Spectra ORB
C Edition

IDL Guide

PRISMTECH

Spectra ORB
C Edition

IDL Guide

PRISMTECH

Part Number: EORBC-IDLG

Doc Issue 31, 2 August 2013

Copyright Notice

Contents

Table of Contents

C Language Mapping

Preface

About the IDL Guide

The *Spectra ORB C Edition IDL Guide* provides instructions and background information needed to understand the IDL language, data types, IDL to C mappings, and how to use the Spectra ORB C Edition IDL to C compiler.

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

Intended Audience

The *IDL Guide* is intended to be used by developers and engineers, working in a distributed computing environment using Spectra ORB, who have a good level of knowledge and experience of CORBA and IDL.

Organisation

The *IDL Guide* is organized as follows:

- Chapter 1, *Introduction to IDL*, as the title states, introduces the OMG's Interface Definition Language (IDL).
- Chapter 2, *Concepts and Features*, describes the IDL basics and provides essential background information.
- Chapter 3, *Compiling an IDL Specification* describes how to write IDL files which will be used to generate applications' native language source files (in C).
- Section 4, *IDL to C Mapping*, describes the OMG's IDL to C language mapping for the ORB.

Conventions

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

 Item of special significance or where caution needs to be taken.

i Item contains helpful hint or special information.

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

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

C C language specific.

C++ C++ language specific.

Java Java language specific.

Hypertext links are shown as *blue italic underlined*.

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

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

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

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

// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, “”);
```

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

Sans-serif Bold is used to indicate user-related actions, *e.g.* **File > Save** from a menu.

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

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters	**European Head Office**
PrismTech Corporation 400 TradeCenter Suite 5900 Woburn, MA 01801 USA	PrismTech Limited PrismTech House 5th Avenue Business Park Gateshead NE11 0NG UK
Tel: +1 781 569 5819	Tel: +44 (0)191 497 9900 Fax: +44 (0)191 497 9901

Web: *http://www.prismtech.com*
Technical questions: *crc@prismtech.com* (Customer Response Center)
Sales enquiries: *sales@prismtech.com*

CORBA IDL

CHAPTER

1 Introduction to IDL

The Interface Definition Language (IDL) enables interfaces to be defined which are independent of any particular programming language. This enables the components of CORBA-based distributed applications to be written in whichever programming language is most appropriate, where different components of the same application can even be written in different languages. For example, a client could be written in Java and a server in C, C++, or even COBOL.

This capability provides enormous power and flexibility, especially where legacy applications must be made to communication with new applications or components which have been written in a different language.

In CORBA, the object's interface defines the operations the object provides and how a client interacts with the object. The interface represents the contract between an object and a client that attempts to use that object. The interface defines the object's reference and allows an object to declare its type and the names of its inherited types. It also provides the names of the operations it supports and the parameters and return types of those operations.

To implement an object described in IDL, the interface description must be translated into the appropriate programming language constructs as defined by CORBA's IDL mapping for that language. Translation is performed by an IDL compiler supplied with the ORB. The IDL compiler is described in Using the IDL Compiler on page 17.

The IDL to C language mapping is compliant with the OMG C Language Mapping Specification, OMG document 99-07-35.

1.1 Defining the Object's Interface

In IDL, an object's interface consists of a set of named operations and the parameters supplied to those operations. The following example shows how IDL would define the interface of an object in a simple application:

```
interface SimpleObject
{
   boolean oneOperation (in string aString);
   void anotherOperation (in string anotherString);
};
```

A description of an interface in IDL is called an IDL *specification* of that interface. IDL specifications should be created in a source file with a `.idl` file extension.

IDL source files are passed to the ORB's IDL compiler, to generate new files with programming-language constructs and implementations that can be used in an application. The IDL interface construct maps directly to the programming language's construct for an object reference:

The following sections describe IDL and the constructs you can use to define an object's interface.

CHAPTER

2 Concepts and Features

The OMG's Interface Definition Language (IDL) is a modelling language for defining interfaces. The interface definitions created are platform and language independent. The IDL interface definitions enable applications, which have been implemented in different programming languages, for example C, C++, Java, or even COBOL, to transparently communicate with each other through ORB middleware.

IDL compilers generate programming code for specific languages (such C, C++, Java, or COBOL) using the IDL interface definitions mentioned above. An individual IDL compiler is designed to generate programming code for a specific language and in accordance with an appropriate mapping specification defined by the OMG.

IDL compilers use the interface definitions to create the appropriate classes, methods, operations, parameters and attributes needed for the specific programming language used by the developer. The developers use the IDL-generated files as the basis for implementing client and server components of their distributed, CORBA-based applications.

This section describes the basics of the IDL language and how to use it to create programming language-specific applications. The complete description of the syntax and semantics of IDL are given in the OMG's Common Object Request Broker Architecture and Specification document.

2.1 IDL Language Basics

The IDL syntax is similar to C++ syntax. The key differences between IDL and C++ syntax are:

- a function return type is mandatory
- a name must be supplied with formal parameter in an operation declaration
- a parameter list consisting of a single void token does not act as an empty parameter list
- tags are required for structures, discriminated unions, and enumerations
- integer types cannot be declared as `int` or `unsigned`; they must be explicitly declared as `short`, `long`, or `long long`
- `char` types cannot be qualified as `signed` or `unsigned`

2.1.1 IDL Conventions

The Interface Definition Language uses the conventions described below.

2.1.1.1 Comments

There are two comment delimiters in IDL:

- The comment pair `/*` and `*/` is used to delimit comments that span multiple lines.
- The double slash `//` begins a comment that is terminated at the end of a line. The comment may be placed after a code statement on the same line or on a line by itself.

Comments do not nest.

2.1.1.2 Identifiers

Identifiers are names for IDL definitions such as constants, types, and operations. An identifier is an arbitrarily long sequence of letters, digits, and the underscore (_) character. Identifiers must begin with a letter.

In this example, `MaxVal` is an IDL identifier for a long integer:

```
long MaxVal;
```

Identifiers are scoped. See *Modules* on page 15 for information on how scoping works with identifiers.

Identifiers are case sensitive in that a given identifier must be spelled identically with respect to case throughout an IDL specification. However, using two different identifiers which differ only in case will produce a compilation error.

IDL reserves a set of keywords that cannot be used as identifiers. These reserved keywords are shown in Table 1, *IDL Keywords*.

Table 1 IDL Keywords

abstract	double	local	raises	truncatable
any	enum	long	readonly	typedef
attribute	exception	module	sequence	unsigned
boolean	factory	native	short	union
case	FALSE	Object	string	ValueBase
char	fixed	octet	struct	valuetype
const	float	oneway	supports	void
context	in	out	switch	wchar
custom	inout	private	TRUE	wstring
default	interface	public		

2.1.2 The typedef Mechanism

Use the `typedef` mechanism to name a data type. A `typedef` definition begins with the keyword `typedef`, followed by the data type and `typedef` name. You can name basic data types, constructed data types, and template types using `typedef` declarations. `typedefs` are typically used to name template types, such as sequences and arrays.

2.1.3 Data Types

IDL supports the basic data types such as char, short, long, and float; structured types such as structs, unions, and enumerations; template types such as sequences, arrays, and strings; and structured exceptions. The interface type can also be used as an operation parameter. An interface argument is passed as an object reference that refers to the instance of the object implementation it represents. CORBA defines an `any` type that can represent any IDL type and can be used as an argument or value type wherever multiple different types are acceptable. You can use typedefs or create an alias for any of these types.

An IDL compiler for a specific programming language maps basic IDL data types to the appropriate native data types of that programming language.

2.1.3.1 Basic Types

The basic IDL data types are listed in Table 2, *Basic IDL Data Types*.

Table 2 Basic IDL Data Types

IDL Identifier	Type	Description
`any`	Any type	Self-describing values that can express any IDL type
`boolean`	Boolean type	1 or 0 (zero)
`char`	Char type	8-bit ISO Latin 1 characters
`double`	Floating-point Type	IEEE double-precision floating-point numbers
`float`	Floating-point Type	IEEE single-precision floating-point numbers
`long`	32-bit integer	-2^31 ... 2^31-1
`long double`	Floating-point type	IEEE double-extended floating-point numbers
`long long`	64-bit integer	-2^63 ... 2^63-1
`octet`	Octet type	8-bit quantity that is guaranteed not to undergo any conversion during communication
`short`	16-bit integer	-2^15 ... 2^15-1
`unsigned long`	32-bit integer	0 ... 2^32-1
`unsigned long long`	64-bit integer	0 ... 2^64-1
`unsigned short`	16-bit integer	0 ... 2^16-1

2.1.3.2 Constructed Types

IDL provides three constructed data types: structures, unions, and enumerations.

2.1.3.2.1 Structures

A structure allows related data to be grouped under a single name. Structures are declared with the keyword `struct`, followed by an identifier and a list of members enclosed in braces. Each struct member must be an IDL type[1].

This example of an IDL structure named `date` has three members: two of type `unsigned short` and one of type `short`.

```
struct date
{
   unsigned short month;
   unsigned short day;
   short year;
};
```

2.1.3.2.2 Discriminated Unions

A union groups items of different types and sizes, but only one member of the union has a useful value at any one time. This implies a saving in the amount of storage allocated to a union.

Discriminated unions use a discriminator (or switch) to help keep track of which union member has the most current value assigned to it. Each member of an IDL discriminated union has an associated case label which must match (or be castable to) the type of the switch field. The switch field uses the case label to determine which union member to use. An optional member with a `default` case label can be used. Access to the switch and the related element is language-mapping dependent.

An IDL discriminated union is declared with the keyword `union`, followed by an identifier for the union. This is followed by the keyword `switch` and a switch type enclosed in parentheses. Finally, a list of members that have case labels is enclosed in braces. Each union member must be an IDL type.

The switch type can be long, long long, short, unsigned long, unsigned long long, unsigned short, char, boolean, or enum.

This example shows an IDL discriminated union named `token`.

```
union token switch (long)
{
   case 1: char cval;
   case 2: float fval;
   case 3: double dval;
   default: long lval;
};
```

2.1.3.2.3 Enumerations

An enumeration is an ordered list of identifiers, referred to as enumerators. Enumerations are declared with the keyword `enum`, followed by an identifier and a comma-separated list of enumerators enclosed in braces.

The order in which the identifiers are named defines their relative order. This order allows two enumerators to be compared. By default, the first enumerator has a value of 0; the value of each subsequent enumerator is incremented by one.

1. A *type* means all types that can be defined as well as basic types.

This example shows an IDL enumerated type named `workday`.

