MICRO FOCUS

Micro Focus VisiBroker-RT for C++ Version 6.0

Programmer's Reference

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

2020-11-06

Contents

Preface

VisiBroker-RT for C++ allows you to develop and deploy distributed object-based applications, as defined in the Common Object Request Broker (CORBA) specification.

The VisiBroker-RT for C++ *Programmer's Reference Guide* provides a description of the classes and interfaces supplied with VisiBroker-RT for C++, the programmer tools, and command line options. It is written for C++ programmers who are familiar with object-oriented development.

This Preface highlights the latest features, and identifies typographical and platform conventions used throughout the manual. It also tells you where to find additional information about Common Object Request Broker Architecture (CORBA) and the remaining VisiBroker-RT for C++ documentation set, and how to contact Micro Focus support.

What's New

This manual has been updated to reflect the latest VisiBroker-RT for C++ release. The new features and enhancements include:

Does it make sense to describe the below stuff as "new" any more, or drop the section? If kept, need to add anything new in the 2020 release.

- **CORBA 2.5 compliance:** VisiBroker-RT for C++ is fully compliant with the CORBA specification (version 2.5) from the Object Management Group (OMG). For more details, refer to the CORBA specification located at http://www.omg.org.
- **Minimum CORBA 1.0 compliance.** VisiBroker-RT for C++ is fully compliant with the Minimum CORBA specification (version 1.0) from the Object Management Group (OMG). For more details, refer to the Minimum CORBA specification located at ftp:// ftp.omg.org/pub/docs/orbos/98-08-04.pdf
- **Real-Time CORBA 1.0 compliance.** VisiBroker-RT for C++ is fully compliant with the Real-Time CORBA specification (version 1.0) from the Object Management Group (OMG). For more details, refer to the Real-Time CORBA specification located at ftp:// ftp.omg.org/pub/docs/ptc/99-05-03.pdf.
- **Naming Service:** The new VisiBroker-RT for C++ Naming Service. Provides support for the OMG specified Interoperable Naming Service specification. The corbaloc and corbaname functionality supports stringfied object references which can be used in an Internet environment. This allows you to refer to objects by a URL. See Chapter 15, "Using the Naming Service" of the VisiBroker-RT for C++ Programmers guide for a description of how to use the Naming Service.
- **Portable Object Adaptor (POA):** The POA offers portability on the server side. This feature replaces the Basic Object Adapter (BOA). Although BOA is being deprecated, VisiBroker-RT for C++ 6.0 will still support BOA functionality. See Chapter 7, "Using POAs" of the VisiBroker-RT for C++ Programmers guide for an explanation of how to use the POA.
- **Objects by value (OBV) or Value tf types:** Previous versions of CORBA allowed you to pass objects between clients and servers by reference. However, CORBA 2.3 allows you to pass objects by value between clients and servers using VisiBroker-RT for C++. OBV is interoperable with other

2.3-compliant ORBs. See Chapter 25, "Using valuetypes" of the VisiBroker-RT for C++ Programmers guide for more information on this feature.

- **Property Management:** This feature provides you with a way to centralize management of properties. Using the Property Management, you can get/set the value of configurable properties of VisiBroker. See Chapter 11, "Setting properties" of the VisiBroker-RT for C++ Programmers guide for more information on the Property Management.
- **Quality of Service (QoS):** This feature, which implements the CORBA 2.3 Messaging Specification, allows you to define policies that influence how connections are made. You perform client-side policy management by setting properties that are associated with connections or client/server pairs. See "Client basics" on page 9-1 of the VisiBroker-RT for C++ Programmers guide for a description of the QoS features.
- **Interceptors and object wrappers:** The ORB provides a set of APIs known as interceptors which provide a way to plug in additional ORB behavior such as support for transactions and security, which may be defined on either the client or server side. One of the main difference in this release is that now the interceptors have scope. See VisiBroker-RT 6.0 of the VisiBroker-RT for C++ Programmers guide for more information on how to use the 6.0 style interceptors and object wrappers.
- **Pluggable Transport Interface:** This feature provides support for the use of transport protocols besides TCP for the transmission of CORBA invocations. The Interface supports the 'plugging in' of multiple transport protocols simultaneously and is designed to provide a common interface that is suitable for use with a wide variety of transport types. The interface uses CORBA standard classes wherever possible, but is itself VisiBroker proprietary.
- **VisiBroker Logging:** This feature allows applications to log messages and have them directed, via configurable *logging forwarders,* to an appropriate destination or destinations. The ORB itself uses this mechanism for the output of any error, warning or informational messages.

  The application can choose to log its and the ORB's messages to the same destination, producing a single message log for the entire system, or to log messages from different sources to independent destinations.

Organization of this Manual

This manual includes the following sections:

- "Programmer Tools" provides information about the programming tools used to compile C++ stubs and to populate the Interface Repository.
- "IDL to C++ Language Mapping" details the C++ to CORBA mapping specifications, including data types, strings, constant, type definitions, enumerations, and modules.
- "Generated Interfaces and Classes"describes the classes generated by VisiBroker's IDL compiler.
- "Core Interfaces and Classes" describes the VisiBroker-RT for C++ core interfaces and classes.
- "Dynamic Interfaces and Classes" describes the Dynamic Invocation Interface used by clients, and the Dynamic Skeleton Interface used by object servers.

- "Interface Repository Interfaces and Classes" describes the classes and interfaces used to access the Interface Repository.
- "Activation Interfaces and Classes" describes the interfaces and classes used to activate object implementations.
- "Naming Service Interfaces and Classes" describes the interfaces and classes used with the VisiBroker Naming Service.
- "Event Service Interfaces and Classes" describes the interfaces and classes used with the VisiBroker Event Service.
- "Portable Interceptor Interfaces and Classes for C++" describes the VisiBroker-RT for C++ implementation of *Portable Interceptors* interfaces and classes defined by the OMG Specification.
- "4.x Interceptor and Object Wrapper Interfaces and Classes" describes the interfaces and classes that you can use with 4.x *interceptors* and *object wrappers* to create interceptors for client or server-side message processing.
- "Real-Time CORBA Interfaces and Classes" describes the Real-Time CORBA interfaces and classes supported by VisiBroker-RT for C++.
- "Pluggable Transport Interface Classes" describes the classes that constitute the VisiBroker Pluggable Transport Interface. These classes can be used to "plug-in" an application specific transport for ORB communications.
- "VisiBroker Logging Classes"describes the interfaces to use for generating log messages as well as configuring and managing the VisiBroker Log Service.
- "Quality of Service Interfaces and Classes" describes the interfaces you can use to set policy values.
- "IOP and IIOP Interfaces and Classes" describes the CORBA-defined header and message formats.
- "Marshal Buffer Interfaces and Classes" describes the classes and methods for creating and processing message buffers.
- "Location Service Interfaces and Classes" describes how to use the location service to discover objects implemented on your network.
- "Initialization Interfaces and Classes" describes the interfaces and methods for initializing interceptors and other services.
- "Appendix: Using Command-Line Options" explains the ORB, BOA, and location service options that can be passed as command-line arguments when your application is started.
- "Appendix: Using VisiBroker Properties" provides lists of the properties that are available in VisiBroker-RT for C++.

Manual conventions

This section identifies the VisiBroker-RT for C++ *Programmer's Reference Guide*'s typographical and platform conventions.

Typographic conventions

This manual uses the following conventions:

Convention	Used for
Boldface	Bold type indicates that syntax should be typed exactly as shown. For UNIX, used to indicate database names, file names, and similar terms.
italics	Italics indicates information that the user or application provides, such as variables in syntax diagrams. It is also used to introduce new terms.
`computer`	Computer typeface is used for sample command lines and code.
`bold computer`	In code examples, important statements appear in boldface
UPPERCASE	Uppercase letters indicate Windows file names.
[ ]	Brackets indicate optional items.
...	An ellipsis indicates the continuation of previous lines of code or that the previous argument can be repeated.
|	A vertical bar separates two mutually exclusive choices.

Platform conventions

This manual uses the following conventions—where necessary—to indicate that information is platform-specific:

Struck through terms below don't appear in the manual as indicators - delete?

Convention	Used for
Windows	All Windows (Windows NT, Windows 2000, Windows XP) development hosts
~~**WinNT**~~	~~Windows NT development host platform~~
~~**Win2000**~~	~~Windows 2000/XP development host only~~
UNIX	All UNIX development host platforms including Solari
~~**Solaris**~~	~~Solaris development host only~~
Tornado	VisiBroker-RT for C++ for Tornado only
C++	VisiBroker-RT for C++

VisiBroker Library conventions

This manual uses the following conventions—where necessary—to indicate that information is VisiBroker library specific or to indicate that VisiBroker interfaces are not supported in certain versions of the VisiBroker libraries.

This icon indicates functionality that is not supported in the VisiBroker-RT Minimum Corba Library.

Where to find additional information

For more information about VisiBroker-RT for C++, refer to these information sources:

- VisiBroker-RT for C++ *Release Notes* contain late-breaking information about the current release of VisiBroker-RT for C++.
- VisiBroker-RT for C++*Installation Guide*. This guide contains the instructions for installing VisiBroker-RT for C++ on Windows and UNIX host systems as well as information for deploying distributed applications built using VisiBroker-RT for C++.
- VisiBroker-RT for C++ *Developer's Guide* provides information on developing distributed object-based applications in C++.
- For more information about the CORBA specification, refer to *The Common Object Request Broker: Architecture and Specification.* This document is available from the Object Management Group and describes the architectural details of CORBA.

  You can access the CORBA specification at the OMG web site: https://www.omg.org/.

Contacting Micro Focus

Old version has been replaced with the standard MF section.

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The *Product Updates* section of the Micro Focus SupportLine Web site, where you can download fixes and documentation updates.
- The *Examples and Utilities* section of the Micro Focus SupportLine Web site, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page, then click *Support*.

Note:

Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Also, visit:

- The Micro Focus Community Web site, where you can browse the Knowledge Base, read articles and blogs, find demonstration programs and examples, and discuss this product with other users and Micro Focus specialists.
- The Micro Focus YouTube channel for videos related to your product.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the *Product Updates* section of the Micro Focus SupportLine Web site, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page, then click *Support*.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- https://www.microfocus.com/products/corba/visibroker/ (VisiBroker trial software)
- https://supportline.microfocus.com/login.aspx (Micro Focus support login)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

https://software.microfocus.com/en-us/select/email-subscription

Programmer Tools

This chapter describes the programmer tools offered by VisiBroker-RT for C++. For information about the syntax used with these tools, see "Preface".

Arguments/Options

All VisiBroker-RT for C++ programmer's tools have both general and specific arguments. The specific arguments and options for each tool are listed in the section for the tool. The general options are listed below.

General options

The following options are common to all programmer tools:

Option	Description
-J<*java option*>	Passes the *java_option* directly to the Java virtual machine.
-VBJversion	Prints the VisiBroker-RT for C++ version.
-VBJdebug	Prints the VisiBroker-RT for C++ debug information.
-VBJclasspath	Specifies the classpath, precedes the CLASSPATH environment variable.
-VBJprop <*name*> [=<*value*>]	Passes the name/value pair to the java virtual machine.
-VBJjavaavm <*jvmpath*>	Specifies the path of the java virtual machine.
-VBJaddJar <*jarfile*>	Appends the jarfile to the CLASSPATH before executing the java virtual machine.

Note:

On UNIX platforms, the `-J` option is only available with VisiBroker for Java on Solaris.

General Information

The VisiBroker-RT for C++ programming tools described in this chapter differ, depending on whether you have a UNIX or a Windows environment. The UNIX version of each tool is listed first followed by the Windows version.

UNIX

For UNIX users, to view options for a command, enter

Syntax	Example
command name -\?	idl2cpp -\?

Windows

For Windows users, to view options for a command, enter

Syntax	Example
command name -?	idl2cpp -?

idl2cpp

This command implements VisiBroker's IDL to C++ compiler, which generates client stubs and server skeleton code from an IDL file.

Syntax

```
idl2cpp [arguments] infile(s)
```

`idl2cpp` takes an IDL file as input and generates the corresponding C++ classes for the client and server side, client stubs, and server skeleton code.

The infile parameter represents the IDL file for which you wish C++ code to be generated and the arguments provide various controls over the resulting code.

Example

```
idl2cpp -hdr_suffix hx -server_ext _serv -no_tie -no_excep-spec
bank.idl
```

Argument	Description
-D, _define *foo*[=bar]	Defines a preprocessor macro *foo*, optionally with a value *bar*.
-I, -include <*dir*>	Specifies an additional directory for #include searching.
-P, no_line_directives	Suppresses the generation of line number information. The default is off.
-H, _list_includes	Prints the full paths of included files on the standard error output. The default is off.
-C, -retain_comments	Retains comments from IDL file when the C++ code is generated. Otherwise, the comments will not appear in the C++ code.
-U, -undefine *foo*	Undefines a preprocessor macro *foo*.
-{no_]idl_strict	Specifies strict OMG standard interpretation of the IDL source. By default, the OMG standard interpretation is not used
-[no_]warn_ unrecognized_pragmas	Generates a warning if a #pragma is not recognized.
-[no_]back_compat_ mapping	with VisiBroker 3.x.
_[no_]boa	Specifies the generation of BOA compatible code. By default, this code is not generated.
-[no_]comments	Specifies that comments be place in generated code. The default is on.
-gen_include_files	Specifies the generation of code for #include files. By default, this code is not generated.
-list_files	Specifies that files written during code generation be listed. By default, this list is not created.
-[no_]obj_wrapper	Generates stubs and skeletons with object wrapper support. It also generates the base typed object wrapper from which all other object wrappers inherit, and a default object wrapper that performs the untyped object wrapper calls. When this option is not set, idl2cpp does not generate code for object wrappers.
-root_dir <path>	Specifies the directory where the generated code is to be written; the same as setting -hdr_dir and -src_dir to <path>. By default, the code is written to the current directory.
-[no_]servant	Specifies the generation of the server-side code. By default, the servant is generated.
-tie	Generates the _tie template classes. By default, _tie classes are generated.

Argument	Description
-[no_]warn_missing_ define	Warns if any forward declared names were never defined. The default is on.
-client_ext <string>	Specifies the file extension to be used for client files that are generated. The default extension is (_c). To generate client files without an extension, specify **none** as the value for <file_extension>.
-server_ext <string>	Specifies the file extension to be used for server files that are generated. The default extension is (_s). To generate server files without an extension, specify **none** as the value for <file_extension>.
-corba_inc <filename>	Causes the #include <filename> directive to be inserted in generated code instead of the usual #include <corba.h> directive. By default, #include <corba.h> is inserted into generated code.
-excep_spec	Generates exception specifications for methods. By default, exception specifications are not generated.

Windows -export <tag>	Defines a tag name to be inserted into every client-side declaration (class, function, etc.) that is generated. Specifying "-export _MY_TAG" when invoking idl2cpp will result in a class definition like this:
	`class _MY_TAG Bank{...}`
	instead of:
	`class Bank {...}`
	By default, no tag names for client-side declarations are generated.

Windows -export_skel <tag>	Defines a tag name to be inserted into just the server-side declarations that are generated. Specifying "-export _MY_TAG" when invoking idl2cpp will result in a class definition like this:
	`class _MY_TAG _sk_Bank{...}`
	instead of:
	`class _sk_Bank {...}`
	By default, no tag names for server-side declarations are generated.

-hdr_dir <path>	Specifies the directory where the generated include files (_c.hh and _s.hh) are to be written. By default, the code is written to the current directory.
-src_dir <path>	Specifies the directory where the generated source files (_c.cc and _s.cc) are to be written. By default, the code is written to the current directory.
-hrd_suffix <string>	Specifies the header filename extension (.hh)
-src_suffix <string>	Specifies the source filename extension (.cc)
-impl_base_object <C++ type>	Causes the classes in all generated code to be inherited from object_name instead of CORBA::Object. By default, all classes in generated code are inherited from CORBA::Object.
-namespace	Implements modules as namespaces. The default is off.
-pretty_print	Generates -pretty _print methods. By default, all methods will be printed this way.
-stdstream	Generates class stream operators with standard iostream classes in their signature.
-target <compiler>	Specifies the compiler to be used for code generation.
-type_code_info	Enables the generation of type code information needed for client programs that intend to use the Dynamic Invocation Interface. For more information, see "Dynamic Interfaces and Classes". By default, type code information is not generated.

Argument	Description
-version	Displays the software version number of the VisiBroker-RT for C++ idl2cpp compiler.
-imp_inherit	Generates implementation inheritance. The default is off.
-map_keyword <keywrd> <map>	Adds <keywrd> as a keyword and associates with it the mapping indicated. Any IDL identifier that conflicts with <keywrd> will be mapped in C++ to <map>. This prevents clashes between keywords and names used in C++ code. All C++ keywords have default mappings—they do not need to be specified using this option.
-h, -help, -usage, -?	Specifies that help information be printed.
-file1 [file2] …	Specifies one or more files to be processed, or "_" for stdin.

idl2ir

This command allows you to populate an interface repository with objects defined in an Interface Definition Language source file.

Syntax

```
idl2ir [-ir <IR_name>] [-replace] {filename.idl}
```

Example

```
idl2ir -ir my_repository -replace bank/Bank.idl
```

Description

The `idl2ir` command takes an IDL file as input, binds itself to an interface repository server, and populates the repository with the IDL constructs contained in `infile`. If the repository already contains an item with the same name as an item in the IDL file, the old item will be replaced if the `-replace` option is specified.

Note

The `idl2ir` command does not handle anonymous arrays or sequences properly. To work around this problem, `typedefs` must be used for all sequences and arrays.

Option	Description
-D, _define *foo*[=bar]	Defines a preprocessor macro *foo*, optionally with a value *bar*.
-I, -include <*dir*>	Specifies an additional directory for #include searching.
-P, no_line_directives	Suppresses the generation of line number information. The default is off.
-H, _list_includes	Prints the full paths of included files on the standard error output. The default is off.
-C, -retain_comments	Retains comments from IDL file when the C++ code is generated. Otherwise, the comments will not appear in the C++ code.
-U, -undefine *foo*	Undefines a preprocessor macro *foo*.
-{no_]idl_strict	Specifies strict OMG standard interpretation of the IDL source. By default, the OMG standard interpretation is not used
-[no_]warn_ unrecognized_pragmas	Generates a warning if a #pragma is not recognized.
-[no_]back_compat_ mapping	Specifies the use of mapping that is backward compatible with VisiBroker 3.x.

Option	Description
-irep name	Specifies the instance name of the interface repository to which idl2ir will attempt to bind. If no name is specified, idl2ir will bind itself to the interface repository server found in the current domain. The current domain is defined by the OSAGENT_PORT environment variable.
-deep	Specifies deep (versus shallow) merges. The default is off.
-replace	Replaces definitions instead of updating them.
-h, -help, -usage, -?	Prints help information.
-version	Displays the software version number of VisiBroker-RT for C++ idl2ir tool.
file1 [file2] …	Specifies the one or more files to be processed.

ir2idl

This command allows you to populate an interface repository with objects defined in an Interface Definition Language (IDL) source file.

Syntax

```
ir2idl [options] {idl filename}
```

Example

```
ir2idl -ir my_repository -replace bank/Bank.idl
```

Description

The `idl2ir` command binds to the IR and prints the contents in IDL format.

Options

The following options are available for ir2idl.

Option	Description
-irep <*irep name*>	Specifies the name of the interface repository.
-o, <*file*>	Specifies the name of the output file, or "_" for stdout.
-strict	Specifies strict adherence to OMG-standard code generation. The default is on.
-version	Displays or prints out the version of the VisiBroker-RT for C++ ir2idl tool that you are currently running
-h, -help, -usage, -?	Prints help information.

IDL to C++ Language Mapping

This chapter discusses the IDL to C++ language mapping provided by the VisiBroker-RT for C++ idl2cpp compiler, which strictly complies with the CORBA C++ language mapping specification.

Primitive data types

The basic data types provided by the Interface Definition Language are summarized in Table 1, "IDL primitive type mappings". Due to hardware differences between platforms, some of the IDL primitive data types have a definition that is marked "platform dependent." On a platform that has 64-bit integral representations, for example the `long` type, would still be only 32 bits. You should refer to the included file **orbtypes.h** for an exact mapping of these primitive data types for your particular platform.

Table 1 IDL primitive type mappings

IDL type	VisiBroker-RT for C++ type	C++ definition
short	CORBA::Short	short
long	CORBA::Long	platform dependent
unsigned short	CORBA::UShort	unsigned short
unsigned long	CORBA::ULong	unsigned long
float	CORBA::Float	float
double	CORBA::Double	double
char	CORBA::Char	char
boolean	CORBA::Boolean	unsigned char
octet	CORBA::Octet	unsigned char
long long	CORBA::LongLong	platform dependent
ulong long	CORBA::ULongLong	platform dependent

Caution

The IDL boolean type is defined by the CORBA specification to have only one of two values: 1 or 0. Using other values for a boolean will result in undefined behavior.

Strings

String types in IDL may specify a length or may be unbounded, but both are mapped to the C++ type `char *`. You must use the functions shown in Example 1 for dynamically allocating strings to ensure that your applications and VisiBroker use the same memory management facilities. All CORBA string types are null-terminated.

Example 1 Methods for allocating and freeing memory for strings

```
class CORBA
  {
  ...
  static char *string_alloc(CORBA::ULong len); static void
  string_free(char *data);
  ...
  };
```

Method	Description
`CORBA::string_alloc`	Dynamically allocates a string and returns a pointer to the string. A NULL pointer is returned if the allocation fails. The length specified by the len parameter does not need to include the NULL terminator.
`CORBA::string_free`	Releases the memory associated with a string that was allocated with `CORBA::string_alloc`.

String_var Class

In addition to mapping an IDL `string` to a `char *`, the IDL compiler generates a `String_var` class that contains a pointer to the memory allocated to hold the string. When a `String_var` object is destroyed or goes out of scope, the memory allocated to the string is automatically freed. Example 2 shows the `String_var` class and the methods it supports. For more information on the `_var` classes, see "<class_name>_var".

Example 2 `String_var` class

```
class CORBA {
  class String_var {
    protected:
      char* _p;
      ...
    public:
      String_var();
      String_var(char *p);
      ~String_var();
      String_var& operator=(const char *p);
      String_var& operator=(char *p);
      String_var& operator=(const String_var& s);
      operator const char *() const;
      operator char *();
      char &operator[](CORBA::ULong index);
      char operator[](CORBA::ULong index) const;
      friend ostream& operator<<(ostream&, const String_var&);
      inline friend Boolean operator==(const String_var& s1,
          const String_var& s2);
      ...
    };
    ...
};
```

Constants

Example 3 and Example 4 show how IDL constants defined outside of any interface specification will be mapped directly to a C++ constant declaration.

Example 5, Example 6, and Example 7 show how constants defined within an interface specification are declared in the include file and assigned a value in the source file.

Example 3 Top-level definitions in IDL

```
const string str_example = "this is an example";
const long long_example = 100;
const boolean bool_example = TRUE;
```

Example 4 Resulting C++ code for constants

```
const char* str_example = "this is an example";
const CORBA::Long long_example = 100;
const CORBA::Boolean bool_example = 1;
```

Example 5 IDL definitions from the example.idl file

```
interface example {
  const string str_example = "this is an example";
  const long long_example = 100;
  const boolean bool_example = TRUE;
};
```

Example 6 C++ code generated to the example_client.hh file

```
class example :: public virtual CORBA::Object
{
  ...
  static const char *str_example; /* "this is an example" */
  static const CORBA::Long long_example; /* 100 */
  static const CORBA::Boolean bool_example; /* 1 */
  ...
};
```

Example 7 C++ code generated to the example_client.cc file

```
const char *example::str_example = "this is an example";
const CORBA::Long example::long_example = 100;
const CORBA::Boolean example::bool_example = 1;
```

Special cases involving constants

Under some circumstances, the IDL compiler must generate C++ code containing the value of an IDL constant rather than the name of the constant. Example 8 and Example 9 show how the value of the constant `len` must be generated for the typedef `V` to allow the C++ code to compile properly.

Example 8 Definition of an IDL constant with a value

```
// IDL
interface foo {
  const long length = 10;
  typedef long V[length];
};
```

Example 9 Generation of an IDL constant's value in C++

```
class foo : public virtual CORBA::Object
{
  const CORBA::Long length;
  typedef CORBA::Long V[10];
};
```

Enumerations

Example 10 and Example 11 show how enumerations in IDL map directly to C++ enumerations.

Example 10 IDL definition of an enumeration

```
// IDL
enum enum_type {
  first,
  second,
  third
};
```

Example 11 Enumerations in IDL map directly to C++ enums

```
// C++ code
enum enum_type {
  first,
  second,
  third
};
```

Type definitions

Example 12 and Example 13 show how type definitions in IDL are mapped directly to C++ type definitions. If the original IDL type definition maps to several C++ types, the IDL compiler generates the corresponding aliases for each type in C++.

Example 14, Example 15, Example 16, and Example 17 show other type definition mapping examples.

Example 12 Simple type definitions in IDL

```
// IDL
typedef octet example_octet;
typedef enum enum_values {
  first,
  second,
  third
} enum_example;
```

Example 13 Mapping of simple type definitions from IDL to C++

```
// C++
typedef octet example_octet;
enum enum_values {
  first,
  second,
  third
};
typedef enum_values enum_example;
```

Example 14 IDL typedef of an interface

```
// IDL
interface A1;
typedef A1 A2;
```

Example 15 Mapping the IDL interface type definition in C++

```
// C++
class A1;
typedef A1 *A1_ptr;
typedef A1_ptr A1Ref;
class A1_var;

typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1Ref A2Ref;
typedef A1_var A2_var;
```

Example 16 IDL typedef of a sequence

```
// IDL
typedef sequence<long> S1;
typedef S1 S2;
```

Example 17 Mapping the IDL sequence type definition to C++

```
// C++
class S1;
typedef S1 *S1_ptr; typedef S1_ptr S1Ref; class S1_var;

typedef S1 S2;
typedef S1_ptr S2_ptr;
typedef S1Ref S2Ref;
typedef S1_var S2_var;
```

Modules

The OMG IDL to C++ language mapping specifies that an IDL `module` should be mapped to a C++ `namespace` with the same name. Since few compilers currently support the `namespace`, the C++ language mapping allows the use of `class` in its place. Example 19 shows how VisiBroker-RT for C++'s IDL compiler maps `module` to `class`.

Example 18 IDL module definition

```
// IDL module ABC
{
  ...
};
```

Example 19 Mapping an IDL module to a C++ class

```
// C++ class ABC
{
  ...
};
```

Note

For compilers that do support namespaces, use the -namespace option with idl2cpp to generate modules on namespaces.

Complex data types

The C++ mappings for IDL structures, unions, sequences, and arrays depend on whether or not the data members they contain are of a fixed or variable length. These types are considered to have variable lengths. As a result, any complex data type that contains a `structure`, `union`, `sequence`, or `array` will also have a variable length.

Complex data types include:

- `Any` type
- `string` type, bounded or unbounded
- `sequence` type, bounded or unbounded
- Object reference
- Other `structures` or `unions` that contain a variable-length member
- `array` with variable-length elements
- `typedef` with variable-length elements.

Table 2 Summary of C++ mappings for complex data types

IDL type	C++ mapping
`struct` (fixed length)	`struct` and `_var` class
`struct` (variable length)	`struct` and `_var` class (variable length members are declared with their respective `T_var` class)
`union`	`class` and `_var` class
`sequence`	`class` and `_var` class
`array`	`array`, `array_slice`, `array_forany`, and `array_var`

Fixed-length structures

Example 20 and Example 21 show how fixed-length structures in IDL are mapped to C++ code. In addition to the structure, VisiBroker-RT for C++'s IDL compiler will also generate an `example_var` class for the structure. For more information on the `_var` class, see "<class_name>_var".

Example 20 Fixed-length structure definition in IDL

```
// IDL
struct example {
  short a;
  long b;
};
```

Example 21 Mapping a fixed-length IDL structure to C++

```
// C++
struct example {
  CORBA::Short a;
  CORBA::Long b;
};

class example_var
{
  ...
  private:
    example *_ptr;
};
```

Using fixed-length structures

Example 22 shows that to access the fields of the `_var` class `ex2`, the `->` operator must always be used. When `ex2` goes out of scope, the memory allocated to it will be freed automatically.

Example 22 Use of the example structure and the example_var class

```
// Declare an example struct and initialize its fields.
example ex1 = { 2, 5 };

// Declare a _var class and assign it to a newly created
// example structure.
// The _var points to an allocated struct with un-initialized
// fields.
example_var ex2 = new example;

// Initialize the fields of ex2 from ex1
ex2->a = ex1.b;
```

Variable length structures

Example 23 and Example 24 show how you could modify the `example` structure, replacing the `long` member with a `string` and adding an object reference, to change to a variable-length structure.

Example 23 Variable length structure definitions in IDL

```
// IDL
interface ABC {
  ...
};
struct vexample {
  short a;
  ABC c;
  string name;
};
```

Example 24 Mapping a variable-length structure to C++

```
// C++
struct vexample {
  CORBA::Short a;
  ABC_var c;
  CORBA::String_var name;
  (vexample& operator=(const vexample& s);
};

class vexample_var {
  ...
};
```

Notice how the `ABC` object reference is mapped to an `ABC_var` class. In a similar fashion, the string `name` is mapped to a `CORBA::String_var` class. In addition, an assignment operator is also generated for variable-length structures.

Memory management for structures

The use of `_var` classes in variable-length structures ensures that memory allocated to the variable-length members are managed transparently.

- If a structure goes out of scope, all memory associated with variable-length members is automatically freed.
- If a structure is initialized or assigned and then re-initialized or reassigned, the memory associated with the original data is always freed.
- When a variable-length member is assigned to an object reference, a copy is always made of the object reference. If a variable-length member is assigned to a pointer, no copying takes place.

Unions

Example 26 show how an IDL `union` is mapped to a C++ class with methods for setting and retrieving the value of the data members. A data member, `named _d`, of the discriminant type is also defined. The value of this discriminant is not set when the union is first created, so an application must set it before using the `union`. Setting any data member using one of the provided methods automatically sets the discriminant.

Table 3 describes some of the methods in the `un_ex` class.

Table 3 Methods generated for the `un_ex` class

Method	Description
`un_ex()`	The default constructor sets the discriminant to the default value but does not initialize any of the other data members.
`un_ex(const un_ex& obj)`	The copy constructor performs a deep copy of the source object.

Method	Description
~un_ex()	The destructor frees all memory owned by the union.
operator=(const un_ex& obj)	The assignment operator performs a deep copy, releasing old storage, if necessary.

Example 25 IDL union containing a struct

```
// IDL
struct st_ex
{
  long abc;
};
union un_ex switch(long)
{
  case 1: long x; // a primitive data type
  case 2: string y; // a simple data type
  case 3: st_ex z; // a complex data type
};
```

Example 26 Mapping an IDL union to a C++ class

```
// C++ struct st_ex
{
    CORBA::Long abc;
};

class un_ex
{
  private:
    CORBA::Long _disc;
    CORBA::Long _x;
    CORBA::String_var _y;
    st_ex _z;
  public:
    un_ex();
    ~un_ex();
    un_ex(const un_ex& obj);
    un_ex& operator=(const un_ex& obj);
    void x(const CORBA::Long val);
    const CORBA::Long x() const;
    void y(char *val);
    void y(const char *val);
    void y(const CORBA::String_var& val);
    const char *y() const;
    void z(const st_ex& val);
    const st_ex& z() const;
    st_ex& z();
    CORBA::Long  _d();
    void _d(CORBA::Long);
    ...
  };
```

Managed types for unions

In addition to the `un_ex` class shown in Example 26, an `un_ex_var` class would also be generated. See “<class_name>_var” for details on the `_var` classes.

Memory management for unions

Here are some important points to remember about memory management of complex data types within a union:

- When you use an accessor method to set the value of a data member, a deep copy is performed. You should pass parameters to accessor methods by value for smaller types, or by a constant reference for larger types.
- When you set a data member using an accessor method, any memory previously associated with that member is freed. If the member being

assigned is an object reference, the reference count of that object will be incremented before the accessor method returns.

- A `char *` accessor method will free any storage before ownership of the passed pointer is assumed.
- Both `const char *` and `String_var` accessor methods will free any old memory before the new parameter's storage is copied.
- Accessor methods for array data members will return a pointer to the array slice. For more information, see "Array slices".

Sequences

IDL sequences, both bounded and unbounded, are mapped to a C++ class that has a current length and a maximum length. The maximum length of a bounded sequence is defined by the sequence's type. Unbounded sequences can specify their maximum length when their C++ constructor is called. The current length can be modified programmatically. Example 27 and Example 28 show how an IDL sequence is mapped to a C++ class with accessor methods.

Note

When the length of an unbounded sequence exceeds the maximum length you specify, VisiBroker-RT for C++ will transparently allocate a larger buffer, copy the old buffer to the new buffer, and free the memory allocated to the old buffer. No attempt will be made, however, to free any unused memory if the maximum length decreases.

Example 27 IDL unbounded sequence

```
// IDL
typedef sequence<long> LongSeq;
```

Example 28 Mapping an IDL unbounded sequence to a C++ class

```
// C++
class LongSeq
{
  public:
    LongSeq(CORBA::ULong max=0);
    LongSeq(CORBA::ULong max=0, CORBA::ULong length,
            CORBA::Long *data, CORBA::Boolean release = 0);
    LongSeq(const LongSeq&);
    ~LongSe q();
    LongSeq& operator=(const LongSeq&);
    CORBA::ULong maximum() const;
    void   length(CORBA::ULong len);
    CORBA::ULong length() const;
    const CORBA::ULong& operator[](CORBA::ULong index) const;
    ...
    static LongSeq *_duplicate(LongSeq* ptr);
    static void _release(LongSeq *ptr);
    static CORBA::Long *allocbuf(CORBA::ULong nelems);
    static void freebuf(CORBA::Long *data);

  private:
    CORBA::Long* _contents;
    CORBA::ULong _count;
    CORBA::ULong _num_allocated;
    CORBA::Boolean _release_flag;
    CORBA::Long _ref_count;
};
```

Table 4 Synopsis of methods generated for the unbounded sequence in Example 28, Mapping an IDL unbounded sequence to a C++ class

Method	Description
`LongSeq(CORBA::ULong max=0)`	The constructor for an unbounded sequence takes a maximum length as an argument. Bounded sequences have a defined maximum length.
`LongSeq(CORBA::ULong max=0, CORBA::ULong length, CORBA::Long *data, CORBA::Boolean release=0)`	This constructor allows you to set the maximum length, the current length, a pointer to the data buffer associated and a `release` flag. If `release` is not zero, VisiBroker will free memory associated with the data buffer when increasing the size of the sequence. If release is zero, the old data buffer's memory is not freed. Bounded sequences have all of these parameters except for max.
`LongSeq(const LongSeq&)`	The copy constructor performs a deep copy of the source object.
`~LongSeq();`	The destructor frees all memory owned by the sequence only if the release flag had a non-zero value when constructed.
`operator=(const LongSeq&j)`	The assignment operator performs a deep copy, releasing old storage, if necessary.
`maximum()`	Returns the size of the sequence.
`length()`	Two methods are defined for setting and returning the length of the sequence.
`operator[]()`	Two indexing operators are provided for accessing an element within a sequence. One operator allows the element to be modified and one allows only read access to the element.
`_release()`	Releases the sequence. If the constructor's release flag was non-zero when the object was created and the sequence element type is a string or object reference, each element will be released before the buffer is released.
`allocbuf() freebuf()`	You should use these two static methods to allocate or free any memory used by a sequence

Managed types for sequences

In addition to the `LongSeq` class shown in Example 28, a LongSeq_var class would also be generated. See "<class_name>_var" for details on the classes. In addition to the usual methods, there are two indexing methods defined for sequences.

Example 29 Two indexing methods added for _var classes representing sequences

```
CORBA::Long& operator[](CORBA::ULong index);
const CORBA::Long& operator[](CORBA::ULong idex) const;
```

Memory management for sequences

You should carefully consider the memory management issues listed below. Example 31 contains sample C++ code that illustrates these points.

- If the release flag was set to a non-zero value when the sequence was created, the sequence will assume management of the user's memory. When an element is assigned, the old memory is freed before ownership of the memory on the right-hand side of the expression is assumed.
- If the release flag was set to a non-zero value when a sequence containing strings or object references was created, each element will be released before the sequence's contents buffer is released and the object is destroyed.
- Avoid assigning a sequence element using the `[]` operator unless the release flag was set to one, or memory management errors may occur.
- Sequences created with the release flag set to zero should not be used as input/output parameters because memory management errors in the object server may result.
- Always use `allocbuf` and `freebuf` to create and free storage used with sequences.

Example 30 IDL specification for an unbounded sequence

```
// IDL
typedef sequence<string, 3> String_seq;
```

Example 31 Example of memory management with two bounded sequences

```
// C++
char *static_array[] = ("1", "2", "3"};
char *dynamic_array = StringSeq::allocbuf(3);

// Create a sequence, release flag is set to FALSE by default
StringSeq static_seq(3, static_array);

// Create another sequence, release flag set to TRUE
StringSeq dynamic_seq(3, dynamic_array, 1);

static_seq[1] = "1";// old memory not freed, no copying occurs

char *str = string_alloc(2);
dynamic_seq[1] = str;// old memory is freed, no copying occurs
```

Arrays

IDL arrays are mapped to C++ arrays, which can be statically initialized. If the array elements are strings or object references, the elements of the C++ array will be of the type `_var`. Example 32 and Example 33 show three arrays with different element types.

Example 32 IDL array definitions

```
// IDL interface Intf
{
  ...
};
typedef long L[10];
typedef string S[10];
typedef Intf A[10];
```

Example 33 Mapping IDL arrays to C++ arrays

```
// C++
typedef CORBA::Long L[10];
typedef CORBA::String_var S[10];
typedef Intf_var A[10];
```

The use of the managed type _var for strings and object references allows memory to be managed transparently when array elements are assigned.

Array slices

The array_slice type is used when passing parameters for multi-dimensional arrays. VisiBroker's IDL compiler also generates a _slice type for arrays that contains all but the first dimension of the array. The array _slice type provides a convenient way to pass and return parameters. Example 34 and Example 35 show two examples of the _slice type.

Example 34 IDL definition of multi-dimensional arrays

```
// IDL
typedef long L[10];
typedef string str[1][2][3];
```

Example 35 Generation of the _slice type

```
// C++
typedef CORBA::Long L_slice;
typedef CORBA::String_var str_slice[2][3];
```

Managed types for arrays

In addition to generating a C++ array for IDL arrays, VisiBroker's IDL compiler will also generate a _var class. This class offers some additional features for array.

- operator[] is overloaded to provide intuitive access to array elements.
- Constructor and assignment operator are provided that take a pointer to an array _slice object as an argument.

Example 36 IDL definition of an array

```
// IDL
typedef long L[10];
```

Example 37 _var class generated for arrays

```
// C++ class L_var
{
  public:
    L_var();
    L_var(L_slice *slice);
    L_var(const L_var& var);
    ~L_var();
    L_var& operator=(L_slice *slice);
    L_var& operator=(const L_var& var);
    CORBA::Long& operator[](CORBA::ULong index);
    operator L_slice *();
    operator L &() const;
    ...
  private:
    L_slice*_ptr;
};
```

Type-safe arrays

A special _forany class is generated to handle arrays with elements mapped to the type any. As with the _var class, the _forany class allows you to access the underlying array type. The _forany class does not release any memory upon destruction because the

_any type maintains ownership of the memory. The _forany class is not implemented as a typedef because it must be distinguishable from other types for overloading to function properly.

Example 38 IDL array definition

```
// IDL
typedef long L[10];
```

Example 39 _for any class generated for an IDL array

```
// C++
class L_forany
{
  public:
    L_forany();
    L_forany(L_slice *slice);
    ~L_forany();
    CORBA::Long& operator[](CORBA::ULong index);
    const CORBA::Long& operator[](CORBA::ULong index) const;
    operator L_slice *();
    operator L &() const;
    operator const L & () const;
    operator const L& () const;
    L_forany& operator=(const L_forany obj);
    ...
  private:
    L_slice*_ptr;
};
```

Memory management for arrays

VisiBroker's IDL compiler generates four functions for allocating, duplicating, copying, and releasing the memory associated with arrays. These functions allow the ORB to manage memory without having to override the `new` and `delete` operators.

Example 40 IDL array definition

```
// IDL
typedef long L[10];
```

Example 41 Methods generated for allocating and releasing array memory

```
// C++
inline L_slice *L_alloc();// Dynamically allocates array.
  // Returns NULL on failure.
inline void L_free(L_slice *data);// Releases array memory
  // allocated with L_alloc.
inline void L_copy(L:slice *_to, L_slice *_from)
  //Copies the contents of the
  //_from array to the _to array
  //inline L_slice *L_dup
  //(const L_slice *_date)
  //Returns a new copy of _date
//array
```

Principal

A `Principal` represents information about client applications that are making operation requests on an `object` implementation. The IDL interface of `Principal` does not define any operations. The `Principal` is implemented as a sequence of octets. The `Principal` is set by the client application and checked by the ORB implementation. VisiBroker-RT for C++ treats the `Principal` as an opaque type and its contents are never examined by the ORB.

Valuetypes

An IDL valuetype is mapped to a C++ class with the same name as the IDL valuetype. This class is an abstract base class with pure virtual accessor and modifier functions corresponding to the state members of the valuetype and pure virtual functions corresponding to the operations of valuetype.

A C++ class whose name is formed by adding an "OBV_" to the fully scoped name of the valuetype provides default implementations for the accessors and modifiers of the abstract base class.

Applications are responsible for the creation of valuetype instances. After creation, these applications deal with those instances using only pointers. Unlike object references which map to C++ _ptr types that may be implemented either as actual C++ pointers or as C++ pointer-like objects, handles to C++ valuetype instances are actual C++ pointers. This helps to distinguish them from object references.

Unlike mapping for interfaces, the reference counting for valuetype must be implemented by the instance of the valuetypes. The _var type for a valuetype automates the reference counting. Example 42 illustrates these features.

Example 42 _var type for a valuetype to automate reference counting

```
Valuetype Example { Short op1();
Long op2( in Example x ); Private short val1;
Public long val2;
};
```

Example 43 shows the C++ mapping of the IDL definition for the following three classes.

Example 43 C++ mapping for the IDL definitions

```
class Example : public virtual CORBA::ValueBase {
public:
  virtual CORBA::Short op1() = 0;
  virtual CORBA::Long op2(Example_ptr _x) = 0;
// pure virtual getter/setters for all public state
// These accessors are just like C++ union members since
// by reference accessors allow read/write access
virtual void val2(const CORBA::Long _val2) = 0;
virtual const CORBA::Long val2() const = 0;

protected:
Example() {}
virtual ~Example() {}
virtual void val1(const CORBA::Short _val1) = 0;
virtual const CORBA::Short val1() const = 0;

  private:
void operator=(const Example&);
};
class OBV_Example: public virtual Example{

public:
virtual void val2(const CORBA::Long _val2) {
_obv_val2 = _val2;
}
virtual const CORBA::Long val2() const {
return _obv_val2;
}

protected:
virtual void val1(const CORBA::Short _val1) {
_obv_val1 = _val1;
}
virtual const CORBA::Short val1() const {
  return _obv_val1; }
OBV_Example() {}
virtual ~OBV_Example() {}
OBV_Example(const CORBA::Short _val1,
    const CORBA::Long _val2) {
    _obv_val1 = _val1;
    _obv_val2 = _val2;
}
CORBA::Short _obv_val1;
CORBA::Long _obv_val2;
```

```
};

class Example_init : public CORBA::ValueFactoryBase {
};
```

The `_init` class is a provision for implementing a Factory for the Valuetypes. Since valuetypes are passed by value over the wire, the receiving end of a streamed out valuetype usually implements a factory to create a valuetype instance from the stream. Both the Server and the Client should implement it if there is a possibility of receiving a valuetype over the stream. The `_init` class, as shown in Example 44, must also implement `create_for_unmarshal` that returns a `CORBA::ValueBase *`.

Example 44 -init class example

```
class Example_init_impl: public Example_init{
public:
   Example_init; _impl();
  virtual ~Example_init();
CORBA::ValueBase * create_for_unmarshal() {
...// return an Example_ptr
  }
};
```

A valuetype can derive from other valuetypes as follows:

Example 45 IDL for the valuetype derived from other valuetypes

```
Valuetype DerivedExample: Example{
Short op3();
};
```

The C++ interfaces for the `DerivedExample` class are as follows:

Example 46 C++ generate for the derived valuetype

```
// idl valuetype: DerivedExample
class DerivedExample : public virtual Example {
public:
  virtual CORBA::Short op3() = 0;
protected:
  DerivedExample() {}
  virtual ~DerivedExample() {}
private:
  void operator=(const DerivedExample&);
};
class OBV_DerivedExample: public virtual DerivedExample,
      public virtual OBV_Example{
protected:
  OBV_DerivedExample() {}
  virtual ~OBV_DerivedExample() {}
};
class DerivedExample_init : public CORBA::ValueFactoryBase {
};
```

A derived valuetype can be truncated to the base valuetype as shown in Example 47. This is required if the receiving end of the stream does not know how to construct a derived valuetype but can construct only the base valuetype.

Example 47 truncated derived valuetype

```
valuetype DerivedExample : truncatable Example {
};
```

The mapping is similar to regular derived valuetypes except that extra information is added to the `Type` information of the `DerivedExample` class to indicate the truncatability to the base class Example.

A valuetype can not derive from an interface but it can support one or more interfaces by providing all the operations of the interfaces. An IDL keyword, **supports,** is introduced for this purpose.

Example 48 IDL keyword support for the derived valuetype

```
interface myInterface{
   long op5();
};

valuetype IderivedExample supports myInterface {
  Short op6();
};
```

The C++ mapping for this will be as follows:

Example 49 C++ for the derived valuetype

```
// idl valuetype: DerivedExample
class IderivedExample : public virtual CORBA::ValueBase {
public:
  virtual CORBA::Short op6() = 0;
  virtual CORBA::Long op5() = 0;

protected:
  IderivedExample() {}
  virtual ~IderivedExample() {}
private:
  void operator=(const IderivedExample&);
};

class OBV_IderivedExample: public virtual IderivedExample{
protected:
  OBV_IderivedExample() {}
  virtual ~OBV_IderivedExample() {}
};
```

For reference counting, the C++ mapping provides two standard classes. The first class is `CORBA::DefaultValueRefCountBase`, which serves as a base class for any application provided concrete valuetypes that do not derive from any IDL interfaces. For these kinds of valuetypes, the applications are also free to implement their own reference counting mechanisms. The second class is `PortableServer::ValueRefCountBase`, which must serve as a base class for any application provided a concrete valuetype class which does derive from one or more IDL interfaces.

Valuebox

A valuebox is a valuetype applied to structures, unions, any, string, basic types, object references, enums, sequence, and array types. These types do not support method, inheritance, or interfaces. A valuebox is ref counted and is derived from `CORBA::DefaultValueRefCountBase`. The mapping is different for different underlying types. All valuebox C++ classes provide `_boxed_in()`, `boxed_out()`, and `_boxed_inout()` for mapping to the underlying types. The factory for a valuebox id automatically registered by the generated stub.

See the *OMG CORBA 2.3 idl2cpp specification*, Chapter 1.17, for more information. The factory for a valuebox is automatically registered by the generated stub.

Abstract Interfaces

Abstract interfaces are used to determine at runtime, if an object is passed by reference (IOR) or by value (valuetype.) A prefix “abstract” is used for this purpose before an interface declaration.

Example 50 IDL code sample

```
    Abstract interface foo {
    Void func():

}
```

A valuetype that supports an abstract interface, can be passed as that abstract interface. The abstract interface is declared as follows:

Example 51 Valuetype as the abstract interface

```
    Valuetype vt supports foo {
    ...
};
```

Similarly, an interface that needs to be passed as an abstract interface is declared as follows:

Example 52 Interface as the abstract interface

```
interface intf : foo {
}
```

The c++ mapping for the previously declared abstract interface foo, results in the following classes:

Example 53 C++ mapping of the abstract interface

```
class foo_var : public CORBA::_var{
  ...
}
class foo_out{
  ...
};
class foo : public virtual CORBA:::AbstractBase{
  private;
  ...
    void operator=(const foo&) {}
  protected:
    foo();
    foo(const foo& ref) {} virtual ~foo() {}
  public:
    static CORBA::Object* _factory():
    foo_ptr _this();
    static foo_ptr _nil() { ... }
    static foo_ptr _narrow(CORBA::AbstractBase* _obj);
    static foo_ptr _narrow(CORBA::Object_ptr _obj);
    static foo_ptr _narrow(CORBA::ValueBase_ptr _obj);

    virtual void func() = 0;

...
};
class _vis_foo_stub : public virtual foo, public virtual
CORBA_Object {
  public :
  _vis_foo_stub() {}
  virtual ~_vis_foo_stub() {}
  ...
  virtual void func():

  ...
}
```

There is a `_var` class, an `_out` class, and a class derived from `CORBA::AbstractBase` that implements the methods described in the previous code samples. Example 53 also includes three _narrow methods that can be used to narrow an object, abstract interface, or a valuetype to the declared abstract interface type. Example 53 also includes a dummy stub stub generated for the *foo* class. This stub accommodates cases where full type information is not available. See the *CORBA 2.3 IDL2CPP specification, section 1.18, Mapping for Abstract Interfaces* for more information on this topic.

Generated Interfaces and Classes

This chapter describes classes generated by VisiBroker-RT for C++'s IDL compiler, their uses, and their features.

Overview

VisiBroker-RT for C++'s IDL compiler can generate a variety of classes that makes it easier for you to develop client applications and object servers. Many of these generated classes are available for CORBA classes.

- stub classes
- servant classes
- tie
- classes
- var classes

<Interface_name>

`class <interface_name>`

The `<interface_name>` class is generated for a particular IDL interface and, is intended for use by client applications. This class provides all of the methods defined for a particular IDL interface. When a client uses an object reference to invoke methods on the object, the stub methods are actually invoked. The stub methods allow a client operation request to be packaged, sent to the object implementation, and the results to be reflected. This entire process is transparent to the client application.

Note

You should never modify the contents of a stub class generated by the IDL compiler.

<Interface_name>ObjectWrapper

This class is used to derive typed object wrappers and is generated for all your interfaces when you invoke the `idl2cpp` command with the `-obj_wrapper` option, as described in "`-obj_wrapper`" on page 2-1. For complete details on using the object wrapper feature, see the VisiBroker-RT for C++ *Programmer's Guide*.

```
static void add(CORBA::ORB_ptr orb, CORBA::ObjectFactory factory,
  VISObjectWrapper::Location loc);
```

Adds a typed object wrapper from a client application. If more than one typed object wrapper is installed, they will be invoked in the order in which they were registered.

Parameter	Description
`orb`	The `ORB` the client wishes to use, returned by the `ORB_init` method.

Parameter	Description
factory	The factory method for the object wrapper class that you want to add.
loc	The location of the object wrapper being added, which should be one of the following values: VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both

```
static void remove(CORBA::ORB_ptr orb, CORBA::ObjectFactory
factory, VISObjectWrapper::Location loc);
```

Removes an un-typed object wrapper from a server application.

Parameter	Description
orb	The ORB the client wishes to use, returned by the ORB_init method.
factory	The factory method for the object wrapper class that you want to remove.
loc	The location of the object wrapper being removed, which should be one of the following values: VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both

POA<class_name>

```
class _POA_<class_name>
```

The _POA_<class_name> class is an abstract base class generated by the IDL compiler, which is used to derive an object implementation class. Object implementations are usually derived from a servant class, which provides the necessary methods for receiving and interpreting client operation requests.

The previous _sk_<class_name> is only generated if you use idl2cpp -boa.

tie<class_name>

```
class _tie_<class_name>
```

The _tie_<class_name> class is generated by the IDL compiler to aid in the creation of delegation implementations. The tie class allows you to create an object implementation that delegates all operation requests to another object. This allows you to use existing objects that you do not wish to inherit from the CORBA::Object class.

<class_name>_var

```
class <class_name>_var
```

The <class_name>_var class is generated for an IDL interface and provides simplified memory management semantics.

Core Interfaces and Classes

This chapter describes the VisiBroker for C++ core interfaces and classes.

PortableServer::AdapterActivator

Adapter activators are associated with Portable Object Adapters (POAs) which they supply with the ability to create child POAs on demand, as a side-effect of receiving a request which names the child POA (or one of its children), or when the `find_POA` method is called with an activate parameter set to `TRUE`.

PortableServer::AdapterActivator methods

```
CORBA::Boolean unknown_adapter(POA_ptr parent, const char*
  name);
```

This method is called when the ORB receives a request for an object reference which identifies a target POA that does not exist. The ORB invokes this method once for each POA that must be created in order for the POA to exist (starting with the ancestor POA closest to the root POA).

Parameter	Description
`parent`	The parent POA associated with the adapter activator on which the method is to be invoked.
`name`	The name of the POA to be created (relative to the parent).

BindOptions

Note

This structure is deprecated since VisiBroker 4.0.

```
struct BindOptions
```

This structure is used to specify options to the `_bind` method, described in the section "Object". Each ORB instance has a global `BindOptions` structure that is used for all `_bind` invocations that do not specify bind options. You can modify the default bind options using the `Object::_default_bind_options` method.

Bind options may also be set for a particular object and will remain in effect for the lifetime of the connection to that object.

Include file

The **corba.h** file should be included when you use this structure.

BindOptions members

`CORBA::Boolean` **`defer_bind;`**

If set to `TRUE`, the establishment of the connection between client and the object implementation will be delayed until the first client operation is issued. If set to 3`FALSE`, the `_bind` method will establish the connection immediately.

`CORBA::Boolean` **`enable_rebind;`**

If set to `TRUE` and the connection is lost, due to a network failure or some other error, the ORB will attempt to re-establish a connection to a suitable object implementation. If set to `FALSE`, no attempt will be made to reconnect the client with the object implementation.

`CORBA::Long` **`max_bind_tries;`**

This member has been disabled and is no longer used by the ORB. Setting this field has **no** effect on the _bind behavior.

`CORBA::ULong` **`send_timeout;`**

This member specifies the maximum time in seconds that a client is to block waiting to send an operation request. If the request times out, `CORBA::NO_RESPONSE` exception will be raised and the connection to the server will be destroyed. The default value of 0 implies the client should block indefinitely.

`CORBA::ULong` **`receive_timeout;`**

This member specifies the maximum time in seconds that a client is to block waiting for a response to an operation request. If the request times out, `CORBA::NO_RESPONSE` exception will be raised and the connection to the server will be destroyed. The default value of 0 implies the client should block indefinitely.

`CORBA::ULong` **`connection_timeout;`**

This member specifies the maximum time in seconds that a client is to wait for a connection. If the time specified is exceeded, a `CORBA::NO_IMPLEMENT` exception is raised. The default value of 0 implies that the default system time-out for connections should be used.

BOA

Note

This class is deprecated since VisiBroker 4.0

`class` **`BOA`**

The `BOA` class represents the Basic Object Adaptor and provides methods for creating and manipulating objects and object references. Object servers use the BOA to activate and deactivate object implementations and to specify the thread policy they wish to use.

You do not instantiate a `BOA` object. Instead, you obtain a reference to a `BOA` object by invoking the `ORB::BOA_init` method, described on page 5-17.

VisiBroker-RT for C++ provides extensions to the CORBA BOA specification which are covered in "VisiBroker extensions to CORBA::BOA" on page 5-7. These methods provide for the management of connections, threads, and the activation of services.

Include file

The **corba.h** file should be included when you use this class.

CORBA::BOA methods

```
void change_implementation(const
  extension::CreationImplDef& _old_info, const
  extension::CreationImplDef& _new_info)
```

This method changes the implementation definition associated with the specified object. You should use this method with caution. The implementation name should not be changed and you must ensure that the new implementation definition specifies the same type of object as the original definition. If the `ImplementationDef_ptr` does not point to a `CreationImplDef` pointer, this method will fail.

Parameter	Description
`Object_ptr`	A pointer to the object whose implementation is to be changed.
`impl`	A pointer to the new implementation definition for this object. This must actually be a `CreationImplDef_ptr` cast to an `ImplementationDef_ptr`.

```
CORBA::Object_ptr create(const CORBA::ReferenceData&,
  extension::CreationImplDef&)
```

This method registers the specified implementation with the OAD.

Note

Since the OAD is not supported in VisiBroker-RT for C++ this method always returns `CORBA::Object::_nil()`.

Parameter	Description
`ReferenceData`	This parameter is not used, but is provided for compliance with the CORBA specification.
`CreationImplDef`	This pointer's true type is CreationImplDef. It provides the interface name, object name, path name of the executable and the activation policy and other parameters.

```
void deactivate_impl(extension::ImplementationDef_ptr)
```

This method causes requests to the implementation to be discarded. The method deactivates the implementation specified by the `ImplementationDef_ptr`. Once this method is called, no further client requests are delivered to the object within this implementation until the objects and implementation are re-activated. Calling the `impl_is_ready` or

`obj_is_ready` methods causes the implementation to again accept requests.

Parameter	Description
`ImplementationDef_ptr`	This pointer's true type is CreationImplDef and provides the interface name, object name, path name of the executable and activation policy, along with other parameters.

