Dynamic Invocation Interface 5

The Dynamic Invocation Interface (DII) describes the client's side of the interface that allows dynamic creation and invocation of request to objects. 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::".

Contents

This chapter contains the following sections.

5.1 Overview

The Dynamic Invocation Interface (DII) allows dynamic creation and invocation of requests to objects. A client using this interface to send a request to an object obtains the same semantics as a client using the operation stub generated from the type specification.

A request consists of an object reference, an operation, and a list of parameters. The ORB applies the implementation-hiding (encapsulation) principle to requests.

In the Dynamic Invocation Interface, parameters in a request are supplied as elements of a list. Each element is an instance of a **NamedValue** (see "Common Data Structures" on page 5-2). Each parameter is passed in its native data form.

Parameters supplied to a request may be subject to run-time type checking upon request invocation. Parameters must be supplied in the same order as the parameters defined for the operation in the Interface Repository.

The user exception **WrongTransaction** is defined in the CORBA module, prior to the definitions of the ORB and Request interfaces, as follows:

```
exception WrongTransaction {};
```

This exception can be raised only if the request is implicitly associated with a transaction (the current transaction at the time that the request was issued).

5.1.1 Common Data Structures

The type **NamedValue** is a well-known data type in OMG IDL. It can be used either as a parameter type directly or as a mechanism for describing arguments to a request. The type **NVList** is a pseudo-object useful for constructing parameter lists. The types are described in OMG IDL and C, respectively, as:

```
typedef unsigned long Flags;

struct NamedValue {
    Identifier      name;       // argument name
    any             argument;   // argument
    long            len;        // length/count of argument value
    Flags           arg_modes;// argument mode flags
};

CORBA_NamedValue * CORBA_NVList;                    /* C */
```

NamedValue and **Flags** are defined in the CORBA module.

The **NamedValue** and **NVList** structures are used in the request operations to describe arguments and return values. They are also used in the context object routines to pass lists of property names and values. Despite the above declaration for **NVList**, the **NVList** structure is partially opaque and may only be created by using the ORB **create_list** operation.

For out parameters, applications can set the **argument** member of the **NamedValue** structure to a value that includes either a NULL or a non-NULL storage pointer. If a non-null storage pointer is provided for an out parameter, the ORB will attempt to use the storage pointed to for holding the value of the out parameter. If the storage pointed to is not sufficient to hold the value of the out parameter, the behavior is undefined.

A named value includes an argument name, argument value (as an **any**), length of the argument, and a set of argument mode flags. When named value structures are used to describe arguments to a request, the names are the argument identifiers specified in the OMG IDL definition for a specific operation.

As described in Section 19.7, "Mapping for Basic Data Types," on page 19-10, an **any** consists of a **TypeCode** and a pointer to the data value. The TypeCode is a well-known opaque type that can encode a description of any type specifiable in OMG IDL. See this section for a full description of TypeCodes.

For most data types, **len** is the actual number of bytes that the value occupies. For object references, **len** is 1. Table 5-1 shows the length of data values for the C language binding. The behavior of a NamedValue is undefined if the **len** value is inconsistent with the TypeCode.

Table 5-1 C Type Lengths

Data type: X	Length (X)
short	sizeof (CORBA_short)
unsigned short	sizeof (CORBA_unsigned_short)
long	sizeof (CORBA_long)
unsigned long	sizeof (CORBA_unsigned_long)
long long	sizeof (CORBA_long_long)
unsigned long long	sizeof (CORBA_unsigned_long_long)
float	sizeof (CORBA_float)
double	sizeof (CORBA_double)
long double	sizeof (CORBA_long_double)
fixed<d,s>	sizeof (CORBA_fixed_d_s)
char	sizeof (CORBA_char)
wchar	sizeof (CORBA_wchar)
boolean	sizeof (char)
octet	sizeof (CORBA_octet)
string	strlen (string) /* does NOT include '\0' byte! */
wstring	number of wide characters in string, not including wide null terminator
enum E {};	sizeof (CORBA_enum)
union U { };	sizeof (U)
struct S { };	sizeof (S)
Object	1

Table 5-1 C Type Lengths *(Continued)*

Data type: X	Length (X)
array N of type T1	Length (T1) * N
sequence V of type T2	Length (T2) * V /* V is the actual, dynamic, number of elements */

The **arg_modes** field is defined as a bitmask (long) and may contain the following flag values:

CORBA::ARG_IN	The associated value is an input only argument.
CORBA::ARG_OUT	The associated value is an output only argument.
CORBA::ARG_INOUT	The associated value is an in/out argument.

