to the application for use in future ORB and OA operations.

The ORB and OA initialization operations must be ordered with ORB occurring before OA: an application cannot call OA initialization routines until ORB initialization routines have been called for the given ORB. The operation to initialize an application in the ORB and get its pseudo-object reference is not performed on an object. This is because applications do not initially have an object on which to invoke operations. The ORB initialization operation is an application's bootstrap call into the CORBA world. The **ORB_init** call is part of the CORBA module but not part of the ORB interface.

Applications can be initialized in one or more ORBs. When an ORB initialization is complete, its pseudo reference is returned and can be used to obtain other references for that ORB.

In order to obtain an ORB pseudo-object reference, applications call the **ORB_init** operation. The parameters to the call comprise an identifier for the ORB for which the pseudo-object reference is required, and an **arg_list**, which is used to allow environment-specific data to be passed into the call. PIDL for the ORB initialization is as follows:

```
// PIDL
module CORBA {
        typedef string ORBid;
        typedef sequence <string> arg_list;
        ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

The identifier for the ORB will be a name of type **CORBA::ORBid**. All **ORBid** strings other than the empty string are allocated by ORB administrators and are not managed by the OMG. **ORBid** strings other than the empty string are intended to be used to uniquely identify each ORB used within the same address space in a multi-ORB application. These special **ORBid** strings are specific to each ORB implementation and the ORB administrator is responsible for ensuring that the names are unambiguous.

If an empty **ORBid** string is passed to **ORB_init**, then the **arg_list** arguments shall be examined to determine if they indicate an ORB reference that should be returned. This is achieved by searching the **arg_list** parameters for one preceded by "**-ORBid**" for example, "**-ORBid example_orb**" (the white space after the "**-ORBid**" tag is ignored) or "**-ORBidMyFavoriteORB**" (with no white space following the "**-ORBid**" tag). Alternatively, two sequential parameters with the first being the string "**-ORBid**" indicates that the second is to be treated as an **ORBid** parameter. If an empty string is passed and no **arg_list** parameters indicate the ORB reference to be returned, the default ORB for the environment will be returned.

Other parameters of significance to the ORB can also be identified in **arg_list**, for example, "**Hostname**," "**SpawnedServer**," and so forth. To allow for other parameters to be specified without causing applications to be re-written, it is necessary to specify the parameter format that ORB parameters may take. In general, parameters shall be formatted as either one single **arg_list** parameter:

–ORB<suffix><optional white space> <value>

or as two sequential **arg_list** parameters:

-ORB<suffix>

<value>

Regardless of whether an empty or non-empty **ORBid** string is passed to **ORB_init**, the **arg_list** 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 **arg_list** are ignored. All other **-ORB<suffix>** parameters in the **arg_list** may be of significance during the ORB initialization process.

Before **ORB_init** returns, it will remove from the **arg_list** parameter all strings that match the **-ORB<suffix>** pattern described above and that are recognized by that ORB implementation, along with any associated sequential parameter strings. If any strings in **arg_list** that match this pattern are not recognized by the ORB implementation, **ORB_init** will raise the BAD_PARAM system exception instead.

The **ORB_init** operation may be called any number of times and shall return the same ORB reference when the same **ORBid** string is passed, either explicitly as an argument to **ORB_init** or through the **arg_list**. All other **-ORB<suffix>** parameters in the **arg_list** may be considered on subsequent calls to **ORB_init**.

4.7 Obtaining Initial Object References

Applications require a portable means by which to obtain their initial object references. References are required for the root POA, POA Current, Interface Repository and various Object Services instances. (The POA is described in the Portable Object Adaptor chapter; the Interface Repository is described in the Interface Repository chapter; Object Services are described in *CORBAservices: Common Object Services Specification*.) The functionality required by the application is similar to that provided by the Naming Service. However, the OMG does not want to mandate that the Naming Service be made available to all applications in order that they may be portably initialized. Consequently, the operations shown in this section provide a simplified, local version of the Naming Service that applications can use to obtain a small, defined set of object references which are essential to its operation. Because only a small well-defined set of objects are expected with this mechanism, the naming context can be flattened to be a single-level name space. This simplification results in only two operations being defined to achieve the functionality required.

Initial references are not obtained via a new interface; instead two operations are provided in the ORB pseudo-object interface, providing facilities to list and resolve initial object references.

list_initial_services