`void` **`deactivate_obj`**`(CORBA::Object_ptr)`

This method requests the `BOA` to deactivate the specified object. Once this method is invoked, the `BOA` does not deliver any requests to the object until `obj_is_ready` or `impl_is_ready` is invoked.

Parameter	Description
`Object_ptr`	A pointer to the object to be deactivated.

`void` **`dispose`**`(CORBA::Object_ptr)`

This method unregisters the implementation of the specified object from the Object Activation Daemon. Once this method is invoked, all references to the specified object are invalid and any connections to this object implementation are broken. If the object has been allocated, it is the application's responsibility to delete the object.

Note

Since the OAD is not supported in VisiBroker-RT for C++ this method does nothing.

Parameter	Description
`ReferenceData`	This parameter is not used, but is provided for compliance with the CORBA specification.
`CreationImplDef`	This pointer's true type is CreationImplDef. It provides the interface name, object name, path name of the executable and the activation policy and other parameters.
`Object_ptr`	Pointer to the object to be unregistered.

`static CORBA::BOA_ptr` **`_duplicate`**`(CORBA::BOA_ptr` **`ptr`**`);`

This static method duplicates the specified BOA pointer and returns a pointer to the duplicated BOA.

Parameter	Description
`ptr`	The ORB pointer to be duplicated.

`void` **`exit_impl_ready`**`()`

This method provides backward compatibility with earlier releases of VisiBroker-RT for C++. It invokes `BOA::shutdown`, described in "void shutdown()" on page 5-7, which causes a previous invocation of the `impl_is_ready` method to return.

```
CORBA::ReferenceData_ptr get_id(CORBA::Object_ptr)
```

This method returns the reference data for the specified object. The reference data is set by the object implementation at activation time and is guaranteed to remain constant throughout the life of the object.

Parameter	Description
obj	A pointer to the object whose reference data is to be returned.

```
CORBA::Principal_ptr get_principal(CORBA::Object_ptr
  obj, CORBA::Environment_ptr env=NULL)
```

This method returns the Principal object associated with the specified object. This method may only be called by an object implementation during the processing of a client operation request.

Parameter	Description
obj	A pointer to the object whose implementation is to be changed.
env	A pointer to the Environment object associated with this Principal.

```
void impl_is_ready(const char *service_name,
  extension::Activator_ptr activator, CORBA::Boolean
  block = 1)
```

This method instructs the BOA to delay activation of the object implementation associated with the specified service_name until a client requests the service. Once a client requests the service, the specified Activator object is to be used to activate the object implementation. If block is set to 0, this method will block the caller until the exit_impl_ready method is invoked.

Parameter	Description
service_name	The service name associated with the specified Activator object.
activator	The Activator to be used to activate the object implementation
block	If set to 1, indicates that this method should block the caller. If set to zero, the method will not block. The default behavior is to block.

```
void impl_is_ready(extension::ImplementationDef_ptr
  impl=NULL)
```

This method notifies the BOA that one or more objects in the server is ready to receive service requests. This method blocks the caller until the `exit_impl_ready` method is invoked. If all objects that the implementation is offering have been created through C++ instantiation and activated using the `obj_is_ready` method, do not specify the `ImplementationDef_ptr`.

An object implementation may offer only one object and may want to defer the activation of that object until a client request is received. In these cases, the object implementation does not need to first invoke the

obj_is_ready method. Instead, it may simply invoke this method, passing the ActivationImplDef pointer as its single object.

Parameter	Description
impl	This pointer's true type is ActivationImplDef and provides the interface name, object name, path name of the executable and activation policy, along with other parameters. See "ImplementationDef" for a complete discussion of the ActivationImplDef class.

```
static CORBA::BOA_ptr _nil()
```

This static method returns a NULL BOA pointer that can be used for initialization purposes.

```
void obj_is_ready(CORBA::Object_ptr obj,
  extension::ImplementationDef_ptr impl_ptr = NULL)
```

This method notifies the BOA that the specified object is ready for use by clients. There are two different ways to use this method:

- Objects that have been created using C++ instantiation should only specify a pointer to the object and let the ImplementationDef_ptr default to NULL.
- Objects whose creation is to be deferred until the first client request is received should specify a NULL Object_ptr and provide a pointer to an ActivationImplDef object that has been initialized.

Parameter	Description
obj	A pointer to the object to be activated.
impl_ptr	An optional pointer to an ActivationImplDef object.

```
static RegistrationScope scope()
```

This static method returns the registration scope of the BOA. The registration scope of an object can be SCOPE_GLOBAL or SCOPE_LOCAL. Only objects with a global scope are registered with the osagent.

```
static void scope(RegistrationScope val)
```

This static method changes the registration scope of the BOA to the specified value.

Parameter	Description
val	The scope for this BOA. Must be one of the following values: • LOCAL_SCOPE—For transient objects. • GLOBAL_SCOPE—For objects registered with the Smart Agent.

```
void shutdown()
```

This method causes a previous invocation of the impl_is_ready method to return.

VisiBroker extensions to CORBA::BOA

`CORBA::ULong` **`connection_max`**`()`

This method returns the maximum number of connections allowed.

`void` **`connection_max`**`(CORBA::ULong` **`max_conn`**`)`

This method is used by servers to set the maximum number of connections allowed. This property can also be set by using the command-line argument `-OAConnectionMax`, described in "Appendix: Using Command-Line Options".

Parameter	Description
`max_conn`	The maximum number of connections allowed.

`CORBA::ULong` **`thread_max`**`()`

This method returns the maximum number of threads allowed if the `TSession` thread policy has been selected.

`void` **`thread_max`**`(CORBA::ULong` **`max`**`)`

This method sets the maximum number of threads allowed when the TSession thread policy has been selected. If the current number of threads exceeds this number, the necessary number of extra threads are destroyed as soon as they are no longer in use.

Parameter	Description
`max`	The maximum number of threads to be allowed.

`CORBA::ULong` **`thread_stack_size`**`()`

This method returns the maximum number of threads allowed when the `TPool` thread policy is selected.

`void` **`thread_stack_size`**`(CORBA::ULong` **`size`**`)`

This method sets the maximum number of threads allowed when the TPool thread policy is selected. If the current number of threads exceeds that number, the necessary number of extra threads is destroyed as soon as they are no longer in use.

Parameter	Description
`size`	The new stack size to be set.

CompletionStatus

`enum` **`CompletionStatus`**

This enumeration represents how an operation request completed.

CompletionStatus members

COMPLETED_YES = 0	Indicates the operation request completed successfully.
COMPLETED_NO = 1	Indicates the operation request was not completed, due to some sort of exception or error.
COMPLETED_MAYBE = 2	Indicates that the operation request may have completed, in spite of an exception or error.

Context

`class` **`CORBA::Context`**

The `Context` class represents information about a client application's environment that is passed to a server as an implicit parameter during static or dynamic method invocations. It can be used to communicate special information that needs to be associated with a request, but is not part of the method's argument list.

The `Context` class consists of a list of properties, stored as name-value pairs, and provides methods for setting and manipulating those properties. A `Context` contains an NVList object and chains the name-value pairs together.

A `Context_var` class is also available and provides simpler memory management semantics.

See also `ORB::get_default_context` in "CORBA::Status get_default_context(CORBA::Context_ptr&);".

Include file

The **corba.h** file should be included when you use this class.

Context methods

```
const char *context_name() const;
```

This method returns the name used to identify this context. If no name was provided when this object was created, it returns a `NULL` value.

```
void create_child(const char * name, CORBA::Context_out
  Context_ptr);
```

This method creates a child `Context` for this object.

Parameter	Description
`name`	The name of the new `Context` object.
`Context_ptr`	A reference to newly created child `Context`.

```
void delete_values(const char *name);
```

This method deletes one or more properties from this object.

Parameter	Description
name	The name of the property, or properties, to be deleted. To delete all matching properties, the name may contain a trailing "*" wildcard character. To delete all properties, specify a single asterisk.

```
static CORBA::Context_ptr _duplicate(CORBA::Context_ptr
  ctx);
```

This method duplicates the specified object.

Parameter	Description
ctx	The object to be duplicated.

```
void get_values(const char *start_scope, CORBA::Flags,
  const char *name, CORBA::NVList_out NVList_ptr)
```

This method searches the Context object hierarchy and retrieves one or more of the name/value pairs specified by the `name` parameter. It then creates an `NVList` object and places the name/value pairs in the `NVList`.

The start_scope parameter specifies the name of the context where the search is to begin. If the property is not found, the search continues up the `Context` object hierarchy until a match is found or until there are no more `Context` objects to search.

Parameter	Description
start_scope	The name of the Context object at which to start the search. If set to `CORBA::Context::_nil()`, the search begins with the current Context. To restrict the search scope can to just the current Context, specify `CORBA::CTX_RESTRICT_SCOPE`.
Flags	An exception is raised if no matching context name is found.
name	The property name to search for. A trailing "*" wildcard character may be used to retrieve all properties that match `name`.
NVList_ptr	A reference to the list of properties found

```
static CORBA::Context_ptr _nil();
```

This method returns a `NULL Context_ptr` suitable for initialization purposes.

```
CORBA::Context_ptr parent();
```

This method returns a pointer to the parent `Context`. If there is no parent `Context`, a `NULL` value is returned.

```
static void _release(CORBA::Context_ptr ctx);
```

This static method releases the specified `Context` object. Once the object's reference count reaches zero, the object is automatically deleted.

Parameter	Description
ctx	The object to be released.

```
void set_one_value(const char *name, const
  CORBA::Any&);
```

This method adds a property to this object using the specified name and value.

Parameter	Description
`name`	The property's name.
`const Any&`	The property's value.

```
void set_values(CORBA::NVList_ptr _list);
```

This method adds one or more properties to this object, using the name/value pairs specified in the `NVList`. When you create the `NVList` object to be used as an input parameter to this method, you must set the `Flags` field to zero and each `Any` object added to the `NVList` must have its `TypeCode` set to `TC_string`. For more information on the `NVList` class, see "NVList methods" in this guide.

Parameter	Description
`_list`	list of name/value pairs to be added to this object.

PortableServer::Current

```
class PortableServer::Current : public CORBA::Current
```

This class provides methods with access to the identity of the object on which the method was called. The Current class provides support for servants which implement multiple objects but can be used within the context of POA-dispatched method invocations on any servant.

PortableServer::Current methods

```
PortableServer::POA get_POA(); POA *get_POA();
```

This method returns a reference to the POA which implements the object in whose context it is called. If this method is called from outside the context of a POA-dispatched method, a `NoContext` exception is raised.

```
PortableServer::ObjectId get_object_id();
```

This method returns the ObjectId which identifies the object in whose context it was called. If this method is called from outside the context of a POA-dispatched method, a `NoContext` exception is raised.

Exception

```
class CORBA::Exception
```

The `Exception` class is the base class of the system exception and user exception classes. For more information, see "SystemException" in this guide.

Include file

You should include the **corba.h** file when using this class.

Object

`class` **`CORBA::Object`**

All ORB objects are derived from the `Object` class, which provides methods for binding clients to objects and manipulating object references as well as querying and setting an object's state. The methods offered by the `Object` class are implemented by the ORB.

VisiBroker-RT for C++ provides extensions to the CORBA Object specification. These are covered in ""VisiBroker extensions to CORBA::Object".

Include file

You should include the file **corba.h** when using this class.

CORBA::Object methods

```
void _create_request(CORBA::Context_ptr ctx, const char
  *operation, CORBA::NVList_ptr arg_list,
  CORBA::NamedValue_ptr result, CORBA::Request_out
  request, CORBA::Flags req_flags);
```

This method creates a Request for an object implementation that is suitable for invocation with the Dynamic Invocation Interface.

Parameter	Description
`ctx`	The Context associated with this request. For more information, see "CompletionStatus".
`operation`	The name of the operation to be performed on the object implementation.
`arg_list`	A list of arguments to pass to the object implementation. See "NVList methods" for more information.
`result`	The result of the operation. See "NamedValue methods" for more information.
`request`	A pointer to the `Request` that is created. See "Request methods" for more information.
`req_flags`	This flag must be set to `OUT_LIST_MEMORY` if one or more of the`NamedValue` items in `arg_list` is an output argument.

```
void _create_request(CORBA::Context_ptr ctx, const char
  *operation,
  CORBA::NVList_ptr arg_list,
  CORBA::NamedValue_ptr result,
  CORBA::ExceptionList_ptr eList,
  CORBA::ContextList_ptr ctxList,
  CORBA::Request_out request,
  CORBA::Flags req_flags);
```

This method creates a `Request` for an object implementation that is suitable for invocation with the Dynamic Invocation Interface.

Parameter	Description
`ctx`	The Context associated with this request. For more information, see "CompletionStatus".
`operation`	The name of the operation to be performed on the object implementation.

Parameter	Description
arg_list	A list of arguments to pass to the object implementation. See "NVList methods" for more information.
result	The result of the operation. See "NamedValue methods" for more information.
eList	A list of exceptions for this request.
ctxList	A list of Context objects for this request.
request	A pointer to the Request that is created. See "Request methods" for more information.
req_flags	This flag must be set to OUT_LIST_MEMORY if one or more of the NamedValue items in arg_list is an output argument.

```
static CORBA::Object_ptr _duplicate(CORBA::Object_ptr
  obj);
```

This static method duplicates the specified Object_ptr and returns a pointer to the object. The object's reference count is increased by one.

Parameter	Description
obj	The object pointer to be duplicated.

```
CORBA::InterfaceDef_ptr _get_interface();
```

This method returns a pointer to this object's interface definition. See "InterfaceDef methods" for more information.

```
CORBA::ULong _hash(CORBA::ULong maximum);
```

This method returns a hash value for this object. This value will not change for the lifetime of this object, however the value is not necessarily unique. If two objects return different hash values, then they are not identical. The upper bound of the hash value may be specified. The lower bound is zero.

Parameter	Description
maximum	The upper bound of the hash value returned.

```
CORBA::Boolean _is_a(const char *logical_type_id);
```

This method returns TRUE if this object implements the interface associated with the repository id. Otherwise, it returns FALSE.

Parameter	Description
logical_type_id	The repository identifier to check

```
CORBA::Boolean _is_equivalent(CORBA::Object_ptr
  other_object);
```

This method returns TRUE if the specified object pointer and this object point to the same object implementation. Otherwise, it returns FALSE.

Parameter	Description
other_object	Pointer to an object that is to be compared to this object.

```
static CORBA::Object_ptr _nil();
```

This static method returns a NULL pointer suitable for initialization purposes.

```
CORBA::Boolean _non_existent();
```

This method returns `TRUE` if the object represented by this object reference no longer exists.

```
CORBA::Request_ptr _request(const char* operation);
```

This method creates a `Request` suitable for invoking methods on this object. A pointer to the `Request` object is returned. See "Request methods" for more information.

Parameter	Description
`operation`	The name of the object method to be invoked.

VisiBroker extensions to CORBA::Object

Note

The following method is deprecated since VisiBroker 4.0.

```
CORBA::BindOptions* _bind_options();
```

This method returns a pointer to the bind options that will be used for this object only. For more information, see "BindOptions".

Note

The following method is deprecated since VisiBroker 4.0.

```
void _bind_options(const CORBA::BindOptions& opt);
```

This method sets the bind options for this object only. The options that are set will remain in effect for the lifetime of the proxy object. Any changes to time-out values will apply to all subsequent send and receive operations as well as any re-bind operations. For more information, see "BindOptions".

Parameter	Description
opt	The new bind options for this object.

```
static CORBA::Object_ptr _bind_to_object(const char
  *rep_id, const char *object_name=NULL, const char
  *host_name=NULL, const CORBA::BindOptions
  *options=NULL, CORBA::ORB_ptr orb=NULL);
```

This method attempts to bind to the object with the specified `repository_id` and `object_name`, on the specified host, using the specified `BindOptions` and `ORB`.

NOTE:

This _bind method must be used if the servant was activated with a POA having the bind_policy value of "BY_INSTANCE".

Parameter	Description
`rep_id`	The repository ID of the desired object.
`object_name`	The name of the desired object.
`host_name`	The name of the desired host where the object implementation is executing.

Parameter	Description
options	The bind options for this connection. See "BindOptions" for more information.
orb	The ORB to use.

```
static CORBA::Object_ptr _bind_to_object(const char
  *logical_type_id, const char *poa_name=NULL, const
  CORBA_Octect_sequence& oid, const char
  *host_name=NULL, const CORBA::BindOptions
  *options=NULL, CORBA::ORB_ptr orb=NULL);
```

This method attempts to bind to the object with the specified `repository_id` and `object_name`, on the specified host, using the specified `BindOptions` and `ORB`.

NOTE:

This _bind method must be used if the servant was activated with a POA having the bind_policy value of "BY_POA". If a bind_policy is not specified during POA creation the default behavior for servant activation is "BY_POA".

Parameter	Description
logical_type_id	The repository ID of the desired object.
poa_name	The name of the poa that the servant was activated on.
oid	The object id (i.e. object name) of the desired object.
host_name	The name of the desired host where the object implementation is executing.
options	The bind options for this connection. See "BindOptions" for more information.
orb	The ORB to use.

Note

The following method is deprecated since VisiBroker 4.0.

```
CORBA::BOA _boa() const;
```

This method returns a pointer to the Basic Object Adaptor with which this object is registered.

```
static CORBA::Object_ptr _clone(CORBA::Object_ptr obj,
  CORBA::Boolean reset_connection = 1UL);
```

This method clones the specified object reference.

Parameter	Description
obj	The object reference to be cloned.
reset_connection	This parameter is not used.

Note

The following method is deprecated since VisiBroker 4.0.

```
static const CORBA::BindOptions *
  _default_bind_options();
```

This method returns a pointer to the global, per client process BindOptions. For more information, see "BindOptions".

Note

The following method is deprecated since VisiBroker 4.0.

```
static void _default_bind_options(const
  CORBA::BindOptions&);
```

This method sets the bind options that will be used by default for all `_bind` invocations that do not specify their own bind options. For more information, see "BindOptions".

```
static const CORBA::TypeInfo *_desc();
```

Returns type information for this object.

```
const char *_interface_name() const;
```

This method returns this object's interface name.

```
CORBA::Boolean _is_bound() const;
```

This method returns `TRUE` if the client process has established a connection to an object implementation.

```
CORBA::Boolean _is_local() const;
```

This method returns `TRUE` if the object implementation resides within the same process or address space as the client application.

```
CORBA::Boolean _is_persistent() const;
```

This method returns `TRUE` if this object is a persistent object, and `FALSE` if it is transient.

```
CORBA::Boolean _is_remote() const;
```

This method returns `TRUE` if the object implementation resides in a different process or address space than the client application. The client and object implementation may or may not reside on the same host.

```
const char *_object_name() const;
```

This method returns the object name associated with this object.

```
CORBA::Long _ref_count() const;
```

Returns the reference count for this object.

```
void _release();
```

Decrements this object's reference count and releases the object if the reference count has reached 0.

```
const char *_repository_id() const;
```

This method returns this object's repository identifier.

```
CORBA::Object_ptr _resolve_reference(const char* id);
```

Your client application can invoke this method on an object reference to resolve the server-side interface with the specified service identifier. This

method causes the `ORB::_resolve_initial_references` method, described in "CORBA::Object_ptr resolve_initial_references(const char * identifier);", to be invoked on the server-side to resolve the specified service. This method returns an object reference which your client can narrow to the appropriate server type.

This method is typically used by client applications that wish to manage a server's attributes.

Parameter	Description
`id`	The name of the interface to be resolved on the server-side.

ORB

class CORBA::ORB

The `ORB` class provides an interface to the Object Request Broker. It offers methods to the client object, independent of the particular `Object` or Object Adaptor.

VisibrokerVisiBroker-RT for C++ provides extensions to the CORBA ORB that are covered in "VisiBroker extensions to CORBA::ORB" on page 5-22.

These methods provide for the management of connections, threads, and the activation of services.

Include file

You should include the file **corba.h** when using this class.

CORBA::ORB methods

```
CORBA::Boolean work_pending();
```

This method returns true if the ORB has any work waiting to be processed.

```
static CORBA::TypeCode_ptr create_alias_tc(const char
  *repository_id, const char
  *type_name, CORBA::TypeCode_ptr original_type);
```

This static method dynamically creates a `TypeCode` for the alias with the specified type and name.

Parameter	Description
`repository_id`	The identifier generated by the IDL compiler or constructed dynamically.
`type_name`	The name of the alias's type.
`original_type`	The type of the original for which this alias is being created.

```
static CORBA::TypeCode_ptr create_array_tc(CORBA::Ulong
  length, TypeCode_ptr element_type);
```

This static method dynamically creates a `TypeCode` for an array.

Parameter	Description
`length`	The maximum number of array elements.
`element_type`	The type of elements stored in this array.

```
static CORBA::TypeCode_ptr create_enum_tc(const char
  *repository_id, const char *type_name, const
  CORBA::EnummemberSeq& members);
```

This static method dynamically creates a `TypeCode` for an enumeration with the specified type and members.

Parameter	Description
repository_id	The identifier generated by the IDL compiler or constructed dynamically.
type_name	The name of the enumeration's type.
members	A list of values for the enumeration's members.

```
void create_environment(CORBA::Environment_out);
```

This method creates an `Environment` object.

Parameter	Description
`env`	The reference that will be set to point to the newly created `Environment`.

```
static CORBA::TypeCode_ptr create_exception_tc(const
  char *repository_id, const char *type_name, const
  CORBA::StructMemberSeq& members);
```

This static method dynamically creates a `TypeCode` for an exception with the specified type and members.

Parameter	Description
`repository_id`	The identifier generated by the IDL compiler or constructed dynamically.
`type_name`	The name of the structure's type.
`members`	A list of values for the structure members.

```
static CORBA::TypeCode_ptr create_interface_tc(const
  char *repository_id, const char *type_name);
```

This static method dynamically creates a `TypeCode` for the interface with the specified type.

Parameter	Description
`repository_id`	The identifier generated by the IDL compiler or constructed dynamically.
`type_name`	The name of the interface's type.

```
void create_list(CORBA::Long, CORBA::NVList_out);
```

This method creates an `NVList` with the specified number of elements and returns a reference to the list.

Parameter	Description
`num`	The number of elements in the list.
`nvlist`	Initialized to point to the newly-created list.

```
void create_named_value(CORBA::NamedValue_out);
```

This method creates a `NamedValue` object.

```
void  create_operation_list(CORBA::OperationDef_ptr,
  CORBA::NVList_out);
```

This method creates an argument list for the specified `OperationDef`object.

```
static CORBA::TypeCode_ptr
  create_recursive_sequence_tc(CORBA::Ulong bound,
  CORBA::Ulong offset);
```

This static method dynamically creates a `TypeCode` for a recursive sequence. The result of this method can be used to create other types. The offset parameter determines which enclosing `TypeCode` describes the elements of this sequence.

Parameter	Description
`bound`	The maximum number of sequence elements.
`offset`	Position within the buffer where the type code for the current element was previously generated.

```
static CORBA::TypeCode_ptr
  create_sequence_tc(CORBA::Ulong bound,
  CORBA::TypeCode_ptr element_type);
```

This static method dynamically creates a `TypeCode` for a sequence.

Parameter	Description
`bound`	The maximum number of sequence elements.
`element_type`	The type of elements stored in this sequence.

```
static CORBA::TypeCode_ptr
  create_string_tc(CORBA::Ulong bound);
```

This static method dynamically creates a `TypeCode` for a string.

Parameter	Description
bound	The maximum length of the string.

```
static CORBA::TypeCode_ptr create_struct_tc(const char
  *repository_id, const char *type_name, const
  CORBA::StructMemberSeq& members);
```

This static method dynamically creates a `TypeCode` for the structure with the specified type and members.

Parameter	Description
`repository_id`	The identifier generated by the IDL compiler or constructed dynamically.
`type_name`	The name of the structure's type.
`members`	A list of values for the structure members.

```
static CORBA::TypeCode_ptr create_union_tc(const char
  *repository_id, const char *type_name,
  CORBA::TypeCode_ptr discriminator_type, const
  CORBA::UnionMemberSeq& members);
```

This static method dynamically creates a `TypeCode` for a union with the specified type, discriminator and members.

Parameter	Description
`repository_id`	The identifier generated by the IDL compiler or constructed dynamically.
`type_name`	The name of the union's type.
`discriminator_type`	The discriminating type for the union.
`members`	A list of values for the union members.

```
CORBA::Status get_default_context(CORBA::Context_ptr&);
```

This method returns the default per-process `Context` maintained by VisiBroker. The default Context is often used in constructing DII requests. See "Context" for more information.

Parameter	Description
`CORBA::Context_ptr&`	The property's value.

```
CORBA::Status get_next_response(CORBA::RequestSeq*&
  req);
```

This method blocks waiting for the response associated with a deferred request. You can use the `ORB::poll_next_response` method to determine if there is a response waiting to be received before you call this method.

Parameter	Description
`req`	Set to point to the request that has been received.

```
ObjectIdList *list_initial_services();
```

This method returns a list of the names of any object services that are available to your application. These services may include the Location Service, Interface Repository, Name Service, or Event Service. You can use any of the returned names with the `ORB::resolve_initial_references` method, described in "CORBA::Object_ptr resolve_initial_references(const char * identifier);" on page 5-21, to obtain the top-level object for that service.

```
char *object_to_string(CORBA::Object_ptr) = 0;
```

This method converts the specified object reference to a string, a process referred to as "stringification" in the CORBA specification. Object references that have been converted to strings can be stored in files, for example. This is an ORB method because different ORB implementations may have different conventions for representing object references as strings.

Note

While an object reference can be made persistent by saving it to a file, the object itself is not made persistent.

Parameter	Description
obj	Pointer to an object that is to be converted to a string.

Note

The following method is deprecated since VisiBroker 4.0.

```
CORBA::BOA_ptr ORB::BOA_init(int& argc, char *const
  *argv, const char *boa_identifier = (char *)NULL);
```

This ORB method returns a handle to the BOAand specifies optional networking parameters. The argc and argv parameters are the same parameters passed to the object implementation process when it is started. See "Appendix: Using Command-Line Options" for a complete description of the BOA_init options that may be specified.

Parameter	Description
argc	The number of arguments passed.
argv	An array of char pointers to the arguments. All but two of the arguments take the form of a keyword and a value and are shown below. This method will ignore any keywords that it does not recognize.
boa_identifier	Identifies the type of BOA to be used. TPool is always used in VisiBroker-RT for C++; specifying TSingle will return a NULL BOA ptr.

```
static CORBA::ORB_ptr ORB_init(int& argc, char *const
  *argv,const char *orb_id = NULL);
```

This method initializes the ORB and is used by both clients and object implementations. It returns a pointer to the ORB that can be used to invoke ORB methods. The argc and argv parameters passed to the application's main function can be passed directly to this method. Arguments accepted by this method take the form of name-value pairs which allows them to be distinguished from other command line arguments. See "Appendix: Using Command-Line Options" for a complete description of the ORB_init options that may be specified.

Parameter	Description
argc	The number of arguments passed.
argv	An array of char pointers to the arguments. All but two of the arguments take the form of a keyword and a value. This method will ignore any keywords that it does not recognize.
orb_id	Identifies the type of ORB to be used. The default is IIOP.

```
void perform_work();
```

This method instructs the ORB to perform some work.

```
CORBA::Boolean poll_next_response();
```

This method returns `TRUE` if a response to a deferred request has been received, otherwise `FALSE` is returned. This call does not block.

```
CORBA::Object_ptr resolve_initial_references(const char
  * identifier);
```

This method resolves one of the names returned by the `ORB::list_initial_services` method, described in "ObjectIdList *list_initial_services();" on page 5-20, to its corresponding implementation object. The resolved object which is returned can then be narrowed to the appropriate server type. If the specified service cannot be found, an `InvalidName` exception will be raised.

Parameter	Description
`identifier`	The name of the service whose top-level object is to be returned. The identifier is not the name of the object to be returned.

```
void send_multiple_requests_deferred(const
  CORBA::RequestSeq& req);
```

This method sends all the client requests in the specified sequence as deferred requests. The ORB will not wait for any responses from the object implementation. The client application is responsible for retrieving the responses to each request using the `ORB::get_next_response` method.

Parameter	Description
`req`	A sequence of deferred requests to be sent.

```
void send_multiple_requests_oneway(const
  CORBA::RequestSeq& req);
```

This method sends all the client requests in the specified sequence as one-way requests. The ORB does not wait for a response from any of the requests because one-way requests do not generate responses from the object implementation.

Parameter	Description
`req`	A sequence of one-way requests to be sent.

```
CORBA::Object_ptr string_to_object(const char *str);
```

This method converts a string representing an object into an object pointer. The string must have been created using the `ORB::object_to_string` method.

Parameter	Description
`str`	A pointer to a string representing an object.

```
static CORBA::ORB_ptr _duplicate(CORBA::ORB_ptr ptr);
```

This static method duplicates the specified ORB pointer and returns a pointer to the duplicated ORB.

Parameter	Description
ptr	The ORB pointer to be duplicated.

```
static CORBA::ORB_ptr _nil();
```

This static method returns a NULL ORB pointer suitable for initialization purposes.

```
void run();
```

This method informs the ORB to start processing work. This ORB receives requests and dispatches them. This call blocks this process until the ORB is shut down.

VisiBroker extensions to CORBA::ORB

```
CORBA::Object_ptr bind(const char *rep_id, const char
  *object_name = (const char*)NULL, const char
  *host_name = (const char*)NULL, CORBA::BindOptions
  *opt = (CORBA::BindOptions*)NULL);
```

This method allows you obtain a generic object reference to an object by specifying the repository id of the object and optionally, its object name and host name where it is implemented.

NOTE:
This bind method must be used if the servant was activated with a POA having the bind_policy value of "BY_INSTANCE".

Parameter	Description
rep_id	The identifier generated by the IDL compiler or constructed dynamically for the object.
object_name	The name of the object. This is an optional parameter.
host_name	The host name where the object implementation is located. This may be specified as an IP address or as a fully qualified host name.
opt	Any bind options for the object. Bind options are described in “BindOptions”.

```
static CORBA::Object_ptr _bind_to_object(const char
  *logical_type_id, const char *poa_name=NULL,
  const CORBA_Octect_sequence& oid, const char
  *host_name=NULL, const CORBA::BindOptions
  *options=NULL);
```

This method allows you obtain a generic object reference to an object by specifying the repository id of the object along with the POA name and the object id (i.e. object name) and optionally the host name where it is implemented.

NOTE:
This bind method must be used if the servant was activated with a POA having the bind_policy value of "BY_POA". If a bind_policy is not specified

during POA creation the default behavior for servant activation is "BY_POA".

Parameter	Description
logical_type_id	The repository ID of the desired object.
poa_name	The name of the poa that the servant was activated on.
oid	The object id (i.e. object name) of the desired object.
host_name	The name of the desired host where the object implementation is executing.
options	The bind options for this connection. See "BindOptions" for more information.

`CORBA::ULong` **`connection_count`**`()`

This method is used by client applications to return the current number of active connections.

`void` **`connection_max`**`(CORBA::ULong` **`max_conn`**`)`

This method is used by client applications to set the maximum number of connections to be allowed. This property can also be set by using the command-line argument `-OAConnectionMax`, described in "Appendix: Using Command-Line Options".

Parameter	Description
max_conn	The maximum number of connections to be allowed

`CORBA::ULong` **`connection_max`**`()`

This method is used by client applications to return the maximum number of connections that will be allowed.

```
static CORBA::TypeCode_ptr
  create_wstring_tc(CORBA::Ulong bound);
```

This static method dynamically creates a `TypeCode` for a Unicode string.

Parameter	Description
bound	The maximum length of the string.

`static VISPropertyManager_pt` **`getPropertyManager`**`()`

This method is used by to get a handle to the VisiBroker Property Manager instance which is being used by the ORB.

static void **shutdown**(CORBA::Boolean **wait_for_completion**`=0`);

This method causes a previous invocation of the `impl_is_ready` method to return. All object adapters are shut down and any associated memory is freed.

PortableServer::POA

`class` **`PortableServer::POA`**

Objects of the POA class manage the implementations of a collection of objects. The POA supports a name space for these objects which are identified by Object Ids. A POA also provides a name space for other POAs in

that a POA must be created as a child of an existing POA, which then forms a hierarchy starting with the root POA.

A POA object must not be exported to outisde of the ORB instance in which they were created, or be stringified. A `MARSHAL` exception is raised if this is attempted.

PortableServer::POA methods

```
PortableServer::ObjectId*
  activate_object(PortableServer::Servant _p_servant);
```

This method generates an object id and returns it. The object id and the specified `_p_servant` are entered into the Active Object Map. If the `UNIQUE_ID` policy is present with the POA and the specified `_p_servant` is already in the Active Object Map, then a `ServantAlreadyActive` exception is raised.

This method requires that the `SYSTEM_ID` and `RETAIN` policies be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
`_p_servant`	The `Servant` to be entered into the Active Object Map.

```
void activate_object_with_id(const
  PortableServer::ObjectId& _oid,
  PortableServer::Servant _p_servant);
```

This method attempts to activate the specified `_oid` and to associate it with the specified `_p_servant` in the Active Object Map. If the `_oid` already has a servant bound to it in the Active Object Map, then an `ObjectAlreadyActive` exception is raised. If the POA has the `UNIQUE_ID` policy present and the `_p_servant` is already in the Active Object map, then a `ServantAlreadyActive` exception is raised.

If the POA has the SYSTEM_ID policy present and it detects that the _oid was not generated by the system or for the POA, then this method raises a BAD_PARAM system exception.

This method requires that the `RETAIN` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
`oid`	The `ObjectId` of the object to be activated.
`_p_servant`	The `Servant` to be entered into the Active Object Map.

```
PortableServer::BindSupportPolicy_ptr
  create_bind_support_policy(
  PortableServer::BindSupportPolicyValue _value);
```

This method returns a pointer to BindSupportPolicy object with the specified _value. The application is responsible for calling the inherited destroy method on the `Policy` object after the `Policy` object is no longer needed.

If no BindSupportPolicy is specified at POA creation, then the default is BY_POA.

Parameter	Description
_value	If set to BY_INSTANCE, all objects activated on this POA are registered with the osagent. The POA must also use the PERSISTENT and RETAIN policy with this value. If set to BY_POA, POA names are registered with the osagent. The POA must also use the PERSISTENT policy with this value. If set to NO_REGISTRATION, neither POAs nor active objects are registered with the osagent

```
PortableServer::ImplicitActivationPolicy_ptr
  create_implicit_activation_policy(
  PortableServer::ImplicitActivationPolicyValue
  _value);
```

This method returns a pointer to an `ImplicitActivationPolicy` object with the specified _value. The application is responsible for calling the inherited destroy method on the `Policy` object after the `Policy` object is no longer needed.

If no `ImplicitActivationPolicy` is specified at POA creation, then the default is NO_IMPLICIT_ACTIVATION.

Parameter	Description
_value	If set to IMPLICIT_ACTIVATION, the POA will support implicit activation of servants: also requires SYSTEM_ID and RETAIN policies. If set to NO_IMPLICIT_ACTIVATION, the POA will not support the implicit activation of servants.

```
PortableServer::ServerEnginePolicy_ptr
  create_server_engine_policy(const
  CORBA::StringSequence _value);
```

This method returns a pointer to a ServerEnginePolicy object with the specified _value. The application is responsible for calling the inherited destroy method on the `Policy` object after the `Policy` object is no longer needed.

If no ServerEnginePolicy is specified at POA creation, then the default is to associate the newly created POA with the IIOP Server Engine (iiop_tp0).

Parameter	Description
_value	This should contain a sequence of strings where each string denotes a installed and configured Server Engine. Each string must match the Server Engine name assigned via the VisiBroker Property Manager.

```
CORBA::Object_ptr create_reference(const char* _intf);
```

This method creates and returns an object reference that encapsulates a POA-generated ObjectId and the specified `_intf` values. The _intf, which may be null, becomes the `type_id` of the generated object reference. This method does not cause an activation to take place. Undefined behavior results if the `_intf` value does not identify the most derived interface of the object or one of its base interfaces. The ObjectId may be obtained by invoking the `POA::reference_to_id` method on the returned Object.

This method requires that the RETAIN policy be present with the POA; otherwise, a WrongPolicy exception is raised.

Parameter	Description
_intf	The repository interface id of the class of the object to be created.

```
CORBA::Object_ptr create_reference_with_id (const
  PortableServer::ObjectId& _oid,const char* _intf);
```

This method creates and returns an object reference that encapsulates the specified _oid and _intf values. The _intf, which may be a null string, becomes the type_id of the generated object reference. A _intf value that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior. This method does not cause an activation to take place. The returned object reference may be passed to clients, so that subsequent requests on those references will cause the object to be activated if necessary, or the default servant used, depending on the applicable policies.

If the POA has the SYSTEM_ID policy present, and it detects the ObjectId value was not generated by the system or for the POA, this method may raise a BAD_PARAM system exception.

Parameter	Description
_oid	The object id for which a reference is to be created.
_intf	The repository interface id of the class of the object to be created.

```
PortableServer::IdAssignmentPolicy_ptr
  create_id_assignment_policy
  (PortableServer::IdAssignmentPolicyValue _value);
```

This method returns a pointer to a IdAssignmentPolicy object with the specified _value. The application is responsible for calling the inherited destroy method on the Policy object after it is no longer needed.

If no IdAssignmentPolicy is specified at POA creation, then the default is SYSTEM_ID.

Parameter	Description
_value	If set to USER_ID, then objects created by the POA are assigned object ids only by the application. If set to SYSTEM_ID, then objects created with the POA are assigned object ids only by the POA.

```
PortableServer::IdUniquenessPolicy_ptr
  create_id_uniqueness_policy
  (PortableServer::IdUniquenessPolicyValue _value);
```

This method returns a pointer to an IdUniquenessPolicy object with the specified _value. The application is responsible for calling the inherited destroy method on the Policy object after it is no longer needed.

If no `IdUniquenessPolicy` is specified at POA creation, then the default is `UNIQUE_ID`.

Parameter	Description
`_value`	If set to `UNIQUE_ID`, servants which are activated with the POA support exactly one object id. If set to `MULTIPLE_ID`, then a servant which is activated with the POA may support one or more object ids.

```
PortableServer::LifespanPolicy_ptr
  create_lifespan_policy
  (PortableServer::LifespanPolicyValue _value);
```

This method returns a pointer to a `LifespanPolicy` object with the specified _value. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `LifespanPolicy` is specified at POA creation, then the default is `TRANSIENT`.

Parameter	Description
`_value`	If set to `TRANSIENT`, then objects implemented in the POA cannot outlive the POA instance in which they were first created. Once a transient POA is deactivated, the use of any object references generated from it result in an `OBJECT_NOT_EXIST` exception being raised. If set to `PERSISTENT`, then the objects implemented in the POA can outlive any process in which they are first created.

```
PortableServer::POA_ptr create_POA(const char*
  _adapter_name, PortableServer::POAManager_ptr
  _a_POAManager, const CORBA::PolicyList& _policies);
```

This method creates a new POA with the specified `_adapter_name`. The new POA is a child of the specified `_a_POAManager`. If a child POA with the same name already exists for the parent POA, an `PortableServer::AdapterAlreadyExists` exception is raised.

The specified `_policies` are associated with the new POA and are used to control its behavior.

Parameter	Description
`_adapter_name`	The name which specifies the new POA.
`_a_POAManager`	The parent POA object of the new POA.
`_policies`	A list of policies which are to apply to the new POA.


```
PortableServer::RequestProcessingPolicy_ptr
  create_request_processing_policy
  (PortableServer::RequestProcessingPolicyValue
  _value);
```

This method returns a pointer to a `RequestProcessingPolicy` object with the specified _value. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no RequestProcessingPolicy is specified at POA creation, then the default is `USE_ACTIVE_OBJECT_MAP_ONLY`.

Parameter	Description
`_value`	If set to `USE_ACTIVE_OBJECT_MAP_ONLY` and the object id is not found in the Active Object Map, then an `OBJECT_NOT_EXIST` exception is returned to the client. (The RETAIN policy is also required.) If set to `USE_DEFAULT_SERVANT` and the object id is not found in the Active Object Map or the `NON_RETAIN` policy is present, and a default servant has been registered with the POA using the set_servant method, then the request is dispatched to the default servant. If no default servant has been registered, then an `OBJ_ADAPTER` exception is returned to the client. (The `MULTIPLE_ID` policy is also required.) If set to `USE_SERVANT_MANAGER` and the object id is not found in the Active Object Map or the `NON_RETAIN` policy is present, and a servant manager has been registered with the POA using the `set_servant_manager` method, then the servant manager is given the opportunity to locate a servant or raise an exception. If no servant manager has been registered, then an `OBJ_ADAPTER` is returned to the client.

```
PortableServer::ServantRetentionPolicy_ptr
  create_servant_retention_policy
  (PortableServer::ServantRetentionPolicyValue
  _value);
```

This method returns a pointer to a `ServantRetentionPolicy` object with the specified `_value`. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `ServantRetentionPolicy` is specified at POA creation, then the default is `RETAIN`.

Parameter	Description
`_value`	If set to `RETAIN`, then the POA will retain active servants in its Active Object Map. If set to `NON_RETAIN`, then servants are not retained by the POA.

```
PortableServer::ThreadPolicy_ptr create_thread_policy
  (PortableServer::ThreadPolicyValue _value);
```

This method returns a pointer to a `ThreadPolicy` object with the specified _value. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `ThreadPolicy` is specified at POA creation, then the default is `ORB_CTRL_MODEL`.

Parameter	Description
`_value`	If set to ORB_CTRL_MODEL, the ORB is responsible for assigning requests for an ORB-controlled POA to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads. If set to SINGLE_THREAD_MODEL, then requests to the POA are processed sequentially. In a multi-threaded environment, all upcalls made by the POA to servants and servant managers are made in a manner that is safe for code that is multi-thread unaware.

```
void deactivate_object(const PortableServer::ObjectId&
  _oid);
```

This method causes the specified `_oid` to be deactivated. An ObjectId which has been deactivated continues to process requests until there are no more active requests for that ObjectId. An ObjectId is removed from the Active Object Map when all requests executing for that ObjectId have completed.

If a `ServantManager` is associated with the POA, then the `ServantActivator::etheralize` method is invoked with the ObjectId and the associated servant after the ObjectId has been removed from the Active Object map. Reactivization for the ObjectId blocks until etherealization, if necessary, has completed. However, the method does not wait for requests or etherealization to complete and always returns immediately after deactivating the specified `_oid`.

This method requires that the `RETAIN` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
`_oid`	The ObjectId of the object to be deactivated.

```
void destroy(CORBA::Boolean _etherealize_objects,
  CORBA::Boolean _wait_for_completion);
```

This method destroys this POA object and all of its descendant POAs. First the children are destroyed and finally the current container POA. If desired, later, a POA with that same name in the same process can be created.

Parameter	Description
`_etherealize_objects`	If TRUE, the POA has the RETAIN policy, and a servant manager has registered with the POA, then the etherealize method is called on each active object in the Active Object Map. The apparent destruction of the POA occurs before the etherealize method is called, and thus any etherealize method which attempts to invoke methods on the POA raises a OBJECT_NOT_EXIST exception.
`_wait_for_completion`	If `TRUE` and the current thread is not in an invocation context dispatched from some POA belonging to the same ORB as this POA, the `destroy` method only returns after all active requests and all invocations of etherealize have completed. If `TRUE` and the current thread is in an invocation context dispatched from some POA belonging to the same ORB as this POA, the `BAD_INV_ORDER` exception is raised and POA destruction does not occur.

```
PortableServer::POA_ptr find_POA(const char*
  _adapter_name,CORBA::Boolean _activate_it);
```

If the POA object on which this method is called is the parent of the POA with the specified `_adapter_name`, the child POA is returned.

Parameter	Description
`_adapter_name`	The name of the AdapterActivator associated with the POA.
`_activate_it`	If set to TRUE and no child POA of the POA specified by `_adapter_name` exists, then the POA's AdapterActivator, if not null, is invoked, and, if it successfully activates the child POA, then that POA is returned. Otherwise an `AdapterNonExistent` exception is raised.

```
PortableServer::Servant get_servant();
```

This method returns the default `Servant` associated with the POA. If no `Servant` has been associated, then a `NoServant` exception is raised.

This method requires that the `USE_DEFAULT_SERVANT` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

```
PortableServer::ServantManager_ptr
  get_servant_manager();
```

This method returns a pointer to the `ServantManager` object associated with the POA. The result is null if no `ServantManager` is associated with the POA.

This method requires that the `USE_SERVANT_MANAGER` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

```
CORBA::Object_ptr
  id_to_reference(PortableServer::ObjectId& _oid);
```

This method returns an object reference if the specified `_oid` value is currently active. If the `_oid` is not active, then an `ObjectNotActive` exception is raised.

This method requires that the `RETAIN` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
`_oid`	The ObjectId of the object for which a reference is to be returned.

```
PortableServer::Servant
  id_to_servant(PortableServer::ObjectId& _oid);
```

This method has three behaviors:

- If the POA has the `RETAIN` policy present and the specified `_oid` is in the Active Object Map, then it returns the servant associated with that object in the Active Object Map.
- If the POA has the `USE_DEFAULT_SERVANT` policy present and a default servant has been registered with the POA, it returns the default servant.
- Otherwise, an `ObjectNotActive` exception is raised.

This method requires that the `USE_DEFAULT_SERVANT` policy be present with the POA; if neither policy is present, a `WrongPolicy` exception is raised.

Parameter	Description
`_oid`	The ObjectId of the object for which a servant is to be returned.

```
PortableServer::Servant
  reference_to_servant(CORBA::Object_ptr _reference);
```

This method has three behaviors:

- If the POA has the `RETAIN` policy and the specified `_reference` is present in the Active Object Map, then it returns the servant associated with that object in the Active Object Map.
- If the POA has the `USE_DEFAULT_SERVANT` policy present and a default servant has been registered with the POA, then it returns the default servant.

- Otherwise, it raises an `ObjectNotActive` exception.

This method requires the `RETAIN` or `USE_DEFAULT_SERVANT` policies to be present; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
`_reference`	The object for which a servant is to be returned.

```
PortableServer::ObjectId*
  reference_to_id(CORBA::Object_ptr _reference);
```

This method returns the ObjectId value encapsulated by the specified `_reference`. The invocation is valid only if the `_reference` was created by the POA on which the method is called. If the `_reference` was not created by the POA, a `WrongAdapter` exception is raised. The object denoted by the `_reference` parameter does not have to be active for this method to succeed.

Though the IDL specifies that a `WrongPolicy` exception may be raised by this method, it is simply declared for possible future extension.

Parameter	Description
`_reference`	The object for which an ObjectID is to be returned.

```
PortableServer::ObjectId*
  servant_to_id(PortableServer::Servant _p_servant);
```

This method has four possible behaviors:

- If the POA has the `UNIQUE_ID` policy present and the specified `_p_servant` is active, then the ObjectId associated with the `_p_servant` is returned.
- If the POA has the `IMPLICIT_ACTIVATION` policy present and either the POA has the `MULTIPLE_ID` policy present or the specified `_p_servant` is not active, then the `_p_servant` is activated using the POA-generated ObjectId and the repository interface id associated with the `_p_servant`, and that ObjectId is returned.
- If the POA has the `USE_DEFAULT_SERVANT` policy present, the specified `_p_servant` is the default servant, then the ObjectId associated with the current invocation is returned.
- Otherwise, a `ServantNotActive` exception is raised.

This method requires that the `USE_DEFAULT_SERVANT` policy or a combination of the `RETAIN` policy and either the `UNIQUE_ID` or `IMPLICIT_ACTIVATION` policies be present; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
`_p_servant`	The Servant for which the ObjectId to be returned is desired.

```
CORBA::Object_ptr
  servant_to_reference(PortableServer::Servant
  _p_servant);
```

This method has four possible behaviors:

- If the POA has both the `RETAIN` and the `UNIQUE_ID` policies present and the specified `_p_servant` is active, then an object reference encapsulating the information used to activate the servant is returned.
- If the POA has both the `RETAIN` and the `IMPLICIT_ACTIVATION` policies present and either the POA has the `MUTLIPLE_ID` policy or the specified

_p_servant is not active, then the _p_servant is activated using a POA-generated ObjectId and repository interface id associated with the _p_servant, and a corresponding object reference is returned.

- If this method was invoked in the context of executing a request on the specified _p_servant, the reference associated with the current invocation is returned.
- Otherwise, a ServantNotActive exception is raised.

This method requires the presence of the RETAIN policy and either the UNIQUE_ID or IMPLICIT_ACTIVATION policies if invoked outside the context of a method dispatched by the POA. If this method is not invoked in the context of executing a request on the specified _p_servant and one of these policies is not present, then a WrongPolicy exception is raised.

Parameter	Description
_p_servant	The Servant for which a reference is to be returned.

```
void set_servant(PortableServer::Servant _p_servant);
```

This method sets the default Servant associated with the POA. The specified Servant will be used for all requests for which no servant is found in the Active Object Map.

This method requires that the USE_DEFAULT_SERVANT policy be present with the POA; otherwise, a WrongPolicy exception is raised.

Parameter	Description
_p_servant	The Servant to be used as the default associated with the POA.

```
void set_servant_manager
  (PortableServer::ServantManager_ptr _imagr);
```

This method sets the default ServantManager associated with the POA. This method may only be invoked after a POA has been created. Attempting to set the ServantManager after one has already been set raises a BAD_INV_ORDER exception.

This method requires that the USE_SERVANT_MANAGER policy be present with the POA; otherwise, a WrongPolicy exception is raised.

Parameter	Description
_imgr	The ServantManager to be used as the default used with the POA.

```
PortableServer::AdapterActivator_ptr the_activator();
```

This method returns the AdapterActivator associated with the POA. When a POA is created, it does not have an AdapterActivator (i.e., the attribute is null). It is system dependent whether a root POA has an activator and the application can assign one as it wishes.

```
void the_activator(PortableServer::AdapterActivator_ptr
  _val);
```

This method sets the AdapterActivator object associated with the POA to the one specified.

Parameter	Description
_val	The ActivatorAdapter to be associated with the POA.

```
char* the_name();
```

This method returns the read-only attribute which identifies the POA relative to its parent.This attribute is assigned at POA creation. The name of the root POA is system dependent and should not be relied upon by the application.

```
PortableServer::POA_ptr the_parent();
```

This method returns a pointer to the POA's parent POA. The parent of the root POA is null.

```
Portableserver::POAManager_ptr the_POAManager();
```

This method returns the read-only attribute which is a a pointer to the POAManager associated with the POA.

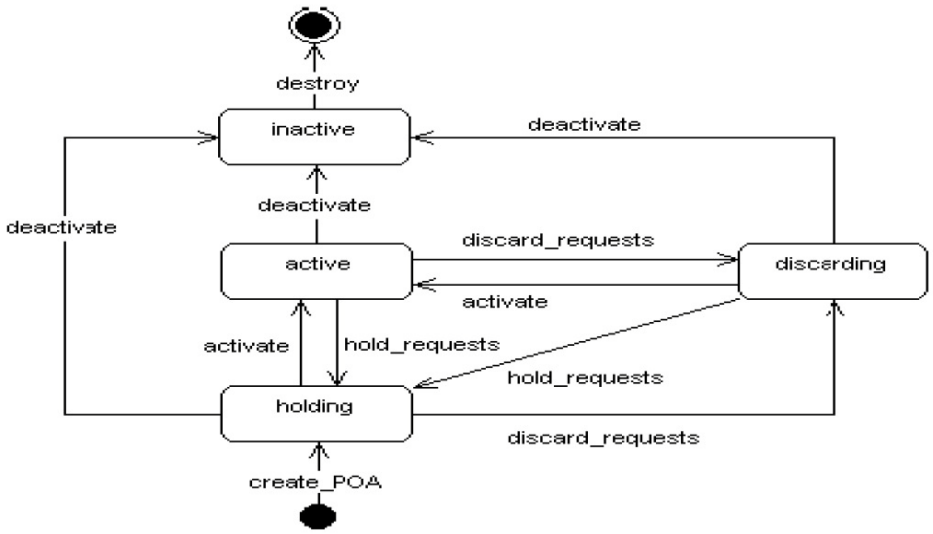
PortableServer::POAManager

Each POA has an associated POA manager which in turn may be associated with one or more POA objects. A POA manager encapsulates the processing state of the POAs with which it is associated.

There are four possible states which a POA manager can be in:

- active
- inactive
- holding
- discarding

A POA manager is created in the holding state. The following illustrates the state which a POA manager transitions to based on the method called.



Include file

You should include the file **poa_c.hh** when using this class.

PortableServer::POAManager methods

void **activate**();

This method changes the state of the POA manager to active, which enables the associated POAs to process requests. If invoked while the POA manager is in the inactive state, the `AdapterInactive` exception is raised.

```
void deactivate(CORBA::Boolean
  _etherealize_objects,CORBA::Boolean
  _wait_for_completion);
```

This method changes the state of the POA manager to inactive, which causes the associated POAs to reject requests that have not begun to be executed, as well as any new requests. If invoked while the POA manager is in the inactive state, the `AdapterInactive` exception is raised.

After the state changes, if the `etherealize_objects` parameter is

- TRUE—the POA manager causes all associated POAs that have the RETAIN and USE_SERVANT_MANAGER policies to perform the etherealize operation on the associated servant manager for all active objects.
- FALSE—the etherealize operation is not called. The purpose is to provide developers with a means to shut down POAs in a crisis (for example, unrecoverable error) situation.

If the `wait_for_completion parameter` is FALSE, this operation returns immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) and, in the case of a TRUE etherealize_objects, all invocations of etherealize have completed for POAs having the RETAIN and USE_SERVANT_MANAGER policies. If the parameter is TRUE and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA, the BAD_INV_ORDER exception is raised and the state is not changed.

```
void discard_requests(CORBA::Boolean
  _wait_for_completion);
```

This method changes the state of the POA manager to discarding, which causes the associated POAs to discard incoming requests. In addition, any requests that have been queued but have not started executing are discarded. When a request is discarded, a `TRANSIENT` system exception is returned to the client. If invoked while the POA manager is in the inactive state, the `AdapterInactive` exception is raised.

If the wait_for_completion parameter is FALSE, this operation returns immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than discarding. If the parameter is TRUE and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER exception is raised and the state is not changed.

```
void hold_requests(CORBA::Boolean
  _wait_for_completion);
```

This method changes the state of the POA manager to holding, which causes the associated POAs to queue incoming requests. Any requests that have been queued but are not executing will continue to be queued while in the holding state. If invoked while the POA manager is in the inactive state, the `AdapterInactive` exception is raised.

Principal

If the wait_for_completion parameter is FALSE, this operation will return immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) and, in the case of a TRUE etherealize_objects, all invocations of etherealize have completed for POAs having the RETAIN and USE_SERVANT_MANAGER policies. If the parameter is TRUE and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER exception is raised and the state is not changed.

Note

This feature is deprecated since VisiBroker 4.0.

```
typedef OctetSequence Principal
```

The `Principal` is used to represent the client application on whose behalf a request is being made. An object implementation can accept or reject a bind request, based on the contents of the client's `Principal`.

Include file

You should include the file **corba.h** when using this typedef.

Principal methods

The `BOA` class provides the `get_principal` method, described in "CORBA::Principal_ptr get_principal(CORBA::Object_ptr obj, CORBA::Environment_ptr env=NULL)" on page 5-5, which returns a pointer to the `Principal` associated with an object. The `Object` class also provides methods for getting and setting the `Principal`.

PortableServer::RefCountServantBase

```
class RefCountServantBase : public ServantBase
```

This class can be used as a standard servant reference counting mix-in class, rather than the `PortableServer::ServantBase` class which is to be used with inheritance class. (Also see "PortableServer::ServantBase".)

Include file

You should include the file **poa_c.hh** when using this class.

PortableServer::RefCountServantBase methods

```
void _add_ref();
```

This method increments the reference count by one.You can override this method from the base class to provide true reference counting.

```
void _remove_ref();
```

This method decrements the reference count by one. You can override this method from the base class to provide true reference counting.

PortableServer::ServantActivator

```
class PortableServer::ServantActivator : public
  PortableServer::ServantManager
```

If the POA has the `RETAIN` policy present, then it uses servant managers that are `PortableServer::ServantActivator` objects.

Include file

You should include the file **poa_c.hh** when using this class.

PortableServer::ServantLocator methods

```
void etherealize(PortableServer::ObjectId&
  oid,PortableServer::POA_ptr adapter,
  PortableServer::Servant serv,CORBA::Boolean
  cleanup_in_progress, CORBA::Boolean
  remaining_activations);
```

This method is called by the specified `adapter` whenever a servant for an object (the specified `oid`) is deactivated, assuming that the `RETAIN` and `USE_SERVANT_MANAGER` policies are present.

Parameter	Description
`oid`	The object id of the object whose servant is to be deactivated.
`adapter`	The POA in whose scope the object was active.
`serv`	The servant which is to be deactivated.
`cleanup_in_progress`	If set to `TRUE`, the reason for the invocation of the method is either that the deactivate or destroy method was called with the `etherealize_objects` parameter set to `TRUE`; otherwise, the method was called for other reasons.
`remaining_activations`	If the specified `serv` is associated with other objects in the specified `adapter` it is set to `TRUE`; otherwise it is `FALSE`.

```
PortableServer::Servant incarnate(const
  PortableServer::ObjectId& oid,
  PortableServer::POA_ptr adapter);
```

This method is called by the POA whenever the POA receives a request for an inactive object (the specified `oid`) assuming that the `RETAIN` and `USE_SERVANT_MANAGER` policies are present.