These flag values identify the parameter passing mode for arguments. Additional flag values have specific meanings for request and list routines, and are documented with their associated routines.

All other bits are reserved. The high-order 16 bits are reserved for implementation-specific flags.

5.1.2 Memory Usage

The values for output argument data types that are unbounded strings or unbounded sequences are returned as pointers to dynamically allocated memory. In order to facilitate the freeing of all “out-arg memory,” the request routines provide a mechanism for grouping, or keeping track of, this memory. If so specified, out-arg memory is associated with the argument list passed to the create request routine. When the list is deleted, the associated out-arg memory will automatically be freed.

If the programmer chooses not to associate out-arg memory with an argument list, the programmer is responsible for freeing each out parameter using **`CORBA_free()`**, which is discussed in Section 19.9, “Mapping for Structure Types,” on page 19-12.

5.1.3 Return Status and Exceptions

In the Dynamic Invocation interface, routines typically indicate errors or exceptional conditions either via programming language exception mechanisms, or via an Environment parameter for those languages that do not support exceptions. Thus, the return type of these routines is void.

Previous versions of CORBA allowed implementations to choose the type they returned from these routines by specifying the return type as a typedef named **CORBA::Status**. Implementations were allowed to define this typedef as either type **void** or as **unsigned long**. Due to the portability problems resulting from this approach, the unsigned long definition of **Status** is deprecated. Use of **unsigned long** status, while legal, is not portable.

The **Status** type has been left in the CORBA module for reasons of backwards compatibility. In the next major revision of CORBA it will be removed entirely and all instances of **Status** will be replaced with **void**.

5.2 Request Operations

The request operations are defined in terms of the Request pseudo-object. The Request routines use the **`NVList`** definition defined in the preceding section.

```
module CORBA {
    interface Request {                                          // PIDL

        Status add_arg (
            in Identifier       name,          // argument name
            in TypeCode         arg_type,      // argument datatype
            in void             * value,       // argument value to be added
            in long             len,           // length/count of argument
                                               value
            in Flags            arg_flags      // argument flags
        );
        Status invoke (
             in Flags           invoke_flags   // invocation flags
        );
        Status delete ();
        Status send (
            in Flags            invoke_flags// invocation flags
        );
        Status get_response (
            in Flags            response_flags // response flags
        ) raises (WrongTransaction);
    };
};
```

5.2.1 create_request

Because it creates a pseudo-object, this operation is defined in the Object interface (see “Object Reference Operations” on page 4-5 for the complete interface definition). The **create_request** operation is performed on the Object which is to be invoked.

```
Status create_request (                                    // PIDL
    in Context          ctx,          // context object for operation
    in Identifier       operation,    // intended operation on object
    in NVList           arg_list,     // args to operation
    inout NamedValue    result,       // operation result
    out Request         request,      // newly created request
    in Flags            req_flags     // request flags
);
```

This operation creates an ORB request. The actual invocation occurs by calling **invoke** or by using the **send / get_response** calls.

The operation name specified on **create_request** is the same operation identifier that is specified in the OMG IDL definition for this operation. In the case of attributes, it is the name as constructed following the rules specified in the ServerRequest interface as described in the DSI in “ServerRequestPseudo-Object” on page 6-3.

The **arg_list**, if specified, contains a list of arguments (input, output, and/or input/output) which become associated with the request. If **arg_list** is omitted (specified as NULL), the arguments (if any) must be specified using the **add_arg** call below.

Arguments may be associated with a request by passing in an argument list or by using repetitive calls to **add_arg**. One mechanism or the other may be used for supplying arguments to a given request; a mixture of the two approaches is not supported.

If specified, the **arg_list** becomes associated with the request; until the **invoke** call has completed (or the request has been deleted), the ORB assumes that **arg_list** (and any values it points to) remains unchanged.

When specifying an argument list, the **value** and **len** for each argument must be specified. An argument’s datatype, name, and usage flags (i.e., in, out, inout) may also be specified; if so indicated, arguments are validated for data type, order, name, and usage correctness against the set of arguments expected for the indicated operation.

An implementation of the request services may relax the order constraint (and allow arguments to be specified out of order) by doing ordering based upon argument name.