```
enum workday { Monday, Tuesday, Wednesday, Thursday, Friday };
```

2.1.3.3 Template Types

IDL provides template data types: sequences, strings, and fixed.

2.1.3.3.1 Sequences

Example

A sequence is a one-dimensional array that can be declared with an optional maximum size. If the sequence is declared with a maximum size, it is referred to as a bounded sequence; if no maximum size is specified, it is referred to as an unbounded sequence. The length of a sequence can change dynamically but the length of a bounded sequence cannot exceed the maximum size fixed at compile time. Sequence elements can be any of the IDL types.

This example shows a bounded sequence. The keyword `typedef` names the sequence data type.

```
typedef sequence<long, 64> vec64;
```

This example shows an unbounded sequence named `vec`.

```
typedef sequence<long> vec;
```

2.1.3.3.2 Strings

A string is a one-dimensional array of 8-bit ISO Latin1 characters that can be declared with an optional maximum length. If the string is declared with a maximum length, it is referred to as a bounded string; if no maximum length is specified, it is referred to as an unbounded string. The length of a string can change dynamically but the length of a bounded string cannot exceed the maximum length fixed at compile time.

This example shows a bounded string. The keyword `typedef` names a bounded string data type:

```
typedef string<16> name16;
```

This example shows an unbounded string named `name`:

```
typedef string name;
```

2.1.3.3.3 Wide Strings and Chars

Wide types (wchar/wstring) are not supported. By default wide types are treated as the non-wide equivalent (string/char) type. If the *`-no_map_wide`* IDL compiler flag is used then the compiler generates an error if a wide type is used.

2.1.3.3.4 Fixed

The fixed type represents fixed point decimal numbers with a specified number of significant digits and scale factor. There can be no more than 31 significant digits in a given fixed point number. The scale factor is a non-negative integer less than or equal to the number of significant digits.

This example defines a `Money` type capable of storing 9-digit figures, 2 of which are to the right of the decimal point.

```
typedef fixed<9,2> Money;
```

The fixed point type can also be used to define integer types with a specified number of significant digits by setting the scale factor to 0. This is illustrated in the following example.

```
typedef fixed<31,0> BigInteger;
```

2.1.3.4 Arrays

IDL defines multi-dimensional, fixed-size arrays. Each dimension of the array has an explicit fixed size that cannot vary at runtime.

An array is declared with a type specifier, an identifier, and the dimensions enclosed in bracket pairs.

- any of the IDL types may be made into a multi-dimensional array.
- each dimension must be a positive integer constant expression.

This example defines a one-dimensional array of bounded strings:

```
typedef string<20> FiveStrings[5];
```

This example defines a two-dimensional matrix of longs:

```
typedef long LongMatrix[4][4];
```

2.1.3.5 The any Type

The `any` type can express any legal IDL value. An `any` is self-describing and is intended to be used as an argument or value type wherever multiple different types are acceptable. An `any` can be passed as an operation argument.

An `any` contains a `TypeCode` and a value which is of the type indicated by the `TypeCode`. The `any` type can be any IDL-specified type, including another `any`. The `TypeCode` allows applications that use it to interpret the type of the data the `any` contains.

The `any` maps into an implementation-language mapped type.

i Support for `anys` must be enabled in the `idlcpp` compiler by using the `-gen_any` command-line switch and linking in the correct `any` libraries. The C `Any` type mapping is defined in the `eOrbC/CORBA/any.h` header file and must be included.

2.1.4 Constant Types

A constant provides a way to declare types that are initialized with values that cannot be changed. Constants are declared with the keyword `const`, followed by a type specifier, an identifier, the assignment operator (=), and the value to which the constant is initialized.

Table 3, *Constants*, on page 11 lists the IDL types which can be used to declare constants.

Table 3 Constants

boolean	short
char	string
double	unsigned long
fixed	unsigned long long
float	unsigned short
long	
long long	
octet	

This example declares a constant `MaxVal` of type `long` with a fixed value of `100`:

```
const long MaxVal = 100;
```

i Long long and unsigned long long constants are limited to long and unsigned long values on platforms with compilers that do not have built-in long long support.

2.1.5 Interfaces

An interface is defined with the keyword `interface` followed by an identifier that names the interface. The interface identifier may optionally be followed by an inheritance specification and the interface body enclosed in braces. The body of an interface may include IDL type definitions.

A forward declaration of an interface consists simply of the keyword `interface` followed by an identifier.

An interface forms a naming scope for identifiers. An identifier that is defined within an interface can have the same name as an identifier that is defined outside the interface. An identifier that is defined within an interface can be used outside of the interface when it is qualified with the name of the interface and the name resolution operator (**::**).

This example shows an IDL specification with an interface object that provides options to start and stop, and returns the errors of a ManagedElement.

```
enum error{NO_ERROR, ELEMENT_BUSY, ELEMENT_NOT_RESPONDING};

typedef sequence<error>  errors;

interface ManagedElement
{

   // Management operations
   boolean start();
   boolean stop();

   errors get_errors();

};
```

The example uses `enum` to define `error`, which enumerates the possible errors. The `typedef` statement defines `errors` as a `sequence` of the `enum error`. The `ManagedElement` interface contains `start()` and `stop()` operations, which return boolean type values, and a `get_errors()` operation that has a return value of type `errors`.

2.1.5.1 Operations

Operation declarations occur in an interface body. Operation declarations consist of a return type followed by an identifier that names the operation, a parameter list enclosed in parentheses, and an optional `raises` expression.

2.1.5.1.1 Return Type

An operation must specify a return type. Return types can be any IDL type. If an operation does not return a result, it must specify the `void` type.

2.1.5.1.2 Parameter Declarations

A parameter list consists of zero or more parameter declarations separated by commas. A parameter declaration consists of a directional attribute followed by an IDL data type and an identifier that names the parameter. The directional attribute specifies the direction in which the parameter is to be passed. Table 4, *Directional Attributes* lists the IDL directional attributes.

Table 4 Directional Attributes

Attribute	Meaning
in	The parameter is passed from the client to the server
out	The parameter is passed back from the server to the client
inout	The parameter is passed in both directions

The following example illustrates an operation which uses the keyword `out` to specify that the `greetstr` string is passed from object to client.

```
string greeting (out string greetstr);
```

2.1.5.1.3 Oneway Operations

An operation declaration can be preceded by the optional keyword `oneway`. A oneway operation is a non-blocking call (it does not suspend the client when it makes the request). A oneway operation is invoked once at most, and delivery of the call is not guaranteed. If the request fails after being issued, the client is not notified. The return type of a oneway operation must be `void` and the parameter list must not contain any `inout` or `out` parameters. Oneway operations can only raise standard exceptions, and will only raise those exceptions if the client ORB encounters an error prior to sending the request.

The following example declares an interface with a single oneway operation:

```
interface alarms
{
   oneway void notify (in string event);
};
```

2.1.5.1.4 Raises Expressions

A `raises` expression specifies which exceptions may be raised by an operation. the expression consists of the keyword `raises` followed by a list of exceptions enclosed in parentheses. The exception list must consist of one or more previously-defined exceptions separated by commas.

The following example declares that a single exception, `NotRunning`, may be raised by the `stop` operation.

```
boolean stop () raises (NotRunning);
```

The `raises` expression specifies any operation-specific exceptions that may be raised by a call to the operation. In addition to these exceptions, IDL defines a set of standard system exceptions which can be raised by the ORB as a result of a call to the operation. These system exceptions must *not* be listed in a `raises` expression. System exceptions may be raised by the ORB even if no operation-specific exceptions have been defined.

2.1.5.2 Attributes

In addition to operation declarations, an interface body can have attribute declarations. An attribute is declared with the keyword `attribute`, which can be preceded by the optional keyword `readonly`, followed by a type specifier and an identifier.

To enable a client to access an attribute value, the IDL compiler maps an IDL attribute declaration to two functions: one to set the value of the attribute and one to retrieve the value of the attribute. The set function takes an input parameter of the same type as the attribute; the get function returns a value of the same type as the attribute. If an attribute is defined as `readonly`, it maps only to the get function.

The following example defines an interface with a single attribute, the string element_owner.

```
interface ManagedElement
{
   attribute string element_owner;
};
```

2.1.5.3 Exceptions

An exception is a data structure that is returned to indicate that a particular type of exceptional condition occurred as a result of a request on an object. There are two types of exceptions: user exceptions and standard exceptions.

2.1.5.3.1 User Exceptions

The exception data structure is similar to a struct in syntax and form. User exceptions are declared with the keyword `exception` and an identifier followed by an optional list of members enclosed in braces. The members of an exception provide additional information you can use to determine which exceptional condition occurred or more detail about the exception that occurred.

Exceptions can not be used as parameters to operations or members of elements or other data types.

To specify what types of exceptions an operation can raise, an optional `raises` expression can follow the parameter list of an operation declaration. See *Raises Expressions* on page 13 for details.

The following example declares an exception named `no_operating_info` in the `ManagedElement` interface. The `reason` member of the `no_operating_info` exception can be used to specify the reason a status could not be returned by the `get_state()` operation. The exception may be raised by the `status` operation.

```
interface ManagedElement
{
   exception no_operating_info { string reason };

   status get_state (out operating_info current_state)
      raises  (no_operating_info);
};
```

2.1.5.3.2 System Exceptions

CORBA defines a set of system exceptions that correspond to standard runtime errors that the ORB may signal as a result of any request. These exceptions are implicitly listed in every operation's `raises` expression.

The *User Guide* includes a list of system exceptions and their minor codes as well as where the exceptions are thrown.

2.1.5.4 Inheritance

Inheritance is a mechanism for defining an interface by adding new elements to an existing interface. An existing interface which the new interface inherits from is called a base interface of the new interface. An interface can be derived from any number of base interfaces.

To specify that one interface inherits from another, follow the interface name in the interface definition with a colon (`:`) and the name of the interface it inherits from. If an interface inherits from multiple interfaces, use commas to separate their names.

A derived interface inherits all the elements (constants, types, attributes, exceptions, and operations) of the base interfaces from which it is derived. If more than one base interface uses the same name for a constant, type, or exception, qualify the name with its interface name in the derived interface. A derived interface ***cannot*** inherit from two interfaces that contain the same operation or attribute name. Additionally, a derived interface ***cannot*** redefine an inherited operation or attribute name.

The following example specifies that the interface `GroupManagedElement` inherits from the previously-defined `ManagedElement` interface.

```
interface GroupManagedElement : ManagedElement
{
   // new operations and attributes
};
```

There can be multiple levels of inheritance. For example, interface X can inherit from base interface Y which in turn inherits from base interface Z. In this situation, Z is called an *indirect base* interface of X. Y is a *`direct base`* interface of X.

2.1.6 Modules

Modules can be used to control the naming scope of identifiers. By properly scoping IDL declarations within modules, you can avoid conflicts between identifiers in different modules. An identifier that is defined within a module can have the same name as an identifier that is defined outside the module.

The following example uses two `MaxVal` identifiers, one within a module named `business` and the other outside of the module.