The user supplies a servant manager implementation which is responsible for locating and creating an appropriate servant that corresponds to the specified `oid` value. The method returns a servant, which is also entered into the Active Object map. Any further requests for the active object are passed directly to the servant associated with it without invoking the servant manager.

If this method returns a servant that is already active for a different object id and if the POA also has the `UNIQUE_ID` policy present, then it raises the `OBJ_ADAPTER` exception.

Parameter	Description
`oid`	The object id of the object whose servant is to be activated.
`adapter`	The POA in whose scope the object is to be activated.

PortableServer::ServantBase

```
class PortableServer::ServantBase
```

The `Portable::ServantBase` class is the base class for your server application.

Include file

You should include the file **poa_c.hh** when using this class.

PortableServer::ServantBase methods

```
void _add_ref();
```

This method adds a reference count for this servant. It should be overridden to provide reference counting functionality for classes derived from this class as the default implementation does nothing.

```
PortableServer::POA_ptr _default_POA();
```

This method returns an Object reference to the root POA of the default ORB in the current process, (i.e., the same return value as an invocation of `ORB::resolve_initial_references("RootPOA")` on the default ORB. Classes derived from the `PortableServer::ServantBase` class may override this method to return the POA of their choice, if desired.

```
CORBA::InterfaceDef_ptr _get_interface();
```

This method returns a pointer to this object's interface definition. See "InterfaceDef methods" for more information.

```
CORBA::Boolean _is_a(const char *rep_id);
```

This method returns `TRUE` if this servant implements the interface associated with the repository id. Otherwise, it returns `FALSE`.

Parameter	Description
rep_id	The repository identifier against which to check.

```
void _remove_ref();
```

This method removes a reference count for this servant. It should be overridden to provide reference counting functionality for classes derived from this class as the default implementation does nothing.

PortableServer::ServantLocator

```
class PortableServer::ServantLocator : public
  PortableServer::ServantManager
```

When the POA has the `NON_RETAIN` policy present, it uses servant managers which are `PortableServer::ServantLocator` objects. The servant returned by the servant manager will be used only for a single request.

Because the POA knows that the servant returned by the servant manager will be used only for a single request, it can supply extra information for the servant manager's methods and the servant manager's pair of methods may do something different than a `PortableServer::ServantLocator` servant manager.

Include file

You should include the file **poa_c.hh** when using this class.

PortableServer::ServantLocator methods

```
PortableServer::Servant preinvoke(const
  PortableServer::ObjectId& oid,
  PortableServer::POA_ptr adapter,const char*
  operation, Cookie& the_cookie);
```

This method is called by the POA whenever the POA receives a request for an object that is not currently active, assuming that the `NON_RETAIN` and `USE_SERVANT_MANAGER` policies are present.

The user-supplied implementation of the servant manager is responsible for locating or creating an appropriate servant that corresponds to the specified `oid` value if possible.

Parameter	Description
`oid`	The object id value that is associated with the incoming request.
`adapter`	The POA in which the object is to be activated.
`operation`	The name of the operation which will be called by the POA when the servant is returned.
`the_cookie`	An opaque value which can be set by the servant manager to be used later in the `postinvoke` method.

```
void postinvoke(const PortableServer::ObjectId& oid,
  PortableServer::POA_ptr adapter,const char*
  operation, Cookie the_cookie,PortableServer::Servant
  the_servant)
```

If the POA has the `NON_RETAIN` and `USE_SERVANT_MANAGER` policies present, this method is called whenever a servant completes a request. This method is considered to be part of the request on an object (that is, if the method finishes normally, but postinvoke raises a system exception, then the method's normal return is overridden; and the request completes with the exception).

Destroying a servant that is known to a POA can lead to undefined results.

Parameter	Description
`oid`	The `ObjectId` value that is associated with the incoming request.
`adapter`	The POA in which the object is to be activated.
`operation`	The name of the operation which will be called by the POA when the servant is returned.
`the_cookie`	An opaque value which can be set by the servant manager in the
`preinvoke .`	method for use in this method
`the_servant`	The servant associated with the object.

PortableServer::ServantManager

`class` **`PortableServer::ServantManager`**

Servant managers are associated with Portable Object Adapters (POAs). A servant manager allows a POA to activate objects on demand when the POA receives a request targeted for an inactive object.

The PortableServer::ServantManager class has no methods; rather it is the base class for two other classes: the `PortableServer::ServantActivator` and the `Portableserver::ServantLocator` classes. For more details, see "PortableServer::ServantActivator" and "PortableServer::ServantLocator". The use of these two classes depends on the POA's policies: `RETAIN` for the `PortableServer::ServantActivator` and `NON_RETAIN` for the `Portableserver::ServantLocator`.

Include file

You should include the file **poa_c.hh** when using this class.

Environment

`class` **`CORBA::Environment`**

The `Environment` class is used for reporting and accessing both system and user exceptions on platforms where C++ language exceptions are not supported. When an interface specifies that user exceptions may be raised by the object's methods, the `Environment` class becomes an explicit parameter of that method. If an interface does not raise any exceptions, the `Environment` class is an implicit parameter and is only used for reporting system exceptions. If an `Environment` object is not passed from the client to a stub, the default of per-object `Environment` is used.

Multithreaded applications have a global `Environment` object for each thread that is created. Applications that are not multithreaded have just one global `Environment` object.

Include file

You should include the **corba.h** file when you use this class.

```
Environment();
```

This method creates an `Environment` object. This is equivalent to calling the `ORB::create_environment` method.

```
static COBRA::Environment&
  CORBA::current_environment();
```

This static method returns a reference to the global `Environment` object for the application process. In multithreaded applications, it returns the global `Environment` object for this thread.

```
void exception(COBRA::Exception *exp);
```

This method records the `Exception` object passed as an argument. The `Exception` object must be dynamically allocated because the specified object will assume ownership of the `Exception` object and will delete it when the `Environment` itself is deleted. Passing a NULL pointer to this method is equivalent to invoking the `clear` method on the `Environment`.

Parameter	Description
`exp`	A pointer to a dynamically allocated Exception object to be recorded for this Environment.

```
CORBA::Exception *exception() const;
```

This method returns a pointer to the `Exception` currently recorded in this `Environment`. You must not invoke `delete` on the `Exception` pointer returned by this call. If no `Exception` has been recorded, a `NULL` pointer will be returned.

```
void clear();
```

This method will cause this `Environment` to delete any `Exception` object that it holds. If this object holds no exception, this method has no effect.

SystemException

```
class CORBA::SystemException : public CORBA::Exception
```

The `SystemException` class is used to report standard system errors encountered by the ORB or by the object implementation. This class is derived from the `Exception` class, described in "Exception" on page 5-11, which provides methods for printing the name and details of the exception to an output stream.

`SystemException` objects include a completion status which indicates if the operation that caused the exception was completed. `SystemException` objects also have a minor code that can be set and retrieved.

Include file

The **corba.h** file should be included when you use this class.

SystemException methods

```
CORBA::SystemException(CORBA::ULong minor = 0,
  CORBA::CompletionStatus status =
  CORBA::COMPLETED_NO);
```

This method creates a `SystemException` object with the specified properties.

Parameter	Description
`minor`	The minor code.

`status`	The completion status, one of:
	`CORBA::COMPLETED_YES`
	`CORBA::COMPLETED_NO`
	`CORBA::COMPLETED_MAYBE`

```
CORBA::CompletionStatus completed() const;
```

This method returns `TRUE` if this object's completion status is set to `COMPLETED_YES`.

```
void completed(CORBA::CompletionStatus status);
```

This method sets the completion status for this object.

Parameter	Description

`status`	The completion status, one of:
	`CORBA::COMPLETED_YES`
	`CORBA::COMPLETED_NO`
	`CORBA::COMPLETED_MAYBE`

```
CORBA::ULong minor() const;
```

This method returns this object's minor code.

```
void minor(CORBA::ULong val);
```

This method sets the minor code for this object.

Parameter	Description
`val`	The minor code.

```
static CORBA::SystemException
  *_downcast(CORBA::Exception *exc);
```

This method attempts to downcast the specified `Exception` pointer to a `SystemException` pointer. If the supplied pointer points to a `SystemException` object or an object derived from `SystemException`, a pointer to the object is returned. If the supplied pointer does not point to a `SystemException` object, a `NULL` pointer is returned.

Note

The reference count for the `Exception` object is not incremented by this method.

Parameter	Description
`exc`	An Exception pointer to be downcasted.

Exception name	Description
`BAD_INV_ORDER`	Routine invocations out of order.
`BAD_OPERATION`	Invalid operation.
`BAD_CONTEXT`	Error processing context object.
`BAD_PARAM`	An invalid parameter was passed.
`BAD_TYPECODE`	Invalid typecode.
`COMM_FAILURE`	Communication failure.
`DATA_CONVERSION`	Data conversion error.
`FREE_MEM`	Unable to free memory
`IMP_LIMIT`	Implementation limit violated.
`INITIALIZE`	ORB initialization failure.
`INTERNAL`	ORB internal error.
`INTF_REPOS`	Error accessing interface repository.
`INV_FLAG`	Invalid flag was specified.
`INV_INDENT`	Invalid identifier syntax.
`INV_OBJREF`	Invalid object reference specified.
`MARSHAL`	Error marshalling parameter or result.
`NO_IMPLEMENT`	Operation implementation not available.
`NO_MEMORY`	Dynamic memory allocation failure.
`NO_PERMISSION`	No permission for attempted operation.
`NO_RESOURCES`	Insufficient resources to process request.
`NO_RESPONSE`	Response to request not yet available.
`OBJ_ADAPTOR`	Failure detected by object adaptor.
`OBJECT_NOT_EXIST`	Object is not available.
`PERSIST_STORE`	Persistent storage failure.
`TRANSIENT`	Transient failure.
`UNKNOWN`	Unknown exception

UserException

`class` **`CORBA::UserException`** `:` `public` **`CORBA::Exception`**

The `UserException` base class is used to derive the user exceptions that your object implementations may want to raise. This class is derived from the `Exception` class, described in “Exception”, which provides methods for printing the name and details of the exception to an output stream.

Include file

The **corba.h** file should be included when you use this class.

UserException methods

CORBA::UserException();

To his method creates a `UserException` object with the specified properties.

Parameter	Description
minor	The minor code.

`status`	The completion status, one of:
	`CORBA::COMPLETED_YES`
	`CORBA::COMPLETED_NO`
	`CORBA::COMPLETED_MAYBE`

UserException derived classes

```
class AdapterAlreadyExists :
  public CORBA::UserException class AdapterInactive :
  public CORBA::UserException class AdapterNonExistent :
  public CORBA::UserException class InvalidPolicy :
  public CORBA::UserException class NoServant :
  public CORBA::UserException class ObjectAlreadyActive :
  public CORBA::UserException class ObjectNotActive  :
  public CORBA::UserException class ServantAlreadyActive
  :
  public CORBA::UserException class ServantNotActive :
  public  CORBA::UserException class WrongAdapter :
  public CORBA::UserException
```

TCKind

enum **TCKind**

This enumeration describes the various types that a `TypeCode` object, described in "TypeCode", may represent.The values are shown in the following table.

Name	Method
`tk_null`	NULL
`tk_void`	void
`tk_short`	short
`tk_long`	long
`tk_ushort`	unsigned short
`tk_ulong`	unsigned long
`tk_float`	float
`tk_double`	double
`tk_boolean`	boolean
`tk_char`	char
`tk_octet`	octet string
`tk_any`	Any
`tk_TypeCode`	TypeCode
`tk_Principal`	Principal
`tk_objref`	object reference
`tk_struct`	struct
`tk_union`	union

Name	Method
tk_enum	enum
tk_string	string
tk_sequence	sequence
tk_array	array
tk_alias	alias
tk_except	exception
tk_longlong	long long
tk_ulonglong	unsigned long long
tk_longdouble	long double
tk_wchar	Unicode character
tk_wstring	Unicode string
tk_fixed	fixed type
tk_value	value
tk_value_box	value box
tk_native	native type
tk_abstract_interface	abstract interface

TypeCode

`class` **`CORBA::TypeCode`**

The `TypeCode` class represents the various types that can be defined in IDL. Type codes are most often used to describe the type of value being stored in an `Any` object, described in"CORBA::Any();". Type codes may also be passed as parameters to method invocations.

`TypeCode` objects can be created using the various `CORBA::ORB.create_<type>_tc` methods, whose description begins in"Object". You may also use the constructors listed here.

Include file

The **corba.h** file should be included when you use this class.

TypeCode constructors

CORBA::TypeCode(CORBA::TCKind **kind**, CORBA::Boolean **is_constant**);

This method constructs a TypeCode object for types that do not require any additional parameters. A BAD_PARAM exception is raised if kind is not a valid type for this constructor.

Parameter	Description
kind	Describes the type of object being represented. Must be one of the following: CORBA::tk_null, CORBA::tk_void, CORBA::tk_short, CORBA::tk_long, CORBA::tk_ushort, CORBA::tk_ulong, CORBA::tk_float, CORBA::tk_double, CORBA::tk_boolean, CORBA::tk_char, CORBA::tk_octet, CORBA::tk_any, CORBA::tk_TypeCode, CORBA::tk_Principal, CORBA::tk_longlong, CORBA::tk_ulonglong, CORBA::tk_longdouble, or CORBA::tk_wchar, CORBA::tk_fixed, CORBA::tk_value, CORBA::tk_value_box, CORBA::native, CORBA::tk_abstract_interface.
is_constant	If true, the type being represented is to be considered a constant. Otherwise, the object is not a constant.

TypeCode methods

CORBA::TypeCode_ptr **content_type**() const;

This method returns the TypeCode of the elements in a sequence or array. It also will return the type of an alias. A BadKind exception is raised if this object's kind is not CORBA::tk_sequence, CORBA::tk_array, or CORBA::tk_alias.

CORBA::Long **default_index**() const;

This method returns the default index of a TypeCode representing a union. If this object's kind is not CORBA::tk_union, a BadKind exception is raised.

CORBA::TypeCode_ptr **discriminator_type**() const;

This method returns the discriminator type of a TypeCode representing a union. If this object's kind is not CORBA::tk_union, a BadKind exception is raised.

CORBA::Boolean **equal**(CORBA::TypeCode_ptr **tc**) const;

This method compares this object with the specified TypeCode. If they match in every respect, true is returned. Otherwise, false is returned.

Parameter	Description
tc	The object to be compared to this object.

const char* **id**() const;

This method returns the repository identifier of the type being represented by this object. If the type being represented does not have a repository

identifier, a `BadKind` exception is raised. Types that have a repository identifier include:

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_alias
- CORBA::tk_except
- CORBA::tk_objref

`CORBA::TCKind` **`kind`**`() const`

This method returns this object's kind.

`CORBA::ULong` **`length`**`() const;`

This method returns the length of the string, sequence, or array represented by this object. A `BadKind` exception is raised if this object's kind is not `CORBA::tk_string`, `CORBA::tk_sequence`, or `CORBA::tk_array`.

`CORBA::ULong` **`member_count`**`() const;`

This method returns the member count of the type being represented by this object. If the type being represented does not have members, a `BadKind` exception is raised. Types that have members include:

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_except

`CORBA::Any_ptr` **`member_label`**`(CORBA::ULong` **`index`**`) const;`

This method returns the label of the member with the specified index from a `TypeCode` object for a `union`. If this object's kind is not `CORBA::tk_union`, a `BadKind` exception is raised. If the index is invalid, a `Bounds` exception is raised.

Parameter	Description
`index`	The label of the union member whose type is to be returned. This index is zero-based.

`const char *`**`member_name`**`(CORBA::ULong` **`index`**`) const;`

This method returns the name of the member with the specified index from the type being represented by this object. If the type being represented does not have members, a `BadKind` exception is raised. If the index is invalid, a `Bounds` exception is raised. Types that have members include:

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_except

Parameter	Description
`index`	The index of the member whose name is to be returned. This index is zero-based.

```
CORBA::TypeCode_ptr member_type(CORBA::ULong index)
  const;
```

This method returns the type of the member with the specified index from the type being represented by this object. If the type being represented does not have members with types, a `BadKind` exception is raised. If the index is invalid, a `Bounds` exception is raised. Types that have members include:

- CORBA::tk_union
- CORBA::tk_except

Parameter	Description
`index`	The index of the member whose type is to be returned. This index is zero-based.

```
const char *name() const;
```

This method returns the name of the type being represented by this object. If the type being represented does not have a name, a `BadKind` exception is raised. Types that have a name include:

- CORBA::tk_objref
- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_alias
- CORBA::tk_except

```
static CORBA::TypeCode_ptr
  _duplicate(CORBA::TypeCode_ptr obj)
```

This static method duplicates the specified `TypeCode`.

Parameter	Description
obj	The object to be duplicated.

```
static CORBA::TypeCode_ptr _nil()
```

This static method returns a `NULL TypeCode` pointer that can be used for initialization purposes.

```
static void _release(CORBA::TypeCode_ptr obj)
```

This static method decrements the reference count to the specified object. If the reference count is zero, it also frees all memory that it is managing and then deletes the object.

Parameter	Description
obj	The object to be released.

```
CORBA::Boolean equivalent (CORBA_TypeCode_ptr tc) const
```

The `equivalent` operation is used by the ORB when determing the type equivalence for values stored in an IDL `Any`

```
CORBA_TypeCode_ptr get_compact_typecode() const
```

The `get_compact_code` operation strips out all optional name & member name fields, but it leaves all alias typecodes intact.

```
virtual CORBA::Visibility
  member_visibility(CORBA::ULong index) const;
```

The `member_visibility` operation can only be invoked and non_boxed valuetype TypeCodes. It returns the Visibility of the valuetype member identified by index.

```
virtual CORBA::ValueModifier type_modifier() const;
```

The `type_modifier` operations can be invoked on non-boxed valuetype TypeCodes. This method returns the ValueModifier that applies to the valuetype represented by the target Typecode.

```
virtual CORBA::TypeCode_ptr concrete_base_types()
```

The `concrete_base_types` operations can be invoked on non-boxed valuetype TypeCodes. If the value represented by the target TypeCode has a concrete base valuetype, this method returns a TypeCode for the concrete base, otherwise it returns a nil TypeCode reference.

SupportServices

```
class SupportServices
```

This class provides support for user registration of VisiBroker Services.

```
static SupportServices * instance();
```

This method returns a reference to the SupportServices instance, which can then be used to register VisiBroker services via the `register_service_object()` method.

```
void register_service_object(const char* objectId,
  CORBA_Object_ptr obj);
```

This method is provided to allow for the registration of any VisiBroker Services. The Service provider will specify their Service Name along with a Object pointer to the service object. Users of the Service can then acquire a handle to the Service by calling `orb->resolve_initial_references("service_name")`.

Parameter	Description
`obj`	CORBA Object which implements the Service.

Include file

The **supportServicesLib.h** file should be included when you use this class.

Dynamic Interfaces and Classes

This chapter describes the classes that support the Dynamic Invocation Interface used by client applications, and the Dynamic Skeleton Interface used by object servers.

Note

The Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI) is not supported as part of the "minimum CORBA" version of VisiBroker-RT for C++ (that is, `liborb_min.o`).

The "minimum CORBA" OMG specification identifies dynamic functionality which should be excluded from an ORB, in an effort to reduce the ORB footprint.

For details, see the minimum CORBA specification document, OMG document number **orbos/ 98-08-04**. This document is available for download using the URL ftp://ftp.omg.org/pub/docs/orbos/98-08-04.pdf.

Author note: the link above doesn't work

Any

`class CORBA::Any`

This class is used to represent an IDL type so that its value may be passed in a type-safe manner. Objects of this class have a pointer to a `TypeCode` that defines the object's type and a pointer to the value associated with the object. Methods are provided to construct, copy, and destroy an object as well as to initialize and query the object's type and value. In addition, streaming operators are provided to read and write the object to a stream.

Include file

The **corba.h** file should be included when you use this structure.

Any methods

`CORBA::Any``();`

This is the default constructor that creates an empty `Any` object.

`CORBA::Any``(const CORBA::Any& val);`

This is a copy constructor that creates an `Any` object that is a copy of the specified target.

Parameter	Description
`val`	The object to be copied.

```
CORBA::Any(CORBA::TypeCode_ptr tc, void *value,
  CORBA::Boolean release = 0);
```

This constructor creates an `Any` object initialized with the specified value and TypeCode.

Parameter	Description
`tc`	The `TypeCode` of the value contained by this Any.
`value`	The value contained by this `Any`.
`release`	If set to `TRUE`, the memory associated with this `Any` object's value will be released when this `Any` object is destroyed.

```
static CORBA::Any_ptr _duplicate(CORBA::Any_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to it.

Parameter	Description
`ptr`	The `Any` to be duplicated.

```
static CORBA::Any_ptr _nil();
```

This static method returns a NULL pointer that can be used for initialization purposes.

```
static void _release(CORBA::Any_ptr *ptr);
```

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
`ptr`	The `Any` to be released.

Insertion operators

```
void   operator<<=(CORBA::Short);
    void operator<<=(CORBA::UShort);
    void operator<<=(CORBA::Long);
    void operator<<=(CORBA::ULong);
    void operator<<=(CORBA::Float);
    void operator<<=(CORBA::Double);
    void operator<<=(const CORBA::Any&);
    void operator<<=(const char *);
    void operator<<=(CORBA::LongLong);
    void operator<<=(CORBA::ULongLong);
    void operator<<=(CORBA::LongDouble);
```

These operators will initialize this object with the specified value, automatically setting the appropriate `TypeCode` for the value. If this `Any` object was constructed with the `release` flag set to true, the value previously stored in this `Any` object will be released before the new value is assigned.

```
void operator<<=(CORBA::TypeCode_ptr tc);
```

Extraction operators

```
CORBA::Boolean operator>>=(CORBA::Short&) const;
    CORBA::Boolean operator>>=(CORBA::UShort&) const;
    CORBA::Boolean operator>>=(CORBA::Long&) const; CORBA::Boolean
    operator>>=(CORBA::ULong&) const; CORBA::Boolean
    operator>>=(CORBA::Float&) const; CORBA::Boolean
    operator>>=(CORBA::Double&) const; CORBA::Boolean
    operator>>=(CORBA::Any&) const; CORBA::Boolean
    operator>>=(char *&) const; CORBA::Boolean
    operator>>=(CORBA::LongLong&) const; CORBA::Boolean
    operator>>=(CORBA::ULongLong&) const; CORBA::Boolean
    operator>>=(CORBA::LongDouble&) const;
```

These operators store the value from this object into the specified target. If the `TypeCode` of the target does not match the `TypeCode` of the stored value, false is returned and no value is extracted. Otherwise, the stored value will be assigned to the target and true will be returned.

```
CORBA::Boolean operator>>=(CORBA::TypeCode_ptr& tc)
  const;
```

This method extracts the `TypeCode` of the value stored in this object.

Parameter	Description
`tc`	The object where the typecode for this `Any` is to be stored.

ContextList

`class` **CORBA::ContextList**

This class contains a list of contexts that may be associated with an operation request. See “Request”.

ContextList methods

CORBA::ContextList();

This method constructs an empty `Context` list.

~CORBA::ContextList();

This method is the default destructor.

`void` **add**(`const char` ***ctx**);

This method adds the specified context to this object’s list.

Parameter	Description
`ctx`	The context to be added to the list.

```
void add_consume(char *ctx);
```

This method adds the specified context code to this object's list. Ownership of the passed argument is assumed by this `ContextList`. You should not attempt to access or free the argument after invoking this method.

Parameter	Description
`ctx`	The context to be added to the list.

```
CORBA::ULong count() const;
```

This method returns the number of items currently stored in the list.

```
const char *item(CORBA::Long index);
```

This method returns a pointer to the context that is stored in the list at the specified index. If the index in invalid, a `NULL` pointer is returned. You should not attempt to access or free the argument after invoking this method. To remove a context from the list, use the remove method.

Parameter	Description
`index`	The index of the context to be returned. The index is zero-based.

```
void  remove(CORBA::long index);
```

This method removes from the list the context with the specified index. If the index is invalid, no removal will occur.

Parameter	Description
`index`	The index of the context to be removed. The index is zero-based.

```
static CORBA::ContextList_ptr
  _duplicate(CORBA::ContextList_ptr ptr);
```

This static method increments the reference count for the object and then returns a pointer to it.

Parameter	Description
`ptr`	The object to be duplicated.

```
static CORBA::ContextList_ptr _nil();
```

This static method returns a NULL pointer that can be used for initialization purposes.

```
static void _release(CORBA::ContextList *ptr);
```

This static method decrements the reference count for this object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
`ptr`	The object to be released.

DynamicImplementation

```
class PortableServer::DynamicImplementation : public
  PortableServer::ServantBase
```

This base class is used derive object implementations that wish to use the Dynamic Skeleton Interface instead of a skeleton class generated by the IDL compiler. You must provide an implementation of the `invoke` and `_primary-interface()` methods when deriving from this class.

DynamicImplementation methods

```
virtual void invoke(CORBA::ServerRequest_ptr request) =
  0;
```

This method will be invoked by the POA whenever client operation requests are received for your object implementation. You must provide an implementation of this method which validates the `ServerRequest` object's contents, performs the necessary processing to fulfill the request, and returns the results to the client. For more information on the `ServerRequest` class, see "ServerRequest".

Parameter	Description
`request`	The `ServerRequest` object that represents the client's operation request.

DynAny

```
virtual CORBA::RepositoryId _primary_interface(const
  PortableServer::ObjectId& oid PortableServer::POA_ptr
  poa) const;
```

The `_primary_interface()` method will be invoked only by the POA in the context of serving a CORBA request. Invoking this method in other circumstances may lead to unpredictable results. The `_primary_interface` method receives an `ObjectId` value and a `POA_ptr` as input parameters and returns a valid RepositoryId representing the most-derived interface for that `oid`.

```
class DynamicAny::DynAny : public CORBA::Pseudo Object
```

A `DynAny` object is used by a client application or server to create and interpret data types at run-time which were not defined at compile-time. A `DynAny` may contain a basic type (such as a `boolean`, `int`, or `float`) or a complex type (such as s `struct` or `union`). The type contained by a `DynAny` is defined when it is created and may not be changed during the lifetime of the object.

A `DynAny` object may represent a data type as one or more components, each with its own value. The `next`, `seek`, `rewind`, and `current_component` methods are provided to help you navigate through the components.

A `DynAny` object is created by a `DynAnyFactory` object by first calling `ORB::resolve_initial_references`("`DynAnyFactory`"). The factory is then used to create basic or complex types. The `DynAnyFactory` belongs to the `DynamincAny` module.

`DynAny` objects for basic types are created using the `DynAnyFactory::create_ dyn_any_from_type_code` method, described in "Core Interfaces and Classes".

A `DynAny` object may also be created and initialized from an `Any` object using the `DynAnyFactory::create_dyn_any` method, also described in "Core Interfaces and Classes".

The following interfaces are derived from `DynAny` and provide support for constructed types that are dynamically managed.

Constructed type	Interface
Array	DynArray
Enumeration	DynEnum
Sequence	DynSequence
Structure	DynStruct
Union	DynUnion

Include file

The **dynany.h** file should be included when you use this structure.

Important usage restrictions

`DynAny` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `DynAny::to_any` method to convert a `DynAny` object to an `Any`, which can be used as a parameter.

DynAny methods

```
void assign(CORBA::DynAny_ptr dyn_any);
```

Initializes the value in this object from the specified `DynAny`.

A type mismatch exception is raised if the type contained in the `Any` does not match the type contained by this object.

```
CORBA::DynAny_ptr copy();
```

Returns a copy of this object.

```
virtual CORBA::DynAny_ptr current_component();
```

Returns the current component in this object.

```
virtual void destroy();
```

Destroys this object.

```
virtual void from_any(CORBA::Any& value);
```

Initializes the current component of this object from the specified `Any` object.

A type mismatch exception is raised if the `TypeCode` of value contained in the `Any` does not match the `TypeCode` defined for this object when it was created.

If the passed in `Any` does not contain a legal value the operation raises an invalid value exception.

Parameter	Description
value	An `Any` object containing the value to set for this object

```
virtual boolean next();
```

Advances to the next component, if one exists, and returns `true`. If there are no more components, `false` is returned.

```
virtual void rewind();
```

Returns to the first component contained in this object's sequence. A subsequent invocation of the `current_component` method will return the first component in the sequence.

If this object contains only one component, invoking this method will have no effect.

```
virtual CORBA::Boolean seek(CORBA::Long index);
```

If this object contains multiple components, this method advances to the component with the specified index and returns `true`. A subsequent invocation of the `current_component` method will return the component with the specified index.

If there is no component at the specified index, `false` is returned.

Parameter	Description
`index`	The zero-base index of the desired component.

```
virtual CORBA::ULong component_count( ):
```

Returns the number of components of the value of the DynAny as an unsigned long.

```
virtual CORBA::Any* to_any( );
```

Returns an Any object containing the value of the DynAny.

```
CORBA::TypeCode_ptr type();
```

Returns the TypeCode for the value stored in the DynAny.

```
virtual CORBA::DynAny equal()
```

Compares two DynAny values for equality.

Extraction methods

A set of methods is provided which return the type contained in this `DynAny` object's current component. Example 54 shows the name of each of the extraction methods.

A `TypeMismatch` exception is raised if the value contained in this `DynAny` does not match the expected return type for the extraction method used.

Example 54 Extraction methods offered by the DynAny class

```
virtual CORBA::Any* get_any();
  virtual CORBA::Boolean get_boolean();
  virtual CORBA::Char get_char();
  virtual CORBA::Double get_double();
  virtual CORBA::Float get_float();
  virtual CORBA::Long get_long();
  virtual CORBA::LongDouble get_longdouble();
  virtual CORBA::Long get_longlong();
  virtual CORBA::Octet get_octet();
  virtual CORBA::Object_ptr get_reference();
  virtual CORBA::Short get_short();
  virtual char* get_string();
  virtual CORBA::TypeCode_ptr get_typecode();
  virtual CORBA::ULong get_ulong();
  virtual CORBA::UlongLong get_ulonglong();
  virtual CORBA::UShort get_ushort();
  virtual CORBA::WChar get_wchar();
  virtual CORBA::WChar* get_wstring();
  virtual DynamicAny::DynAny* get_dyn_any();
  virtual CORBA::ValueBase* get_val();
```

Insertion methods

A set of methods is provided that copies a particular type of value to this `DynAny` object's current component. Example 55 shows the list of methods provided for inserting various types.

These methods will raise an `InvalidValue` exception if the inserted object's type does not match the `DynAny` object's type.

Example 55 Insertion methods offered by the DynAny class

```
virtual void insert_any(const CORBA:Any& value);
  virtual void insert_boolean(CORBA::Boolean value);
  virtual void insert_char(CORBA::char value);
  virtual void insert_double(CORBA::Double value);
  virtual void insert_float(CORBA::Float value);
  virtual void insert_long(CORBA::Long value);
  virtual void insert_longdouble(CORBA::LongDouble value);
  virtual void insert_longlong(CORBA::LongLong value);
  virtual void insert_octet(CORBA::Octet value);
  virtual void insert_reference(CORBA:Object_ptr value);
  virtual void insert_short(CORBA::Short value);
  virtual void insert_string(const char* value);
  virtual void insert_typecode(CORBA:TypeCode_ptr value);
  virtual void insert_ulong(CORBA::ULong value);
  virtual void insert_ulonglong(CORBA::ULongLong value);
  virtual void insert_ushort(CORBA::UShort value);
  virtual void insert_wchar(CORBA::WChar value);
  virtual void insert_wstring(const CORBA::WChar* value);
  virtual void insert_dyn_any
  (DynamicAny::DynAny_ph value);
  virtual void insert_val(count CORBA::ValueBase& value);
```

DynAnyFactory

```
class DynamicAny::DynAnyFactory : public
  CORBA::PseudoObject
```

A `DynAnyFactory` object is used to create a new `DynAny` object from an `any` value by invoking an operation on this object. A reference to the `DynAnyFactory` object is obtained by calling `ORB::resolve_initial_references("DynAnyFactory")`.

DynAnyFactory methods

```
DynAny_ptr create_dyn_any (const CORBA::Any& value);
```

Creates a `DynAny` object of the specified value.

Parameter	Description
`value`	A new `DynAny` object of a specified value.

```
DynAny_ptr create_dyn_any_from_type_code
  (CORBA::TypeCode_ptr value);
```

Creates a `DynAny` object of the specified type.

Parameter	Description
`type`	A new `DynAny` object of a specified type.

DynArray

```
class DynamicAny::DynArray : public VISDynComplex
```

Objects of this class are used by a client application or server to create and interpret array data types at run-time which were not defined at compile-time. A `DynArray` may contain a a sequence of basic type (such as a `boolean`, `int`, or `float`) or a constructed type (such as `struct` or `union`). The type contained by a `DynArray` is defined when it is created and may not be changed during the lifetime of the object.

The `next`, `rewind`, `seek`, and `current_component` methods, inherited from `DynAny`, may be used to navigate through the components.

The `VISDynComplex` class is a helper class that allows the ORB to manage complex `DynAny` types.

Important usage restrictions

`DynArray` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `DynAny::to_any` method to convert a `DynArray` object to a sequence of `Any` objects, which can be used as a parameter.

DynArray methods

`virtual void **destroy**();`

Destroys this object.

`CORBA::AnySeq* **get_elements**()`

Returns a sequence of `Any` objects containing the values stored in this object.

`void **set_elements**(CORBA::AnySeq& **_value**);`

Sets the elements contained in this object from the specified sequence of `Any` objects.

`DynamicAny::DynAnySeq* **get_elements_as_dyn_any**();`

Returns a sequence of `DynAny` objects contained within.

```
void set_elements_as_dyn_any (const
  DynamicAny::DynAnySeq& value);
```

Sets the elements contained in the object from the specified sequence of `DynAny` objects.

An `InvalidValue` exception will be raised if the number of elements in `_value` is not equal to the number of elements in this `DynArray`. A type mismatch exception is raised if the type of the `Any` values do not match the `TypeCode` of the `DynAny`.

Parameter	Description
`_value`	An array of `Any` objects whose values will be set in this `DynArray`.

DynEnum

`class **DynamicAny::DynEnum** : public DynamicAny::DynAny`

Objects of this class are used by a client application or server to create and interpret enumeration values at runtime which were not defined at compile-time.

Since this type contains a single component, invoking the `DynAn::rewind` and `DynAny::next` methods on a `DynEnum` object will always return `FALSE`.

Important usage restrictions

`DynEnum` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `to_any` method to convert a `DynEnum` object to an `Any`, which can be used as a parameter.

DynEnum methods

`void **from_any**(const CORBA::Any& **value**);`

Initializes the value of this object from the specified `Any` object.

An `Invalid` exception is raised if the `TypeCode` of value contained in the `Any` does not match the `TypeCode` defined for this object when it was created.

Parameter	Description
`value`	An `Any` object containing the value to set for this object.

`CORBA::Any* **to_any**();`

Returns an Any object containing the value of the current component.

`char* **get_as_string**();`

Returns the `DynEnum` object's value as a string.

`void **set_as_string**(const char* **value_as_string**);`

Sets the value contained in this `DynEnum` from the specified string.

Parameter	Description
`value_as_string`	A string that will be used to set the value in this `DynEnum`.

`CORBA::ULong **get_as_ulong**()`

Returns a `int` containing the `DynEnum` object's value.

`void **set_as_ulong**(CORBA::ULong **value_as_ulong**)`

Sets the value contained in this `DynEnum` from the specified `CORBA::Ulong`.

Parameter	Description
`value_as_ulong`	An integer that will be used to set the value in this `DynEnum`.

DynSequence

`class **DynamicAny::DynSequence** : public DynamicAny::DynArray`

Objects of this class are used by a client application or server to create and interpret sequence data types at run-time which were not defined at compile-time. A `DynSequence` may contain a sequence of basic type (such as a `boolean`, `int`, or `float`) or a constructed type (such as a `struct` or `union`). The type contained by a `DynSequence` is defined when it is created and may not be changed during the lifetime of the object.

The `next`, `rewind`, `seek`, and `current_component` methods may be used to navigate through the components.

Important usage restrictions

`DynSequence` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `to_any` method to convert a `DynSequence` object to a sequence of `Any` objects, which can be used as a parameter.

DynSequence methods

`CORBA::ULong` **`get_length`**`()`

Returns the number of components contained in this `DynSequence`.

`void` **`set_length`**`(CORBA::ULong` **`length`**`);`

Sets the number of components contained in this `DynSequence`.

If you specify a length that is less than the current number of components, the sequence will be truncated.

Parameter	Description
`length`	The number of components to be contained in this `DynSequence`.

`CORBA::AnySeq *` **`get_elements`**`();`

Returns a sequence of Any objects containing the value stored in this object.

`void` **`set_elements`** `(const AnySeq& _value)`

Sets the elements within this object with specified sequence of Any objects.

`set _elements as dyn_any and get_elements_as_dyn_any;`

See "DynArray" on page 6-10 for more details.

DynStruct

`class` **`DynamicAny::DynStruct`** `:public VISDynComplex`

Objects of this class are used by a client application or server to create and interpret structures at run-time which were not defined at compile-time.

The `next`, `rewind`, `seek`, and `current_component` methods may be used to navigate through the structure members.

You create an `DynStruct` object by invoking the `DynAnyFactory::create_dyn_any_from_typecode` method.

Important usage restrictions

`DynStruct` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `to_any` method to convert a `DynStruct` object to an `Any` objects, which can be used as a parameter.

DynStruct methods

`void` **`destroy`**`();`

Destroys this object.

`CORBA::FieldName` **`current_member_name`**`();`

Returns the member name of the current component.

`CORBA::TCKind` **`current_member_kind`**`();`

Returns the `TypeCode` associated with the current component.

`DynamicAny::NameValuePairSeq` **`get_members`**`();`

Returns the members of the structure as a sequence of `NameValuePair` objects.

void **set_members**(const DynamicAny::NameValuePairSeq& **value**);

Sets the structure members from the array of `NameValuePair` objects.

DynamicAny::Name DynAnyPairSeq get_members_as_dyn_any();

Returns the members of the structure as `NameDynAnyPair` sequence.

```
void set_members_as_dyn_any (const
  DynamicAny::nameDynAnyPairSeq value);
```

Sets the structure members from `NameDynAnyPai`r objects.

An `InvalidValue` exception is raised if the length of the value sequence is not equal to the number of members of `DynStruct`, and a `type mismatch` exception is raised when any of the element's typecode does not match that of the structure.

DynUnion

`class` **`DynamicAny::DynUnion`** `: public VISDynComplex`

This interface is used by a client application or server to create and interpret unions at run-time which were not defined at compile-time. The `DynUnion` contains a sequence of two elements; the union discriminator and the actual member.

The `next`, `rewind`, `seek`, and `current_component` methods may be used to navigate through the components.

You create an `DynUnion` object by invoking the `DynamicAny::DynAnyFactory::create_dyn_any_from_type_code` method.

Important usage restrictions

`DynUnion` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `DynAny::to_any` method to convert a `DynUnion` object to an `Any` objects, which can be used as a parameter.

DynUnion methods

`DynamicAny::DynAny_ptr` **`get_discriminator`**`();`

Returns a `DynAny` object containing the discriminator for the union.

`CORBA::TCKind` **`discriminator_kind`**`();`

Returns the type code of the discriminator for the union.

`DynamicAny::DynAny_ptr` **`member`**`();`

Returns a `DynAny` object for the current component which represents a member in the union.

`CORBA::TCKind` **`member_kind`**`();`

Returns the type code for the current component, which represents a member in the union.

`CORBA::FieldName` **`member_name`**`();`

Returns the member name of the current component.

`void set_discriminator (DynamicAny::DynAny_ptr` **`value`**`);`

Sets the discriminator of this DynUnion to the specified value.

`void set_to_default_member());`

Sets the discriminator to a value that is consistent with the value of the default case of a union.

`void set_to_no_active_member();`

Sets the discriminator to a value that does not correspond to any of the union's case labels.

`boolean has_no_active_member();`

Returns true if the union has no active member (that is, the union's value consists solely of its discriminator because the discriminator has a value that is not listed as an explicit case label).

ExceptionList

`class` **`CORBA::ExceptionList`**

This class contains a list of type codes that represent exceptions that may be raised by an operation request. See "Request" on page 6-21.

ExceptionList methods

`CORBA::ExceptionList``();`

This method constructs an empty exception list.

CORBA::ExceptionList(CORBA::ExceptionList& list);

This is a copy constructor.

Parameter	Description
list	The list to be copied.

~CORBA::ExceptionList();

This method is the default destructor.

void **add**(CORBA::TypeCode_ptr **tc**);

This method adds the specified exception type code to this object's list.

Parameter	Description
tc	The type code of an exception to be added to the list.

void **add_consume**(CORBA::TypeCode_ptr **tc**);

This method adds the specified exception type code to this object's list. Ownership of the passed argument is assumed by this ExceptionList. You should not attempt to access or free the argument after invoking this method.

Parameter	Description
tc	The type code of an exception to be added to the list.

CORBA::ULong **count**() const;

This method returns the number of items currently stored in the list.

CORBA::TypeCode_ptr **item**(CORBA::Long **index**);

This method returns a pointer to the TypeCode stored in the list at the specified index. If the index is invalid, a NULL pointer is returned. You should not attempt to access or free the argument after invoking this method. To remove a TypeCode from the list, use the remove method.

Parameter	Description
index	The index of the type code to be returned. The index is zero-based.

void **remove**(CORBA::long **index**);

This method removes from the list, the TypeCode with the specified index. If the index is invalid, no removal will occur.

Parameter	Description
index	The index of the type code to be removed. The index is zero-based.

static CORBA::ExceptionList_ptr
_duplicate(CORBA::ExceptionList_ptr **ptr**);

This static method increments the reference count for the specified object and then returns a pointer to it.

Parameter	Description
ptr	The object to be duplicated.

```
static CORBA::ExceptionList_ptr _nil();
```

This static method returns a NULL pointer that can be used for initialization purposes.

```
static void _release(CORBA::ExceptionList *ptr);
```

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
ptr	The object to be released.

NamedValue

```
class CORBA::NamedValue
```

The `NamedValue` class is used to represent a name-value pair used as a parameter or return value in a Dynamic Invocation Interface request. Objects of this class are grouped into an `NVList`, described in "NVList" on page 6-18. The `Any` class is used to represent the value associated with this object. The `Request` class is described in "Request".

Include file

You should include the file **corba.h** when using this class.

NamedValue methods

```
CORBA::Flags flags() const;
```

This method returns the flag defining how this name-value pair is to be used. One of the following is returned.

`ARG_IN`	This object represents an input parameter.
`ARG_OUT`	This object represents an output parameter.
`ARG_INOUT`	This object represents both an input and output parameter.
`IN_COPY_VALUE`	This value can be specified in combination with the `ARG_INOUT` flag to specify that the ORB should make a copy of the parameter. This allows the ORB to release memory associated with this parameter without impacting the client application's memory.

```
const char *name() const;
```

This method returns the name portion of this object's name-value pair. You should never release the storage pointed to by the return argument.

```
CORBA::Any *value() const;
```

This method returns the value portion of this object's name-value pair. You should never release the storage pointed to by the return argument.

```
static CORBA::NamedValue_ptr
  _duplicate(CORBA::NamedValue_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to it.

Parameter	Description
ptr	The object to be duplicated.

```
static CORBA::NamedValue_ptr _nil();
```

This static method returns a NULL pointer that can be used for initialization purposes.

```
static void _release(CORBA::NamedValue *ptr);
```

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
ptr	The object to be released.

NVList

```
class CORBA::NVList
```

The `NVList` class is used to contain a list of `NamedValue` objects, described in "NamedValue" in this guide, and is used to pass parameters associated with a Dynamic Invocation Interface request. The `Request` class is described in "Request".

Several methods are provided for adding items to the list. You should never release the storage pointed to by the return argument. Always use the `remove`method to delete an item from the list.

Include file

You should include the file **corba.h** when using this class.

NVList methods

```
CORBA::NamedValue_ptr add(CORBA::Flags flags);
```

This method adds a `NamedValue` object to this list, initializing only the flags. Neither the name or value of the added object are initialized. A pointer is returned which can be used to initialize the name and value attributes of the `NamedValue`. You should never release the storage associated with the return argument.

Parameter	Description
flags	The flag indicating the intended use of the `NamedValue` object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.

```
CORBA::NamedValue_ptr add_item(const char *name,
  CORBA::Flags flags);
```

This method adds a `NamedValue` object to this list, initializing the object's `flags` and `name` attributes. A pointer is returned which can be used to initialize the value attribute of the `NamedValue`. You should never release the storage associated with the return argument.

Parameter	Description
`name`	The name
`flags`	The flag indicating the intended use of the `NamedValue` object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.

```
NamedValue_ptr add_item_consume(char *nm, CORBA::Flags
  flags);
```

This method is the same as the `add_item` method, except that the NVList takes over the management of the storage pointed to by `nm`. You will not be able to access `nm` after this method is called because the list may have copied and released it. When this item is removed, the storage associated with it is automatically freed.

Caution

You should never release the memory associated with this method's return value.

Parameter	Description
`name`	The name
`flags`	The flag indicating the intended use of the `NamedValue` object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.

```
CORBA::NamedValue_ptr add_value(const char *name, const
  CORBA::Any *value, CORBA::Flags flags);
```

This method adds a `NamedValue` object to this list, initializing the name, value, and flags. A pointer to the `NamedValue` object is returned. You should never release the storage associated with the return argument.

Parameter	Description
`name`	The name
`value`	The value
`flags`	The flag indicating the intended use of the `NamedValue` object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.

```
NamedValue_ptr add_value_consume(char *nm, CORBA::Any
  *value, CORBA::Flags flags);
```

This method is the same as the add_value method, except that the NVList takes over the management of the storage pointed to by nm and value. You will not be able to access nm or value after this method is called because the list may have copied and released them. When this item is removed, the storage associated with it is automatically freed.

Parameter	Description
`name`	The name

Parameter	Description
value	The value
flags	The flag indicating the intended use of the NamedValue object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.

```
CORBA::Long count() const;
```

This method returns the number of NamedValue objects in this list.

```
static CORBA::Boolean CORBA::is_nil(NVList_ptr obj);
```

This method returns true if the specified NamedValue pointer is NULL.

Parameter	Description
obj	The object pointer to be checked.

```
NamedValue_ptr item(CORBA::Long index);
```

This method returns the NamedValue in the list with the specified index. Never release the storage associated with the return argument.

Parameter	Description
index	The index of the desired NamedValue object. Note that indexing is zero-based.

```
static void CORBA::release(CORBA::NVList_ptr obj);
```

This static method releases the specified object.

Parameter	Description
obj	The object to be released.

```
Status remove(CORBA::Long index);
```

This method deletes the NamedValue object from this list, located at the specified index. Storage associated with items in the list that were added using the add_item_consume or add_value_consume methods will be released before the item is removed.

Parameter	Description
index	The index of the NamedValue object. Note that indexing is zero-based.

```
static CORBA::NVList_ptr _duplicate(CORBA::NVList_ptr
  ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to it.

Parameter	Description
ptr	The object to be duplicated.

```
static CORBA::NVList_ptr _nil();
```

This static method returns a NULL pointer that can be used for initialization purposes.

```
static void _release(CORBA::NVList *ptr);
```

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
ptr	The object to be released

Request

```
class CORBA::Request
```

The `Request` class is used by client applications to invoke an operation on an ORB object using the Dynamic Invocation Interface. A single ORB object is associated with a given `Request` object. The `Request` represents an operation that is to be performed on the ORB object. It includes the arguments to be passed, the `Context`, and an `Environment` object, if any. Methods are provided for invoking the request, receiving the response from the object implementation, and retrieving the result of the operation.

You can create a `Request` object by using the `Object::_create_request`, described in "CORBA::Object methods".

Note that a `Request` object will retain ownership of all return parameters, so you should never attempt to free them.

Include file

The **corba.h** file should be included when you use this class.

Request methods

```
CORBA::Any& add_in_arg();
```

This method adds an unnamed input argument to this `Request` and returns a reference to the Any object so that you can set its name, type, and value.

```
CORBA::Any& add_in_arg(const char *name);
```

This method adds a named input argument to this `Request` and returns a reference to the `Any` object so that you can set its type, and value.

Caution

You should never release the memory associated with this method's return value.

Option	Description
`name`	The name of the input argument to be added.

```
CORBA::Any& add_inout_arg();
```

This method adds an unnamed `inout` argument to this `Request` and returns a reference to the Any object so that you can set its name, type, and value.

```
CORBA::Any& add_inout_arg(const char *name);
```

This method adds a named inout argument to this `Request` and returns a reference to the `Any` object so that you can set its type and value.

Option	Description
`name`	The name of the inout argument to be added.

```
CORBA::Any& add_out_arg();
```

This method adds an unnamed output argument to this `Request` and returns a reference to the Any object so that you can set its name, type, and value.

```
CORBA::Any& add_out_arg(const char *name);
```

This method adds a named output argument to this `Request` and returns a reference to the `Any` object so that you can set its type, and value.

Option	Description
`name`	The name of the output argument to be added.

```
CORBA::NVList_ptr arguments();
```

This method returns a pointer to an `NVList` object containing the arguments for this request. The pointer can be used to set or retrieve the argument values. For more information on `NVList`, see "NVList".

Caution

You should never release the memory associated with this method's return value.

```
CORBA::ContextList_ptr contexts();
```

This method returns a pointer to a list of all the `Context` objects that are associated with this `Request`. For more information on the `Context` class, see "Context".

Caution

You should never release the memory associated with this method's return value.

```
CORBA::Context_ptr ctx() const;
```

This method returns a pointer to the `Context` associated with this request.

```
void ctx(CORBA::Context_ptr ctx);
```

This method sets the `Context` to be used with this request. For more information on the `Context` class, see "Context".

Option	Description
`ctx`	The `Context` object to be associated with this request.

```
CORBA::Environment_ptr env();
```

This method returns a pointer to the `Environment` associated with this request. For more information on the `Environment` class, see "Environment".

```
CORBA::ExceptionList_ptr exceptions();
```

This method returns a pointer to a list of all the exceptions that this request may raise.

Caution

You should never release the memory associated with this method's return value.

```
void get_response();
```

This method is used after the `send_deferred` method has been invoked to retrieve a response from the object implementation. If there is no response available, this method blocks the client application until a response is received.

```
void invoke();
```

This method invokes this `Request` on the ORB object associated with this request. This method will block the client until a response is received from the object implementation. This `Request` should be initialized with the target object, operation name and arguments before this method is invoked.

```
const char* operation() const;
```

This method returns the name of the operation that this request will represent.

```
CORBA::Boolean poll_response();
```

This non-blocking method is invoked after the `send_deferred` method to determine if a response has been received. This method returns true if a response has been received, otherwise false is returned.

```
CORBA::NamedValue_ptr result();
```

This method returns a pointer to a `NamedValue` object where the return value for the operation will be stored. The pointer can be used to retrieve the result value after the request has been processed by the object implementation. For more information on the `NamedValue` class, see "NamedValue".

```
CORBA::Any& return_value();
```

This method returns a reference to an `Any` object that represents the return value of this `Request` object.

```
void set_return_type(CORBA::TypeCode_ptr tc);
```

This method sets the `TypeCode` of the return value that is expected. You must set the return value's type before using the `invoke` method or one of the `send` methods.

Option	Description
`tc`	The return value's type.

`void` **`send_deferred`**`();`

Like the `invoke` method, this method sends this `Request` to the object implementation. Unlike the `invoke` method, this method does not block waiting for a response. The client application can retrieve the response using the `get_response` method.

`void` **`send_oneway`**`();`

This method invokes this `Request` as a *oneway* operation. Oneway operations do not block and do not result in a response being sent from the object implementation to the client application.

`CORBA::Object_ptr` **`target`**`() const;`

This method returns a reference to the target object on which this request will operate.

`static CORBA::Request_ptr` **`_duplicate`**`(CORBA::Request_ptr` **`ptr`**`);`

This static method increments the reference count for the specified object and then returns a pointer to it.

Option	Description
`ptr`	The object to be duplicated.

`static CORBA::Request_ptr` **`_nil`**`();`

This static method returns a NULL pointer that can be used for initialization purposes.

`static void` **`_release`**`(CORBA::Request` ***ptr`**`);`

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

Option	Description
`ptr`	The object to be released.

ServerRequest

`class` **`CORBA::ServerRequest`**

The `ServerRequest` class is used to represent an operation request received by an object implementation that is using the Dynamic Skeleton Interface. When the POA receives a client operation request, it invokes the object implementation's `invoke` method and passes an object of this type.

This class provides the methods needed by the object implementation to determine the operation being requested and the arguments. It also provides methods for setting the return value and reflecting exceptions to the client application.

You should never attempt to free memory associated with any value returned by this class.

Include file

The **corba.h** file should be included when you use this class.

ServerRequest methods

```
void arguments(CORBA::NVList_ptr param);
```

This method obtains the parameter list for this request.

Option	Description
`params`	The parameter list to be filled in. You must initialize this list with the appropriate number of `Any` objects and set their type and flag values prior to invoking this method.

```
CORBA::Context_ptr ctx()
```

This method returns the `Context` object associated with the request.

Caution

You should never release the memory associated with this method's return value.

```
void exception(CORBA::Any_ptr exception);
```

This method is used to reflect the specified exception to the client application.

Option	Description
`exception`	The exception that was raised. If this pointer is NULL, a `CORBA::UnknownUserException` will be reflected.

```
const char *operation() const;
```

Returns the name of the operation being requested.

```
const char* op_name() const
```

This method returns the operation name associated with the request. The object implementation uses this name to determine if the request is valid, to perform the appropriate processing to fulfill the request, and to return the appropriate value to the client.

```
void params(CORBA::NVList_ptr params);
```

This method accepts an `NVList` object, initialized with the appropriate number of `Any` objects, and fill it in with the parameters supplied by the client.

Option	Description
`params`	The parameter list to be filled in. You must initialize this list with the appropriate number of `Any` objects and set their type and flag values prior to invoking this method.

```
void  result(CORBA::Any_ptr result);
```

This method sets the result that is to be reflected to the client application.

Option	Description
result	An Any object representing the return value.

```
void set_exception(const CORBA::Any& a);
```

This method sets the exception that is to be reflected to the client application.

Parameter	Description
a	An Any object representing the exception.

a An Any object representing the exception.

```
void set_result(const CORBA::Any& a);
```

This method sets the result that is to be reflected to the client application.

Parameter	Description
a	An Any object representing the exception.

```
static CORBA::ServerRequest_ptr
  _duplicate(CORBA::ServerRequest_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to it.

Parameter	Description
ptr	The object to be duplicated.

```
static CORBA::ServerRequest_ptr _nil();
```

This static method returns a NULL pointer that can be used for initialization purposes.

```
static void _release(CORBA::ServerRequest *ptr);
```

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
ptr	The object to be released.

Interface Repository Interfaces and Classes

This chapter describes the classes and interfaces that you can use to access the interface repository. The interface repository maintains information on modules and the interfaces they contain as well as other types like operations, attributes, and constants.

Availability

Note that the Interface Repository (IR) is available **ONLY** on the development host. VisiBroker-RT for C++ does **NOT** provide an Interface Repository as runtime library.

Additionally the IR provides functionality which address the more Dynamic aspects of CORBA, and therefore the IR is excluded as per the "minimum CORBA" OMG specification. The "minimum CORBA" OMG specification identifies dynamic functionality which should be excluded from an ORB, in an effort to reduce the ORB footprint.

For details, see the minimum CORBA specification document, OMG document number orbos/ 98-08-04. This document is available for download using the URL ftp://ftp.omg.org/pub/docs/orbos/98-08-04.pdf.

Link doesn't work.

AliasDef

```
class CORBA::AliasDef : public CORBA::TypedefDef
```

This class is derived from the `TypedefDef` class and represents an alias for a `typedef` that is stored in the interface repository. This class provides methods for setting and obtaining the `IDLType` of the original `typedef`.

For more information on the `TypedefDef` class, see "TypedefDef". For more information on the `IDLType` class, see "IDLType".

AliasDef methods

```
CORBA::IDLType original_type_def();
```

This method returns the `IDLType` of the original typedef for which this object is an alias.

```
void original_type_def(CORBA::IDLType_ptr val);
```

This method sets the `IDLType` of the original `typedef` for which this object is an alias.

Parameter	Description
`val`	The IDLType to set for this alias.

ArrayDef

```
class CORBA::ArrayDef : public CORBA::IDLType
```

This class is derived from the `IDLType` class and represents an array that is stored in the interface repository. It provides methods for setting and obtaining the type of the elements in the array as well as the length of the array.

ArrayDef methods

```
CORBA::TypeCode element_type();
```

This method returns the `TypeCode` of the array's elements.

```
CORBA::IDLType_ptr element_type_def();
```

This method returns the `IDLType` of the elements stored in this array.

```
void element_type_def(CORBA:IDLType_ptr
  element_type_def);
```

This method sets the `IDLType` of the elements stored in the array.