The context properties associated with the operation are passed to the object implementation. The object implementation may not modify the context information passed to it.

The operation result is placed in the **result** argument after the invocation completes.

The **req_flags** argument is defined as a bitmask (**long**) that may contain the following flag values:

CORBA::OUT_LIST_MEMORY indicates that any out-arg memory is associated with the argument list (**NVList**).

Setting the OUT_LIST_MEMORY flag controls the memory allocation mechanism for out-arg memory (output arguments, for which memory is dynamically allocated). If OUT_LIST_MEMORY is specified, an argument list must also have been specified on

the **create_request** call. When output arguments of this type are allocated, they are associated with the list structure. When the list structure is freed (see below), any associated out-arg memory is also freed.

If OUT_LIST_MEMORY is *not* specified, then each piece of out-arg memory remains available until the programmer explicitly frees it with procedures provided by the language mappings (See Section 19.19, “Argument Passing Considerations,” on page 19-21; Section 20.27, “NVList,” on page 20-71; and Section 22.24, “Argument Passing Considerations,” on page 21-17.

5.2.2 add_arg

```
Status add_arg (                                                    // PIDL
    in Identifier       name,           // argument name
    in TypeCode         arg_type,       // argument datatype
    in void             * value,        // argument value to be added
    in long             len,            // length/count of argument value
    in Flags            arg_flags       // argument flags
);
```

add_arg incrementally adds arguments to the request.

For each argument, minimally its **value** and **len** must be specified. An argument’s data type, name, and usage flags (i.e., in, out, inout) may also be specified. If so indicated, arguments are validated for data type, order, name, and usage correctness against the set of arguments expected for the indicated operation.

An implementation of the request services may relax the order constraint (and allow arguments to be specified out of order) by doing ordering based upon argument name.

The arguments added to the request become associated with the request and are assumed to be unchanged until the invoke has completed (or the request has been deleted).

Arguments may be associated with a request by specifying them on the **create_request** call or by adding them via calls to **add_arg**. Using both methods for specifying arguments, for the same request, is not currently supported.

In addition to the argument modes defined in “Common Data Structures” on page 5-2, **arg_flags** may also take the flag value:IN_COPY_VALUE. The argument passing flags defined in “Common Data Structures” may be used here to indicate the intended parameter passing mode of an argument.

If the IN_COPY_VALUE flag is set, a copy of the argument value is made and used instead. This flag is ignored for inout and out arguments.

5.2.3 invoke

```
Status invoke (                                          // PIDL
    in Flags        invoke_flags        // invocation flags
);
```

This operation calls the ORB, which performs method resolution and invokes an appropriate method. If the method returns successfully, its result is placed in the **result** argument specified on **create_request**. The behavior is undefined if the **Request** pseudo-object has already been used with a previous call to **invoke**, **send**, or **send_multiple_requests**.

5.2.4 delete

```
Status delete ( );                                       // PIDL
```

This operation deletes the request. Any memory associated with the request (i.e., by using the **IN_COPY_VALUE** flag) is also freed.

5.3 Deferred Synchronous Operations

5.3.1 send

```
Status send (                                            // PIDL
    in Flags        invoke_flags        // invocation flags
);
```

send initiates an operation according to the information in the Request. Unlike **invoke, send** returns control to the caller without waiting for the operation to finish. To determine when the operation is done, the caller must use the **get_response** or **get_next_response** operations described below. The out parameters and return value must not be used until the operation is done.

Although it is possible for some standard exceptions to be raised by the **send** operation, there is no guarantee that all possible errors will be detected. For example, if the object reference is not valid, **send** might detect it and raise an exception, or might return before the object reference is validated, in which case the exception will be raised when **get_response** is called.

If the operation is defined to be **oneway** or if INV_NO_RESPONSE is specified, then **get_response** does not need to be called. In such cases, some errors might go unreported, since if they are not detected before **send** returns there is no way to inform the caller of the error.