```
typedef string ObjectId;
typedef sequence <ObjectId> ObjectIdList;
ObjectIdList list_initial_services ();
```

resolve_initial_references

```
exception InvalidName {};

Object resolve_initial_references (
    in ObjectId identifier
) raises (InvalidName);
```

The **resolve_initial_references** operation is an operation on the ORB rather than the Naming Service's **NamingContext**. The interface differs from the Naming Service's resolve in that **ObjectId** (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the name space to one context.

ObjectIds are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this route. Unlike the ORB identifiers, the **ObjectId** name space requires careful management. To achieve this, the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

Currently, reserved **ObjectIds** are **RootPOA**, **POACurrent**, **InterfaceRepository,** **NameService**, **TradingService**, **SecurityCurrent**, **TransactionCurrent,** and **DynAnyFactory.**

To allow an application to determine which objects have references available via the initial references mechanism, the **list_initial_services** operation (also a call on the ORB) is provided. It returns an **ObjectIdList,** which is a sequence of **ObjectIds**. **ObjectIds** are typed as strings. Each object, which may need to be made available at initialization time, is allocated a string value to represent it. In addition to defining the id, the type of object being returned must be defined (i.e., "**InterfaceRepository**" returns an object of type **Repository**, and "**NameService**" returns a **CosNamingContext** object).

The application is responsible for narrowing the object reference returned from **resolve_initial_references** to the type which was requested in the ObjectId. For example, for InterfaceRepository the object returned would be narrowed to **Repository** type.

In the future, specifications for Object Services (in *CORBAservices: Common Object Services Specification*) will state whether it is expected that a service's initial reference be made available via the **resolve_initial_references** operation or not (i.e., whether the service is necessary or desirable for bootstrap purposes).

4.8 Current Object

ORB and CORBA services may wish to provide access to information (context) associated with the thread of execution in which they are running. This information is accessed in a structured manner using interfaces derived from the **Current** interface defined in the CORBA module.

Each ORB or CORBA service that needs its own context derives an interface from the CORBA module's **Current**. Users of the service can obtain an instance of the appropriate **Current** interface by invoking **ORB::resolve_initial_references**. For example the Security service obtains the **Current** relevant to it by invoking

ORB::resolve_initial_references(“SecurityCurrent”)

A CORBA service does not have to use this method of keeping context but may choose to do so.

```
module CORBA {
    // interface for the Current object
    interface Current {
    };
};
```

Operations on interfaces derived from **Current** access state associated with the thread in which they are invoked, not state associated with the thread from which the **Current** was obtained. This prevents one thread from manipulating another thread's state, and avoids the need to obtain and narrow a new **Current** in each method's thread context.

Current objects must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. **Current**s are per-process singleton objects, so no destroy operation is needed.

4.9 Policy Object

4.9.1 Definition of Policy Object

An ORB or CORBA service may choose to allow 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. A CORBA service does not have to use this method of accessing operating options, but may choose to do so. The *Security Service* in particular uses this technique for associating *Security Policy* with objects in the system.

```
module CORBA {
    typedef unsigned long PolicyType;

    // Basic IDL definition
    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. Additionally, vendors may reserve blocks of 4096 PolicyType values identified by a 20 bit *Vendor PolicyType Valueset ID* (**VPVID**) for their own use.

PolicyType which is an unsigned long consists of the 20-bit **VPVID** in the high order 20 bits, and the vendor assigned policy value in the low order 12 bits. The **VPVID**s *0 through \xf* are reserved for OMG. All values for the standard **PolicyTypes** are allocated within this range by OMG. Additionally, the **VPVID**s *\xfffff* is reserved for experimental use and **OMGVMCID** (Section 3.17.1, "Standard Exception Definitions," on page 3-52) is reserved for OMG use. These will not be allocated to anybody. Vendors can request allocation of **VPVID** by sending mail to tag-*request@omg.org*.

When a **VMCID** (Section 3.17, "Standard Exceptions," on page 3-51) is allocated to a vendor automatically the same value of **VPVID** is reserved for the vendor and vice versa. So once a vendor gets either a **VMCID** or a **VPVID** registered they can use that value for both their minor codes and their policy types.