Parameter	Description
`element_type_def`	The `IDLType` of the elements in the array.

```
CORBA::ULong length();
```

This method returns the number of elements in the array.

```
void length(CORBA::ULong length);
```

This method sets the number of elements in the array.

Parameter	Description
length	The number of elements in the array.

AttributeDef

```
class CORBA::AttributeDef : public CORBA::Contained,
  public CORBA::Object
```

The class is used to represent an interface attribute that is stored in the interface repository. It provides methods for setting and obtaining the attribute's mode, `typedef`. A method is also provided for obtaining the attribute's type.

AttributeDef methods

```
CORBA::AttributeMode mode();
```

This method returns the mode of the attribute. It might be either `CORBA::AttributeMode ATTR_READONLY` for read only attributes or `CORBA::AttributeMode ATTR_NORMAL` for read-write ones. See "AttributeMode".

`void mode(CORBA::AttributeMode _val);`

This method sets the mode of the attribute.

Parameter	Description
`_val`	The mode to set.

`CORBA::TypeCode_ptr type();`

This method returns the `TypeCode` representing the attribute's type.

`CORBA::IDLType_ptr type_def();`

This method returns this object's `IDLType`.

`void type_def(CORBA::IDLType_ptr type_def);`

This method sets the `IDLType` for this object.

Parameter	Description
`type_def`	The `IDLType` of this object.

AttributeDescription

`struct CORBA::AttributeDescription`

The `AttributeDescription` structure describes an attribute that is stored in the interface repository.

AttributeDescription members

`CORBA::Identifier_var name`

This member represents the name of the attribute.

`CORBA::RepositoryId_var id`

This member represents the repository id of the attribute.

`CORBA::RepositoryId_var defined_in`

This member represents the repository id of the interface in which this attribute is defined.

`CORBA::String_var version`

This member represents the attribute's version.

`CORBA::TypeCode_var type`

This member represents the attribute's IDL type.

CORBA::AttributeMode **mode**

This member represents the mode of this attribute.

AttributeMode

`enum` **`CORBA::AttributeMode`**

The enumeration defines the values used to represent the mode of an attribute; either read-only or normal (read-write).

AttributeMode values

Constant	Represents
`ATTR_NORMAL`	A read-write attribute.
`ATTR_READONLY`	A read-only attribute.

ConstantDef

`class` **`CORBA::ConstantDef`** `: public CORBA::Contained`

The class is used to represent a constant definition that is stored in the interface repository. This interface provides methods for setting and obtaining the constant's type, value, and `typedef`.

ConstantDef methods

`CORBA::TypeCode_ptr` **`type`**`();`

This method returns the `TypeCode` representing the object's type.

`CORBA::IDLType_ptr` **`type_def`**`();`

This method returns this object's `IDLType`.

`void` **`type_def`**`(CORBA::IDLType_ptr` **`type_def`**`);`

This method sets the `IDLType` of the constant.

Parameter	Description
`type_def`	The `IDLType` of this constant.

`CORBA::Any` ***`value`**`();`

This method returns a pointer to an `Any` object representing this object's value.

`void` **`value`**`(CORBA::Any&` **`_val`**`);`

This method sets the value for this constant.

Parameter	Description
_val	An `Any` object that represents this object's value.

ConstantDescription

struct **CORBA::ClassName**

The `ConstantDescription` structure describes a constant that is stored in the interface repository.

ConstantDescription members

`CORBA::Identifier_var` **`name`**

This member represents the name of the constant.

`CORBA::RepositoryId_var` **`id`**

This member represents the repository id of the constant.

`CORBA::RepositoryId_var` **`defined_in`**

This member represents the name of the module or interface in which this constant is defined.

`CORBA::String_var` **`version`**

This member represents the constant's version.

`CORBA::TypeCode_var` **`type`**

This member represents the constant's IDL type.

`CORBA::Any` **`value`**

This member represents the value of this constant.

Contained

`class` **`CORBA::Contained`** `: public CORBA::IRObject, public CORBA::Object`

The `Contained` class is used to derive all interface repository objects that are themselves contained within another interface repository object. This class provides methods for:

- Setting and retrieving the object's name and version.
- Determining the `Container` that contains this object.
- Obtaining the object's absolute name, containing repository, and description.
- Moving an object from one container to another.

Include file

The **corba.h** file should be included when you use this class.

```
interface Contained: IRObject {
  attribute RepositoryId id;
  attribute Identifier name;
  attribute String_var version;

  readonly attribute Container defined_in;
  readonly attribute ScopedName absolute_name;
  readonly attribute Repository containing_Repository;

  struct Description {
```

```
    DefinitionKind kind;
    any value;
  };
  Description describe();
  void move(
    in Container new_Container,
    in Identifier new_name,
    in String_var new_version
  );
};
```

Contained methods

`CORBA::String_var` **`absolute_name`**`();`

This method returns the absolute name, which uniquely identifies this object within its containing `Repository`. If the object's `defined_in` attribute (set when the object is created) references a `Repository`, then the absolute name is simply the object's name preceded by the string "::".

`CORBA::Repository_ptr` **`containing_repository`**`();`

Returns the repository that contains this object.

`CORBA::Container_ptr` **`defined_in`**`();`

Returns the Container where this object is defined.

`Description*` **`describe`**`();`

Returns this object's description. See "Description" for more information on the `Description` structure.

`CORBA::String_var` **`id`**`();`

Returns this object's repository identifier.

`void` **`id`**`(const char *id);`

Sets the repository identifier that uniquely identifies this object.

Parameter	Description
`id`	The repository identifier for this object.

`CORBA::String_var` **`name`**`();`

This method returns the name of the object, which uniquely identifies it within the scope of its container.

`void` **`name`**`(const char * val);`

This method sets the name of the contained object.

Parameter	Description
`name`	The object's name.

`CORBA::String_var` **`version`**`();`

This method returns the object's version, which distinguishes this object from other objects that have the same name.

```
void version(CORBA::String_var& val);
```

This method sets this object's version.

Parameter	Description
val	The object's version.

```
void move(CORBA::Container_ptr new_container, const
  char *new_name, CORBA::String_var& new_version);
```

Moves this object from its current `Container` to the `new_container`.

Parameter	Description
new_container	The Container to which this object is being moved.
new_name	The new name for the object.
new_version	The new version specification for the object.

Container

```
class CORBA::Container : public CORBA::Container,
  public CORBA::Object
```

The `Container` class is used to create a containment hierarchy in the interface repository. A `Container` object holds object definitions derived from the `Contained` class. All object definitions derived from the `Container` class, with the exception of the `Repository` class, also inherit from the `Contained` class.

The `Container` provides methods to create types of IDL types defined in **orbtypes.h**, including `InterfaceDef`, `ModuleDef` and `ConstantDef` classes, except `ValueMEmberDef`. Each definition that is created will have its `defined_in` attribute initialized to point to this object.

Include file

The **corba.h** file should be included when you use this class.

```
interface Container: IRObject {
  Contained lookup(in ScopedName search_name);
  ContainedSeq contents(
    in DefinitionKind limit_type,
    in boolean exclude_inherited
  );
  ContainedSeq lookup_name(
    in Identifier search_name,
    in long levels_to_search,
    in CORBA::DefinitionKind limit_type,
    in boolean exclude_inherited
  );
  struct Description {
    Contained Contained_object;
    DefinitionKind kind;
    any value;
  };
  typedef sequence<Description> DescriptionSeq; DescriptionSeq
  describe_contents(
    in DefinitionKind limit_type,
    in boolean exclude_inherited,
    in long max_returned_objs
);
```

Container methods

```
CORBA::ContainedSeq * contents(CORBA::DefinitionKind
  limit_type, CORBA::Boolean exclude_inherited);
```

This method returns a list of contained object definitions directly contained or inherited into the container. You can use this method to navigate through the hierarchy of object definitions in the `Repository`. All object definitions contained by modules in the `Repository` are returned, followed by all object definitions contained within each of those modules.

Parameter	Description
`limit_type`	The interface object types to be returned. Specifying `dk_all` will return objects of all types.
`exclude_inherited`	If set to TRUE, inherited objects will not be returned.

```
CORBA::AliasDef_ptr create_alias(const char * id,
  const char *name, const CORBA::String_var& version,
  CORBA::IDLType_ptr original_type);
```

This method creates a `AliasDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
`id`	The alias's id.
`name`	The alias's name.
`version`	The alias's version.
`original_type`	The type of the object for which this object is an alias.

```
CORBA::ConstantDef_ptr create_constant(const char * id,
  const char *name, const CORBA::String_var& version,
  CORBA::IDLType_ptr type, const CORBA::Any& value);
```

This method creates a `ConstantDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
`id`	The constant's id.
`name`	The constant's name.
`version`	The constant's version.
`type`	The type of the value specified below.
`value`	The constant's value.

```
CORBA::EnumDef_ptr create_enum(const char * id, const
  char *name, const CORBA::String_var& version, const
  CORBA::EnumMemberSeq& members);
```

This method creates a `EnumDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
`id`	The enumeration's id.
`name`	The enumeration's name.
`version`	The enumeration's version.
`members`	A list of the enumeration's fields.

```
CORBA::ExceptionDef_ptr create_exception(const char *
  id, const char *name, const CORBA::String_var&
  version, const CORBA::StructMEmberSeq& members);
```

This method creates a `ExceptionDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
`id`	The exception's id.
`name`	The exception's name.
`version`	The exception's version.
`members`	The sequence for the structure's fields, if any.

```
CORBA::InterfaceDef_ptr create_interface(const char *
  id, const char *name, const CORBA::String_var&
  version, const CORBA::InterfaceDefSeq&
  base_interfaces) (CORBA:Boolean is_abstract);:
```

This method creates a `InterfaceDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
`id`	The interface's id.
`name`	The interface's name.
`version`	The interface's version.
`base_interfaces`	A list of all interfaces that this interface inherits from.
`is_abstract`	Indicates whether or not this is an abstract interface.

```
CORBA::ModuleDef_ptr create_module(const char * id,
  const char *name, const CORBA::String_var& version);
```

This method creates a `ModuleDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
`id`	The module's id.
`name`	The module's name.
`version`	The module's version.

```
CORBA::StructDef_ptr create_struct(const char * id,
  const char *name, const CORBA::String_var& version,
  const CORBA::StructMemberSeq& members);
```

This method creates a `StructureDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
`id`	The structure's id.
`name`	The structure's name.
`version`	The structure's version.
`members`	The sequence for the structure's fields.

```
CORBA::UnionDef_ptr create_union(const char * id, const
  char *name, const CORBA::String_var& version,
```

```
CORBA::IDLType_ptr discriminator_type, const
CORBA::UnionMemberSeq& members);
```

This method creates a `UnionDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
`id`	The union's id.
`name`	The union's name.
`version`	The union's version.
`discriminator_type`	The type of the union's discriminant value.
`members`	The sequence of each of the union's fields.

```
CORBA::DescriptionSeq *
  describe_contents(CORBA::DefinitionKind limit_type,
  CORBA::Boolean exclude_inherited, CORBA::Long
  max_returned_objs);
```

This method returns a description for all definitions directly contained by or inherited into this container.

Parameter	Description
`limit_type`	The interface object types whose descriptions are to be returned. Specifying `dk_all` will return the descriptions for objects of all types.
`exclude_inherited`	If set to `true`, descriptions for inherited objects will not be returned.
`max_returned_objs`	The maximum number of descriptions to be returned. Setting this parameter to –1 will return all objects.

```
CORBA::Contained_ptr lookup(const char *search_name);
```

This method locates a definition relative to this container, given a scoped name. An absolute scoped name, one beginning with "::", may be specified to locate a definition within the enclosing repository. If no object is found, a `NULL` value is returned.

Parameter	Description
`search_name`	The object's interface name.

```
CORBA::ContainedSeq * lookup_name(const char
  *search_name, CORBA::Long levels_to_search,
  CORBA::DefinitionKind limit_type, CORBA::Boolean
  exclude_inherited);
```

This method locates an object by name within a particular object. The search can be constrained by the number of levels in the hierarchy to be searched, the object type, and whether inherited objects should be returned.

Parameter	Description
`search_name`	The contained object's name.
`levels_to_search`	The number of levels in the hierarchy to search. Setting this parameter to a value of –1 will cause all levels to be searched. Setting this parameter to 1 will search only this object.
`limit_type`	The interface object types to be returned. Specifying `dk_all` will return objects of all types.
`exclude_inherited`	If set to `true`, inherited objects will not be returned.

```
CORBA::ValueDef_ptr create_value(const char * id, const
  char *name, const char version, CORBA::boolean
  is_custom, CORBA::boolean is_abstract, const
  CORBA::ValueDef_ptr _base_value, CORBA::boolean
  is_truncatable, const CORBA::ValueDefSeq&
  abstract_base_values, const CORBA::InterfaceDefSeq&
  supported _interfaces, const CORBA.InitializerSeq&
  initializers)
```

This method creates a `ValueDef` object in this `Container` with the specified attributes and returns a reference to the newly created object.

Parameter	Descrption
`id`	The structure's repository id.
`name`	The structure's name.
`version`	The structure's version.
`is_custom`	If set to `true`, creates a custom valuetype.
`is_abstract`	If set to `true`, creates and abstract valuetype.
`base_values`	The list of supported base values.
`is_trucatable`	If set to `true`, creates a truncatable valuetype.
`abstract_base_values`	The list of supported abstract base values.
`supported_interfaces`	The list of supported interfaces.
`initializer`	The list of initializers this value type supports

```
CORBA::ValueBoxDef_ptr create_value_box(const char* id,
  const char* name, const char* version,
  CORBA::IDLType_ptr original_type)
```

This method creates a `ValueBoxDef` object in this `Container` with the specified attributes and returns a reference to the newly created object.

Parameter	Description
`id`	The structure's repository id.
`name`	The structure's name.
`version`	The structure's version.
`original_type`	The IDL type of the original object for which this is an alias.

DefinitionKind

`enum` **`CORBA::DefinitionKind`**

The `DefinitionKind` enumeration contains the constants that define the possible types of interface repository objects. There are a set of integer constants, prefixed with `dk_`, that correspond to all the possible kinds of repository objects.

DefinitionKind values

Constant	Represents
`dk_none`	Exclude all types (used in repository lookup methods)
`dk_all`	All possible types (used in repository lookup methods)
`dk_Alias`	Alias
`dk_Array`	Array

Constant	Represents
dk_Attribute	Alias
dk_Constant	Constant
dk_Enum	Enum
dk_Exception	Exception
dk_Fixed	Fixed
dk_Interface	Interface
dk_Module	Module
dk_Native	Native
dk_Operation	Interface Operation
dk_Primitive	Primitive type (such as int or long)
dk_Repository	Repository
dk_Sequence	Sequence
dk_String	String
dk_Struct	Struct
dk_Typedef	Typedef
dk_Union	Union
dk_Value	ValueType
dk_ValueBox	ValueBox
dk_ValueMember	ValueMember
dk_Wstring	Unicode string

Description

`struct CORBA::Container::`**`Description`**

This structure provides a generic description for items in the interface repository that are derived from the `Contained` class.

Description members

`CORBA::Contained_var` **`contained_object`**

The object contained in this struct.

`CORBA::DefinitionKind` **`kind`**

The object's kind.

`CORBA::Any` **`value`**

The object's value.

EnumDef

`class` **`CORBA::EnumDef`** `: public CORBA::TypedefDef, public CORBA::Object`

The class is used to represent an enumeration that is stored in the interface repository. This interface provides methods for setting and retrieving the enumeration's list of members.

EnumDef methods

`CORBA::EnumMemberSeq *members() ;`

This method returns the enumeration's list of members.

`void members(CORBA::EnumMemberSeq members) ;`

This method sets the enumeration's list of members.

Parameter	Description
`members`	The list of members.

ExceptionDef

`class ExceptionDef : public CORBA::Contained`

The class is used to represent an exception that is stored in the interface repository. This class provides methods for setting and retrieving the exception's list of members as well as a method for retrieving the exception's `TypeCode`.

ExceptionDef methods

`CORBA::StructMemberSeq *members() ;`

This method returns this exception's list of members.

`void members(CORBA:StructMemberSeq& members) ;`

This method sets the exception's list of members.

Parameter	Description
`members`	The list of members

`CORBA::TypeCode_ptr type() ;`

This method returns the `TypeCode` that represents this exception's type.

ExceptionDescription

`struct CORBA::ExceptionDescription`

This structure is used to represent information about an exception that is stored in the interface repository.

ExceptionDescription members

`CORBA::String_var defined_in`

This member represents the repository Id of the module or interface in which this exception is defined.

`CORBA::String_var` **id**

This member represents the repository id of the exception.

`CORBA::String_var` **name**

This member represents the name of the exception.

`CORBA::TypeCode_var` **type**

This member represents the exception's IDL type.

`CORBA::String_var` **version**

This member represents the exception's version.

FixedDef

`CORBA::`**FixedDef** `public CORBA::IDLType, public CORBA::Object`

This interface is used to represent a fixed definition that is stored in the Interface Repository.

Methods

`CORBA::UShort digits();`

This method sets the number of digits for the fixed type.

`void digits (CORBA::UShort _digits);`

This method sets the attribute for fixed type.

`CORBA::Short scale ();`

This method sets the scale for the fixed type.

`void scale (CORBA::Short _scale);`

This method sets the attribute for the fixed type.

FullInterfaceDescription

`struct` **CORBA::FullInterfaceDescription**

The `FullInterfaceDescription` structure describes an interface that is stored in the interface repository.

FullInterfaceDescription members

`CORBA::String_var` **Name**

This member represents the name of the interface.

`CORBA::String_var` **`id`**

This member represents the repository id of the interface.

`CORBA::String_var` **`defined_in`**

This member represents the name of the module or interface in which this interface is defined.

`CORBA::String_var` **`version`**

This member represents the interface's version.

`CORBA::OpDescriptionSeq` **`operations`**

This member represents a list of operations offered by this interface.

`CORBA::AttrDescriptionSeq` **`attributes`**

This member represents a list of attributes contained in this interface.

`CORBA::RepositoryIdSeq` **`base_interfaces`**

This member represents a list of interfaces from which this interface inherits.

`CORBA::RepositoryIdSeq derived_interfaces`

This member represents a list of interfaces derived from this interface.

`CORBA::TypeCode_var` **`type`**

This member represents this interface's `TypeCode`.

`CORBA::Boolean` **`is_abstract`**

Indicates whether or not this interface is abstract.

FullValueDescription

`struct CORBA::`**`FullValueDescription`**

This structure is used to represent a full value definition that is stored in the Interface Repository.

Variables

`CORBA::String_var` **`name`**

This variable represents name of the valuetype.

`CORBA::String_var` **`id`**

This variable represents the repository id of the valuetype.

`CORBA::Boolean` **`is_abstract`**

If this variable is set to true, specifies an abstract valuetype.

`CORBA::Boolean` **`is_custom`**

If this variable is set to true, specifies custom marshalling for the valuetype.

`CORBA::String_var` **`defined_in`**

This variable represents the repository Id of the module in which this valuetype is defined.

`CORBA::String_var` **`version`**

This variable represents the valuetype's version.

`CORBA::OpDescriptionSeq` **`operations`**

This variable represents the list of operations offered by the valuetype.

`CORBA::AttrDescriptionSeq` **`attributes`**

This variable represents valuetype's list of valuetype's member attributes.

`CORBA::.ValueMemberSeq` **`members`**

This variable represents an array of value definitions.

IDLType

`CORBA::InitializerSeq` **`initializers`**

This variable represents an array of initializers.

`CORBA::RepositoryIdSeq` **`supported_interfaces`**`;`

This variable represents the list of supported interfaces.

`CORBA::RepositoryIdSeq` **`abstract_base_values`**`;`

This variable represents the list of abstract value types from which this valuetype inherits.

`CORBA::Boolean` **`is_truncatable`**`;`

This variable is set to true, if the value can be safely truncated to its base valuetype.

`CORBA::String_var` **`base_values`**`;`

This variable represents a description of the value type from which this valuetype inherits.

`CORBA::TypeCode_var` **`type`**

This variable represents the valuetype's IDL type code.

`class` **`CORBA::IDLType`** `: public CORBA::IRObject, public CORBA::Object`

The `IDLType` class provides an abstract interface that is inherited by all interface repository definitions that represent IDL types. This class provides

a method for returning an object's `Typecode`, which identifies the object's type. The `IDLType` is unique; the `Typecode` is not.

Include file

You should include the file **corba.h** when using this class.

```
interface IDLType:IRObject {
  readonly attribute TypeCode type;
};
```

IDLType methods

`CORBA::Typecode_ptr` **`type`**`();`

This method returns the typecode of the current `IRObject`.

InterfaceDef

`class` **`CORBA::InterfaceDef`** `: public CORBA::Container, public CORBA::Contained, public CORBA::IDLType`

The `InterfaceDef` class is used to define an ORB object's interface that is stored in the interface repository.

For more information, see "Container", "Contained", and "IDLType".

Include file

You should include the file **corba.h** when you use this class.

```
interface InterfaceDef: Container, Contained, IDLType {
typedef sequence<RepositoryId> RepositoryIdSeq;
typedef sequence<OperationDescription> OpDescriptionSeq;
typedef sequence<AttributeDescription> AttrDescriptionSeq;
  attribute InterfaceDefSeq base_interfaces;
  attribute boolean is_abstract;
  readonly attribute InterfaceDefSeq derived_interfaces
  boolean is_a(in RepositoryId interface_id);
  struct FullInterfaceDescription {
Identifier name;
RepositoryId id;
RepositoryId defined_in;
String_var version;
OpDescriptionSeq operations;
AttrDescriptionSeq attributes;
RepositoryIdSeq base_interfaces;
RepositoryIdSeq derived_interfaces;
TypeCode type;
  boolean is_abstract;
};

FullInterfaceDescription describe_interface();

  AttributeDef create_attribute(
    in RepositoryId id,
    in Identifier name,
    in String_var version,
    in IDLType type,
    in CORBA::AttributeMode mode
  );
  OperationDef create_operation(
    in RepositoryId id,
    in Identifier name,
    in String_var version,
    in IDLType result,
    in OperationMode mode,
```

```
      in ParDescriptionSeq params,
      in ExceptionDefSeq exceptions,
      in ContextIdSeq contexts
    );
    struct InterfaceDescription {
      Identifier name;
      RepositoryId id;
      RepositoryId defined_in;
      String_var version;
      RepositoryIdSeq base_interfaces;
      boolean is_abstract;
    };
  };
```

InterfaceDef methods

```
CORBA::InterfaceDefSeq *base_interfaces();
```

This method returns a list of interfaces from which this class inherits.

```
void base_interfaces(const CORBA::InterfaceDefSeq&
  val);
```

This method sets the list of the interfaces from which this class inherits.

Parameter	Description
`val`	The list of interfaces from which this interface inherits.

```
CORBA::AttributeDef_ptr create_attribute(const char *
  id, const char * name, const CORBA::String_var&
  version, CORBA::IDLType_ptr type,
  CORBA::AttributeMode mode);
```

This method returns a pointer to a newly created AttributeDef that is contained in this object. The `id`, `name`, `version`, `type`, and `mode` are set to the specified value.

Parameter	Description
`id`	The interface id to use.
`name`	The interface name to use.
`version`	The interface version to use.
`mode`	The interface mode. See "AttributeMode" on page 7-4 for a list of possible values.

```
CORBA::OperationDef_ptr create_operation(const char
  *id, const char *name, CORBA::String_var& version,
  CORBA::IDLType_ptr result, CORBA::OperationMode mode,
  const CORBA::ParDescriptionSeq& params, const
  CORBA::ExceptionDefSeq& exceptions, const
  CORBA::ContextIdSeq& contexts);
```

This method creates a new `OperationDef` that is contained by this object, using the specified parameters. The `defined_in` attribute of the newly created `OperationDef` is set to identify this `InterfaceDef`.

Parameter	Description
`id`	The interface id for this operation.
`name`	The name of this operation.
`version`	The operation's version.

Parameter	Description
result	The IDL type returned by the operation.
mode	The mode of this operation—oneway or normal.
params	The list of parameters to pass to this operation.
exceptions	The list of exceptions raised by this operation.
contexts	Context lists are names of values expected in context and passed along with the request.

```
CORBA::InterfaceDef::FullInterfaceDescription
  *describe_interface();
```

This method returns a `FullInterfaceDescription`, which describes this object's interface.

```
CORBA::Boolean is_a(const char * interface_id);
```

This method returns true if this interface is identical to or inherits from, directly or indirectly, from the specified interface.

Parameter	Description
interface_id	The id of the interface to be checked against this interface.

InterfaceDescription

```
struct:CORBA:: InterfaceDescription
```

This structure describes an object that is stored in the interface repository.

InterfaceDescription members

```
CORBA::String_var name
```

The name of the interface.

```
CORBA::String_var id
```

The interface's repository identifier.

```
CORBA::String_var defined_in
```

The name of the repository Id in which the interface is defined.

```
CORBA::String_var version
```

The interface's version.

```
CORBA::RepositoryIdSeq  base_interfaces
```

A list of base interfaces for this interface.

```
CORBA::Boolean is_abstract
```

Indicates whether or not this interface is abstract.1

IRObject

`class` **`IRObject`** `: CORBA::Object`

The `IRObject` class offers the most generic interface for interface repository objects. The `Container` class, IDLType, Contained, and others are derived from this class.

Include file

You should include the file **corba.h** when you use this class.

```
interface IRObject {
  readonly attribute DefinitionKind def_kind;
    void destroy();
};
```

IRObject methods

`CORBA::DefinitionKind` **`def_kind`**`();`

This method returns the type of this interface repository object. See "DefinitionKind" for a list of possible types.

`void` **`destroy()`**`;`

This method deletes this object from the interface repository. If this object is a `Container`, all of its contents will also be deleted. If the object is currently contained by another object, it will be removed. The `destroy` method returns the `Exception(CORBA::BAD_PARAM)` when invoked on a `PrimitiveDef` or `Repository` object. The `Repository` class is described in "Repository".

ModuleDef

`class` **`ModuleDef`** `: CORBA::Container,CORBA::Contained`

The class is used to represent an IDL module in the interface repository.

ModuleDescription

`struct` **`ModuleDescription`**

The `ModuleDescription` structure describes a module that is stored in the interface repository.

ModuleDescription members

`CORBA::String_var` **`name`**

This member represents the name of the module.

`CORBA::String_var` **`id`**

This member represents the repository id of the module.

`CORBA::String_var` **`defined_in`**

This member represents the name of the repository Id in which this module is defined.

`CORBA::String_var` **`version`**

This member represents the module's version.

NativeDef

`class CORBA::`**`NativeDef`**

This interface is used to represent a native definition that is stored in the Interface Repository.

OperationDef

`class` **`CORBA::OperationDef`** `: public virtual CORBA::Contained, public CORBA::Object`

The `OperationDef` class contains information about an interface operation that is stored in the interface repository. This class is derived from the `Contained` class, which is described in "Contained". The inherited `describe` method returns a `OperationDescription` structure that provides complete information on the operation.

Include file

You should include the file **corba.h** when you use this class.

```
interface OperationDef: Contained {
  typedef sequence<ParameterDescription> ParDescriptionSeq;
  typedef Identifier ContextIdentifier;
  typedef sequence<ContextIdentifier> ContextIdSeq;
  typedef sequence<ExceptionDef> ExceptionDefSeq;
  typedef sequence<ExceptionDescription> ExcDescriptionSeq;
  readonly attribute TypeCode result;
  attribute IDLType result_def;
  attribute ParDescriptionSeq params;
  attribute CORBA::OperationMode mode;
  attribute ContextIdSeq contexts;
  attribute ExceptionDefSeq exceptions;
readonly attribute OperationKind bind;
};
struct OperationDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  String_var version;
  TypeCode result;
  OperationMode mode;
  ContextIdSeq contexts;
  ParDescriptionSeq parameters;
  ExcDescriptionSeq exceptions;
};
```

OperationDef methods

`CORBA::ContextIdSeq * contexts();`

This method returns a list of context identifiers that apply to the operation.

`void context(const CORBA::ContextIdSeq& val);`

This method sets the list of the context identifiers that apply to this operation.

Parameter	Description
`val`	The list of context identifiers.

`CORBA::ExceptionDefSeq * exceptions();`

This method returns a list of the exception types that can be raised by this operation.

`void exceptions(const CORBA::ExceptionDefSeq& val);`

This method sets the list of exception types that may be raised by this operation.

Parameter	Description
`val`	The list of exceptions that this operation may raise.

`CORBA::OperationMode mode();`

This method returns the mode of the operation represented by this `OperationDef`. The mode may be normal or oneway. Operations that have a normal mode are synchronous and return a value to the client application. Oneway operations do not block and no response is sent from the object implementation to the client.

`void mode(CORBA::OperationMode val);`

This method sets the mode of the operation.

Parameter	Description
`val`	The desired mode of this operation, either `OP_ONEWAY` or `OP_NORMAL`. See "OperationMode" on page 7-26 for more information.

`CORBA::ParDescriptionSeq * params();`

This method returns a pointer to a list of `ParameterDescription` structures that describe the parameters to this `OperationDef`.

`void params(const CORBA::ParDescriptionSeq& val);`

This method sets the list of the `ParameterDescription` structures for this `OperationDef`. The order of the structures is significant and should correspond to the order defined in the IDL definition for the operation.

Parameter	Description
`val`	The list of ParameterDescription structures.

`CORBA::TypeCode_ptr` **`result`**`();`

This method returns a pointer to a `TypeCode` representing the type of the value returned by this `Operation`. The `TypeCode` is a read-only attribute.

`CORBA::IDLType_ptr` **`result_def`**`();`

This method returns a pointer to the definition of the IDL type returned by this `OperationDef`.

`void` **`result_def`**`(CORBA::IDLType_ptr` **`val`**`);`

This method sets the definition of the type returned by this `OperationDef`.

Parameter	Description
`val`	A pointer to the type definition to use

OperationDescription

`struct CORBA::`**`OperationDescription`**

The `OperationDescription` structure describes an operation that is stored in the interface repository.

OperationDescription members

`CORBA::String_var` **`name`**

This variable represents the name the of the operation.

`CORBA::String_var` **`id`**

This variable represents the repository id of the operation.

`CORBA::String_var` **`defined_in`**

This variable represents the repository Id of the interface or valuetype in which this operation is defined.

`CORBA::String_var` **`version`**

This variable represents the operation’s version.

`CORBA::TypeCode_var` **`result`**

This variable represents the operation’s result.

`CORBA::OperationMode` **`mode`**

This variable represents the operation’s mode.

`CORBA::ContextIdSeq` **`contexts`**

This variable represents the operation’s associated context list.

`CORBA::ParameterDescriptionSeq` **`parameters`**

This variable represents the operation’s parameters.

`CORBA::ExceptionDescriptionSeq` **`exceptions`**

This variable represents the exceptions that this operation may raise.

OperationMode

`enum CORBA:`**`OperationMode`**

The enumeration defines the values used to represent the mode of an operation; either oneway or normal. Oneway operations are those for which the client application does not expect a response. Normal requests involve a response being sent to the client by the object implementation that contains the results of the request.

OperationMode values

Constant	Represents
`OP_NORMAL`	A normal operation request.
`OP_ONEWAY`	A one-way operation request.

ParameterDescription

`struct CORBA::`**`ParameterDescription`**

The `ParameterDescription` structure describes a parameter for an operation that is stored in the interface repository.

ParameterDescription members

`CORBA::String_var` **`name`**

This member represents the name of the parameter.

`CORBA::TypeCode_var` **`type`**

This member represents the parameter's typecode.

`CORBA::IDLType_var` **`type_def`**

This member represents the parameter's IDL type.

`CORBA::ParameterMode` **`mode`**

This member represents the parameter's mode.

ParameterMode

`enum CORBA::`**`ParameterMode`**

The enumeration defines the values used to represent the mode of a parameter for an operation.

ParameterMode values

Constant	Represents
PARAM_IN	Parameter is for input from the client to the server.
PARAM_OUT	Parameter is for output of results from the server to the client.
PARAM_INOUT	Parameter may be used for both input from the client and output from the server.

PrimitiveDef

```
class PrimitiveDef : public CORBA::IDLType, public
  CORBA::Object
```

The class is used to represent a primitive (such as an `int` or a `long`) that is stored in the interface repository. It provides a method for retrieving the kind of primitive that is being represented.

PrimitiveDef methods

```
CORBA::PrimitiveKind kind();
```

This method returns the kind of primitive represented by this object.

PrimitiveKind

```
enum CORBA::PrimitiveKind
```

The `PrimitiveKind` enumeration contains the constants that define the primitive types of objects that may be stored in the interface repository.

PrimitiveKind values

Constant	Represents
pk_null	Null value
pk_void	Void
pk_short	Short
pk_long	Long
pk_ushort	Unsigned short
pk_ulong	Unsigned long
pk_float	Float
pk_double	Double
pk_boolean	Boolean
pk_char	Character
pk_octet	Octet
pk_any	Any
pk_TypeCode	TypeCode
pk_Principal	Principal
pk_string	String

Constant	Represents
pk_objref	Object reference
pk_longlong	Long long
pk_ulonglong	Unsigned long long
pk_longdouble	Long double
pk_wchar	Unicode character
pk_wstring	Unicode string

Repository

```
class Repository : public CORBA::Container, public
  CORBA::Object
```

The `Repository` class provides access to the interface repository and is derived from the `Container` class, described in "Container".

Include file

You should include the file **corba.h** when using this class.

```
interface Repository: Container {
  Contained lookup_id(in RepositoryId search_id);
  PrimitiveDef get_primitive(in CORBA::PrimitiveKind kind);
  StringDef create_string(in unsigned long bound);
  WStringDef create_wstring (in unsigned long bound);
  SequenceDef create_sequence(
    in unsigned long bound,
    in IDLType element_type
  };
  ArrayDef create_array(
    in unsigned long length,
    in IDLType element_type
  );
  FixedDef create_fixed(
    in unsigned short digits,
    in short scale
  );
};
```

Repository methods

```
CORBA::ArrayDef_ptr create_array(CORBA::ULong length,
  CORBA::IDLType_ptr element_type);
```

This method creates a new `ArrayDef` and returns a pointer to that object.

Parameter	Description
length	The maximum number of elements in the array. This value must be greater than zero.
element_type	The `IDLType` of the elements stored in the array.

```
CORBA::SequenceDef_ptr create_sequence(CORBA::ULong
  bound, CORBA::IDLType_ptr element_type);
```

This method creates a new `SequenceDef` object and returns a pointer to that object.

Parameter	Description
`bound`	The maximum number of items in the sequence. This value must be greater than zero.
`element_type`	A pointer to the `IDLType` of the items stored in the sequence.

```
CORBA::StringDef_ptr create_string(CORBA::Ulong bound);
```

This method creates a new `StringDef` object and returns a pointer to that object.

Parameter	Description
`bound`	The maximum length of the string. This value must be greater than zero.

```
CORBA::StringDef_ptr create_wstring(CORBA::Ulong
  bound);
```

This method creates a new `WstringDef` object and returns a pointer to that object.

Parameter	Description
`bound`	The maximum length of the string. This value must be greater than zero.

```
CORBA::PrimitiveDef_ptr
  get_primitive(CORBA::PrimitiveKind kind);
```

This method returns a reference to a `PrimitiveKind`.

Parameter	Description
kind	The reference to be returned.

```
CORBA::Contained_ptr lookup_id(const char * search_id);
```

This method searches for an object in the interface repository that matches the specified search id. If no match is found, a NULL value is returned.

Parameter	Description
`search_id`	The identifier to use for the search.

```
CORBA::FixedDef_ptr create_fixed(CORBA::UShort digits,
  CORBA::Short scale)
```

This method sets the number of digits and the scale for the fixed type.

Parameter	Description
`Ushort digits`	The number of digits for the fixed type.
`short scale`	The scale of the fixed type

SequenceDef

```
class SequenceDef : public CORBA::IDLType, public
  CORBA::Object
```

The class is used to represent a sequence that is stored in the interface repository. This interface provides methods for setting and retrieving the sequence's bound and element type.

SequenceDef methods

```
CORBA::ULong bound()
```

This method returns the bounds of the sequence.

```
void bound(CORBA::LUong bound)
```

Should this be ULong? And where does "members" in the table below fit in? Should the parameter be "bound" as in the StringDef equivalent below?

This method sets the bound of the sequence.

Parameter	Description
`members`	The list of members.

```
CORBA::TypeCode_ptr element_type() ;
```

This method returns a `TypeCode` representing the type of elements in this sequence.

```
CORBA::IDLType_ptr element_type_def() ;
```

This method returns the IDL type of the elements stored in this sequence.

```
void element_type_def(CORBA::IDLType_ptr
  element_type_def);
```

This method sets the IDL type for the elements stored in this sequence.

Parameter	Description
element_type_def	The IDL type to set.

StringDef

```
class StringDef : public CORBA::IDLType, public
  CORBA::Object
```

The class is used to represent a `String` that is stored in the interface repository. This interface provides methods for setting and retrieving the bounds of the string.

StringDef methods

`CORBA::ULong` **`bound`**`() ;`

This method returns the bounds of the `String`.

`void` **`bound`**`(CORBA::ULong` **`bound`**`) ;`

This method sets the bounds of the `String`.

Parameter	Description
`bound`	The list of members.

StructDef

`class` **`StructDef`** `: public CORBA::TypedefDef, public CORBA::Container, public CORBA::Object`

The class is used to represent a structure that is stored in the interface repository.

StructDef methods

`CORBA::StructMemberSeq *`**`members`**`() ;`

This method returns the structure's list of members.

`void` **`members`**`(CORBA::StructMemberSeq&` **`members`**`) ;`

This method sets the structure's list of members.

Parameter	Description
`members`	The list of members.

StructMember

`struct CORBA::`**`StructMenber`**

This interface is used to define the member for the struct. It uses the name and type variables in the definition.

StructMember methods

`CORBA::String_var` **`name`**

This variable represents name of the type.

`CORBA::TypeCode_var` **`type`**

This variable represents the type's IDL type.

`CORBA::IDLType_var` **`type_def`**

This variable represents the IDL type's IDL type definition.

TypedefDef

```
class TypedefDef : public CORBA::Contained, public
  CORBA::IDLType, public CORBA::Object
```

This abstract base class represents a user-defined structure that is stored in the interface repository. The following interfaces all inherit from this interface:

- "AliasDef"
- "EnumDef"
- "ExceptionDef"
- "StructDef"
- "UnionDef"
- "WstringDef"

TypeDescription

structure **TypeDescription**

The `TypeDescription` structure contains the information that describes a type for an operation stored in the interface repository.

TypeDescription members

`CORBA::String_var` **`name`**

This member represents the name of the type.

`CORBA::String_var` **`id`**

This member represents the repository id of the type.

`CORBA::String_var` **`defined_in`**

This member represents the name of the module or interface in which this type is defined.

`CORBA::String_var` **`version`**

This member represents the type's version.

`CORBA::TypeCode_var` **`type`**

This member represents the type's IDL type.

UnionDef

```
class UnionDef : public CORBA::TypedefDef, public
  CORBA::Container, public CORBA::Object
```

The class is used to represent a `Union` that is stored in the interface repository. This class provides methods for setting and retrieving the union's list of members and discriminator type.

UnionDef methods

`CORBA::TypeCode_ptr` **`discriminator_type`**`() ;`

This method returns the `TypeCode` of the discriminator for the `Union`.

`CORBA::IDLType_ptr` **`discriminator_type_def`**`() ;`

This method returns the IDL type of the union's discriminator.

`void` **`discriminator_type_def`**`(CORBA::IDLType_ptr` **`discriminator_type_def`**`);`

This method sets the IDL type of the union's discriminator.

Parameter	Description
`discriminator_type_def`	The list of members.

`CORBA::UnionMemberSeq` ***`members`**`() ;`

This method returns the union's list of members.

`void` **`members`**`(CORBA::UnionMembersSeq&` **`members`**`) ;`

This method sets the union's list of members.

Parameter	Description
`members`	The list of members.

UnionMember

`struct CORBA::`**`UnionMember`**

The `UnionMember` struct contains information that describes a union that is stored in the interface repository.

UnionMember members

`CORBA::String_var` **`name`**

This member represents the name of the union.

`CORBA::Any` **`label`**

This member represents the label of the union.

`CORBA::TypeCode_var` **`type`**

This member represents the union's typecode.

`CORBA::IDLType_var` **`type_def`**

This member represents the union's IDL type.

ValueBoxDef

```
class ValueBoxDef public CORBA::Contained, public
  COBRA::IDLType, public CORBA::Object
```

This interface is used as a simple valuetype that contains a single public member of any IDL type. ValueBoxDef is a simplified version of ValueType:

```
public valuetype <IDLType> value;
```

This declaration is almost equal to valuetype boxed type `<IDLType>` but `ValueBoxDef` is not the same as simple `ValueTypeDef`.

Methods

```
CORBA::IDLType_ptr original_type_def ();
```

This method identifies the type being boxed.

```
void original_type_def (CORBA::IDLType_ptr
  original_type_def);
```

This method sets the type being boxed.

ValueDef

```
class CORBA::ValueDef public CORBA::Container, public
  CORBA::Contained, public CORBA::IDLType, public
  CORBA::Object
```

This interface is used to describe the IDL value type called a construct. This interface is very close to a class type. It represent a value definition that is stored in the Interface Repository.

Methods

```
CORBA::InterfaceDefSeq supported_interfaces( );
```

This method lists the interfaces which this value type supports.

```
void supported_interfaces (const
  CORBA::interfaceDefSeq& supported_interfaces);
```

This method sets the supported interfaces.

```
CORBA::InitializerSeq& initializers( );
```

This method lists the initializers.

```
void initializers (const CORBA::InitializerSeq&
  initializers);
```

This method sets the initializers.

CORBA.ValueDef_ptr **base_value**( );

This method describes the value types from which this value inherits.

```
void base_value (CORBA::ValueDef_ptr base_value);
```
This method sets the value types

```
CORBA.ValueDefSeq& abstract_base_values( );
```
This method lists the abstract value types from which this value inherits.

```
void abstract_base_values (const CORBA::ValueDef[Seq&
  abstract_base_values);
```
This method defines the abstract value type's base value.

```
CORBA::Boolean is_abstract( );
```
This method is true if the value is an abstract value type.

```
void is_abstract (CORBA::Boolean is_abstract);
```
This method sets the valuetype to be an abstract value type.

```
CORBA::Boolean is_custom( );
```
This method is true if the value uses custom marshalling.

```
void is_custom (CORBA::Boolean is_custom);
```
This method sets the custom marshalling for the value.

```
CORBA::Boolean is_trucatable( ):
```
This method is true if the value can be safely truncated from its base value.

```
void is_truncatable (CORBA::Boolean is_truncatable);
```
This method sets the truncated attribute for this value.

```
CORBA::Boolean is_a (const char* value_id);
```
This method returns true if the value on which it is invoked either is identical to or inherits, directly or indirectly from the interface or value defined by its ID parameter. Otherwise it returns false.

```
CORBA::ValueDef _ptr FullValueDescription*
  describe_value ();
```
This method returns a `FullValueDescription` describing the value including its operations and attributes.

```
CORBA::ValueMemberDef_ptr create_value_member (const
  Char* id, const Char* name, const Char* version,
  CORBA::IDLType_ptr type_def, CORBA::short access);
```
This method returns a new `ValueMemberDef` contained in the ValueDef on which it is invoked.

Parameter	Description
`id`	The repository id for this type.
`name`	The name of this type.
`version`	The object's version.

Parameter	Description
type_def	The value's IDL type.
short access	The access value.

```
CORBA::AttributeDef_ptr create_attribute (const Char*
  id, const Char* name, const Char* version,
  CORBA::IDLType_ptr type, CORBA::AttributeMode mode);
```

This method creates a new attribute definition for this valuetype and returns a new AttributeDef for it.

Parameter	Description
id	The repository id for this type.
name	The name of this type.
version	The object's version.
type	The type's IDL type.
mode	The object's mode.

```
CORBA::OperationDef_ptr create_operation (const Char*
  id, const Char*  name, const Char* version,
  CORBA::IDLType_ptr result, CORBA::OperationMode
  mode,const CORBA::ParDescriptionSeq& params, const
  CORBA::ExceptionDefSeq& exceptions, const
  CORBA::ContextIDSeq& contexts);
```

This method creates a new `Operation` for this valuetype and returns an `OperatioDef` for it.

Parameter	Description
id	The repository id for this type.
name	The name of this type.
version	The object's version.
result	The IDL type of the operation.
mode	The object's mode.
params	The list of the operation's parameters.
exceptions	The list of the operation's exceptions.
contexts	The list of the operation's contexts.

ValueDescription

```
struct CORBA::ValueDescription
```

This interface is used to represent a description of the value type that is stored in the Interface Repository.

Values

```
CORBA::String_var name
```

This variable represents name of the type.

```
CORBA::String_var id
```

This variable represents the repository id of the type.

`CORBA::Boolean` **`is_abstract`**

This variable is true if the value is an abstract value type.

`CORBA::Boolean` **`is_custom`**

This variable is true if the valuetype is custom marshalled.

`CORBA::String_var defined_in.`

This variable represents the repository Id of the module in which this type is defined.

`CORBA::String_var` **`version`**

This variable represents the type's version.

`CORBA::RepositoryIdSeq&` **`supported_interfaces`**

This variable represents the list of interfaces which this value type supports.

`CORBA::RepositoryIdSeq&` **`abstract_base_values`**

This variable represents the list of abstract value types from which this value inherits.

`CORBA::Boolean` **`is_truncatable`**

This variable represents the value type's setting for whether or not it can safely be truncated to its base value type.

`CORBA::String_var` **`base_value`**

This variable represents the value types from which this value inherits.

WstringDef

```
class WstringDef : public CORBA::IDLType, public
  CORBA::Object
```

This class is used to represent a Unicode string that is stored in the interface repository. It provides methods for setting and retrieving the bounds of the string.

WStringDef methods

`CORBA::ULong` **`bound`**`() ;`

This method returns the bounds of the `Wstring`.

`void` **`members`**`(CORBA::ULong` **`bound`**`) ;`

This method sets the bounds of the `Wstring`.

Parameter	Description
`members`	The list of members.

Activation Interfaces and Classes

This chapter describes the interfaces and classes used in the activation of object implementations.

ImplementationDef

Note

This feature is deprecated since VisiBroker 4.0.

`class` **`CORBA::ImplementationDef`**

The `ImplementationDef` class is used in the activation and deactivation of object implementations when using the Basic Object Adapter. It contains the object name, interface name, and reference data associated with an object implementation.

Include file

The **corba.h** file should be included when you use this class.

ImplementationDef methods

```
ImplementationDef(const char *interface_name, const
  char *object_name, const CORBA::ReferenceData& id);
```

This method creates an `ImplementationDef` object, initialized with the specified parameters.

Parameter	Description
`interface_name`	The interface name for the object implementation.
`object_name`	The object name for the object implementation.
`id`	The reference data for the object implementation. Reference data is not interpreted by the ORB and can contain any application-specific data you desire.

```
CORBA::ReferenceData_ptr id() const;
```

Returns the reference data identifier for the implementation. Reference data is not interpreted by the ORB and can contain any application-specific data you desire.

```
void id(const CORBA::ReferenceData& data);
```

Sets the reference data identifier for the implementation. Reference data is not interpreted by the ORB and can contain any application-specific data you desire.

Parameter	Description
`data`	The implementation's reference data identifier.

```
const char *interface_name() const;
```

This method returns a string containing the interface name of the object implementation.

```
void *interface_name(const char * val);
```

This method sets the interface name for the object implementation.

Parameter	Description
val	The new interface name for the object implementation.

```
const char *object_name() const;
```

This method returns a string containing the object name of the object implementation.

```
void *object_name(const char * val);
```

This method sets the object name for the object implementation.

Parameter	Description
val	The new object name for the object implementation.

StringSequence

```
class CORBA::StringSequence
```

The `StringSequence` class can be used to contain a list of arguments or environment variables.

Include file

The **corba.h** file should be included when you use this class.

StringSequence methods

```
CORBA::StringSequence(CORBA::ULong max = 0);
```

This method creates a `StringSequence` object with the specified length.

Parameter	Description
max	The maximum number of arguments in the list. The default length is 0.

```
CORBA::StringSequence(CORBA::ULong max, CORBA::ULong
  length, char **data, CORBA::Boolean release = 0);
```

This method creates a `StringSequence` object with the specified parameters.

Parameter	Description
max	The maximum number of arguments in the list.
length	The length of the sequence.
data	The strings that will make up the sequence.
release	If set to 1, all memory associated with the list will be released when this object is destroyed.

```
~CORBA::StringSequence();
```

This method destroys this object.

Methods

```
static char **allocbuf(CORBA::ULong nelemes);
```

This method allocates memory to accommodate the number of list elements specified.

Parameter	Description
nelemes	The number of elements in the list.

```
CORBA::ULong compare(const CORBA::StringSequence& seq1,
  const CORBA::StringSequence& seq2);
```

This method compares two `StringSequence` objects and returns 0 if they are equal; otherwise a non-zero value is returned.

Parameter	Description
seq1	The first object to be compared.
seq2	The second object to be compared.

```
static void freebuf(char **data);
```

This method frees the memory associated with the specified pointer.

Parameter	Description
data	The list memory to be freed.

```
static void freebuf_elems(char **data, CORBA::ULong
  nelemes);
```

This method allocates memory to accommodate the number of list elements specified.

Parameter	Description
data	The list memory to be freed.
nelemes	The number of elements.

```
CORBA::ULong hash(CORBA::StringSequence&);
```

This returns a hash value for the specified object.

Parameter	Description
StringSequence	The `StringSequence` for which a hash value is returned.

```
CORBA::ULong length() const;
```

This method returns the number of elements in the sequence.

```
void length(CORBA::ULong);
```

This method sets the number of elements in the sequence.

Parameter	Description
ULong	The new length

```
CORBA::ULong maximum() const;
```

This method returns the number of arguments in the list.

```
CORBA::StringSequence& operator=(const
  CORBA::StringSequence& seq);
```

This operator allows a `StringSequence` to be copied through assignment.

Parameter	Description
`seq`	The object to be copied.

```
CORBA::StringSequence& operator[](const CORBA::ULong
  index);
```

This operator allows arguments within a `StringSequence` to be accessed with an index.

Parameter	Description
`index`	The zero-based index of the desired string sequence.

```
static void *_release(CORBA::StringSequence* ptr);
```

This method releases the specified `StringSequence` object.

Parameter	Description
`obj`	The `ImplementationDef` object to be duplicated.

Naming Service Interfaces and Classes

This chapter describes the interfaces and classes for the VisiBroker-RT for C++ Naming Service.

NamingContext

`class` **`NamingContext`** `: public virtual CORBA_Object`

This object is used to contain and manipulate a list of names that are bound to ORB objects or to other `NamingContext` objects. Client applications use this interface to `resolve` or `list` all of the names within that context. Object implementations use this object to `bind` names to object implementations or to bind a name to a `NamingContext` object. IDL sample 9.1 shows the IDL specification for the `NamingContext`.

Example 56 IDL specification for the NamingContext interface

```
module CosNaming {
  interface NamingContext {
    void bind(in Name n, in Object obj)
      raises(NotFound, CannotProceed, InvalidName,
             AlreadyBound);
    void rebind(in Name n, in Object obj)
      raises(NotFound, CannotProceed, InvalidName);
    void bind_context(in Name n, in NamingContext nc)
      raises(NotFound, CannotProceed, InvalidName,
             AlreadyBound);
    void rebind_context(in Name n, in NamingContext nc)
      raises(NotFound, CannotProceed, InvalidName);
    Object resolve(in Name n)
      raises(NotFound, CannotProceed, InvalidName);
    void unbind(in Name n)
      raises(NotFound, CannotProceed, InvalidName);
    NamingContext new_context();
    NamingContext bind_new_context(in Name n)
      raises(NotFound, CannotProceed, InvalidName,
             AlreadyBound);
    void destroy()
      raises(NotEmpty);
    void list(in unsigned long how_many,
        out BindingList bl,
        out BindingIterator bi);
  };
};
```

NamingContext methods

`virtual void` **`bind`**`(const Name& -n, CORBA::Object _ptr` **`_obj`**`): raises(NotFound, CannotProceed, InvalidName, AlreadyBound);`

This method attempts to bind the specified `Object` to the specified `Name` by resolving the context associated with the first `NameComponent` and then binding the object to the new context using the following `Name`:

```
Name[NameComponent2,...,NameComponent(n-1),NameComponentn]
```

This recursive process of resolving and binding continues until the context associated with the `NameComponent` *(n–1)* is resolved and the actual name-to-object binding is stored. If parameter `n` is a simple name, the obj will be bound to `n` within this `NamingContext`.

Parameter	Description
`n`	A Name, initialized with the desired name for the object.
`obj`	The object to be named.

The following exceptions may be raised by this method.

Exception	Description
`NotFound`	The Name, or one of its components, could not be found.
`CannotProceed`	One of the `NameComponent` objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
`InvalidName`	The specified `Name` has zero name components or the `id` field of one of its name components is an empty string.
`AlreadyBound`	The `Name` on a `bind` or `bind_context` operation has already been bound to another object within the `NamingContext`.

```
virtual void rebind(const Name& _n, CORBA::Object _ptr
  _obj) raises(NotFound, CannotProceed, InvalidName);
```

This method is exactly the same as the `bind` method, except that the `AlreadyBound` exception will never be raised. If the specified `Name` has already been bound to another object, that binding is replaced by the new binding.

Parameter	Description
`n`	A `Name` structure, initialized with the desired name for the object.
`obj`	The object to be named.

The following exceptions may be raised by this method.

Exception	Description
`NotFound`	The Name, or one of its components, could not be found.
`CannotProceed`	One of the `NameComponent` objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
`InvalidName`	The specified `Name` has zero name components or the `id` field of one of its name components is an empty string.

```
virtual void bind_context(const Name& _n,
  NamingContext_ptr _nc) raises(NotFound,
  CannotProceed, InvalidName, AlreadyBound);
```

This method is identical to the `bind` method, except that the supplied `Name` will be associated with a `NamingContext`, not an arbitrary ORB object.

Parameter	Description
`n`	A `Name` structure, initialized with the desired name for the naming context. The first (*n–1*) `NameComponent` structures in the sequence must resolve to a `NamingContext`.
`nc`	The `NamingContext` object to be bound.

The following exceptions may be raised by this method.

Exception	Description
NotFound	The Name, or one of its components, could not be found.
CannotProceed	One of the NameComponent objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.
AlreadyBound	The Name on a bind or bind_context operation has already been bound to another object within the NamingContext.

```
virtual void rebind_context(const Name& _n,
  NamingContext _ptr _nc) raises(NotFound,
  CannotProceed, InvalidName);
```

This method is exactly the same as the bind_context method, except that the AlreadyBound exception will never be raised. If the specified Name has already been bound to another naming context, that binding is replaced by the new binding.

Parameter	Description
n	A Name structure, initialized with the desired name for the naming context.
nc	The NamingContext object to be rebound.

The following exceptions may be raised by this method.

Exception	Description
NotFound	The Name, or one of its components, could not be found.
CannotProceed	One of the NameComponent objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.

```
virtual CORBA::Object _ptr resolve(const Name& _n)
  raises(NotFound, CannotProceed, InvalidName);
```

This method attempts to resolve the specified Name and return an object reference. If parameter n is a *simple name*, it is resolved relative to this NamingContext.

If n is a *complex name,* it is resolved using the context associated with the first NameComponent. Next, the new context to resolve the following Name:

```
Name[NameComponent(2),...,NameComponent(n-1),NameComponentn]
```

This recursive process continues until the object associated with the *nth* NameComponent is returned.

Parameter	Description
n	A Name structure, initialized with the name for the desired object.

The following exceptions may be raised by this method.

Exception	Description
NotFound	The Name, or one of its components, could not be found.
CannotProceed	One of the NameComponent objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.

```
virtual void unbind(const Name& _n) raises(NotFound,
  CannotProceed, InvalidName);
```

This method performs the inverse operation of the bind method, removing the binding associated with the specified Name.

Parameter	Description
n	A Name structure, initialized with the desired name to be unbound.

The following exceptions may be raised by this method.

Exception	Description
NotFound	The Name, or one of its components, could not be found.
CannotProceed	One of the NameComponent objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.

```
virtual NamingContext_ptr new_context();
```

This method creates a new naming context. The newly created context will be implemented within the same server as this object. The new context is initially not bound to any Name.

```
virtual NamingContext_ptr bind_new_context(const Name&
  _n) raises(NotFound, CannotProceed, InvalidName,
  AlreadyBound);
```

This method creates a new context and binds it to the specified Name within this Context.

Parameter	Description
n	A Name structure, initialized with the desired Name for the newly created NamingContext object.

The following exceptions can be raised by this method.

Exception	Description
NotFound	The Name, or one of its components, could not be found.
CannotProceed	One of the NameComponent objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned NamingContext.
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.
AlreadyBound	The Name on a bind or bind_context operation has already been bound to another object within the NamingContext.

```
virtual void destroy() raises(NotEmpty);
```

This method deactivates this naming context. Any subsequent attempt to invoke operations on this object will raise a CORBA::OBJECT_NOT_EXIST runtime exception.