The following invocation flags are currently defined for **send:**

CORBA::INV_NO_RESPONSE indicates that the invoker does not intend to wait for a response, nor does it expect any of the output arguments (in/out and out) to be updated. This option may be specified even if the operation has not been defined to be **oneway.**

5.3.2 *send_multiple_requests*

```
/* C */
CORBA_Status CORBA_send_multiple_requests (
CORBA_Requestreqs[],          /* array of Requests */
CORBA_Environment*env,
CORBA_longcount,              /* number of Requests */
CORBA_Flagsinvoke_flags
);

// C++

class ORB
{
public:
    typedef sequence<Request_ptr> RequestSeq;
...
    Status send_multiple_requests_oneway(const RequestSeq &);
    Status send_multiple_requests_deferred(const RequestSeq &);
};
```

The Smalltalk mapping of send multiple requests is as follows:

```
sendMultipleRequests: aCollection
sendMultipleRequestOneway: aCollection
```

send_multiple_requests initiates more than one request in parallel. Like **send, send_multiple_requests** returns to the caller without waiting for the operations to finish. To determine when each operation is done, the caller must use the **get_response** or **get_next_response** operations described below.

The degree of parallelism in the initiation and execution of the requests is system dependent. There are no guarantees about the order in which the requests are initiated. If INV_TERM_ON_ERR is specified, and the ORB detects an error initiating one of the requests, it will not initiate any further requests from this list. If INV_NO_RESPONSE is specified, it applies to all of the requests in the list.

The following invocation flags are currently defined for **send_multiple_requests**:

CORBA::INV_NO_RESPONSE indicates that the invoker does not intend to wait for a response, nor does it expect any of the output arguments (inout and out) to be updated. This option may be specified even if the operation has not been defined to be **oneway**.

CORBA::INV_TERM_ON_ERR means that if one of the requests causes an error, the remaining requests are not sent.

5.3.3 get_response

```
Status get_response (                                   // PIDL
    in Flags        response_flags      // response flags
) raises (WrongTransaction);
```

get_response determines whether a request has completed. If **get_response** indicates that the operation is done, the out parameters and return values defined in the Request are valid, and they may be treated as if the Request **invoke** operation had been used to perform the request.

If the RESP_NO_WAIT flag is set, **get_response** returns immediately even if the request is still in progress. Otherwise, **get_response** waits until the request is done before returning.

The following response flag is defined for **get_response:**

CORBA::RESP_NO_WAIT indicates that the caller does not want to wait for a response.

A request has an associated transaction context if the thread originating the request had a non-null transaction context and the target object is a transactional object. The **get_response** operation may raise the **WrongTransaction** exception if the request has an associated transaction context, and the thread invoking **get_response** either has a null transaction context or a non-null transaction context that differs from that of the request.

5.3.4 get_next_response

```
/* C */
CORBA_Status CORBA_get_next_response (
CORBA_Environment*env,
CORBA_Flags     response_flags,
CORBA_Request   *req
);

// C++
class ORB
{
public:
    Boolean poll_next_response();
    Status get_next_response(RequestSeq*&);
};
```

The Smalltalk mapping of get_next_response is as follows:

pollNextResponse
getNextResponse

get_next_response returns the next request that completes. Despite the name, there is no guaranteed ordering among the completed requests, so the order in which they are returned from successive **get_next_response** calls is not necessarily related to the order in which they finish.

If the RESP_NO_WAIT flag is set, and there are no completed requests pending, then **get_next_response** returns immediately. Otherwise, **get_next_response** waits until some request finishes.

The following response flag is defined for **get_next_response:**

CORBA::RESP_NO_WAIT indicates that the caller does not want to wait for a response.

A request has an associated transaction context if the thread originating the request had a non-null transaction context and the target object is a transactional object. The **get_next_response** operation may raise the **WrongTransaction** exception if the request has an associated transaction context, and the thread invoking **get_next_response** has a non-null transaction context that differs from that of the request.

5.4 List Operations

The list operations use the named-value structure defined above.The list operations that create **NVList** objects are defined in the ORB interface described in the ORB Interface chapter, but are described in this section. The **NVList** interface is shown below.

```
interface NVList {                                              // PIDL
    Status add_item (
        in Identifier   item_name,          // name of item
        in TypeCode     item_type,          // item datatype
        in void         *value,             // item value
        in long         value_len,          // length of item value
        in Flags        item_flags          // item flags
    );
    Status free ( );
    Status free_memory ( );
    Status get_count (
        out long        count               // number of entries in the list
    );
};
```

Interface NVList is defined in the CORBA module.

5.4.1 create_list

This operation, which creates a pseudo-object, is defined in the ORB interface and excerpted below.

```
Status create_list (                                        //PIDL
    in long         count,      // number of items to allocate for list
    out NVList      new_list    // newly created list
);
```

This operation allocates a list of the specified size, and clears it for initial use. List items may be added to the list using the **add_item** routine. Alternatively, they may be added by indexing directly into the list structure. A mixture of the two approaches for initializing a list, however, is not supported.