```
const long MaxVal = 100;

module business
{
   const long MaxVal = 50;
};
```

An identifier defined within a module can be used outside of that module when it is qualified with the name of the module and the name resolution operator (`::`). In the example above, the `MaxVal` identifier defined within the `business` module can be referred to using `business::MaxVal`.

Modules can be used to scope identifiers of IDL data types, constants, exceptions, interfaces, and other modules.

The following example uses modules to control the scope of two interface definitions with the same name:

```
module Europe
{
   interface ManagedElement
   {
      // operations
   };
};

module USA
{
   interface ManagedElement
   {
      // operations
   };
};
```

An identifier can only be defined once in a module, however it can be redefined in nested modules. An identifier can be used in an unqualified form within a particular module and will be resolved by successively searching farther out in enclosing modules.

2.1.6.1 The CORBA Module

The CORBA specification includes pre-defined names which can be used in your IDL specification. This includes interface names such as `TypeCode`. To avoid conflicts between pre-defined CORBA names and user-defined names, CORBA names are defined in a `CORBA` module. To use CORBA names in your IDL specification, qualify them with `CORBA::`, for example, `CORBA::TypeCode`.

This does ***not*** apply to IDL keywords. For example, use `Object` instead of `CORBA::Object`.

CHAPTER

3 Compiling an IDL Specification

The first step in developing an application is to define the operations that the object is intended to provide and to specify that interface using IDL. The IDL files are then compiled into C source and header files using the IDL to C compiler, `idlc`.

3.1 Using the IDL Compiler

Interface specifications written in IDL are stored in IDL source files. These files must have an *`.idl`* extension in order to be recognised by the IDL compiler.

The ORB's IDL to C compiler, *`idlc`*, is located in the following directory:

```
$EORBHOME/bin/$EORBENV
```

on Windows:

```
%EORBHOME%\bin\%EORBENV%
```

i The C IDL compiler is a Java application and requires a JRE to run.

3.1.1 Command Syntax

The `idlc` compiler is run from the command line as follows:

```
%  idlc [options] <idl_files>
```

where

`<idl_files>` is a list of one or more developer-written IDL source files
`[options]` is a list of zero or more optional command-line options.

Using `idlc` with no file name specified displays usage information. Specifying a file name with no other parameters generates only the mappings file (equivalent to using the `-mappings` parameter).

The complete list of command-line parameters is described in Table 5, *IDL Compiler Options*, on page 17. Note that command-line parameters are case sensitive.

Table 5 IDL Compiler Options

Option	Description
`-bis <suffix>`	Generated base implementation file name suffix (default `B.c`)
`-both`	Generates both server and client side.
`-chs <suffix>`	Generated client header file name suffix (default `C.h`)

Table 5 IDL Compiler Options (Continued)

Option	Description
`-cis <suffix>`	Generated client implementation file name suffix (default `C.c`)
`-client`	Generates client side only.
`-D <symbol>`	Define symbol for pre-processing IDL. The compiler defines two symbols by default: `_EORB_` and `_EORB_C_`. These can be used to guard Spectra ORB-specific IDL constructs.
`-debug`	Provides debugging information.
`-dll <name>`	Generate linkage declarations for Windows/Win32 systems.
`-ft`	Enable fault-tolerant stub semantics, where some system exceptions will cause a transparent reinvocation of a failed operation.
`-gen_any`	Generates code for the support of anys. Must be used with Dynamic Anys.
`-gen_any_exc`	Generate code for managing exception types in Any.
`-gen_any_names`	Generate code for Anys typecode names. Names are optional.
`-gen_impl`	Generate skeleton implementation functions.
`-gen_impl_exc`	Generate skeleton implementation functions that raise the `NO_IMPLEMENT` system exception.
`-gen_impl_prefix`	Adds prefix to generated implementation functions. Useful for services like Event Service where service and clients require different implementation functions.
`-gen_impl_cpp`	Generate C++ servant implementation.
`-gen_vepv`	Generate servant EPV structures.
`-help`	Shows the help screen.
`-I<directory>`	Includes `directory` in search path for preprocessor.
`-ihs <suffix>`	Generated interface header file name suffix (default `.h`)
`-lightweight`	Generate code for SCA V4 lightweight profile.
`-lis <suffix>`	Generated local implementation file name suffix (default `C_i.c`)
`-locate`	Generates stubs that will always do an initial GIOP locate request when first used.

Table 5 IDL Compiler Options (Continued)

Option	Description
`-mappings`	Generates language mappings only.
`-nocoloc`	Generates stubs only for remote calls.
`-nolock`	Disable client-side concurrency locking.
`-nopragma`	Ignore pragma prefix directives (`#pragma`) for the generation of type information.
`-no_map_wide`	Do not map wchar/wstring types to char/string. Generate compiler error instead.
`-no_pool`	Allocate all skeleton parameters from heap.
`-no_vepv_init`	Do not initialize servant with generated EPV structures in initialization function.
`-op_check`	Generate additional skeleton code to validate operation names. May be required if interoperating with an ORB using DII.
`-output` *(or* `-o`*)*	Output directory for generated files.
`-server`	Generates server side only.
`-shs <`*suffix*`>`	Generated server header file name suffix (default `S.h`)
`-sis <`*suffix*`>`	Generated server implementation file name suffix (default `S.c`)
`-skel`	Generate server-side skeletons only.
`-sync_server`	Default to sync with server for oneway operations.
`-sync_target`	Default to sync with target for oneway operations.
`-syntax`	Checks syntax only.
`-version`	Prints the compiler version.
`-vis <`*suffix*`>`	Generated servant implementation file name suffix (default `S_i.c`)
`-zc_client`	Enable client zero copy semantics.

3.1.2 Optimising Performance

If performance is a concern, then a number of techniques can be used to optimise both the generated code and the ORB runtime.

1. *Use IDL structs to group data types of the same size:*

   If a data struct is aligned in memory in the same way as it is encoded in a CDR stream, then the ORB may be able to detect this and directly copy data into the stream rather than individually marshal each element. Conversely, if a variable-length data type (such as a string) is a member of a struct then the ORB will

not be able to make such an optimisation. In general, defining data types in a way that matches as closely as possible the CDR encoding of those data types is most likely to give optimal marshaling performance.

Note that the ORB cannot always detect if struct data types can be copied directly to CDR. The IDL copy pragma (see Section 3.3, *Pragma support*, on page 22) can be used to flag struct types that can be copied in this way. *Warning*: only use this pragma if you are sure of what you are doing.

2. *Use IDL compiler flags to generate optimal code:*

`-nocoloc` : Disables generation of code to support co-located calls. Use this when the client and server will never be co-located in the same process (*i.e.*, requests will always be handled *via* a transport connection).

`-nolock` : Disables client-side concurrency locking. Use this when only a single thread will ever make a request on an instance of a client-side object reference.

`-zc_client` : Enables zero copy client semantics for sequence data. When the data buffer in a sequence can be mapped dirrectly to its CDR encoding (such as for sequences of simple types), then sequence data returned from an operation (`out` arguments) can be mapped directly to the CDR buffer associated with the thread making the request. This data is valid until the thread is used to make a subsequent request, when the CDR buffer is re-used. Note that this breaks strict CORBA semantics, so care should be taken when using this option.

3. *Use ORB initialization arguments to optimise runtime performance:*

`-ORBPrivateServerConn` : For each server-side connection use a dedicated request-handling thread (as opposed to one taken from a thread pool).

`-ORBPrivateClientConn` : For each client object reference create a new transport connection to handle all requests; this connection is released when the object reference is destroyed.

`-ORBFixedPOA` : This disables POA locking when resolving the servant used to handle a request. Can be used when servants are not dynamically created or destroyed when the POA is active (*i.e.*, all servants are created before the ORB is run or the POA is activated).

4. *Use a transport other than IIOP:*

The ORB supports a variety of pluggable transports, many of which give greater performance that the standard IIOP TCP-based transport. For example, for intra-node communications both UIOP (unix domain socket) and MQIOP (POSIX message queue) typically give better performance than IIOP. It is also possible to install multiple transports so that an application can use one transport for intra-node communication and another for inter-node communication (typically IIOP). See the ORB User Guide for more information on pluggable transports.

3.2 IDL Compiler Output

Given an IDL specification file called `hello.idl`, the IDL compiler generates the files shown in *Figure 1*.

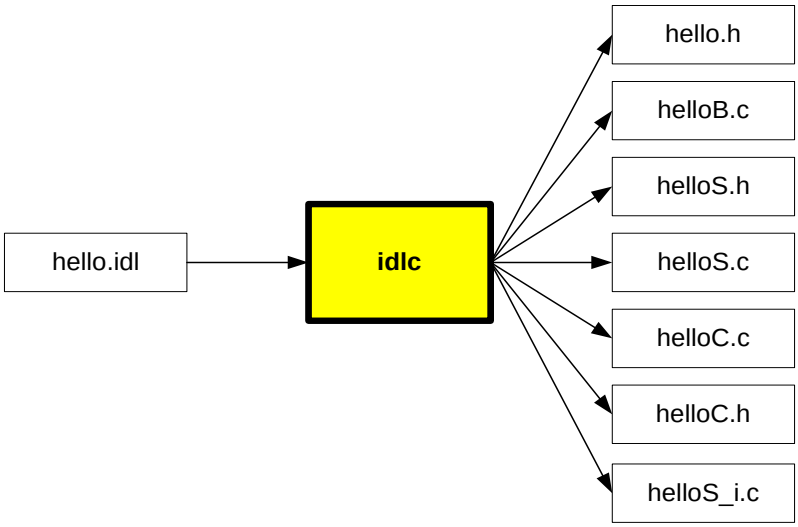

Figure 1 Files Generated

Here is a brief description of their content and function.

- `hello.h` is the ***interface header file***. It defines the C types associated with the IDL types defined in the IDL specification. The C types in this header file are used by client and server programs and object implementations. For clients, this header file provides remote access to object references through stubs that correspond to the IDL-defined `interface` operations. A client program includes this header file and makes a request by calling one of the stub routines on an object reference.
- `helloB.c` is the ***interface implementation file*** and contains implementations of the types declared in `hello.h`. The interface implementation file is compiled and linked with a client program and with object implementations.
- `helloS.h` is the ***implementation base header file*** and provides C types and EPV and VEPV structs that you use in your object implementation. An object implementation file includes this header file, which in turn includes the interface header file, `hello.h`.
- `helloS.c` is the ***object implementation base dispatch file*** and contains the implementations of operation dispatch functions required by object implementations. The object implementation dispatch file is compiled and linked with a object implementation.
- `helloS_i.c` contains generated servant skeleton implementation functions and EPV structures.
- `helloC.h` is the ***client header file*** and provides the C stub machinery definitions when generated.
- `helloC.c` is the client functions interface to the ORBs marshalling engine internal APIs in the runtime libraries of the ORB.

3.2.1 Using the IDL Compiler Output

The header files produced by the IDL compiler are used as follows:

- the client's modules must include the client header file (using a `#include` directive)
- the server's modules must include the server header file (using a `#include` directive)

The client and server implementation files produced by the IDL compiler should be compiled and the resulting library linked in with the application.

3.2.2 Initializing IDL modules

For each IDL file compiled the IDL compiler will generate an initialization function of the form '*`EORB_IDL<name>_init (void)`*'. This function initializes C data structures required by the ORB and should be called in the main program before the ORB is initialized. If one IDL file includes another IDL file then its initialization function will itself call the initialization functions of any included modules.

3.2.3 Creating Libraries

By default the IDL compiler generates three types of code: code used by the client, code used by the server, and common code used by both client and server. The IDL compiler flags *`-client`*, *`-server`*, *`-both`* and *`-skel`* control what code is generated.

Depending on the application this code can be placed into libraries in a number of ways.

- For a *client-only* application the client and shared code should be used.
- For a *server-only* application then the server and shared code should be used.
- For an application with both client and server components then all code should be used.

An alternative is to put client and shared code into a client-side library and server-only code into a server library; client applications then link with only the client library whereas server applications link with both client and server libraries.

The optimal distribution of code components into libraries really depends on the application client/server architecture.

Table 6 Compiler Flags

Compiler Flag	Code Generated
`-client`	Client and Shared
`-server`	Server and Shared
`-both`	Client, Server and Shared
`-skel`	Server

3.3 Pragma support

The IDL compiler supports a number of pragma directives that can be included in IDL files to affect the generated code. These are described in Table 7, *Pragma directives*, below:

Table 7 Pragma directives

pragma	Description
`#pragma copy <scoped_name>`	This pragma only applies to struct types and indicates that type can be directly copied as its layout matches the CDR encoding of its elements.
`#pragma prefix "<name>"`	This adds the string prefix to the start of the name component of generated repository identifiers
`#pragma any <scoped_name>`	This generates *any* support for the scoped type and all subtypes within that scope

The following example code illustrates the use of these pragma directives:

```
#pragma prefix "prismtech.com"

module Test
{
   interface MyFace {};
   struct MyStruct
   {
      unsigned short us;
      octet o1;
      octet o2;
   };
};

#pragma any Test::MyFace
#pragma copy Test::MyStruct
```

C Language Mapping

CHAPTER

4 IDL to C Mapping

This section describes the IDL to C language mapping for the ORB.

To avoid confusion, each code snippet used in the section is preceded by a comment to identify whether it is C or IDL code, as follows:

```
// IDL
/* C */
```

4.1 Namespaces

C has no support for namespaces, therefore C types will explicitly include the IDL namespace information as part of the naming convention. The namespace name is prepended to the type name separated by an underscore.

For example, IDL *`var1`* in the *`name1`* namespace becomes *`name1_var1`* in C.

4.1.1 Scoped Names

A C program must always use the global name for a type, constant, exception, or operation. The C global name corresponding to an OMG IDL global name is derived by converting the `::` symbol to _ (an underscore) and deleting any leading underscore. For example, the IDL name *`example::color`* becomes *`example_color`* in C.

Note that the use of underscores to replace the `::` separators can lead to ambiguity if the IDL specification contains identifiers with underscores in them. Due to such ambiguities, it is advisable to avoid using underscores in IDL identifiers.

Example

The IDL enumerator *`color`* is scoped within the *`example`* interface as follows:

```
// IDL
interface example
{
   enum color {red, green, blue};
};
```

To access this enumerator in C, the following code would be required:

```
/* C */
#include "example.h"

example_color C = example_red;
```

4.1.2 Modules

Because there is no namespace support in the C language, there is no explicit mapping for modules. The module name will be prepended to the encompassed interfaces and types in order to make those names globally unique in the C code.

4.2 Interfaces

All interfaces must be defined with a global scope. This means that interfaces cannot be nested.

The IDL interface maps to C code as follows:

```
// IDL
interface example1
{
   long op1 (in long arg1);
};
```

```
/* C */
typedef CORBA_Object example1;
extern CORBA_long example1_op1
(
   example1 o,
   CORBA_long arg1,
   CORBA_Environment *ev
);
```

To permit the C programmer to use typed references, a type with the name of the interface is defined to be a `CORBA_Object`.

All typed interface references to an object are of the well-known, opaque type `CORBA_Object`. The C representation of `CORBA_Object` is a pointer and object references in C are mapped to `void*` pointers.

The literal `CORBA_OBJECT_NIL` is legal wherever a `CORBA_Object` may be used. It is guaranteed to pass the `is_nil` operation defined in *The Common Object Request Broker: Architecture and Specification*s, *ORB Interface* chapter, *Nil Object References* section.

IDL permits arguments, return results, and components of constructed types to be interface references, as in the following example:

```
// IDL
#include "example1.idl"

interface example2
{
   example1 op2 ();
};
```

This maps to the following C declaration:

```
/* C */
#include "example1.h"

typedef CORBA_Object example2;
extern example1 example2_op2 (example2 o, CORBA_Environment *ev);
```

C code for invoking this operation is as follows:

```
/* C */
#include "example2.h"

example1 ex1;
example2 ex2;
CORBA_Environment ev;

/* code for binding ex2 */
ex1 = example2_op2 (ex2, &ev);
```

4.2.1 Local Interfaces

The OMG IDL to C language mapping specification does not cover the mappings for local interfaces as support for local interfaces was added after the specification was completed. Spectra ORB supports local interfaces through it's own proprietary mapping based closely on the standard mapping for normal interfaces.

As far as a client is concerned the generated local interface stubs are exactly equivalent to a normal stub. The only essential difference being that a local object reference is simply the address of a local object implementation. See the *local* example for details.

4.2.2 Header Files

Multiple interfaces can be defined in a single source file but by convention each interface is stored in a separate source file.

The IDL compilers will, by default, generate a header file named `Foo.h` from `Foo.idl`. This file should be `#included` by clients and implementations of the interfaces defined in `Foo.idl`.

Inclusion of `Foo.h` is sufficient to define all global names associated with the interfaces in `Foo.idl` and any interfaces from which they are derived.

4.3 Data Type Mappings

4.3.1 Constants

IDL constants (specified with the `const` keyword) are `#defined` in the C code. Constant identifiers can be referenced at any point in the code where a literal of that type is legal.

The fact that constants are `#defined` may lead to ambiguities in code. To avoid conflicts, all names which are `#defined` by the mappings for any of the structured types (see *Constructed Types* on page 31) begin with an underscore.

In the mappings for wide character and wide string constants, the literals are preceded by the keyword `L` in C, as in the following example

```
// IDL
const wstring ws = “Hello World”;
```

```
/* C */
#define ws L”Hello World”
```

4.3.2 Basic Data Types

The basic data types have the mappings shown in Table 8, *Basic Data Type Mappings*, on page 30. Typedefs for the C data types listed in *Table 8* are provided.

Table 8 Basic Data Type Mappings

IDL Data Type	C Data Type
`short`	`CORBA_short`
`long`	`CORBA_long`
`unsigned short`	`CORBA_unsigned_short`
`unsigned long`	`CORBA_unsigned_long`
`float`	`CORBA_float`
`double`	`CORBA_double`
`char`	`CORBA_char`
`boolean`	`CORBA_boolean`

`any`	`typedef struct`
	`{`
	`  CORBA_TypeCode _type;`
	`  void * _value;`
	`}`
	`CORBA_any;`

The C mapping of the IDL `boolean` type is `unsigned char` with only the values 1 (`TRUE`) and 0 (`FALSE`) defined (other values produce undefined behaviour). `CORBA_boolean` is provided for consistency with the other basic data type mappings.

4.3.2.1 CORBA_any

The `CORBA_any` structure in C has the following form:

```
/* C */
typedef struct CORBA_any
{
   CORBA_TypeCode _type;
   void *_value;
} CORBA_any;
```

The `_value` member of the `CORBA_any` is a pointer to the actual value of the datum. Note that this holds true when the datum is itself implemented as a pointer (e.g., in the case of a CORBA string, the `_value` member would be a pointer `(CORBA_char**)` to `string(CORBA_char*)`.

4.3.2.1.1 Deallocating Memory

If an `any` is returned from an operation with its release flag set to `FALSE`, calling `CORBA_free()` on the returned `any*` will not deallocate the memory pointed to by `_value`. The default value of the release flag is `FALSE`.

The following two ORB-supplied functions in C allow for the setting and checking of the any release flag:

```
/* C */
void CORBA_any_set_release (CORBA_any*, CORBA_boolean);
CORBA_boolean CORBA_any_get_release (CORBA_any*);
```

Before calling CORBA_free() on the _value member of an any directly, you should check the release flag using CORBA_any_get_release. If it returns FALSE, you should not invoke CORBA_free() on the _value member; doing so produces undefined behaviour.

CORBA_any_set_release can be used to set the state of the release flag. If the flag is set to TRUE, CORBA_free() can be safely used on the returned any* to deallocate the memory pointed to by _value.

4.3.2.2 Enums

The C mapping of IDL enum types is an unsigned integer type capable of representing 2^{32} enumerations. Each enumerator in an enum is #defined with an appropriate unsigned integer value conforming to the ordering constraints.

4.3.3 Constructed Types

The mapping for IDL constructed types (structs, unions, arrays, and sequences) can vary slightly depending on whether the data structure is *fixed-length* or *variable-length*.

A type is variable-length if it is one of the following types:

- The type any
- A bounded or unbounded string or wide string
- A bounded or unbounded sequence
- An object reference or reference to a transmissible pseudo-object
- A struct or union that contains a member whose type is variable-length
- An array with a variable-length element type
- A typedef to a variable-length type

Any other data structure is, by definition, fixed-length.

The reason for treating fixed- and variable-length data structures differently is to allow more flexibility in the allocation of out parameters and return values from an operation. This flexibility allows a client-side stub for an operation that returns a sequence of strings, for example, to allocate all the string storage in one area that is deallocated in a single call. The mapping of a variable-length type as an out parameter or operation return value is a pointer to the associated class or array, as shown in Table 9, *Basic Argument and Result Passing*, on page 46.

4.3.3.0.1 Allocation Functions

For types whose parameter passing modes require heap allocation, the ORB provides allocation functions. These types include variable-length `struct`, `union`, `sequence`, `any`, `string`, `wstring`, and `arrays` of a variable-length type. The return value of these allocation functions must be freed using `CORBA_free()`.

For each of these listed types, a type-specific allocation function is provided. The allocation functions are defined with a global scope using the fully-scoped name of the type converted into a C language name followed by the suffix `__alloc` (note the double underscore). For example, for type `T`:

```
/* C */
T *T__alloc ();
```

For `any`, `string`, and `wstring` types, the allocation functions, respectively, are:

```
/* C */
CORBA_any *CORBA_any__alloc ();
char *CORBA_string__alloc ();
CORBA_wchar* CORBA_wstring__alloc (CORBA_unsigned_long len);
```

4.3.3.1 Structures

OMG IDL structures map directly onto C `structs`. The IDL compiler does not pack the C structures that result from the mapping.

4.3.3.2 Unions

IDL discriminated unions are mapped onto C `structs`, as in the following example:

```
// IDL
union Foo switch (long)
{
   case 1: long x;
   case 2: float y;
   default: char z;
};
```

```
/* C */
typedef struct
{
   CORBA_long _d;
   union
   {
      CORBA_long x;
      CORBA_float y;
      CORBA_char z;
   } _u;
} Foo;
```

The discriminator in the C struct is always referred to as `_d`. The union elements are accessed via the `_u` member.

The elements of the union are referenced as in normal C, as shown in the following example:

```
/* C */
Foo *v;

/* make a call that returns a pointer to a Foo in v */
```

```
switch (v->_d)
{
   case 1: printf ("x = %ld\n", v->_u.x); break;
   case 2: printf ("y = %f\n", v->_u.y); break;
   default: printf ("z = %c\n", v->_u.z); break;
}
```

4.3.3.3 Sequences

The IDL data type `sequence` permits passing of bounded and unbounded arrays between objects. The `sequence` type maps to a C `struct` as follows:

```
// IDL
sequence<type,size>
```

```
/* C */
typedef struct
{
   CORBA_unsigned_long _maximum;
   CORBA_unsigned_long _length;
   type *_buffer;
   CORBA_boolean _release;
} CORBA_sequence_type;
```

The `ifndef` condition is needed to prevent duplicate definition where the same type is used more than once. The type of the `_buffer` member of the C struct is derived from the type of the IDL sequence prepended by the string `CORBA_`. For example, in:

```
// IDL
typedef sequence<long,10> vec;
```

the type of the `_buffer` member is `CORBA_long`, as follows:

```
/* C */
typedef struct
{
   CORBA_unsigned_long _maximum;
   CORBA_unsigned_long _length;
   CORBA_long *_buffer;
} vec;
```

If the IDL type consists of more than one identifier, for example `unsigned long`, the generated type name in C consists of the string `CORBA_sequence_` followed by each identifier concatenated with underscores. For example, `unsigned long` maps to `CORBA_sequence_unsigned_long`.

If the IDL type is a sequence, the string `CORBA_sequence_` is used to generate the type name. For example `sequence<sequence<long> >` maps to `CORBA_sequence_sequence_long`.

It is necessary to know the this type name when declaring an instance of the struct in C. For example, an instance of the C struct defined in the above example is declared as follows:

```
/* C */
vec x = {10L, 0L, (CORBA_long *)NULL};
```

Prior to passing *&x* as an `in` parameter, you must set the *_buffer* member to point to a `CORBA_long` array of 10 elements and set the *_length* member to the actual number of elements to be passed.

Nothing needs to be done prior to passing the address of a *vec10** as an `out` parameter or receiving a *vec10** as the function return. The client stub will automatically allocate storage for the returned sequence. For bounded sequences, the client stub allocates a buffer of the specified size; for unbounded sequences, it allocates a buffer big enough to hold what was returned by the object. Upon successful return from the invocation, the *_maximum* member will contain the size of the allocated array, the *_buffer* member will point at allocated storage, and the *_length* member will contain the number of values that were returned in the *_buffer* member. The client is responsible for freeing the allocated sequence using `CORBA_free()`.

Prior to passing *&x* as an `inout` parameter, you must set the *_buffer* member to point to a *CORBA_long* array of 10 elements. The *_length* member must be set to the actual number of elements to be passed. Upon successful return from the invocation, the *_length* member will contain the number of values that were copied into the buffer pointed to by the *_buffer* member. If more data must be returned than the original buffer can hold, the called function can deallocate the original *_buffer* member using `CORBA_free()` (honouring the release flag) and assign *_buffer* to point to new storage.

For bounded sequences, the *_length* or *_maximum* members must not be set to a value larger than the specified bound.

4.3.3.3.1 Allocation Functions

The ORB provides a buffer allocation function for each sequence type. These functions allocate vectors of type `T` for `sequence<T>`. They are defined at the global scope and named similarly to sequences, as follows:

```
/* C */
T *CORBA_sequence_T_allocbuf (CORBA_unsigned_long len);
```

where `T` refers to the type name.

Buffers allocated using these allocation functions are freed using `CORBA_free()`, with the restrictions described below.

Example

```
// IDL
sequence<sequence<long> >
```

```
/* C */
T *CORBA_sequence_sequence_long_allocbuf (CORBA_unsigned_long len);
```

4.3.3.3.2 Deallocating Memory

If a sequence is returned from an operation with its release flag set to `FALSE`, calling `CORBA_free()` on the returned sequence will not deallocate the memory pointed to by `_buffer`. The default value of the release flag is `FALSE`.

The following two ORB-supplied functions in C allow for the setting and checking of the sequence release flag:

```
/* C */
void CORBA_sequence_set_release (void*, CORBA_boolean);
CORBA_boolean CORBA_sequence_get_release (void*);
```

Before calling `CORBA_free()` on the `_buffer` member of a sequence directly, you should check the release flag using `CORBA_sequence_get_release`. If it returns `FALSE`, you should not invoke `CORBA_free()` on the `_buffer` member; doing so produces undefined behaviour.

`CORBA_sequence_set_release` can be used to set the state of the release flag. If the flag is set to `TRUE`, `CORBA_free()` can be safely used on the returned sequence to deallocate the memory pointed to by `_buffer`.

4.3.3.4 Arrays

IDL arrays map directly to C arrays. For each named array type in IDL, the mapping also provides a C `typedef` pointer to the array's slice. A slice of an array is another array with all the dimensions of the original except the first.

For example, given the following OMG IDL definition:

```
// IDL
typedef long LongArray[4][5];
```

The C mapping provides the following definitions:

```
/* C */
typedef CORBA_long LongArray[4][5];
typedef CORBA_long LongArray_slice[5];
```

The generated name of the slice `typedef` is created by appending "`_slice`" to the original array name.

If the return result (or an `out` parameter for an array holding a variable-length type) of an operation is an array, the array storage is dynamically allocated by the stub. A pointer to the array slice of the dynamically allocated array is returned as the value of the client stub function. This storage is not automatically unallocated, so when the data is no longer needed you must return the dynamically allocated storage by calling `CORBA_free()`.

An array `T` of a variable-length type is dynamically allocated using the following ORB-supplied function:

```
/* C */
T_slice *T__alloc ();
```

This function is identical to the allocation functions for constructed types except that the return type is pointer to array slice, not pointer to array. See *Allocation Functions* on page 32 for details.

4.3.3.5 Strings

IDL strings are mapped to 0-byte terminated character arrays. The length of the string is encoded in the character array itself through the placement of the 0 byte. The storage for C strings is one byte longer than the stated IDL bound. The mapping is as follows:

```
// IDL
typedef string<10> sten;
typedef string sinf;

/* C */
typedef CORBA_char *sten;
typedef CORBA_char *sinf;
```

Instances of these types are declared as follows:

```
/* C */
sten s1 = NULL;
sinf s2 = NULL;
```

Prior to passing `s1` or `s2` as an `in` parameter, you must assign the address of a character buffer containing a 0-byte terminated string to the variable. The caller cannot pass a null pointer as the string argument.

Nothing needs to be done prior to passing `&s1` or `&s2` as an `out` parameter or receiving a `sten` or `sinf` as the return result. The client stub will automatically allocate storage for the returned buffer. For bounded strings, the client stub allocates a buffer of the specified size; for unbounded strings, it allocates a buffer big enough to hold the returned string. Upon successful return from the invocation, the character pointer will contain the address of the allocated buffer. The client is responsible for freeing the allocated storage using `CORBA_free()`.

Prior to passing `&s1` or `&s2` as an `inout` parameter, you must assign the address of a character buffer containing a 0-byte terminated array to the variable. If the returned string is larger than the original buffer, the client stub will call `CORBA_free()` on the original string and allocate a new buffer for the new string. The client should therefore never pass an `inout` string parameter that was not allocated using `CORBA_string_alloc()`. The client is responsible for freeing the allocated storage using `CORBA_free()`, regardless of whether or not a reallocation was necessary.

4.3.3.5.1 Allocation

Strings are dynamically allocated using the following ORB-supplied function:

```
/* C */
CORBA_char *CORBA_string_alloc (CORBA_unsigned_long len);
```

This function allocates `len+1` bytes, enough to hold the string and its terminating NULL character.

Strings allocated in this manner are freed using `CORBA_free()`.

4.3.3.6 Wide Strings

The mapping for wide strings is similar to that of strings, except that wide strings are mapped to wide-null-terminated, wide-character arrays instead of 0-byte terminated character arrays. Wide strings are dynamically allocated using the following ORB-supplied function instead of `CORBA_string_alloc`:

```
/* C */
CORBA_wchar* CORBA_wstring_alloc (CORBA_unsigned_long len);
```

The length argument `len` is the number of `CORBA_WChar` units to be allocated, including one additional unit for the null terminator.

4.3.4 Fixed Types

If a platform has a native fixed-point decimal type, matching the CORBA specifications of the `fixed` type, then the OMG IDL `fixed` type is mapped to the native type. Otherwise, the mapping is as in the following example:

```
// IDL
fixed<15,5> dec1;
typedef fixed<9,2> money;
```

```
/* C */
typedef struct
{
   CORBA_unsigned_short _digits;
   CORBA_short _scale;
   CORBA_char _value[(15+2)/2];
} CORBA_fixed_15_5;

CORBA_fixed_15_5 dec1 = {15u, 5};

typedef struct
{
   CORBA_unsigned_short _digits;
   CORBA_short _scale;
   CORBA_char _value[(9+2)/2];
} CORBA_fixed_9_2;

typedef CORBA_fixed_9_2 money;
```

An instance of `money` from this example is declared as follows:

```
/* C */
money bags = {9u, 2};
```

The following functions and operations on the `fixed` type are provided by the mapping:

```
/* C */
/* Conversions: all signs are the same. */
CORBA_long CORBA_fixed_integer_part (const void *fp);
CORBA_long CORBA_fixed_fraction_part (const void *fp);
void CORBA_fixed_set (void *rp, const CORBA_long i, const CORBA_long f);

/* Operations, of the form: r = f1 op f2 */
void CORBA_fixed_add (void *rp, const void *f1p,const void *f2p);
void CORBA_fixed_sub (void *rp, const void *f1p,const void *f2p);
void CORBA_fixed_mul (void *rp, const void *f1p,const void *f2p);
void CORBA_fixed_div (void *rp, const void *f1p,const void *f2p);
```

Since C does not support parameterised types, the fixed arguments are represented as `void*` pointers and the type information is conveyed within the representation itself. Thus the `_digits` and `_scale` of every fixed operand must be set prior to invoking these functions. Only the `_value` field of the result, denoted by `*rp`, may be left unset.

Instances of the `fixed` type are dynamically allocated using the following ORB-supplied function:

```
/* C */
CORBA_fixed_d_s* CORBA_fixed_alloc (CORBA_unsigned_short d);
```

Since IDL exceptions are allowed to have no members, but C `structs` must have at least one member, IDL exceptions with no members map to C `structs` with one member. This member is opaque to applications. Both the type and the name of the single member are implementation-specific.

4.3.4.1 Fixed-point Arithmetic

When using fixed-point arithmetic functions (*`CORBA_fixed_add()`*, *`CORBA_fixed_sub()`*, *`CORBA_fixed_mul()`* and *`CORBA_fixed_div()`*), users must set the *`_digits`* and *`_scale`* of the result prior to invoking these functions. The maximum number of digits necessary to hold the result must be calculated (refer to the *CORBA Specification* which lists the formulas that can be used to calculate the maximum number of digits and scale needed to hold the result). The `_digits` value is then used to allocate memory for the result by invoking the *`CORBA_fixed_alloc()`* function. `CORBA_fixed_alloc()` sets the `_digit` field of the result. The user must then set the *`_scale`* field of the result. The `_digits` and `_scale` of every fixed operand must be set prior to invoking the arithmetic functions. Only the `_value` field of the result may be left unset; otherwise the behavior of the functions is undefined.

For example:

```
CORBA_fixed_5_2 f1 = { 5u, 2 };
CORBA_fixed_7_3 f2 = { 7u, 3 };
CORBA_fixed *pFixed = 0;
CORBA_unsigned_short digits = 0;
CORBA_short scale = 0;

CORBA_fixed_set (&f1, 999, 99);
CORBA_fixed_set (&f2, 1234, 567);

scale = max (f1->_scale, f2->_scale);
digits = max (f1->_digits - f1->_scale,
f2->_digits - f2->_scale) +
scale + 1;

pFixed = CORBA_fixed_alloc (digits);
pFixed->_scale = scale;
```

After the arithmetic function has returned, the `_scale` and the `_digits` fields of the result may have changed and will now hold the actual number of digits and scale used by the result, which may not be the maximum value that was previously calculated by the user.

The user can obtain the integer and fraction parts of the result using the following functions:

```
CORBA_long CORBA_fixed_integer_part (const void *fp);
CORBA_long CORBA_fixed_fraction_part (const void *fp);
```

The above functions return CORBA_long data types and will not return more than nine significant digits. If the result held in the integer part is a large number that exceeds nine significant digits, the leftmost 9 digits held in the _value field are returned. The user must check the number of integer digits to see if this has occured:

```
CORBA_short integerDigits = pFixed->_digits -
   pFixed->_scale;
```

If required, the user can then manually extract the remaining integer digits from the `_value` field of the result (refer to the *CORBA Specification* for details of how the `_value` field stores the digits). As `CORBA_ action_part()` returns, for example consider the following number, which has a `_scale` of 6:

```
1000.000660
```

CORBA_fixed_fraction_part() will return 660.

Division by zero or dividing into zero will result in a value of zero, and will not throw an error.

4.4 Exceptions

4.4.1 Mapping Exception Types

Each IDL exception type is mapped to a `struct` tag and a `typedef` with the C global name for the exception. An identifier for the exception in string literal form is `#defined` and a type-specific allocation function is created.

The following example illustrates this mapping:

```
// IDL
exception foo
{
   long dummy;
};
```

```
/* C */
typedef struct foo
{
   CORBA_long dummy;
   /* ...may contain additional
    * implementation-specific members...
    */
} foo;
#define ex_foo <unique identifier for exception>
foo *foo__alloc ();
```

The generated allocation function dynamically allocates an instance of the exception and returns a pointer to it. Each exception type has its own dynamic allocation function. Exceptions allocated using a dynamic allocation function are freed using `CORBA_free()`.

4.4.2 Handling Exceptions

Since the C language does not provide native exception handling support, applications pass and receive exceptions via the `CORBA_Environment` parameter passed to each IDL operation. The `CORBA_Environment` type is partially opaque. The C declaration contains at least the following:

```
/* C */
typedef struct CORBA_Environment
{
   CORBA_exception_type _major;
} CORBA_Environment;
```

Upon return from an invocation, the `_major` field indicates whether the invocation terminated successfully. `_major` can have a value of `CORBA_NO_EXCEPTION`, `CORBA_USER_EXCEPTION`, or `CORBA_SYSTEM_EXCEPTION`. If the value is `CORBA_USER_EXCEPTION` or `CORBA_SYSTEM_EXCEPTION`, any exception parameters signalled by the object can be accessed.

4.4.2.1 Functions

Five functions are defined on a `CORBA_Environment` structure for accessing exception information. Their signatures are:

```
/* C */
extern void CORBA_exception_set
(
   CORBA_Environment *ev,
   CORBA_exception_type major,
   CORBA_char *except_repos_id,
   void *param
);
extern CORBA_char *CORBA_exception_id (CORBA_Environment *ev);
extern void *CORBA_exception_value (CORBA_Environment *ev);
extern void CORBA_exception_free (CORBA_Environment *ev);
extern CORBA_any* CORBA_exception_as_any (CORBA_Environment *ev);
```

4.4.2.1.1 CORBA_exception_set()

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

- If the `major` parameter has the value `CORBA_NO_EXCEPTION`, this is a normal outcome to the operation. In this case, both `except_repos_id` and `param` must be `NULL`. Note that it is not necessary to invoke `CORBA_exception_set()` to indicate a normal outcome; it is the default behaviour if the method simply returns.

- Any other value of `major` specifies either a user-defined or system exception. The `except_repos_id` parameter is the repository ID representing the exception type. If the exception is declared to have members, the `param` parameter must be the address of an instance of the exception struct containing the parameters. In this case, the exception struct must be allocated using the appropriate `T__alloc()` function. The `CORBA_exception_set()` function adopts the allocated memory and frees it when it no longer needs it. Once the allocated exception struct is passed to `CORBA_exception_set()`, the application is not allowed to access it because it no longer owns it. If the exception takes no parameters, `param` must be `NULL`.

If the `CORBA_Environment` argument to `CORBA_exception_set()` already has an exception set in it, that exception is properly freed before the new exception information is set.

4.4.2.1.2 CORBS_exception_id()

`CORBA_exception_id()` returns a pointer to the character string identifying the exception. The character string contains the repository ID for the exception. If invoked on a `CORBA_Environment` which identifies a non-exception (where `_major==CORBA_NO_EXCEPTION`), a null pointer is returned. Ownership of the returned pointer does not transfer to the caller. Instead, the pointer remains valid until `CORBA_exception_free()` is called.

4.4.2.1.3 CORBA_exception_value()

`CORBA_exception_value()` returns a pointer to the structure corresponding to this exception. If invoked on a `CORBA_Environment` which identifies a non-exception (where `_major==CORBA_NO_EXCEPTION`), or an exception for which there is no associated information, a null pointer is returned. Ownership of the returned pointer does not transfer to the caller. The pointer remains valid until `CORBA_exception_free()` is called.

4.4.2.1.4 CORBA_exception_free()

`CORBA_exception_free()` frees any storage which was allocated in the construction of the `CORBA_Environment` or adopted by the `CORBA_Environment` when `CORBA_exception_set()` is called on it, and sets the `_major` field to `CORBA_NO_EXCEPTION`. The `CORBA_exception_free()` function can be invoked regardless of the value of the `_major` field.

4.4.2.1.5 CORBA_exception_as_any()

`CORBA_exception_as_any()` returns a pointer to a `CORBA_any` containing the exception. This allows a C application to deal with exceptions for which it has no static (compile-time) information. If invoked on a `CORBA_Environment` which identifies a non-exception (where `_major==CORBA_NO_EXCEPTION`), a null pointer is returned. Ownership of the returned pointer does not transfer to the caller; instead. The pointer remains valid until `CORBA_exception_free()` is called. If it is necessary to take ownership of an exception, then the `_value` field of the returned `any` should be set to NULL.

4.4.2.2 Example of Exception Handling

The following interface defines a single operation which returns no results and can raise a `BadCall` exception:

```
// IDL
interface exampleX
{
   exception BadCall
   {
      string<80> reason;
   };
   void op () raises (BadCall);
};
```

The following C code shows how to invoke the operation and recover from an exception:

```
/* C */
#include "exampleX.h"
CORBA_Environment ev;
exampleX obj;
exampleX_BadCall *bc;
/*
 * some code to initialize obj to a reference to an object
 * supporting the exampleX interface
 */
exampleX_op (obj, &ev);
switch (ev._major)
{
   case CORBA_NO_EXCEPTION:  /* successful outcome*/
      /* process out and inout arguments */
      break;
   case CORBA_USER_EXCEPTION:  /* a user-defined exception */
      if (strcmp (ex_exampleX_BadCall,CORBA_exception_id (&ev)) == 0)
      {
         bc = (exampleX_BadCall*)CORBA_exception_value (&ev);
         fprintf (stderr, "exampleX_op () failed - reason:
%s\n",bc->reason);
      }
      else  /* should never get here ... */
      {
         fprintf ( stderr,"unknown user-defined exception -%s\n",
            CORBA_exception_id (&ev));
      }
      break;
   default:  /* standard exception */
      /*
       * CORBA_exception_id() can be used to determine
       * which particular standard exception was
       * raised; the minor member of the struct
       * associated with the exception (as yielded by
       * CORBA_exception_value()) may provide additional
       * system-specific information about the exception
       */
       break;
}
/* free any storage associated with exception */
CORBA_exception_free (&ev);
```

4.5 Operations

An operation in IDL maps to a C function with the same name as the operation.

The C function name is prepended with the interface name and may also include a module name prefix. See Section 4.1, *Namespaces*, on page 27 for details of naming conventions.

4.5.1 Oneway Operations

A `oneway` operation in IDL specifies a best-effort operation. A client invocation of a `oneway` operation is attempted once at most, and the delivery of the operation to the object implementation is not guaranteed. A `oneway` operation must have a `void` return value and `in` parameters only (no return value and no `inout` or `out` parameters). The operation also cannot have a `raises` expression, since the best-effort semantics do not allow the object implementation to notify the client of exceptional conditions. The C signature for a `oneway` operation does not differ from the signature for a normal operation with the same parameters.

4.5.2 Arguments to Operations

4.5.2.1 Implicit Arguments

In addition to the operation-specific parameters, all operations declared in an interface have additional standard parameters as follows:

- The first parameter to each operation is a `CORBA_Object` input parameter. This parameter designates the object to process the request.
- The last parameter to each operation is a `CORBA_Environment*` output parameter. This parameter permits the return of exception information.

The `CORBA_Object` type is an opaque type. The `CORBA_Environment` type is partially opaque.

Example