Before using this method, all `Name` objects that have been bound relative to this `NamingContext` should be unbound using the `unbind` method. Any attempt to destroy a `NamingContext` that is not empty will cause a `NotEmpty` exception to be raised.

```
virtual void list(CORBA::ULong _how_many,
  BindingList_out _bl, BindingIterator_out _bi)
```

This method returns all of the bindings contained by this context. Up to "*how_many*" Names are returned with the `BindingList`. Any left over bindings will be returned via the `BindingIterator`. The returned `BindingList` and `BindingIterator`, described in detail on "Binding and BindingList" and can be used to navigate the list of names.

Parameter	Description
`how_many`	The maximum number of Names to be returned in the list.
`bl`	A list of Names returned to the caller. The number of names in the list will not exceed `how_many`.
`bi`	An iterator for use in traversing the rest of the Names.

NamingContextExt

```
class NamingContextExt : public virtual NamingContect,
  public virtual CORBA Object
```

The `NamingContextExt` interface, which extends `NamingContext`, provides the operations required to use stringified names and URLs.

Example 57 IDL Specification for the NamingContextExt interface

```
module CosNaming {
  interface NamingContextExt {
    typedef string StringName;
    typedef string Address;
    typedef string URLString;

    StringName to_string(in Name n)
      raises(InvalidName);
    Name to_name(in StringName sn)
      raises(InvalidName);

    exception InvalidAddress {};

    URLString to_url(in Address addr, in StringName sn)
      raises(InvalidAddress, InvalidName);
    Object resolve_str(in StringName n)
      raises(NotFound, CannotProceed, InvalidName,
          AlreadyBound);
;
```

NamingContextExt methods

```
virtual char* to_string(const Name& _n)
  raises(InvalidName);
```

This operation returns the stringified representation of the specified Name.

Parameter	Description
n	A Name structure initialized with the desired name for object.

The following exceptions can be raised by this method.

Exception	Description
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.

```
virtual Name* to_name(const char* _sn);
  raises(InvalidName);
```

This operation returns a Name object for the specified stringified name.

Parameter	Description
sn	The stringified name of an object.

The following exceptions can be raised by this method.

Exception	Description
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.

```
virtual char* to_url(const char* _addr, const char*
  _sn); raises(InvalidAddress, InvalidName);
```

This operation returns a fully-formed string URL given the specified URL component and the stringified name.

Parameter	Description
addr	A URL component of the form "myhost.inprise.com:800". If the Address is empty, it is the local host.
sn	A stringified name of an object

The following exceptions can be raised by this method.

Exception	Description
InvalidAddress	The specified Address is malformed.
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.

```
virtual CORBA::Object _ptr resolve_str(const char* _n);
  raises(NotFound, CannotProceed, InvalidName,
  AlreadyBound);
```

This operation returns a Name object for the specified stringified name.

Parameter	Description
n	A stringified name of an object.

The following exceptions can be raised by this method.

Exception	Description
NotFound	The Name, or one of its components, could not be found.
CannotProceed	One of the NameComponent objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned NamingContext.
InvalidName	The specified Name has zero name components or the id field of one of its name components is an empty string.
AlreadyBound	The Name on a bind or bind_context operation has already been bound to another object within the NamingContext.

NamingLib

class **NamingLib**

The NamingLib interface provides the operations required to create the Initial Orphaned Naming Context Servant.

NamingLib methods

```
static POA_CosNaming::NamingContext *
  create_NamingServiceServant();
```

This operation returns a POA_CosNaming::NamingContext servant object which the user then activates on their POA.

After the user has registered their POA_CosNaming::NamingContext servant with their POA, this new NamingContext can then be made accessible via resolve_initial_references() by registering the NamingContext object with the ORB via the interface "void register_service_object(const char* objectId, CORBA_Object_ptr obj);".

Binding and BindingList

The Binding, BindingList, and BindingIterator interfaces are used to describe the name-object bindings contained in a NamingContext. The Binding struct encapsulates one such pair. The binding_name field represents the Name and the binding_type indicates whether the Name is bound to an ORB object or a NamingContext object.

The BindingList is a sequence of Binding structures contained by a NamingContext object. An example program that uses the BindingList can be found in "Using the Naming Service" in the VisiBroker-RT for C++ *Programmer's Guide*.

Example 58 IDL specification for the Binding structure

```
module CosNaming {
  enum BindingType {
    nobject,
    ncontext
  }
  struct Binding {
    Name binding_name;
    BindingType binding_type;
  };
  typedef sequence<Binding> BindingList;
};
```

BindingIterator

`class` **`BindingIterator`** `: public virtual CORBA_Object`

This object allows a client application to walk through the unbounded collection of bindings returned by the `NamingContext` operation `list`, described in "virtual void list(CORBA::ULong _how_many, BindingList_out _bl, BindingIterator_out _bi)". An example program that uses the `BindingIterator` can be found in the chapter "Using the Naming Service," of the VisiBroker-RT for C++ *Programmer's Guide*.

Example 59 IDL specification for the BindingIterator interface

```
module CosNaming {
  interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
         out BindingList b);
    void destroy();
  };
};
```

BindingIterator methods

`virtual CORBA::Boolean` **`next_one`**`(Binding_out` **`b_`**`);`

This method returns the next `Binding` from the collection. `CORBA::FALSE` is returned if the list has been exhausted. Otherwise, `CORBA::TRUE` is returned.

Parameter	Description
`b`	The next Binding object from the list.

`virtual CORBA::Boolean` **`next_n`**`(CORBA::ULong` **`_how_many`**`, BindingList_out` **`_b`**`);`

This method returns a `BindingList` containing the number of requested `Binding` objects from the list. The number of bindings returned may be less than the requested amount if the list is exhausted. `CORBA::FALSE` is returned when the list has been exhausted. Otherwise, `CORBA::TRUE` is returned.

Parameter	Description
`how_many`	The maximum number of Binding object desired.
`b`	A BindList containing no more than the requested number of Binding objects.

`virtual void` **`destroy`**`();`

This method destroys this object and releases the memory associated with the object. Failure to call this method will result in increased memory usage.

Event Service Interfaces and Classes

This chapter describes the interfaces and classes for the VisiBroker-RT for C++ Event Service.

EventLib

`class` **`EventLib`**

The `EventLib` interface provides the operations required to create a Event Channel Factory servant.

EventLib methods

```
static POA_CosEventChannelAdmin::EventChannelFactory
  *create_EventFactoryServant(CORBA::Ulong
  maxQueueLenght = MQL_DEFAULT);
```

This operation returns a `POA_CosEventChannelAdmin::EventChannelFactory` servant object which the user then activates on their POA.

After the user has registered their `POA_CosEventChannelAdmin::EventChannelFactory` servant with their POA, this new EventChannelFactory can then be made accessible via `resolve_initial_references()` by registering the `EventChannelFactory` obect with the ORB via the interface "void register_service_object(const char* objectId, CORBA_Object_ptr obj);".

ConsumerAdmin

`public interface` **`ConsumerAdmin`** `extends ConsumerAdminPOA`

This interface is used by consumer applications to obtain a reference to a proxy supplier object. This is the second step in connecting a consumer application to an `EventChannel`.

IDL definition

```
module CosEventChannelAdmin {
  interface ConsumerAdmin {
    ProxyPushConsumer obtain_push_supplier();
    ProxyPullConsumer obtain_pull_supplier();
  };
};
```

ConsumerAdmin methods

```
public ProxyPushSupplier obtain_push_supplier();
```

The `obtain_push_supplier` method is invoked if the calling consumer application is implemented using the push model. If the application is

implemented using the pull model, the `obtain_pull_supplier` method should be invoked.

`public ProxyPullSupplier` **`obtain_pull_supplier`**`();`

The returned reference is used to invoke either the `connect_push_consumer`, described in "ProxyPushConsumer", or the `connect_pull_consumer` method, described in "ProxyPullConsumer".

EventChannel

`public interface` **`EventChannel`**

The `EventChannel` provides the administrative operations for adding suppliers and consumers to the channel and for destroying the channel. For information on creating an event channel, see "EventChannelFactory".

Suppliers and consumers both use the `_bind` method to obtain an `EventChannel` reference. As with any `_bind` invocation, the caller can optionally specify the object name of the desired `EventChannel` as well as any desired bind options. These arguments can be passed to the supplier or consumer as initial parameters or they may be obtained from the Naming Service, if it is available. If the object name is not specified, a suitable `EventChannel` will be located by VisiBroker-RT for C++. Once a supplier or consumer is connected to an `EventChannel`, they may then invoke any of the `EventChannel` methods.

Methods

Example 60 Supplier binding to an EventChannel with the object name "power"

```
    ...
    CosEventChannelAdmin::EventChannel_var my_channel =
         CosEventChannelAdmin::EventChannel::_bind("power");
    CosEventChannelAdmin::SupplierAdmin_var =
         channel->for_suppliers();
    ...
}
```

`ConsumerAdmin` **`for_consumers`**`();`

This method returns a `ConsumerAdmin` object that can be used to add consumers to this `EventChannel`.

`SupplierAdmin` **`for_suppliers`**`();`

This method returns a `SupplierAdmin` object that can be used to add suppliers to this `EventChannel`.

`void` **`destroy`**`();`

This method destroys this `EventChannel`.

EventChannelFactory

`public interface` **`EventChannelFactory`**

The `EventChannelFactory` provides methods for creating, locating, and destroying event channels.

IDL definition

```
module CosEventChannelAdmin { interface EventCHannelFactory {
exception AlreadyExists(); exception ChannelsExist();
EventChannel create(CORBA::ULong = MQL_DEFAULT); EventChannel
create_by_name(in string name,CORBA::ULong =MQL_DEFAULT)
raises(AlreadyExists);
EventChannel lookup_by_name(in string name);
void remove(const char * name, CORBA::Boolean destroy); void
destroy()
raises(ChannelsExist)
};
};
```

EventChannelFactory methods

`EventChannel` **`create`**`();`

This method creates an anonymous, transient event channel.

`EventChannel` **`create_by_name`**`(in string name)`
`raises(AlreadyExists);`

This method creates a named, persistent event channel. If an event channel with the specified name has already been created, an `AlreadyExists` exception is raised.

`EventChannel` **`lookup_by_name`**`(in string name);`

This method attempts to return the `EventChannel` with the specified name. If no channel exits with the specified name, a `NULL` value is returned.

`void` **`remove`**`();`

This method removes the specified channel from the Event Channel Factory's list of managed channels, additionally the specified channel will be destroyed if the *destroy* paramater is true.

`void` **`destroy`**`();`

This method destroys this event channel factory. If any event channels exists which still belong to this factory, this method will raise a ChannelsExist exception.

Channels can be removed by calling the remove method on the factory.

ProxyPullConsumer

`public interface` **`ProxyPullConsumer`**

This interface is used by a pull supplier application and provides the `connect_pull_supplier` method for connecting the supplier's `PullSupplier`-derived object to the `EventChannel`. An `AlreadyConnected` exception will be raised if an attempt is made to connect the same proxy more than once.

IDL definition

```
module CosEventChannelAdmin {
  exception AlreadyConnected();
  interface ProxyPullConsumer : CosEventComm::PullConsumer {
    void connect_pull_supplier(in CosEventComm::PullSupplier
           pull_supplier)
       raises(AlreadyConnected);
  };
};
```

ProxyPushConsumer

`public interface` **ProxyPushConsumer**

This interface is used by a push supplier application and provides the `connect_push_supplier` method, used for connecting the supplier's `PushSupplier`-derived object to the `EventChannel`. An `AlreadyConnected` exception will be raised if an attempt is made to connect the same `proxy` more than once.

IDL definition

```
module CosEventChannelAdmin {
  exception AlreadyConnected();
  interface ProxyPushConsumer : CosEventComm::PushConsumer {
    void connect_push_supplier(in CosEventComm::PushSupplier
         push_supplier)
       raises(AlreadyConnected);
  };
};
```

ProxyPullSupplier

`public interface` **ProxyPullSupplier**

This interface is used by a pull consumer application and provides the `connect_pull_consumer` method, used for connecting the consumer's `PullConsumer`-derived object to the `EventChannel`. An `AlreadyConnected` exception will be raised if an attempt is made to connect the same `PullConsumer` more than once.

IDL definition

```
module CosEventChannelAdmin {
  exception AlreadyConnected();
  interface ProxyPullSupplier : CosEventComm::PullSupplier {
    void connect_pull_consumer(in CosEventComm::PullConsumer
           pull_consumer)
       raises(AlreadyConnected);
  };
};
```

ProxyPushSupplier

public interface **ProxyPushSupplier**

This interface is used by a push consumer application and provides the `connect_push_consumer` method, used for connecting the consumer's `PushConsumer`-derived object to the `EventChannel`. An `AlreadyConnected` exception will be raised if an attempt is made to connect the same `PushConsumer` more than once.

IDL definition

```
module CosEventChannelAdmin {
  exception AlreadyConnected();
  interface ProxyPushSupplier : CosEventComm::PushSupplier {
    void connect_push_consumer(in CosEventComm::PushConsumer
          push_consumer)
       raises(AlreadyConnected);
  };
};
```

PullConsumer

public interface **PullConsumer**

This interface is used to derive consumer objects that use the pull model of communication. The `pull` method is called by a consumer whenever it wants data from the supplier. A `Disconnected` exception will be raised if the supplier has disconnected.

The `disconnect_push_consumer` method is used to deactivate this consumer if the channel is destroyed.

IDL definition

```
module CosEventChannelAdmin {
  exception Disconnected {};
  interface PushConsumer {
    void push(in any data) raises(Disconnected);
    void disconnect_push_consumer();
  };
};
```

PushConsumer

public interface **PushConsumer**

This interface is used to derive consumer objects that use the push model of communication. The `push` method is used by a supplier whenever it has data for the consumer. A `Disconnected` exception will be raised if the consumer has disconnected.

IDL definition

```
module CosEventComm {
  exception Disconnected();
  interface PushConsumer {
    void push(in any data) raises(Disconnected);
    void disconnect_push_consumer();
  };
};
```

PullSupplier

`public interface` **`PullSupplier`**

This interface is used to derive supplier objects that use the pull model of communication.

IDL definition

```
module CosEventComm {
  interface PullSupplier {
    any pull() raises(Disconnected);
    any try_pull(out boolean has_event) raises(Disconnected);
    void disconnect_pull_supplier();
  };
};
```

PullSupplier methods

`any` **`pull`**`();`

This method blocks until there is data available from the supplier. The data is returned an `Any` type. If the consumer has disconnected, this method raises a `Disconnected` exception.

`any` **`try_pull`**`(out boolean has_event);`

This non-blocking method attempts to retrieve data from the supplier. When this method returns, `has_event` is set to `the` value `true` and the data is returned as an `Any` type if there was data available. If has_event is set to the value `false`, then no data was available and the return value will be `NULL`.

`void` **`disconnect_pull_supplier`**`();`

This method deactivates this pull server if the channel is destroyed.

PushSupplier

public interface **PushSupplier**

This interface is used to derive supplier objects that use the push model of communication. The `disconnect_push_supplier` method is used by the `EventChannel` to disconnect supplier when it is destroyed.

IDL definition

```
module CosEventComm {
  exception AlreadyConnected();
  interface PushSupplier {
    void disconnect_push_supplier();
  };
};
```

SupplierAdmin

`public interface` **`SupplierAdmin`**

This interface is used by supplier applications to obtain a reference to the proxy consumer object. This is the second step in connecting a supplier application to an `EventChannel`.

IDL definition

```
module CosEventChannelAdmin {
  interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ProxyPullConsumer obtain_pull_consumer();
  };
};
```

`public ProxyPushConsumer` **`obtain_push_consumer();`**

The `obtain_push_consumer` method is invoked if the supplier application is implemented using the push model. If the application is implemented using the pull model, the `obtain_pull_consumer` method should be invoked.

`public ProxyPullConsumer` **`obtain_pull_consumer`**`();`

The returned reference is used to invoke the either the `connect_push_supplier`, described in “ProxyPushSupplier”, or the `connect_pull_supplier` method, described in “ProxyPullSupplier”.

Portable Interceptor Interfaces and Classes for C++

This chapter describes the VisiBroker-RT for C++ implementation of Portable Interceptors interfaces and classes defined by the OMG Specification. For a complete description of these interfaces and classes, refer to OMG Final Adopted Specification, ptc/2001-04-03, Portable Interceptors.

Note

Refer to the Portable Interceptors chapter in the VisiBroker-RT for C++ *Developer's Guide* before using these interfaces.

Introduction

VisiBroker-RT for C++ provides a set of APIs known as interceptors which provide a way to plug in additional VisiBroker ORB behavior such as support for transactions and security. Interceptors are hooked into the VisiBroker ORB through services that can intercept the normal flow of execution of the VisiBroker ORB. The table below lists the types of interceptor that VisiBroker supports.

Table 5 Types of Interceptor

Interceptor Type	Description
Portable Interceptor	Portable Interceptors are OMG standardized feature that allow writing of portable code for interceptors and use it with different vendor ORBs.
4.x Interceptors	4.x Interceptors are VisiBroker-RT for C++ proprietary interceptors defined in VisiBroker version 4.x.

For more information about using the 4.x interceptors, refer to the section Using 4.x interceptor in the *VisiBroker-RT for C++Developer's Guide* and the 4.x Interceptor and object wrapper interfaces and classes for C++ in the *VisiBroker-RT for C++ Programmer's Reference*.

The table below lists the two forms of Portable Interceptor.

Table 6 Types of Portable Interceptor

Portable Interceptor type	Description
Request Interceptor	Request Interceptors can be used to enable VisiBroker ORB services to transfer context information between clients and servers. Request Interceptors are further divided into Client Request Interceptors and Server Request Interceptors.
IOR Interceptors	IOR interceptor is used to enable an VisiBroker ORB service to add information in an IOR describing the server's or object's ORB service related capabilities. For example, a security service (like SSL) can add its tagged component into the IOR so that clients recognizing that component can establish the connection with the server based on the information in the component.

For more information about using the Portable Interceptors, refer to the *Using Portable Interceptors* section in the *VisiBroker-RT for C++ Developer's Guide*.

ClientRequestInfo

```
class PortableInterceptor::ClientRequestInfo : public
  virtual RequestInfo
```

This class is derived from `RequestInfo`. It is passed to client side interceptors point.

Some methods on `ClientRequestInfo` are not valid at all interception points. The table below shows the validity of each attribute or method. If it is not valid, attempting to access it will result in a `BAD_INV_ORDER` being raised with a standard minor code of 14.

Table 7 ClientRequestInfo validity

	send_ request	send_poll	receive_ reply	receive_ exception	receive_ other
request_id	yes	yes	yes	yes	yes
operation	yes	yes	yes	yes	yes
arguments	yes[1]	no	yes	no	no
exception	yes	no	yes	yes	yes
contexts	yes	no	yes	yes	yes
operation_context	yes	no	yes	yes	yes
result	no	no	yes	no	no
response_expected	yes	yes	yes	yes	yes
sync_scope	yes	no	yes	yes	yes
reply_status	no	no	yes	yes	yes
forward_reference	no	no	no	no	yes[2]
get_slot	yes	yes	yes	yes	yes
get_request_service_context	yes	no	yes	yes	yes
get_reply_service_context	no	no	yes	yes	yes
target	yes	yes	yes	yes	yes
effective_target	yes	yes	yes	yes	yes

	send_ request	send_poll	receive_ reply	receive_ exception	receive_ other
effective_profile	yes	yes	yes	yes	yes
received_exception	no	no	no	yes	yes
received_exception_id	no	no	no	yes	no
get_effective_component	yes	no	yes	yes	yes
get_effective_components	yes	no	yes	yes	yes
get_request_policy	yes	no	yes	yes	yes
add_request_service_cont ext	yes	no	no	no	no

[1] When `ClientRequestInfo` is passed to `send_request()`, there is an entry in the list for every argument, whether in, inout, or out. But only the in and inout arguments will be available.

[2] If the `reply_status()` does not return `LOCATION_FORWARD`, accessing this attribute will raise `BAD_INV_ORDER` with a standard minor code of 14.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

ClientRequestInfo methods

```
virtual CORBA::Object_ptr target() = 0;
```

This method returns the object which the client called to perform the operation. See `effective_target()` below.

```
virtual CORBA::Object_ptr effective_target() = 0;
```

This method returns the actual object on which the operation will be invoked. If the `reply_status()` returns `LOCATION_FORWARD`, then on subsequent requests, `effective_target()` will contain the forwarded IOR while `target` will remain unchanged.

virtual IOP::TaggedProfile* **effective_profile**() = 0;

This method returns the profile, in the form of `IOP::TaggedProfile`, that will be used to send the request. If a location forward has occurred for this operation's object and that object's profile changed accordingly, then this profile will be that located profile.

```
virtual CORBA::Any* received_exception() = 0;
```

This method returns the data, in the form of `CORBA::Any`, that contains the exception to be returned to the client.

If the exception is a user exception which cannot be inserted into a `CORBA::Any` (e.g., it is unknown or the bindings don't provide the `TypeCode`), then this attribute will be a `CORBA::Any` containing the system exception `UNKNOWN` with a standard minor code of 1. However, the `RepositoryId` of the exception is available in the `received_exception_id` attribute.

```
virtual char* received_exception_id() = 0;
```

This method returns the ID of the `received_exception` to be returned to the client.

```
virtual IOP::TaggedComponent*
  get_effective_component(CORBA::ULong _id) = 0;
```

This methods returns the `IOP::TaggedComponent` with the given ID from the profile selected for this request.

If there is more than one component for a given component ID, it is undefined which component this operation returns. If there is more than one component for a given component ID, `get_effective_components()` will be called instead.

If no component exists for the given component ID, this operation will raise `BAD_PARAM` with a standard minor code of 28.

Parameter	Description
`id`	The ID of the component which is to be returned.

```
virtual IOP::TaggedComponentSeq*
  get_effective_components(CORBA::ULong _id) = 0;
```

This method returns all the tagged components with the given ID from the profile selected for this request. This sequence is in the form of an `IOP::TaggedComponentSeq`.

If no component exists for the given component ID, this operation will raise `BAD_PARAM` with a standard minor code of 28.

Parameter	Description
`id`	The ID of the components which are to be returned.

```
virtual CORBA::Policy_ptr
  get_request_policy(CORBA::ULong _type) = 0;
```

This method returns the given policy in effect for this operation.

If the 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, `INV_POLICY` with a standard minor code of 2 is raised.

Parameter	Description
`_type`	The type of policy which specifies the policy to be returned.

```
virtual void add_request_service_context(const
  IOP::ServiceContext& _service_context, CORBA::Boolean
  _replace) = 0;
```

This method allows Interceptors to add service contexts to the request.

There is no declaration of the order of the service contexts. They may or may not appear in the order that they are added.

Parameter	Description
`_service_context`	The `IOP::ServiceContext` to be added to the request.
`_replace`	Indicates the behavior of this method when a service context already exists with the given ID. If false, then `BAD_INV_ORDER` with a standard minor code of 15 is raised. If true, then the existing service context is replaced by the new one.

ClientRequestInterceptor

```
class PortableInterceptor::ClientRequestInterceptor :
  public virtual Interceptor
```

This `ClientRequestInterceptor` class is used to derive user-defined client side interceptor. A `ClientRequestInterceptor` instance is registered with the VisiBroker ORB (see "ORBInitializer").

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

ClientRequestInterceptor methods

```
virtual void send_request(ClientRequestInfo_ptr _ri)
  = 0;
```

This `send_request()` interception point allows an Interceptor to query request information and modify the service context before the request is sent to the server.

This interception point may raise a system exception. If it does, no other Interceptors' `send_request()` interception points are called. Those Interceptors on the Flow Stack are popped and their `receive_exception()` interception points are called.

This interception point may also raise a `ForwardRequest` exception (see "ForwardRequest"). If an Interception raises this exception, no other Interceptors' `send_request` methods are called. The remaining Interceptors in the Flow Stack are popped and have their `receive_other()` interception point called.

Parameter	Description
`_ri`	This is the `ClientRequestInfo` instance to be used by Interceptor.

```
virtual void send_poll(ClientRequestInfo_ptr _ri) = 0;
```

This `send_poll()` interception point allows an Interceptor to query information during a Time-Independent Invocation (TII) polling get reply sequence.However, as the VisiBroker ORB does not support TII, this `send_poll()` interception point will never be called.

Parameter	Description
`_ri`	This is the `ClientRequestInfo` instance to be used by Interceptor.

virtual void **receive_reply**(ClientRequestInfo_ptr **_ri**) = 0;

This `receive_reply()` interception point allows an Interceptor to query the information on a reply after it is returned from the server and before control is returned to the client.

This interception point may raise a system exception. If it does, no other Interceptors' `receive_reply()` methods are called. The remaining

Interceptors in the Flow Stack will have their `receive_exception()` interception point called.

Parameter	Description
`_ri`	This is the `ClientRequestInfo` instance to be used by Interceptor.

```
virtual void receive_exception(ClientRequestInfo_ptr
  _ri) = 0;
```

This `receive_exception()` interception point is called when an exception occurs. It allows an Interceptor to query the exception's information before it is raised to the client.

This interception point may raise a system exception. This has the effect of changing the exception which successive Interceptors popped from the Flow Stack receive on their calls to `receive_exception()`. The exception raised to the client will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may also raise a `ForwardRequest` exception (see "ForwardRequest"). If an Interceptor raises this exception, no other Interceptors' `receive_exception()` interception points are called. The remaining Interceptors in the Flow Stack are popped and have their `receive_other()` interception point called.

Parameter	Description
`_ri`	This is the `ClientRequestInfo` instance to be used by Interceptor.

```
virtual void receive_other(ClientRequestInfo_ptr _ri) =
  0;
```

This `receive_other()` interception point allows an Interceptor to query the information available when a request results in something other than a normal reply or an exception. For example, a request could result in a retry (e.g., a GIOP Reply with a `LOCATION_FORWARD` status was received); or on asynchronous calls, the reply does not immediately follow the request, but control will return to the client and an ending interception point will be called.

For retries, depending on the policies in effect, a new request may or may not follow when a retry has been indicated. If a new request does follow, while this request is a new request, with respect to Interceptors, there is one point of correlation between the original request and the retry: because control has not returned to the client, the request scoped `PortableInterceptor::Current` for both the original request and the retrying request is the same (see "Current").

This interception point may raise a system exception. If it does, no other Interceptors' `receive_other()` interception points are called. The remaining Interceptors in the Flow Stack are popped and have their `receive_exception()` interception point called.

This interception point may also raise a `ForwardRequest` exception (see "ForwardRequest"). If an Interceptor raises this exception, successive Interceptors' `receive_other()` methods are called with the new information provided by the `ForwardRequest` exception.

Parameter	Description
`_ri`	This is the `ClientRequestInfo` instance to be used by Interceptor.

Codec

`class` **`IOP::Codec`**

The formats of IOR components and service context data used by ORB services are often defined as CDR encapsulations encoding instances of IDL defined data types. The `Codec` provides a mechanism to transfer these components between their IDL data types and their CDR encapsulation representations.

A `Codec` is obtained from the `CodecFactory`. The `CodecFactory` is obtained through a call to `ORB::resolve_initial_references("CodecFactory")`.

Include file

Include the **IOP_c.hh** file when you use this class.

Codec member classes

class Codec::**InvalidTypeForEncoding** : public CORBA_UserException

This exception is raised by `encode()` or `encode_value()` when an invalid type is specified for the encoding.

class Codec::**FormatMismatch** : public CORBA_UserException

This exception is raised by `decode()` or `decode_value()` when the data in the octet sequence cannot be decoded into a `CORBA::Any`.

`class Codec::`**`TypeMismatch`** `: public CORBA_UserException`

This exception is raised by `decode_value()` when the given `TypeCode` does not match the given octet sequence.

Codec methods

```
virtual CORBA::OctetSequence* encode(const CORBA::Any&
  _data) = 0;
```

This method converts the given data in the form of a `CORBA::Any` into an octet sequence based on the encoding format effective for this `Codec`. This octet sequence contains both the `TypeCode` and the data of the type.

This operation may raise `InvalidTypeForEncoding`.

Parameter	Description
`_data`	The data, in the form of a `CORBA::Any`, to be encoded into an octet sequence.

```
virtual CORBA::Any* decode(const CORBA::OctetSequence&
  _data) = 0;
```

This method decodes the given octet sequence into a `CORBA::Any` object based on the encoding format effective for this `Codec`.

This method raises `FormatMismatch` if the octet sequence cannot be decoded into a `CORBA::Any`.

Parameter	Description
`_data`	The data, in the form of an octet sequence, to be encoded into a `CORBA::Any`.

```
virtual CORBA::OctetSequence* encode_value(const
  CORBA::Any& _data) = 0;
```

This method converts the given `CORBA::Any` object into an octet sequence based on the encoding format effective for this `Codec`. Only the data from the `CORBA::Any` is encoded, not the `TypeCode`.

This operation may raise `InvalidTypeForEncoding`.

Parameter	Description
`_data`	An octet sequence containing the data from the encoded `CORBA::Any`.

```
virtual CORBA::Any* decode_value(const
  CORBA::OctetSequence& _data, CORBA::TypeCode_ptr _tc)
  = 0;
```

This method decodes the given octet sequence into a `CORBA::Any` based on the given `TypeCode` and the encoding format effective for this `Codec`.

This method raises `FormatMismatch` if the octet sequence cannot be decoded into an`CORBA::Any`.

Parameter	Description
`_data`	The data, in the form of an octet sequence to be decoded into a `CORBA::Any`.
`tc`	The Typecode to be used to decode the data.

CodecFactory

```
class IOP::CodecFactory
```

This class is used to obtained a `Codec`. The `CodecFactory` is obtained through a call to `ORB::resolve_initial_references("CodecFactory")`.

Include file

Include the **IOP_c.hh** file when you use this class.

CodecFactory member

```
class CodecFactory::UnknownEncoding : public
  CORBA_UserException
```

This exception is raised if `CodecFactory` cannot create a `Codec`. See `create_codec()`function below.

CodecFactory method

```
virtual Codec_ptr create_codec(const Encoding& _enc)
  = 0;
```

This `create_codec()` method creates a `Codec` of the given encoding.

This method raises `UnknownEncoding` if this factory cannot create a Codec of the given encoding.

Parameter	Description
`enc`	This specifies the encoding to be used for creating a `Codec`.

Current

```
class PortableInterceptor::Current : public virtual
  CORBA::Current, public virtual CORBA_Object
```

The `Current` class is merely a slot table, the slots of which are used by each service to transfer their context data between their context and the request's or reply's service context.

Each service that wishes to use `Current` reserves a slot or slots at initialization time (see `allocate_slot_id()` on page 11-20) and uses those slots during the processing of requests and replies.

Before an invocation is made, `Current` is obtained via a call to

`ORB::resolve_initial_references("PICurrent").`

From within the interception points, the data on `Current` that has moved from the thread scope to the request scope is available via the `get_slot()` method on the `RequestInfo` object. A `Current` can still be obtained via `resolve_initial_references()`, but that is the Interceptor's thread scope `Current`.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

Current methods

```
virtual CORBA::Any* get_slot(CORBA::ULong _id);
```

A service can get the slot data it sets in `PICurrent` via the `get_slot()` method. The data is in the form of a `CORBA::Any` object.

If the given slot has not been set, a `CORBA::Any` containing a type code with a `TCKind` value of `tk_null` and no value is returned.

If `get_slot()` is called on a slot that has not been allocated, `InvalidSlot` is raised.

If `get_slot()` is called from within an ORB initializer (see "ORBInitializer" on page 11-17), `BAD_INV_ORDER` with a minor code of 14 is raised.

Parameter	Description
`_id`	The `SlotId` of the slot from which the data will be returned.

```
virtual void set_slot(CORBA::ULong _id, const
  CORBA::Any& _data);
```

A service sets data in a slot with `set_slot()`. The data is in the form of a `CORBA::Any object`.

If data already exists in that slot, it is overridden.

If `set_slot()` is called on a slot that has not been allocated, `InvalidSlot` is raised.

If `set_slot()` is called from within an ORB initializer (see "ORBInitializer") `BAD_INV_ORDER` with a minor code of 14 is raised.

Parameter	Description
`_id`	The `SlotId` of the slot from which the data will be set.
`_data`	The data, in the form of a `CORBA::Any` object, which will be set to the identified slot.

Encoding

`struct` **`IOP::Encoding`**

This structure defines the encoding format of a `Codec`. It details the encoding format, such as CDR Encapsulation encoding, and the major and minor versions of that format.

- `ENCODING_CDR_ENCAPS`, version 1.0;
- `ENCODING_CDR_ENCAPS`, version 1.1;
- `ENCODING_CDR_ENCAPS`, version 1.2;
- `ENCODING_CDR_ENCAPS` for all future versions of GIOP as they arise.

Include file

Include the **IOP_c.hh** file when you use this `struct`.

Encoding members

`CORBA::Short` **`format`**`;`

This member holds the encoding format for a `Codec`.

`CORBA::Octet` **`major_version`**`;`

This member holds the major version number for a `Codec`.

`CORBA::Octet` **`minor_version`**`;`

This member holds minor version number for a `Codec`.

ExceptionList

`class` **`Dynamic::ExceptionList`**

This class is used to hold exceptions information returned from the method `exceptions()` in the class `RequestInfo`. It is an implementation of variable-length array of type `CORBA::TypeCode`. The length of `ExceptionList` is available at run-time.

For more information, see `exceptions()` in "RequestInfo methods".

Include file

Include the **Dynamic_c.hh** file when you use this class.

ForwardRequest

```
class PortableInterceptor::ForwardRequest : public
  CORBA_UserException
```

The `ForwardRequest` exception is the means by which an Interceptor can indicate to the ORB that a retry of the request should occur with the new object given in the exception. This behavior of causing a retry only occurs if the ORB receives a `ForwardRequest` from an interceptor. If `ForwardRequest` is raised anywhere else it is passed through the ORB as is normal for a user exception.

If an Interceptor raises a `ForwardRequest` exception in response to a call of an interceptor, no other Interceptors are called for that interception point. The remaining Interceptors in the Flow Stack will have their appropriate ending interception point called: `receive_other()` on the client, or `send_other()` on the server. The `reply_status()` in the `receive_other()` or `send_other()` will return `LOCATION_FORWARD`.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

Interceptor

```
class PortableInterceptor::Interceptor
```

This is the base class from which all interceptors are derived.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

Interceptor methods

```
virtual char* name() = 0;
```

This method returns the name of the Interceptor. Each Interceptor may have a name which can be used to order the lists of Interceptors. Only one Interceptor of a given name can be registered with the VisiBroker ORB for each Interceptor type. An Interceptor may be anonymous, i.e., has an empty string as the `name` attribute. Any number of anonymous Interceptors may be registered with the VisiBroker ORB.

```
virtual void destroy() = 0;
```

This method is called during `ORB::destroy()`. When `ORB::destroy()` is called by an application, the VisiBroker ORB:

- waits for all requests in progress to complete;
- calls the `Interceptor::destroy()` method for each interceptor;
- completes destruction of the ORB.

Method invocations from within `Interceptor::destroy()` on object references for objects implemented on the ORB being destroyed result in undefined behavior. However, method invocations on objects implemented on VisiBroker ORB other than the one being destroyed are permitted. (This

means that the VisiBroker ORB being destroyed is still capable of acting as a client, but not as a server.)

IORInfo

```
class PortableInterceptor::IORInfo
```

The `IORInfo` interface provides the server side ORB service with access to the applicable policies during IOR construction and the ability to add components. The ORB passes an instance of its implementation of this interface as a parameter to `IORInterceptor::establish_components()`.

The table below defines the validity of each attribute or method in `IORInfo` in the methods defined in the `IORInterceptor`.

Table 8 IORInfo validity

	establish_ components	**components_ established**
get_effective_policy	yes	yes
add_component	yes	no
add_component_to_profile	yes	no
manager_id	yes	yes
state	yes	yes
adapter_template	no	yes
current_factory	no	yes

If an illegal call is made to an attribute or method in `IORInfo`, the `BAD_INV_ORDER` system exception is raised with a standard minor code value of 14.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

IORInfo methods

```
virtual CORBA::Policy_ptr
  get_effective_policy(CORBA::ULong _type) = 0;
```

An ORB service implementation may determine what server side policy of a particular type is in effect for an IOR being constructed by calling the `get_effective_policy()` method. When the IOR being constructed is for an object implemented using a POA, all Policy objects passed to the `PortableServer::POA::create_POA()` call that created that POA are accessible via `get_effective_policy`.

If a policy for the given type is not known to the ORB, then this method will raise `INV_POLICY` with a standard minor code of 3.

Parameter	**Description**
`_type`	The `CORBA::PolicyType` specifying the type of policy to return.

```
virtual void add_ior_component(const
  IOP::TaggedComponent& _a_component) = 0;
```

This method is called from `establish_components()` to add a tagged component to the set which will be included when constructing IORs. The components in this set will be included in all profiles.

Any number of components may exist with the same component ID.

Parameter	Description
`_a_component`	The `IOP::TaggedComponent` to be added.

```
virtual void add_ior_component_to_profile(const
  IOP::TaggedComponent&_a_component,
  CORBA::ULong _profile_id) = 0;
```

This method is called from `establish_components()` to add a tagged component to the set which will be included when constructing IORs. The components in this set will be included in the specified profile.

Any number of components may exist with the same component ID.

If the given profile ID does not define a known profile or it is impossible to add components to that profile, `BAD_PARAM` is raised with a standard minor code of 29.

Parameter	Description
`_a_component`	The `IOP::TaggedComponent` to be added.
`_profile_id`	The `IOP::ProfileId` of the profile to which this component will be added.

```
virtual CORBA::Long manager_id() = 0;
```

This method returns the attribute that provides an opaque handle to the manager of the adapter. This is used for reporting state changes in adapters managed by the same adapter manager.

```
virtual CORBA::Short state() = 0;
```

This method returns the current state of the adapter. This must be one of `HOLDING`, `ACTIVE`, `DISCARDING`, `INACTIVE`, `NON_EXISTENT`.

```
virtual ObjectReferenceTemplate_ptr adapter_template()
  = 0;
```

This method returns the attribute that provides a means to obtain an object reference template whenever an ior interceptor is invoked. There is no standard way to directly create an object reference template. The value of `adapter_template()` returns is the template created for the adapter policies and IOR interceptor calls to `add_component()` and `add_component_to_profile()`. The value of the `adapter_template()` returns is never changed for the lifetime of the object adapter.

virtual ObjectReferenceFactory_ptr **current_factory**() = 0;

This method returns the attribute provides access to the factory that will be used by the adapter to create object references. `current_factory()` initially has the same value as the `adapter_template` attribute, but this can be changed by setting `current_factory` toanothe r factory. All object references created by the object adapter must be created by calling the `make_object()` method on `current_factory`.

```
virtual void current_factory(ObjectReferenceFactory_ptr
  _current_factory) = 0;
```

This method sets the `current_factory` attribute. The value of the `current_factory` attribute that is used by the adapter can only be set during the call to the `components_established` method.

Parameter	Description
`_current_factory`	The `current_factory` object which is to be set.

IORInfoExt

```
class IORInfoExt : public PortableInterceptor::IORInfo
```

This is the VisiBroker extensions to Portable Interceptors to allow installing of a POA scoped Server Request Interceptor. This IORInfoExt interface is inherited from IORInfo interface and has additional methods to support POA scoped Server Request Interceptor.

Include file

Include the **PortableInterceptorExt_c.hh** file when you use this class.

IORInfoExt methods

```
virtual void add_server_request_interceptor(
  ServerRequestInterceptor_ptr _interceptor) = 0;
```

This method is used to add a POA-scoped server side request Interceptor to a service.

Parameter	Description
`_interceptor`	The `ServerRequestInterceptor` to be added.

```
virtual char* full_poa_name();
```

This method return the full POA name.

IORInterceptor

```
class PortableInterceptor::IORInterceptor : public
  virtual Interceptor
```

In some cases, a portable ORB service implementation may need to add information describing the server's or object's ORB service related capabilities to object references in order to enable the ORB service implementation in the client to function properly.

This is supported through the `IORInterceptor` and `IORInfo` interfaces.

The IOR Interceptor is used to establish tagged components in the profiles within an IOR.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

IORInterceptor methods

```
virtual void establish_components(IORInfo_ptr _info)
  = 0;
```

A server side ORB calls the `establish_components()` method on all registered `IORInterceptor` instances when it is assembling the list of components that will be included in the profile or profiles of an object reference.

This method is not necessarily called for each individual object reference. In the case of the POA, these calls are made each time `POA::create_POA()` is called. In other adapters, these calls would typically be made when the adapter is initialized.

The adapter template is not available at this stage since information (the components) needed in the adapter template is being constructed.

Parameter	Description
`_info`	The `IORInfo` instance used by the ORB service to query applicable policies and add components to be included in the generated IORs.

```
virtual void components_established(IORInfo_ptr _info)
  = 0;
```

After all of the `establish_components()` methods have been called, the `components_established()` methods are invoked on all registered IOR interceptors. The adapter template is available at this stage. The `current_factory` attribute may be get or set at this stage.

Any exception that occurs in `components_established()` is returned to the caller of `components_established()`. In the case of the POA, this causes the `create_POA` call to fail, and an `OBJ_ADAPTER` exception with a standard minor code of 6 is returned to the invoker of `create_POA()`.

Parameter	Description
`_info`	The `IORInfo` instance used by the ORB service to access applicable policies.

```
virtual void adapter_manager_state_changed(CORBA::Long
  _id, CORBA::Short _state) = 0;
```

Any time the state of an adapter manager changes, the `adapter_manager_state_changed()` method is invoked on all registered IOR interceptors.

If a state change is reported through `adapter_manager_state_changed()`, it is not reported through `adapter_state_changed()`.

Parameter	Description
`_id`	The `IORInfo` instance used by the ORB service to access applicable policies.
`_state`	The new state of the object adapter.

```
virtual void adapter_state_changed(const
  ObjectReferenceTemplateSeq&_templates,
  CORBA::Short _state) = 0;
```

Object adapter state changes are reported to this method any time the state of one or more adapters changes for reasons unrelated to adapter

manager state changes. The templates argument identifies the object adapters that have changed state by the template ID information. The sequence contains the adapter templates for all object adapters that have made the state transition being reported.

Parameter	Description
`_templates`	This identifies the object adapters that have changed state by the template ID information.
`_state`	The new state of the object adapter.

ORBInitializer

`class PortableInterceptor::ORBInitializer`

An Interceptor is registered by registering an associated `ORBInitializer` object which implements the `ORBInitializer` class. When an ORB is initializing, it will call each registered `ORBInitializer`, passing it an `ORBInitInfo` object which is used to register its Interceptor.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

ORBInitializer methods

`virtual void` **`pre_init`**`(ORBInitInfo_ptr` **`_info`**`) = 0;`

This method is called during ORB initialization. If it is expected that initial services registered by an interceptor will be used by other interceptors, then those initial services are registered at this point via calls to `ORBInitInfo::register_initial_reference()`.

Parameter	Description
`_info`	This object provides initialization attributes and methods by which Interceptors can be registered.

`virtual void` **`post_init`**`(ORBInitInfo_ptr` **`_info`**`) = 0;`

This method is called during ORB initialization. If a service must resolve initial references as part of its initialization, it can assume that all initial references will be available at this point.

Calling the `post_init()` methods is not the final task of ORB initialization. The final task, following the `post_init()` calls, is attaching the lists of registered interceptors to the ORB. Therefore, the ORB does not contain the interceptors during calls to `post_init()`. If an ORB-mediated call is made from within `post_init()`, no request interceptors will be invoked on that call. Likewise, if a method is performed which causes an IOR to be created, no IOR interceptors will be invoked.

Parameter	Description
`_info`	This object provides initialization attributes and methods by which Interceptors can be registered.

ORBInitInfo

```
class PortableInterceptor::ORBInitInfo
```

This `ORBInitInfo` class is passed to `ORBInitializer` object for registering interceptors.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

ORBInitInfo member classes

```
class DuplicateName : public CORBA_UserException;
```

Only one Interceptor of a given name can be registered with the ORB for each Interceptor type. If an attempt is made to register a second Interceptor with the same name, `DuplicateName` is raised.

An Interceptor may be anonymous, i.e., has an empty string as the name attribute. Any number of anonymous Interceptors may be registered with the ORB so, if the Interceptor being registered is anonymous, the registration operation will not raise `DuplicateName`.

```
class InvalidName: public CORBA_UserException
```

This exception is raised by `register_initial_reference()` and `resolve_initial_references()`. `register_initial_reference()` raises `InvalidName` if:

- this method is called with an empty string id; or
- this method is called with an id that is already registered, including the default names defined by OMG.

`resolve_initial_references()` raises `InvalidName` if the name to be resolved is invalid.

ORBInitInfo methods

```
virtual CORBA::StringSequence* arguments() = 0;
```

This method returns the arguments passed to `ORB_init()`. They may or may not contain the ORB's arguments.

```
virtual char* orb_id() = 0;
```

This method returns the ID of the ORB being initialized.

```
virtual IOP::CodecFactory_ptr codec_factory() = 0;
```

This method returns the `IOP::CodecFactory`. The `CodecFactory` is normally obtained via a call to `ORB::resolve_initial_references("CodecFactory")`, but since the ORB is not yet available and Interceptors, particularly when processing service contexts, will require a `Codec`, a means of obtaining a `Codec` is necessary during ORB initialization.

```
virtual void register_initial_reference(const char*
  _id, CORBA::Object_ptr _obj) = 0;
```

If this method is called with an id, "Y", and an object, YY, then a subsequent call to `register_initial_reference()` will return object YY.

This method is identical to `ORB::register_initial_reference()`. This same functionality exists here because the ORB, not yet fully initialized, is not yet available but initial references may need to be registered as part of Interceptor registration. The only difference is that the version of this method on the ORB uses PIDL (`CORBA::ORB::ObjectId` and `CORBA::ORB::InvalidName`) whereas the version in this interface uses IDL defined in this interface; the semantics are identical.

`register_initial_reference()` raises `InvalidName` if:

- this method is called with an empty string id; or
- this method is called with an id that is already registered, including the default names defined by OMG.

Parameter	Description
`_id`	The ID by which the initial reference will be known.
`_obj`	The initial reference itself.

```
virtual CORBA::Object_ptr
  resolve_initial_references(const char* _id) = 0;
```

This method is only valid during `post_init()`. It is identical to `ORB::resolve_initial_references()`. This same functionality exists here because the ORB, not yet fully initialized, is not yet available but initial references may be required from the ORB as part of Interceptor registration.

Parameter	Description
`_id`	The ID by which the initial reference will be known.

If the name to be resolved is invalid, `resolve_initial_references()` will raise `InvalidName`.

```
virtual void add_client_request_interceptor(
  ClientRequestInterceptor_ptr _interceptor) = 0;
```

This method is used to add a client side request Interceptor to the list of client side request Interceptors.

If a client side request Interceptor has already been registered with this Interceptor's name, `DuplicateName` will be raised.

Parameter	Description
`_interceptor`	The `ClientRequestInterceptor` to be added.

```
virtual void add_server_request_interceptor(
  ServerRequestInterceptor_ptr _interceptor) = 0;
```

This method is used to add a server side request Interceptor to the list of server side request Interceptors.

If a server side request Interceptor has already been registered with this Interceptor's name, `DuplicateName` is raised.

Parameter	Description
`_interceptor`	The `ServerRequestInterceptor` to be added.

```
virtual void add_ior_interceptor(IORInterceptor_ptr
  _interceptor) = 0;
```

This method is used to add an IOR Interceptor to the list of IOR Interceptors. If an IOR Interceptor has already been registered with this Interceptor's name, `DuplicateName` is raised.

Parameter	Description
`_interceptor`	The `IORInterceptor` to be added.

```
virtual CORBA::ULong allocate_slot_id() = 0;
```

This method returns the index to the slot which has been allocated.

A service calls `allocate_slot_id` to allocate a slot on `PortableInterceptor::Current`.

Note

While slot id's can be allocated within an ORB initializer, the slots themselves cannot be initialized. Calling `set_slot()` or `get_slot()` on the Current (see "Current") within an ORB initializer will raise a BAD_INV_ORDER with a minor code of 14.

```
virtual void register_policy_factory(CORBA::ULong
  _type, PolicyFactory_ptr _policy_factory) = 0;
```

This method registers a `PolicyFactory` for the given `PolicyType`.

If a `PolicyFactory` already exists for the given `PolicyType`, `BAD_INV_ORDER` is raised with a standard minor code of 16.

Parameter	Description
`_type`	The `CORBA::PolicyType` that the given PolicyFactory serves.
`_policy_factory`	The factory for the given `CORBA::PolicyType`.

Parameter

```
struct Dynamic::Parameter
```

This structure holds the parameter information. This structure is the element used in `ParameterList` (see "ParameterList" for more information).

Include file

Include the **Dynamic_c.hh** file when you use this `struct`.

Parameter members

```
CORBA::Any argument;
```

This member stores the parameter data in the form of `CORBA::Any`.

```
CORBA::ParameterMode mode;
```

This member specifies the mode of a parameter. Its value can be one of the enum values:

`PARAM_IN`, `PARAM_OUT` or `PARAM_INOUT`.

ParameterList

```
class Dynamic::ParameterList
```

This class is used to pass parameters information returned from the method `arguments()` in the class `RequestInfo`. It is an implementation of variable-length array of type `Parameter`. The length of `ParameterList` is available at run-time.

For more information, see `arguments()` in "RequestInfo methods".

Include file

Include the **Dynamic_c.hh** file when you use this class.

PolicyFactory

```
class PortableInterface::PolicyFactory
```

A portable ORB service implementation registers an instance of the `PolicyFactory` interface during ORB initialization. The POA is required to preserve any policy which is registered with `ORBInitInfo` in this manner.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

PolicyFactory method

```
virtual CORBA::Policy_ptr create_policy(CORBA::ULong
  _type, const CORBA::Any& _value) = 0;
```

The ORB calls `create_policy()` on a registered `PolicyFactory` instance when `CORBA::ORB::create_policy()` is called for the `PolicyType` under which the `PolicyFactory` has been registered. The `create_policy()` method then returns an instance of the appropriate interface derived from `CORBA::Policy` whose value corresponds to the specified `CORBA::Any`. If it cannot, it will raise an exception as described for `CORBA::ORB::create_policy()`.

Parameter	Description
`_type`	A `CORBA::PolicyType` specifying the type of policy being created.
`_value`	A `CORBA::Any` containing data with which to construct the `CORBA::Policy`.

RequestInfo

`class PortableInterceptor::RequestInfo`

This is the base class from which `ClientRequestInfo` and `ServerRequestInfo` are derived. Each interception point is given an object through which the Interceptor can access request information. client side and server side interception points are concerned with different information, so there are two information objects: `ClientRequestInfo` is passed to the client side interception points and `ServerRequestInfo` is passed to the server side interception points. But there is information that is common to both, so they both inherit from this common interface: `RequestInfo`.

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

RequestInfo methods

`virtual CORBA::ULong` **`request_id`**`() = 0;`

This method returns the ID which uniquely identifies an active request / reply sequence. Once a request / reply sequence is concluded this ID may be reused.

Note

This ID is not the same as the GIOP `request_id`. If GIOP is the transport mechanism used, then these IDs may very well be the same, but this is not guaranteed nor required.

`virtual char*` **`operation`**`() = 0;`

This method returns name of the operation being invoked.

`virtual Dynamic::ParameterList*` **`arguments`**`() = 0;`

This method returns a Dynamic::ParameterList containing the arguments on the operation being invoked. If there are no arguments, this attribute will be a zero length sequence.

`virtual Dynamic::ExceptionList*` **`exceptions`**`() = 0;`

This method returns a `Dynamic::ExceptionList` describing the `TypeCodes` of the user exceptions that this operation invocation may raise. If there are no user exceptions, this attribute will be a zero length sequence.

`virtual CORBA::StringSequence*` **`contexts`**`() = 0;`

This method returns a `CORBA::StringSequence` describing the contexts that may be passed on this operation invocation. If there are no contexts, this attribute will be a zero length sequence.

`virtual CORBA::StringSequence*` **`operation_context`**`() = 0;`

This method returns a `CORBA::StringSequence` containing the contexts being sent on the request.

```
virtual CORBA::Any* result() = 0;
```

This method returns the data, in the form of `CORBA::Any`, that contains the result of the operation invocation. If the operation return type is void, this attribute will be a `CORBA::Any` containing a type code with a `TCKind` value of `tk_void` and no value.

```
virtual CORBA::Boolean response_expected() = 0;
```

This method returns a boolean value which indicates whether a response is expected.

On the client, a reply is not returned when `response_expected()` is false, so `receive_reply()` cannot be called. `receive_other()` is called unless an exception occurs, in which case `receive_exception()` is called.

```
virtual CORBA::Short sync_scope() = 0;
```

This method returns an attribute, defined in the Messaging specification, is pertinent only when `response_expected()` is false. If `response_expected()` is true, the value of `sync_scope()` is undefined. It defines how far the request will progress before control is returned to the client. This attribute may have one of the following values:

- `Messaging::SYNC_NONE`
- `Messaging::SYNC_WITH_TRANSPORT`
- `Messaging::SYNC_WITH_SERVER`
- `Messaging::SYNC_WITH_TARGET`

On the server, for all scopes, a reply will be created from the return of the target operation call, but the reply will not return to the client. Although it does not return to the client, it does occur, so the normal server side interception points are followed (that is, `receive_request_service_contexts()`, `receive_request()`, `send_reply()` or `send_exception()`).

For `SYNC_WITH_SERVER` and `SYNC_WITH_TARGET`, the server does send an empty reply back to the client before the target is invoked. This reply is not intercepted by server side Interceptors.

```
virtual CORBA::Short reply_status() = 0;
```

This method returns an attribute which describes the state of the result of the operation invocation. Its value can be one of the following:

- `PortableInterceptor::SUCCESSFUL = 0`
- `PortableInterceptor::SYSTEM_EXCEPTION = 1`
- `PortableInterceptor::USER_EXCEPTION = 2`
- `PortableInterceptor::LOCATION_FORWARD = 3`
- `PortableInterceptor::TRANSPORT_RETRY = 4`

On the client:

- Within the `receive_reply` interception point, this attribute will only be `SUCCESSFUL`.
- Within the `receive_exception` interception point, this attribute will be either `SYSTEM_EXCEPTION` or `USER_EXCEPTION`.
- Within the `receive_other` interception point, this attribute will be any of: `SUCCESSFUL`, `LOCATION_FORWARD`, or `TRANSPORT_RETRY`. SUCCESSFUL means an asynchronous request returned successfully. `LOCATION_FORWARD` means that a reply came back with `LOCATION_FORWARD` as its status. `TRANSPORT_RETRY`means that the transport mechanism indicated a retry - a GIOP reply with a status of `NEEDS_ADDRESSING_MODE`, for instance.

On the server:

- Within the `send_reply` interception point, this attribute will only be `SUCCESSFUL`.
- Within the `send_exception` interception point, this attribute will be either `SYSTEM_EXCEPTION` or `USER_EXCEPTION`.
- Within the `send_other` interception point, this attribute will be any of: `SUCCESSFUL`, or `LOCATION_FORWARD`. `SUCCESSFUL` means an asynchronous request returned successfully. `LOCATION_FORWARD`means that a reply came back with `LOCATION_FORWARD` as its status.

```
virtual CORBA::Object_ptr forward_reference() = 0;
```

If the `reply_status()` returns `LOCATION_FORWARD`, then this method returns an object to which the request will be forwarded. It is indeterminate whether a forwarded request will actually occur.

```
virtual CORBA::Any* get_slot(CORBA::ULong _id) = 0;
```

This method returns the data, in the form of a `CORBA::Any`, from the given slot of the

`PortableInterceptor::Current` that is in the scope of the request.

If the given slot has not been set, then a `CORBA::Any` containing a type code with a `TCKind` value of `tk_null` is returned.

If the ID does not define an allocated slot, `InvalidSlot` is raised. See "Current" for an explanation of slots and the `PortableInterceptor::Current`.

Parameter	Description
`_id`	The `SlotId` of the slot which is to be returned.

```
virtual IOP::ServiceContext*
  get_request_service_context(CORBA::ULong _id) = 0;
```

This method returns a copy of the service context with the given ID that is associated with the request.

If the request's service context does not contain an entry for that ID, `BAD_PARAM` with a standard minor code of 26 is raised.

Parameter	Description
`_id`	The `IOP::ServiceContext` of the slot which is to be returned.

```
virtual IOP::ServiceContext*
  get_reply_service_context(CORBA::ULong _id) = 0;
```

This method returns a copy of the service context with the given ID that is associated with the reply.

If the request's service context does not contain an entry for that `ID`, `BAD_PARAM` with a standard minor code of 26 is raised.

Parameter	Description
`_id`	The `IOP::ServiceContext` of the slot which is to be returned.

ServerRequestInfo

```
class PortableInterceptor::ServerRequestInfo : public
  virtual RequestInfo
```

This class is derived from `RequestInfo`. It is passed to server side interception points.

Some methods on `ServerRequestInfo` are not valid at all interception points. The table below shows the validity of each attribute or method. If it is not valid, attempting to access it will result in a `BAD_INV_ORDER` being raised with a standard minor code of 14.