4.9.1.1 Copy

Policy copy();

Return Value

This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain, or object.

4.9.1.2 Destroy

void destroy();

This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.

Exception(s)

CORBA::NO_PERMISSION - raised when the policy object determines that it cannot be destroyed.

4.9.1.3 Policy_type

readonly attribute policy_type

Return Value

This readonly attribute returns the constant value of type **PolicyType** that corresponds to the type of the **Policy** object.

4.9.2 Creation of Policy Objects

A generic ORB operation for creating new instances of Policy objects is provided as described in this section.

```
module CORBA {

    typedef short PolicyErrorCode;
    const PolicyErrorCode BAD_POLICY = 0;
    const PolicyErrorCode UNSUPPORTED_POLICY = 1;
    const PolicyErrorCode BAD_POLICY_TYPE = 2;
    const PolicyErrorCode BAD_POLICY_VALUE = 3;
    const PolicyErrorCode UNSUPPORTED_POLICY_VALUE = 4;

    exception PolicyError {PolicyErrorCode reason;};

    interface ORB {

        .....

        Policy create_policy(
            in PolicyType type,
            in any val
        ) raises(PolicyError);
    };
};
```

4.9.2.1 PolicyErrorCode

A request to create a **Policy** may be invalid for the following reasons:

BAD_POLICY - the requested **Policy** is not understood by the ORB.

UNSUPPORTED_POLICY - the requested **Policy** is understood to be valid by the ORB, but is not currently supported.

BAD_POLICY_TYPE - The type of the value requested for the **Policy** is not valid for that **PolicyType**.

BAD_POLICY_VALUE - The value requested for the **Policy** is of a valid type but is not within the valid range for that type.

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.

4.9.2.2 PolicyError

```
exception PolicyError {PolicyErrorCode reason;};
```

PolicyError exception is raised to indicate problems with parameter values passed to the **ORB::create_policy** operation. Possible reasons are described above.

4.9.2.3 *INV_POLICY*

```
exception INV_POLICY
```

Due to an incompatibility between **Policy** overrides, the invocation cannot be made. This is a standard system exception that can be raised from any invocation.

4.9.2.4 *Create_policy*

The ORB operation **create_policy** can be invoked to create new instances of policy objects of a specific type with specified initial 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, it raises the PolicyError exception with the appropriate reason as described in “PolicyErrorCode” on page 4-22.

```
Policy create_policy(
    in PolicyType type,
    in any val
) raises(PolicyError);
```

Parameter(s)

type - the **PolicyType** of the policy object to be created.

val - the value that will be used to set the initial state of the **Policy** object that is created.

ReturnValue

Reference to a newly created **Policy** object of type specified by the **type** parameter and initialized to a state specified by the **val** parameter.

Exception(s)

PolicyError - raised when the requested policy is not supported or a requested initial state for the policy is not supported.

When new policy types are added to CORBA or CORBA Services specification, it is expected that the IDL type and the valid values that can be passed to **create_policy** also be specified.

4.9.3 *Usages of Policy Objects*

Policy Objects are used in general to encapsulate information about a specific policy, with an interface derived from the policy interface. The type of the Policy object determines how the policy information contained within it is used. Usually a Policy object is associated with another object to associate the contained policy with that object.

Objects with which policy objects are typically associated are Domain Managers, POA, the execution environment, both the process/capsule/ORB instance and thread of execution (Current object) and object references. Only certain types of policy object can be meaningfully associated with each of these types of objects.

These relationships are documented in sections that pertain to these individual objects and their usages in various core facilities and object services. The use of Policy Objects with the POA are discussed in the *Portable Object Adaptor* chapter. The use of Policy objects in the context of the Security services, involving their association with Domain Managers as well as with the Execution Environment are discussed in *CORBAservices, Security Service* chapter.

In the following section the association of Policy objects with the Execution Environment is discussed. In “Management of Policy Domains” on page 4-28 the use of Policy objects in association with Domain Managers is discussed.