An **NVList** is a partially opaque structure. It may only be allocated via a call to **create_list.**

5.4.2 add_item

```
Status add_item (                                           // PIDL
    in Identifier   item_name,      // name of item
    in TypeCode     item_type,      // item datatype
    in void         *value,         // item value
    in long         value_len,      // length of item value
    in Flags        item_flags      // item flags
);
```

This operation adds a new item to the indicated list. The item is added after the previously added item.

In addition to the argument modes defined in Section 5.1.1, **item_flags** may also take the following flag values: IN_COPY_VALUE, DEPENDENT_LIST. The argument passing flags defined in “Common Data Structures” on page 5-2 may be used here to indicate the intended parameter passing mode of an argument.

If the IN_COPY_VALUE flag is set, a copy of the argument value is made and used instead.

If a list structure is added as an item (e.g., a “sublist”), the DEPENDENT_LIST flag may be specified to indicate that the sublist should be freed when the parent list is freed.

5.4.3 free

```
Status free ( );                                            // PIDL
```

This operation frees the list structure and any associated memory (an implicit call to the list **free_memory** operation is done).

5.4.4 free_memory

```
Status free_memory ( );                                    // PIDL
```

This operation frees any dynamically allocated out-arg memory associated with the list. The list structure itself is not freed.

5.4.5 get_count

```
Status get_count (                                         // PIDL
    out long            count       // number of entries in the list
);
```

This operation returns the total number of items allocated for this list.

5.4.6 create_operation_list

This operation, which creates a pseudo-object, is defined in the ORB interface.

```
Status create_operation_list (                             // PIDL
    in OperationDef     oper,       // operation
    out NVList          new_list    // argument definitions
);
```

This operation returns an **NVList** initialized with the argument descriptions for a given operation. The information is returned in a form that may be used in *Dynamic Invocation* requests. The arguments are returned in the same order as they were defined for the operation.

The list **free** operation is used to free the returned information.

5.5 Context Objects

A context object contains a list of properties, each consisting of a name and a string value associated with that name. By convention, context properties represent information about the client, environment, or circumstances of a request that are inconvenient to pass as parameters.

Context properties can represent a portion of a client's or application's environment that is meant to be propagated to (and made implicitly part of) a server's environment (for example, a window identifier, or user preference information). Once a server has been invoked (i.e., after the properties are propagated), the server may query its context object for these properties.

In addition, the context associated with a particular operation is passed as a distinguished parameter, allowing particular ORBs to take advantage of context properties, for example, using the values of certain properties to influence method binding behavior, server location, or activation policy.

An operation definition may contain a clause specifying those context properties that may be of interest to a particular operation. These context properties comprise the minimum set of properties that will be propagated to the server's environment (although a specified property may have no value associated with it). The ORB may choose to pass more properties than those specified in the operation declaration.

When a context clause is present on an operation declaration, an additional argument is added to the stub and skeleton interfaces. When an operation invocation occurs via either the stub or Dynamic Invocation interface, the ORB causes the properties which were named in the operation definition in OMG IDL and which are present in the client's context object, to be provided in the context object parameter to the invoked method.

Context property names (which are strings) typically have the form of an OMG IDL identifier, or a series of OMG IDL identifiers separated by periods. A context property name pattern is either a property name, or a property name followed by a single "*." Property name patterns are used in the **context** clause of an operation definition and in the **get_values** operation (described below).

A property name pattern without a trailing "*" is said to match only itself. A property name pattern of the form "<name>*" matches any property name that starts with <name> and continues with zero or more additional characters.

Context objects may be created and deleted, and individual context properties may be set and retrieved. There will often be context objects associated with particular processes, users, or other things depending on the operating system, and there may be conventions for having them supplied to calls by default.

It may be possible to keep context information in persistent implementations of context objects, while other implementations may be transient. The creation and modification of persistent context objects, however, is not addressed in this specification.

Context objects may be "chained" together to achieve a particular defaulting behavior.

Properties defined in a particular context object effectively override those properties in the next higher level. This searching behavior may be restricted by specifying the appropriate scope and the "restrict scope" option on the Context **get_values** call.

Context objects may be named for purposes of specifying a starting search scope.