Table 9 `ServerRequestInfo`

	receive_ request_ service_ contexts	receive_ request	send_ reply	send_ exception	send_ other
request_id	yes	yes	yes	yes	yes
operation	yes	yes	yes	yes	yes
arguments	no	yes[1]	yes	no[2]	no[2]
exception	no	yes	yes	yes	yes
contexts	no	yes	yes	yes	yes
operation_context	no	yes	yes	no	no
result	no	no	yes	no	no
response_expected	yes	yes	yes	yes	yes
sync_scope	yes	yes	yes	yes	yes
reply_status	no	no	yes	yes	yes
forward_reference	no	no	no	no	yes[2]
get_slot	yes	yes	yes	yes	yes
get_request_service_ context	yes	yes	yes	yes	yes
get_reply_service_ context	no	no	yes	yes	yes
sending_exception	no	no	no	yes	no
object_id	no	yes	yes	yes[3]	yes[3]
adapter_id	no	yes	yes	yes[3]	yes[3]
server_id	no	yes	yes	yes	yes
orb_id	no	yes	yes	yes	yes
adapter_name	no	yes	yes	yes	yes
target_most_derived_ interface	no	yes	no[4]	no[4]	no[4]
get_server_policy	yes	yes	yes	yes	yes
set_slot	yes	yes	yes	yes	yes
target_is_a	no	yes	no[4]	no[4]	no[4]
add_reply_service_ context	yes	yes	yes	yes	yes

[1] When `ServerRequestInfo` is passed to `receive_request()`, there is an entry in the list for every argument, whether in, inout, or out. But only the in and inout arguments will be available.

[2] If the `reply_status()` does not return `LOCATION_FORWARD`, accessing this attribute will raise `BAD_INV_ORDER` with a standard minor code of 14.

[3] If the servant locator caused a location forward, or raised an exception, this attribute / method may not be available in this interception point.

NO_RESOURCES with a standard minor code of 1 will be raised if it is not available.

4 The method is not available in this interception point because the necessary information requires access to the target object's servant, which may no longer be available to the ORB. For example, if the object's adapter is a POA that uses a ServantLocator, then the ORB invokes the interception point after it calls ServantLocator::postinvoke().

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

ServerRequestInfo methods

```
virtual CORBA::Any* sending_exception() = 0;
```

This method returns data, in the form CORBA::Any, that contains the exception to be returned to the client.

If the exception is a user exception which cannot be inserted into a CORBA::Any (e.g., it is unknown or the bindings don't provide the TypeCode), then this attribute will be an CORBA::Any containing the system exception UNKNOWN with a standard minor code of 1.

```
virtual char* server_id() = 0;
```

This method returns the value that was passed into the ORB::init call using the - ORBServerId argument when the ORB was created.

```
virtual char* orb_id() = 0;
```

The method returns the value that was passed into the ORB::init() call.

In Java, this is accomplished using the -ORBid argument in the ORB.init call that created the ORB containing the object adapter that created this template. What happens if the same ORBid is used on multiple ORB::init() calls in the same server is currently undefined.

```
virtual CORBA::StringSequence* adapter_name() = 0;
```

The method returns the name for the object adapter, in the form of CORBA::StringSequence, that services requests for the invoked object. In the case of the POA, the adapter_name is the sequence of names from the root POA to the POA that services the request. The root POA is not named in this sequence.

```
virtual CORBA::OctetSequence* object_id() = 0;
```

This method returns the opaque object_id, in the form of CORBA::OctetSequence, that describes the target of the operation invocation.

```
virtual CORBA::OctetSequence* adapter_id() = 0;
```

This method returns opaque identifier for the object adapter, in the form of CORBA::OctetSequence.

```
virtual char* target_most_derived_interface() = 0;
```

This method returns the `RepositoryID` for the most derived interface of the servant.

```
virtual CORBA::Policy_ptr
  get_server_policy(CORBA::ULong _type) = 0;
```

This method returns the policy in effect for this operation for the given policy type. The returned `CORBA::Policy` object will only be a policy whose type was registered via `register_policy_factory()`.

If a policy for the given type was not registered via `register_policy_factory`, this method will raise `INV_POLICY` with a standard minor code of 3.

Parameter	Description
`_type`	The `CORBA::PolicyType` which specifies the policy to be returned.

```
virtual void set_slot(CORBA::ULong _id, const
  CORBA::Any& _data) = 0;
```

This method allows an Interceptor to set a slot in the `PortableInterceptor::Current` that is in the scope of the request. If data already exists in that slot, it will be overwritten.

If the ID does not define an allocated slot, `InvalidSlot` is raised.

See "Current" for an explanation of slots and `PortableInterceptor::Current`.

Parameter	Description
`_id`	The `SlotId` of the slot.
`_data`	The data, in the form of a `CORBA::Any`, to store in that slot.

```
virtual CORBA::Boolean target_is_a(const char* _id) =
  0;
```

This method returns true if the servant is the given `RepositoryId`, `false` if it is not.

Parameter	Description
`_id`	The caller wants to know if the servant is this `CORBA::RepositoryId`.

```
virtual void add_reply_service_context(const
  IOP::ServiceContext&_service_context,
  CORBA::Boolean _replace) = 0;
```

This method allows Interceptors to add service contexts to the request.

There is no declaration of the order of the service contexts. They may or may not appear in the order that they are added.

Parameter	Description
`_service_context`	The `IOP::ServiceContext` to add to the reply.
`_replace`	Indicates the behavior of this method when a service context already exists with the given ID. If false, then BAD_INV_ORDER with a standard minor code of 15 is raised. If true, then the existing service context is replaced by the new one.

ServerRequestInterceptor

class **PortableInterceptor::ServerRequestInterceptor** : public virtual Interceptor

This `ServerRequestInterceptor` class is used to derive user-defined server side interceptor. A `ServerRequestInterceptor` instance is registered with the ORB (see "ORBInitializer").

Include file

Include the **PortableInterceptor_c.hh** file when you use this class.

ServerRequestInterceptor methods

```
virtual void receive_request_service_contexts
  (ServerRequestInfo_ptr _ri) = 0;
```

At this `receive_request_service_contexts()` interception point, Interceptors must get their service context information from the incoming request and transfer it to `PortableInterceptor::Current`'s slots.

This interception point is called before the servant manager is called. Operation parameters are not yet available at this point. This interception point may or may not execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' `receive_request_service_contexts()` interception points are called. Those Interceptors on the Flow Stack are popped and their `send_exception()` interception points are called.

This interception point may also raise a `ForwardRequest` exception (see "ForwardRequest"). If an Interceptor raises this exception, no other Interceptors' `receive_request_service_contexts()` methods are called. Those Interceptors on the Flow Stack are popped and their `send_other` interception points are called

Parameter	Description
`_ri`	This is the `ServerRequestInfo` instance to be used by Interceptor.

```
virtual void receive_request(ServerRequestInfo_ptr _ri)
  = 0;
```

This `receive_request()` interception point allows an Interceptor to query request information after all the information, including method parameters, are available. This interception point will execute in the same thread as the target invocation.

In the DSI model, since the parameters are first available when the user code calls `arguments()`, `receive_request()` is called from within `arguments()`. It is possible that `arguments()` is not called in the DSI model. The target may call `set_exception()` before calling `arguments()`. The ORB will guarantee that `receive_request()` is called once, either through `arguments()` or through `set_exception()`. If it is called through `set_exception()`, requesting the `arguments()` will result in `NO_RESOURCES` being raised with a standard minor code of 1.

This interception point may raise a system exception. If it does, no other Interceptors' `receive_request()` methods are called. Those Interceptors on the Flow Stack are popped and their `send_exception` interception points are called.

This interception point may also raise a `ForwardRequest` exception (see "ForwardRequest"). If an Interceptor raises this exception, no other Interceptors' `receive_request()` methods are called. Those Interceptors on the Flow Stack are popped and their `send_other()` interception points are called.

Parameter	Description
`_ri`	This is the `ServerRequestInfo` instance to be used by Interceptor.

```
virtual void send_reply(ServerRequestInfo_ptr _ri) = 0;
```

This `send_reply()` interception point allows an Interceptor to query reply information and modify the reply service context after the target operation has been invoked and before the reply is returned to the client. This interception point will execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' `send_reply()` interception points are called. The remaining Interceptors in the Flow Stack will have their `send_exception()` interception point called.

Parameter	Description
`_ri`	This is the `ServerRequestInfo` instance to be used by Interceptor.

```
virtual void send_exception(ServerRequestInfo_ptr _ri)
  = 0;
```

This `send_exception()` interception point is called when an exception occurs. It allows an Interceptor to query the exception information and modify the reply service context before the exception is raised to the client. This interception point will execute in the same thread as the target invocation.

This interception point may raise a system exception. This has the effect of changing the exception which successive Interceptors popped from the Flow Stack receive on their calls to `send_exception`. The exception raised to the client will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may also raise a `ForwardRequest` exception (see "ForwardRequest". If an Interceptor raises this exception, no other Interceptors' `send_exception()` interception points are called. The remaining Interceptors in the Flow Stack will have their `send_other` interception points called.

```
virtual void send_other(ServerRequestInfo_ptr _ri) = 0;
```

This `send_other()` interception point allows an Interceptor to query the information available when a request results in something other than a normal reply or an exception. For example, a request could result in a retry (e.g., a GIOP Reply with a `LOCATION_FORWARD` status was received). This interception point will execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' `send_other()` methods are called. The remaining Interceptors in the Flow Stack will have their `send_exception` interception points called.

This interception point may also raise a `ForwardRequest` exception.

4.x Interceptor and Object Wrapper Interfaces and Classes

This chapter describes the interfaces and classes that you can use with 4.x interceptors and object wrappers.

Note

To begin with this section, read the chapters on 4.x interceptors and object wrappers in the VisiBroker-RT for C++ *Developer's Guide* before using these interfaces.

Introduction

4.x Interceptors are interceptors that are defined and implemented in VisiBroker version 4.x and later. Similar to Portable Interceptor, 4.x interceptor offers VisiBroker-RT for C++ ORB services a mechanism to intercept normal flow of execution of the ORB. The table below lists the three forms of 4.x interceptor.

Table 10 Types of Interceptor

Interceptor Type	Description
Client Interceptor	System level interceptors which can be used to hook ORB services such as transactions and security into the client ORB processing..
Server Interceptor	System level interceptors which can be used to hook ORB services such astransactions and security into the server ORB processing.
Object Wrappers	User level interceptors which provide a simple mechanism for users to intercept calls to stubs and skeletons. These allow for simple tracing and data caching among other things.

For more information about using the 4.x interceptors and object wrapper interfaces and classes for C++, refer to the 4.x interceptors and object wrapper interfaces and classes for C++ section in the VisiBroker *Programmer's Reference*.

For more information about how to use the object wrapper, refer to using the Object Wrappers section in the VisiBroker *Developer's Guide*.

InterceptorManagers

Interceptors are installed and managed via interceptor managers. The InterceptorManager interface is the generic interceptor manager from which all interceptor-specific managers inherit. An InterceptorManager type is associated with each interceptor type. An InterceptorManager holds a list or chain of a particular kind of interceptors, all of which have the same scope and need to start at the same time. Therefore, global interceptors, such as POALifeCycle and Bind have global InterceptorManagers while scoped interceptors, per-POA and per-object, have an InterceptorManager for each scope. Each scope, either global, POAs, or objects, may hold multiple types

of interceptors. You get the right kind of manager for a particular interceptor from an InterceptorManagerControl.

Global interceptors may be handed additional interceptor managers to install localized interceptors, for example, per-POA interceptors use the POAInterceptorManager.

To obtain an instance of the global interceptor manager, InterceptorManager, call `ORB.resolve_initial_references` and pass the `StringInterceptorManager` as an argument.

This value is only available when the ORB is in administrative mode, that is, during ORB initialization. It can only be used to install global interceptors such as, POALifeCycle interceptors or Bind interceptors.

The POA interceptor manager is a per-POA manager and is only available to POALifeCycleInterceptors during their create call. POALifeCycleInterceptors may set up all other server side interceptors during the call to create. The Bind Interceptor Manager is avper-object manager and is only available to Bind interceptors during their `bind_succeeded()` call. Bind interceptors may set up ClientRequest interceptors during the `bind_succeeded` call.

IOR templates

In addition to the interceptor, the Interoperable Object Reference (IOR) template may be modified directly on the POAIntercptorManager interface during the call to `POALifeCycleInterceptor::create( )`. The IOR template is a full IOR value with the `type_id` not set, and all `GIOP::ProfileBodyValues` have incomplete object keys. The POA sets the `type_id` and fills in the object keys of the template before calling the IORCreationInterceptors.

InterceptorManager

`class` **`Interceptor::InterceptorManager`**

This is the base class from which all interceptor managers are derived. Interceptor managers are interfaces which are used to manage the installation and removal of interceptors from the system.

InterceptorManagerControl

```
class Interceptor::InterceptorManagerControl public
  CORBA::PseudoObject
```

This is the class that is responsible for controlling a set of related interceptor managers. It holds all available managers identified by a string that corresponds to the type of interceptors to be managed. There is one InterceptorManagerControl per scope.

Include file

Include the **interceptor_c.hh** file when you use this class.

InterceptorManagerInterceptor method

```
InterceptorManager_ptr get_manager(const char name);
```

This method returns an instance of the InterceptorManager which returns a string identifying the manager.

Parameter	Description
name	The name of the interceptor

BindInterceptor

```
class Interceptor::BindInterceptor public
  VISPseudoInterface
```

You can use this class to derive your own interceptor for handling bind and rebind events for a client or server application. The Bind Interceptors are global interceptors invoked on the client side before and after binds.

If an exception is thrown during a bind, the remaining interceptors in the chain are not called and the chain is truncated to only those interceptors already called. Exceptions thrown during `bind_succeeded` or `bind_failed` are ignored.

Include file

You should include the **interceptor_c.hh** file when you use this class.

BindInterceptor methods

```
virtual IOP::IORValue_ptr bind(IOP::IORValue_ptr ior,
  CORBA::Object_ptr obj, CORBA::Boolean rebind,
  VISClosure& closure);
```

This method is called during all ORB bind operations.

Parameter	Description
`ior`	The Interoperable Object Reference (IOR) for the server object to which the client is binding.
`obj`	The client object which is being bound to the server. The object will not be properly initialized at this time, so do not attempt an operation on it. However, it may be stored in a data structure and used after the bind has completed.
`rebind`	An attempt to rebind to the server. After a bind() has failed, depending on the current quality of service, a rebind may be attempted.
`closure`	A new closure object for the bind operation. The closure will be used in corresponding calls to either `bind_failure` or `bind_succeeded`.
`return`	Returns a new IOR, if the bind operation is to be continued using this new IOR. Otherwise, it returns a null value and the bind will proceed using the original IOR. Returning the same IOR as the parameter passed in is incorrect and generates an exception at bind time.

```
virtual IOP::IORValue_ptr bind_failed(IOP::IORValue_ptr
  ior, CORBA::Object_ptr obj, VISClosure& closure);
```

This method is called if a bind operation failed.

Parameter	Description
ior	The IOR of the server object on which the bind operation failed.
obj	The client object which is being bound to the server.
closure	The closure object previously given in the bind call.
return	Returns a new IOR if a rebind is to be attempted against this IOR. Otherwise, it returns null, and a rebind is not attempted.

```
virtual void bind_succeeded(IOP::IORValue_ptr ior,
  CORBA::Object_ptr obj, CORBA::Long profileIndex,
  InterceptorManagerControl_ptr interceptorControl,
  VISClosure& closure);
```

This method is called if a bind operation succeeded.

Parameter	Description
ior	The IOR of the server object on which the bind operation succeeded.
obj	The client object which is being bound to the server.
interceptorControl	This Manager provides a list of the types of Managers.
closure	The closure object previously given in the bind call.

BindInterceptorManager

```
class Interceptor::BindInterceptorManager public
  InterceptorManager, public VISPseudoInterface
```

This is the class that manages all the global bind interceptors. It only has one public method, which allows you to register interceptors.

The `BindInterceptorManager` must always be used at `ORB_init( )`. It has no effect after the orb is initialized. Therefore, it only needs to be used in the context of a loader class that inherits from `VISinit`.

To obtain a `BindInterceptorManager` from the `InterceptorManagerControl`, use `InterceptorManagerControl::get_manager( )` with the identification string `Bind`.

Include file

You should include the **interceptor_c.hh** file when you use this class.

BindInterceptorManager method

```
void add(BindInterceptor _ptr interceptor);
```

This method is used to add a BindInterceptor to the list of interceptors to be started at bind time.

ClientRequestInterceptor

```
class Interceptor::ClientRequestInterceptor public
  VISPseudoInterface
```

You use this class to derive your own client side interceptor. The Client Request interceptors may be installed during the `bind_succeeded` call of a bind interceptor and remain active for the duration of the connection. The methods defined in your derived class will be invoked by the ORB during the preparation or sending of an operation request, during the receipt of a reply message, or if an exception is raised.

Include file

Include the i**nterceptor_c.hh** file when you use this class.

ClientRequestInterceptor methods

```
virtual void preinvoke_premarshal(CORBA::Object_ptr
  target, const char* operation,
  IOP::ServiceContextList& service_contexts,
  VisClosure& closure);
```

This method is invoked by the ORB on every request, before the request has been marshalled. An exception thrown from this interceptor results in the request being completed immediately. In this case, the chain is shortened to only those interceptors that have already fired, the request will not be sent, and exception_occurred() is called on all interceptors still in the chain.

Parameter	Description
`target`	The client object which is being bound to the server.
`operation`	The name of the operation being invoked.
`service_context`	The services assigned by the ORB. These services are identified by a tag registered with the OMG.
`closure`	The closure object previously given in the bind call.

```
virtual void preinvoke_postmarshal(CORBA::Object_ptr
  target, CORBA_MarshalOutBuffer& payload, VISClosure&
  closure);
```

This method is invoked after every request has been marshaled, but before it was sent.

If an exception is thrown in this method:

- the rest of the chain is not invoked,
- the request is not sent to the server, and
- `exception_occurred()` is called on the whole interceptor chain.

Parameter	Description
`target`	The client object which is being bound to the server.
`payload`	Marshalled buffer.
`closure`	The closure object previously given in the bind call.

```
virtual void postinvoke(CORBA::Object_ptr target, const
  IOP::ServiceContextList& service_contexts,
  CORBA_MarshalInBuffer& payload,
  CORBA::Environment_ptr env, VISClosure& closure);
```

This method is invoked after a request completes correctly or by throwing an exception. It is called after the `ServantLocator` has been invoked. Should an interceptor in the chain throw an exception, that interceptor also calls `exception_occurred()` and all remaining interceptors in the chain call `exception()` instead of calling `postinvoke()`.

The `CORBA::Environment` parameter is changed to reflect this exception, even when a two-way call had already written an exception in that argument.

Parameter	Description
`target`	The client object which is being bound to the server.
`service_context`	The services assigned by the ORB. These services are identified by a tag registered with the OMG.
`payload`	Marshalled buffer.
`env`	Contains information on the exception that was raised.
`closure`	The closure object previously given in the bind call.

virtual void **exception_occurred**(CORBA::Object_ptr **target**, CORBA::Environment_ptr **env**, VISClosure& **closure**);

This method is invoked by the ORB when an exception is thrown before the invocation. All exceptions thrown after the invocation are gathered in the environment parameter of the `postinvoke` method.

Parameter	Description
`target`	The client object which is being bound to the server.
`env`	Contains information on the exception that was raised.
`closure`	The closure object previously given in the bind call.

ClientRequestInterceptorManager

```
class Interceptor::ClientRequestInterceptorManager :
  public InterceptorManager,public VISPseudoInterface
```

This is the class that holds the chain of `ClientRequestInterceptors` for the current object.

A `ClientRequestInterceptorManager` should be used inside of the `BindInterceptor::bind_succeeded( )` method within the scope set by the `InterceptorManagerControl` passed as an argument to `bind_succeeded( )`.

Include file

Include the **interceptor_c.hh** when you use this class.

ClientRequestInterceptorManager methods

```
virtual void add(ClientRequestInterceptor_ptr
  interceptor);
```

This method may be invoked to add a `ClientRequestInterceptor` to the local chain.

```
virtual void remove(ClientRequestInterceptor_ptr
  interceptor);
```

This method removes a ClientRequestInterceptorManager.

POALifeCycleInterceptor

```
class InterceptorManager::POALifeCycleInterceptor
  public VISPseudoInterface
```

The `POALifeCycleInterceptor` is a global interceptor which is invoked every time a POA is created or destroyed. All other server side interceptors may be installed either as global interceptors or for specific POAs. You install the `POALifeCycleInterceptor` through the `POALifeCycleInterceptorManager` interface. See "POALifeCycleInterceptorManager". The `POALifeCycleInterceptor` is called during POA creation and destruction.

Include file

Include the **PortableServerExt_c.hh** file when you use this class.

POALifeCycleInterceptor methods

```
virtual void create(PortableServer::POA_ptr poa,
  CORBA::PolicyList& policies, IOP::IORValue*&
  iorTemplate,
  interceptor::InterceptorManagerControl_ptr poaAdmin);
```

This method is invoked when a new POA is created either explicitly through a call to `create_POA` or via `AdapterActivator`. With `AdapterActivator`, the interceptor is called only after the `unknown_adapter` method successfully returns from the `AdapterActivator`. The create method is passed as a reference to the recently created POA and as a reference to that POA instance's `POAInterceptorManager`.

Parameter	Description
`poa`	The ID associated with the current POA being created.
`policies`	The policies for the POA being created.
`iorTemplate`	The IOR template is a full IOR value with the `type_id` not set, and all `GIOP::ProfileBodyValues` will have incomplete object keys.
`poaAdmin`	The control for the POA being created. See "InterceptorManagerControl" on page 12-3 for more information.

```
virtual void destroy(PortalServer::POA_ptr poa);
```

This method is called before a POA is destroyed and all of its objects have been etherealized. It guarantees that destroy will be called on all interceptors before create will be called again for a POA with the same name. If the destroy operation throws a system exception, the exception is ignored, and the remaining interceptors are called.

Parameter	Description
`poa`	The Portable Object Adapter (POA) being destroyed.

POALifeCycleInterceptorManager

```
class InterceptorExt::POALifeCycleInterceptorManager
  public interceptor::InterceptorManager, public
  VISPseudoInterface
```

This class manages all `POALifeCycle` global interceptors. There is a single instance of the `POALifeCycleInterceptorManager` defined in an ORB.

The scope of this interface is global, per-ORB. This class is only active during `ORB_init()` time.

Include file

Include the **PortalServerExt_c.hh** file when you use this class.

POALifeCycleInterceptorManager method

```
virtual void add(POALifeCycleInterceptor_ptr
  interceptor);
```

This method may be invoked to add a POALifeCycleInterceptor to the global chain of POALifeCycle interceptors.

Parameter	Description
`interceptor`	The interceptor to be added.

ActiveObjectLifeCycleInterceptor

```
class
  PortableServerExt::ActiveObjectLifeCycleInterceptor
  public VISPseudoInterface
```

The `ActiveObjectLifeCycleInterceptor` interceptor is called when objects are added and removed from the active object map. It is only used when POA has RETAIN policy. This lass is a POA-scoped interceptor which may be installed by a `POALifeCycleInterceptor` when the POA is created.

Include file

Include the **PortableServerExt_c.hh** file when you use this class.

ActiveObjectLifeCycleInterceptor methods

```
virtual void create(const PortableServer::ObjectId&
  oid, PortableServer::ServantBase* ,
  PortableServer::POA_ptr adapter);
```

This method is invoked after an object has been added to the Active Object Map, either through explicit or implicit activation, using either direct APIs or a `ServantActivator`. The object reference and the POA of the new active object are passed as parameters.

Parameter	Description
`oid`	Object ID for the object currently activated.

Parameter	Description
servant	Associated servant.
activator	The Portable Object Adapter (POA) being created or destroyed.

```
virtual void destroy(const PortableServer::ObjectId&
  oid, PortableServer::ServantBase* servant,
  PortableServer::POA_ptr adapter);
```

This method is called after an object has been deactivated and etherealized. The object reference and the POA of the object are passed as parameters.

Parameter	Description
oid	Object ID for the object currently activated.
servant	Associated servant.
activator	The Portable Object Adapter (POA) being created or destroyed.

ActiveObjectLifeCycleInterceptorManager

```
class
  PortableServerExt::ActiveObjectLifeCycleInterceptorMa
  nager public interceptor::InterceptorManager, public
  VISPseudoInterface
```

This is the class that manages all `ActiveObjectLifeCycleInterceptors` registered in its scope. Each POA has one single `ActiveObjectLifeCycleInterceptorManager`.

Include file

Include the **PortableServer_c.hh** file when you use this class.

ActiveObjectLifeCycleInterceptorManager method

```
virtual void add(ActiveObjectLifeCycleInterceptor
  interceptor_ptr interceptor);
```

This method may be invoked to add an `ActiveObjectLifeCycleInterceptor` to the chain.

ServerRequestInterceptor

```
class Interceptor::ServerRequestInterceptor public
  VISPseudoInterface
```

The `ServerRequestInterceptor` class is a POA-scoped interceptor which may be installed by a `POALifeCycleInterceptor` at POA creation time. This class may be used to perform access control, to examine and insert service contexts, and to change the reply status of a request.

Include file

Include the i**nterceptor_c.hh** file when you use this class.

ServerRequestInterceptor methods

```
virtual void preinvoke(CORBA::Object_ptr _target, const
  char* operation, const IOP::ServiceContextList&
  service_contexts, CORBA::MarshalInBuffer& payload,
  VISClosure& closure) raises
  (ForwardRequestException);
```

This method is invoked by the ORB on every request, before the request is demarshaled. An exception thrown from this interceptor results in the request being completed immediately. This method is called before any `ServantLocators` are invoked. The result may be that the servant may not be available while this method is running.

Parameter	Description
`target`	The client object that is being bound to the server.
`operation`	Identifies the name of the operation being invoked.
`service_contexts`	Identifies the services assigned by the Orb. These services are registered with the OMG.
`payload`	Marshalled buffer.
`closure`	May contain data saved by one interceptor method that can be retrieved later by another interceptor method.

```
virtual void postinvoke_premarshal(CORBA::Object_ptr
  target, IOP::ServiceContextList&
  ServiceContextList,CORBA::Environment_ptr env,
  VISClosure& closure);
```

This method is invoked after an upcall to the servant but before marshalling the reply. An exception here is handled by interrupting the chain: the request is not sent to the server and `exception_occurred()` is called on all interceptors in the chain.

Parameter	Description
`target`	The client object that is being bound to the server.
`ServiceContextList`	Identifies the services assigned by the Orb. These services are registered with the OMG.
`env`	Contains information on the exception that was raised.
`closure`	May contain data saved by one interceptor method that can be retrieved later by another interceptor method.

virtual void **postinvoke_postmarshal**(CORBA::Object_ptr **_target**, CORBA::MarshalOutBuffer& **_payload**, VISClosure& **_closure**);

This method is invoked after marshalling the reply but before sending the reply to the client. Exceptions thrown here are ignored. The entire chain is guaranteed to be called.

Parameter	Description
`target`	The object to which that application was attempting to bind.
`payload`	Marshalled buffer.
`closure`	May contain data saved by one interceptor method that can be retrieved later by another interceptor method.

```
virtual void exception_occurred(CORBA::Object_ptr
  _target, CORBA::Environment_ptr _env, VISClosure&
  _closure);
```

This method is invoked by the ORB when an `exceptionoccurred` interceptor is called on all remaining interceptors in the chain after an exception occurred in one of the `prepare_reply` interceptors. An exception thrown during this call replaces the existing exception in the environment.

Parameter	Description
`target`	The client object which is being bound to the server.
`payload`	Contains information on the exception that was raised.
`closure`	May contain data saved by one interceptor method that can be retrieved later by another interceptor method.

ServerRequestInterceptorManager

```
class Interceptor::ServerRequestInterceptorManager
  public InterceptorManager, public VISPseudoInterface
```

This is the class that manages all ServerRequestInterceptors registered in its scope. Each POA has one single ServerRequestInterceptorManager.

Include file

Include the **interceptor_c.hh** file when you use this class.

ServerRequestInterceptorManager method

```
virtual void add(ServerRequestInterceptor_ptr
  interceptor);
```

Invoke this method to add a `ServerRequestInterceptor` to the chain.

IORCreationInterceptor

```
class PortableServerExt::IORCreationInterceptor public
  VISPseudoInterface
```

The IORCreationInterceptor is a per-POA interceptor which may be installed by a POALifeCycleInterceptor at POA creation time. The interceptor may be used to modify IORs by adding additional profiles or components. This class is typically used to support services such as transactions or firewall.

This kind of interceptor is used to automatically change the IOR templates on certain classes of POAs whose names and identities may not be known at development time. This may be the case with services such as Transaction and Firewall.

Note

To change all the IORs created by a POA, simply modify the IORTemplate for that POA. The change will apply only to newly created IORs and not to any existing ones.

Making radical changes to the IOR is not recommended.

Include file

Include the **PortableServerExt_c.hh** file when you use this class.

IORInterceptor method

```
virtual void create(PortableServer::POA poa,
  IOP::IORValue*& ior);
```

The method is called whenever the POA needs to create an object reference. It takes the POA and the IORValue for the reference as arguments. The interceptor may modify the IORValue by adding additional profiles or components, or changing the existing profiles or components.

Parameter	Description
`poa`	The ID associated with the current poa being created.
`ior`	The IOR for the server object with which the client is binding.

IORCreationInterceptorManager

```
class PortableServerExt::IORCreationInterceptorManager
  public interceptor::InterceptorManager, public
  VISPseudoInterface
```

This is the class that is used to manage (add) IOR interceptors to the local chain. Each POA has one single `IORInterceptorManager`.

Include file

Include the **PortableServerExt_c.hh** file when you use this class.

IORCreationInterceptorManager method

```
virtual void add(IORCreationInterceptor_ptr
  _interceptor);
```

This method may be invoked to add an `IORInterceptor` to the local chain.

VISClosure

```
struct VISClosure
```

This structure is used to store data so that it can be shared between different invocations of interceptor methods. The data that is stored is un-typed and can represent state information related to an operation request or a bind or locate request. It is used in conjunction with theVISClosureData class.

Include file

Include the **vclosure.h** file when you use this class.

VISClosure members

`CORBA::ULong` **`id`**

You can use this data member to uniquely identify this object if you are using more than one VISClosure object.

`void`

This data member points to the un-typed data that may be stored or accessed by an interceptor method.

`VISClosureData` **`*managedData`**

This data member points to the VISClosureData class that represents the actual data. You may cast your managed data to this type.

VISClosureData

`class` **`VISClosureData`**

This class represents managed data that can be shared between different invocations of interceptor methods.

VISClosureData methods

`virtual` **`~VisClosureData`**`();`

This is the default destructor.

`virtual void` **`_release();`**

Releases this object and decrements the reference count. When the reference count reaches 0, the object is deleted.

ChainUntypedObjectWrapperFactory

```
class
  VISObjectWrapper::ChainUntypedObjectWrapperFactory :
  public UntypedObjectWrapperFactory
```

This interface is used by a client or server application to add or remove an `UntypedObjectWrapperFactory` object. An `UntypedObjectWrapperFactory` is used to create an `UntypedObjectWrapper` for each object a client application binds to or for each object implementation created by a server application.

Refer to the Using Object Wrappers section in the *VisiBroker-RT for C++ Developer's Guide* for more information about how to use the object wrappers.

Include file

Include the **vobjwrap.h** file when you use this class.

ChainUntypedObjectWrapperFactory methods

```
void add(UntypedObjectWrapperFactory_ptr factory,
  Location loc);
```

This method adds the specified un-typed object wrapper factory for a client application, server application, or collocated application.

If your application is acting as both a client application and a server application, that is, a collocated application, you can install an un-typed object wrapper factory. If you do so, the wrapper's methods are invoked for both invocations on bound objects and operation requests received by object implementations. In other words, they are invoked on both the

client and server portions of the application.

Note

On the client side, un-typed object wrapper factories must be defined before any objects are bound. On the server side, un-typed object wrapper factories must be defined before an invocation for an object implementation is received.

Parameter	Description
`factory`	A pointer to the factory to be registered.

`loc`	The location of the factory being added, which should be one of the following values:
	`VISObjectWrapper::Client`
	`VISObjectWrapper::Server`
	`VISObjectWrapper::Both`

```
void remove(UntypedObjectWrapperFactory_ptr factory,
  Location loc);
```

This method removes the specified un-typed object wrapper factory from the specified location.

If your application is acting as both a client and a server, you can remove the object wrapper factories for either the client side objects, server side implementations, or both.

Note

Removing one or more object wrapper factories from a client does not affect objects of that class which are already bound by the client. Only subsequently bound objects will be affected.

Removing object wrapper factories from a server does not affect object implementations that have already serviced requests. Only subsequently created object implementations will be affected.

Parameter	Description
`factory`	A pointer to the factory to be registered.

`loc`	The location of the factory being removed, which should be one of the following values:
	`VISObjectWrapper::Client`
	`VISObjectWrapper::Server`
	`VISObjectWrapper::Both`

```
static CORBA::ULong count(Location loc);
```

This static method returns the number of un-typed object wrapper factories installed for the specified location.

UntypedObjectWrapper

```
class VISObjectWrapper::UntypedObjectWrapper : public
  VISResource
```

You use this class to derive and implement an un-typed object wrapper for a client application, a server application, or co-located application. When you derive an un-typed object wrapper from this class, you define a `pre_method` method that is invoked before a request is issued by a client application or before it is processed by an object implementation on the server side. You also define a `post_method` method that will be invoked after an operation request is processed by an object implementation on the server side or after a reply has been received by a client application.

You must also derive a factory class that will create your un-typed wrapper objects. Derive it from the `UntypedObjectWrapperFactory` class, described in "UntypedObjectWrapperFactory".

Refer to the VisiBroker-RT for C++ *Developer's Guide* for more information about how to use the object wrappers.

Include file

Include the **vobjwrap.h** file when you use this class.

UntypedObjectWrapper methods

```
virtual void pre_method(const char* operation,
  CORBA::Object_ptr target, VISClosure& closure);
```

This method is invoked before an operation request is sent on the client side or before it is processed by an object implementation on the server side.

Parameter	Description
`operation`	The name of the operation being requested.
`target`	The object that is the target of the request
`closure`	The `Closure` object can be used to pass data between object wrapper methods.

```
virtual void post_method(const char* operation,
  CORBA::Object_ptr target, CORBA::Environment& env,
  VISClosure& closure);
```

This method is invoked after an operation request has been processed by the object implementation on the server side or before the reply message is processed by the stub on the client side.

Parameter	Description
`operation`	The name of the operation being requested.
`target`	The object that is the target of the request

Parameter	Description
env	An Environment object that is used to reflect exceptions that might have occurred in the processing of the operation request.
closure	The Closure object can be used to pass data between object wrapper methods.

UntypedObjectWrapperFactory

class **VISObjectWrapper::UntypedObjectWrapperFactory**

You use this interface to derive your own un-typed object wrapper factories. Your factory will be used to create an instance of your un-typed object wrapper for an application whenever a new object is bound to or an object implementation services a request.

Include file

Include the **vobjwrap.h** file when you use this class.

UntypedObjectWrapperFactory constructor

```
UntypedObjectWrapperFactory(Location loc,
  CORBA::Boolean doAdd=1);
```

Creates an un-typed object wrapper factory for the specified location and by default registers it with the ChainUntypedObjectWrapperFactory. If your application is acting as both a client application and a server application, you can install an un-typed object wrapper factory so the wrapper's methods will be invoked for both invocations on bound objects and operation requests received by object implementations.

If you don't want to use the default parameter, you can specify that the doAdd not be performed. However, to create an untyped object wrapper, you will have to call ChainUntypedObjectWrapper::add.

Parameter	Description
loc	The location of the factory being added, which should be one of the following values: VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both
doAdd	A flag specifying whether or not the factory is to be registered.

UntypedObjectWrapperFactory methods

```
virtual UntypedObjectWrapper_ptr
  create(CORBA::Object_ptr target, Location loc);
```

This method is called to create an instance of your type of UntypedObjectWrapper. Your implementation of this method can examine the type of bound object or object implementation to determine whether or not it wants to create an object wrapper for that object. With the loc

parameter, you specify whether the create request is called to wrap a client object or a server implementation.

Parameter	Description
`target`	The object being bound by a client application for which the un-typed object wrapper is being created. If this method is being invoked on the server side, this represents the object implementation that is being created.
`loc`	The location of the factory being added.

Real-Time CORBA Interfaces and Classes

This chapter describes the Real-Time CORBA interfaces and classes supported by VisiBroker-RT for C++.

Note

Read the chapter on Real-Time CORBA in the VisiBroker-RT for C++ *Developer's Guide* before using these interfaces.

Introduction

Real-Time CORBA provides a set of APIs that support the development of predictable CORBA-based systems, through the control of the number and priority of threads involved in the execution of CORBA invocations.

The majority of the Real-Time CORBA API is specified in IDL, and is mapped to C++ according to the rules of the CORBA C++ language mapping. The Real-Time CORBA IDL is scoped within module RTCORBA, and hence the C++ class names are all prefixed 'RTCORBA::'.

The following Real-Time CORBA interfaces and classes are described in the sections that follow:

- RTCORBA::ClientProtocolPolicy
- RTCORBA::Current
- RTCORBA::Mutex
- RTCORBA::NativePriority
- RTCORBA::Priority
- RTCORBA::PriorityMapping
- RTCORBA::PriorityModel
- RTCORBA::PriorityModelPolicy
- RTCORBA::RTORB
- RTCORBA::ServerProtocolPolicy
- RTCORBA::ThreadpoolId
- RTCORBA::ThreadpoolPolicy

Include file

To use any of the Real-Time CORBA features described in this chapter, the application should include the file **`rtcorba.h`**, which is one of the include files supplied with VisiBroker.

RTCORBA::ClientProtocolPolicy

`class` **`RTCORBA::ClientProtocolPolicy`** `: CORBA::Policy`

An instance of this Real-Time Policy type is created by calling the `create_client_protocol_policy` method of RTCORBA::RTORB. The Policy instance may then be used to configure the selection of communication protocols on the client-side of VisiBroker-RT for C++ applications. The order of the Protocols in the `ProtocolList` dictates the order which the client-side ORB will use to attempt to connect to the CORBA Object.

IDL

```
// IDL
  module RTCORBA {
  // Locality Constrained interface
  interface ProtocolProperties {};
  struct Protocol {
    IOP::ProfileId protocol_type;
    ProtocolProperties orb_protocol_properties;
    ProtocolProperties transport_protocol_properties;
  };
  typedef sequence <Protocol> ProtocolList;
  // Client Protocol Policy
  const CORBA::PolicyType CLIENT_PROTOCOL_POLICY_TYPE = 1237;
  // locality constrained interface
  interface ClientProtocolPolicy : CORBA::Policy {
    readonly attribute ProtocolList protocols;
  };
  interface RTORB {
    ...
  ClientProtocolPolicy create_client_protocol_policy (
    in ProtocolList protocols
  );
  };
```

RTCORBA::Current

```
class RTCORBA::Current : public CORBA::Object

  typedef RTCORBA::Current* Current_ptr
  class RTCORBA::Current_var
```

The class `RTCORBA::Current` provides methods that allow a Real-Time CORBA Priority value to be associated with the current thread of execution, and the reading of the Real-Time CORBA Priority value presently associated with the current thread.

When a Real-Time CORBA Priority value is associated with the current thread, that value is immediately used to set the Native Priority of the underlying thread. The Native Priority value to apply to the thread is obtained via the currently installed Priority Mapping.

Where the Client Propagated Priority Model is in use, the Priority associated with a thread will also determine the priority of CORBA invocations made from that thread. For details see the section "Real-Time CORBA Priority Models" in the *Programmers Guide*.

`RTCORBA::Current` is defined in IDL, as a locality constrained interface. (See the above section "Locality Constrained Interfaces" for an explanation of this term.) Hence applications handle `RTCORBA::Current` via CORBA Object References, using the C++ classes `RTCORBA::Current_ptr` and `RTCORBA::Current_var`.

See also the section "RTCORBA::Priority".

RTCORBA::Current Creation and Destruction

`RTCORBA::Current` is a special interface. Applications need not be concerned with which instance of it they are dealing. A reference to `RTCORBA::Current` is obtained through the `resolve_initial_references` method of `RTCORBA::RTORB`, and is released in the normal way when it is no longer required. For details see the section "Real-Time CORBA Current" in the *Programmers Guide*.

IDL

```
//Locality Constrained Object
interface Current {
  attribute Priority the_priority;
};
```

RTCORBA::Current methods

`void` **`the_priority`**`(Priority _val);`

Associate the `RTCORBA::Priority` value `_val` with the current thread of execution.

Parameter	Description
`_val`	The Priority value to associate with the thread.

`Priority` **`the_priority`**`();`

Get the `RTCORBA::Priority` value associated with the current thread of execution.

RTCORBA::Mutex

```
class RTCORBA::Mutex : public CORBA::Object

typedef RTCORBA::Mutex* RTCORBA::Mutex_ptr
  class RTCORBA::Mutex_var

class TimeBase {
  typedef unsigned long long TimeT;
  };
```

The interface `RTCORBA::Mutex` provides applications with a mutex synchronization primitive that is guaranteed to have the same priority inheritance properties as mutexes used internally by VisiBroker to protect ORB resources.

`RTCORBA::Mutex` is defined in IDL, as a locality constrained interface. (See the section "Locality Constrained Interfaces" for an explanation of this term.) Hence applications handle `RTCORBA::Mutex` instances via CORBA Object References, using the C++ classes `RTCORBA::Mutex_ptr` and `RTCORBA::Mutex_var`.

The paragraph above is one of several references to a "Locality Constrained Interfaces" section. I can't find any such section. Any ideas?

See also the section "RTCORBA::RTORB".

Mutex Creation and Destruction

A new RTCORBA::Mutex is obtained using the **create_mutex** operation of the RTCORBA::RTORB interface. The new RTCORBA::Mutex is created in an unlocked state.

When the RTCORBA::Mutex is no longer needed, it is destroyed using the **destroy_mutex** operation of RTCORBA::RTORB. See the section "RTCORBA::RTORB" for details.

Note that if the `RTCORBA::Mutex_var` type is used in place of the `RTCORBA::Mutex_ptr` type, the reference is automatically released when the `_var` instance goes out of scope, but the `RTCORBA::Mutex` instance it refers to is not automatically destroyed. The `RTCORBA::Mutex` instance must still be destroyed with a call to **`destroy_mutex`**.

IDL

```
// Locality Constrained Object
interface Mutex {
  void lock( );
  void  unlock( );
  boolean try_lock ( in TimeBase::TimeT max_wait );
};

// defined in TimeBase.idl
module TimeBase {
  typedef unsigned long long TimeT;
};
```

RTCORBA::Mutex Methods

`void` **`lock`**`();`

Lock the RTCORBA::Mutex. When the RTCORBA::Mutex object is in the unlocked state, the first thread to call the lock() operation will cause the Mutex object to change to the locked state. Subsequent threads that call the lock() operation while the Mutex object is still in the locked state will block until the owner thread unlocks it.

`void` **`unlock`**`();`

Unlock the locked RTCORBA::Mutex.

`CORBA::Boolean` **`try_lock`**`( const TimeBase::TimeT _max_wait );`

Attempt to lock the RTCORBA::Mutex, waiting for a maximum of _max_wait amount of time. Returns TRUE if the lock is successfully taken within the time, or FALSE if it could not be taken before the time expired.

Parameter	Description
`_max_wait`	The maximum amount of time to wait for the lock, in 100 nanosecond ticks. A value of 0 means do not wait for the lock.

RTCORBA::NativePriority

`typedef CORBA::Short` **`RTCORBA::NativePriority`**

The type RTCORBA::NativePriority is used to represent priorities in the priority scheme of the particular Operating System that the Real-Time ORB is running on.

Real-Time CORBA applications only use RTCORBA::NativePriority values in special circumstances:

- When defining a Priority Mapping. See the section "RTCORBA::PriorityMapping".
- When interacting directly with the Operating System, or with some other non-CORBA subsystem, that works in terms of Native Priorities. This should still be done via the installed Priority Mapping. See the section "Using Native Priorities in VisiBroker Application Code" in the *Programmers Guide*.

Normally, within a Real-Time CORBA application, priorities will be expressed in terms of `RTCORBA::Priority` values. See the section "RTCORBA::Priority".

IDL

```
typedef CORBA::Short NativePriority;
```

RTCORBA::Priority

```
typedef CORBA::Short RTCORBA::Priority
  static const Priority  RTCORBA::minPriority;  //  0
  static const Priority RTCORBA::maxPriority; // 32767
```

The type RTCORBA::Priority should be used to represent priority values in a

Real-Time The only time a Real-Time CORBA application should use Native Priority values is when interacting directly with the Operating System or some other non-CORBA subsystem. Even then, this the Programmers GuideRTCORBA::Priority values may be anywhere in the range 0 to 32767. However, it is not expected that this full range of priorities will be used in a Real-Time CORBA system. Instead, the application system designer should decide on a suitable range of priorities for that system, and implement a Priority Mapping that only allows priority values in that range. For many applications the default valid range of 0 to 31 will be acceptable, but there may still be reasons to override the default Priority Mapping.

Again, see the section "RTCORBA::PriorityMapping" for details.

IDL

```
typedef CORBA::Short Priority;
  static const Priority minPriority; // 0
  static const Priority maxPriority; // 32767
```

RTCORBA::PriorityMapping

```
class RTCORBA::PriorityMapping
```

The RTCORBA::PriorityMapping class facilitates the mapping of RTCORBA::Priority values to and from the Native Priority scheme of the Operating System the Real-Time ORB is running on. The ORB calls out to a Priority Mapping object whenever it needs to map a RTCORBA::Priority value to a RTCORBA::NativePriority value or vice versa.

A Real-Time CORBA application should describe its priorities in terms of RTCORBA::Priority values. However, the application may need to make explicit use of the installed Priority Mapping, in order to interact directly with the Operating System or some other non-CORBA subsystem. For

details see the section "Using Native Priorities in VisiBroker Application Code" in the *Programmers Guide.* The range of RTCORBA::Priority values supported by a Priority Mapping should always start from zero. The Real-Time ORB expects RTCORBA::Priority zero to be valid. Also, this convention makes integration of different Real-Time CORBA systems on the same node easier.

PriorityMapping Creation and Destruction

It is not necessary to create instances of a Priority Mapping in the code of a normal Real-Time CORBA application. The available Priority Mapping is automatically used by the ORB, and may be accessed by the application if necessary.

Exactly one Priority Mapping is 'installed' at any one time. A 'default' Priority Mapping is provided, that is installed by default. This Default Priority Mapping may be overridden by installing an application-implemented Priority Mapping object. The installation process is described in the section "Replacing the Default Priority Mapping" in the *Programers Guide*.

IDL

```
// 'native' IDL type
native PriorityMapping;
```

The RTCORBA::PriorityMapping IDL type is defined as a 'native' IDL type. This means that its mapping to different programming languages is defined on a per-language basis. The C++ class representing RTCORBA::PriorityMapping has the following declaration:

```
//C++
class PriorityMapping {
  public:
  virtual CORBA::Boolean to_native(
    RTCORBA::Priority corba_priority,
    RTCORBA::NativePriority &native_priority )=0;
  virtual CORBA::Boolean to_CORBA(
    RTCORBA::NativePriority native_priority,
    RTCORBA::Priority &corba_priority )=0;
  virtual RTCORBA::Priority max_priority() = 0;
  PriorityMapping();
  virtual ~PriorityMapping() {}
  static RTCORBA::PriorityMapping * instance();
};
```

PriorityMapping Methods

```
static RTCORBA::PriorityMapping * instance();
```

This static method can be used by Real-Time CORBA applications to access the currently installed Priority Mapping. For details see the section "Using Native Priorities in VisiBroker Application the Programmers Guide for details.

This method is implemented by VisiBroker.

```
virtual RTCORBA::Priority max_priority() = 0;
```

This method returns the maximum Real-Time CORBA Priority value that is valid using this Priority Mapping. For example, if the installed Priority

Mapping maps Real-Time CORBA Priorities in the range 0 to 31, the value 31 will be returned every time this method is called.

This method must be implemented when implementing a new Priority Mapping.

```
virtual CORBA::Boolean to_CORBA (
  RTCORBA::NativePriority native_priority,
  RTCORBA::Priority &corba_priority ) = 0;
```

This method maps a given Native Priority value, `native_priority`, to a Real-Time CORBA Priority value. If the Native Priority value is in the range supported by this Priority Mapping, the resultant Real-Time CORBA Priority value is stored in `corba_priority`, and TRUE is returned. Otherwise `corba_priority` is not changed, and FALSE is returned.

This method must be implemented when implementing a new Priority Mapping.

Parameter	Description
`native_priority`	The Native Priority value to be mapped to a Real-Time CORBA Priority.
`corba_priority`	The variable to assign the mapped Real-Time CORBA Priority value to.

```
virtual CORBA::Boolean to_native ( RTCORBA::Priority
  corba_priority, RTCORBA::NativePriority
  &native_priority ) = 0;
```

This method maps a given Real-Time CORBA Priority value, corba_priority, to a Native Priority value. If the Real-Time CORBA Priority value is in the range supported by this Priority Map-ping, the resultant Native Priority value is stored in native_priority, and TRUE is returned. Otherwise native_prioirty is not changed, and FALSE is returned.

This method must be implemented when implementing a new Priority Mapping.

Parameter	Description
`corba_priority`	The Real-Time CORBA Priority value to be mapped to a Native Priority.
`native_priority`	The variable to assign the mapped Native Priority value to.

RTCORBA::PriorityModel

```
enum RTCORBA::PriorityModel {
  CLIENT_PROPAGATED,
  SERVER_DECLARED
  };
```

This enumeration specifies the two Real-Time CORBA Priority Models : Client Priority Propagation and Server Declared Priority.

These enumeration values are used as values for a parameter to the create_priority_model_policy methods of RTCORBA::RTORB. See the section "RTCORBA::PriorityModelPolicy" for details.

RTCORBA::PriorityModelPolicy

```
class RTCORBA::PriorityModellPolicy : CORBA::Policy
```

An instance of this Real-Time Policy type is created by calling the create_priority_model_policy method of RTCORBA::RTORB. The Policy instance may then be used to configure a Real-Time POA at the time of its creation, by passing it into the create_POA method, as a member of the Policy List parameter.

See the sections "RTCORBA::RTORB" and "RTCORBA::PriorityModel"for more information.

IDL

```
interface ThreadpoolPolicy : CORBA::Policy {
  readonly attribute ThreadpoolId threadpool;
  };
```

RTCORBA::RTORB

```
class RTCORBA::RTORB : public CORBA::Object

typedef RTCORBA::RTORB* RTCORBA::RTORB_ptr
  class RTCORBA::RTORB_var
```

The interface RTCORBA::RTORB provides methods for the management of Real-Time CORBA Threadpools and Mutexes, and to create instances of Real-Time CORBA Policies.

RTCORBA::RTORB is defined in IDL, as a locality constrained interface. (See the above section "Locality Constrained Interfaces" for an explanation of this term.) Hence applications handle RTCORBA::RTORB via CORBA Object References, using the C++ classes RTCORBA::RTORB_ptr and RTCORBA::RTORB_var.

See also the sections "RTCORBA::Mutex", "RTCORBA::Priority", "RTCORBA::ThreadpoolId" and "RTCORBA::ThreadpoolPolicy". For details on the use of Real-Time CORBA Threadpools see the section "Threadpools" in the *Programmers Guide*.

RTORB Creation and Destruction

The Real-Time ORB does not need to be explicitly initialized - it is initialized implicitly as part of the regular `CORBA::ORB_init` call.

To use the Real-Time ORB operations, the application must have a reference to the Real-Time ORB instance. This reference can be obtained any time after the call to `ORB_init`, and is obtained through the `resolve_initial_references` operation on `CORBA::ORB`, with the object id string "`RTORB`" as the parameter. For details, see the section "Real-Time CORBA ORB" in the *Programmers Guide*.

IDL

```
// locality constrained interface
interface RTORB {
```

```
    Mutex create_mutex();

    void destroy_mutex( in Mutex the_mutex );
      exception InvalidThreadpool {};

    ThreadpoolId create_threadpool (
      in unsigned long stacksize,
      in unsigned long static_threads,
      in unsigned long dynamic_threads,
      in Priority default_priority,
      in boolean allow_request_buffering,
      in unsigned long max_buffered_requests,
      in unsigned long max_request_buffer_size );

      void destroy_threadpool( in ThreadpoolId threadpool )
        raises (InvalidThreadpool);

      void threadpool_idle_time( in ThreadpoolId threadpool,
      in unsigned long seconds )
      raises (InvalidThreadpool);

      PriorityModelPolicy create_priority_model_policy(
        in PriorityModel priority_model,
        in Priority server_priority );

      ThreadpoolPolicy create_threadpool_policy(
        in ThreadpoolId threadpool );
  };
```

RTORB Methods

`Mutex_ptr` **`create_mutex`**`();`

Create a new Real-Time CORBA Mutex and return a reference to it.

`void` **`destroy_mutex`**`( Mutex_ptr _the_mutex );`

Destroy a Real-Time CORBA Mutex.

Parameter	Description
`_the_mutex`	Reference of the Mutex to destroy.

```
ThreadpoolId create_threadpool(
  CORBA::ULong _stacksize,
  CORBA::ULong _static_threads,
  CORBA::ULong _dynamic_threads,
  Priority _default_priority,
  CORBA::Boolean _allow_request_buffering = 0,
  CORBA::ULong _max_buffered_requests = 0,
  CORBA::ULong _max_request_buffer_size = 0 );
```

Create a new Real-Time CORBA Threadpool with the specified configuration, and return a RTCORBA::ThreadpoolId for it.

Parameter	Description
`_stacksize`	Stacksize, in bytes, for each thread in the Threadpool.
`_static_threads`	Number of threads to create at the time of Threadpool creation. May be zero, as long as `_dynamic_threads` is non-zero.
`_dynamic_threads`	Number of extra threads that may be created, if all the statically created threads are in use and more threads are required. May be zero (so that no more threads may be dynamically created), as long as `_static_threads` is non-zero.

Parameter	Description
_default_priority	The Real-Time CORBA Priority that the threads will have when they are idle in the Threadpool.
_allow_request_ buffering	Boolean flag to enable request buffering when all threads are in use. Not supported by VisiBroker. The value of this parameter is ignored.
_max_buffered_ requests	Maximum number of requests to buffer when all threads are in use. Not supported by VisiBroker. The value of this parameter is ignored.
_max_request_buffer_ size	Maximum amount of data to buffer, in bytes, when all threads are in use. Not supported by VisiBroker. The value of this parameter is ignored.

```
void destroy_threadpool( ThreadpoolId _threadpool );
```

Destroy a Real-Time CORBA Threadpool. The Threadpool must not be in use by any Object Adapter, or the operation will fail, and a CORBA system exception is thrown.

Parameter	Description
_threadpool	The ThreadpoolId of the Threadpool to destroy.

```
void threadpool_idle_time(
  ThreadpoolId _threadpool,
  CORBA::ULong _seconds );
```

Set the time, in seconds, that dynamically allocated threads will remain idle before they are garbage collected. Configured on a per-Threadpool basis. The default is to garbage collect dynamically allocated threads after 300 seconds.

This method is a proprietary VisiBroker extension.

Parameter	Description
_threadpool	The ThreadpoolId of the Threadpool to set the Idle Time for.
_seconds	The maximum number of seconds that a dynamically allocated thread may be idle in this Threadpool before it is destroyed. Statically allocated threads are not destroyed.

```
PriorityModelPolicy create_priority_model_policy(
  in PriorityModel _priority_model,
  in Priority _server_priority );
```

Create an instance of the `RTCORBA::PriorityModelPolicy` policy object, for use in configuring one or more Real-Time POAs. See also the sections "RTCORBA::PriorityModel" and "RTCORBA::PriorityModelPolicy"

Parameter	Description
`_priority_model`	RTCORBA::SERVER_DECLARED, for the Server Declared Priority Model or RTCORBA::CLIENT_PROPAGATED for the Client Priority Propagation Model.
`_server_priority`	In the Server Model, the Real-Time CORBA Priority that invocations on objects activated on this POA will be executed at, provided a Priority value is not associated with the individual object at the time of activation. In the Client Model, the Real-Time CORBA Priority that invocations on objects activated on this POA will be executed at if they come from a non-Real-Time CORBA client or a Real-Time CORBA client that has not specified a Real-Time CORBA Priority on RTCORBA::Current before making the invocation.

```
ThreadpoolPolicy create_threadpool_policy( in
  ThreadpoolId threadpool );
```

Create an instance of the `RTCORBA::ThreadpoolPolicy` policy object, for use in configuring one or more Real-Time POAs.

Parameter	Description
`_threadpool`	The `ThreadpoolId` of the Threadpool to associate this POPA with.

RTCORBA::ServerProtocolPolicy

```
class RTCORBA::ServerProtocolPolicy : CORBA::Policy
```

An instance of this Real-Time Policy type is created by calling the create_server_protocol_policy method of RTCORBA::RTORB. The Policy instance may then be used to configure the selection of communication protocols on a Real-Time POA at the time of its creation, by passing it into the create_POA method, as a member of the Policy List parameter. The order of the Protocols in the ProtocolList dictates the order which the IOR Profile(s) will appear in the IOR of the CORBA Objects activated on that POA.