4.9.4 Policy Associated with the Execution Environment

Certain policies that pertain to services like security (e.g., QOP, Mechanism, invocation credentials etc.) are associated by default with the process/capsule(RM-ODP)/ORB instance (hereinafter referred to as “capsule”) when the application is instantiated together with the capsule. By default these policies are applicable whenever an invocation of an operation is attempted by any code executing in the said capsule. The Security service provides operations for modulating these policies on a per-execution thread basis using operations in the **Current** interface. Certain of these policies (e.g., invocation credentials, qop, mechanism etc.) which pertain to the invocation of an operation through a specific object reference can be further modulated at the client end, using the **set_policy_overrides** operation of the **Object** reference. For a description of this operation see “Overriding Associated Policies on an Object Reference” on page 4-14. It associates a specified set of policies with a newly created object reference that it returns.

The association of these overridden policies with the object reference is a purely local phenomenon. These associations are never passed on in any IOR or any other marshaled form of the object reference. the associations last until the object reference in the capsule is destroyed or the capsule in which it exists is destroyed.

The policies thus overridden in this new object reference and all subsequent duplicates of this new object reference apply to all invocations that are done through these object references. The overridden policies apply even when the default policy associated with **Current** is changed. It is always possible that the effective policy on an object reference at any given time will fail to be successfully applied, in which case the invocation attempt using that object reference will fail and return a CORBA::NO_PERMISSION exception. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. These are listed in the Security specification. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

In general the policy of a specific type that will be used in an invocation through an specific object reference using a specific thread of execution is determined first by determining if that policy type has been overridden in that object reference. if so then the overridden policy is used. if not then if the policy has been set in the thread of execution then that policy is used. If not then the policy associated with the capsule is used. For policies that matter, the ORB ensures that there is a default policy object of each type that matters associated with each capsule (ORB instance). Hence, in a correctly implemented ORB there is no case when a required type policy is not available to use with an operation invocation.

4.9.5 Specification of New Policy Objects

When new **PolicyType**s are added to CORBA specifications, the following details must be defined. It must be clearly stated which particular uses of a new policy are legal and which are not:

- Specify the assigned **CORBA::PolicyType** and the policy's interface definition.
- If the **Policy** can be created through **CORBA::ORB::create_policy**, specify the allowable values for the any argument 'val' and how they correspond to the initial state/behavior of that **Policy** (such as initial values of attributes). For example, if a Policy has multiple attributes and operations, it is most likely that create_policy will receive some complex data for the implementation to initialize the state of the specific policy:

```
//IDL
struct MyPolicyRange {
      long low;
      long high;
};

const CORBA::PolicyType MY_POLICY_TYPE = 666;
interface MyPolicy : Policy {
      readonly attribute long low;
      readonly attribute long high;
};
```

  If this sample **MyPolicy** can be constructed via create_policy, the specification of **MyPolicy** will have a statement such as: “When instances of **MyPolicy** are created, a value of type **MyPolicyRang**e is passed to **CORBA::ORB::create_policy** and the resulting MyPolicy's attribute 'low' has the same value as the **MyPolicyRange** member 'low' and attribute 'high' has the same value as the **MyPolicyRange** member 'high'.

- If the **Policy** can be passed as an argument to **POA::create_POA**, specify the effects of the new policy on that **POA**. Specifically define incompatibilities (or inter-dependencies) with other **POA** policies, effects on the behavior of invocations on objects activated with the **POA**, and whether or not presence of the POA policy implies some **IOR** profile/component contents for object references created with

that **POA**. If the **POA** policy implies some addition/modification to the object reference it is marked as “client-exposed” and the exact details are specified including which profiles are affected and how the effects are represented.

- If the component which is used to carry this information. can be set within a client to tune the client's behavior, specify the policy's effects on the client specifically with respect to (a) establishment of connections and reconnections for an object reference; (b) effects on marshaling of requests; (c) effects on insertion of service contexts into requests; (d) effects upon receipt of service contexts in replies. In addition, incompatibilities (or inter-dependencies) with other client-side policies are stated. For policies that cause service contexts to be added to requests, the exact details of this addition are given.
- If the **Policy** can be used with **POA** creation to tune **IOR** contents and can also be specified (overridden) in the client, specify how to reconcile the policy's presence from both the client and server. It is strongly recommended to avoid this case! As an exercise in completeness, most **POA** policies can probably be extended to have some meaning in the client and vice versa, but this does not help make usable systems, it just makes them more complicated without adding really useful features. There are very few cases where a policy is really appropriate to specify in both places, and for these policies the interaction between the two must be described.
- Pure client-side policies are assumed to be immutable. This allows efficient processing by the runtime that can avoid re-evaluating the policy upon every invocation and instead can perform updates only when new overrides are set (or policies change due to rebind). If the newly specified policy is mutable, it must be clearly stated what happens if non-readonly attributes are set or operations are invoked that have side-effects.
- For certain policy types, override operations may be disallowed. If this is the case, the policy specification must clearly state what happens if such overrides are attempted.