```
// IDL
interface example4
{
   long op5 (in long arg6);
};
```

The C function for the `op5` operation will have the following function signature:

```
/* C */
CORBA_long example4_op5
(
   PortableServer_Servant _servant,
   CORBA_long arg6,
   CORBA_Environment* _env
);
```

The `_servant` parameter is the pointer to the servant incarnating the CORBA object on which the request was invoked. The `_env` parameter is used for raising exceptions. Note that the names of the `_servant` and `_env` parameters are standardised to allow the bodies of method functions to refer to them portably.

4.5.2.2 Functions with Empty Argument Lists

A function declared in IDL with an empty argument list is defined to take *no* operation-specific arguments but the C mapping of the function will contain the implicit arguments as outlined above.

4.5.2.3 Argument Passing Considerations

For all IDL types, if the IDL signature specifies that an argument is an `out` or `inout` parameter, then the caller must always pass the address of a variable of that type (or the value of a pointer to that type). The called function must dereference the parameter to get to the type. For arrays, the caller must pass the address of the first element of the array.

For `in` parameters, the value of the parameter must be passed for all of the basic types, enumeration types, and object references. For all arrays, the address of the first element of the array must be passed. For all other structured types, the address of a variable of that type must be passed, regardless of whether they are fixed- or variable-length. For strings, a `char*` or `wchar*` must be passed.

For `inout` parameters, the address of a variable of the correct type must be passed for all of the basic types, enumeration types, object references, and structured types. For strings, the address of a `char*` or a `wchar *` must be passed. For all arrays, the address of the first element of the array must be passed.

Consider the following IDL specification:

```
// IDL
interface foo
{
   typedef long Vector[25];
   void bar (out Vector x, out long y);
};
```

A client which invoked the `bar` operation would require the following C code:

```
/* C */
foo object;
foo_Vector_slice x;
CORBA_long y;
CORBA_Environment ev;

/* code to bind object to instance of foo */

foo_bar (object, &x, &y, &ev);
```

For `out` parameters of type variable-length `struct`, variable-length `union`, `string`, `sequence`, an array holding a variable-length type, or `any`, the ORB will allocate storage for the output value using the appropriate type-specific allocation function. The client may use and retain that storage indefinitely, and must indicate when the value is no longer needed by calling the procedure `CORBA_free()`, whose signature is:

```
/* C */
extern void CORBA_free (void *storage);
```

The parameter to `CORBA_free()` is the pointer used to return the `out` parameter. `CORBA_free()` releases the ORB-allocated storage occupied by the `out` parameter, including storage indirectly referenced, such as in the case of a sequence of strings or

array of object reference. If a client does not call `CORBA_free()` before reusing the pointers that reference the `out` parameters, that storage might be wasted. Passing a null pointer to `CORBA_free()` is allowed; `CORBA_free()` simply ignores it and returns without error.

4.5.3 Return Results

An operation terminates successfully by executing a `return` statement returning the declared operation value. Prior to returning the result of a successful invocation, the method code must assign legal values to all `out` and `inout` parameters.

The method terminates with an error by executing the `CORBA_exception_set` operation (see the *Spectra ORB C Edition User Guide*) prior to executing a `return` statement. When raising an exception, the method code is *not* required to assign legal values to any `out` or `inout` parameters. Due to restrictions in C, however, it must return a legal function value.

4.5.3.1 Return Result Passing Considerations

When an operation returns a non-void result, the following rules hold:

1. If the return result type is `float`, `double`, `long`, `short`, `unsigned long`, `unsigned short`, `char`, `wchar`, `fixed`, `boolean`, `octet`, `Object`, or an `enumeration`, then the value is returned as the operation result.
2. If the return result type is `struct` or `union`, then the value of the C struct representing that type is returned as the operation result.
3. If the return result is one of the variable-length types `struct`, `union`, `sequence`, or `any`, then a pointer to a C struct representing that type is returned as the operation result.
4. If the return result is of type `string` or `wstring`, then a pointer to the first character of the string is returned as the operation result.
5. If the return result is of type `array`, then a pointer to the slice of the array is returned as the operation result.

The following example illustrates the mapping of an IDL interface to C functions.

```
// IDL
interface X
{
   struct y
   {
      long a;
      float b;
   };
   long op1 ();
   y op2 ();
};
```

```
/* C */
typedef CORBA_Object X;
typedef struct X_y
{
   CORBA_long a;
   CORBA_float b;
} X_y;
```

```
extern CORBA_long X_op1 (X object, CORBA_Environment *ev);
extern X_y X_op2 (X object, CORBA_Environment *ev);
```

For operation results of type variable-length `struct`, variable-length `union`, `wstring`, `string`, `sequence`, `array`, or `any`, the ORB will allocate storage for the return value using the appropriate type-specific allocation function. The client may use and retain that storage indefinitely and must indicate when the value is no longer needed by calling `CORBA_free()`.

4.5.4 Argument and Result Passing Summary

Table 9 summarizes what a client passes as an argument to a stub and receives as a result. For brevity, the `CORBA_` prefix is omitted from type names in the tables.

Table 9 Basic Argument and Result Passing

Data Type	In	Inout	Out	Return
`short`	`short`	`short*`	`short*`	`short`
`long`	`long`	`long*`	`long*`	`long`
`long long`	`long_long`	`long_long*`	`long_long*`	`long_long`
`unsigned short`	`unsigned_short`	`unsigned_short*`	`unsigned_short*`	`unsigned_short`
`unsigned long`	`unsigned_long`	`unsigned_long*`	`unsigned_long*`	`unsigned_long`
`unsigned long long`	`unsigned_long_long`	`unsigned_long_long*`	`unsigned_long_long*`	`unsigned_long_long`
`float`	`float`	`float*`	`float*`	`float`
`double`	`double`	`double*`	`double*`	`double`
`long double`	`long_double`	`long_double*`	`long_double*`	`long_double`
`fixed<d,s>`	`fixed_d_s*`	`fixed_d_s*`	`fixed_d_s*`	`fixed_d_s`
`boolean`	`boolean`	`boolean*`	`boolean*`	`boolean`
`char`	`char`	`char*`	`char*`	`char`
`octet`	`octet`	`octet*`	`octet*`	`octet`
`enum`	`enum`	`enum*`	`enum*`	`enum`
`object reference ptr` [1]	`objref_ptr`	`objref_ptr*`	`objref_ptr*`	`objref_ptr`
`struct, fixed`	`struct*`	`struct*`	`struct*`	`struct`
`struct, variable`	`struct*`	`struct*`	`struct**`	`struct*`
`union, fixed`	`union*`	`union*`	`union*`	`union`

Table 9 Basic Argument and Result Passing (Continued)

Data Type	In	Inout	Out	Return
`union, variable`	`union*`	`union*`	`union**`	`union*`
`string`	`char*`	`char**`	`char**`	`char*`
`sequence`	`sequence*`	`sequence*`	`sequence**`	`sequence*`
`array, fixed`	`array`	`array`	`array`	`array slice*` [2]
`array, variable`	`array`	`array`	`array slice**` [2]	`array slice*` [2]
`any`	`any*`	`any*`	`any**`	`any*`

1. Including pseudo-object references.
2. A slice is an array with all the dimensions of the original except the first one.

A client is responsible for providing storage for all arguments passed as `in` arguments. Refer to Table 10, *Client Argument Storage Responsibilities* for storage responsibilities for other argument types. Note that the numbered cases in *Table 10* are described below the table.

Table 10 Client Argument Storage Responsibilities

	Argument (refer to key, below)		
Type	**Inout Param**	**Out Param**	**Return Result**
`short`	1	1	1
`long`	1	1	1
`unsigned short`	1	1	1
`unsigned long`	1	1	1
`float`	1	1	1
`double`	1	1	1
`boolean`	1	1	1
`char`	1	1	1
`octet`	1	1	1
`enum`	1	1	1
`object reference ptr`	2	2	2
`struct, fixed`	1	1	1
`struct, variable`	1	3	3
`union, fixed`	1	1	1
`union, variable`	1	3	3
`string`	4	3	3

Table 10 Client Argument Storage Responsibilities (Continued)

	Argument (refer to key, below)		
Type	**Inout Param**	**Out Param**	**Return Result**
`sequence`	5	3	3
`array, fixed`	1	1	6
`array, variable`	1	6	6
`any`	5	3	3

1. The caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For `inout` parameters, the caller provides the initial value and the called function may change that value. For `out` parameters, the caller allocates the storage but need not initialize it and the called function sets the value. Function returns are by value.
2. The caller allocates storage for the object reference. For `inout` parameters, the caller provides an initial value. If the called function wants to reassign the `inout` parameter, it must first call `CORBA_Object_release()` on the original input value. To continue to use an object reference passed in as an `inout`, the caller must first duplicate the reference. The client is responsible for the release of all `out` parameters and return object references. Release of all object references embedded in other `out` and return structures is performed automatically as a result of calling `CORBA_free`.
3. For `out` parameters, the caller allocates a pointer and passes it by reference to the called function. The called function sets the pointer to point to a valid instance of the parameter's type. For returns, the function returns a similar pointer. The function is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. Following the completion of a request, the caller is not allowed to modify any values in the returned storage. To do so, the caller must first copy the returned instance into a new instance, then modify the new instance.
4. For `inout` strings, the caller provides storage for both the input string and the `char*` pointing to it. The called function may deallocate the input string and reassign the `char*` to point to new storage to hold the output value. The size of the `out` string is therefore not limited by the size of the `in` string. The caller is responsible for freeing the storage for the `out`. The called function is not allowed to return a null pointer for an `inout`, `out`, or return value.
5. For `inout` sequences and `anys`, assignment or modification of the sequence or `any` may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the boolean release in the sequence or `any`.
6. For `out` parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the called function. The called function sets the pointer to point to a valid instance of the array. For returns, the function returns a similar pointer. The function is not allowed to return a null pointer in either case. In both cases, the caller is

responsible for releasing the returned storage. Following the completion of a request, the caller is not allowed to modify any values in the returned storage. To do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

4.5.5 Inheritance and Operation Names

IDL permits the specification of interfaces that inherit operations from other interfaces. As C does not natively support inheritance, the mapping ensures that an object can access an interface's inherited operation as if it was directly declared in the new interface. The following example illustrates this:

```
// IDL

interface example1
{
   long op1 (in long arg1);
};

interface example2 : example1
{
   void op2 (in long arg2, out long arg3);
};
```

```
/* C */

typedef CORBA_Object example1;
extern CORBA_long example1_op1
(
   example1 o,
   CORBA_long arg1,
   CORBA_Environment *ev
);

typedef CORBA_Object example2;
extern CORBA_long example2_op1
(
   example2 o,
   CORBA_long arg1,
   CORBA_Environment *ev
);
extern void example2_op2
(
   example2 o,
   CORBA_long arg2,
   CORBA_long *arg3,
   CORBA_Environment *ev
);
```

In the above example, `op1` can be accessed as if it was directly declared in `example2`. Of course, the programmer could also invoke `example1_op1` on an object of type `example2`. The virtual nature of operations in interface definitions will cause invocations of either function to cause the same method to be invoked.

4.6 Attributes

When attributes are mapped to C, the generated code includes operations to *get* and *set* (read and write) the attributes. Attributes declared as `readonly` in IDL have *get* functions only. These operations are not defined in IDL so there is no direct mapping from the IDL functions to the C functions.

The following example illustrates this.

```
// IDL
interface foo
{
   struct position_t
   {
      float x, y;
   };

   attribute float radius;
   readonly attribute position_t position;
};
```

```
/* C */
typedef struct foo_position_t
{
   CORBA_float x, y;
} foo_position_t;

extern CORBA_float foo__get_radius (foo o, CORBA_Environment *ev);
extern void foo__set_radius
(
   foo o,
   CORBA_float r,
   CORBA_Environment *ev
);
extern foo_position_t foo__get_position (foo o, CORBA_Environment *ev);
```

Note that two underscore characters (__) separate the name of the interface from the words `get` or `set` in the names of the functions.

If the *set* accessor function fails to set the attribute value, the method returns an exception.

4.7 Pseudo-objects

There are several interfaces that are defined as pseudo-objects in the C language mapping. A client makes calls on a pseudo object in the same way as on an ordinary CORBA object. The ORB may implement the pseudo-object directly and there are restrictions on what a client may do with the pseudo-object.

The ORB itself is a pseudo-object with the following partial definition:

```
// IDL
interface ORB
{
   string object_to_string (in Object obj);
   Object string_to_object (in string str);
};
```

A C program can convert an object reference into its string form, just as if the ORB were an ordinary object, by calling:

```
/* C */
CORBA_char *str = CORBA_ORB_object_to_string (orb, obj, env);
```

The C library contains the routine `CORBA_ORB_object_to_string`, and it does not do a real invocation. The `orb` object is an object reference that specifies which ORB to use, since it is possible to choose which ORB should be used to convert an object reference to a string.

Although operations on pseudo-objects are invoked in the usual way defined by the C language mapping, there are restrictions on them. In general, a pseudo-object cannot be specified as a parameter to an operation on an ordinary object. Pseudo-objects do not have definitions in the interface repository.

4.8 Object Implementation Mapping

This section describes the IDL to C language mapping that apply specifically to the portable object adapter, such as how the implementation methods are connected to the skeleton. In general, given the non-object-oriented nature of C, polymorphism and commonly used POA functionality has to be effected using special structures containing function pointers to base class and derived class implementation functions.

4.8.1 Operation-specific Details

Generally, for those parameters that are operation-specific, the method implementing the operation receives the same values that would be passed to the stubs.

4.8.2 Servant Mapping

A *servant* is a language-specific entity that can incarnate a CORBA object. In C, a servant is composed of a data structure that holds the state of the object along with a collection of *method functions* that manipulate that state to implement the CORBA object.

The `PortableServer::Servant` type maps into C as follows:

```
/* C */
typedef void* PortableServer_Servant;
```

`Servant` is mapped to a `void*` rather than a pointer to `ServantBase` so that all servant types for derived interfaces can be passed to all the operations that take a `Servant` parameter without requiring casting. However, it is expected that an instance of `PortableServer_Servant` points to an instance of a `PortableServer_Servant` or its equivalent for derived interfaces.

4.8.3 PortableServer Functions

Objects registered with POAs use the `PortableServer_POA_ObjectId` type, which is a sequence of octets, as object identifiers. Because C programmers will often want to use strings as object identifiers, the C mapping provides several conversion functions that convert strings to `ObjectId` types and vice-versa. These functions are:

```
/* C */

extern CORBA_char* PortableServer_ObjectId_to_string
(
   PortableServer_ObjectId* id,
   CORBA_Environment* env
);

extern PortableServer_ObjectId* PortableServer_string_to_ObjectId
(
   CORBA_char* str,
   CORBA_Environment* env
);
```

These functions follow the normal C mapping rules for parameter passing and memory management. If conversion of an `ObjectId` to a string would result in illegal characters in the string (such as a `NUL`), the first two functions raise the `CORBA_BAD_PARAM` exception.

4.8.4 Servant Structure Initialization

Each servant requires initialization and etherealization (or finalization) functions. For `PortableServer_ServantBase`, the following C functions are provided:

```
/* C */
void PortableServer_ServantBase__init (PortableServer_Servant,
   CORBA_Environment*);
void PortableServer_ServantBase__fini (PortableServer_Servant,
   CORBA_Environment*);
```

These functions are named by appending `__init` and `__fini`, respectively, (note the double underscores) to the name of the servant.

The first argument to the `init` function is a valid `PortableServer_Servant`. The `init` function will perform ORB-specific initialization of the `PortableServer_ServantBase`.

The `fini` function only cleans up ORB-specific private data. It is the default finalization function for servants. It does not make any assumptions about where the servant is allocated, such as assuming that the servant is heap-allocated and trying to call `CORBA_free()` on it.

Normally, the `PortableServer_ServantBase__init` and `PortableServer_ServantBase__fini` functions are not invoked directly by applications but rather by interface-specific initialization and finalization functions generated by the IDL compiler. The address of a servant shall be passed to the `init` function before the servant is allowed to be activated or registered with the POA in any way. The results of failing to properly initialize a servant via the appropriate `init` function may include memory access violations that could crash the application.

If the IDL compiler has generated an VEPV and EPV structures (with the `-gen_vepv` flag) then by default the generated `__init` function will use these to initialize the servant. To disable this behaviour use the `-no_vepv_init` flag.

4.8.5 Application Servants

It is expected that applications will create their own servant structures so that they can add their own servant-specific data members to store object state. For the `Counter` example shown above, an application servant would probably have a data member used to store the counter value:

```
/* C */
typedef struct AppServant
{
   POA_Counter base;
   CORBA_long value;
} AppServant;
```

The application might contain the following implementation of the `Counter_add` operation:

```
/* C */
CORBA_long
app_servant_add (PortableServer_Servant _servant, CORBA_long val,
                 CORBA_Environment* _env)
{
   AppServant* self = (AppServant*)_servant;
   self->value += val;
   return self->value;
}
```

The application could initialize the servant statically as follows:

```
/* C */
static AppServant myServant;
```

Before registering or activating this servant, the application should call:

```
/* C */
CORBA_Environment env;
POA_Counter__init (&my_servant, &env);
```

4.8.6 Method Signatures

With the POA, implementation methods have signatures that are identical to the stubs except for the first argument. For example, consider the following interface defined in IDL:

```
// IDL
interface example4
{
   long op5 (in long arg6);
};
```

In C, a method function for the `op5` operation must have the following function signature:

```
/* C */
CORBA_long example4_op5
(
   PortableServer_Servant _servant,
```

```
   CORBA_long arg6,
   CORBA_Environment* _env
);
```

The `_servant` parameter is the pointer to the servant incarnating the CORBA object on which the request was invoked. The method can obtain the object reference for the target CORBA object by using the `POA_Current` object. The `_env` parameter is used for raising exceptions. Note that the names of the `_servant` and `_env` parameters are standardized to allow the bodies of method functions to refer to them portably.

The method terminates successfully by executing a `return` statement returning the declared operation value. Prior to returning the result of a successful invocation, the method code must assign legal values to all `out` and `inout` parameters.

The method terminates with an error by executing the `CORBA_exception_set` operation (described in *Handling Exceptions* on page 40) prior to executing a `return` statement. When raising an exception, the method code is not required to assign legal values to any `out` or `inout` parameters. Due to restrictions in C, however, it must return a legal function value.

4.8.7 Object Data Structure

C does not directly support the binding of data and executable code into a single entity. Therefore, the ORB will pass a structure containing an object's data to each method of the object as a parameter. This parameter represents the object's data members and maintains an object's state.

For example, the following code snippets represent a bank account object implemented in C.

```
/* C */

/* Declare the servant's member variables in a struct */
typedef struct
{
   /* Needed by the ORB as first member of struct*/
   PortableServer_Servant servant;
   /* Member data */
   double balance;
   double interestRate;
   Date lastWithdrawal;
} sBankAccount;

double getAccountBalance_i (PortableServer_Servant _servant,
   CORBA Environment _env)
{
   /* Typecast the struct so that we can get at
   member variables safely */
   sBankAccount* pBankAccount = (sBankAccount*)servant;
   /* return the bank account balance */
   return pBankAccount->balance;
} /* getAccountBalance_i */
```

The `PortableServer_Servant`, which is passed in as a parameter to `getAccountBalance_i`, is a pointer to an `sBankAccount` structure. We need to typecast the structure to an `sBankAccount` since the ORB only knows of `PortableServer_Servant` types. It is important that the first member of the structure

be of type `PortableServer_Servant`, since the ORB does not have any information regarding user-defined types such as `sBankAccount` structure. By making this the first member of the structure, we provide a means by which structures of different types can be handled, stored, and passed around the ORB in a type-safe manner.

For those familiar with C++, the `PortableServer_Servant` parameter can be thought of as the `this` pointer secretly passed during all C++ member calls.

Index

Index

Symbols

A

B

C

D

E

G

I

K

M

N

O

P

R

S

T

U

W