IDL

```
// IDL
  module RTCORBA {
  // Locality Constrained interface
  interface ProtocolProperties {};
  struct Protocol {
    IOP::ProfileId protocol_type;
    ProtocolProperties orb_protocol_properties;
    ProtocolProperties transport_protocol_properties;
  };
  typedef sequence <Protocol> ProtocolList;
  // Server Protocol Policy
  const CORBA::PolicyType SERVER_PROTOCOL_POLICY_TYPE = 1236;
```

```
// locality constrained interface
interface ServerProtocolPolicy : CORBA::Policy {
  readonly attribute ProtocolList protocols;
};
interface RTORB {
  ...
ServerProtocolPolicy create_server_protocol_policy
  ( in ProtocolList protocols
);
};
```

RTCORBA::ThreadpoolId

`typedef CORBA::ULong` **`RTCORBA::ThreadpoolId`**

Values of the type RTCORBA::ThreadpoolId are used to identify Real-Time CORBA Thread-pools. A value of this type is returned from the create_threadpool method of RTCORBA::RTORB.

The id may be used to initialize an instance of a Threadpool Policy, which in turn may be passed in to a call to create_POA, as a member of the PolicyList parameter, to configure a Real-Time POA. For details, see the sections "RTCORBA::RTORB", "RTCORBA::ThreadpoolPolicy" and the section "Association of an Object Adapter with a Threadpool" in the *Programmers Guide*.

IDL

```
typedef unsigned long ThreadpoolId;
```

RTCORBA::ThreadpoolPolicy

`class` **`RTCORBA::ThreadpoolPolicy`** `: CORBA::Policy`

An instance of this Real-Time Policy type is created by calling the create_threadpool_policy method of RTCORBA::RTORB. The Policy instance may then be used to configure a Real-Time POA at the time of its creation, by passing it into the create_POA method, as a member of the Policy List parameter.

See the sections "RTCORBA::RTORB" and "RTCORBA::ThreadpoolId" for more information.

IDL

```
interface ThreadpoolPolicy : CORBA::Policy {
  readonly attribute ThreadpoolId threadpool;
  };
```

Pluggable Transport Interface Classes

This chapter describes the classes of the Pluggable Transport Interface provided by VisiBroker-RT for C++. For information on how to implement support for a transport protocol via the VisiBroker Pluggable Transport Interface, see the chapter "VisiBroker Pluggable Transport Interface" in the Programmers Guide.

VISPTransConnection

```
class VISPTransConnection
```

This class is the abstract base class for a connection class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol.

Each instance of the derived class will represent a single connection between a server and a client. VisiBroker will request instances of this class be created (via the corresponding factory class, see "VISPTransConnectionFactory") on both the client and server side of the ORB, whenever a new connection is required.

Include file

The **vptrans.h** file should be included to use this class.

VISPTransConnection methods

```
virtual void close() = 0;
```

To be implemented by the derived connection class. This method closes the connection in an orderly fashion. This method must be able to close the connection from either the client- or the server-side of a connection.

```
virtual void connect(CORBA::ULongLong _timeout) = 0;
```

To be implemented by the derived connection class. This method will be called by the client-side ORB, and must communicate with the remote peer's 'Listener' instance to setup a new connection on the server-side.

The function does not return any error code, but should throw exceptions if any transport layer errors occur. Any exception may be thrown, including a CORBA User Exception, as the exception will be thrown back to the client CORBA application. CORBA::COMM_FAILURE is one possible exception that could be thrown.

The timeout value is in specified in milliseconds. A value of 0 means no timeout (block forever), and this is the default value, which is used unless the timeout is set through the VisiBroker policy system. If the transport does not support timeouts on connect, it still can be used successfully. In

this case the connect call must always block until the connection is established or has failed.

Parameter	Description
`_timeout`	Timeout value to use, in milliseconds. 0 indicates no timeout (block forever).

```
virtual void flush() = 0;
```

To be implemented by the derived connection class. If this transport buffers data, this method should immediately send all data buffered for output, and block until the data is sent. Otherwise, there is nothing to be done and it can return immediately.

```
virtual IOP::ProfileValue_ptr getPeerProfile() = 0;
```

To be implemented by the derived connection class. This method should return a copy of the Profile describing the peer endpoint used in this connection. The copy must be created on the heap and the caller is responsible for releasing the used memory. The Profile does not describe the actual connection for this instance, but the Profile of the 'Listener' endpoint used during the 'connect' call.

```
virtual CORBA::Long id() = 0;
```

To be implemented by the derived connection class. This method must return a unique number for each connection instance. The ID only needs to be unique for *this* transport. It is used to lookup/locate a connection instance during request dispatching for this transport.

```
virtual CORBA::Boolean isBridgeSignalling() = 0;
```

To be implemented by the derived connection class. This method is used to indicate to the ORB which worker thread 'cooling' strategy is to be used. If the method returns 0 (FALSE), it means that the protocol plugin itself is going to handle the re-reading of the connection after a request has been read. This is only possible if the plugin is capable of doing a blocking read with timeout on the protocol endpoint.

If it cannot or chooses not to, this method should return 1 (TRUE), and the transport bridge will notify the thread if another request becomes available or the when the timeout is reached.

Note that thread cooling only occurs if a cooling time is configured for that protocol instance.

```
virtual CORBA::Boolean isConnected() = 0;
```

To be implemented by the derived connection class. This method should return 1 (TRUE), if the remote peer is still connected. If the connection was closed by the peer or any error condition exists that prevents the use of this connection, it must return 0 (FALSE).

```
virtual CORBA::Boolean isDataAvailable() = 0;
```

To be implemented by the derived connection class. This method should return 1 (TRUE), if data is ready to be read from the connection. Otherwise, it should return 0 (FALSE).

```
virtual CORBA::Boolean no_callback() = 0;
```

To be implemented by the derived connection class. This method indicates whether a connection of this transport can be used to reverse the client/server setup and call back to a servant in the client code. It should return 0 (FALSE) if it can not, which will cause the ORB to create a new connection for this kind of call, or 1 (TRUE) if it can.

This feature is provided to support Bi-Directional IIOP, that was introduced in GIOP-1.2. See the CORBA specification for details.

```
virtual void read(CORBA::Boolean _isFirst,
  CORBA::Boolean _isLast, char* _data, CORBA::ULong
  _offset, CORBA::ULong _length, CORBA::ULongLong
  _timeout)= 0;
```

To be implemented by the derived connection class. This method reads data from the connection. It does *not* return any error code, but must signal transport related errors by throwing exceptions.

The arguments describe a byte array with a given length that needs to be filled. This function *must* either fill the complete byte array successfully, timeout, or throw an exception.

The timeout parameter's value defaults to 0 unless the user sets it through the VisiBroker QoS policies. A value of 0 indicates no timeout, and hence that the read should block forever waiting for data. Therefore, if this transport does not support timeouts on read/write, it still can be used successfully. In this case the read call must always block until all data has arrived.

Parameter	Description
_isFirst	TRUE if this is the first time data is being read from the connection.
_isLast	TRUE if this is the last time data is being read from the connection.
_data	Byte array to read data into.
_offset	Offset into the array at which to start storing the read data.
_length	The number of bytes of data to be read.
_timeout	Timeout value to use, in milliseconds. 0 indicates no timeout (block forever).

```
virtual void setupProfile(const char* prefix,
  VISPTransProfileBase_ptr peer) = 0;
```

To be implemented by the derived connection class. This method is used to tell a newly created client-side connection object what peer it should try to connect to in later steps. (When connect() is called.)

The given base class should be cast to the Profile class type of the particular transport and all member data in the connection should be initialized from

that instance. A prefix string is also passed, for property lookup, in case additional property parameters need to be read.

Parameter	Description
prefix	String prefix of the form "vbroker.se.<SE_name>.scm.<SCM_name>" that the method can use to read any protocol-specific VisiBroker properties that may have been set to configure this instance.
peer	Profile for the Listening endpoint that this connection will connect to. Given as an instance of this protocol's Profile class, passed as a pointer to the base VISPTransProfile class.

```
virtual CORBA::Boolean waitNextMessage(CORBA::ULong
  _timeout) = 0;
```

To be implemented by the derived connection class. This method should block the calling thread until either data has arrived on this connection or the given timeout (in milliseconds) has expired. It should return 1 (TRUE) if data is available, or 0 (FALSE) if not.

Note that a value of 0 for the _timeout parameter should never occur (as in this case the ORB should not call this method). Therefore receiving this value should be handled as an error, perhaps by logging an error message.

Parameter	Description
_timeout	Maximum amount of time to wait for a message (in seconds). 0 means wait forever.

```
virtual void write(CORBA::Boolean _isFirst,
  CORBA::Boolean _isLast, char* _data, CORBA::ULong
  _offset, CORBA::ULong _length, CORBA::ULongLong
  _timeout)= 0;
```

To be implemented by the derived connection class. This method sends data through the connection to the remote peer. It does *not* return any error code, but must signal transport related errors by throwing exceptions.

The arguments describe a byte array with a given length that needs to be sent. This function *must* either send the complete byte array successfully, timeout, or throw an exception.

The timeout parameter's value defaults to 0 unless the user sets it through the VisiBroker QoS policies. A value of 0 indicates no timeout, and hence that the write should block forever waiting for data. Therefore, if this transport does not support timeouts on read/write, it still can be used successfully. In this case the write call must always block until all data has arrived.

Parameter	Description
_isFirst	TRUE if this is the first time data is being sent through the connection.
_isLast	TRUE if this is the last time data is being sent through the connection.
_data	Byte array of data that needs ot be sent.
_offset	Offset into the array at which to start storing the read data.
_length	The number of bytes of data to be sent.
_timeout	Timeout value to use, in milliseconds. 0 indicates no timeout (block forever).

VISPTransConnectionFactory

`class` **`VISPTransConnectionFactory`**

This class is the abstract base class for a connection factory class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol.

A singleton instance of the derived class is registered with VisiBroker, via the VISPTransRegistrar class, described on page 14-12. The ORB calls the connection factory object to create instances of the connection class of the associated transport. The connection class is the corresponding class derived from class "VISPTransConnection".

Include file

The **vptrans.h** file should be included to use this class.

VISPTransConnectionFactory methods

`VISPTransConnection_ptr` **`create`**`(const char*` **`prefix`**`)`

To be implemented by the derived connection factory class. This method creates a new instance of the corresponding connection class and returns the pointer to it cast to the base class type. The caller is responsible for the destruction of the instance when it is no longer required.

Parameter	Description
`prefix`	String prefix of the form "`vbroker.se.<SE_name>.scm.<SCM_name>`" that the method can use to read any protocol-specific VisiBroker properties that may have been set to configure this connection factory.

VISPTransListener

`class` **`VISPTransListener`**

This class is the abstract base class for a listener factory that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol.

Instances of the derived class are created each time a Server Engine is created that includes Server Connection Managers ('SCMs') that specify the particular transport protocol. One instance is created per SCM instance that specifies the protocol.

The listener instances are used by the server-side ORB to wait for incoming connections and requests from clients. New connections and requests on existing connections are signalled by the listener to the ORB via the Pluggable Transport Interface's Bridge class (see "VISPTransBridge").

When a request is received on an existing connection, the connection goes through a 'Dispatch Cycle'. The Dispatch Cycle starts when the connection delivers data to the transport layer. In this initial state, the arrival of this data must be signalled to the ORB via the Bridge and then the Listener ignores the connection until the Dispatch process is completed (in the mean

time, the connection is said to be in the 'dispatch state'). The connection is returned to the initial state when the ORB makes a call to the Listener's `completedData()` method. During the dispatch state the ORB will read directly from the connection until all requests are exhausted, avoiding any overhead incurred by the Bridge-Listener communication.

In most cases, the transport layer uses blocking calls that wait for new connections. In order to handle this situation, the Listener should be made a subclass of the class VISThread and start a separate thread of execution that can be blocked without holding up the whole ORB. (See the MQ example transport.)

Include file

The **vptrans.h** file should be included to use this class.

VISPTransListener methods

```
virtual void completedData(CORBA::Long id) = 0;
```

To be implemented by the derived listener class. This method is called when the ORB has completed reading a request from the connection with the given id and wants the Listener once again to signal any new incoming requests on that connection (via the Bridge).

Parameter	Description
`id`	Id of the connection that may once again be listened on.

```
virtual void destroy() = 0;
```

To be implemented by the derived listener class. This method instructs the Listener instance to tear down its endpoint and close all related active connections.

```
virtual IOP::ProfileValue_ptr getListenerProfile() = 0;
```

To be implemented by the derived listener class. This method should return the Profile describing the Listener instance's endpoint on this transport. The returned Profile should be a copy on the heap and the caller (the ORB) takes over memory management of it.

```
virtual CORBA::Boolean isDataAvailable(CORBA::Long id)
  = 0;
```

To be implemented by the derived connection factory class. This method should return 1 (TRUE), if the connection with the given Id has data ready to be read. Returns 0 (FALSE) otherwise. Normally the call should just be forwarded to the transport layer to find out.

Parameter	Description
`id`	Id of the connection that should be queried to see if data is available.

```
virtual void setBridge(VISPTransBridge* up) = 0;
```

To be implemented by the derived listener class. This method establishes the 'link' to the Pluggable Transport Bridge instance to be used by this

Listener instance. The pointer it passes to the Listener should be stored to allow 'upcalls' to be made into ORB when necessary.

Parameter	Description
up	Pointer to Pluggable Transport Bridge instance that the Listener instance should use to communicate with the ORB.

VISPTransListenerFactory

class **VISPTransListenerFactory**

This class is the abstract base class for a listener factory class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol.

A singleton instance of the derived class is registered with VisiBroker, via the VISPTransRegistrar class. The ORB calls this object to create instances of the listener class of the associated transport. The listener class is the corresponding class derived from class VISPTransListener, as described in the section "VISPTransListener".

Include file

The **vptrans.h** file should be included to use this class.

VISPTransListenerFactory methods

VISPTransListener_ptr **create**(const char* **propPrefix**)

To be implemented by the derived listener factory class. This method creates a new instance of the corresponding listener class and returns the pointer to it cast to the base class type. The caller (the ORB) is responsible for the destruction of the instance when it is no longer required.

Parameter	Description
propPrefix	String prefix of the form "vbroker.se.<SE_name>.scm.<SCM_name>" that the method can use to read any protocol-specific VisiBroker properties that may have been set to configure the listener instance or the particular listener instance that is being created. Note that the factory can pass the prefix into the constructor of the listener instance it is creating, to allow it to read properties itself. This would require the derived listener class to have a constructor that takes the prefix as a parameter.

VISPTransProfileBase

```
class VISPTransProfileBase :
  public GIOP::ProfileBodyValue,
  public CORBA_DefaultValueRefCountBase
```

This class is the abstract base class for a Profile class that must be implemented for each transport protocol that is to be plugged in to

VisiBroker, to allow VisiBroker to work with that particular transport protocol.

This class provides the functionality to convert between a transport specific endpoint description and an CORBA IOP based IOR that can be exchanged with other CORBA implementations. It is also used during the process of binding a client to a server, by passing a ProfileValue to a 'parsing' function that has to return TRUE or FALSE, to determine whether a particular IOR is usable for this transport or not.

An instance of the derived Profile class is frequently passed to functions via a pointer to its base class type. In order to support safe runtime downcasting with any C++ compiler, a '_downcast' function must be provided that can test if the cast is legal or not. See the 'MQ' example code for an example.

Include file

The **vptrans.h** file should be included to use this class.

VISPTransProfileBase methods

```
static GIOP::ObjectKey* convert(const
  PortableServer::ObjectId& seq);
```

Converts octet sequence representation of an Object Key into the in-memory representation.

Parameter	Description
seq	Octet sequence version of Object Key, to be converted into in-memory representation.

void **object_key**(GIOP::ObjectKey_ptr **k**);

Set the Object Key for this Profile instance.

Parameter	Description
k	Object key

```
const GIOP::ObjectKey_ptr object_key() const;
```

Get the Object Key for this Profile instance.

```
void  version(const  GIOP::Version& v);
```

Set the GIOP version for this Profile.

Parameter	Description
v	GIOP version.

```
GIOP::Version& version();
```

Get the GIOP version of this Profile. const GIOP::Version& **version**() const; Get the GIOP version of this Profile.

```
static const VISValueInfo& _info();
```

Get the VisiBroker ValueInfo for this Profile type.

VISPTransProfileBase members

```
static const VISValueInfo& _stat_info;
```

Stores the VisiBroker ValueInfo for this particular Profile type.

VISPTransProfileBase base class methods

```
IOP::ProfileValue_ptr copy()
```

To be implemented by the derived listener factory class. This method should make an exact copy on the free store and return a pointer to it. It is good coding practice to use the copy constructor inside of this function.

```
CORBA::Boolean matchesTemplate(IOP::ProfileValue_ptr
  body)
```

To be implemented by the derived Profile class. This method should return 1 (TRUE) if there is an IOR in the given data, that can be used to connect through this transport.

Otherwise return 0 (FALSE).

Parameter	Description
body	Profile to be checked, to see if it can be used by this transport.

```
IOP::ProfileId tag()
```

To be implemented by the derived Profile class. This method should return the unique tag value for this Profile.

```
IOP::TaggedProfile* toTaggedProfile()
```

To be implemented by the derived Profile class. This method should return a tagged (stringified) Profile instance created with the values read from this instance's member data.

```
static VISPTransProfileBase*
  _downcast(CORBA::ValueBase* vbptr);
```

To be implemented by the derived Profile class. Function to downcast a base class pointer to an instance of this Profile class.

Parameter	Description
vbptr	Profile instance passed as base Value type pointer.

```
virtual void* _safe_downcast(const VISValueInfo &info)
  const;
```

To be implemented by the derived listener factory class. Virtual method called by ORB during downcast, to check type info data.

Parameter	Description
info	VisiBroker Value Info for this Profile type.

VISPTransProfileFactory

`class` **`VISPTransProfileFactory`**

This class is the abstract base class for a Profile factory class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol.

A singleton instance of the derived class is registered with VisiBroker, via the `VISPTransRegistrar` class. The ORB calls this object to create instances of the Profile class of the associated transport. The Profile class is the corresponding class derived from class `VISPTransProfileBase`, as described in the section "VISPTransProfileBase".

Include file

The **vptrans.h** file should be included to use this class.

VISPTransProfileFactory methods

`IOP::ProfileValue_ptr` **`create`**`(const IOP::TaggedProfile&` **`profile`**`)`

Read the tagged IOR and create a Profile describing a Listener endpoint.

Parameter	Description
`profile`	CDR encoded IOR to be read.

`CORBA::ULong` **`hash`**`(VISPTransProfileBase_ptr` **`prof`**`)`

Support the optimized storage of profiles in a hashed lookup table by calculating a hash number for the given instance. Return 0 if you do not provide hash values.

Parameter	Description
`prof`	Profile instance to produce hash value for.

`IOP::ProfileId` **`getTag`**`()`

Return the unique Profile Id tag for the type of Profile created by this factory.

VISPTransBridge

`class` **`VISPTransBridge`**

This class provides a generic interface between the transport classes and the ORB. It provides methods to signal various events occuring in the transport layer.

Include file

The **vptrans.h** file should be included to use this class.

VISPTransBridge methods

`CORBA::Boolean` **`addInput`**`(VISPTransConnection_ptr` **`con`**`)`

Send a connection request to the ORB through the bridge, by passing a pointer to the Connection instance representing the Listener endpoint. The returned flag signals whether the ORB has accepted the new connection (returns 1 (TRUE)) or refused it (returns 0 (FALSE)). The latter might happen due to resource constraints or due to a restriction on connections (set up through the property system).

Parameter	Description
`con`	Connection object representing the Listener endpoint wish to connect to.

`void` **`signalDataAvailable`**`(CORBA::Long` **`conId`**`)`

Passes the connection id to the ORB of a connection that just got new data from the transport layer. This will start the dispatch cycle for incoming requests.

Parameter	Description
`conId`	Connection Id of connection want to indicate data is available on.

`void` **`closedByPeer`**`(CORBA::Long` **`conId`**`)`

Tell the ORB that the connection with the given id was closed by the remote peer.

Parameter	Description
`conId`	Connection Id of connection want to indicate was closed by the remote peer.

VISPTransRegistrar

`class` **`VISPTransRegistrar`**

This class must be used to register a new transport with the ORB. The protocol name string given during registration is used as identifier of this transport and must be unique in the scope of that ORB. It is also used as a prefix in the name string of properties related to this transport.

Include file

The **vptrans.h** file should be included to use this class.

VISPTransRegistrar methods

```
static void addTransport(const char* protocolName,
  VISPTransConnectionFactory*connFac,
```

```
  VISPTransListenerFactory*  listFac,
  VISPTransProfileFactory* profFac)
```

Register the protocol name string and the three Factory instances used to create specific classes for this transport. This method is static and can therefore be called at any time during the initialization of the ORB.

Parameter	Description
protocolName	Name to be used to identify this transport protocol.
connFac	Pointer to singleton instance of connection factory.
listFac	Pointer to singleton instance of Listener factory.
profFac	Pointer to singleton instance of Profile factory.

VisiBroker Logging Classes

This chapter describes the VisiBroker Logging classes. Read the VisiBroker Logging chapter in the VisiBroker-RT for C++ Programmer's Guide before using these classes.

Introduction

VisiBroker-RT for C++ provides a logging mechanism which allows applications to log messages and have them directed, via configurable *logging forwarders,* to an appropriate destination or destinations. The ORB itself uses this mechanism for the output of any error, warning or informational messages.

VisiBroker Logging employs one or more Logger objects, that applications (including the ORB) may log messages to. When a message is logged to a Logger, it is queued rather than being output by the calling thread.

Each Logger has one or more Forwarders associated with it: application-definable pieces of code that read the queued messages and forward them to desired destinations such as standard error, a file or over a network. All the Forwarders associated with a given Logger run on a single Forwarder Thread. The priority of the Forwarder Thread is configurable.

However, forwarding is not enabled when a Logger is created. Messages logged before forwarding is enabled are queued until it is enabled. This allows messages to be logged before the Logger or all of the output destinations have been fully configured (for example during static initialization of C++ constructors.)

The ORB uses a special Logger instance (the 'Default Logger'), which is created automatically the first time the ORB logs a message to it. Applications can log messages to the Default Logger as well, to integrate their logging output with that of the ORB, or they can create one or more other Loggers, to log messages independently. The 'standard error' iostream is the default destination for messages logged to the Default Logger.

The following interfaces and classes are described in the sections that follow:

- `VISLogArgs`
  - `VISLogArgsType, VISLogInteger, VISLogString, VISLogBoolean`
- `VISLogApplicationFields`
- `VISLogger`
- `VISLoggerForwarder`
- `VISLoggerManager`
- `VISLogMessage`
- `VISLoggerStaticInfo`

Include file

To use any of the VisiBroker Logging features described in this chapter, the application should include the file **`vlogger.h`**, which is one of the include files supplied with VisiBroker.

VISLogArgs

```
class VISLogArgs;

  class VISLogArgsType;
  class VISLogInteger : public VISLogArgsType;
  class VISLogString : public VISLoogArgsType
```

The VISLogArgs class is used to pass a set of arguments as part of a message logged through the VisiBroker Logging mechanism. The log method of the VISLogger class takes a pointer to a VISLogArgs object as a parameter (which may be null.) The same pointer value will be passed to each VISLoggerForwarder that that message is forwarded to - either as a field of the VISLogMessage structure given as a parameter of the forward_message method or as a parameter to the handle_memory_failure method.

Note that the VISLogger that the pointer to a VISLogArgs object is passed to takes ownership of that VISLogArgs instance, and destroys it after it has been used by all the VISLoggerForwarder instances associated with that VISLogger.

VISLogArgs Methods

```
VISLogArgs( VISLogArgsType *param1,
  VISLogArgsType *param2 = 0,
  VISLogArgsType *param3 = 0,
  VISLogArgsType *param4 = 0,
  VISLogArgsType *param5 = 0,
  VISLogArgsType *param6 = 0,
  VISLogArgsType *param7 = 0,
  VISLogArgsType *param8 = 0,
  VISLogArgsType *param9 = 0,
  VISLogArgsType *param10 = 0 );
```

The VISLogArgs constructor is used to create a new VISLogArgs instance, that may contain between one and ten arguments. Each argument is passed in as a parameter to the constryctor, and may be of any type derived from the VISLogArgsType base class. Integer, String and Boolean argument types are provided. See the sub-sections below.

Note that the VISLogArgs object takes ownership of the arguments that are passed in to its constructor. It, and the arguments, are destroyed by the VISLogger that it is passed into, after it has been used by all the VISLoggerForwarders associated with that VISLogger.

Parameter	Description
_threadpool	The ThreadpoolId of the Threadpool to associate this POA with.

VISLogArgsType

```
class VISLogArgsType;

enum Type { INTEGER, STRING, BOOLEAN };
```

The VISLogArgsType class is the base class for the different argument types that may be stored in a VISLogArgs object.

VISLogArgsType Methods

VISLogArgsType(Type type);

The VISLogArgsType constructor is called as a base class initializer by the constructor of each of the particular Log Argument types : VISLogInteger, VISLogString and VISLogBoolean. The type parameter is set to the appropriate value of the VISLogArgsType::Type enumeration.

Parameter	Description
type	Type of the particular Log Argument

Type **data_type**();

Returns the particular type of this Log Argument, as a value of the VISLogArgsType::Type enumeration.

VISLogInteger

class **VISLogInteger** : public VISLogArgsType;

The VISLogInteger class is use to store an integer value as part of a VISLogArgs object, that canbe used as part of a message logged via a VISLogger.

VISLogInteger Methods

VISLogInteger(CORBA::Long integer);

The VISLogInteger constructor takes an integer value as a parameter. This is the value that the VISLogInteger instance stores.

Parameter	Description
integer	Integer value to store in the VISLogInteger instance

CORBA::Long **integer_value**();

Returns the integer value stored in the VISLogInteger instance.

VISLogString

class **VISLogString** : public VISLogArgsType;

The VISLogString class is use to store a string value as part of a VISLogArgs object, that can be used as part of a message logged via a VISLogger.

VISLogString Methods

VISLogString(const char * string, int destroy_flag = 0);

The VISLogString constructor takes a string value as a parameter. This is the value that the VISLogString instance stores. The destroy flag determines whether the VISLogString copies the passed in string (default)

or takes ownership of the passed in copy and destroys it after it is no longer needed for logging.

Parameter	Description
string	String value to store in the VISLogInteger instance.
destroy_flag	Flag indicating whether the VISLogString copies the passed in string (default) or takes ownership of the passed in copy and destroys it is no longer needed for logging.

```
const char* string_value();
```

Returns a pointer to the string value stored in the VISLogString instance.

VISLogBoolean

```
class VISLogBoolean : public VISLogArgsType;
```

The VISLogBoolean class is use to store a boolean value as part of a VISLogArgs object, that canbe used as part of a message logged via a VISLogger.

VISLogBoolean Methods

```
VISLogBoolean( const CORBA::Boolean boolean );
```

The VISLogBoolean constructor takes a boolean value as a parameter. This is the value that the VISLogBoolean instance stores.

Parameter	Description
boolean	Boolean value to store in the VISLogBoolean instance.

```
CORBA::Boolean boolean_value();
```

Returns the boolean value stored in the VISLogBoolean instance.

VISLogApplicationFields

```
class VISLogApplicationFields;
```

The VISLogApplicationFields class is a base class provided to allow an application to pass any additional information it wants as part of a logged message.

The log method of the VISLogger class takes a pointer to a VISLogApplicationFields object as a parameter (which may be null.) The same pointer value will be passed to each VISLoggerForwarder that that message is forwarded to - either as a field of the VISLogMessage structure given as a parameter of the forward_message method or as a parameter to the handle_memory_failure method.

To pass additional information with a log message, an application derives a class from the VISLogApplicationFields class, and passes a pointer to an instance of this derived class as a parameter when it logs messages.

Note that the VISLogger that the pointer to a VISLogApplicationFields object is passed to takes ownership of that VISLogApplicationFields instance, and

destroys it after it has been used by all the VISLoggerForwarder instances associated with that VISLogger.

VISLogApplicationFields Methods

VISLogApplicationFields(CORBA::Long type_id);

The VISLogApplicationFields constructor is called as a base class initializer by the constructor of a particular application-defined application fields class. The type id is an application-defined integer value, that can be used to distinguish one application fields class instance from another.

Parameter	Description
type_id	Application-assigned integer value, that can be used to distinguish instances of a particular type of derived application fields class.

CORBA::Long **type_id**();

Returns the application-assigned integer value associated with a particular derived application fields class type.

VISLogger

class **VISLogger** : public VISThread;

VISLogger class instances represent individual VisiBroker Logger instances. A new VISLogger instance can be explicitly created by the application by calling the get_logger method of the VISLoggerManager class. A Default Logger, with the name 'DefaultLogger' is automatically created by the ORB and used to log ORB events.

VISLogger Methods

```
void log( const  char *source_name,
  VISLogLevellevel,
  const char *message_key,
  VISLogArgs *message_args,
  const char *source_thread_identifier,
  const char *location_code,
  VISLogApplicationFields * application_fields );
```

Log a message via this VisiBroker Logger instance. The caller continues to own the source_name, message_key, source_thread_identifier and location_code strings, and the Logger makes its own copies of them. But the Logger takes ownership of the message_args and application_fields objects, if either or both is supplied.

Parameter	Description
source_name	Application or application component that is logging the message.
level	Log Level of the message. Messages logged by the ORB use this field to indicate one of four levels.
message_key	What kind of message this is. The ORB uses a fixed set of message keys, so that there is a well known set of message types.

Parameter	Description
message_args	VISLogArgs pointer. May be null. The Logger takes ownership of the object.
source_thread_ identifier	Thread that logged this message. If this field is left null, the ORB will provide a default value.
location_code	Location in application code that is logging this message on this occasion. For ORB log messages, this is the source code file name and line number of the calling line of ORB code (produced using the ANSI C FILE and LINE macros.)
application_fields	VISLogApplicationFields pointer. May be null. The Logger takes ownership of the object.

```
VISLoggerStaticInfo& static_info();
```

Return a pointer to the VISLoggerStatic information for this VISLogger instance.

```
void static_info( VISLoggerStaticInfo& static_info );
```

Set the VISLoggerStatic information for this VISLogger instance.

Parameter	Description
static_info	VISLoggerStatic information to apply to the VISLogger.

```
void add_forwarder( VISLoggerForwarder_ptr forwarder );
```

Register an application-defined VISLoggerForwarder with this VISLogger instance. The VISLoggerForwarder will be added to the list of already installed Forwarders that are associated with this VISLogger instance.

Parameter	Description
forwarder	Pointer to the VISLoggerForwarder instance to register with this VISLogger.

```
void remove_forwarder( VISLoggerForwarder_ptr forwarder
  );
```

Unregister an application-defined VISLoggerForwarder instance previously registered with this VISLogger instance.

Parameter	Description
forwarder	Pointer to the VISLoggerForwarder instance to unregister.

```
void remove_default_forwarder();
```

Stop the Default Forwarder from being executed for this VISLogger.

```
void forwarder_priority( CORBA::Short priority );
```

Set the Real-Time CORBA Priority that the Forwarder thread associated with this VISLogger will run at. Ignored if set after forwarding is enabled (which happens automatically at the time of ORB initialization in the case of the Default Logger.)

Parameter	Description
priority	Real-Time CORBA Priority value to run the Forwarder thread associated with this Logger at.

```
CORBA::Short forwarder_priority();
```

Get the Real-Time CORBA Priority that the Forwarder thread associated with this Logger is/will be running at.

```
void enable_forwarding();
```

Enable the forwarding of logged messages to all the Forwarders currently registered with this Logger. At this time the Forwarder thread for this Logger is created. Occurs automatically during ORB_init for the Default Logger, but may still be forced to occur earlier by calling this method.

VISLoggerForwarder

```
class VISLoggerForwarder;
```

This class is used as a base class for Logger Forwarder implementations that an application wished to create. The application inherits from this class, overrides the implementation of the methods described below, and registers an instance of the derived class with a VISLogger in order to control how that VISLogger forward logged messages.

VISLoggerForwarder Methods

```
virtual void forward_message( VISLogMessage * message
  );
```

Forward the message that is passed as a VISLogMessage parameter. Note that the VISLogger keeps ownership of the VISLogMessage and all its members, so that the Forwarder must copy any data that it wishes to retain or pass to a function or method that wil retain it.

Parameter	Description
message	Pointer to a VISLogMessage structure that stores the information for a logged message.

```
virtual void handle_memory_failure(
  CORBA::ULongLongmessage_identifier, CORBA::ULongLong
  message_creation_time, VISLogLevellevel,
  const char *source_host,
  const char *source_name,
  const char *location_code,
  CORBA::ULongsource_process_identifier,
  const char *source_thread_identifier,
  VISLogApplicationFields * application_fields,
  const char *message_key,
  VISLogArgs *message_args );
```

Called when a memory allocation failure occurs at any point in the queuing of a logged message. In this case the individual pieces of input information used to assemble a VISLogMessage structure are passed as parameters. This method is called in the context of the thread that was logging the message (from inside the call to VISLogger::log.) The parameters are owned by the caller, and the implementation of the method probably should not attempt to copy them, given that this method is only called when a

memory allocation failure has occurred. Some of the parameters may be null pointers, depending on when the memory allocation failure occurred.

Parameter	Description
message_identifer	Message sequence number, stating at one and incrementing for each message logged to that Logger.
message_creation_time	Timestamp, taken from the system clock at the time the message was logged (rather than forwarded.) Held in the TimeBase::TimeT format: one unit is 100 nanoseconds or one tenth of a microsecond.
level	Log Level of the message. Messages logged by the ORB use this field to indicate one of four levels.
source_host	From VISStaticInfo for that Logger.
source_name	Application or application component that is logging the message.
location_code	Location in application code that is logging this message on this occasion. For ORB log messages, this is the source code file name and line number of the calling line of ORB code (produced using the ANSI C_FILE and _LINE_ macros.)
source_process_identifier	From VISStaticInfo for that Logger.
source_thread_identifier	Thread Id of thread that message is logged from
application_fields	VISLogApplicationFields pointer. May be null. The Logger takes ownership of the object.
message_key	Message key parameter passed in to log call.
message_args	VISLogArgs parameter passed in to log call.

VISLoggerManager

```
class VISLoggerManager;
```

The VISLoggerManager class has methods to allow the management of VISLogger instances (creation/destruction) and to control the level of logging output created by different components of VisiBroker.

VISLoggerManager Methods

```
static VISLoggerManager_ptr instance();
```

Return a pointer to the singleton VISLoggerManager instance, which is created upon first use (the first time this method is called.)

```
VISLogger_ptr get_logger( const char * logger_name,
  CORBA::Boolean create_flag = 1 );
```

Return a pointer to a Logger with the specified name. If the Logger does not already exist, and the create_flag is true (default behavior) then a Logger of that name is created.

Parameter	Description
logger_name	Name of the Logger to look up/create.
create_flag	Choose whether a Logger of that name will be created if it doesn't currently exist.

`void` **`destroy_logger`**`( const char * logger_name );`

Destroy the specified Logger. Blocks until all the messages currently queued for forwarding have been forwarded.

Parameter	Description
`logger_name`	Name of Logger to destroy.

`void` **`ORB_log_level`**`( VISLogLevel level );`

Set the maximum log level that is logged from the main, ORB component of VisiBroker.

Parameter	Description
`level`	The maximum log level that is to be logged from the main, ORB component of VisiBroker.

`VISLogLevel` **`ORB_log_level`**`();`

Get the maximum log level that is logged from the main, ORB component of VisiBroker.

`void` **`POA_log_level`**`( VISLogLevel level );`

Set the maximum log level that is logged from the POA component of VisiBroker.

Parameter	Description
`level`	The maximum log level that is to be logged from the POA component of VisiBroker.

`VISLogLevel` **`POA_log_level`**`();`

Get the maximum log level that is logged from the POA component of VisiBroker.

`void` **`OSAgent_log_level`**`( VISLogLevel level );`

Set the maximum log level that is logged from the OSAgent component of VisiBroker.

Parameter	Description
`level`	The maximum log level that is to be logged from the OSAgent component of VisiBroker.

`VISLogLevel` **`OSAgent_log_level`**`();`

Get the maximum log level that is logged from the OSAgent component of VisiBroker.

`void` **`LocSvc_log_level`**`( VISLogLevel level );`

Set the maximum log level that is logged from the Location Service component of VisiBroker.

Parameter	Description
`level`	The maximum log level that is to be logged from the Location Service component of VisiBroker.

```
VISLogLevel LocSvc_log_level();
```

Get the maximum log level that is logged from the Location Service component of VisiBroker.

```
void CosName_log_level( VISLogLevel level );
```

Set the maximum log level that is logged from the COS Naming Service component of VisiBroker.

Parameter	Description
level	The maximum log level that is to be logged from the Naming Service component of VisiBroker.

```
VISLogLevel CosName_log_level();
```

Set the maximum log level that is logged from the COS Naming Service component of VisiBroker.

```
void CosEvent_log_level( VISLogLevel level );
```

Set the maximum log level that is logged from the COS Event Service component of VisiBroker.

Parameter	Description
level	The maximum log level that is to be logged from the Event Service component of VisiBroker.

```
VISLogLevel CosEvent_log_level();
```

Get the maximum log level that is logged from the COS Event Service component of VisiBroker.

```
void default_forwarder_thread_priority( CORBA::Short
  priority );
```

Set the default Real-Time CORBA Priority that the Forwarder thread of a Logger will run at if it is not explicitly set on the Logger instance before forwarding is enabled.

Parameter	Description
priority	Real-Time CORBA Priority to use as default for the Forwarder thread priority, if not explicitly configured for a Logger instance.

```
CORBA::Short default_forwarder_thread_priority();
```

Get the default Real-Time CORBA Priority that the Forwarder thread of a Logger will run at if it is not explicitly set on the Logger instance before forwarding is enabled.

VISLogMessage

```
struct VISLogMessage {
  CORBA::ULongLongmessage_identifier;
  CORBA::ULongLongmessage_creation_time;
  VISLogLevellevel;
  const char *source_host;
  const char *source_name;
```

```
  onst char *location_code;
  CORBA::ULongsource_process_identifier;
  const char *source_thread_identifier;
  VISLogApplicationFields * application_fields;
  const char *message_key;
  VISLogArgs *message_args;

  VISLogMessage() {}
  VISLogMessage();
  };
```

A data structure that is assembled by the VISLogger when a message is logged to it, to store all the information associated with that message. When the message is forwarded, a pointer to the VISLogMessage instance created for a message is passed in turn to each of the Forwarders associated with that Logger. The fields have the following meaning.

Parameter	Description
message_identifer	Message sequence number, stating at one and incrementing for each message logged to that Logger.
message_creation_time	Timestamp, taken from the system clock at the time the message was logged (rather than forwarded.) Held in the TimeBase::TimeT format: one unit is 100 nanoseconds or one tenth of a microsecond.
level	Log Level of the message. Messages logged by the ORB use this field to indicate one of four levels.
source_host	From VISStaticInfo for that Logger.
source_name	Application or application component that is logging the message.
location_code	Location in application code that is logging this message on this occasion. For ORB log messages, this is the source code file name and line number of the calling line of ORB code (produced using the ANSI C_FILE and LINE_ macros.
source_process_ identifier	From VISStaticInfo for that Logger.
source_thread_ identifier	Thread Id of thread that message is logged from.
application_fields	VISLogApplicationFields pointer. May be null. The Logger takes ownership of the object.
message_key	Message key parameter passed in to log call.
message_args	VISLogArgs parameter passed in to log call.

VISLoggerStaticInfo

```
struct VISLoggerStaticInfo {
  const char * source_host;
  CORBA::ULong source_process_identifier;

  VISLoggerStaticInfo::VISLoggerStaticInfo();
  ~VISLoggerStaticInfo();
  void operator=(const struct VISLoggerStaticInfo&
  info);
  };
```

A data structure that holds information that is common to all messages logged by a Logger. May be read, modified and set for each Logger instance. The members have the following significance:

Parameter	Description
`source_host`	Defaults to the hostname or dot notation IP address for the host the Logger is running on. The application may assign any string value.
`source_process_identifier`	Defaults to the Process Id for the process that the Logger is running in (if running on an OS that uses the process model, otherwise a null string.) The application may assign any string value.

Quality of Service Interfaces and Classes

This chapter describes the VisiBroker-RT for C++ implementation of the Quality of Service APIs. See "PortableServer::POA" for information about creating policies.

CORBA::PolicyManager

`class CORBA::`**`PolicyManager`**

This class is used to set and access policy overrides at the ORB level.

IDL definition

```
module CORBA {
  interface PolicyManager {
  PolicyList get_policy_overrides(in PolicyTypeSeq ts);
  void set_policy_overrides(in PolicyList policies,
    in SetOverrideType set_add)
    raises (InvalidPolicies);
  };
};
```

Methods

`PolicyList get_policy_overrides (PolicyTypeSeq` **`ts`**`);`

This method returns a policy list containing the policies of the requested policy types. If the specified sequence is empty (that is, if the length of the list is zero), all Policies at this scope are returned. If none of the requested policy types is set at the target **PolicyManager**, an empty sequence is returned.

`void set_policy_overrides (PolicyList` **`policies`**`,`
`  CORBA::SetOverrideType` **`set_add`**`)`

This method updates the current set of policies with the requested list of policy overrides. Invoking set_policy_overrides with an empty sequence of policies and a mode of SET_OVERRIDE removes all overrides from a PolicyManager. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempt to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception. If the request would put the set of overriding policies for the target PolicyManager in an inconsistent state, no policies are changed or added, and the exception InvalidPolicies is raised. There is no evaluation of compatibility with policies set within other PolicyManagers.

Note

The set_policy_overrides() method throws an exception in Messaging, but doesn't in CORBA 2.3. The PolicyManager::set_policy_overrides throws InvalidPolicies.

Parameter	Description
policies	A sequence of references to Policy objects.
set_add	Indicates whether these policies should be added (ADD_OVERRIDE) to any other overrides that already exist in the PolicyManager, or added to a clean PolicyManager free of any other overrides (SET_OVERRIDE).

CORBA::PolicyCurrent

class CORBA::PolicyCurrent

This class provides access to policies overridden at the thread level and is defined with operations for querying and applying quality of service values to a thread. Policies defined at the thread level override any system defaults or values set at the ORB level but not those at the Object level. The instance belonging to the current thread is accessible by using resolve_initial_reference("PolicyCurrent") and narrowing down to PolicyCurrent.

IDL definition

```
interface PolicyCurrent : PolicyManager, Current {
};
```

CORBA::Object

class CORBA::**Object**

The Visibroker implementation of the Quality of Service API allows policies to be assigned to objects, threads, and ORBs. Policies assigned to Objects override all other policies.

IDL definition

```
#pragma prefix "omg.org"
module CORBA {
interface Object {
  Policy get_client_policy(in PolicyType type);
  Policy get_policy(in PolicyType type);
  PolicyList get_policy_overrides(in PolicyTypeSeq types);
  Object set_policy_overrides(in PolicyList policies,in
        SetOverrideType set_add)
    raises (InvalidPolicies);
  boolean validate_connection(out PolicyList
        inconsistent_policies);
  };
};
```

Methods

CORBA::Policy_ptr get_client_policy(CORBA::PolicyType **type);**

Returns the effective overriding Policy for the object reference. The effective override is obtained by first checking for an override of the given PolicyType at the Object scope, then at the Current scope, and finally at the ORB

scope. If no override is present for the requested PolicyType, the system-dependent default value for that PolicyType is used. Portable applications are expected to set the desired "defaults" at the ORB scope since default Policy values are not specified.

```
CORBA::Policy_ptr get_policy(CORBA::PolicyType type);
```

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 or validate_connection 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 exception CORBA::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 RebindPolicy has a value of TRANSPARENT, the effective Policy may change from invocation to invocation due to transparent rebinding.

Note

In the Visibroker implementation, this method gets the Policy assigned to an Object, thread or ORB.

```
CORBA::Object set_policy_overrides(const PolicyList& _policies,
  CORBA::SetOverrideType _set_add);
```

This method works as does the PolicyManager method of the same name. However, it updates the current set of policies of an Object, thread or ORB with the requested list of Policy overrides. In addition, this method returns a CORBA::Object whereas other methods of the same name return void.

In CORBA 2.3, CORBA::Object::set_policy_overrides does not throw exceptions, but it does for Messaging.

```
CORBA::Boolean validate_connection(PolicyList
  inconsistent_policies);
```

Returns the value TRUE if the current effective policies for the Object will allow an invocation to be made. If the object reference is not yet bound, a binding will occur as part of this operation. If the object reference is already bound, but current policy overrides have changed or for any other reason the binding is no longer valid, a rebind will be attempted regardless of the setting of any RebindPolicy override. The validate_connection operation is the only way to force such a rebind when implicit rebinds are disallowed by the current effective RebindPolicy. The attempt to bind or rebind may involve processing GIOP LocateRequests by the ORB. Returns the value FALSE if the current effective policies would cause an invocation to raise the system exception INV_POLICY. If the current effective policies are incompatible, the out parameter inconsistent_policies contains those policies causing the incompatibility. This returned list of policies is not

guaranteed to be exhaustive. If the binding fails due to some reason unrelated to policy overrides, the appropriate system exception is raised.

Messaging::RebindPolicy

`class Messaging::RebindPolicy`

The Visibroker implementation of RebindPolicy is a complete implementation of RebindPolicy as defined in the orbos/98-05-05 Messaging Specification with enhancements to support failover.

The RebindPolicy of an ORB determines how it handles GIOP location-forward messages and object failures. The ORB handles fail-over/rebind by looking at the effective policy at the CORBA::Object instance.

The OMG implementation, derived from `CORBA::Policy`, determines whether the ORB may transparently rebind once it is successfully bound to a target server. The extended implementation determines whether the ORB may transparently failover once it is successfully bound to a target Object, thread, or ORB.

IDL definition

```
#pragma prefix "omg.org"
module Messaging {
  typedef short RebindMode;
  const CORBA::PolicyType REBIND_POLICY_TYPE = 23;
  interface RebindPolicy CORBA::Policy {
    readonly attribute RebindMode rebind_mode;
  };
}
```

Policy values

Note

Policies are enforced only after a successful bind.

The OMG Policy values that can be set as the Rebind Policy are:

Policy value	Description
TRANSPARENT	This policy allows the ORB to silently handle object-forwarding and necessary reconnection when making a remote request. This is the least restrictive OMG policy value.
NO_REBIND	This policy allows the ORB to silently handle reopening of closed connections while making a remote request, but prevents any transparent object-forwarding that would cause a change in the client-side effective QoS policies.
NO_RECONNECT	This policy prevents the ORB from silently handling object-forwards or the reopening of closed connections. This is the most restrictive OMG policy value.

The VisiBroker-specific values that can be set as the Rebind Policy are:

Policy value	Description
VB_TRANSPARENT	This policy extends TRANSPARENT behavior to failover conditions in the object, thread and ORB. This is the default policy. If this policy is set, if a remote invocation fails because the server object goes down, then the ORB tries to reconnect to another server using the osagent. The ORB masks the communication failure and does not throw an exception to the client.
VB_NOTIFY_REBIND	VB_NOTIFY_REBIND behaves as does VB_TRANSPARENT but throws an exception when the communication failure is detected. It will try to transparently reconnect to another object if the invocation is re-attempted.
VB_NO_REBIND	VB_NO_REBIND does no failover. It only allows the client ORB to reopen a closed GIOP re-connection to the same server; it does not allow object forwarding of any kind.

Messaging::RelativeRequestTimeoutPolicy

`class Messaging::RelativeRequestTimeoutPolicy`

The Visibroker implementation of RelativeRequestTimeoutPolicy is a complete implementation of RelativeRequestTimeoutPolicy as defined in the orbos/98-05-05 Messaging Specification.

RelativeRequestTimeoutPolicy is used to indicate the relative amount of time for which a Request may be delivered. After this amount of time the Request is cancelled. This policy is applied to both synchronous and asynchronous invocations.

When instances of **RelativeRequestTimeoutPolicy** are created, a value of type **TimeBase::TimeT** is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override.

IDL definition

```
#pragma prefix "omg.org"
module Messaging {
   const CORBA::PolicyType RELATIVE_REQ_TIMEOUT_POLICY_TYPE = 31;
   interface RelativeRequestTimeoutPolicy CORBA::Policy {
   readonly attribute TImeBase::TimeT relative_expiry;
   };
}
```

Messaging::RelativeRoundtripTimeoutPolicy

`class Messaging::RelativeRoundtripTimeoutPolicy`

The Visibroker implementation of RelativeRoundtripTimeoutPolicy is a complete implementation of RelativeRoundtripTimeoutPolicy as defined in the orbos/98-05-05 Messaging Specification.

RelativeRoundtripTimeoutPolicy is used to indicate the relative amount of time for which a Request or its corresponding Reply may be delivered. After this amount of time the Request is cancelled (if a response has not yet been received from the target) or the Reply is discarded (if the Request had

already been delivered and a Reply returned from the target). This policy is applied to both synchronous and asynchronous invocations.

When instances of **RelativeRoundtripTimeoutPolicy** are created, a value of type **TimeBase::TimeT** is passed to **CORBA::ORB::create_polic**y. This policy is only applicable as a client-side override.

IDL definition

```
#pragma prefix "omg.org"
module Messaging {
  const CORBA::PolicyType RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
  interface RelativeRoundtripTimeoutPolicy CORBA::Policy {
    readonly attribute TImeBase::TimeT relative_expiry;
  };
}
```

QoSExt::DeferBind Policy

`class QoSExt::DeferRebindPolicy`

By default, the ORB connects to the (remote) object when it receives a bind() or a string_to_object call.

If set to TRUE, this policy changes this behavior; it causes the ORB to delay contacting the Object until the first invocation.

IDL definition

```
#pragma prefix "highlander.com"
module QoSExt {
  interface DeferBindPolicy :CORBA::Policy {
     readonly attribute boolean value;
  };
};
```

QosExt::RelativeConnectionTimeoutPolicy

`class Messaging::RelativeConnectionTimeoutPolicy`

RelativeConnectionTimeoutPolicy is used to indicate the relative amount of time after which an attempt to connect to the server ORB using one of the available communication endpoints is aborted. After this amount of time the connection attempt is aborted. This policy is applied to both synchronous and asynchronous invocations.

When instances of **RelativeConnectionTimeoutPolicy** are created, a value of type **TimeBase::TimeT** is passed to **CORBA::ORB::create_polic**y. This policy is only applicable as a client-side override.

IDL definition

```
module QoSExt {
  interface RelativeConnectionTimeoutPolicy :CORBA::Policy {
    readonly attribute TimeBase::TimeT relative_expiry;
  };
  };
}
```

QoSExt::SmartBind Policy

`class QoSExt::SmartBindPolicy`

SmartBindPolicy is a local object (i.e. locality constrained) derived from **CORBA::Polic**y. It is used to control the VisiBroker SmartBinding optimization. The currently supported values are:

- QosExt::SMARTBIND_OFF

  When SmartBindPolicy is set to `QosExt::SMARTBIND_OFF`, communications between the VisiBroker client and server will use the local IP LOOPBACK interface, thereby ignoring any optimization. This option existed with prior versions of VisiBroker-RT for C++ as a bind option that could be specified as a parameter to the _ bind method.

- QosExt::SMARTBIND_POA_TRANSPARENT

  When SmartBindPolicy is set to `QosExt::SMARTBIND_POA_TRANSPARENT`, all co-located invocations (i.e. between VisiBroker clients and servants in the same address space) are optimized. Using this policy value all POA policies and states applicable to that CORBA Server are honored.

- QosExt::SMARTBIND_CACHED

  When SmartBindPolicy is set to `QosExt::SMARTBIND_CACHED`, all co-located invocations (i.e. between VisiBroker clients and servants in the same address space) are optimized. Using this policy value the servant pointer is cached during the initial invocation to the CORBA object. Subsequent requests to this server will use this cached pointer, thereby ignoring all POA policies and POA states. This policy value provides the highest level of optimization.

  This cached pointer to the servant can be updated by calling _bind. This may be useful in cases where the servant goes away and the client needs to update its cached pointer to a new instance of the servant. In that case, the client application can catch the generated CORBA exception and call _bind again to update the cached pointer.

  If the POA that the servant is activated with was created with a value other than USE_ACTIVE_OBJECT_MAP_ONLY for the RequestProcessingPolicy, the SMARTBIND_CACHE behavior will revert to `QosExt::SMARTBIND_POA_TRANSPARENT`.

The default value for this policy is `QosExt::SMARTBIND_CACHED`. This policy applies to both synchronous and asynchronous invocations.

This policy is only applicable as a client-side override.

IDL definition

```
module QoSExt {
...
const CORBA::PolicyType SMART_BIND_POLICY_TYPE = 0x48454900;
/* This policy is an extension to the bind_options for the policy
framework. It allows to switch off smart binding on a per client
basis. */
typedef unsigned short SmartBindPolicyValue;
const SmartBindPolicyValue SMARTBIND_OFF = 0;
const SmartBindPolicyValue SMARTBIND_FULLY_TRANSPARENT = 1;
const SmartBindPolicyValue SMARTBIND_POA_TRANSPARENT = 2;
const SmartBindPolicyValue SMARTBIND_CACHED = 3;

interface SmartBindPolicy : CORBA::Policy {
/**
  Returns the current setting of the SmartBindPolicy @returns
  the setting
```

```
**/
readonly attribute SmartBindPolicyValue value;
};
};
```

IOP and IIOP Interfaces and Classes

This chapter describes the VisiBroker implementation of the key General Inter-ORB Protocol interfaces and other structures defined by the CORBA specification. For a complete description of these interfaces, refer to Chapter 15 of the OMG CORBA/IIOP Specification (Version2.3).

GIOP::MessageHeader

`struct` **`MessageHeader`**

This structure is used to represent information about a GIOP message.

MessageHeader members

`CORBA::Char` **`magic`**`[4];`

This string should always contain "GIOP".

`Version` **`GIOP_version`**`;`

Indicates the version of the protocol being used. This structure contains a major and minor version number, as shown. The major version should be set to 1 and the minor version should be set to 2, unless it is an older version, e.g., VisiBroker 3.x, in which case the minor version should be set to 0.

```
struct Version {
  CORBA::Octet major;
  CORBA::Octet minor;
};
```

`CORBA::Boolean` **`byte_order`**`;`

Set to `TRUE` to indicate that little-endian byte ordering is used in the message. If set to `FALSE`, big-endian byte ordering is used in the message.

`CORBA::Octet` **`message_type`**`;`

Indicates the type of message that follows the header. This should be one of the following values.

```
enum MsgType {
  Request,
  Reply,
  CancelRequest,
  LocateRequest,
  LocateReply,
  CloseConnection,
  MessageError,
  Fragment
};
```

`CORBA::ULong` **`message_size`**`;`

Indicates the length of the message that follows this header.

GIOP::CancelRequestHeader

`struct` **`CancelRequestHeader`**

This structure is used to represent information about a cancel request message header.

CancelRequestHeader members

`CORBA::ULong` **`request_id`**;

This data member represents the request identifier that is being cancelled.

GIOP::LocateReplyHeader

`struct` **`LocateReplyHeader`**

This structure is used to represent a message that is sent in reply to a locate request message. Additional data follows this header if the locate_status is set to `OBJECT_FORWARD`.

LocateReplyHeader members

`CORBA::ULong` **`request_id`**;

The request identifier of the original request.

`LocateStatusType` **`locate_status`**;

GIOP::LocateRequestHeader

`structure` **`LocateRequestHeader`**

This structure represents a message containing a request to locate an object.

LocateRequestHeader members

`CORBA::ULong` **`request_id`**;

Represents the request identifier for this message and is used to distinguish between multiple outstanding messages.

`GIOP::TargetAddress` **`target`**;

Represents the object to be located. The target is a union of three different things: object key, profile, IOR.

GIOP::ReplyHeader

```
struct ReplyHeader {};
```

This structure represents the reply header of a reply message that is sent to a client in response to a request message.

Include file

The **giop_c.hh** file should be included when you use this structure. This file is already included in corba.h in the installation/include directory.

ReplyHeader members

```
CORBA::ULong request_id;
```

Should be set to the same `request_id` as the request message for which this reply is associated.

```
ReplyStatusType reply_status;
```

Indicates the status of the reply and should be set to one of the following enum values:

- NO_EXCEPTION
- USER_EXCEPTION
- SYSTEM_EXCEPTION
- LOCATION_FORWARD
- LOCATION_FORWARD_PERM
- NEEDS_ADDRESSING_MODE

```
IOP::ServiceContextList service_info;
```

A list of service context information that may be passed from the server to the client.

GIOP::RequestHeader

```
struct RequestHeader {};
```

This structure represents the request header of a request message that is sent to an object implementation.

Include file

The **giop_c.hh** file should be included when you use this structure.

RequestHeader members

```
CORBA::ULong request_id;
```

A unique identifier used to associate a reply message with a particular request message.

`CORBA::Boolean` **`response_expected`**;

Set to FALSE if the request is a oneway operation for which a reply is not expected. Set to TRUE for operation requests and other requests that expect a reply.