4.9.6 Standard Policies

Table 4-1 below lists the standard policy types that are defined by various parts of CORBA and CORBA Services in this version of CORBA.

Table 4-1 Standard Policy Types

Policy Type	Policy Interface	Defined in Sect./Page	Uses create_ policy
SecClientInvocationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecTargetInvocationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecApplicationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecClientInvocationAudit	SecurityAdmin::AuditPolicy	Security Service	No
SecTargetInvocationAudit	SecurityAdmin::AuditPolicy	Security Service	No

Table 4-1 Standard Policy Types

Policy Type	Policy Interface	Defined in Sect./Page	Uses create_ policy
SecApplicationAudit	SecurityAdmin::AuditPolicy	Security Service	No
SecDelegation	SecurityAdmin::DelegationPolicy	Security Service	No
SecClientSecureInvocation	SecurityAdmin::SecureInvocationPolicy	Security Service	No
SecTargetSecureInvocation	SecurityAdmin::SecureInvocationPolicy	Security Service	No
SecNonRepudiation	NRService::NRPolicy	Security Service	No
SecConstruction	CORBA::SecConstruction	CORBA Core - ORB Interface chapter	No
SecMechanismPolicy	SecurityLevel2::MechanismPolicy	Security Service	Yes
SecInvocationCredentialsPolicy	SecurityLevel2::InvocationCredentialsPolicy	Security Service	Yes
SecFeaturesPolicy	SecurityLevel2::FeaturesPolicy	Security Service	Yes
SecQOPPolicy	SecurityLevel2::QOPPolicy	Security Service	Yes
THREAD_POLICY_ID	PortableServer::ThreadPolicy	CORBA Core - Portable Object Adapter chapter	Yes
LIFESPAN_POLICY_ID	PortableServer::LifespanPolicy	CORBA Core - Portable Object Adapter chapter Core Chapter 11	Yes
ID_UNIQUENESS_POLICY_ID	PortableServer::IdUniquenessPolicy	CORBA Core - Portable Object Adapter chapter Core Chapter 11	Yes
ID_ASSIGNMENT_POLICY_ID	PortableServer::IdAssignmentPolicy	CORBA Core - Portable Object Adapter chapter	Yes
IMPLICIT_ACTIVATION_POLICY_ID	PortableServer::ImplicitActivationPolicy	CORBA Core - Portable Object Adapter chapter	Yes
SERVENT_RETENTION_POLICY_ID	PortableServer::ServentRetentionPolicy	CORBA Core - Portable Object Adapter chapter	Yes
REQUEST_PROCESSING_POLICY_ID	PortableServer::RequestProcessingPolicy	CORBA Core - Portable Object Adapter chapter	Yes
BIDIRECTIONAL_POLICY_TYPE	BiDirPolicy::BidirectionalPolicy	CORBA Core - General Inter-ORB Protocol chapter	Yes
SecDelegationDirectivePolicy	SecurityLevel2::DelegtionDirectivePolicy	Security Service	Yes
SecEstablishTrustPolicy	SecurityLevel2::EstablishTrustPolicy	Security Service	Yes

4.10 Management of Policy Domains

4.10.1 Basic Concepts

This section describes how policies, such as security policies, are associated with objects that are managed by an ORB. The interfaces and operations that facilitate this aspect of management is described in this section together with the section describing **Policy** objects.

4.10.1.1 Policy Domain

A policy domain is a set of objects to which the policies associated with that domain apply. These objects are the domain members. The policies represent the rules and criteria that constrain activities of the objects which belong to the domain. On object reference creation, the ORB implicitly associates the object reference with one or more policy domains. Policy domains provide leverage for dealing with the problem of scale in policy management by allowing application of policy at a domain granularity rather than at an individual object instance granularity.