5.6 Context Object Operations

When performing operations on a context object, properties are represented as named value lists. Each property value corresponds to a named value item in the list.

A property name is represented by a string of characters (see "Identifiers" on page 3-6 for the valid set of characters that are allowed). Property names are stored preserving their case, however names cannot differ simply by their case.

The Context interface is shown below.

```
module CORBA {

    interface Context {                                          // PIDL
        Status set_one_value (
            in Identifier    prop_name,     // property name to add
            in string        value          // property value to add
        );
        Status set_values (
            in NVList        values         // property values to be
                                            changed
        );
        Status get_values (
            in Identifier    start_scope,   // search scope
            in Flags         op_flags,      // operation flags
            in Identifier    prop_name,     // name of property(s) to
                                            retrieve
            out NVList       values         // requested property(s)
        );
        Status delete_values (
            in Identifier    prop_name      // name of property(s) to
                                            delete
        );
        Status create_child (
            in Identifier    ctx_name,      // name of context object
            out Context      child_ctx      // newly created context
                                            object
        );
        Status delete (
            in Flags         del_flags      // flags controlling deletion
        );
    };
};
```

5.6.1 get_default_context

This operation, which creates a Context pseudo-object, is defined in the ORB interface (see “Converting Object References to Strings” on page 4-3 for the complete ORB definition).

```
Status get_default_context (                                     // PIDL
    out Context          ctx                // context object
    );
```

This operation returns a reference to the default process context object. The default context object may be chained into other context objects. For example, an ORB implementation may chain the default context object into its User, Group, and System context objects.

5.6.2 set_one_value

```
Status set_one_value (                                          // PIDL
    in Identifier       prop_name,      // property name to add
    in string           value           // property value to add
);
```

This operation sets a single context object property. Currently, only string values are supported by the context object.

5.6.3 set_values

```
Status set_values (                                             // PIDL
    in NVList           values          // property values to be changed
);
```

This operation sets one or more property values in the context object. In the NVList, the flags field must be set to zero, and the TypeCode field associated with an attribute value must be TC_string. Currently, only string values are supported by the context object.

5.6.4 get_values

```
Status get_values (                                             // PIDL
    in Identifier       start_scope,    // search scope
    in Flags            op_flags,       // operation flags
    in Identifier       prop_name,      // name of property(s) to retrieve
    out NVList          values          // requested property(s)
);
```

This operation retrieves the specified context property value(s). If **prop_name** has a trailing wildcard character (“*”), then all matching properties and their values are returned. The values returned may be freed by a call to the list **free** operation.

If no properties are found, an error is returned and no property list is returned.

Scope indicates the context object level at which to initiate the search for the specified properties (e.g., “_USER”, “_SYSTEM”). If the property is not found at the indicated level, the search continues up the context object tree until a match is found or all context objects in the chain have been exhausted.

Valid scope names are implementation-specific.

If scope name is omitted, the search begins with the specified context object. If the specified scope name is not found, an exception is returned.

The following operation flags may be specified:

CORBA::CTX_RESTRICT_SCOPE - Searching is limited to the specified search scope or context object.

5.6.5 delete_values

```
Status delete_values (                                          // PIDL
    in Identifier       prop_name       // name of property(s) to delete
);
```

This operation deletes the specified property value(s) from the context object. If **prop_name** has a trailing wildcard character (“*”), then all property names that match will be deleted.

Search scope is always limited to the specified context object.

If no matching property is found, an exception is returned.

5.6.6 create_child

```
Status create_child (                                           // PIDL
    in Identifier       ctx_name,       // name of context object
    out Context         child_ctx       // newly created context object
);
```

This operation creates a child context object.

The returned context object is chained into its parent context. That is, searches on the child context object will look in the parent context (and so on, up the context tree), if necessary, for matching property names.

Context object names follow the rules for OMG IDL identifiers (see “Identifiers” on page 3-6).

5.6.7 delete

```
Status delete (                                                 // PIDL
    in Flags            del_flags           // flags controlling deletion
);
```

This operation deletes the indicated context object.

The following option flags may be specified:

CORBA::CTX_DELETE_DESCENDENTS deletes the indicated context object and all of its descendent context objects, as well.

An exception is returned if there are one or more child context objects and the CTX_DELETE_DESCENDENTS flag was not set.

5.7 Native Data Manipulation

A future version of this specification will define routines to facilitate the conversion of data between the list layout found in **NVList** structures and the compiler native layout.