GIOP::TargetAddress **_target**;

Represents the object that is the target of the request. The target is a union of the following three things: object key, profile, and IOR. Object keys are stored in a vendor-specific format and are generated when an `IOR` is created.

`CORBA::String_var` **`oper`**;

Identifies the operation being requested on the target object. This member is the same as the `operator` member, except that it is a managed type.

const char ***operation**;

Identifies the operation being requested on the target object. This member is the same as the `oper` member, except that it is not a managed type.

IOP::ServiceContextList **service_context**;

A list of service context information that may be passed from the client to the server.

IIOP::ProfileBody

`struct` **`ProfileBody`**;

This structure contains information about the protocol supported by an object.

```
module IIOP {
...
  struct ProfileBody {
    Version iiop_version;
    string host;
    unsigned short port;
    sequence<octet> object_key;
    sequence<IOP::taggedCoomponent> components;
}
...
```

ProfileBody members

`Version` **`iiop_version`**;

Represents the version of IIOP supported.

`CORBA::String_var` **`host`**;

Represents the name of the host where the server hosting the object is running.

`CORBA::UShort` **`port`**;

Indicates the port number to use for establishing a connection to the server hosting the object.

`CORBA::OctetSequence` **`object_key`**;

Object keys are stored in a vendor-specific format and are generated when an `IOR` is created.

`IIOP::MultiComponentProfile` **`components`**;

A sequence of `TaggedComponents` which contain which contain information about the protocols that are supported.

`struct` **`IOR`** {}

This structure represents an Interoperable Object Reference and is used to provide important information about object references. Your client application can create a stringified IOR by invoking the `ORB::object_to_string`method described in "char *object_to_string(CORBA::Object_ptr) = 0;" on page 5-20.

Include file

The **giop_c.hh** file should be included when you use this structure.

IOR members

`CORBA::String_var type_id;`

This data member describes the type of object reference that is represented by this IOR.

`TaggedProfileSequence` **`profiles`**;

This data member represents a sequence of one or more `TaggedProfile` structures, which contain information about the protocols that are supported.

`static CORBA::Boolean` **`is_nil`**`(IOP::IOR` **`*i`**`);`

This method returns `TRUE` if the specified pointer is `NULL`.

IOP::TaggedProfile

`struct` **`TaggedProfile`**

This structure represents a particular protocol that is supported by an Interoperable Object Reference (IOR).

TaggedProfile members

`ProfileID` **`tag`**;

This data member represents the contents of the profile data and should be one of the following values.

Value	Description
`TAG_INTERNET_IOP`	Indicates the protocol is standard IIOP.
`TAG_MUTIPLE_COMPONENTS`	Indicates the profile data contains a list of ORB services available using the protocol.

Value	Description
TAG_VSGN_LOCATOR	Indicates that the IOR is an interim, pseudo-object that is used until the real IOR is received by the osagent.
TAG_LOCAL_IPC_IOP	Indicates the protocol is IOP over a local IPC mechanism.

CORBA_OctetSequence **profile_data**;

This data member encapsulates all the protocol information needed to invoke an operation on an IOR.

Marshal Buffer Interfaces and Classes

This chapter describes the buffer class used for marshalling data to a buffer when creating an operation request or a reply message. It also describes the buffer class used for extracting data from a received operation request or reply message.

CORBA::MarshalInBuffer

`class` **`CORBA::MarshalInBuffer`** `: public VISistream`

This class represents a stream buffer that allows IDL types to be read from a buffer and may be used by interceptor methods that your implement. See "Portable Interceptor Interfaces and Classes for C++" for more information on the interceptor interfaces.

The `CORBA::MarshalInBuffer` class is used on the client side to extract the data associated with a reply message. It is used on the server side to extract the data associated with an operation request. This class provides a wide range of methods for retrieving various types of data from the buffer.

This class provides several static methods for testing and manipulating `CORBA::MarshalInBuffer` pointers.

A `CORBA::MarshalInBuffer_var` class is also offered, which provides a wrapper that automatically manages the contained object.

Include file

The **mbuf.h** file should be included when you use this class. This file gets included in **corba.h**. So, you don't have to separately include **mbuf.h**.

CORBA::MarshalInBuffer constructor/ destructor

```
CORBA::MarshalInBuffer(char *read_buffer, CORBA::ULong
  length, CORBA::Boolean release_flag=0, CORBA::Boolean
  byte_order = CORBA::ByteOrder);
```

This is the default constructor.

Parameter	Description
`read_buffer`	The buffer where the marshalled data will actually be stored.
`length`	The maximum number of bytes that may be stored in read_buffer.
`release_flag`	If set to TRUE, the memory associated with read_buffer will be freed when this object is destroyed. The default value is FALSE.
`byte_order`	Set this to TRUE to indicate that little-endian byte ordering is being used. Set to FALSE to indicate that big-endian byte ordering is being used.

```
virtual ~CORBA::MarshalInBuffer();
```

This is the default destructor. The buffer memory associated with this object will be released if the `release_flag` is set to `TRUE`. The `release_flag` may be set when the object is created or by invoking the `release_flag` method, described in "void release_flag(CORBA::Boolean val);".

CORBA::MarshalInBuffer methods

```
char *buffer() const;
```

Returns a pointer to the buffer associated with this object.

```
void byte_order(CORBA::Boolean val) const;
```

Sets the byte ordering for this message buffer.

Parameter	Description
`val`	Set this to TRUE to indicate that little-endian byte ordering is being used. Set to FALSE to indicate that big-endian byte ordering is being used.

```
CORBA::Boolean byte_order() const;
```

Returns `TRUE` if the buffer is using little-endian byte ordering. `FALSE` is returned if big-endian byte ordering is being used.

```
CORBA::ULong curoff() const;
```

Returns the current offset within the buffer associated with this object.

```
virtual VISistream& get(char& data); virtual
  VISistream& get(unsigned char& data);
```

These methods allow you to retrieve a single character from the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

Parameter	Description
`data`	The location where the retrieved char or unsigned char is to be stored.

```
virtual VISistream& get(<data_type> data, unsigned
  size);
```

These methods allow you to retrieve a sequence of data from the buffer at the current location. There is a separate method for each of the listed target data types.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

Parameter	Description
data	The location where the retrieved data is to be stored. The supported target data types are: char* unsigned char* short* unsigned short* int* unsigned int* long* unsigned long* float* double* long double* VISLongLong* VISULongLong* wchar_t*
size	The number of the specified data types to be retrieved.

```
virtual VISistream& getCString(char* data, unsigned
  maxlen);
```

This method allows you to retrieve a character string from the buffer at the current location. It returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

Parameter	Description
data	The location where the retrieved character string is to be stored.
maxlen	The maximum number of characters to be retrieved.

```
virtual const CORBA::WChar *getWString(CORBA::ULong&
  len);
```

This method returns a pointer to a location within the buffer containing a widecharacter string. Wide characters in VisiBroker are two bytes wide.

Parameter	Description
len	The offset of the desired data within the buffer.

```
virtual int is_available(unsigned long size);
```

Returns 1 if the specified `size` is less than or equal to the size of the buffer associated with this object.

Parameter	Description
size	Number of bytes that need to fit within this buffer.

```
virtual CORBA::ULong length() const;
```

Returns the total number of bytes in this object's buffer.

```
virtual void new_encapsulation() const;
```

Resets the starting offset within the buffer to 0.

```
void release_flag(CORBA::Boolean val);
```

Enables or disables the automatic freeing of buffer memory when this object is destroyed.

Parameter	Description
val	If val is set to TRUE, the buffer memory for this object will be freed when this object is destroyed. If val is set to FALSE, the buffer will not be freed when this object is destroyed

```
CORBA::Boolean release_flag() const;
```

Returns TRUE if the automatic freeing of this object's buffer memory is enabled, otherwise FALSE is returned.

```
void reset();
```

Resets the starting offset, current offset and seek position to zero.

```
void rewind();
```

Resets the seek position to 0.

```
CORBA::ULong seekpos(CORBA::ULong pos);
```

Sets the current offset to the value contained in pos. If pos specifies an offset that is greater than the size of the buffer, a CORBA::BAD_PARAM exception is raised.

```
static CORBA::MarshalInBuffer
  *_duplicate(CORBA::MarshalInBuffer_ptr ptr);
```

Returns a duplicate pointer to this object pointed to by ptr and increments this object's reference count.

```
static CORBA::MarshalInBuffer *_nil();
```

Returns a NULL pointer of type CORBA::MarshalInBuffer.

```
static void _release(CORBA::MarshalInBuffer_ptr ptr);
```

Reduces the reference count of the object pointed to by ptr. If the reference count is then 0, the object is destroyed. If the object's release_flag was set to true when it was constructed, the buffer associated with the object will freed.

CORBA::MarshalInBuffer operators

```
virtual VISistream& operator>>(<data_type> data);
```

This stream operator allows you to add data of the specified source data_type to the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just written.

Parameter	Description
`data`	The data to be written to the buffer. The supported source data types are: char*& char& unsigned char& short& unsigned short& int& unsigned int& long& unsigned long& float& double& long double& wchar_t*& wchar_t&

CORBA::MarshalOutBuffer

`class` **`CORBA::MarshalOutBuffer`** `: public VISostream`

This class represents a stream buffer that allows IDL types to be written to a buffer and may be used by interceptor methods that you implement. See "Portable Interceptor Interfaces and Classes for C++" for more information on the interceptor interfaces.

The `CORBA::MarshalOutBuffer` class is used on the client side to marshal the data associated with an operation request. It is used on the server side to marshal the data associated with a reply message. This class provides a wide range of methods for adding various types of data to the buffer or for retrieving what was written from the buffer.

This class provides several static methods for testing and manipulating `CORBA::MarshalOutBuffer` pointers.

A `CORBA::MarshalOutBuffer_var` class is also offered, which provides a wrapper that automatically manages the contained object.

Include file

The **mbuf.h** file should be included when you use this class. This file gets included in **corba.h**. So, you don't have to separately include **mbuf.h**.

CORBA::MarshalOutBuffer constructors

```
CORBA::MarshalOutBuffer(CORBA::ULong initial_size =
  255,CORBA::Boolean release_flag = 0);
```

Creates a marshalOutBuffer of size `initial-size`. The MarshalOutBuffers are capable of resizing themselves during a put operation. The size doubles ensuring each resize operation.

Parameter	Description
`initial_size`	The initial size of the buffer associated with this object. The default size is 255 bytes.
`release_flag`	If set to TRUE, the memory associated with `read_buffer` will be freed when this object is destroyed. The default value is FALSE.

```
CORBA::MarshalOutBuffer(char *read_buffer, CORBA::ULong
  len, CORBA::Boolean release_flag=0);
```

Creates an object with the specified buffer, buffer length and release flag value.

Parameter	Description
`read_buffer`	The buffer where the marshalled data will actually be stored.
`length`	The maximum number of bytes that may be stored in `read_buffer`.
`release_flag`	If set to TRUE, the memory associated with `read_buffer` will be freed when this object is destroyed. The default value is FALSE.

CORBA::MarshalOutBuffer destructor

```
virtual ~CORBA::MarshalOutBuffer();
```

This is the default destructor. The buffer memory associated with this object will be released if the `release_flag` is set to `TRUE`. The `release_flag` may be set when the object is created or by invoking the `release_flag` method, described on “CORBA::Boolean release_flag() const;”.

CORBA::MarshalOutBuffer methods

```
char *buffer() const;
```

Returns a pointer to the buffer associated with this object.

```
CORBA::ULong curoff() const;
```

Returns the current offset within the buffer associated with this object.

```
virtual CORBA::ULong length() const;
```

Returns the total number of bytes in this object’s buffer.

```
virtual void new_encapsulation() const;
```

Resets the starting offset within the buffer to 0.

```
virtual VISostream& put(char data);
```

Adds a single character to the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just added.

Parameter	Description
data	The char to be stored.

```
virtual VISostream& put(const <data_type> data,
  unsigned size);
```

These methods allow you to store a sequence of data in the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just added.

Parameter	Description
data	The data to be stored. The supported source data types are: char* unsigned char* short* unsigned short* int* unsigned int* long* unsigned long* float* double* long double* VISLongLong* VISULongLong* wchar_t*
size	The number of the specified data types to be stored.

```
virtual VISostream& putCString(const char* data);
```

This methods allows you to store a character string into the buffer at the current location. It returns a pointer to the location within the buffer immediately following the end of the data that was just added.

Parameter	Description
data	The character string to be stored.

```
void release_flag(CORBA::Boolean val);
```

Enables or disables the automatic freeing of buffer memory when this object is destroyed.

Parameter	Description
val	If val is set to TRUE, the buffer memory for this object will be freed when this object is destroyed. If val is set to FALSE, the buffer will not be freed when this object is destroyed.

```
CORBA::Boolean release_flag() const;
```

Returns TRUE if the automatic freeing of this object's buffer memory is enabled, otherwise FALSE is returned.

```
void reset();
```

Resets the starting offset, current offset and seek position to zero.

```
void rewind();
```

Resets the seek position to 0.

```
CORBA::ULong seekpos(CORBA::ULong pos);
```

Sets the current offset to the value contained in `pos`. If `pos` specifies an offset that is greater than the size of the buffer, a `CORBA::BAD_PARAM` exception is raised.

```
static CORBA::MarshalOutBuffer
  *_duplicate(CORBA::MarshalOutBuffer_ptr ptr);
```

Returns a duplicate pointer to this object pointed to by `ptr` and increments this object's reference count.

```
static CORBA::MarshalOutBuffer *_nil();
```

Returns a `NULL` pointer of type `CORBA::MarshalOutBuffer`.

```
static void _release(CORBA::MarshalOutBuffer_ptr ptr);
```

Reduces the reference count of the object pointed to by `ptr`. If the reference count is then 0, the object is destroyed. If the object's `release_flag` was set to true when it was constructed, the buffer associated with the object will be freed.

CORBA::MarshalOutBuffer operators

```
virtual VISostream& operator<<(<data_type> data);
```

This stream operator allows you to add data of the specified `data_type` to the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just written.

Parameter	Description
`data`	The data to be obtained to the buffer. The supported data types are: const char* char unsigned char short unsigned short int unsigned int long unsigned long float double long double VISLongLong VISULongLong wchar_t* wchar_t

Location Service Interfaces and Classes

This chapter describes the interfaces you can use to locate object instances on a network of Smart Agents. For more information on the Location Service, see Chapter 14, "Using the Location Service" in the VisiBroker-RT for C++ Programmer's Guide.

Note

The libraries libagentsupport.o and liblocsupport.o are required when building a VisiBroker-RT 6.0 application to support use of the VisiBroker Location Service. For a description of all the libraries provided by the VisiBroker-RT for C++ product please refer to "Step 5: Selecting VisiBroker Libraries" in the *VisiBroker-RT for C++ Programmer's Guide*.

Agent

`class` **`Agent`** `: public CORBA::Object`

This class provides methods that enable you to locate all instances of a particular object on a network of Smart Agents. The methods offered by this class are divided into two categories; those that query a Smart Agent for data about objects and those that deal with *triggers*.

Your client application can obtain object information based on an interface repository ID alone or in combination with an instance name.

Triggers allow your client application to be notified of changes in the availability of one or more object instances.

Command-line options for applications using the Location Service are described in "Location service options" on page A-6.

```
interface Agent {
  HostnameSeq all_agent_locations()
    raises (Fail);
  RepositoryIdSeq all_repository_ids()
    raises (Fail);
  ObjSeqSeq all_available()
    raises (Fail);
  ObjSeq all_instances (in string repository_id)
    raises (Fail);
  ObjSeq all_replica (in string repository_id, in string
        instance_name)
    raises (Fail);
  DescSeqSeq all_available_descs()
    raises (Fail);
  DescSeq all_instances_descs (in string repository_id)
    raises (Fail);
  DescSeq all_replica_descs (in string repository_id,
    in string instance_name)
    raises (Fail);
  void reg_trigger(in TriggerDesc desc, in TriggerHandler
        handler)
    raises (Fail);
  void unreg_trigger(in TriggerDesc desc, in TriggerHandler
        handler)
    raises (Fail);
  attribute boolean willRefreshOADs;
};
```

Include file

You should include the **locate_c.hh** file when you use this class.

Agent methods

`ObjLocation::HostnameSeq_ptr` **`all_agent_locations`**`();`

Returns a sequence of host names representing the hosts on which osagent processes are currently executing.

See also "<type>Seq".

This method throws the following exceptions:

Exception	Description
`Fail`	The `FailReason` values that may be presented include: NO_AGENT_AVAILABLE NO_SUCH_TRIGGER AGENT_ERROR See "Fail" on page 19-7 for a discussion of the `Fail` class.

`ObjLocation::ObjSeq_ptr` **`all_available`**`();`

Returns a sequence of object references for all objects currently registered with some Smart Agent on the network.

See also "<type>Seq".

This method throws the following exceptions:

Exception	Description
`Fail`	The `FailReason` values that may be presented include: NO_AGENT_AVAILABLE NO_SUCH_TRIGGER AGENT_ERROR See "Fail" for a discussion of the `Fail` class.

`ObjLocation::DescSeqSeq_ptr` **`all_available_desc`**`();`

Returns descriptions for all objects currently registered with a Smart Agent on the network. The description information returned is organized by repository id.

See also "<type>SeqSeq".

This method throws the following exceptions:

Exception	Description
`Fail`	The `FailReason` values that may be presented include: NO_AGENT_AVAILABLE NO_SUCH_TRIGGER AGENT_ERROR See "Fail" on page 19-7 for a discussion of the `Fail` class.

```
ObjLocation::ObjSeq_ptr all_instances(const char
  *repository_id);
```

Returns a sequence of object references to all instances with the specified `repository_id`.

See also "<type>Seq".

Parameter	Description
`repository_id`	The repository ID of the object references to be retrieved.

This method throws the following exceptions:

Exception	Description
`Fail`	Any of the `FailReason` values, other than NO_SUCH_TRIGGER, may be presented. See "Fail" for a discussion of the `Fail` class.

```
ObjLocation::DescSeq_ptr all_instances_descs(const char
  *repository_id);
```

Returns description information for all object instances with the specified `repository_id`.

See also "<type>Seq".

Parameter	Description
`repository_id`	The repository ID of the object descriptions to be retrieved.

This method throws the following exceptions:

Exception	Description
`Fail`	Any of the `FailReason` values, other than NO_SUCH_TRIGGER, may be presented. See "Fail" for a discussion of the `Fail` class.

```
ObjLocation::ObjSeq_ptr all_replica(const char
  *repository_id, const char *instance_name);
```

Returns a sequence of object references for objects with the specified `repository_id` and `instance_name`.

See also "<type>Seq".

Parameter	Description
`repository_id`	The repository ID of the object references to be retrieved.
`instance_name`	The instance name of the object references to be returned.

This method throws the following exceptions:

Exception	Description
`Fail`	Any of the `FailReason` values, other than `NO_SUCH_TRIGGER`, may be presented. See "Fail" for a discussion of the `Fail` class.

```
ObjLocation::DescSeq_ptr all_replica_descs(const char
  *repository_id, const char *instance_name);
```

Returns a sequence of description information for all object instances with the specified `repository_id` and `instance_name`.

See also "<type>Seq".

Parameter	Description
`repository_id`	The repository ID of the object descriptions to be retrieved.
`instance_name`	The instance name of the object descriptions to be retrieved.

This method throws the following exceptions:

Exception	Description
`Fail`	Any of the `FailReason` values, other than `NO_SUCH_TRIGGER`, may be presented. See "Fail" for a discussion of the `Fail` class.

```
void reg_trigger(const ObjLocation::TriggerDesc& desc,
  ObjLocation::TriggerHandler_ptr hdlr);
```

Registers the trigger handler `hdlr` for object instances that match the description information specified in `desc`.

Note

A `TriggerHandler` will be invoked every time an object that satisfies the trigger's description becomes available. If you are only interested in learning when the first instance of the object becomes available, you should use the `unreg_trigger` method to remove the trigger after the first notification is received.

Parameter	Description
`desc`	The object instance description information, which can contain combinations of the following information: repository ID instance name hostname You can provide more or less information to narrow or widen the object instances to be monitored.
`hdlr`	The trigger handler object being registered.

This method throws the following exceptions:

Exception	Description
`Fail`	Any of the `FailReason` values, other than `NO_SUCH_TRIGGER`, may be presented. See "Fail" for a discussion of the `Fail` class.

```
void unreg_trigger(const ObjLocation::TriggerDesc&
  desc, ObjLocation::TriggerHandler_ptr hdlr);
```

Unregisters the trigger handler `hdlr` for object instances that match the description information specified in `desc`.

Parameter	Description
`desc`	The object description information.
`hdlr`	The trigger handler object being unregistered.

This method throws the following exceptions:

Exception	Description
`Fail`	The only FailReason value possible is `NO_SUCH_TRIGGER`. See "Fail" for a discussion of the `Fail` class.

```
CORBA::Boolean willRefreshOADs();
```

Returns `TRUE` if the set of Object Activation Daemon is updated each time a method offered by this class is invoked, otherwise `FALSE` is returned. If the cache is not refreshed on each invocation, the following conditions may occur:

- All objects will still be reported, but their descriptor's `activable` flag may be incorrect.
- Any attempt to verify the existence of an object registered with an OAD that has been started since the last refresh of the OAD cache will cause those objects to be activated by the OAD.

```
void willRefreshOADs(CORBA::Boolean val);
```

This class maintains a set of Object Activation Daemons. This method enables or disables the automatic refreshing of the OADs contained in this set.

Parameter	Description
`val`	If `TRUE`, the OAD set will be refreshed whenever a method offered by this class is invoked.

Desc

```
struct Desc;
```

This structure contains information you use to describe the characteristics of an object. You pass this structure as an argument to several of the Location Service methods described in the chapter. The *Desc* structure, or a sequence of them, is returned by some of the Location Service methods.

See also "<type>Seq".

```
module ObjLocation {
struct Desc {
  Object ref;
  IIOP::ProfileBody iiop_locator;
  string repository_id;
  string instance_name;
  boolean activable;
  string agent_hostname;
};
...
```

Desc members

`Object` **`ref`**;

A reference to the object being described.

`IIOP::ProfileBody` **`iiop_locator`**;

Represents profile data for the object, described in “IIOP::ProfileBody”.

`CORBA::String_var` **`repository_id`**;

The object’s repository identifier.

Fail

`CORBA::String_var` **`instance_name`**;

The object’s instance name.

`CORBA::Boolean` **`activable`**;

Set to `TRUE` to indicate that this object is registered with the Object Activation Daemon. It is set to `FALSE` to indicate that the object was started manually and is registered with the osagent.

`CORBA::String_var` **`agent_hostname`**;

The name of the host running the Smart Agent with which this object is registered.

`class` **`Fail`** `: public CORBA::UserException`

This exception class may be thrown by the `Agent` class to indicate various errors. The data member `FailReason` is used to indicate the nature of the failure.

Fail members

`FailReason` **`reason`**;

Set to one of the following values to indicate the nature of the failure:

```
enum FailReason {
  NO_AGENT_AVAILABLE,
  INVALID_REPOSITORY_ID,
  INVALID_OBJECT_NAME,
  NO_SUCH_TRIGGER,
  AGENT_ERROR
};
```

TriggerDesc

`struct` **`TriggerDesc`**;

This structure contains information you use to describe the characteristics of one or more objects for which you wish to register a `TriggerHandler`,

described in "TriggerHandler". The `host_name` and `instance_name` members may be set to `NULL` to monitor the widest possible set of objects. The more information that is specified, the smaller the set of objects will be.

```
module ObjLocation {
   ...
     struct TriggerDesc {
        string repository_id;
        string instance_name;
        string host_name;
}; ...
```

TriggerDesc members

`CORBA::String_var` **`repository_id`**;

Represents the repository identifiers of the objects to be monitored by the `TriggerHandler`. May be set to NULL to include all possible repository identifiers.

`CORBA::String_var` **`instance_name`**;

Represents the instance name of the object to be monitored by the `TriggerHandler`. May be set to NULL to include all possible instance names.

`CORBA::String_var` **`host_name`**;

Represents the host name where the object or objects to be monitored by the `TriggerHandler` are located. May be set to NULL to include all hosts in the network.

TriggerHandler

You use this base class to derive your own callback object to be invoked every time an object becomes available or unavailable. You specify the criteria for the object or objects in which you are interested. You register your `TriggerHandler` object using the `Agent::reg_trigger` method, described in "void reg_trigger(const ObjLocation::TriggerDesc& desc, ObjLocation::TriggerHandler_ptr hdlr);".

You must provide implementations for the `impl_is_ready` and `impl_is_down` methods.

```
interface TriggerHandler {
  void impl_is_ready(in Desc desc);
  void impl_is_down(in Desc desc);
};
```

Include file

You should include the **locate_c.hh** file when you use this class.

TriggerHandler methods

`virtual void` **`impl_is_ready`**`(const Desc&` **`desc`**`);`

This method is invoked by the Location Service when an object instance matching the criteria specified in `desc` becomes accessible.

Parameter	Description
`desc`	The object description information.

`virtual void` **`impl_is_down`**`(const Desc&` **`desc`**`);`

This method is invoked by the Location Service when an object instance matching the criteria specified in `desc` is no longer accessible.

Parameter	Description
`desc`	The object description information.

<type>Seq

This is a generalized class description for the following sequence classes used by the Location Service:

Class	Description
`DescSeq`	A sequence of `Desc` structures.
`HostnameSeq`	A sequence of host names.
`ObjSeq`	A sequence of object references.
`RepositoryIdSeq`	A sequence of repository identifiers.

Each class represents a particular sequence of `<type>`. The Location Service returns lists of information to your client application in the form of sequences which are mapped to one of these classes.

Each class offers operators for indexing items in the sequence just as you would a C++ array. The also offer methods for obtaining the length of the array, and for setting the array length.

Example 61 shows the correct way to index a `HostnameSeq` returned from the `Agent::all_agent_locations` method.

Example 61 Indexing a HostnameSeq_var class

```
...
ObjLocation::HostnameSeq_var hostnames(myAgent->
         all_agent_locations());

for (CORBA::ULong i=0; i < hostnames->length(); i++) {
  cout << "Agent host #" << i+1 << ": " << hostnames[i] << endl;
}
...
```

See also "<type>SeqSeq".

<type>Seq methods

`<type>&` **`operator[]`**`(CORBA::ULong` **`index`**`) const;`

Returns a reference to the element in the sequence identified by `index`.

Caution

You must use a CORBA::ULong type for the index. Using an int type may lead to unpredictable results.

Parameter	Description
`index`	The index of the element to be returned. This index is zero-based.

This method throws the following exceptions:

Exception	Description
`CORBA::BAD_PARAM`	The index specified is less than zero or greater than the size of the sequence.

```
CORBA::ULong length() const;
```

Returns the number of elements in the sequence.

```
void length(CORBA::ULong len);
```

Sets the maximum length of the sequence to the value contained in `len`.

Parameter	Description
`len`	The new length for the sequence.

<type>SeqSeq

This is a generalized class description for the following classes used by the Location Service:

Class	Description
`DescSeqSeq`	A sequence of `DescSeq` objects.
`ObjSeqSeq`	A sequence of `ObjSeq` objects.

Each class represents a particular sequence of `<type>Seq`. Some Location Service methods return lists of information to your client application in the form of sequences of sequences which are mapped to one of these classes.

Each class offers operators for indexing items in the sequence just as you would a C++ array. The class also offer methods for obtaining the length of the array, and for setting the array length.

See also “<type>Seq”.

<type>SeqSeq methods

```
<type>Seq& operator[](CORBA::ULong index) const;
```

Returns a reference to the element in the sequence identified by `index`. The reference is to a one dimensional sequence, described in “<type>Seq”.

Caution

You must use a CORBA::ULong type for the index. Using an int type may lead to unpredictable results.

Parameter	Description
`index`	The index of the element to be returned. This index is zero-based

This method throws the following exceptions:

Exception	Description
`CORBA::BAD_PARAM`	The index specified is less than zero or greater than the size of the sequence.

```
CORBA::ULong length() const;
```

Returns the number of elements in the sequence.

```
void length(CORBA::ULong len);
```

Sets the maximum length of the sequence to the value contained in `len`.

Parameter	Description
`len`	The new length for the sequence.

Initialization Interfaces and Classes

This chapter describes the interfaces and classes that are provided for statically initializing ORB services, such as interceptors, fatal handlers, and new handlers.

VISInit

`class` **`VISInit`**

This abstract base class provides for the static initialization of service classes before and after the ORB and has been initialized. By deriving your service class from `VISInit` and declaring it statically, you ensure that your service class instance will be properly initialized.

The ORB will invoke the `VISInit::ORB_init` and `VISInit::ORB_initialized` whenever the application calls the `CORBA::ORB_init` method. By providing your own implementations of these methods, you may add any needed initialization that must be performed for your service.

In typical C++ fashion, you can provide your own `new_handler()` function using the `set_new_handler()` method. If you have installed a `new_handler()` function, then `VISInit::ORB_init` does not install its own; however, if no `new_handler()` function is installed, `VISInit::ORB_init` installs the ORB specific `new_handler()` function.

Include file

The **vinit.h** file should be included when you use this class.

VISInit constructors/destructors

`VISInit``();`

This is the default constructor.

`VISInit``(CORBA::Long` **`init_priority`**`);`

This constructor creates a `VISInit`-derived object with the specified priority, which determines when it will be initialized relative to other `VISInit`-derived objects.

Internal VisiBroker-RT for C++ classes which need to be initialized before user-defined classes have a negative priority value. The lowest priority value currently used by VisiBroker internal classes is –10.

Note

You should set a priority value less than –10 and no lower than -50 if your class must be initialized before the VisiBroker-RT for C++ internal classes. Setting a priority of less than -50 can produce undefined behavior.

If no priority value is specified, the default value is 0, which means that the class will be initialized after all of the internal VisiBroker-RT for C++ classes.

Parameter	Description
init_priority	The initialization priority for this object. A negative priority value will cause this class to be initialized earlier. A positive priority value will cause this class to be initialized later.

```
virtual ~VISInit();
```

This is the default destructor.

VISInit methods

```
virtual void ORB_init(int& argc, char * const *argv,
  CORBA::ORB_ptr orb);
```

This method will be called during ORB initialization after the command line arguments have been parsed.

Parameter	Description
argc	The count of arguments.
argv	An array of argument pointers.
orb	The ORB being initialized.

```
virtual void ORB_initialized(CORBA::ORB_ptr orb);
```

This method will be called after the ORB is initialized.

Parameter	Description
orb	The ORB being initialized.

Since the BOA has been replaced by the POA this method is being deprecated in VisiBroker 4.0. It is supported only for backwards compatibility.

```
virtual void BOA_init(int& argc, char * const *argv,
  CORBA::BOA_ptr boa);
```

This method will be called when the BOA is initialized. Your implementation should provide for the initialization of the server-side interceptor factory that you wish to use.

Parameter	Description
argc	The count of arguments.
argv	An array of argument pointers.
boa	The BOA being initialized.

VISUtil

```
virtual void ORB_shutdown()
```

This method will be called when the ORB is shutdown.

`class` **`VISUtil`**

This base class provides several general purpose services that application programmers may find useful.

Include file

The **vutil.h** file should be included when you use this class.

VISUtil methods

```
static void set_user_fatal(void
  (*new_user_fatal(void));
```

This method can be called at any time to install a user defined fatal error handler. The user defined fatal error handler will be called when the ORB's `vis_fatal()` method is invoked.

Parameter	Description
`*new_user_fatal(void)`	A pointer to the user defined fatal error handler method.

Appendix: Using Command-Line Options

This appendix describes the options that may be set for the Basic Object Adaptor, the Object Request Broker, and the Location Service.

BOA_init() method (deprecated since VisiBrokerRT 4.0)

The `BOA_init()` method is used by your object implementation to set such options as the desired thread policy or the TCP/IP port number to be used. ese parameters are passed as arguments to the task calling `BOA_init()` when it is started.

Tornado

```
-> start_corba "-OAipAddr 199.99.129.33 -OAport 19000"
```

Example 62 shows the definition of the `BOA_init()` method and the arguments it accepts. Like the `ORB_init()` method, the `argc` and `argv` parameters passed to `BOA_init()` are the same arguments that were passed to your object implementation's main routine. The `BOA_init()` method will ignore any arguments it does not recognize.

Example 62 BOA_init() method

```
class CORBA {
  ...
  static BOA_ptr BOA_init(int& argc, char *const *argv,
    const char *boa_identifier = "VIS_BOA");
  ...
};
```

After this method has been invoked, all the recognized BOA arguments will be stripped from the original parameter list so that they will not interfere with any other argument processing that your object implementation requires.

BOA options

The following table ummarizes the `BOA_init()` options.

Table 11 BOA_init() options used by object implementations

Type/Value pair	Purpose
`-OAagent <0 | 1>`	If set to 0, this option specifies that new instances of objects being registered will not be exported through the Smart Agent. Client attempts to locate these instances with `_bind()` will fail. The default value is 1.
`-OAconnectionMax <#>`	Specifies the maximum number of connections allowed when `-OAid TSession` is selected.
`-OAconnectionMaxIdle <#>`	Specifies the time (in seconds) which a connection can idle without any traffic. Connections that idle beyond this time can be shutdown by VisiBroker. By default, this is set to 0 seconds which means that connections will never automatically timeout. This option should be set for Internet applications.

Type/Value pair	Purpose
-OAgarbageCollectTimer <#>	Specifies the time (in seconds) that the adapter waits before checking for idle connections and threads to be cleaned up. The default period is 30 seconds. The adapter checks for threads that have been idle for longer than the time specified by -OAthreadMaxIdle and for connections that have been idle for longer than the time specified by -OAconnectionMaxIdle.
-OAid <TPool | TSession>	Specifies the thread policy for multithreaded servers to be used by the BOA. The default is TPool, except for backward compatibility where TSession is the default. If you specify a value other than TPool or TSession, a nil BOA_ptr will be returned by the BOA_init method.
-OAipAddr <hostname | ip_address>	Specifies the hostname or IP address to be used for the BOA. Use this option if your machine has multiple network interfaces and the BOA is associated with just one address. If no option is specified, the host's default address is used.
-OAlocalIPC <0 | 1>	When a client and an object implementation reside on the same host and this is set to 1, a local inter-process communication method will be used instead of a socket connection. On Windows platforms, shared memory will be used. On UNIX platforms, the UNIX Domain Protocol will be used. If set to 0, a socket connection will be used. The default is 1.
-OAport <port_number>	Specifies the port number to be used by the BOA when listening for new connections.
-OArcvbufsize <buffer_size>	Specifies the size of the buffer (in bytes) used to receive messages. If not specified, a default value (dependent upon your operating system) will be used. This argument can be used to significantly impact performance or benchmark results.
Windows -OAsendbufsize <buffer_size>	Specifies the size of the buffer (in bytes) used to send messages. If not specified, a default value (dependent upon your operating system) will be used. This argument can be used to significantly impact performance or benchmark results.
-OAshmsize <size>	Specifies the size of the send and receive segments (in bytes) in shared memory. If your client program and object implementation communicate via shared memory, you may use this option to enhance performance. This option is only supported on Windows platforms.
-OAtcpNoDelay <0 | 1>	When set to 1, it sets all sockets to immediately send requests. The default value of 0 allows sockets to send requests in batches as buffers fill. This argument can be used to significantly impact performance or benchmark results.
-OAthreadMax <#>	Specifies the maximum number of threads allowed when-OAid TPool is selected.
-OAthreadMaxIdle <#>	Specifies the time (in seconds) which a thread can exist without servicing any requests. Threads that idle beyond the time specified can be returned to the system. By default, this is set to 300 seconds.
-OAthreadStackSize <stack_size>	Specifies the maximum thread stack size (in bytes) allowed when -OAid TPool or -OAid TSession is selected

ORB_init() method

The `ORB_init()` method is used by applications to set such options as the IP address and port number of the Smart Agent to be used. These parameters may be passed as arguments when `ORB_init()` is called.

Example 63 Specifying the Server Manager Name when starting up a VisiBroker Application

```
...
  char *argv[] = {"DO_CORBA","-
  Dvbroker.orb.enableserverManager","true","-
  Dvbroker.serverManager.name","MyServerManager"};
  /*_____________________________*/
  /* Call ORB_init*/
  /*_____________________________*/
  VISTRY
  {
    // Initialize the ORB
    orb = CORBA::ORB_init(argc, argv);
...
```

Example 64 shows the definition of the `ORB_init()` method and the arguments it accepts. Like the `BOA_init()` method, the `argc` and `argv` parameters passed to `ORB_init()` are the same arguments that were passed to your client program's main routine. The `ORB_init()` method will ignore any arguments it does not recognize.

Example 64 ORB_init() method definition

```
class CORBA {
  ...
  static ORB_ptr ORB_init(int& argc, char *const *argv,
           const char *orb_id = (char *)NULL);
  ...
};
```

After this method has been invoked, all the recognized ORB arguments will be stripped from the original parameter list so that they will not interfere with any other argument processing that your client program requires.

ORB options

All but one of the ORB options take the form of type-value pairs. Table 12 summarizes the `ORB_init()` options.

Table 12 ORB_init() options

Type/Value pair	Purpose
`-ORBagent <0 | 1>`	If set to 0, this option specifies that no Smart Agent will be contacted to locate servers, and VisiBroker proprietary `_bind()` will not work. The default value is 1.
`-ORBagentAddr <hostname | ip_address>`	Specifies the hostname or IP address of the host running the Smart Agent this client should use. If a Smart Agent is not found at the specified address or if this option is not specified, broadcast messages will be used to locate a Smart Agent.
`-ORBagentPort <port_number>`	Specifies the port number of the Smart Agent. This option can be useful if multiple ORB domains are required, as described in the VisiBroker-RT for C++ *Programmer's Guide*, Chapter 13, "Using the Smart Agent." If not specified, a default port number of 14000 will be used.

Type/Value pair	Purpose
`-ORBbackCompat <0 | 1>`	If set to 1, this option specifies that backward compatibility with VisiBroker 2.0 should be provided. The default is 0.
`-ORBbackdii <0 | 1>`	If set to 1, this option specifies that support for the 1.0 IDL-to-C++ mapping should be provided. If set to 0 or not specified at all, the new 1.1 mapping will be used. The default setting is 0. If `-ORBbackCompat` is set to 1, this option will automatically be set to 1.
`-ORBconnectionMax <#>`	Specifies the maximum number of outgoing connections that are allowed. If you do not specify this option, the default is to allow an unlimited number of connections.
`-ORBconnectionMaxIdle <#>`	This specifies the number of seconds that an outgoing connection can idle before it is shutdown by VisiBroker. By default, this is set to 0, which means that connections will never timeout. This option should be set for Internet applications.
`-ORBir_name <ir_name>`	Specifies the name of the Interface Repository to be accessed when the `Object::get_interface()` method is invoked on object implementations.
`-ORBir_ior <ior_string>`	Specifies the IOR of the Interface Repository to be accessed when the `Object::get_interface()` method is invoked on object implementations.
`-ORBnullString <0 | 1>`	If set to 1, this option specifies that the ORB will allow C++ NULL strings to be streamed. The NULL strings will be marshalled as strings of length 0—as opposed to the empty string (“”) which is marshalled as a string of length 1, with the sole character of “\0”. If set to 0, attempts to marshal out a NULL string will throw `CORBA::BAD_PARAM`. Attempts to marshal in a NULL string will throw `CORBA::MARSHAL`. The default setting is 0. If `-ORBbackCompat` is set to 1, this option will automatically be set to 1.
`-ORBPropTable <table_name>`	Specifies the name of a VisiBroker Property Table containing VisiBroker properties and their assigned values.
`-ORBrcvbufsize <buffer_size>`	Specifies the size of the TCP buffer (in bytes) used to receive responses. If not specified, a default buffer size will be used. This argument can be used to significantly impact performance or benchmark results.
`-ORBsecuresetattr <0 | 1>`	If set to 0, a client can use the ORB management interface to set the server’s attributes. By default, this value is set to 1 and a CORBA::NO_PERMISSION exception is thrown if a user attempts to set a server’s attribute with the ORB management interface.
`-ORBsecureShutdown <0 | 1>`	If set to 0, a user can use the shutdown command from the ORB management interface to shutdown the server. By default, this value is set to 1, and a CORBA::NO_PERMISSION exception is thrown if a user attempts to use the shutdown command.
`-ORBsendBind <0 | 1>`	If set to 1, this option provides backward compatibility between the `bind` call and VisiBroker-RT for C++ 2.x. If you do not specify, the default value is 0.

Type/Value pair	Purpose
-ORBsendbufsize <buffer_size>	Specifies the size of the TCP buffer (in bytes) used tosend client requests. If not specified, a default buffer size will be used. This argument can be used to significantly impact performance or benchmark results.
-ORBtcpNoDelay <0 | 1>	When set to 1, it sets all sockets to immediately send requests. The default value of 0 allows sockets to send requests in batches as buffers fill. This argument can be used to significantly impact performance or benchmark results.

Location service options

These command-line options can be used by your client program to control various Location Service features. When your client application invokes the `ORB_init` method, the Location Services will be initialized and will receive any command-line arguments you have specified. Command-line options for the Location Service will be processed and stripped from the argument list. All unrecognized options will be ignored.

As with the command line options for the BOA and ORB, the Location Service options take the form of type-value pairs.

Tornado

```
prompt> start_corba "-LOCdebug 1 -LOCtimeout 10 -LOCverify 0"
```

The table below summarizes the Location Service command-line options.

Table 13 Location Service command-line options

Type/Value Pair	Purpose
-LOCdebug <0 | 1>	If set to one, enables the using the Location Service debugging output, described in Chapter 18, "Marshal buffer interfaces and classes." If this option is not specified, debugging output is disabled.
-LOCtimeout <seconds>	Indicates the number of seconds to wait for a response from a server when verifying the existence of an object. This option is only used when -LOCverify has been set to one. The default value is one second.
-LOCverify <0 | 1>	If set to 1, the Location Service will verify the existence of an object before returning an object reference to the client application. If set to 0, the Location Service will offer faster performance, but it may not return the most current information. The default value for this option is 0.

Appendix: Using VisiBroker Properties

This appendix describes the properties that can be set in VisiBroker-RT for C++.

OSAgent (Smart Agent) properties

The following table lists the VisiBroker-RT for C++ OSAgent properties.

Property	Default	Description
`vbroker.agent.addr`	*null*	The IP address or host name of the host running the OSAgent. The default *null* value instructs VisiBroker applications to use the value from the OSAGENT_ADDR environment variable. If this OSAGENT_ADDR variable is not set, then it is assumed that OSAgent is running on the local host.
`vbroker.agent.addrFile`	*null*	Specifies a file that stores the IP address(es) or host name(s) where OSAgent maybe found.
`vbroker.agent.debug`	`false`	When set to true, the system displays debugging information about VisiBroker applications communicating with the OSAgent.
`vbroker.agent.localFile`	*null*	Specifies which network interface to use on multi-home machines. This used to be the OSAGENT_LOCAL_FILE environment variable.
`vbroker.agent.enableLocator`	`true`	When set to false, this property does not allow VisiBroker applications to communicate with OSAgent.
`vbroker.agent.clientHandlerPort`	*null*	Specifies the port that the OSAgent uses to verify the existence of a client—in this case the server. The default value *null* means that the OSAgent picks a random port number.
`vbroker.agent.port`	`14000`	The port number that defines a domain within your network. All VisiBroker applications and the OSAgent works together when they have the same port number. This property is same as the OSAGENT_PORT environment variable.

ORB properties

The following table lists the VisiBroker-RT for C++ ORB properties.

Property	Default	Description
vbroker.orb.activateIOR	*null*	Allows the launched server to easily establish contact with the OAD that launched it.
vbroker.orb.oadUID	0	Used to ensure that the OAD that launched the server still exists.
vbroker.orb.propStorage	*null*	Specifies a property file that contains property values.
vbroker.orb.backCompat	false	When set to true, the server is operating in backward compatibility mode.
vbroker.orb.nullstring	false	When set to true, passing a null string causes a BAD_PARAM exception to be thrown. Passing zero length strings is allowed if vbroker.orb.backCompat is true.
vbroker.orb.admDir	*null*	Specifies the administration directory at which various system files are located. This property can be set using the VBROKER_ADM environment variable.
vbroker.orb.isNTService	false	When set to true, this property allows OAD to be running as NT system service, so that the it will not exit when the current user logs out.
vbroker.orb.obv.debug	false	When set to true, this property allows the ORB's object by value implementation to display debugging information.
vbroker.orb.dynamicLibs	*null*	Lists available services.
vbroker.orb.enableKeyId	true	When set to false, this property disables the use of key ids in client requests.
vbroker.orb.enableServerManager	false	When set to true, this property enables Server Manager within a server so that clients can access to it.
vbroker.orb.realtime.threadScheduling.enable	false	Enable the control of VisiBroker-RT 6.0 internal thread priorities (e.g. DSUser, VisLogger,...)
vbroker.orb.input.maxBuffers	16	Specifies the maximum number of input buffers retained in a pool.
vbroker.orb.input.buffSize	255	Specifies the size of the input buffer.
vbroker.orb.keyIdCacheMax	16384	Specifies maximum size of the object key id cache in a server.
vbroker.orb.keyIdCacheMin	64	Specifies minimum size of the object key id cache in a server.
vbroker.orb.output.maxBuffers	16	Specifies the maximum number of output buffers retained in a pool.
vbroker.orb.output.buffSize	255	Specifies the size of the output buffer.
vbroker.orb.initRef	*null*	Specifies the initial reference.
vbroker.orb.defaultInitRef	*null*	Specifies the default initial reference.
vbroker.orb.boa_map.TSingle	boa_s	Maps the boa bid policy of single threaded to *boa_s*.
vbroker.orb.boa_map.TPool	boa_tp	Maps the boa bid policy of thread pool to *boa_tp*.
vbroker.orb.boa_map.TSession	boa_ts	Maps the boa bid policy of thread session to *boa_ts.*

Property	Default	Description
vbroker.orb.boa_map.TPool_LIOP	boa_ltp	Maps the boa bid policy of local thread pool to *boa_s.*
vbroker.orb.alwaysProxy	false	When set to true, specifies that clients must always connect to the server using the Gatekeeper.
vbroker.orb.gatekeeper.ior	*null*	Forces the client application to always connect to server through the Gatekeeper whose IOR is provided.
vbroker.locator.ior	*null*	Specifies an IOR of the Gatekeeper which will be used as proxy to OSAgent. If this property is not set, the Gatekeeper specified by the vbroker.orb.gatekeeper.ior property is used for this purpose.
vbroker.orb.exportFirewallPath	false	Forces the server application to include firewall information as part of any servant's IOR which this server exposes (use `Firewall::FirewallPolicy` in your code to force it selectively per POA).
vbroker.orb.proxyPassthru	false	Forces `PASSTHROUGH` firewall mode globally in the application scope (use `QoSExt::ProxyModePolicy` in your code to force it selectively per object or per orb).
vbroker.orb.bids.critical	inprocess	The critical bid has highest precedence no matter where it is specified in the bid order. If there are multiple values for critical bids, then their relative importance is decided by the `bidOrder` property.
vbroker.orb.bidOrder	inprocess: liop: ssl: iiop: proxy: hiop: locator	You can specify the relative order of importance for the various transports. Transports are given precedence as follows: **1** inprocess **2** liop **3** ssl **4** iiop **5** proxy **6** hiop **7** locator The transports that appear first have higher precedence. For example: If an IOR contains both LIOP and IIOP profiles, the first chance goes to LIOP. Only if the LIOP fails is IIOP used. (The critical bid, specified by the vbroker.orb.bids.critical property, has highest precedence no matter where it is specified in the bid order.)
vbroker.orb.defAddrMode	`0` (Key)	The default addressing mode that client VisiBroker ORB uses. If it is set to `0`, the addressing mode is *Key*, if set to `1`, the addressing mode is *Profile*, if set to `2`, the addressing mode is *IOR.*
vbroker.orb.gatekeeper.ior	null	Forces the client application to always connect to server through the Gatekeeper whose IOR is provided.
vbroker.locator.ior	null	Specifies an IOR of the Gatekeeper which will be used as proxy to OSAgent. If this property is not set, the Gatekeeper specified by the vbroker.orb.gatekeeper.ior property is used for this purpose.

Server Manager properties

The following table lists the VisiBroker-RT for C++ Server Manager properties.

Property	Default	Description
`vbroker.serverManager.name`	`null`	Specifies the name of the Server Manager.
`vbroker.serverManager.enableOperations`	`true`	When set to `true`, this property enables operations exposed by the Server Manager to be invoked.
`vbroker.serverManager.enableSet Property`	`true`	When set to `true`, this property enables properties exposed by the Server Manager to be changed.

Location Service properties

The following table lists the VisiBroker-RT for C++ Location Service properties.

Property	Default	Description
`vbroker.locationservice.debug`	`false`	When set to true, this property allows the Location Service to display debugging information.
`vbroker.locationservice.verify`	`false`	When set to true, this property allows Location Service to check the existence of an object referred by an object reference sent down from the OSAgent.
`vbroker.locationservice.timeout`	`1`	Specifies the connect/receive/send time-out when trying to interact with the location service.

Interface Repository Resolver properties

The following table lists the VisiBroker-RT for C++ Interface Repository Resolver properties.

Property	Default	Description
`vbroker.ir.debug`	`false`	When set to `true`, this property allows the IR resolver to display debugging information.
`vbroker.ir.ior`	*`null`*	Specifies that when the `vbroker.ir.name` property is *null*, the ORB tries to use this property value to locate the IR
`vbroker.ir.name`	*`null`*	Specifies the name that will be used by the ORB to locate the IR.

TypeCode properties

The following table lists the VisiBroker-RT for C++ TypeCode properties.

Property	Default	Description
`vbroker.tyecode.debugp`	`false`	When set to `true`, this property allows the typecode code to display debugging information.
`vbroker.typecode.noIndirection`	`false`	When set to `true`, this property does not allow the use of indirection when writing a recursive typecode.

Client-Side IIOP Connection properties

The following table lists the VisiBroker-RT for C++ Client-Side IIOP Connection properties.

Property	Default	Description
vbroker.ce.iiop.ccm.connectionCacheMax	5	Specifies the maximum number of cache connection on a client. The connection will be cached when the client releases it. So, the next time the client needs a new connection, it first tries to collect an available one from the cache instead of just creating a new one.
vbroker.ce.iiop.ccm.connectionMax	0	Specifies the maximum number of the total connections within a client. This is equal to active connections plus the ones that are cached. The default value of zero means that the client will not try to close any of the old active or cached connections.
vbroker.ce.iiop.ccm.connectionMaxIdle	360	Specifies the time in seconds that the client uses to determine if a cached connection should be closed. If a cached connection has been idle longer than this time, then the client closes it.
vbroker.ce.iiop.connection.rcvBufSize	0	Specifies the size of the receive socket buffer. The default value 0 implies system dependent value
vbroker.ce.iiop.connection.sendBufSize	0	Specifies the size of the send socket buffer. The default value 0 implies system dependent value.
vbroker.ce.iiop.connection.tcpNoDelay	false	When set to true, the server's socket are configured to send any data written to them immediately instead of batching the data as the buffer fills.
vbroker.ce.iiop.connection.noCallback	false	When set to true, this property allows for callback capabilities from the server back to the client.
vbroker.ce.iiop.connection.socketLinger	0	Specifies a TCP/IP setting.
vbroker.ce.iiop.connection.keepAlive	true	Specifies a TCP/IP setting.

Client-Side LIOP Connection properties

VisiBroker-RT for C++ for Tornado does not support LIOP connections.

Server-Side Thread Session Connection properties

The table below lists the VisiBroker Edition for C++ Server-Side thread session connection properties

Server Engines (xxx), Server Connection Manager (yyy))

- boa_ts: This default server engine property is used specifically for BOA backward-compatibility option.
- iiop_ts: This default server engine is used in the ThreadSession variation with a default IIOP listener.

Table 14 Server-Side Thread Session Connection properties

Property	Default	Description
vbroker.se.xxx.host	*null*	Host name that can be used by this server engine. If this variable is set to null (the default), the host name from the system is used.
vbroker.se.xxx.scms	*xxx=boa_ts, boa_ts xxx=iiop_ts, iiop_ts*	List of Server Connection Manager name(s).
vbroker.se.xxx.scm.yyy.manager.type	*Socket*	Specifies the type of Server Connection Manager.
vbroker.se.xxx.scm.yyy.manager.connectionMax	0	Specifies the maximum number of connections the server accepts. The default value 0 sets no restriction.
vbroker.se.xxx.scm.yyy.manager.connectionMaxIdle	0	Specifies the time in seconds that the server uses to determine if an inactive connection should be closed or not.
vbroker.se.xxx.scm.yyy.listener.type	IIOP	Specifies the type of protocol the listener is using.
vbroker.se.xxx.scm.yyy.listener.port	0	Specifies the port number that is used with the host name property. The default value 0 means the system picks a random port number.
vbroker.se.xxx.scm.yyy.listener.proxyPort	0	Specifies the proxy port number that is used with the proxy host name property. The default value 0 means the system picks a random port number.
vbroker.se.xxx.scm.yyy.listener.rcvBufSize	0	Specifies the size of the receive socket buffer. The default value 0 implies system dependent value
vbroker.se.xxx.scm.yyy.listener.sendBufSize	0	Specifies the size of the send socket buffer. The default value 0 implies system dependent value.
vbroker.se.xxx.scm.yyy.listener.socketLinger	0	A TCP/IP setting.
vbroker.se.xxx.scm.yyy.listener.keepAlive	TRUE	A TCP/IP setting.
vbroker.se.xxx.scm.yyy.dispatcher.type	*Thread Session*	Specifies the type of thread dispatcher used in the Server Connection Manager.
vbroker.se.xxx.scm.yyy.dispatcher.threadStackSize	0	The size of the thread stack.
vbroker.se.xxx.scm.yyy.dispatcher.coolingTime	3	The time in seconds before a connected thread returns back to the thread pool. This time allows the thread to server more than one request.

Server-Side Thread Pool Connection properties

The following table lists the VisiBroker-RT for C++ Server-Side thread pool connection properties

Server Engines (xxx), Server Connection Manager (yyy)

- boa_tp
- se_iiop_tp0, scm_iiop_tp0 (General thread pool)

Property	Default	Description
vbroker.se.xxx.host	null	Specifies the host name that can be used by this server engine. The default value *null* means use the host name from the system.
vbroker.se.xxx.scms	*xxx=boa_tp, boa_tp xxx=se_iiop_tp0, scm_iiop_tp0*	List of Server Connection Manager name(s).

Property	Default	Description
vbroker.se.xxx.scm.yyy.manager.type	Socket	Specifies type of Server Connection Manager.
vbroker.se.xxx.scm.yyy.manager.connectionMax	0	Specifies the maximum number of cache connections the server will accept. The default value 0 implies no restriction.
vbroker.se.xxx.scm.yyy.manager.connectionMaxIdle	0	Specifies the time in seconds that the server uses to determine if an inactive connection should be closed or not.
vbroker.se.xxx.scm.yyy.manager.garbageCollectTimer	30	Specifies the garbage-collect timer (in seconds) for connections
vbroker.se.xxx.scm.yyy.listener.type	IIOP	Specifies the type of protocol the listener is using.
vbroker.se.xxx.scm.yyy.listener.port	0	Specifies the port number that is used with the host name property. The default value 0 means the system will pick a random port number.
vbroker.se.xxx.scm.yyy.listener.proxyPort	0	Specifies the proxy port number that is used with the proxy host name property. The default value 0 means the system will pick a random port number.
vbroker.se.xxx.scm.yyy.listener.rcvBufSize	0	Specifies the size of the receive socket buffer. The default value 0 implies system dependent value.
vbroker.se.xxx.scm.yyy.listener.sendBufSize	0	Specifies the size of the send socket buffer. The default value 0 implies system dependent value.
vbroker.se.xxx.scm.yyy.listener.socketLinger	0	Specifies a TCP/IP setting.
vbroker.se.xxx.scm.yyy.listener.keepAlive	true	Specifies a TCP/IP setting.

Properties that support bidirectional communication

The table below lists the properties that support bidirectional communication. These properties are evaluated only once—when the SCMs are created. In all cases, the `exportBiDir` and `importBiDir` properties on the SCMs are given priority over the `enableBiDir` property. In other words, if both properties are set to conflicting values, the SCM-specific properties will take effect. This allows you to set the `enableBiDir` property globally and specifically turn off bidirectionality in individual SCMs.

Table 15 Bidirectional communication properties

Property	Default	Description
vbroker.orb.enableBiDir	none	You can selectively make bidirectional connections. If the client defines `broker.orb.enableBiDir=client` and the server defines `vbroker.orb.enableBiDir=server` the value of `vbroker.orb.enableBiDir` at the gatekeeper determines the state of the connection. Values of this property are: `server`, `client`, `both` or `none`.

Property	Default	Description
vbroker.se.<se>.scm.<scm>.manager.exportBiDir	By default, not set by the ORB.	A client-side property. Setting it to true enables creation of a bidirectional callback POA on the specified server engine. Setting it to false disables creation of a bidirectional POA on the specified server engine.
vbroker.se.<se>.scm.<scm>.manager.importBiDir	By default, not set by the ORB.	A server-side property. Setting it to true allows the server-side to reuse the connection already established by the client for sending requests to the client. Setting it to false prevents reuse of connections in this fashion.