4.10.1.2 Policy Domain Manager

A policy domain includes a unique object, one per policy domain, called the domain manager, which has associated with it the policy objects for that domain. The domain manager also records the membership of the domain and provides the means to add and remove members. The domain manager is itself a member of a domain, possibly the domain it manages.

4.10.1.3 Policy Objects

A policy object encapsulates a policy of a specific type. The policy encapsulated in a policy object is associated with the domain by associating the policy object with the domain manager of the policy domain.

There may be several policies associated with a domain, with a policy object for each. There is at most one policy of each type associated with a policy domain. The policy objects are thus shared between objects in the domain, rather than being associated with individual objects. Consequently, if an object needs to have an individual policy, then it must be a singleton member of a domain.

4.10.1.4 Object Membership of Policy Domains

Since the only way to access objects is through object references, associating object references with policy domains, implicitly associates the domain policies with the object associated with the object reference. Care should be taken by the application that is creating object references using POA operations to ensure that object references

to the same object are not created by the server of that object with different domain associations. Henceforth whenever the concept of “object membership” is used, it actually means the membership of an object reference to the object in question.

An object can simultaneously be a member of more than one policy domain. In that case the object is governed by all policies of its enclosing domains. The reference model allows an object to be a member of multiple domains, which may overlap for the same type of policy (for example, be subject to overlapping access policies). This would require conflicts among policies defined by the multiple overlapping domains to be resolved. The specification does not include explicit support for such overlapping domains and, therefore, the use of policy composition rules required to resolve conflicts at policy enforcement time.

Policy domain managers and policy objects have two types of interfaces:

- The operational interfaces used when enforcing the policies. These are the interfaces used by the ORB during an object invocation. Some policy objects may also be used by applications, which enforce their own policies.

  The caller asks for the policy of a particular type (e.g., the delegation policy), and then uses the policy object returned to enforce the policy. The caller finding a policy and then enforcing it does not see the domain manager objects and the domain structure.

- The administrative interfaces used to set policies (e.g., specifying which events to audit or who can access objects of a specified type in this domain). The administrator sees and navigates the domain structure, so he is aware of the scope of what he is administering.

Note – This specification does not include any explicit interfaces for managing the policy domains themselves: creating and deleting them; moving objects between them; changing the domain structure and adding, changing, and removing policies applied to the domains.

4.10.1.5 Domains Association at Object Reference Creation

When a new object reference is created, the ORB implicitly associates the object reference (and hence the object that it is associated with) with the following elements forming its environment:

- One or more *Policy Domains*, defining all the policies to which the object associated with the object reference is subject.
- The *Technology Domains*, characterizing the particular variants of mechanisms (including security) available in the ORB.

The ORB will establish these associations when one of the object reference creation operations of the POA is called. Some or all of these associations may subsequently be explicitly referenced and modified by administrative or application activity, which might be specifically security-related but could also occur as a side-effect of some other activity, such as moving an object to another host machine.

In some cases, when a new object reference is created, it needs to be associated with a new domain. Within a given domain a construction policy can be associated with a specific object type thus causing a new domain (i.e., a domain manager object) to be created whenever an object reference of that type is created and the newly created object reference associated with the new domain manager. This construction policy is enforced at the same time as the domain membership (i.e., by the POA when it creates an object reference).

4.10.1.6 Implementor's View of Object Creation

For policy domains, the construction policy of the application or factory creating the object proceeds as follows. The application (which may be a generic factory) calls one of the object reference creation operations of the POA to create the new object reference. The ORB obtains the construction policy associated with the creating object, or the default domain absent a creating object.

By default, the new object reference that is created is made a member of the domain to which the parent belongs. Non-object applications on the client side are associated with a default, per-ORB instance policy domain by the ORB.

Each domain manager has a construction policy associated with it, which controls whether, in addition to creating the specified new object reference, a new domain manager is created with it. This object provides a single operation **make_domain_manager** which can be invoked with the **constr_policy** parameter set to TRUE to indicate to the ORB that new object references of the specified type are to be associated their own separate domains. Once such a construction policy is set, it can be reversed by invoking **make_domain_manager** again with the **constr_policy** parameter set to FALSE.

When creating an object reference of the type specified in the **make_domain_manager** call with **constr_policy** set to TRUE, the ORB must also create a new domain for the newly created object reference. If a new domain is needed, the ORB creates both the requested object reference and a domain manager object. A reference to this domain manager can be found by calling **get_domain_managers** on the newly created object reference.

While the management interface to the construction policy object is standardized, the interface from the ORB to the policy object is assumed to be a private one, which may be optimized for different implementations.

If a new domain is created, the policies initially applicable to it are the policies of the enclosing domain. The ORB will always arrange to provide a default enclosing domain with default ORB policies associated with it, in those cases where there would be no such domain as in the case of a non-object client invoking object creation operations.

The calling application, or an administrative application later, can change the domains to which this object belongs, using the domain management interfaces, which will be defined in the future.

Since the ORB has control only over domain associations with object references, it is the responsibility of the creator of new object to ensure that the object references that are created to the new object are associated meaningfully with domains.

4.10.2 Domain Management Operations

This section defines the interfaces and operations needed to find domain managers and find the policies associated with these. However, it does not include operations to manage domain membership, structure of domains, or to manage which policies are associated with domains.

This section also includes the interface to the construction policy object, as that is relevant to domains. The basic definitions of the interfaces and operations related to these are part of the CORBA module, since other definitions in the CORBA module depend on these.

```
module CORBA {
    interface DomainManager {
        Policy get_domain_policy (
            in PolicyType policy_type
        );
    };

    const PolicyType SecConstruction = 11;

    interface ConstructionPolicy: Policy{
        void make_domain_manager(
            in CORBA::InterfaceDef object_type,
            in boolean constr_policy
        );
    };

    typedef sequence <DomainManager> DomainManagersList;
};
```

4.10.2.1 Domain Manager

The domain manager provides mechanisms for:

- Establishing and navigating relationships to superior and subordinate domains.
- Creating and accessing policies.

There should be no unnecessary constraints on the ordering of these activities, for example, it must be possible to add new policies to a domain with a preexisting membership. In this case, some means of determining the members that do not conform to a policy that may be imposed is required. It should be noted that interfaces for adding new policies to domains or for changing domain memberships have not currently been standardized.

All domain managers provide the **get_domain_policy** operation. By virtue of being an object, the Domain Managers also have the **get_policy** and **get_domain_managers** operations, which is available on all objects (see “Getting Policy Associated with the Object” on page 4-13 and “Getting the Domain Managers Associated with the Object” on page 4-15).

CORBA::DomainManager::get_domain_policy

This returns the policy of the specified type for objects in this domain.

```
Policy get_domain_policy (
    in PolicyType policy_type
);
```

Parameter(s)

policy_type - The type of policy for objects in the domain which the application wants to administer. For security, the possible policy types are described in *CORBAservices: Common Object Services Specification,* Security chapter, Security Policies Introduction section.

Return Value

A reference to the policy object for the specified type of policy in this domain.

Exception(s)

CORBA::INV_POLICY - raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

4.10.2.2 Construction Policy

The construction policy object allows callers to specify that when instances of a particular object reference are created, they should be automatically assigned membership in a newly created domain at creation time.

CORBA::ConstructionPolicy::make_domain_manager

This operation enables the invoker to set the construction policy that is to be in effect in the domain with which this **ConstructionPolicy** object is associated. Construction Policy can either be set so that when an object reference of the type specified by the input parameter is created, a new domain manager will be created and the newly created object reference will respond to **get_domain_managers** by returning a reference to this domain manager. Alternatively the policy can be set to associate the newly created object reference with the domain associated with the creator. This policy is implemented by the ORB during execution of any one of the object reference creation operations of the POA, and results in the construction of the application-specified object reference and a Domain Manager object if so dictated by the policy in effect at the time of the creation of the object reference.

```
void make_domain_manager (
    in InterfaceDef object_type,
    in boolean constr_policy
);
```

Parameter(s)

object_type - The type of the object references for which Domain Managers will be created. If this is nil, the policy applies to all object references in the domain.

constr_policy - If TRUE the construction policy is set to create a new domain manager associated with the newly created object reference of this type in this domain. If FALSE construction policy is set to associate the newly created object references with the domain of the creator or a default domain as described above.

4.11 Thread-Related Operations

To support single-threaded ORBs, as well as multi-threaded ORBs that run multi-thread-unaware code, several operations are included in the ORB interface. These operations can be used by single-threaded and multi-threaded applications. An application that is a pure ORB client would not need to use these operations. Both the **ORB::run** and **ORB::shutdown** are useful in fully multi-threaded programs.

Note – These operations are defined on the ORB rather than on an object adapter to allow the main thread to be used for all kinds of asynchronous processing by the ORB. Defining these operations on the ORB also allows the ORB to support multiple object adapters, without requiring the application main to know about all the object adapters. The interface between the ORB and an object adapter is not standardized.

4.11.1 work_pending

```
boolean work_pending( );
```

This operation returns an indication of whether the ORB needs the main thread to perform some work.

A result of TRUE indicates that the ORB needs the main thread to perform some work and a result of FALSE indicates that the ORB does not need the main thread.

4.11.2 perform_work

```
void perform_work();
```

If called by the main thread, this operation performs an implementation-defined unit of work; otherwise, it does nothing.

It is platform-specific how the application and ORB arrange to use compatible threading primitives.

The **work_pending()** and **perform_work()** operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multi-threaded server would need a polling loop only if there were both ORB and other code that required use of the main thread.

Here is an example of such a polling loop:

```
// C++
for (;;) {
   if (orb->work_pending()) {
      orb->perform_work();
   };
   // do other things
   // sleep?
};
```

Once the ORB has shutdown, **work_pending** and **perform_work** will raise the BAD_INV_ORDER exception with minor code 4. An application can detect this exception to determine when to terminate a polling loop.

4.11.3 run

void run();

This operation provides execution resources to the ORB so that it can perform its internal functions. Single threaded ORB implementations, and some multi-threaded ORB implementations, need the use of the main thread in order to function properly. For maximum portability, an application should call either **run** or **perform_work** on its main thread. **run** may be called by multiple threads simultaneously.

This operation will block until the ORB has completed the shutdown process, initiated when some thread calls **shutdown**.

4.11.4 shutdown

```
void shutdown(
    in boolean          wait_for_completion
);
```

This operation instructs the ORB to shut down, that is, to stop processing in preparation for destruction.

Shutting down the ORB causes all object adapters to be destroyed, since they cannot exist in the absence of an ORB. Shut down is complete when all ORB processing (including request processing and object deactivation or other operations associated with object adapters) has completed and the object adapters have been destroyed. In the case of the **POA**, this means that all object etherealizations have finished and root **POA** has been destroyed (implying that all descendent **POA**s have also been destroyed).

If the **wait_for_completion** parameter is **TRUE**, this operation blocks until the shut down is complete. If an application does this in a thread that is currently servicing an invocation, the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, since blocking would result in a deadlock.

If the **wait_for_completion** parameter is **FALSE**, then **shutdown** may not have completed upon return. An ORB implementation may require the application to call (or have a pending call to) **run** or **perform_work** after **shutdown** has been called with its parameter set to **FALSE**, in order to complete the shutdown process.

While the ORB is in the process of shutting down, the ORB operates as normal, servicing incoming and outgoing requests until all requests have been completed. An implementation may impose a time limit for requests to complete while a **shutdown** is pending.

Once an ORB has shutdown, only object reference management operations(**duplicate**, **release** and **is_nil**) may be invoked on the ORB or any object reference obtained from it. An application may also invoke the destroy operation on the ORB itself. Invoking any other operation will raise the BAD_INV_ORDER system exception with the OMG minor code 4.

4.11.5 destroy

void destroy();

This operation destroys the ORB so that its resources can be reclaimed by the application. Any operation invoked on a destroyed ORB reference will raise the OBJECT_NOT_EXIST exception. Once an ORB has been destroyed, another call to **ORB_init** with the same **ORBid** will return a reference to a newly constructed ORB.

If **destroy** is called on an ORB that has not been shut down, it will start the shut down process and block until the ORB has shut down before it destroys the ORB. If an application calls **destroy** in a thread that is currently servicing an invocation, the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, since blocking would result in a deadlock.

For maximum portability and to avoid resource leaks, an application should always call **shutdown** and **destroy** on all ORB instances before exiting.